Unit - 5
Design and Analysis of Algorithms

- Randomized Hiring problem
- Randomized Quicksort
- Definitions of P, NP, NP-Hard, NP-Complete
- Relationship between the complexity classes
- Showing a problem is NP-Complete
- CLIQUE, INDSET, VERTEX-COVER is NPC.
- Why Approximation Algorithms
- VERTEX-COVER has a polynomial time approximation algorithm

The hiring problem

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency will send you one candidate each day. You will interview that person and then decide to either hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a large hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode. It assumes that the candidates for the office assistant job are numbered 1 through n. The procedure assumes that you are able to, after interviewing candidate i, determine if candidate i is the best candidate you have seen so far. To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

```
HIRE-ASSISTANT(n)
1  best ← 0        ▷ candidate 0 is a least-qualified dummy candidate
2  for i ← 1 to n
3        do interview candidate i
4           if candidate i is better than candidate best
5             then best ← i
6                  hire candidate i
```

Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Let m be the number of people hired. Then the total cost associated with this algorithm is $O(nc_i + mc_h)$. No matter how many people we hire, we always interview n candidates and thus always incur the cost nc_i associated with interviewing. We therefore concentrate on analyzing mc_h, the hiring cost. This quantity varies with each run of the algorithm.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview. This situation occurs if the candidates come in increasing order of quality, in which case we hire n times, for a total hiring cost of $O(nc_h)$.

Analysis of the hiring problem using indicator random variables

Returning to the hiring problem, we now wish to compute the expected number of times that we hire a new office assistant. In order to use a probabilistic analysis, we assume that the candidates arrive in a random order, as discussed in the previous section. (We shall see in Section 5.3 how to remove this assumption.) Let X be the random variable whose value equals the number of times we hire a new office assistant. We could then apply the definition of expected value from equation (C.19) to obtain

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\} ,$$

but this calculation would be cumbersome. We shall instead use indicator random variables to greatly simplify the calculation.

To use indicator random variables, instead of computing $\mathrm{E}[X]$ by defining one variable associated with the number of times we hire a new office assistant, we define n variables related to whether or not each particular candidate is hired. In particular, we let X_i be the indicator random variable associated with the event in which the ith candidate is hired. Thus,

$$X_i = \mathrm{I}\{\text{candidate } i \text{ is hired}\} = \begin{cases} 1 & \text{if candidate } i \text{ is hired} , \\ 0 & \text{if candidate } i \text{ is not hired} , \end{cases} \tag{5.2}$$

and

$$X = X_1 + X_2 + \cdots + X_n . \tag{5.3}$$

Candidate i is hired, in line 5, exactly when candidate i is better than each of candidates 1 through $i-1$. Because we have assumed that the candidates arrive in a random order, the first i candidates have appeared in a random order. Any one of these first i candidates is equally likely to be the best-qualified so far. Candidate i has a probability of $1/i$ of being better qualified than candidates 1 through $i-1$ and thus a probability of $1/i$ of being hired. By Lemma 5.1, we conclude that

$$\mathrm{E}[X_i] = 1/i \,. \tag{5.4}$$

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.3))} \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} 1/i \\
&= \ln n
\end{aligned}$$

Even though we interview n people, we only actually hire approximately $\ln n$ of them, on average. We summarize this result in the following lemma.

Lemma 5.2

Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has a total hiring cost of $O(c_h \ln n)$.

Randomized Quicksort

Quicksort: Given array of some length n,

1. Pick an element p of the array as the pivot (or halt if the array has size 0 or 1).
2. Split the array into sub-arrays LESS, EQUAL, and GREATER by comparing each element to the pivot. (LESS has all elements less than p, EQUAL has all elements equal to p, and GREATER has all elements greater than p).

Basic-Quicksort: Run the Quicksort algorithm as given above, always choosing the leftmost element in the array as the pivot.

Best Case Running Time: O(nlogn)

Worst Case Running Time: O(n^2)

Average Case Running Time: O(nlogn)

Randomized Quicksort

Quicksort: Given array of some length n,

1. Pick an element p of the array as the pivot (or halt if the array has size 0 or 1).
2. Split the array into sub-arrays LESS, EQUAL, and GREATER by comparing each element to the pivot. (LESS has all elements less than p, EQUAL has all elements equal to p, and GREATER has all elements greater than p).

Randomized-Quicksort: Run the Quicksort algorithm as given above, each time picking a *random* element in the array as the pivot.

To prove: Expected Running Time: O(nlogn)

This is called a Worst-case Expected-Time bound.

Theorem 3.3 *For Randomized Quicksort, the expected number of comparisons is at most* $2n \ln n$.

Define random variable X_{ij} to be 1 if the algorithm *does* compare the ith smallest and jth smallest elements in the course of sorting, and 0 if it does not. Let X denote the total number of comparisons made by the algorithm. Since we never compare the same pair of elements twice, we have

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij},$$

Let ***ei*** be the ith smallest element and ***ej*** be the jth smallest element in the array. **Assume: ei < ej.**

Case 1: If ei < p < ej, then ei and ej will never be compared as they fall into different partitions.

Case 2: If p = ei or p = ej, then we do compare p (say = ei) with ej.

Case 3: If p < ei < ej or ei < ej < p, then ei and ej will fall into same bucket and another pivot will be chosen.

Probability that ei and ej will be compared = 2 / (j-i+1). Hence,

the probability that $X_{ij} = 1$ is $2/(j - i + 1)$.

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij},$$

$$\mathbf{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}].$$

$$\mathbf{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2/(j-i+1).$$

$$\mathbf{E}[X] = \sum_{i=1}^{n} 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1}\right).$$

The quantity $1 + 1/2 + 1/3 + \ldots + 1/n$, denoted H_n, is called the "nth harmonic number" and is in the range $[\ln n, 1 + \ln n]$ (this can be seen by considering the integral of $f(x) = 1/x$). Therefore,

$$\mathbf{E}[X] < 2n(H_n - 1) \leq 2n \ln n. \quad \blacksquare$$

P, NP, NP-Hard, NP-Complete -Definitions

P: the class of problems that have polynomial-time deterministic algorithms.

- That is, they are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n
- A deterministic algorithm is (essentially) one that always computes the correct answer

Sample Problems in P

- Fractional Knapsack
- MST
- Sorting
- Others?

Decision problems

- The solution is simply “Yes” or “No”.
- Optimization problem : more difficult

  Decision problem
- E.g. the traveling salesperson problem
  - Optimization version:

    Find the shortest tour
  - Decision version:

    Is there a tour whose total length is less than or equal to a constant C ?

The class *NP*

NP: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a nondeterministic algorithm)

- (A *deterministic* computer is what we know)
- A *nondeterministic* computer is one that can "guess" the right answer or solution

- Think of a nondeterministic computer as a parallel machine that can freely spawn ***an infinite number*** of processes
- Thus *NP* can also be thought of as the class of problems "whose solutions can be verified in polynomial time"
- Note that *NP* stands for "Nondeterministic Polynomial-time"

Sample Problems in NP

- Fractional Knapsack
- MST
- Others?
  - Traveling Salesman
  - Graph Coloring
  - Satisfiability (SAT)
    - the problem of deciding whether a given Boolean formula is satisfiable

NP algorithm

- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.

NP problem

- If a decision problem can be solved by a NP algorithm, this problem is called an NP (nondeterministic polynomial) problem.
- NP problems : (must be decision problems)

Subset Sum Problem: Given **N** non-negative integers $\mathbf{a_1 \ldots a_N}$ and a target sum **K**, the task is to decide if there is a subset having a sum equal to **K**.

Example: Let {5, 7, 3, 9, 1}, Target Sum: 12

Is {5,7} a solution : YES

Is {7,3,1} a solution: NO

Satisfiability is in NP

Literal: A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause: A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form: A propositional formula Φ that is the conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT: Given CNF formula Φ, does it have a satisfying truth assignment?

3-SAT: SAT where each clause contains exactly 3 literals.

Ex: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$

Yes: x_1 = true, x_2 = true x_3 = false.

P And NP Summary

- **P** = set of problems that can be solved in polynomial time
  - Examples: Fractional Knapsack, …
- **NP** = set of problems for which a solution can be verified in polynomial time
  - Examples: Fractional Knapsack,…, TSP, CNF SAT, 3-CNF SAT
- Clearly **P** $\subseteq$ **NP**
- Open question: Does **P** = **NP**?
  - P $\neq$ NP

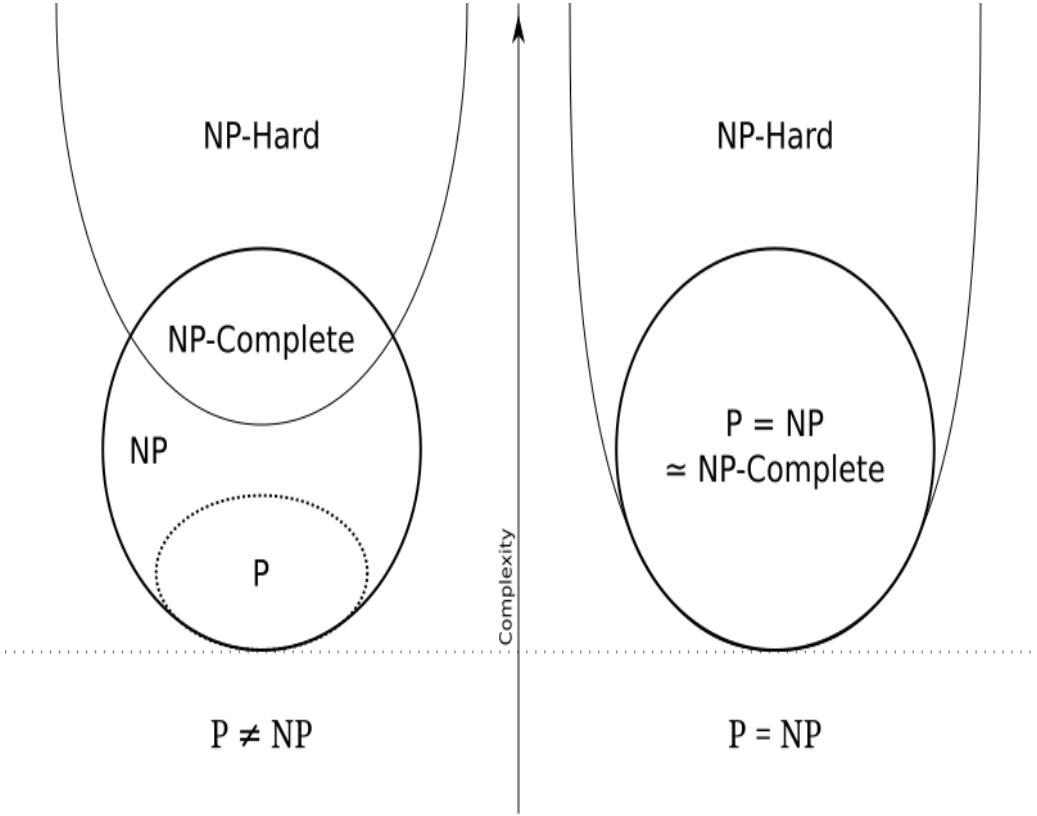
- A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, ...
- Ex:- Hamiltonian Cycle

NP-complete problems

- A problem is **NP-complete** if the problem is both
  - NP-hard, and
  - NP.

NP-Hard and NP-Complete

- If R is *polynomial-time reducible* to Q, we denote this $R \leq_p Q$
- Definition of NP-Hard and NP-Complete:
  - If all problems $R \in \mathbf{NP}$ are *polynomial-time* reducible to Q, then Q is *NP-Hard*
  - We say Q is *NP-Complete* if Q is NP-Hard and $Q \in \mathbf{NP}$
- If $R \leq_p Q$ and R is NP-Hard, Q is also NP-Hard



TRUE

1. P is a subset of NP
2. NPC is a subset of NP
3. NPC is a subset of NPH

NOT KNOWN

1. NP is a subset of P
2. P = NP

If P = NP

1. P = NP = NPC is a subset of NPH

If P is not equal to NP

1. P (intersection) NPC = empty
2. P (intersection) NPH = empty

How to show that a problem L is NP-Complete

1. Prove: L is in NP.
2. Prove: L is NP-Hard. That is, all problems in NP are poly-time reducible to L.

Statement 2 seems to be impractical since we cannot enumerate all NP problems and reduce it to L.

Trick: Cook-Levin Theorem : SAT is NP-Complete.

This means: SAT is NP-Hard and hence Every NP problem is reducible to SAT.

If we show SAT is reducible to L, then by transitivity, Every NP problem is reducible to L.
Hence L is NP-Hard.

Using Transitivity

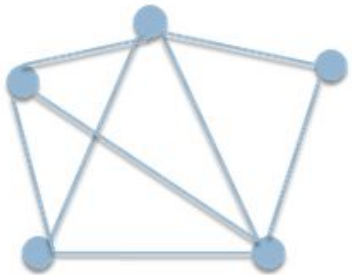
1. SAT is NP-Complete (Cook-Levin theorem) Proof Omitted
2. Hence, every NP problem is reducible to SAT.
3. SAT is reducible to 3SAT. Proof Omitted
4. Combining 2 and 3, 3SAT is NP-Hard.
5. We know that 3SAT is in NP and hence 3SAT is NP-Complete
6. What more? *(Proof idea will be discussed for the following)*
    a. CLIQUE, INDSET, VERTEX-COVER are in NP.
    b. 3SAT is reducible to CLIQUE
    c. CLIQUE is reducible to INDSET
    d. INDSET is reducible to VERTEX-COVER

Hence CLIQUE, INDSET, VERTEX-COVER are NP-Complete.

CLIQUE

A *clique* in an undirected graph G = (V, E) is a subset V' $\subseteq$ V of vertices that are fully connected, i.e. every vertex in V' is connected to every other vertex in V'

CLIQUE problem: Does G contain a clique of size k?



Is there a clique of size 4 in this graph?

CLIQUE in a graph

Every edge is a 2-clique

Hence for a given graph G, the problem of

"Does there exists a clique of size 2 in G"

has always an YES answer

"Does there exists a clique of size 3 in G"

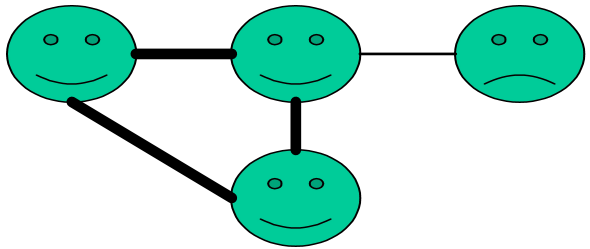
Yes for the depicted graph

No for ??? graph

"Does there exists a clique of size 4 in G"

No for the depicted graph

Size of maximum Clique is 3



First, prove that CLIQUE is in NP

It is simple to find a verifier for CLIQUE, using the nodes in the clique as the certificate:

- 1. Verify that the clique is a subgraph of the correct size
- 2. Check that the graph contains edges connecting the nodes in the clique.
- If 1 and 2 are both true, accept, otherwise reject.

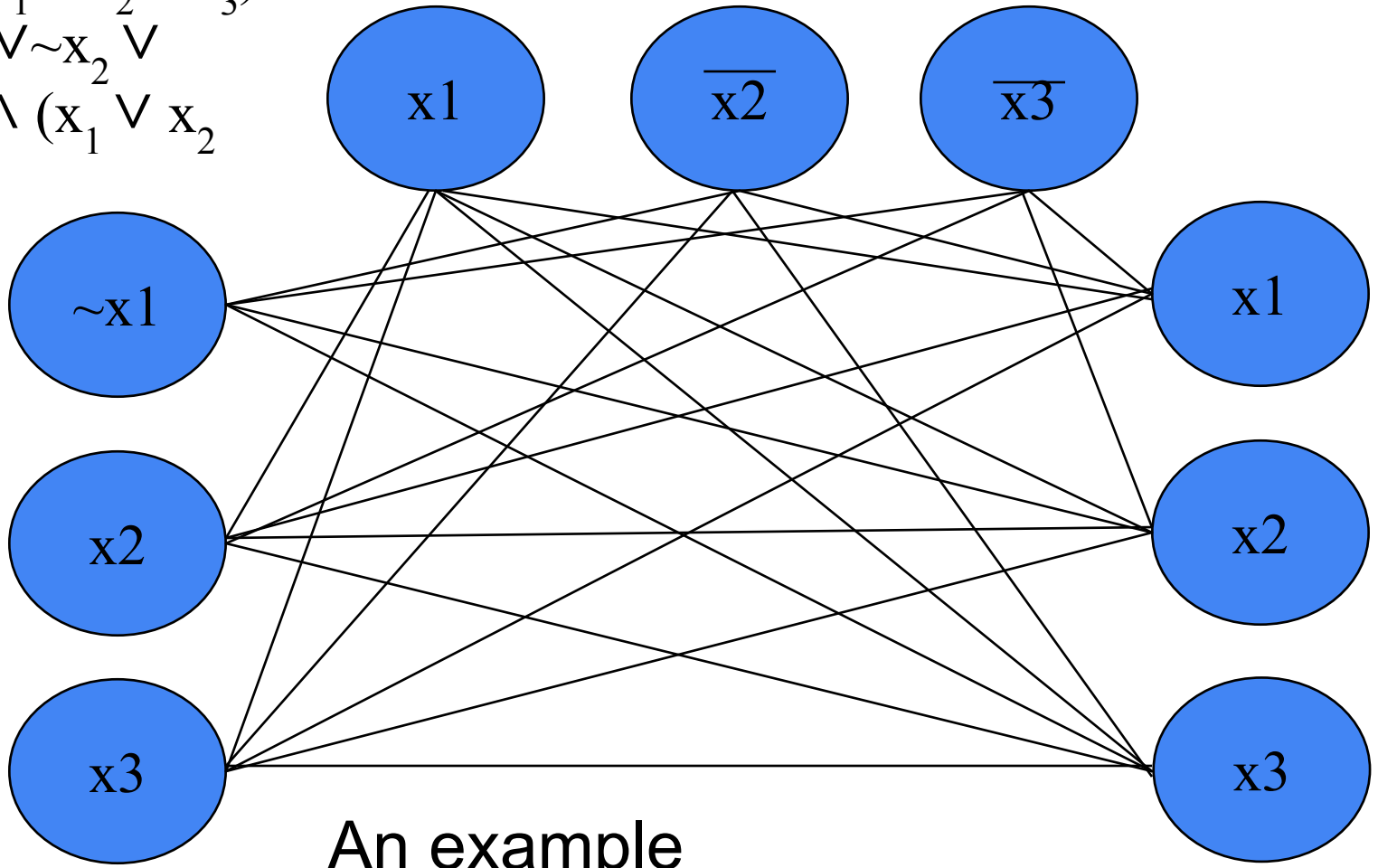
Now, a polynomial time reduction from 3SAT

- We will construct a graph G based on a formula φ.
- Let k be the number of clauses in φ.
- G will have a clique of size k iff φ is satisfiable.
-

Construction of G

- Each clause in φ will be represented as a set of three nodes, one for each literal.
- The nodes will all be connected, except that no nodes from the same clause will be connected and no two nodes with contradictory labels will be connected (i.e. x1 and ~x1 shall not share an edge)

$\varphi=(\sim x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \sim x_2 \vee \sim x_3) \wedge (x_1 \vee x_2 \vee x_3)$



An example

Proof (optional)

Theorem: φ is satisfiable iff G has a k-clique:

Assume φ is satisfiable. In each clause, at least one literal is true. Select one node which has a true literal as its label from each clause. The nodes selected form a k-clique since the nodes are selected one from each clause. The labels are connected since they cannot be contradictory.

(Conversely) Assume G has a k-clique. Assign each of the literals in the labels of the clique to be true. Because nodes from the same clause are not connected, a k-clique must visit k clauses. No contradictions will be encountered since contradictory labels are not connected. Thus each clause contains at least one true literal, which means that φ is satisfied.

Independent Set

Within a graph $G = (V, E)$, an *independent set* is a set of vertices $X \subseteq V$ such that E contains no edges between two vertices in X.

IndSet: Given k_i, $G_i = (V_i, E_i)$, does G_i have an independent set of size k_i?

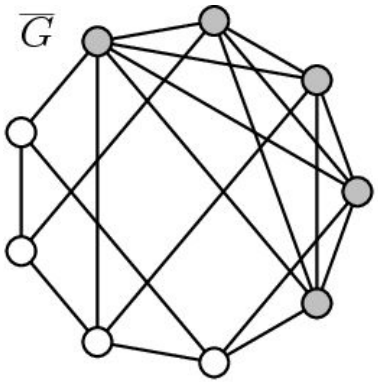
IndSet $\in \mathcal{NP}$. Give the independent set as a certificate to be verified, check the graph to make sure that no edges between its vertices exist.

Vertex Cover

Within a graph $G = (V, E)$, a *vertex cover* is a set of vertices $X \subseteq V$ such that, for each $e = (u, v) \in E$, $u \in X$ or $v \in X$ (or both).

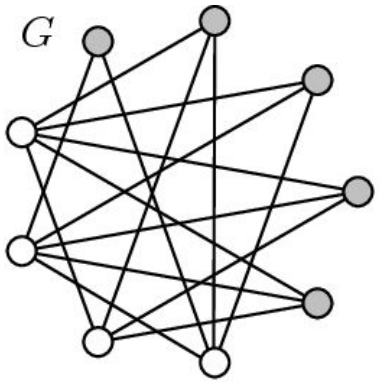
VertCov: Given $k \geq 0$, $G = (V, E)$, does G have a vertex cover of size k?

VertCov $\in \mathcal{NP}$. Give the vertex cover as a certificate to be verified, mark edges of each vertex in the given cover, verify that every edge in E is marked.



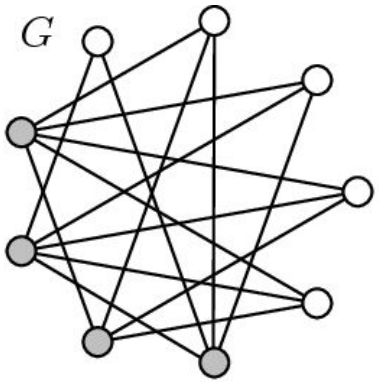
V' is a clique
of size k in $\overline{G}$

$\Leftrightarrow$



V' is a independent set
of size k in G

$\Leftrightarrow$



$V \setminus V'$ is a vertex cover
of size $n - k$ in G

CLIQUE $\leq_p$ INDSET

Given a black box that solves:
INDSET: Given k_i, $G_i = (V_i, E_i)$, does G_i have an independent set of size k_i?
how can you use it to solve:
CLIQUE: Given k_c, $G_c = (V_c, E_c)$, does G_c have a clique of size k_c?

Let $G_i = \overline{G_c}$, $k_i = k_c$.
G_c has a clique of size k_c if and only if G_i has an independent set of size k_i.

INDSET $\leq_p$ VERTCOV

Given a black box that solves:
VERTCOVER: Given k_v, $\mathbf{G_v} = (\mathbf{V_v}, \mathbf{E_v})$, does $\mathbf{G_v}$ have a vertex cover of size k_v?
how can you use it to solve:
INDSET: Given k_i, $G_i = (V_i, E_i)$, does G_i have an independent set of size k_i?

Let $\mathbf{G_v} = G_i$, $k_v = |V_i| - k_i$.
G_i has an ind. set of size k if and only if $\mathbf{G_v}$ has a vertex cover of size k_v.

OPTIONAL

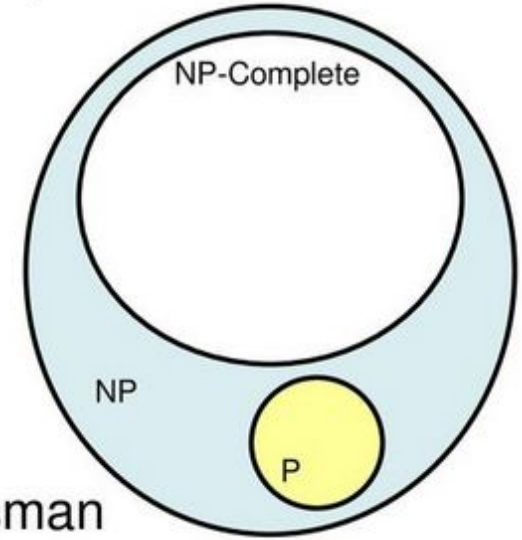
CLIQUE $\leq_p$ INDSET and INDSET $\leq_p$ VERTCOV, so CLIQUE $\leq_p$ VERTCOV.

CLIQUE $\leq_p$ VERTEX COVER

Given a black box that solves:
VERTCOVER: Given k_v, $\mathbf{G_v} = (\mathbf{V_v}, \mathbf{E_v})$, does $\mathbf{G_v}$ have a vertex cover of size k_v?
how can you use it to solve:
CLIQUE: Given k_c, $G_c = (V_c, E_c)$, does G_c have a clique of size k_c?

Let $\mathbf{G_v} = \overline{G_c}$, $k_v = |V_c| - k_c$.
G_c has a clique of size k_c if and only if $\mathbf{G_v}$ has a vertex cover of size k_v.

Populating the NP-Completeness Universe



- Circuit Sat $<_P$ 3-SAT
- 3-SAT $<_P$ Independent Set
- 3-SAT $<_P$ Vertex Cover
- Independent Set $<_P$ Clique
- 3-SAT $<_P$ Hamiltonian Circuit
- Hamiltonian Circuit $<_P$ Traveling Salesman
- 3-SAT $<_P$ Integer Linear Programming
- 3-SAT $<_P$ Graph Coloring
- 3-SAT $<_P$ Subset Sum
- Subset Sum $<_P$ Scheduling with Release times and deadlines

Approximation Algorithms

Q. Suppose I need to solve an NP-hard problem. What should I do?
A. Theory says you're unlikely to find a poly-time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in poly-time.
- Solve arbitrary instances of the problem.

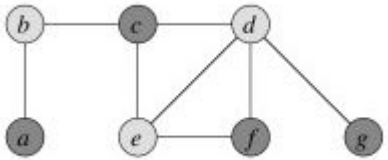
ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is!

Approximation Algorithm - Vertex Cover

The light grey vertices form the minimum vertex cover.



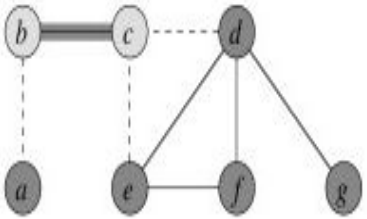
However the problem of finding vertex cover is NP-Complete. Hence we have a polynomial approximation algorithm

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

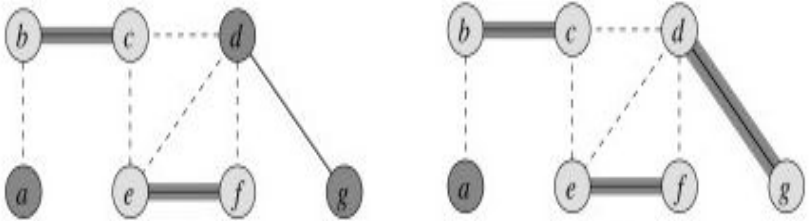
Example

Suppose we have this input graph:

Suppose then that edge $\{b, c\}$ is chosen. The two incident vertices are added to the cover and all other incident edges are removed from consideration:



Iterating now for edges $\{e, f\}$ and then $\{d, g\}$:



```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

OPTIONAL

Let *A* be the set of edges chosen in line 4 of the algorithm. Any vertex cover must cover at least one endpoint of every edge in *A*. No two edges in *A* share a vertex (see algorithm), so in order to cover *A*, the optimal solution C^* must have at least as many vertices:

$$|A| \leq |C^*|$$

Since each execution of line 4 picks an edge for which neither endpoint is yet in *C* and adds these two vertices to *C*, then we know that

$$|C| = 2|A|$$

Therefore:

$$|C| \leq 2|C^*|$$

That is, |*C*| cannot be larger than twice the optimal, so is a 2-approximation algorithm for Vertex Cover.