Introduction to Complexity Theory

Stefan D. Bruda

CS 317, Fall 2024

Problems and problems

- When designing algorithms we think about problems individually
- Nice to know in advance what kind of algorithms we can design
  - Convenient then to consider classes or problems with similar algorithmic properties
  - Focus on decision problems = problems with true/false answers or positive/negative instances
- Example of meaningful complexity classes:
  - Semi-decidable problems, for which we have algorithms that can provide the correct answer to positive instances but not to negative instances
    - Includes exactly all specifiable problems
  - Decidable problems, for which algorithms exist
  - Intractable problems, for which it is proven that no polynomial algorithm exist (decidability in Presburger arithmetic, position evaluation in Go, etc.)
  - Tractable problems, for which polynomial algorithms exist (sorting, shortest path, optimal BST, etc.)

- When designing algorithms we think about problems individually
- Nice to know in advance what kind of algorithms we can design
  - Convenient then to consider classes or problems with similar algorithmic properties
  - Focus on decision problems = problems with true/false answers or positive/negative instances
- Example of meaningful complexity classes:
  - Semi-decidable problems, for which we have algorithms that can provide the correct answer to positive instances but not to negative instances
    - Includes exactly all specifiable problems
  - Decidable problems, for which algorithms exist
  - Intractable problems, for which it is proven that no polynomial algorithm exist (decidability in Presburger arithmetic, position evaluation in Go, etc.)
  - Tractable problems, for which polynomial algorithms exist (sorting, shortest path, optimal BST, etc.)

  - Neither here nor there: problems with no known polynomial time algorithms but not proven to have no such an algorithm

THE HALTING PROBLEM

- Input: A string P describing an algorithm and a string w as input for P
- Output: TRUE if P halts on w and FALSE if P runs forever on w

Theorem (Alan Mathison Turing, 1939)

The halting problem is undecidable (semi-decidable but not decidable)

THE HALTING PROBLEM

- Input: A string P describing an algorithm and a string w as input for P
- Output: TRUE if P halts on w and FALSE if P runs forever on w

Theorem (Alan Mathison Turing, 1939)

The halting problem is undecidable (semi-decidable but not decidable)

- Suppose that the problem is decidable and so is solved by HALT(P, w)
- Consider then the following algorithm:

```
algorithm DIAG(x):
    if HALTS(x, x) then
        while TRUE do nothing
    else return TRUE
```

- Does DIAG(DIAG) halt?

The halting problem

- Input: A string P describing an algorithm and a string w as input for P
- Output: TRUE if P halts on w and FALSE if P runs forever on w

Theorem (Alan Mathison Turing, 1939)

The halting problem is undecidable (semi-decidable but not decidable)

- Suppose that the problem is decidable and so is solved by HALT(P, w)
- Consider then the following algorithm:

```
algorithm DIAG(x):
    if HALTS(x, x) then
        while TRUE do nothing
    else return TRUE
```

- Does DIAG(DIAG) halt?
  - If it halts then HALTS(DIAG, DIAG) returns TRUE (since HALTS solves the halting problem), which means that DIAG(DIAG) does not halt, a contradiction
  - If it does not halt; then HALTS(DIAG, DIAG) returns FALSE (since HALTS solves the halting problem), which means that DIAG(DIAG) halts and returns TRUE, another contradiction

Theorem (Henry Gordon Rice, 1951)

All nontrivial and extensional questions about algorithms are undecidable

Problems redefined

- Abstract problem: relation Q over the set I of problem instances and the set S of problem solutions: $Q \subseteq I \times S$
  - Complexity theory deals with decision problems or languages ($S = \{0, 1\}$)
    - I partitioned into positive and negative problem instances
    - Technically a language is a set of strings
    - A problem $Q \subseteq I \times \{0, 1\}$ ca be rewritten as the language (set) $L(Q) = \{w \in I : (w, 1) \in Q\}$
  - Many abstract problems are optimization problems instead; however, we can usually restate an optimization problem as a decision problem which requires the same amount of resources to solve

Problems redefined

- Abstract problem: relation Q over the set I of problem instances and the set S of problem solutions: $Q \subseteq I \times S$
  - Complexity theory deals with decision problems or languages ($S = \{0, 1\}$)
    - I partitioned into positive and negative problem instances
    - Technically a language is a set of strings
    - A problem $Q \subseteq I \times \{0, 1\}$ ca be rewritten as the language (set) $L(Q) = \{w \in I : (w, 1) \in Q\}$
  - Many abstract problems are optimization problems instead; however, we can usually restate an optimization problem as a decision problem which requires the same amount of resources to solve
- Concrete problem: an abstract decision problem with $I = \{0, 1\}^*$
  - Abstract problem mapped on concrete problem using an encoding $e : I \to \{0, 1\}^*$
  - $Q \subseteq I \times \{0, 1\}$ mapped to the concrete problem $e(Q) \subseteq e(I) \times \{0, 1\}$
  - Encodings will not affect the performance of an algorithm as long as they are polynomially related
- An algorithm solves a concrete problem in time $O(T(n))$ whenever the algorithm produces in $O(T(n))$ time a solution for any problem instance i with $|i| = n$

LANGUAGES? PROBLEMS?

- Complexity theory analyzes problems from the perspective of how many resources (e.g., time, storage) are necessary to solve them
  - Given some abstract problem that requires certain resource (time) bounds to solve, it is generally easy to find a language that requires the same resource bounds to decide
  - Sometime (but not always) finding an algorithm for deciding the language immediately implies an algorithm for solving the problem

- Complexity theory analyzes problems from the perspective of how many resources (e.g., time, storage) are necessary to solve them
  - Given some abstract problem that requires certain resource (time) bounds to solve, it is generally easy to find a language that requires the same resource bounds to decide
  - Sometime (but not always) finding an algorithm for deciding the language immediately implies an algorithm for solving the problem
- Traveling salesman (TSP): Given $n \geq 2$, a matrix $(d_{ij})_{1 \leq i,j \leq n}$ with $d_{ij} > 0$ and $d_{ii} = 0$, find a permutation π of $\{1, 2, \ldots, n\}$ such that $c(\pi)$, the cost of π is minimal, where $c(\pi) = d_{\pi_1\pi_2} + d_{\pi_2\pi_3} + \cdots + d_{\pi_{n-1}\pi_n} + d_{\pi_n\pi_1}$
  - TSP the language (take 1): $\{((d_{ij})_{1 \leq i,j \leq n}, B) : n \geq 2, B \geq 0$, there exists a permutation π such that $c(\pi) \leq B\}$
  - TSP the language (take 2), or the Hamiltonian graphs: Exactly all the graphs that have a (Hamiltonian) cycle that goes through all the vertices exactly once

- Clique: Given an undirected graph $G = (V, E)$, find the maximal set $C \subseteq V$ such that $\forall\, v_i, v_j \in C : (v_i, v_j) \in E$ (C is a clique of G)
  - Clique, the language: $\{(G = (V, E), K) : K \geq 2 :$ there exists a clique C of V such that $|C| \geq K\}$

- Clique: Given an undirected graph $G = (V, E)$, find the maximal set $C \subseteq V$ such that $\forall\, v_i, v_j \in C : (v_i, v_j) \in E$ (C is a clique of G)
  - Clique, the language: $\{(G = (V, E), K) : K \geq 2 :$ there exists a clique C of V such that $|C| \geq K\}$
- SAT: Fix a set of variables $X = \{x_1, x_2, \ldots, x_n\}$ and let $\overline{X} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$
  - An element of $X \cup \overline{X}$ is called a literal
  - A formula (or set/conjunction of clauses) is $\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m$ where $\alpha_i = x_{a_1} \vee x_{a_2} \vee \cdots \vee x_{a_k}$, $1 \leq i \leq m$, and $x_{a_i} \in X \cup \overline{X}$
  - An interpretation (or truth assignment) is a function $I : X \to \{\top, \bot\}$
  - A formula F is satisfiable iff there exists an interpretation under which F evaluates to $\top$.
  - SAT $= \{F : F$ is satisfiable $\}$
- 2-SAT, 3-SAT are variants of SAT (with the number of literals in every clause restricted to a maximum of 2 and 3, respectively)

Languages? Problems? (cont'd)

- Clique: Given an undirected graph $G = (V, E)$, find the maximal set $C \subseteq V$ such that $\forall\, v_i, v_j \in C : (v_i, v_j) \in E$ (C is a clique of G)
  - Clique, the language: $\{(G = (V, E), K) : K \geq 2 :$ there exists a clique C of V such that $|C| \geq K\}$
- SAT: Fix a set of variables $X = \{x_1, x_2, \ldots, x_n\}$ and let $\overline{X} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$
  - An element of $X \cup \overline{X}$ is called a literal
  - A formula (or set/conjunction of clauses) is $\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m$ where $\alpha_i = x_{a_1} \vee x_{a_2} \vee \cdots \vee x_{a_k}$, $1 \leq i \leq m$, and $x_{a_i} \in X \cup \overline{X}$
  - An interpretation (or truth assignment) is a function $I : X \to \{\top, \bot\}$
  - A formula F is satisfiable iff there exists an interpretation under which F evaluates to $\top$.
  - SAT $= \{F : F$ is satisfiable $\}$
- 2-SAT, 3-SAT are variants of SAT (with the number of literals in every clause restricted to a maximum of 2 and 3, respectively)
- Note in passing: Sometimes SAT (2-SAT, 3-SAT) is called CNF (2-CNF, 3-CNF) because the input formulae are written in conjunctive normal form

NONDETERMINISTIC ALGORITHMS

- A nondeterministic algorithm is an algorithm that can be in more places at once while deciding a problem
  - Sole additional operation is the nondeterministic guess of a bit: GUESS returns 0 and 1 at the same time
  - After a guess the algorithm continues in parallel for both cases 0 and 1
  - The algorithm returns TRUE iff at least one of the parallel paths return TRUE
  - Running time: the running time of the longest parallel path
  - Example:

algorithm ISCOMPOSITE(k)**:**
- $f_1 \leftarrow 1$
- $f_2 \leftarrow 1$
- **for** $i = 1$ **to** $\log k$ **do** $f_1 \leftarrow 2 \times f_1 + \text{GUESS}$
- **for** $i = 1$ **to** $\log k$ **do** $f_2 \leftarrow 2 \times f_2 + \text{GUESS}$
- **return** $k = f_1 \times f_2$

    - Running time: $O(n + t_\times(n))$, with $t_\times(n)$ the time it takes to multiply n-bit numbers
    - Correctness: f_1 and f_2 range over all $\log k$-bit numbers = all possible factors of n
    - If $f_1 \times f_2$ is never equal to n then all paths return FALSE (so ISCOMPOSITE returns FALSE), otherwise at least one path returns TRUE (so ISCOMPOSITE returns TRUE)

NONDETERMINISTIC ALGORITHMS (CONT'D)

Theorem

A nondeterministic algorithm with $r(n)$ running time can be simulated by a (deterministic) algorithm in $O(2^{r(n)})$ time

Nondeterministic algorithms (cont'd)

Theorem

A nondeterministic algorithm with $r(n)$ running time can be simulated by a (deterministic) algorithm in $O(2^{r(n)})$ time

- After a guess follow the two paths sequentially, one after the other
- The running time doubles after each guess
- In the worst case every step is a guess, hence the $O(2^{r(n)})$ overall running time

Nondeterministic algorithms (cont'd)

Theorem

A nondeterministic algorithm with $r(n)$ running time can be simulated by a (deterministic) algorithm in $O(2^{r(n)})$ time

- After a guess follow the two paths sequentially, one after the other
- The running time doubles after each guess
- In the worst case every step is a guess, hence the $O(2^{r(n)})$ overall running time

- Alternatively, think about the running paths of a deterministic algorithm as a sequence of states
  - The length of the sequence is the running time
- By contrast the running time of a nondeterministic algorithm branches at each guess forming a binary tree
  - The running time is the height of the tree
  - A deterministic algorithm has to traverse the whole tree

A few complexity classes

- $\mathcal{P}$: The class of exactly all problems solved by (deterministic) algorithms running in $\text{poly}(n) = n^{O(1)}$ time
- $\mathcal{NP}$: The class of exactly all problems solved by nondeterministic algorithms running $\text{poly}(n)$ time
- $\mathcal{EXP}$: The class of exactly all problems solved by (deterministic) algorithms running in $2^{n^{O(1)}}$ time

Corollary

$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP}$

- True or false: $\mathcal{P} = \mathcal{NP}$

- $\mathcal{P}$: The class of exactly all problems solved by (deterministic) algorithms running in poly(n) $= n^{O(1)}$ time
- $\mathcal{NP}$: The class of exactly all problems solved by nondeterministic algorithms running poly(n) time
- $\mathcal{EXP}$: The class of exactly all problems solved by (deterministic) algorithms running in $2^{n^{O(1)}}$ time

Corollary

$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP}$

- True or false: $\mathcal{P} = \mathcal{NP}$ – open question (since 1971, arguably earlier)
- Algorithms for problems in $\mathcal{NP}$ consist of a nondeterministic guessing step followed by a deterministic verification step
  - Clique: guess a set of vertices, then verify that the guessed set is a clique
  - Hamiltonian cycle: guess a permutation of vertices, then verify that the guessed permutation forms a cycle

A FEW COMPLEXITY CLASSES

- $\mathcal{P}$: The class of exactly all problems solved by (deterministic) algorithms running in $\text{poly}(n) = n^{O(1)}$ time
- $\mathcal{NP}$: The class of exactly all problems solved by nondeterministic algorithms running $\text{poly}(n)$ time
- $\mathcal{EXP}$: The class of exactly all problems solved by (deterministic) algorithms running in $2^{n^{O(1)}}$ time

Corollary

$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP}$

- True or false: $\mathcal{P} = \mathcal{NP}$ – open question (since 1971, arguably earlier)
- Algorithms for problems in $\mathcal{NP}$ consist of a nondeterministic guessing step followed by a deterministic verification step
  - Clique: guess a set of vertices, then verify that the guessed set is a clique
  - Hamiltonian cycle: guess a permutation of vertices, then verify that the guessed permutation forms a cycle
- Alternative definition for $\mathcal{NP}$: A problem $Q \subseteq I \times \{0, 1\}$ is in $\mathcal{NP}$ iff the following problem is in $\mathcal{P}$: Given $w \in I$, determine whether $(w, 1) \in Q$
  - The problem becomes easy if we take the guess out

EXAMPLES OF NONDETERMINISTIC ALGORITHMS

algorithm KNAPSACK(C, n, p, w, K):
- // guess a set of objects
- $O \leftarrow \emptyset$
- **for** $i = 1$ **to** n **do**
  - **if** GUESS $= 1$ **then** $O \leftarrow O \cup \{i\}$
- // calculate the weight and profit
- $W \leftarrow 0$
- $P \leftarrow 0$
- **foreach** $i \in O$ **do**
  - $W \leftarrow W + w_i$
  - $P \leftarrow P + p_i$
- **return** $W \leq C \wedge P \geq K$

algorithm CLIQUE$(G = (V, E), K)$:
- // guess a set of vertices
- $C \leftarrow \emptyset$
- **foreach** $v \in V$ **do**
  - **if** GUESS $= 1$ **then** $C \leftarrow C \cup \{v\}$
- // check if C is a clique
- **foreach** $u \in C$ **do**
  - **foreach** $v \in C$ **do**
    - **if** $u \neq v \wedge (u, v) \notin E$ **then**
      - **return** FALSE
- **return** $|C| \geq K$

algorithm GUESSNUMBER(n):
- $k \leftarrow 0$
- **for** $i = 1$ **to** $\log n$ **do**
  - $k \leftarrow 2 \times k +$ GUESS
- **return** k

algorithm TSP$(d_{1\dots n, 1\dots n}, K)$:
- // guess n numbers
- $\pi \leftarrow \langle\rangle$
- **for** $i = 1$ **to** n **do**
  - $\pi \leftarrow \pi + \langle$GUESSNUMBER$(n)\rangle$
- // verify that π is a permutation
- **for** $i = 1$ **to** n **do**
  - **for** $j = 1$ **to** n **do**
    - **if** $\pi_i = \pi_j$ **then** **return** FALSE
- // calculate the cost of cycle π
- $c \leftarrow 0$
- **for** $i = 1$ **to** n **do**
  - $c \leftarrow c + d_{\pi_i, \pi_{(i+1) \bmod n}}$
- **return** $c \leq K$

```
algorithm KNAPSACK(C, n, p, w, K):
    // guess a set of objects
    O ← ∅
    for i = 1 to n do
        if GUESS = 1 then  O ← O ∪ {i}
    // calculate the weight and profit
    W ← 0
    P ← 0
    foreach i ∈ O do
        W ← W + w_i
        P ← P + p_i
    return W ≤ C ∧ P ≥ K

algorithm CLIQUE(G = (V, E), K):
    // guess a set of vertices
    C ← ∅
    foreach v ∈ V do
        if GUESS = 1 then  C ← C ∪ {v}
    // check if C is a clique
    foreach u ∈ C do
        foreach v ∈ C do
            if u ≠ v ∧ (u, v) ∉ E then
                return FALSE
    return |C| ≥ K

algorithm GUESSNUMBER(n):
    k ← 0
    for i = 1 to log n do
        k ← 2 × k + GUESS
    return k

algorithm TSP(d_{1...n,1...n}, K):
    // guess n numbers
    π ← ⟨⟩
    for i = 1 to n do
        π ← π + ⟨GUESSNUMBER(n)⟩
    // verify that π is a permutation
    for i = 1 to n do
        for j = 1 to n do
            if π_i = π_j then  return FALSE
    // calculate the cost of cycle π
    c ← 0
    for i = 1 to n do
        c ← c + d_{π_i, π_((i+1) mod n)}
    return c ≤ K
```

- All the "brute force" solutions discussed earlier are effectively polynomial time nondeterministic algorithms!

- A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q'
  - If Q reduces to Q' then Q is "not harder to solve" than Q'

- A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q'
  - If Q reduces to Q' then Q is "not harder to solve" than Q'
- Polynomial reduction: A language L_1 is polynomial-time reducible to a language L_2 ($L_1 \leq_P L_2$) iff there exists a polynomial algorithm F that computes the function $f : \{0,1\}^* \to \{0,1\}^*$ such that
  $$\forall x \in \{0,1\}^* : x \in L_1 \text{ iff } f(x) \in L_2$$
  - Polynomial reductions show that a problem is not harder to solve than another within a polynomial-time factor

- A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q'
  - If Q reduces to Q' then Q is "not harder to solve" than Q'
- Polynomial reduction: A language L_1 is polynomial-time reducible to a language L_2 ($L_1 \leq_P L_2$) iff there exists a polynomial algorithm F that computes the function $f : \{0,1\}^* \to \{0,1\}^*$ such that
$$\forall x \in \{0,1\}^* : x \in L_1 \text{ iff } f(x) \in L_2$$
  - Polynomial reductions show that a problem is not harder to solve than another within a polynomial-time factor

Lemma

1. *$\leq_P$ is a preorder (reflexive and transitive but not necessarily symmetric or antisymmetric)*
2. $L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P$

- A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q'
  - If Q reduces to Q' then Q is "not harder to solve" than Q'
- Polynomial reduction: A language L_1 is polynomial-time reducible to a language L_2 ($L_1 \leq_P L_2$) iff there exists a polynomial algorithm F that computes the function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that

$$\forall x \in \{0,1\}^* : x \in L_1 \text{ iff } f(x) \in L_2$$

  - Polynomial reductions show that a problem is not harder to solve than another within a polynomial-time factor

Lemma

1. *$\leq_P$ is a preorder (reflexive and transitive but not necessarily symmetric or antisymmetric)*
2. $L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P$

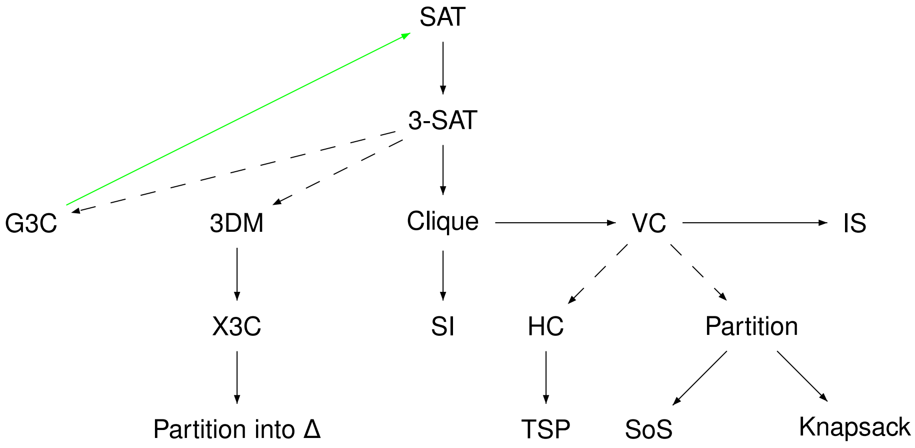
- A problem L is NP-hard iff $\forall L' \in \mathcal{NP} : L' \leq_P L$
- A problem L is NP-complete ($L \in \mathcal{NPC}$) iff L is NP-hard and $L \in \mathcal{NP}$

Theorem

Let L be some NP-complete problem; then $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$

- Are there NP-complete problems at all?
  - SAT $\in \mathcal{NPC}$ (Stephen Cook, 1971)
- The first is the hard one: need to show that every problem in $\mathcal{NP}$ reduces to our problem
- Then in order to find other NP-complete problems all we need to do is to find problems such that some problem already known to be NP-complete reduces to them
  - This works because polynomial reductions are closed under composition = are transitive
- Then it is apparently easy to use the theorem stated earlier:

Let L be some NP-complete problem; then $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$



SAT
3-SAT
G3C
3DM
Clique
VC
IS
X3C
SI
HC
Partition
Partition into Δ
TSP
SoS
Knapsack

- 3-Dimensional Matching (3DM):
  - Input: A set $M \subseteq W \times X \times Y$ where W, X and Y are disjoint sets having the same number q of elements
  - Question: Does M contain a matching?
  - A matching is a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements in M' agree in any position
- Vertex Cover (VC):
  - Input: A Graph $G = (V, E)$ and an integer k, $0 \leq k \leq |V|$
  - Question: Is there a vertex cover of size less than k that is, a subset $V' \subseteq V$, $|V'| \leq k$ such that for all edges $(u, v) \in E$ we have $u \in V' \vee v \in V'$?
- Independent Set (IS):
  - Input: A Graph $G = (V, E)$ and an integer k, $0 \leq k \leq |V|$
  - Question: Does G contain an independent set of size larger than k that is, a subset $V' \subseteq V$, $|V'| \geq k$ such that $(u, v) \notin E$ for all $u, v \in V'$?
- Partition:
  - Input: A finite set A and a size $s(a) \in \mathbb{N}$ for each $a \in A$
  - Question: Is there $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \notin A'} s(a)$?

- Sum of Subsets (SoS):
  - Input: A finite set A, a size $s(a) \in \mathbb{N}$ for each $a \in A$, and $B \in \mathbb{N}$
  - Question: Is there $A' \subseteq A$ such that $\sum_{a\in A'} s(a) = B$?
- Graph 3-Colorability (G3C):
  - Input: A graph G
  - Question: Is the chromatic number of G less than 3?
- Subgraph Isomorphism (SI):
  - Input: Two graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$
  - Question: Does G contain a subgraph isomorphic to H that is, a subgraph $G' = (V, E)$ such that $V \subseteq V_1$, $E \subseteq E_1$, $|V| = |V_2|$, $|E| = |E_2|$, and there is a one-to-one correspondence between E and E_2?
- Exact Covering by 3 Sets (X3C):
  - Input: A finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X
  - Does C contain an exact cover for X that is, a subcollection $C' \subseteq C$ s.t. $|C'| = q$ and every element in X occurs in exactly one member of C'?
- Partition into Triangles:
  - Input: A Graph $G = (V, E)$ such that $|V| = 3q$
  - Question: Is there a partition of V into q disjoint sets $V_1, V_2, \ldots, V_q$ of 3 vertices each such that for each $V_i = v_{i1}, v_{i2}, v_{i3}$ we have $\{(v_{i1}, v_{i2}), (v_{i2}, v_{i3}), (v_{i3}, v_{i1})\} \subseteq E$?

Hard to classify problems

- There are problems that are known to be in neither $\mathcal{P}$ nor $\mathcal{NPC}$
- Example: the language of composite numbers (aka the integer factorization problem)
  - In $\mathcal{NP}$
  - Its complement also in $\mathcal{NP}$
  - Suspected outside $\mathcal{P}$
  - Suspected outside $\mathcal{NPC}$
  - Its placement outside $\mathcal{P}$ crucial to modern cryptography