P and NP

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

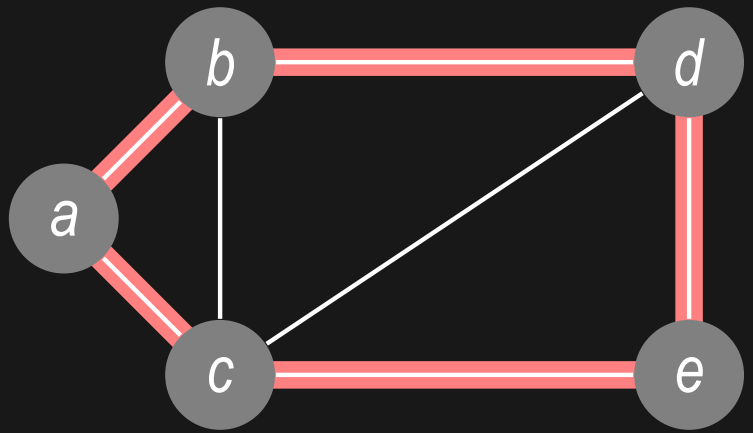
November 24, 2024

Problem Without Polynomial Time Solutions

- So far the algorithms we have considered have all been polynomial in the input size.
  - For example, $O(nm^2), O(n), O(n^2), O(n \log n)$.
- These running times allow for fairly large inputs to be solved efficiently in practice.
- Can all (or most) interesting computational problems be solved in polynomial time?
- It seems unlikely. Some problems may be too difficult.

The Traveling Salesman Problem

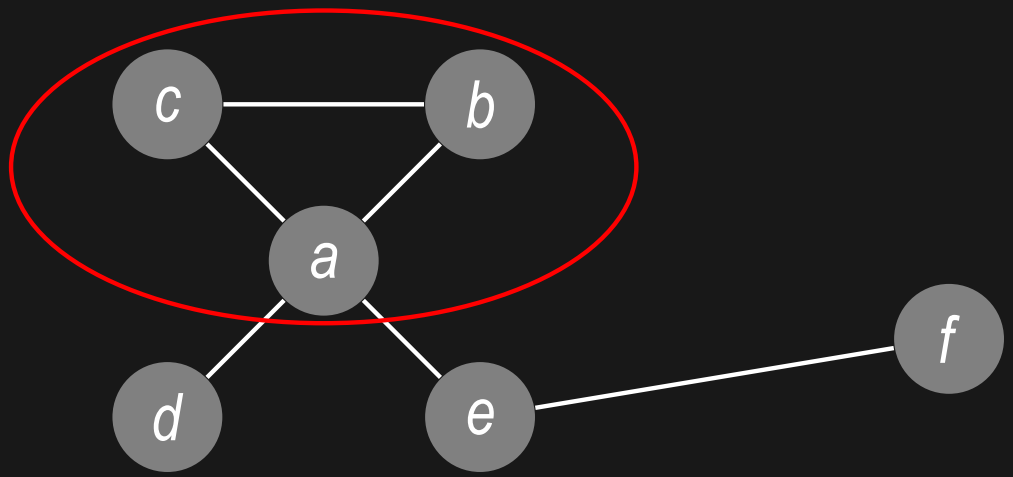
- An example of a difficult problem is the **traveling salesman problem** (TSP).
  - Input: undirected graph with weights on the edges.
  - Output: a shortest cycle which visits each vertex exactly once.
- Idea: There are some set of cities that a salesman needs to visit, and the output will produce the most efficient way to visit each of the cities.



- The best known algorithm: $O(n2^n)$ (exponential in the size of the input).

The Clique Problem

- Another example of a difficult problem is the **clique** problem.
  - Input: undirected graph $G = (V, E)$.
  - Output: largest subset C of V such that every pair of vertices in C has an edge between them (C is called a *clique*).



- Again, the best known algorithm is $O(n2^n)$.

Do Polynomial Solutions for Hard Problems Exist or NOT?

- It would be great if we could design a polynomial time algorithm for these difficult problems. Research has tried for decades and so far has not been able to design such an algorithm.
- This begs the question as to whether we could prove that it is not possible to design such an algorithm.

Similarity of the Difficulty of Some Problems

- If we can show that a problem Π is equivalent to other well studied problems without efficient algorithms, then that strongly suggests that Π does not have an efficient algorithm either.
  - That is, although we cannot solve many problems very fast, we can show that these problems are equivalent. If one day we have a fast solution for one of them, we have fast solutions for all of them.
- This technique has worked for over 10,000 hard problems.

P and NP

The Decision Problem

- A decision problem is a problem in which we want to determine the answer to a yes/no question regarding the input.
- An example of a decision problem using clique:
  - Input: an undirected graph G and a positive integer k.
  - Question: Does G contain a clique of size at least k?
  - Output: Yes or No.
- We could also consider *optimization* variants of the clique problem:
  1. What is the largest k such that G contains a clique of size k?
  2. What is the largest clique of G?

Decision Problems V.S. Normal Optimization Problems

- For each optimization problem, we can always define a corresponding decision problem.
  - In fact in Turing machine model, all problems are decision problems.
  - The solution to the decision problem can be used to solve the optimization problems.
- For example, consider the the decision and optimization problems for the clique problem in the previous slide:
  - If we have a polynomial time solution S_d to the decision problem, we can then run S_d for each $k \in 1, 2, 3, \ldots, n$ and output the largest k such that S_d return yes. This largest k answers the optimization problem.

The Class of P Problems

- The class of P: if a decision problem Π can be solved by a polynomial time algorithm, then Π is in P.
- Since every problem solvable by a computer can be expressed as a decision problem, we can view that all problems in the class of P has polynomial time algorithms.

The Verifier

- To understand the class of NP, we need to learn another concept, the verifier.
- Intuitively, a verifier of a decision problem is an algorithm that determines whether an solution can help the decision problem gives a "Yes" output.
  - For example, in the clique decision problem, if given an subset vertices V' of a graph $G(V, E)$, the verifier verifies if V' is a k-clique or not. If V' is a k-clique, then the clique decision problem can answer "Yes" for this graph.
  - A verifier does not determine the existence of a solution, but verifies the validity of a solution.

The Class of NP Problems

- The class of NP: if a decision problem Π has a polynomial time verifier, then Π is in NP.
  - E.g., the decision variants of clique and TSP are in NP since they have polynomial time verifiers.
- The NP comes from the "Nondeterministic Polynoimal-time" Turning machine.
- Definitions of P and NP using Turning machine.
  - A NP problem can be decided with a Nondeterministic Polynoimal time Turning machine.
  - A P problem can be decided with a Deterministic Polynoimal time Turning machine.

P V.S. NP

- Note that P problems are defined using the decision problems themselves, while NP problems are defined using the verifiers.
- Every problem that is in P is also in NP (i.e. P is a subset of NP). If the problem can be solved in polynomial time, then clearly a solution can be verified to be correct in polynomial time.
- A major open question (perhaps the largest open question in all of computer science), is whether there exists a problem that is in NP that is *not* in P. In other words, does P = NP?

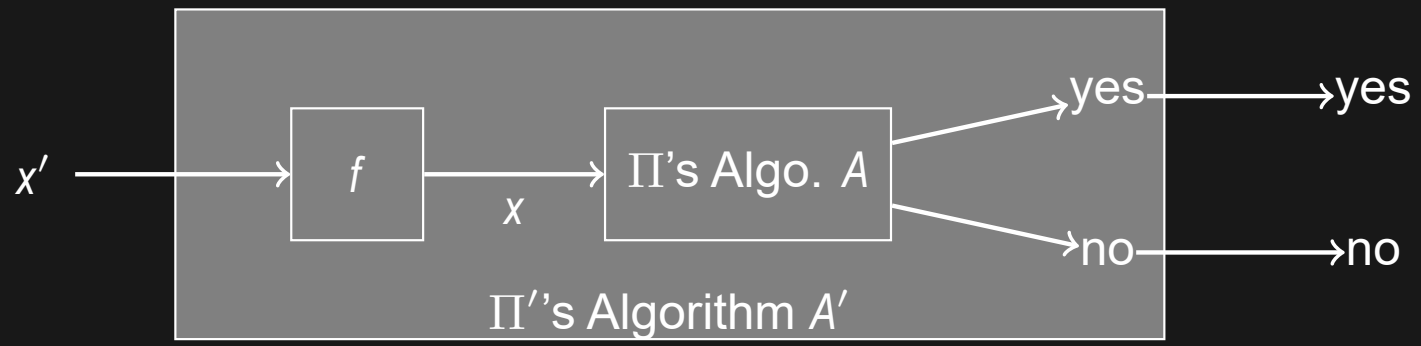
Reducibility

Grouping Problems

- While we don't know if $P = NP$, we do know have to determine if two problems have the same level of difficulty.
  - Therefore, if one problem has polynomial time solution, then the other problem also has polynomial time solution. If one problem does not have polynomial time solution, then the other problem does not have polynomial time solution.
- The process to prove two problems have same level of difficulty is called reduction.

Reducibility

- Π' is polynomial time reducible to Π ($\Pi' \leq_p \Pi$) iff there is a polynomial time function f that maps inputs x' for Π' into inputs x for Π such that for any x' we have $\Pi'(x') = \Pi(f(x'))$.



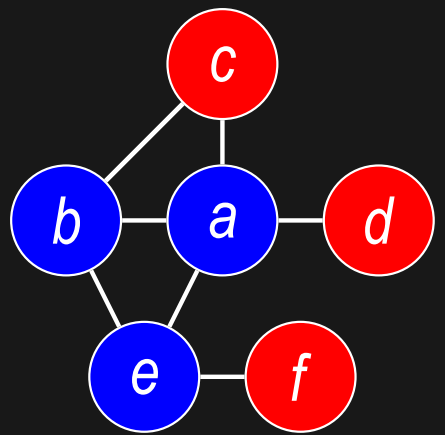
- Intuitively, we need the following to hold true:
  - We can convert x' into an input x such that x' is a "yes instance" for Π' iff x is a "yes instance" for Π.
- If $\Pi' \leq_p \Pi$ then the following facts are true:
  1. if Π is in P, then so is Π'.
  2. if Π is in NP, then so is Π'.
  3. if $\Pi'' \leq_p \Pi'$, then $\Pi'' \leq_p \Pi$ (transitivity).

Intuition of the Reduction Symbol

- If B can be reduced to A ($B \mapsto A$) in polynomial time, it implies
  - If A has a polynomial time algorithm, then B at least has a polynomial time algorithm.
  - If A has a non-polynomial time algorithm, then B at least has a non-polynomial time algorithm.
  - In other words, B is no more harder than A, or, A is harder than or as hard as B.
- Therefore, personally, I don't view $B \leq_p A$ as a reduction representation. Instead, I view it as a relationship of hardness, i.e., A is harder than, or as hard as, B.

An Example of Reduction: Independent Set $\Leftrightarrow$ Clique

- An *independent set* of a graph is a subset of the vertices such that no pair of vertices has an edge between them.
- The independent set problem (IS) is to determine if a graph G has an independent set of size at least k.
- For example, red nodes constitute an independent set, while blue nodes constitute a 3-clique.

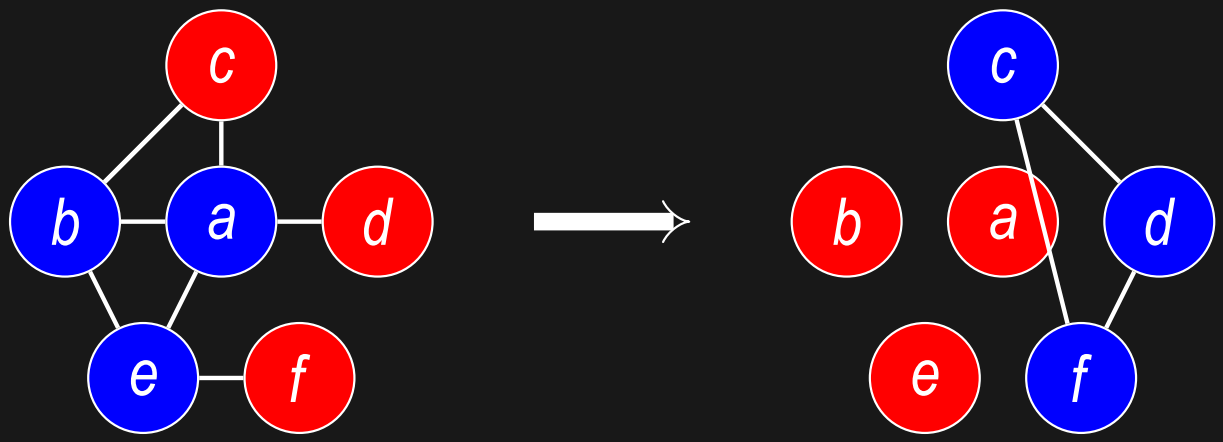


Decision Problems for Independent Set and Clique

- The decision problem for independent set problem:
  - Input: A graph G and an integer k.
  - Problem: Decide if G has an independent set of size $\geqslant k$.
- The decision problem for clique problem:
  - Input: A graph G and an integer k.
  - Problem: Decide if G has a clique of size $\geqslant k$.
- The idea of reducing between independent set and clique problem:
  - If we remove all the edges in a k-clique set V', we can get a k-independent set.
  - If we add edges to all vertices-pairs in a k- V', we can get a k-clique set.

Reducing Independent Set to Clique

1. Convert G to $\overline{G}$, in which $\{u, v\}$ is an edge in $\overline{G}$ *iff* $\{u, v\}$ is not an edge of G. That is, $\overline{G}$ is a complement of G.
   - This conversion is polynomial time since there are at most $O(|V|^2)$ pair of vertices to evaluate.
2. If we can find k-clique V' in $\overline{G}$, then V' is also a a k-independent set of G. Therefore, independent set $\leq_p$ clique.



Note that the above example does not show all the edges of $\overline{G}$.

The Hardest NP Problems

- We now have a method of showing that one problem Π' at worst as hard as another problem Π.
- Suppose there was a problem Π such that for any Π' in NP, it was true that $\Pi' \leq_p \Pi$. That is, Π is as hard as the the hardest problem in NP (or Π is NP-Complete).

Proof of NP-Completeness

Proving a Problem is NP-Complete

- We can prove a decision problem Π is NP-complete following the definition of NP-Complete.
  1. We need to prove Π is NP, by constructing a polynomial time verifier for it. For many problems, this step is simple.
  2. We need to prove that Π is NP hard, by showing that every NP problems can be reduced to Π.
- How do we enumerate the infinite set of NP problems?
  - It is hard, and usually we don't.

Proving a Problem is NP-Complete cont.

- The typical proof strategy to show Π is NP-complete:
  1. We start with a problem Π_{base} that is known to be NP-complete.
  2. Prove Π is NP using either one of the following methods:
     - 2.1 We prove that Π has a polynomial time verifier.
     - 2.2 We prove that Π can be reduced to Π_{base} (i.e, $\Pi \leq_p \Pi_{base}$). This step shows that Π is NP (i.e., a polynomial time verifier for Π_{base} also work on Π).
  3. We prove that Π_{base} can be reduced to Π (i.e, $\Pi_{base} \leq_p \Pi$). By the transitivity of reduction, this step shows all NP problems can be reduced to Π. Therefore, Π is NP-hard.

The First NP-Complete Problem

- The challenge now is that we need to find a NP-Complete problem as Π_{base}.
- The first problem shown to be NP-complete is the Satisfiability problem (SAT):
  - Given: A set of *n* Boolean variables, and a formula ϕ based on these variables:
  - Determine if there exists True/False assignments to the variables so that ϕ is satisfied.
  - For example, is there an T/F assignment to x_1, x_2, x_3 that makes the following Boolean formula True?

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

- The proof showing that SAT is NP-Complete does show that ALL NP problems can be reduced to SAT.

The First NP-Complete Problem cont.

- In 1971, Cook proved that SAT is NP-hard. Note that SAT is in NP (given an assignment, we can determine if it is a satisfying assignment in polynomial time). Therefore SAT is also NP-complete.
- The general idea of the proof:
  - By the definition of NP, given any decision problem in NP, we can construct a nondeterministic Turning machine to solve it in polynomial time.
  - For each input to that machine, build a Boolean formula with four Boolean variables to represent four facts: the inputs are passed to the machine, the machine runs correctly, the machines halts, and the machine answers "Yes" for the decision problem.
  - This Boolean formula is satisfied if and only if there exist inputs for the machine to run correctly and answers "Yes," which is equivalent to determining if machine (and the decision problem) answers "Yes."

Examples of NP-Complete Proof

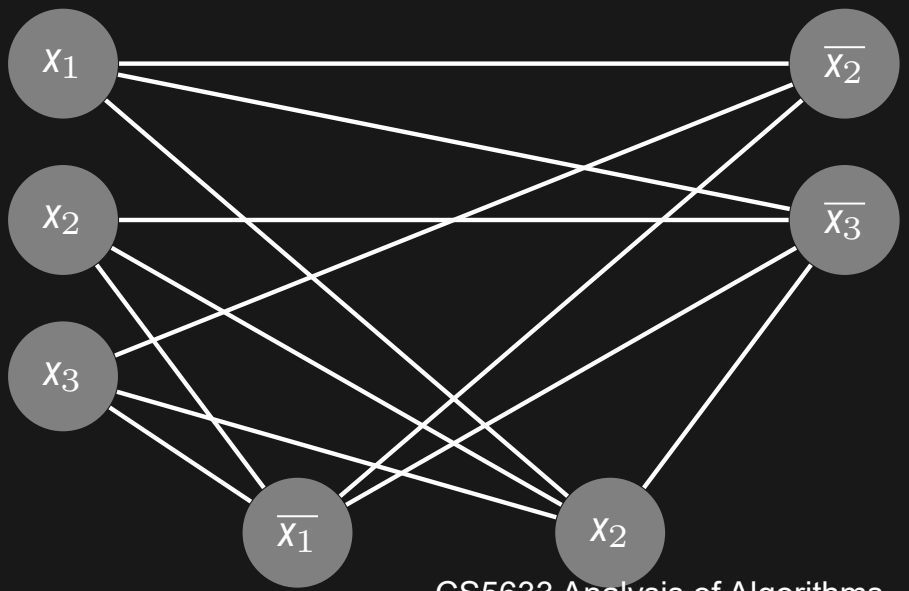
- Recall that the Clique problem asks if there is a clique of size at least k in an input graph G, where a clique is a subset of a graph such that all pairs of vertices in the subset have an edge connecting them.
- Next, we will show that SAT $\leq_p$ Clique. This will imply that Clique is NP-complete.
- Last time, we showed Clique $\leq_p$ Independent Set. By the transitivity of reductions, we will have that SAT $\leq_p$ Clique also implies that Independent Set is NP-complete.

Proving the Clique Problem is NP-Complete

- To show that SAT $\leq$ Clique, we need to convert a the input to SAT (a formula ϕ) into the input for Clique (a graph $G = (V, E)$ and an integer k) such that ϕ is satisfiable iff G has a clique of size at least k.
- Intuitively, if ϕ is satisfiable then G should have a large clique, and if ϕ is not satisfiable then G should not have a large clique. But we don't know if ϕ is satisfiable or not. Regardless, how can we construct G and choose k so that this property holds?
- Notation:
  - We call x_i or $\overline{x_i}$ a literal.
  - A clause is several literals connected with $\vee$, such as $(x_1 \vee x_2 \vee \overline{x_3})$.

Proving the Clique Problem is NP-Complete cont.

- Construction of the graph G:
  - For each clause in ϕ, form a strip of vertices.
  - Within each clause, for each literal t, create a vertex v_t.
  - Add the edge $\{v_t, v_{t'}\}$ iff:
    - t and t' are not in the same clause, and
    - t is not the negation of t'.
- For example, the formula, $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$, maps to graph:



satisfied when

$x_1 = T, x_2 = T, x_3 = F$

Proving the Clique Problem is NP-Complete cont.

- We now prove: a ϕ with k clauses has a satisfying assignment iff G has a clique of size at least k.
  1. Suppose ϕ has a satisfying assignment. Then, there is a literal in each clause that is true. Arbitrarily pick one true literal from each clause, their corresponding vertices in G will a clique of k.
     - 1.1 The number of nodes selected is k because we select one literal from each of the k clauses.
     - 1.2 Each pair of vertices has an edge because the 1) two vertices are not from the same clause; 2) two vertices are both true so they cannot be the negation of each other.

Proving the Clique Problem is NP-Complete cont.

- We now prove: a ϕ with k clauses has a satisfying assignment iff G has a clique of size at least k.
  2. Suppose G has a k-clique, V'. Then by assigning True to each vertices in V', we can get an assignment that satisfies ϕ.
     - 2.1 Each vertex in V' must correspond to one literal of one clause. No two vertices corresponds to the same clause, since vertices from the same clause do no have an edge.
     - 2.2 The assignment is valid as no two True literals are negation to each other.
     - 2.3 Because every clause has a True literal, every clause is True. Consequently, ϕ is True.