Outline

NP-Completeness Theory

- The topics we discussed so far are positive results:

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.
- It studies the problems that cannot be solved efficiently.

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.
- It studies the problems that cannot be solved efficiently.
- Why we study negative results?

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.
- It studies the problems that cannot be solved efficiently.
- Why we study negative results?
- In some sense, the negative results are more important than positive results:

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.
- It studies the problems that cannot be solved efficiently.
- Why we study negative results?
- In some sense, the negative results are more important than positive results:
- The negative result may say that a given problem Q cannot be solved in polynomial time.

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.
- It studies the problems that cannot be solved efficiently.
- Why we study negative results?
- In some sense, the negative results are more important than positive results:
- The negative result may say that a given problem Q cannot be solved in polynomial time.
- Without knowing it, you will have to keep trying to find polynomial time algorithm for solving Q. All your efforts are doomed!

NP-Completeness Theory

- The topics we discussed so far are positive results:
- Given a problem, how to design efficient algorithms for solving it.
- NP-Completeness (NPC for sort) Theory is negative results.
- It studies the problems that cannot be solved efficiently.
- Why we study negative results?
- In some sense, the negative results are more important than positive results:
- The negative result may say that a given problem Q cannot be solved in polynomial time.
- Without knowing it, you will have to keep trying to find polynomial time algorithm for solving Q. All your efforts are doomed!
- NPC Theory tells you when to give up: Don't waste your time on something that is impossible.

What is Computation?

- Computers are powerful, and getting more and more powerful every day.

What is Computation?

- Computers are powerful, and getting more and more powerful every day.
- But there are limitations: There are certain tasks that they cannot do!

What is Computation?

- Computers are powerful, and getting more and more powerful every day.
- But there are limitations: There are certain tasks that they cannot do!
- We have to be more precise: What is Computation?

What is Computation?

- Computers are powerful, and getting more and more powerful every day.
- But there are limitations: There are certain tasks that they cannot do!
- We have to be more precise: What is Computation?

The following quotation is from Electronics Technology and Computer Science, 1940 - 1975: A Coevolution, by Computer Science historian Paul Ceruzzi, published in Annals Hist. Comput. Vol 10, 1989 pp. 257-275:

Quotation

That is the definition of computer science as the study of algorithms - effective procedures - and their implementation by programming languages on digital computer hardware. Implied in this definition is the notion that the algorithm is as fundamental to computing as Newton's Law of Motion to Physics; thus Computer Science is a true science because it is concerned with discovering natural laws about algorithms, ...

What is Computation?

- Computers can only carry out programs. In other words, they can only perform algorithms.

What is Computation?

- Computers can only carry out programs. In other words, they can only perform algorithms.
- Computation is an informal concept. It is generally accepted that Computation = Algorithm.

What is Computation?

- Computers can only carry out programs. In other words, they can only perform algorithms.
- Computation is an informal concept. It is generally accepted that Computation = Algorithm.
- Then what is Algorithm?

What is Computation?

- Computers can only carry out programs. In other words, they can only perform algorithms.
- Computation is an informal concept. It is generally accepted that Computation = Algorithm.
- Then what is Algorithm?

An algorithm for a given problem Q is:

- A sequence of specific and un-ambiguous instructions.
- When the sequence terminates, we get the solution for Q.

What is Computation?

- The first key words are specific and un-ambiguous: we/computers know exactly what should be done next.

What is Computation?

- The first key words are specific and un-ambiguous: we/computers know exactly what should be done next.
- The forms of the instructions are not important. It can be:
  - The addition procedure you learned in the first grade.
  - The machine instructions of a CPU.
  - C++ program.
  - Pseudo-code.
  - High level description (such as Strassen's, Kruskal's algorithms)

What is Computation?

- The first key words are specific and un-ambiguous: we/computers know exactly what should be done next.
- The forms of the instructions are not important. It can be:
  - The addition procedure you learned in the first grade.
  - The machine instructions of a CPU.
  - C++ program.
  - Pseudo-code.
  - High level description (such as Strassen's, Kruskal's algorithms)
- The second key word is terminate. Only when the algorithm stops, we get the answer. If the algorithm doesn't stop, what you do?

What is Computation?

- The first key words are specific and un-ambiguous: we/computers know exactly what should be done next.
- The forms of the instructions are not important. It can be:
  - The addition procedure you learned in the first grade.
  - The machine instructions of a CPU.
  - C++ program.
  - Pseudo-code.
  - High level description (such as Strassen's, Kruskal's algorithms)
- The second key word is terminate. Only when the algorithm stops, we get the answer. If the algorithm doesn't stop, what you do?
- If an "algorithm" is not guaranteed to stop, it is not an algorithm at all.

Outline

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Halting Problem

Input: A program P and an input I for P.
Output: yes if P terminates on I; no if P does not terminate on I.

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Halting Problem

Input: A program P and an input I for P.
Output: yes if P terminates on I; no if P does not terminate on I.

- Intuitively, the only algorithm to solve this general problem is to simulate the execution of P on I.

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Halting Problem

Input: A program P and an input I for P.
Output: yes if P terminates on I; no if P does not terminate on I.

- Intuitively, the only algorithm to solve this general problem is to simulate the execution of P on I.
- If and when the simulation stops, output yes.

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Halting Problem

Input: A program P and an input I for P.
Output: yes if P terminates on I; no if P does not terminate on I.

- Intuitively, the only algorithm to solve this general problem is to simulate the execution of P on I.
- If and when the simulation stops, output yes.
- What to do if the simulation is still running after one day? Wait for one more year, 100 centuries?

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Halting Problem

Input: A program P and an input I for P.
Output: yes if P terminates on I; no if P does not terminate on I.

- Intuitively, the only algorithm to solve this general problem is to simulate the execution of P on I.
- If and when the simulation stops, output yes.
- What to do if the simulation is still running after one day? Wait for one more year, 100 centuries?
- Or shut down the simulation? But then what to output? yes or no?

Limitation of Computation

Absolute limitation for computation: Some problems cannot be solved by an algorithm.

Halting Problem

Input: A program P and an input I for P.
Output: yes if P terminates on I; no if P does not terminate on I.

- Intuitively, the only algorithm to solve this general problem is to simulate the execution of P on I.
- If and when the simulation stops, output yes.
- What to do if the simulation is still running after one day? Wait for one more year, 100 centuries?
- Or shut down the simulation? But then what to output? yes or no?
- There is no guarantee the procedure will stop. This is not an algorithm!

Turing Theorem

Turing Theorem (1936)

There exist no algorithms for solving the Halting Problem.

Turing Theorem

Turing Theorem (1936)

There exist no algorithms for solving the Halting Problem.

This Theorem is a topic in CSE596.

Practical Limitations

- For some problems, the algorithms for solving them exist.

Practical Limitations

Practical Limitations

- For some problems, the algorithms for solving them exist.
- But it takes too much time that they are not practically solvable.

Practical Limitations

- For some problems, the algorithms for solving them exist.
- But it takes too much time that they are not practically solvable.
- Roughly speaking, practically solvable means in polynomial time:

$$
\begin{aligned}
\mathcal{P} &= \text{the set of problems that can be solved in polynomial time} \\
&= \text{the set of problems that can be solved in } O(n^k) \text{ time for some } k.
\end{aligned}
$$

Practical Limitations

- For some problems, the algorithms for solving them exist.
- But it takes too much time that they are not practically solvable.
- Roughly speaking, practically solvable means in polynomial time:

$$\begin{aligned} \mathcal{P} &= \text{the set of problems that can be solved in polynomial time} \\ &= \text{the set of problems that can be solved in } O(n^k) \text{ time for some } k. \end{aligned}$$

- If a problem is not in $\mathcal{P}$, it is practically unsolvable.

Practical Limitations

- For some problems, the algorithms for solving them exist.
- But it takes too much time that they are not practically solvable.
- Roughly speaking, practically solvable means in polynomial time:

$$\begin{aligned}\mathcal{P} &= \text{the set of problems that can be solved in polynomial time} \\ &= \text{the set of problems that can be solved in } O(n^k) \text{ time for some } k.\end{aligned}$$

- If a problem is not in $\mathcal{P}$, it is practically unsolvable.

Why do we define practically solvable as $\mathcal{P}$? (An algorithm with runtime $\Theta(n^{100})$ is not really a practical algorithm.)

Limitation of Computation

- The vast majority of the problems in $\mathcal{P}$ have run time $O(n^k)$ for small k. We rarely see algorithms with $k \geq 5$ or 6.

Limitation of Computation

- The vast majority of the problems in $\mathcal{P}$ have run time $O(n^k)$ for small k. We rarely see algorithms with $k \geq 5$ or 6.
- The barrier between $\Theta(n^k)$ and $\Theta(n^{k-1})$ is relatively low: Once we have a $\Theta(n^{10})$ time algorithm, it may not be very hard to reduce it to say $\Theta(n^8)$.

Limitation of Computation

- The vast majority of the problems in $\mathcal{P}$ have run time $O(n^k)$ for small k. We rarely see algorithms with $k \geq 5$ or 6.
- The barrier between $\Theta(n^k)$ and $\Theta(n^{k-1})$ is relatively low: Once we have a $\Theta(n^{10})$ time algorithm, it may not be very hard to reduce it to say $\Theta(n^8)$.
- On the other hand, the barrier between $\Theta(2^n)$ and $\Theta(n^k)$ is very high: If you can reduce the algorithm runtime of a well known problem from $\Theta(2^n)$ to $\Theta(n^k)$, it would be a major achievement. (For some famous problems, that would earn you a Turing Award!)

Limitation of Computation

- The vast majority of the problems in $\mathcal{P}$ have run time $O(n^k)$ for small k. We rarely see algorithms with $k \geq 5$ or 6.
- The barrier between $\Theta(n^k)$ and $\Theta(n^{k-1})$ is relatively low: Once we have a $\Theta(n^{10})$ time algorithm, it may not be very hard to reduce it to say $\Theta(n^8)$.
- On the other hand, the barrier between $\Theta(2^n)$ and $\Theta(n^k)$ is very high: If you can reduce the algorithm runtime of a well known problem from $\Theta(2^n)$ to $\Theta(n^k)$, it would be a major achievement. (For some famous problems, that would earn you a Turing Award!)
- The definition of $\mathcal{P}$ is largely independent from the computation models. (We will see this later.)

Outline

Examples

- Some problems can be easily solved in polynomial time.

Examples

- Some problems can be easily solved in polynomial time.
- But for similar looking problems, no polynomial time algorithms can be found no matter how hard we try.

Examples

- Some problems can be easily solved in polynomial time.
- But for similar looking problems, no polynomial time algorithms can be found no matter how hard we try.
- We want to identify the properties that make this distinction.

Examples

- Some problems can be easily solved in polynomial time.
- But for similar looking problems, no polynomial time algorithms can be found no matter how hard we try.
- We want to identify the properties that make this distinction.
- If we see a problem Q demonstrates these properties, we would know Q is hard to solve in polynomial time. Then we would not waste our time on it.

Knapsack Problem

- Fractional Knapsack problem can be easily solved in $O(n \log n)$ time by a greedy algorithm.

Knapsack Problem

Knapsack Problem

- Fractional Knapsack problem can be easily solved in $O(n \log n)$ time by a greedy algorithm.
- 0/1 Knapsack problem: No real polynomial time algorithm is known.

Knapsack Problem

- Fractional Knapsack problem can be easily solved in $O(n \log n)$ time by a greedy algorithm.
- 0/1 Knapsack problem: No real polynomial time algorithm is known.

Note 1: The dynamic programming algorithm for the 0/1 knapsack problem takes $\Theta(nK)$ time. It is not really polynomial time, because the runtime depends on the value of K. If K is an n-bit integer, its value is 2^n.

Knapsack Problem

Knapsack Problem

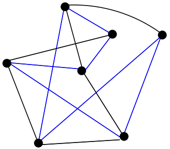
- Fractional Knapsack problem can be easily solved in $O(n \log n)$ time by a greedy algorithm.
- 0/1 Knapsack problem: No real polynomial time algorithm is known.

Note 1: The dynamic programming algorithm for the 0/1 knapsack problem takes $\Theta(nK)$ time. It is not really polynomial time, because the runtime depends on the value of K. If K is an n-bit integer, its value is 2^n.

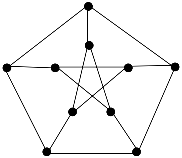
Note 2: The two problems look very similar. Why one is so much harder than the other?

Hamiltonian Cycle

Let G be an undirected graph. A Hamiltonian Cycle (HC) of G is a cycle C in G that passes each vertex of G exactly once.



(a)



(b)

- The blue edges in graph (a) is a HC of G.
- The graph (b) is called the Petersen Graph. It has no HC. (How to show this? It is not easy!)

HC Problem

Input: An undirected graph G.
output: “yes” if G has a HC. “no” if G has no HC.

Hamiltonian Cycle

HC Problem

Input: An undirected graph G.
output: "yes" if G has a HC. "no" if G has no HC.

There is no known polynomial time algorithm for solving this problem.

Euler Tour and Trail

Definition

Let $G = (V, E)$ be an undirected graph.

- A trail of G is a sequence of vertices $W = \langle v_0, v_1, \ldots, v_k \rangle$ such that $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. (W may contain repeated vertices.)
- A trail $W = \langle v_0, v_1, \ldots, v_k \rangle$ of G is called a tour if $v_0 = v_k$.

Euler Tour and Trail

Definition

Let $G = (V, E)$ be an undirected graph.

- A trail of G is a sequence of vertices $W = \langle v_0, v_1, \ldots, v_k \rangle$ such that $(v_{i-1}, v_i) \in E$ for $1 \le i \le k$. (W may contain repeated vertices.)
- A trail $W = \langle v_0, v_1, \ldots, v_k \rangle$ of G is called a tour if $v_0 = v_k$.

- The difference between a path and a trail: a path has no repeated vertices; a trail may have repeated vertices.

Euler Tour and Trail

Definition

Let $G = (V, E)$ be an undirected graph.

- A trail of G is a sequence of vertices $W = \langle v_0, v_1, \ldots, v_k \rangle$ such that $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. (W may contain repeated vertices.)
- A trail $W = \langle v_0, v_1, \ldots, v_k \rangle$ of G is called a tour if $v_0 = v_k$.

- The difference between a path and a trail: a path has no repeated vertices; a trail may have repeated vertices.
- The difference between a cycle and a tour: a cycle has no repeated vertices; a tour may have repeated vertices.

Euler Tour and Trail

Euler Tour and Trail

Let G be an undirected graph.

- An Euler Trail of G is a trail in G that passes each edge of G exactly once.
- An Euler Tour of G is a tour in G that passes each edge of G exactly once.

Euler Tour and Trail

Euler Tour and Trail

Let G be an undirected graph.

- An Euler Trail of G is a trail in G that passes each edge of G exactly once.
- An Euler Tour of G is a tour in G that passes each edge of G exactly once.

Euler Tour Problem

Input: An undirected graph G.
Output: “yes” if G has an Euler Tour; “no” if G has not.

Euler Tour and Trail

Euler Tour and Trail

Let G be an undirected graph.

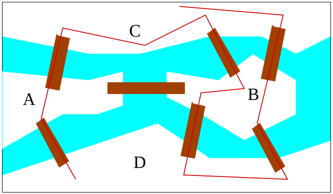
- An Euler Trail of G is a trail in G that passes each edge of G exactly once.
- An Euler Tour of G is a tour in G that passes each edge of G exactly once.

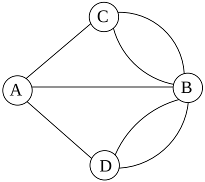
Euler Tour Problem

Input: An undirected graph G.
Output: “yes” if G has an Euler Tour; “no” if G has not.

Euler Trail problem is defined similarly: asking if G has an Euler trail or not.

Euler Tour and Trail





Historical Note:

- The city of Königsberg consists of four islands A, B, C, D separated by the river Pregel, and 7 bridges connecting them.
- Question: Can one take a city walk, crossing each bridge exactly once (without repeating) and come back to where one started?
- The puzzle was circling among Königsberg's high society for long time.
- Euler solved the problem by a simple theorem.

Euler Tour and Trail

Euler Theorem (1736)

Let G be a connected undirected graph.

1. G has an Euler tour iff every vertex of G has even degree.
2. G has an Euler trail iff every vertex of G has even degree, except two vertices.

Euler Tour and Trail

Euler Theorem (1736)

Let G be a connected undirected graph.

1. G has an Euler tour iff every vertex of G has even degree.
2. G has an Euler trail iff every vertex of G has even degree, except two vertices.

By this Theorem, the map of 7 bridges has no Euler trail, nor Euler tour.

Euler Tour and Trail

Euler Theorem (1736)

Let G be a connected undirected graph.

1. G has an Euler tour iff every vertex of G has even degree.
2. G has an Euler trail iff every vertex of G has even degree, except two vertices.

By this Theorem, the map of 7 bridges has no Euler trail, nor Euler tour.

Proof outline of (1):

- Start at any vertex say v_1.
- Travel the graph, each step using only un-traveled edges.
- Go as far as you can go. Stop when you come to a vertex v_k all of whose incident edges have been traveled.
- Because all vertices have even degrees, v_k must be v_1. So we get a tour of G. Call it T_1.

Euler Tour and Trail

- If T_1 contains all edges of G, then T_1 is an Euler tour and we are done.
- If not, let v_2 be a vertex that still has un-traveled incident edges.
- Start at v_2 and repeat above process. We will get another tour T_2 starting and ending at v_2.
- "Insert" T_2 into T_1 at v_2. If this longer Euler tout contains all edges of G, we are done.
- If not, repeat above process, until all edges of G are included.

Euler Tour and Trail

- If T_1 contains all edges of G, then T_1 is an Euler tour and we are done.
- If not, let v_2 be a vertex that still has un-traveled incident edges.
- Start at v_2 and repeat above process. We will get another tour T_2 starting and ending at v_2.
- "Insert" T_2 into T_1 at v_2. If this longer Euler tout contains all edges of G, we are done.
- If not, repeat above process, until all edges of G are included.

Proof of (2): Suppose that G has exactly two odd-degree vertices x and y.

- Add a new edge (x, y) into G. Call the resulting graph G'.

Euler Tour and Trail

- If T_1 contains all edges of G, then T_1 is an Euler tour and we are done.
- If not, let v_2 be a vertex that still has un-traveled incident edges.
- Start at v_2 and repeat above process. We will get another tour T_2 starting and ending at v_2.
- "Insert" T_2 into T_1 at v_2. If this longer Euler tout contains all edges of G, we are done.
- If not, repeat above process, until all edges of G are included.

Proof of (2): Suppose that G has exactly two odd-degree vertices x and y.

- Add a new edge (x, y) into G. Call the resulting graph G'.
- Now all vertices of G' have even degrees. By (1) we can find an Euler tour T' of G'.

Euler Tour and Trail

- If T_1 contains all edges of G, then T_1 is an Euler tour and we are done.
- If not, let v_2 be a vertex that still has un-traveled incident edges.
- Start at v_2 and repeat above process. We will get another tour T_2 starting and ending at v_2.
- "Insert" T_2 into T_1 at v_2. If this longer Euler tout contains all edges of G, we are done.
- If not, repeat above process, until all edges of G are included.

Proof of (2): Suppose that G has exactly two odd-degree vertices x and y.

- Add a new edge (x, y) into G. Call the resulting graph G'.
- Now all vertices of G' have even degrees. By (1) we can find an Euler tour T' of G'.
- By deleting the dummy edge (x, y) from T', we get an Euler tout T of G starting at x and ending at y.

Euler Tour and Trail

- The conditions in Euler's Theorem can be easily checked in $O(n+m)$ time.

Euler Tour and Trail

- The conditions in Euler's Theorem can be easily checked in $O(n+m)$ time.
- Euler tour and trail can also be easily constructed in $O(n+m)$ time (HW problem).

Euler Tour and Trail

- The conditions in Euler's Theorem can be easily checked in $O(n+m)$ time.
- Euler tour and trail can also be easily constructed in $O(n+m)$ time (HW problem).
- The HC problem and the Euler tour problem look similar enough. Why one is very easy, yet another is so hard?

Maximum Matching (MM) Problem

Maximum Matching (MM) Problem

Let $G = (V, E)$ be an undirected graph.

- A matching of G is a subset $M \subseteq E$ such that no two edges in M share a common end vertex.
- A maximum matching of G is a matching M of G with maximum size.
- MM problem: Given G, find a MM of G.

Maximum Matching (MM) Problem

Maximum Matching (MM) Problem

Let $G = (V, E)$ be an undirected graph.

- A matching of G is a subset $M \subseteq E$ such that no two edges in M share a common end vertex.
- A maximum matching of G is a matching M of G with maximum size.
- MM problem: Given G, find a MM of G.

We mentioned earlier that this problem can be solved in polynomial time.

Maximum Independent Set (MIS) Problem

Maximum Independent Set (MIS) Problem

Let $G = (V, E)$ be an undirected graph.

- An independent set of G is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.
- A maximum independent set of G is an independent set I of G with maximum size.
- MIS problem: Given G, find a MIS of G.

Maximum Independent Set (MIS) Problem

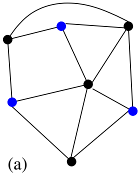
Maximum Independent Set (MIS) Problem

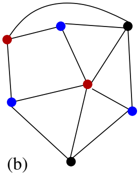
Let $G = (V, E)$ be an undirected graph.

- An independent set of G is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.
- A maximum independent set of G is an independent set I of G with maximum size.
- MIS problem: Given G, find a MIS of G.

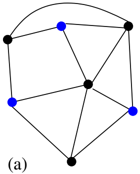
- In a sense, a matching of G is an independent edge set.
- The connection between MIS and the vertex coloring problem: the vertices of $G = (V, E)$ can be colored by k colors iff V can be partitioned into k independent subsets: The vertices with the same color form an independent set of G.

Maximum Independent Set (MIS) Problem


(a)


(b)

Maximum Independent Set (MIS) Problem



- In Fig (a) the blue vertices form an independent set of G.
- in Fig (b), G is colored by three colors. The vertices with the same color form an independent set.
- Although the MIS problem looks very similar to the MM problem, there is no known polynomial time algorithm for solving MIS.

Minimum Spanning Tree (MST) Problem

Minimum Spanning Tree (MST) Problem

Let $G = (V, E)$ be an undirected complete graph. Each edge $e \in E$ has a weight $w(e) \geq 0$.
Find: A spanning tree T of G with minimum total weight $w(T)$.

Minimum Spanning Tree (MST) Problem

Minimum Spanning Tree (MST) Problem

Let $G = (V, E)$ be an undirected complete graph. Each edge $e \in E$ has a weight $w(e) \geq 0$.
Find: A spanning tree T of G with minimum total weight $w(T)$.

- "Complete" means that for any two vertices $u, v \in V$, $(u, v) \in E$.
- In the original definition of MST, we do not required G to be a complete graph.
- The problem defined here is equivalent to the original MST problem:
  - We are given a graph G (not necessarily complete), we want to find a MST of G.
  - Construct a complete graph G_1 by adding dummy edges in to G. The weights of all dummy edges are $+\infty$.
  - Then a MST T_1 of G_1 is also a MST of G. (Because T cannot contain any dummy edges.)

Traveling Salesman Problem (TSP)

Traveling Salesman Problem (TSP)

Input: A complete graph $G = (V, E)$. Each edge $e \in E$ has a weight $w(e) \geq 0$.
Find: A Hamiltonian Cycle C in G with minimum total weight $w(C)$.

Traveling Salesman Problem (TSP)

Traveling Salesman Problem (TSP)

Input: A complete graph $G = (V, E)$. Each edge $e \in E$ has a weight $w(e) \geq 0$.
Find: A Hamiltonian Cycle C in G with minimum total weight $w(C)$.

Application

- A salesman starts from his home city. He must travel each of the n cities once. Then return home.
- $w(u, v)$ is the cost to travel from city u to city v.
- Find the cheapest way to complete his tour.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.
- Despite its simple looking, there is no known polynomial time algorithm for solving it.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.
- Despite its simple looking, there is no known polynomial time algorithm for solving it.
- For MST, we are looking for a spanning tree T of G with minimum $w(T)$.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.
- Despite its simple looking, there is no known polynomial time algorithm for solving it.
- For MST, we are looking for a spanning tree T of G with minimum $w(T)$.
- For TSP, we are looking for a HC C of G with minimum $w(C)$.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.
- Despite its simple looking, there is no known polynomial time algorithm for solving it.
- For MST, we are looking for a spanning tree T of G with minimum $w(T)$.
- For TSP, we are looking for a HC C of G with minimum $w(C)$.
- MST can be easily solved in $O(m \log n)$ time by Kruskal's/Prim's algorithms.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.
- Despite its simple looking, there is no known polynomial time algorithm for solving it.
- For MST, we are looking for a spanning tree T of G with minimum $w(T)$.
- For TSP, we are looking for a HC C of G with minimum $w(C)$.
- MST can be easily solved in $O(m \log n)$ time by Kruskal's/Prim's algorithms.
- Yet, no polynomial time algorithm for TSP.

Traveling Salesman Problem (TSP)

- Since G is complete, any order of the n cities is a HC of G.
- So the number of feasible solutions is $n!$ Hence the brute-force algorithm will take $\Omega(n!)$ time.
- Because its simple definition and easily understood interpretation, TSP is perhaps the best-known graph problem.
- Despite its simple looking, there is no known polynomial time algorithm for solving it.
- For MST, we are looking for a spanning tree T of G with minimum $w(T)$.
- For TSP, we are looking for a HC C of G with minimum $w(C)$.
- MST can be easily solved in $O(m \log n)$ time by Kruskal's/Prim's algorithms.
- Yet, no polynomial time algorithm for TSP.
- MST and TSP look similar. Why their algorithmic properties are so different?

The Goal of NPC Theory

The Goal of NPC Theory

- Try to identify the properties that make a problem computationally intractable (namely, cannot be solved in polynomial time.)

The Goal of NPC Theory

- Try to identify the properties that make a problem computationally intractable (namely, cannot be solved in polynomial time.)
- If a problem Q demonstrates these properties, we will not bother to find polynomial time algorithms for solving it.

The Goal of NPC Theory

The Goal of NPC Theory

- Try to identify the properties that make a problem computationally intractable (namely, cannot be solved in polynomial time.)
- If a problem Q demonstrates these properties, we will not bother to find polynomial time algorithms for solving it.

- We want to identify the problems that are not in P.

The Goal of NPC Theory

The Goal of NPC Theory

- Try to identify the properties that make a problem computationally intractable (namely, cannot be solved in polynomial time.)
- If a problem Q demonstrates these properties, we will not bother to find polynomial time algorithms for solving it.

- We want to identify the problems that are not in P.
- For some problems, this is true for trivial reasons.

Examples

Example:

Input: A set S.
Output: List all subsets of S.

Examples

Example:

Input: A set S.
Output: List all subsets of S.

- There are 2^n subsets of S.
- Just writing them down needs $\Omega(2^n)$ time.
- So trivially, any algorithm for solving it must run in $\Omega(2^n)$ time.
- We are not interested in such trivial reasons.
- So we should rule out these problems.

Outline

Decision Problems

Definition

A problem X is called a decision problem if the output is just 1 bit (yes/no).

Decision Problems

Definition

A problem X is called a decision problem if the output is just 1 bit (yes/no).

- We will concentrate on decision problems. This way, we sure rule out all trivial reasons for exp time.

Decision Problems

Definition

A problem X is called a decision problem if the output is just 1 bit (yes/no).

- We will concentrate on decision problems. This way, we sure rule out all trivial reasons for exp time.
- But by concentrating on decision problems, are we making the theory too narrow, and not applicable to general cases?

Decision Problems

Definition

A problem X is called a decision problem if the output is just 1 bit (yes/no).

- We will concentrate on decision problems. This way, we sure rule out all trivial reasons for exp time.
- But by concentrating on decision problems, are we making the theory too narrow, and not applicable to general cases?
- Fortunately, no. Even though we consider decision problems only, our theory still applies to general cases.

Decision Problems

Definition

A problem X is called a decision problem if the output is just 1 bit (yes/no).

- We will concentrate on decision problems. This way, we sure rule out all trivial reasons for exp time.
- But by concentrating on decision problems, are we making the theory too narrow, and not applicable to general cases?
- Fortunately, no. Even though we consider decision problems only, our theory still applies to general cases.

Fact:

For each optimization problem X, there is a decision version X' of the problem. If we have a polynomial time algorithm for the decision version X', we can solve the original problem X in polynomial time.

Maximum Independent Set (MIS) Problem

Input: A graph $G = (V, E)$.
Find: An independent set I of G with maximum size $|I|$.

Examples

Maximum Independent Set (MIS) Problem

Input: A graph $G = (V, E)$.
Find: An independent set I of G with maximum size $|I|$.

Decision version MIS'

Input: A graph $G = (V, E)$ and an integer $k \leq n$.
Question: Does G have an independent set I such that $|I| \geq k$?

Examples

Maximum Independent Set (MIS) Problem

Input: A graph $G = (V, E)$.
Find: An independent set I of G with maximum size $|I|$.

Decision version MIS'

Input: A graph $G = (V, E)$ and an integer $k \leq n$.
Question: Does G have an independent set I such that $|I| \geq k$?

Suppose we have an algorithm A' for solving MIS'. Then the following algorithm A finds the size of a maximum independent set of G.

Examples

A(G)

1. **for** $k = n$ **downto** 1
2. call $A'(G, k)$
3. **if** $A'(G, k)$ answers "yes" **then output** k and **stop**

Examples

A(G)

1. **for** $k = n$ **downto** 1
2. call $A'(G,k)$
3. **if** $A'(G,k)$ answers "yes" **then output** k and **stop**

- Clearly this algorithm outputs the size of a MIS of G.

Examples

A(G)

1. **for** $k = n$ **downto** 1
2. call $A'(G, k)$
3. **if** $A'(G, k)$ answers "yes" **then output** k and **stop**

- Clearly this algorithm outputs the size of a MIS of G.
- If A' runs in $T(n)$ time then A runs in $nT(n)$ time. So if A' is a poly-time algorithm, so is A.

Examples

A(G)

1. **for** $k = n$ **downto** 1
2. call $A'(G, k)$
3. **if** $A'(G, k)$ answers "yes" **then output** k and **stop**

- Clearly this algorithm outputs the size of a MIS of G.
- If A' runs in $T(n)$ time then A runs in $nT(n)$ time. So if A' is a poly-time algorithm, so is A.
- Once we know the size of the MIS, it's not hard to find the MIS itself. (We have seen this in several dynamic programming alg examples.)

Examples

0/1 Knapsack Problem

Input: n items, each item i has a weight w_i and a profit p_i; and a knapsack with capacity K.
Find: A subset of items with total weight $\leq K$ and maximum total profit.

Examples

0/1 Knapsack Problem

Input: n items, each item i has a weight w_i and a profit p_i; and a knapsack with capacity K.
Find: A subset of items with total weight $\leq K$ and maximum total profit.

Decision version of 0/1 Knapsack Problem

Input: n items, each item i has an integer weight w_i and an integer profit p_i; and a knapsack with integer capacity K. And an integer q.
Question: Is there a subset of items with total weight $\leq K$ and total profit $\geq q$?

Examples

0/1 Knapsack Problem

Input: n items, each item i has a weight w_i and a profit p_i; and a knapsack with capacity K.
Find: A subset of items with total weight $\leq K$ and maximum total profit.

Decision version of 0/1 Knapsack Problem

Input: n items, each item i has an integer weight w_i and an integer profit p_i; and a knapsack with integer capacity K. And an integer q.
Question: Is there a subset of items with total weight $\leq K$ and total profit $\geq q$?

Suppose that we have an algorithm B' for solving the decision version of the 0/1 knapsack problem, with poly runtime $T(n)$. How do we solve the original optimization 0/1 knapsack problem?

Examples

B$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. **for** $q = Q$ **downto** 1
3. call $B'(W[*], P[*], K, q)$
4. **if** $B'(W[*], P[*], K, q)$ answers "yes" **then output** q and **stop**

Examples

B$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. **for** $q = Q$ **downto** 1
3. call $B'(W[*], P[*], K, q)$
4. **if** $B'(W[*], P[*], K, q)$ answers "yes" **then output** q and **stop**

- B correctly computes the maximum profit of the input.

Examples

B$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. **for** $q = Q$ **downto** 1
3. call $B'(W[*], P[*], K, q)$
4. **if** $B'(W[*], P[*], K, q)$ answers "yes" **then output** q and **stop**

- B correctly computes the maximum profit of the input.
- It is not hard to reconstruct the optimal subset.

Examples

B$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. **for** $q = Q$ **downto** 1
3. call $B'(W[*], P[*], K, q)$
4. **if** $B'(W[*], P[*], K, q)$ answers "yes" **then output** q and **stop**

- B correctly computes the maximum profit of the input.
- It is not hard to reconstruct the optimal subset.
- However, the runtime of B is $\Theta(Q \cdot T(n))$. Since it depends on the numerical value of the input integers, this is actually an exponential time algorithm!

Examples

B$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. **for** $q = Q$ **downto** 1
3. call $B'(W[*], P[*], K, q)$
4. **if** $B'(W[*], P[*], K, q)$ answers "yes" **then output** q and **stop**

- B correctly computes the maximum profit of the input.
- It is not hard to reconstruct the optimal subset.
- However, the runtime of B is $\Theta(Q \cdot T(n))$. Since it depends on the numerical value of the input integers, this is actually an exponential time algorithm!
- We can avoid this by using binary search on Q.

Examples

B1$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. $high = Q$; $low = 1$
3. **while** $high > low$ **do:**
4. mid = $\lceil (high + low)/2 \rceil$
5. call $B'(W[*], P[*], K, mid)$
6. **if** $B'(W[*], P[*], K, mid)$ answers "yes" **then** $low = mid + 1$
7. **if** $B'(W[*], P[*], K, mid)$ answers "no" **then** $high = mid - 1$
8. **output** $high$

Examples

B1$(W[*], P[*], K)$

1. Let $Q = \sum_{i=1}^{n} p[i]$
2. $high = Q$; $low = 1$
3. **while** $high > low$ **do:**
4. mid = $\lceil (high + low)/2 \rceil$
5. call $B'(W[*], P[*], K, mid)$
6. **if** $B'(W[*], P[*], K, mid)$ answers "yes" **then** $low = mid + 1$
7. **if** $B'(W[*], P[*], K, mid)$ answers "no" **then** $high = mid - 1$
8. **output** $high$

- The algorithm still computes the optimal profit.
- The runtime is now $O(\log_2 Q \cdot T(n))$. If $T(n)$ is a polynomial in n, so is $\log_2 Q \cdot T(n)$.

Definition of $\mathcal{P}$

Fact

By focusing on decision problems only, we are not losing any generality.

Definition of $\mathcal{P}$

Fact

By focusing on decision problems only, we are not losing any generality.

We now re-define:

Definition

$\mathcal{P}$ = the set of decision problems that have polynomial time algorithms.

Definition of $\mathcal{P}$

Fact

By focusing on decision problems only, we are not losing any generality.

We now re-define:

Definition

$\mathcal{P}$ = the set of decision problems that have polynomial time algorithms.

We want to find the properties of the problems not in $\mathcal{P}$. Suppose we define (warning: this is NOT the correct definition!)

Definition

$\mathcal{NP}$ = the set of decision problems that have NO polynomial time algorithms.

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.
- Some lower bounds are very low degree polynomial. (Sorting needs at least $\Omega(n \log n)$ time.)

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.
- Some lower bounds are very low degree polynomial. (Sorting needs at least $\Omega(n \log n)$ time.)
- Some are based on trivial output size argument: (Matrix Multiplication needs at least $\Omega(n^2)$ time because the output needs at least $\Omega(n^2)$ time to write down.)

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.
- Some lower bounds are very low degree polynomial. (Sorting needs at least $\Omega(n \log n)$ time.)
- Some are based on trivial output size argument: (Matrix Multiplication needs at least $\Omega(n^2)$ time because the output needs at least $\Omega(n^2)$ time to write down.)
- How do we show a single-bit output (decision) problem requires at least $\Omega(n^{100})$ time? $\Omega(n^k)$ for any k?

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.
- Some lower bounds are very low degree polynomial. (Sorting needs at least $\Omega(n \log n)$ time.)
- Some are based on trivial output size argument: (Matrix Multiplication needs at least $\Omega(n^2)$ time because the output needs at least $\Omega(n^2)$ time to write down.)
- How do we show a single-bit output (decision) problem requires at least $\Omega(n^{100})$ time? $\Omega(n^k)$ for any k?
- This is a mission impossible! We don't have a single example.

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.
- Some lower bounds are very low degree polynomial. (Sorting needs at least $\Omega(n \log n)$ time.)
- Some are based on trivial output size argument: (Matrix Multiplication needs at least $\Omega(n^2)$ time because the output needs at least $\Omega(n^2)$ time to write down.)
- How do we show a single-bit output (decision) problem requires at least $\Omega(n^{100})$ time? $\Omega(n^k)$ for any k?
- This is a mission impossible! We don't have a single example.
- With no membership, the $\mathcal{NP}$ class defined this way is useless.

Verification Algorithm

- Then we will not be able to show any problem belongs to this class:
- We have a few examples of lower bound on the complexity of problems.
- Some lower bounds are very low degree polynomial. (Sorting needs at least $\Omega(n \log n)$ time.)
- Some are based on trivial output size argument: (Matrix Multiplication needs at least $\Omega(n^2)$ time because the output needs at least $\Omega(n^2)$ time to write down.)
- How do we show a single-bit output (decision) problem requires at least $\Omega(n^{100})$ time? $\Omega(n^k)$ for any k?
- This is a mission impossible! We don't have a single example.
- With no membership, the $\mathcal{NP}$ class defined this way is useless.
- We have to find a proper definition.

Outline

Verification Algorithm

Certificate

Let Q be a decision problem, and I an instance of Q. We need to decide if I has the required property. (Namely, whether the output on I is yes or no).
A certificate of I is a binary string that "proves" I has the required property.

Verification Algorithm

Certificate

Let Q be a decision problem, and I an instance of Q. We need to decide if I has the required property. (Namely, whether the output on I is yes or no).
A certificate of I is a binary string that "proves" I has the required property.

Hamiltonian Cycle (HC) Problem

- An instance (input) of HC is a graph G.
- The required property: G has a HC.
- A certificate is: A permutation C=$\{ i_1, i_2, \ldots, i_n\}$ of $\{1, 2, \ldots, n\}$ that represents a HC of G. (To be more precise, the certificate is the binary string that describes the permutation.)

0/1 Knapsack Problem

- Instance: n items (each with weight and profit), a capacity K and a target profit t.

Verification Algorithm

0/1 Knapsack Problem

- Instance: n items (each with weight and profit), a capacity K and a target profit t.
- Required property: Is there a subset of items with total weight at most K and total profit at least t?

Verification Algorithm

0/1 Knapsack Problem

- Instance: n items (each with weight and profit), a capacity K and a target profit t.
- Required property: Is there a subset of items with total weight at most K and total profit at least t?
- Certificate: an n-bit vector $\langle x_1, x_2, \ldots, x_n \rangle$ such that:
  - $\sum_{i=1}^{n} x_i \cdot w_i \leq K$ and
  - $\sum_{i=1}^{n} x_i \cdot p_i \geq t$

Verification Algorithm

0/1 Knapsack Problem

- Instance: n items (each with weight and profit), a capacity K and a target profit t.
- Required property: Is there a subset of items with total weight at most K and total profit at least t?
- Certificate: an n-bit vector $\langle x_1, x_2, \ldots, x_n \rangle$ such that:
  - $\sum_{i=1}^{n} x_i \cdot w_i \leq K$ and
  - $\sum_{i=1}^{n} x_i \cdot p_i \geq t$

Verification Algorithm

Let Q be a given decision problem. A verification algorithm for Q is an algorithm that takes two parameters: an input instance I of Q, and a certificate C for I; and outputs "yes".

Verification Algorithm

Definition

$\mathcal{NP}$ = the set of decision problems with certificates of size $O(n^c)$ for some constance c and have polynomial time verification algorithms

Verification Algorithm

Definition

$\mathcal{NP}$ = the set of decision problems with certificates of size $O(n^c)$ for some constance c and have polynomial time verification algorithms

Intuitive meaning of $\mathcal{NP}$: If Q is a problem in $\mathcal{NP}$, then:

- It is easy to prove an input instance I has the required property.

Verification Algorithm

Definition

$\mathcal{NP}$ = the set of decision problems with certificates of size $O(n^c)$ for some constance c and have polynomial time verification algorithms

Intuitive meaning of $\mathcal{NP}$: If Q is a problem in $\mathcal{NP}$, then:

- It is easy to prove an input instance I has the required property.
- It might be much much harder to find the actual structure that has the required property. (But this is not the concern of the verification algorithm.)

Verification Algorithm

Definition

$\mathcal{NP}$ = the set of decision problems with certificates of size $O(n^c)$ for some constance c and have polynomial time verification algorithms

Intuitive meaning of $\mathcal{NP}$: If Q is a problem in $\mathcal{NP}$, then:

- It is easy to prove an input instance I has the required property.
- It might be much much harder to find the actual structure that has the required property. (But this is not the concern of the verification algorithm.)
- It says nothing about the instances I of Q that do not have the required property. (Usually this would be much harder to prove.)

Verification Algorithm: Examples

HC Problem

- Given an instance $G = (V, E)$, you want to convince me that G has a HC.

Verification Algorithm: Examples

HC Problem

- Given an instance $G = (V, E)$, you want to convince me that G has a HC.
- All you need to do is to give me a certificate $C = \langle i_1, i_2, \ldots, i_n \rangle$. (The size of C is $n \log n = O(n^2)$ bits.)

Verification Algorithm: Examples

HC Problem

- Given an instance $G = (V, E)$, you want to convince me that G has a HC.
- All you need to do is to give me a certificate $C = \langle i_1, i_2, \ldots, i_n \rangle$. (The size of C is $n \log n = O(n^2)$ bits.)
- Given this certificate C, I can easily check that C is a HC (namely $(i_t, i_{t+1}) \in E$ for $1 \leq t \leq n$). This checking (verification algorithm) can be done in polynomial time.
- Once the checking is done, I am convinced that G indeed has a HC.

Verification Algorithm: Examples

HC Problem

- Given an instance $G = (V, E)$, you want to convince me that G has a HC.
- All you need to do is to give me a certificate $C = \langle i_1, i_2, \dots, i_n \rangle$. (The size of C is $n \log n = O(n^2)$ bits.)
- Given this certificate C, I can easily check that C is a HC (namely $(i_t, i_{t+1}) \in E$ for $1 \le t \le n$). This checking (verification algorithm) can be done in polynomial time.
- Once the checking is done, I am convinced that G indeed has a HC.
- How to find the certificate C? We don't know and don't care! It's not in the definition.

Verification Algorithm: Examples

0/1 Knapsack Problem

- Given an instance I (n items, capacity K and target profit t), you want to convince me that there is a subset of items with total weight $\leq K$ and total profit $\geq t$.

Verification Algorithm: Examples

0/1 Knapsack Problem

- Given an instance I (n items, capacity K and target profit t), you want to convince me that there is a subset of items with total weight $\leq K$ and total profit $\geq t$.
- All you have to do is to give me a vector $C = \langle x_1, \ldots, x_n \rangle$. ($C$ is a certificate of n bits long.)

Verification Algorithm: Examples

0/1 Knapsack Problem

- Given an instance I (n items, capacity K and target profit t), you want to convince me that there is a subset of items with total weight $\leq K$ and total profit $\geq t$.
- All you have to do is to give me a vector $C = \langle x_1, \ldots, x_n \rangle$. ($C$ is a certificate of n bits long.)
- Given C, I can easily check that the total weight (of the subset C represents) is at most K and the total profit is at least t. (This is the verification algorithm with $O(n)$ runtime.)

Verification Algorithm: Examples

0/1 Knapsack Problem

- Given an instance I (n items, capacity K and target profit t), you want to convince me that there is a subset of items with total weight $\leq K$ and total profit $\geq t$.
- All you have to do is to give me a vector $C = \langle x_1, \ldots, x_n \rangle$. ($C$ is a certificate of n bits long.)
- Given C, I can easily check that the total weight (of the subset C represents) is at most K and the total profit is at least t. (This is the verification algorithm with $O(n)$ runtime.)
- Once the checking is done, I am convinced that the input indeed has the required property.

Verification Algorithm: Examples

0/1 Knapsack Problem

- Given an instance I (n items, capacity K and target profit t), you want to convince me that there is a subset of items with total weight $\leq K$ and total profit $\geq t$.
- All you have to do is to give me a vector $C = \langle x_1, \ldots, x_n \rangle$. ($C$ is a certificate of n bits long.)
- Given C, I can easily check that the total weight (of the subset C represents) is at most K and the total profit is at least t. (This is the verification algorithm with $O(n)$ runtime.)
- Once the checking is done, I am convinced that the input indeed has the required property.
- How to find the vector $C = \langle x_1, \ldots, x_n \rangle$? We don't know and don't care! It's not in the definition.

Verification Algorithm: Examples

0/1 Knapsack Problem

- Given an instance I (n items, capacity K and target profit t), you want to convince me that there is a subset of items with total weight $\leq K$ and total profit $\geq t$.
- All you have to do is to give me a vector $C = \langle x_1, \ldots, x_n \rangle$. ($C$ is a certificate of n bits long.)
- Given C, I can easily check that the total weight (of the subset C represents) is at most K and the total profit is at least t. (This is the verification algorithm with $O(n)$ runtime.)
- Once the checking is done, I am convinced that the input indeed has the required property.
- How to find the vector $C = \langle x_1, \ldots, x_n \rangle$? We don't know and don't care! It's not in the definition.

So HC$\in \mathcal{NP}$ and 0/1 Knapsack$\in \mathcal{NP}$.

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

- How do you convince me G has the required property (i.e G has no HC)?

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

- How do you convince me G has the required property (i.e G has no HC)?
- Say you give me a permutation $C = \langle i_1, \ldots i_n \rangle$ and tell me C is not a HC.

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

- How do you convince me G has the required property (i.e G has no HC)?
- Say you give me a permutation $C = \langle i_1, \ldots i_n \rangle$ and tell me C is not a HC.
- I check. OK, you are right. But so what? May be another permutation of the vertices is a HC?

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

- How do you convince me G has the required property (i.e G has no HC)?
- Say you give me a permutation $C = \langle i_1, \ldots i_n \rangle$ and tell me C is not a HC.
- I check. OK, you are right. But so what? May be another permutation of the vertices is a HC?
- You cannot convince me easily/quickly!

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

- How do you convince me G has the required property (i.e G has no HC)?
- Say you give me a permutation $C = \langle i_1, \ldots i_n \rangle$ and tell me C is not a HC.
- I check. OK, you are right. But so what? May be another permutation of the vertices is a HC?
- You cannot convince me easily/quickly!
- Apparently, there is no polynomial size certificate nor polynomial time verification algorithm for the Non-HC problem.

Verification Algorithm: Examples

Not all decision problems have polynomial size certificates and polynomial time verification algorithms.

Non-HC Problem

Input: A graph $G = (V, E)$
Property: G does not have a HC.

- How do you convince me G has the required property (i.e G has no HC)?
- Say you give me a permutation $C = \langle i_1, \ldots i_n \rangle$ and tell me C is not a HC.
- I check. OK, you are right. But so what? May be another permutation of the vertices is a HC?
- You cannot convince me easily/quickly!
- Apparently, there is no polynomial size certificate nor polynomial time verification algorithm for the Non-HC problem.
- So apparently, this problem is not in $\mathcal{NP}$.

Outline

Non-Deterministic Algorithm

- Using verification algorithm is a newer way to define $\mathcal{NP}$.

Non-Deterministic Algorithm

- Using verification algorithm is a newer way to define $\mathcal{NP}$.
- The traditional way to define $\mathcal{NP}$ is by using non-deterministic algorithm.

Non-Deterministic Algorithm

- Using verification algorithm is a newer way to define $\mathcal{NP}$.
- The traditional way to define $\mathcal{NP}$ is by using non-deterministic algorithm.
- Although they look quite different, they are equivalent.

Non-Deterministic Algorithm

- Using verification algorithm is a newer way to define $\mathcal{NP}$.
- The traditional way to define $\mathcal{NP}$ is by using non-deterministic algorithm.
- Although they look quite different, they are equivalent.

Definition

- The non-deterministic assignment is: $x \leftarrow 0/1$
- It non-deterministically assigns 0 or 1 to the variable x.
- It is considered a basic instruction and takes 1 unit time.

Non-Deterministic Algorithm

Definition

A non-deterministic algorithm is just an ordinary algorithm except that we allow non-deterministic assignment statement in the algorithm (each counts 1 unit time.)

Non-Deterministic Algorithm

Definition

A non-deterministic algorithm is just an ordinary algorithm except that we allow non-deterministic assignment statement in the algorithm (each counts 1 unit time.)

Definition

Let Q be a decision problem. A non-deterministic algorithm A solves Q if the following is true:

- For any "yes" input instance I of Q, there is a sequence of non-deterministic assignment statements so that A output "yes".
- For any "no" input instance I of Q, there is no sequence of non-deterministic assignment statements so that A output "yes".

Non-Deterministic Algorithm: Examples

Definition

$\mathcal{NP} =$ the set of decision problems that can be solved by non-deterministic algorithms in polynomial time.

"NP" stands for: non-deterministic polynomial time.

Non-Deterministic Algorithm: Examples

Definition

$\mathcal{NP} =$ the set of decision problems that can be solved by non-deterministic algorithms in polynomial time.

"NP" stands for: non-deterministic polynomial time.

Example: 0/1 Knapsack Problem

NP-Knapsack(n items, capacity K and target profit t)

1. **for** $i = 1$ **to** n
2. $\quad x_i \leftarrow 0/1$
3. calculate the total weight W and the total profit T of the subset of the items represented by the vector $\langle x_1, \ldots x_n \rangle$.
4. **if** $W \leq K$ and $T \geq t$ **then output** "yes"
5. $\quad\quad$ **else output** "no"

- Clearly the algorithm takes polynomial time.

Non-Deterministic Algorithm: Examples

- Clearly the algorithm takes polynomial time.
- If the input is a yes instance, then there is a sequence of non-deterministic assignments (in line 2) so that the algorithm outputs yes.

Non-Deterministic Algorithm: Examples

- Clearly the algorithm takes polynomial time.
- If the input is a yes instance, then there is a sequence of non-deterministic assignments (in line 2) so that the algorithm outputs yes.
- If the input is a no instance, then there is no sequence of non-deterministic assignments so that the algorithm outputs yes.
- This algorithm solves the 0/1 Knapsack problem by definition.

Non-Deterministic Algorithm: Examples

- Clearly the algorithm takes polynomial time.
- If the input is a yes instance, then there is a sequence of non-deterministic assignments (in line 2) so that the algorithm outputs yes.
- If the input is a no instance, then there is no sequence of non-deterministic assignments so that the algorithm outputs yes.
- This algorithm solves the 0/1 Knapsack problem by definition.
- The loop (line 1-2) guesses a certificate, then the algorithm verifies the certificate.

Non-Deterministic Algorithm: Examples

Example: HC Problem

NP-HC(G)

1. **for** $i = 1$ **to** n
2. guess an integer x_i $(1 \leq x_i \leq n)$ (using non-deterministic assignment statement $\log_2 n$ times.)
3. check $\langle x_1, x_2, \ldots, x_n \rangle$ is a permutation of $\{1, \ldots n\}$
4. check $\langle x_1, x_2, \ldots, x_n \rangle$ is a HC of G
5. **if** both conditions are true **then output** "yes"
6. **else output** "no"

Non-Deterministic Algorithm: Examples

Example: HC Problem

NP-HC(G)

1. **for** $i = 1$ **to** n
2. guess an integer x_i $(1 \leq x_i \leq n)$ (using non-deterministic assignment statement $\log_2 n$ times.)
3. check $\langle x_1, x_2, \ldots, x_n \rangle$ is a permutation of $\{1, \ldots n\}$
4. check $\langle x_1, x_2, \ldots, x_n \rangle$ is a HC of G
5. **if** both conditions are true **then output** "yes"
6. **else output** "no"

- Again, this algorithm solves the HC problem in polynomial time.
- The loop (lines 1-2) just guesses a certificate.
- Then the algorithm verifies the certificate.

Non-Deterministic Algorithm

Fact:

- The class $\mathcal{NP}$ defined by verification algorithm or by non-deterministic algorithm is the same.
- The two definitions are just the different ways to say the same thing.

Non-Deterministic Algorithm

Fact:

- The class $\mathcal{NP}$ defined by verification algorithm or by non-deterministic algorithm is the same.
- The two definitions are just the different ways to say the same thing.

So we have shown (under either of the two definitions)

- 0/1 Knapsack $\in \mathcal{NP}$.
- HC $\in \mathcal{NP}$.
- Similarly we can show TSP $\in \mathcal{NP}$ ... etc.

Non-Deterministic Algorithm

Fact:

- The class $\mathcal{NP}$ defined by verification algorithm or by non-deterministic algorithm is the same.
- The two definitions are just the different ways to say the same thing.

So we have shown (under either of the two definitions)

- 0/1 Knapsack $\in \mathcal{NP}$.
- HC $\in \mathcal{NP}$.
- Similarly we can show TSP $\in \mathcal{NP}$ ... etc.

Fact

By definition we have: $\mathcal{P} \subseteq \mathcal{NP}$.

Non-Deterministic Algorithm

Fact:

- The class $\mathcal{NP}$ defined by verification algorithm or by non-deterministic algorithm is the same.
- The two definitions are just the different ways to say the same thing.

So we have shown (under either of the two definitions)

- 0/1 Knapsack $\in \mathcal{NP}$.
- HC $\in \mathcal{NP}$.
- Similarly we can show TSP $\in \mathcal{NP}$ ... etc.

Fact

By definition we have: $\mathcal{P} \subseteq \mathcal{NP}$.

This is because an ordinary algorithm is just a non-deterministic algorithm in which we do not use the non-deterministic assignment statement.

Big Question

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

Big Question

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

It should not:

- We know HC $\in \mathcal{NP}$ and 0/1 Knapsack $\in \mathcal{NP}$.

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

It should not:

- We know HC $\in \mathcal{NP}$ and 0/1 Knapsack $\in \mathcal{NP}$.
- No polynomial time algorithms for solving them are known.

Big Question

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

It should not:

- We know HC $\in \mathcal{NP}$ and 0/1 Knapsack $\in \mathcal{NP}$.
- No polynomial time algorithms for solving them are known.
- If $\mathcal{P} = \mathcal{NP}$, then we should have polynomial time algorithms for solving them. This is highly unlikely.

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

It should not:

- We know HC $\in \mathcal{NP}$ and 0/1 Knapsack $\in \mathcal{NP}$.
- No polynomial time algorithms for solving them are known.
- If $\mathcal{P} = \mathcal{NP}$, then we should have polynomial time algorithms for solving them. This is highly unlikely.
- So we strongly believe $\mathcal{P} \neq \mathcal{NP}$.

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

It should not:

- We know HC $\in \mathcal{NP}$ and 0/1 Knapsack $\in \mathcal{NP}$.
- No polynomial time algorithms for solving them are known.
- If $\mathcal{P} = \mathcal{NP}$, then we should have polynomial time algorithms for solving them. This is highly unlikely.
- So we strongly believe $\mathcal{P} \neq \mathcal{NP}$.
- To show this, we only need to find one problem $Q \in \mathcal{NP}$ but $Q \notin \mathcal{P}$. (Remember that we tried to find a problem Q not in $\mathcal{P}$ but unable to do?)

Big Question

$$\mathcal{P} = \mathcal{NP}?$$

It should not:

- We know HC $\in \mathcal{NP}$ and 0/1 Knapsack $\in \mathcal{NP}$.
- No polynomial time algorithms for solving them are known.
- If $\mathcal{P} = \mathcal{NP}$, then we should have polynomial time algorithms for solving them. This is highly unlikely.
- So we strongly believe $\mathcal{P} \neq \mathcal{NP}$.
- To show this, we only need to find one problem $Q \in \mathcal{NP}$ but $Q \notin \mathcal{P}$. (Remember that we tried to find a problem Q not in $\mathcal{P}$ but unable to do?)
- To find such a problem Q, we should look at the hardest problems in $\mathcal{NP}$.

Polynomial Time Reduction

We now define the hardest problems in $\mathcal{NP}$.

Polynomial Time Reduction

We now define the hardest problems in $\mathcal{NP}$.

Polynomial Time Reduction

Let P and Q be two decision problems. We say "P is polynomial time reducible to Q (written as $P \leq_{\mathcal{P}} Q$)" if there is an algorithm A such that:

- Given any instance I of P, with input I, the output of A (written as $I' = A(I)$) is an instance I' of Q.
- I is a "yes" instance of P if and only if $I' = A(I)$ is a "yes" instance of Q.
- A runs in polynomial time in n (n is the size of I).

Polynomial Time Reduction

We now define the hardest problems in $\mathcal{NP}$.

Polynomial Time Reduction

Let P and Q be two decision problems. We say "P is polynomial time reducible to Q (written as $P \leq_{\mathcal{P}} Q$)" if there is an algorithm A such that:

- Given any instance I of P, with input I, the output of A (written as $I' = A(I)$) is an instance I' of Q.
- I is a "yes" instance of P if and only if $I' = A(I)$ is a "yes" instance of Q.
- A runs in polynomial time in n (n is the size of I).

Intuitive meaning: If $P \leq_{\mathcal{P}} Q$, then Q is harder than P.

Polynomial Time Reduction: Example

HC Problem

Input: $G = (V, E)$.
Question: Does G have a HC?

Polynomial Time Reduction: Example

HC Problem

Input: $G = (V, E)$.
Question: Does G have a HC?

TSP Problem

Input: A complete graph $H = (V, E)$, an edge weight function $w(*)$ and a target t.
Question: Does H have a HC C with total weight $w(C) \leq t$?

HC Problem

Input: $G = (V, E)$.
Question: Does G have a HC?

TSP Problem

Input: A complete graph $H = (V, E)$, an edge weight function $w(*)$ and a target t.
Question: Does H have a HC C with total weight $w(C) \leq t$?

We will show HC $\leq_{\mathcal{P}}$ TSP. We do this by describing an algorithm A with the required properties.

Polynomial Time Reduction: Example

Given an input $G = (V, E)$ of HC, A needs to construct an instance of the TSP problem. This is done as follows.

A$(G = (V, E))$

1. Construct a complete graph $H = (V, E_H)$. (Namely the vertex set of H is the same as the vertex set of G. H is obtained from G by adding dummy edges to make it complete.)
2. For each edge e in G, let $w(e) = 1$. For each dummy edge e' in H but not in G, let $w(e') = 2$.
3. Let the target $t = n$ (n is the number of vertices in G.)

Polynomial Time Reduction: Example

Given an input $G = (V, E)$ of HC, A needs to construct an instance of the TSP problem. This is done as follows.

A$(G = (V, E))$

1. Construct a complete graph $H = (V, E_H)$. (Namely the vertex set of H is the same as the vertex set of G. H is obtained from G by adding dummy edges to make it complete.)
2. For each edge e in G, let $w(e) = 1$. For each dummy edge e' in H but not in G, let $w(e') = 2$.
3. Let the target $t = n$ (n is the number of vertices in G.)

- The output of A is: $H = (V, E_H)$, a weight function $w(*)$ and a target $t = n$. This is an instance of TSP.

Polynomial Time Reduction: Example

Given an input $G = (V, E)$ of HC, A needs to construct an instance of the TSP problem. This is done as follows.

A$(G = (V, E))$

1. Construct a complete graph $H = (V, E_H)$. (Namely the vertex set of H is the same as the vertex set of G. H is obtained from G by adding dummy edges to make it complete.)
2. For each edge e in G, let $w(e) = 1$. For each dummy edge e' in H but not in G, let $w(e') = 2$.
3. Let the target $t = n$ (n is the number of vertices in G.)

- The output of A is: $H = (V, E_H)$, a weight function $w(*)$ and a target $t = n$. This is an instance of TSP.
- Clearly, A takes polynomial time in n.

Polynomial Time Reduction: Example

Given an input $G = (V, E)$ of HC, A needs to construct an instance of the TSP problem. This is done as follows.

A$(G = (V, E))$

1. Construct a complete graph $H = (V, E_H)$. (Namely the vertex set of H is the same as the vertex set of G. H is obtained from G by adding dummy edges to make it complete.)
2. For each edge e in G, let $w(e) = 1$. For each dummy edge e' in H but not in G, let $w(e') = 2$.
3. Let the target $t = n$ (n is the number of vertices in G.)

- The output of A is: $H = (V, E_H)$, a weight function $w(*)$ and a target $t = n$. This is an instance of TSP.
- Clearly, A takes polynomial time in n.
- The only thing remains to show: G is a yes instance of HC iff $\langle H, w(*), n\rangle$ is a yes instance of TSP.

Polynomial Time Reduction: Example

Suppose G is a yes instance of HC.

- G has a HC C.
- C is also a HC of H.
- The weight of C is $w(C) = n$ (because C contains n edges, all of them are edges in G and have weight 1.)
- So $\langle H, w(*), n \rangle$ is a yes instance of TSP.

Suppose G is a no instance of HC.

- G has no HC.
- Any HC of H contains at least one dummy edge.
- Any HC of H has weight at least $n + 1$ (the best case: $n - 1$ edge from G each with weight 1, and one dummy edge with weight 2).
- So $\langle H, w(*), n \rangle$ is a no instance of TSP.

Polynomial Time Reduction: Example

Suppose G is a yes instance of HC.

- G has a HC C.
- C is also a HC of H.
- The weight of C is $w(C) = n$ (because C contains n edges, all of them are edges in G and have weight 1.)
- So $\langle H, w(*), n\rangle$ is a yes instance of TSP.

Suppose G is a no instance of HC.

- G has no HC.
- Any HC of H contains at least one dummy edge.
- Any HC of H has weight at least $n+1$ (the best case: $n-1$ edge from G each with weight 1, and one dummy edge with weight 2).
- So $\langle H, w(*), n\rangle$ is a no instance of TSP.

This completes the proof of HC $\leq_{\mathcal{P}}$ TSP.

Lemma 34.3

If $P \leq_{\mathcal{P}} Q$ and $Q \in \mathcal{P}$, then $P \in \mathcal{P}$.

Proof.

Since $P \leq_{\mathcal{P}} Q$, we have a poly-time algorithm A that reduces P to Q.
Since $Q \in \mathcal{P}$, we have a poly-time algorithm B that solves Q.
The following algorithm C solves P:
C(I)

1. Call A on I to construct an instance $I' = A(I)$ of Q.
2. Call B on I'.
3. Output "yes" if $B(I')$ outputs "yes".
4. Output "no" if $B(I')$ outputs "no".

□

Polynomial Time Reduction

Proof (continued):
Algorithm C correctly solves P:

- I is a yes instance of P iff $I' = A(I)$ is a yes instance of Q.
- Iff $B(I')$ outputs yes (because B correctly solves Q.)
- So C outputs yes on I iff I is a yes instance of P.

We need to show the run time of C is polynomial.

- Suppose $n = |I|$ is the size of the input I for C.
- Since A is polynomial time, $A(I)$ runs $O(n^k)$ time for some constant k.
- The length of the output $I' = A(I)$ is at most $O(n^k)$ (even if A uses all its runtime to write the output, it can write $O(n^k)$ bits at most.)
- Since B is poly-time algorithm, it runs in $O(N^l)$ time for some l (the input size is N).
- Because the input I' of B has length $O(n^k)$, B will run in $O((n^k)^l)$ time.
- The total runtime of C is $O(n^k + n^{kl})$, which is polynomial in n.

Outline

$\mathcal{NP}$-Hard Problem

A decision problem Q is $\mathcal{NP}$-Hard if for any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}} Q$.

$\mathcal{NP}$-Complete Problems

$\mathcal{NP}$-Hard Problem

A decision problem Q is $\mathcal{NP}$-Hard if for any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}} Q$.

$\mathcal{NP}$-Complete Problem

A decision problem Q is $\mathcal{NP}$-Complete if the following conditions hold:

- $Q \in \mathcal{NP}$.
- Q is $\mathcal{NP}$-hard. (Namely for any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}} Q$).

$\mathcal{NP}$-Complete Problems

$\mathcal{NP}$-Hard Problem

A decision problem Q is $\mathcal{NP}$-Hard if for any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}} Q$.

$\mathcal{NP}$-Complete Problem

A decision problem Q is $\mathcal{NP}$-Complete if the following conditions hold:

- $Q \in \mathcal{NP}$.
- Q is $\mathcal{NP}$-hard. (Namely for any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}} Q$).

Definition

$$\mathcal{NPC} = \text{the set of } \mathcal{NP}\text{-complete problems}$$

Theorem 34.4

If $Q \in \mathcal{NPC}$ and $Q \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

$\mathcal{NP}$-Complete Problems

Theorem 34.4

If $Q \in \mathcal{NPC}$ and $Q \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Intuitive meaning:

- If any $\mathcal{NPC}$ problem Q can be solved in poly-time, then ALL problems in $\mathcal{NP}$ can be solved in poly time.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.

Theorem 34.4

If $Q \in \mathcal{NPC}$ and $Q \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Intuitive meaning:

- If any $\mathcal{NPC}$ problem Q can be solved in poly-time, then ALL problems in $\mathcal{NP}$ can be solved in poly time.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.

Proof:

We already know $\mathcal{P} \subseteq \mathcal{NP}$. Need to show $\mathcal{NP} \subseteq \mathcal{P}$ under the given condition. We need to show for any $P \in \mathcal{NP}$, we have $P \in \mathcal{P}$.

$\mathcal{NP}$-Complete Problems

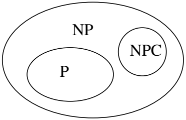
Proof (cont.)

- Pick any $P \in \mathcal{NP}$. Because $Q \in \mathcal{NPC}$, we have $P \leq_{\mathcal{P}} Q$.
- Since $Q \in \mathcal{P}$, Q has a poly time algorithm.
- By Lemma 34.3, $P \in \mathcal{P}$.
- Thus $\mathcal{NP} \subseteq \mathcal{P}$ and we are done.

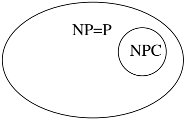
Proof (cont.)

- Pick any $P \in \mathcal{NP}$. Because $Q \in \mathcal{NPC}$, we have $P \leq_{\mathcal{P}} Q$.
- Since $Q \in \mathcal{P}$, Q has a poly time algorithm.
- By Lemma 34.3, $P \in \mathcal{P}$.
- Thus $\mathcal{NP} \subseteq \mathcal{P}$ and we are done.

So there are two possibilities for the relationship between $\mathcal{P}$ and $\mathcal{NP}$:



(1)



(2)

- Case 1: $\mathcal{NPC} \cap \mathcal{P} = \emptyset$.
- Case 2: $\mathcal{NPC} \cap \mathcal{P} \neq \emptyset$.

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.
- We have identified/defined the set $\mathcal{NPC}$ and proved nice theorems. So far so good.

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.
- We have identified/defined the set $\mathcal{NPC}$ and proved nice theorems. So far so good.
- But a critical point is missing: We don't have a single $\mathcal{NPC}$ problem yet!

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.
- We have identified/defined the set $\mathcal{NPC}$ and proved nice theorems. So far so good.
- But a critical point is missing: We don't have a single $\mathcal{NPC}$ problem yet!
- Without a member in $\mathcal{NPC}$, the theory is useless.

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.
- We have identified/defined the set $\mathcal{NPC}$ and proved nice theorems. So far so good.
- But a critical point is missing: We don't have a single $\mathcal{NPC}$ problem yet!
- Without a member in $\mathcal{NPC}$, the theory is useless.
- Saying "the problem Q is in $\mathcal{NPC}$" is a very very very strong statement: How are you going to show for all problems $P \in \mathcal{NP}$ (infinitely many of them), we have $P \leq_{\mathcal{P}} Q$?

$\mathcal{NP}$-Complete Problems

- The case (2) is highly unlikely. But this is not proven.
- $\mathcal{NPC}$ problems are the hardest problems in $\mathcal{NP}$.
- We have identified/defined the set $\mathcal{NPC}$ and proved nice theorems. So far so good.
- But a critical point is missing: We don't have a single $\mathcal{NPC}$ problem yet!
- Without a member in $\mathcal{NPC}$, the theory is useless.
- Saying "the problem Q is in $\mathcal{NPC}$" is a very very very strong statement: How are you going to show for all problems $P \in \mathcal{NP}$ (infinitely many of them), we have $P \leq_{\mathcal{P}} Q$?
- Is there ANY $\mathcal{NPC}$ problem?

- Yes, there is one.

$\mathcal{NP}$-Complete Problems

- Yes, there is one.
- And there are many.

$\mathcal{NP}$-Complete Problems

- Yes, there is one.
- And there are many.

Cook's Theorem (1971)

The Satisfiability (SAT) problem is in $\mathcal{NPC}$.

$\mathcal{NP}$-Complete Problems

- Yes, there is one.
- And there are many.

Cook's Theorem (1971)

The Satisfiability (SAT) problem is in $\mathcal{NPC}$.

Karp's Theorem (1973)

There are 21 other problems in $\mathcal{NPC}$.
(These problems include: HC, TSP, Maximum Independent Set, Maximum Clique, 0/1 Knapsack, Graph Coloring ....)

$\mathcal{NP}$-Complete Problems

- Yes, there is one.
- And there are many.

Cook's Theorem (1971)

The Satisfiability (SAT) problem is in $\mathcal{NPC}$.

Karp's Theorem (1973)

There are 21 other problems in $\mathcal{NPC}$.
(These problems include: HC, TSP, Maximum Independent Set, Maximum Clique, 0/1 Knapsack, Graph Coloring ....)

Levin's Work (1973)

L. A. Levin also formalized the $\mathcal{NPC}$ notion (independent from Cook's work), and provided the $\mathcal{NPC}$ proof of a tiling problem.

$\mathcal{NP}$-Complete Problems

- Since then, more than 3000 natural problems from different fields had been shown to be in $\mathcal{NPC}$.
- Cook received 1982 Turing Award for his work.
- Karp received 1985 Turing Award for his work.

$\mathcal{NPC}$-Theory: Applications

- You work for a software company.
- The boss assigns you a task: Write a program so that user files can be stored in company disks and maximize the company profit.

  Boss: “This little problem looks simple, you should finish within one day.”

$\mathcal{NPC}$-Theory: Applications

- You work for a software company.
- The boss assigns you a task: Write a program so that user files can be stored in company disks and maximize the company profit.

  Boss: “This little problem looks simple, you should finish within one day.”
- Thanks to CSE531, you recognize this is the 0/1 Knapsack problem. It’s not easy. But the boss doesn’t know. You are assigned the task any way.

$\mathcal{NPC}$-Theory: Applications

- You work for a software company.
- The boss assigns you a task: Write a program so that user files can be stored in company disks and maximize the company profit.

  Boss: “This little problem looks simple, you should finish within one day.”
- Thanks to CSE531, you recognize this is the 0/1 Knapsack problem. It’s not easy. But the boss doesn’t know. You are assigned the task any way.
- After two months of hard work and sleepless nights, you still have nothing to show.

$\mathcal{NPC}$-Theory: Applications

- You work for a software company.
- The boss assigns you a task: Write a program so that user files can be stored in company disks and maximize the company profit.

  Boss: “This little problem looks simple, you should finish within one day.”
- Thanks to CSE531, you recognize this is the 0/1 Knapsack problem. It’s not easy. But the boss doesn’t know. You are assigned the task any way.
- After two months of hard work and sleepless nights, you still have nothing to show.
- How do you convince the boss that this is a hard problem and that you should keep your job?

$\mathcal{NPC}$-Theory: Applications

Method 1:

- You say: I worked on this problem for two months. I tried everything possible. But I cannot find a polynomial time algorithm.
- Boss: So what?

Method 1:

- You say: I worked on this problem for two months. I tried everything possible. But I cannot find a polynomial time algorithm.
- Boss: So what?

Method 2:

- You say: I met Prof Karp yesterday. He told me this problem is hard, and there is no polynomial time algorithm.
- Boss: That makes some sense. But this is not a proof.

Method 3:

- You "prove" there is no polynomial time algorithm for 0/1 Knapsack problem. You realize this would earn you a full professorship at MIT. So you quit your job, write a 100+ pages paper and send it to Journal X.
- You write to Editor: I have proved 0/1 Knapsack Problem is not in $\mathcal{P}$. Please consider my paper for publication.
- Editor to you: Dear Author: Thank you for sending us your paper. However, the content of your paper is too advanced for our journal. Please submit it to Journal Y.
- Editor to his colleague: This guy claims he proved $\mathcal{P} \neq \mathcal{NP}$. What a joke!

Method 3:

- You "prove" there is no polynomial time algorithm for 0/1 Knapsack problem. You realize this would earn you a full professorship at MIT. So you quit your job, write a 100+ pages paper and send it to Journal X.
- You write to Editor: I have proved 0/1 Knapsack Problem is not in $\mathcal{P}$. Please consider my paper for publication.
- Editor to you: Dear Author: Thank you for sending us your paper. However, the content of your paper is too advanced for our journal. Please submit it to Journal Y.
- Editor to his colleague: This guy claims he proved $\mathcal{P} \neq \mathcal{NP}$. What a joke!

It is so hard to settle the $\mathcal{NP} = \mathcal{P}?$ question that some researchers believe the current mathematical tools are not powerful enough to settle it one way or another.

Method 4:

You show your problem Q is in $\mathcal{NPC}$.

Method 4:

You show your problem Q is in $\mathcal{NPC}$.

This means:

- Q belongs to the club $\mathcal{NPC}$, which contains many outstanding members.

Method 4:

You show your problem Q is in $\mathcal{NPC}$.

This means:

- Q belongs to the club $\mathcal{NPC}$, which contains many outstanding members.
- Each and every problem in this club has been studied by many researchers.

Method 4:

You show your problem Q is in $\mathcal{NPC}$.

This means:

- Q belongs to the club $\mathcal{NPC}$, which contains many outstanding members.
- Each and every problem in this club has been studied by many researchers.
- But no polynomial time algorithm has been found for any of them.

Method 4:

You show your problem Q is in $\mathcal{NPC}$.

This means:

- Q belongs to the club $\mathcal{NPC}$, which contains many outstanding members.
- Each and every problem in this club has been studied by many researchers.
- But no polynomial time algorithm has been found for any of them.
- Moreover, these problems are computationally equivalent in the sense that if we can find a polynomial time algorithm for ANY of them, then we immediately have polynomial algorithms for solving ALL of them.

Method 4:

You show your problem Q is in $\mathcal{NPC}$.

This means:

- Q belongs to the club $\mathcal{NPC}$, which contains many outstanding members.
- Each and every problem in this club has been studied by many researchers.
- But no polynomial time algorithm has been found for any of them.
- Moreover, these problems are computationally equivalent in the sense that if we can find a polynomial time algorithm for ANY of them, then we immediately have polynomial algorithms for solving ALL of them.
- No, this is still NOT a proof that Q cannot be solved in polynomial time. But this is the strongest evidence you can provide to support your claim.

SAT

Definition

- $x_1, x_2, \ldots, x_n$ are boolean variables. $\bar{x}_k$ is the negation of x_k. x_k and $\bar{x}_k$ are called literals.
- A clause is a boolean formula with the format:

$$C_i = c_{i1} \vee c_{i2} \vee \ldots \vee c_{ij_i}$$

  where $c_{i1}, \ldots c_{ij_i}$ are literals.
- A CNF (conjunctive normal form) formula is a boolean formula of the form:

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

  where each $C_1, C_2, \ldots C_m$ is a clause.

Definition

Let F be a boolean formula with variables $x_1, \ldots, x_n$.

- An assignment assigns 0/1 value to x_i's. (There are 2^n assignments.)

Definition

Let F be a boolean formula with variables $x_1, \ldots, x_n$.

- An assignment assigns 0/1 value to x_i's. (There are 2^n assignments.)
- F is satisfiable if there is an assignment so that F evaluates true.

Definition

Let F be a boolean formula with variables $x_1, \ldots, x_n$.

- An assignment assigns 0/1 value to x_i's. (There are 2^n assignments.)
- F is satisfiable if there is an assignment so that F evaluates true.

Satisfiability (SAT) Problem

Input: A CNF formula F.
Question: Is F satisfiable? (Equivalently: Can we assign 0/1 values to the boolean variables so that $F = 1$ for this assignment?)

SAT: Example

Example

$$F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

- F is a CNF formula. It has 2 clauses and 3 variables.
- Consider the assignment $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$. Then:

$$F = (1 \vee 1 \vee 1) \wedge (0 \vee 0 \vee 0) = 1 \wedge 0 = 0$$

- So this assignment does not satisfy F.
- Consider the assignment $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$. Then:

$$F = (1 \vee 1 \vee 0) \wedge (0 \vee 0 \vee 1) = 1 \wedge 1 = 1$$

- So this assignment satisfies F.
- Thus F is a yes instance of SAT.

Outline

Cook's Theorem

Cook's Theorem: SAT is in $\mathcal{NPC}$.

Cook's Theorem: SAT is in $\mathcal{NPC}$.

To show a problem Q is $\mathcal{NP}$-complete, we need to show two things:

1. Q is in $\mathcal{NP}$.
2. Q is $\mathcal{NP}$-hard.

Cook's Theorem: SAT is in $\mathcal{NPC}$.

To show a problem Q is $\mathcal{NP}$-complete, we need to show two things:

1. Q is in $\mathcal{NP}$.
2. Q is $\mathcal{NP}$-hard.

- To show (1), we can describe either a non-deterministic or a verification poly-time algorithm for solving Q. (They are really the same thing.)

Cook's Theorem: SAT is in $\mathcal{NPC}$.

To show a problem Q is $\mathcal{NP}$-complete, we need to show two things:

1. Q is in $\mathcal{NP}$.
2. Q is $\mathcal{NP}$-hard.

- To show (1), we can describe either a non-deterministic or a verification poly-time algorithm for solving Q. (They are really the same thing.)
- This is fairly easy in most cases.

Cook's Theorem: SAT is in $\mathcal{NPC}$.

To show a problem Q is $\mathcal{NP}$-complete, we need to show two things:

1. Q is in $\mathcal{NP}$.
2. Q is $\mathcal{NP}$-hard.

- To show (1), we can describe either a non-deterministic or a verification poly-time algorithm for solving Q. (They are really the same thing.)
- This is fairly easy in most cases.
- To show (2) is much harder. We will outline the proof.

SAT is in $\mathcal{NP}$

The following non-deterministic algorithm solves SAT in poly-time.

SAT is in $\mathcal{NP}$

The following non-deterministic algorithm solves SAT in poly-time.

NP-SAT(F)
Input: F is a CNF boolean formula with boolean variables $x_1, \ldots, x_n$.

1. **for** $i = 1$ **to** n **do**
2. $\quad x_i \leftarrow 0/1$
3. evaluate F with the values of x_i non-deterministically assigned in (2)
4. **if** F evaluates true **output** yes
5. $\quad$ **else output** no

SAT is in $\mathcal{NP}$

The following non-deterministic algorithm solves SAT in poly-time.

NP-SAT(F)
Input: F is a CNF boolean formula with boolean variables $x_1, \ldots, x_n$.

1. **for** $i = 1$ **to** n **do**
2. $\qquad x_i \leftarrow 0/1$
3. evaluate F with the values of x_i non-deterministically assigned in (2)
4. **if** F evaluates true **output** yes
5. $\qquad$ **else output** no

The lines 1-2 use non-deterministic assignments to guess a certificate $\langle x_1, \ldots x_n \rangle$. Then the algorithm evaluates F and output yes/no according to the value of F.

SAT is $\mathcal{NP}$-hard

SAT is $\mathcal{NP}$-hard (proof outline)

We need to show: For any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}}$ SAT.

SAT is $\mathcal{NP}$-hard

SAT is $\mathcal{NP}$-hard (proof outline)

We need to show: For any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}}$ SAT.

- There are infinitely many problems in $\mathcal{NP}$, how can we prove this?

SAT is $\mathcal{NP}$-hard

SAT is $\mathcal{NP}$-hard (proof outline)

We need to show: For any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}}$ SAT.

- There are infinitely many problems in $\mathcal{NP}$, how can we prove this?
- We need a long detour.

SAT is $\mathcal{NP}$-hard (proof outline)

We need to show: For any $P \in \mathcal{NP}$ we have $P \leq_{\mathcal{P}}$ SAT.

- There are infinitely many problems in $\mathcal{NP}$, how can we prove this?
- We need a long detour.
- We begin by formally define our computation model.

Random Access Machine (RAM)

A RAM consists of:

- A CPU
- A memory, containing memory cells.
- A CPU-memory bus.

SAT is $\mathcal{NP}$-hard

Random Access Machine (RAM)

A RAM consists of:

- A CPU
- A memory, containing memory cells.
- A CPU-memory bus.

In each step, RAM performs a basic instruction, which can be:

- An arithmetic operation (+,-,*,/, etc)
- Branching to another instruction.
- Comparison.
- Read from/write into any memory cell.

SAT is $\mathcal{NP}$-hard

Random Access Machine (RAM)

A RAM consists of:

- A CPU
- A memory, containing memory cells.
- A CPU-memory bus.

In each step, RAM performs a basic instruction, which can be:

- An arithmetic operation (+,-,*,/, etc)
- Branching to another instruction.
- Comparison.
- Read from/write into any memory cell.

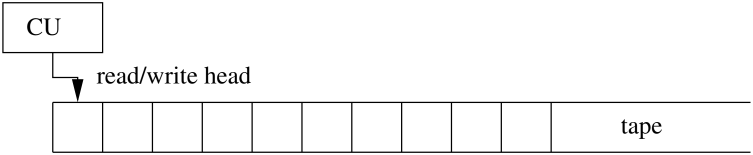
RAM very closely models real computers. It is also the computation model we used throughout this class (implicitly).

Outline

Turing Machine (TM)

A TM consists of:

- A control unit (CU), that can be in any of a fixed number of states.
- A tape divided into cells, numbered by 0, 1, 2, . . .
- A read/write head.



Initial Configuration of a TM:

- The input is written on the tape, staring at cell 0.
- CU is in a special initial state q_0.
- The read/write head is at the cell 0 of the tape.

Turing Machine

Initial Configuration of a TM:

- The input is written on the tape, staring at cell 0.
- CU is in a special initial state q_0.
- The read/write head is at the cell 0 of the tape.

The operation of TM:

- In one step, the TM M does the following, depending on the current state of CU and the content of the cell under the read/write head:
- CU changes to another state.
- The read/write head write some thing in the cell under the head.
- The read/write head moves to left, or right by one cell (or remains at the same location).

Turing Machine

The termination configuration:

- Once the CU enters two special states q_{yes} or q_{no}, the computation stops.

Turing Machine

The termination configuration:

- Once the CU enters two special states q_{yes} or q_{no}, the computation stops.
- If the final CU state is q_{yes}, the input is a yes instance of the problem.

Turing Machine

The termination configuration:

- Once the CU enters two special states q_{yes} or q_{no}, the computation stops.
- If the final CU state is q_{yes}, the input is a yes instance of the problem.
- If the final CU state is q_{no}, the input is a no instance of the problem.

The termination configuration:

- Once the CU enters two special states q_{yes} or q_{no}, the computation stops.
- If the final CU state is q_{yes}, the input is a yes instance of the problem.
- If the final CU state is q_{no}, the input is a no instance of the problem.

Although TM looks very simple with very limited power, it can do everything we can do in computation.

Outline

Church-Turing Thesis

Church-Turing Thesis

Anything that is computable can be done by a TM.

Church-Turing Thesis

Church-Turing Thesis

Anything that is computable can be done by a TM.

- This is NOT a Theorem. It CANNOT be proven.

Church-Turing Thesis

Anything that is computable can be done by a TM.

- This is NOT a Theorem. It CANNOT be proven.
- It is a claim that the informal notion of computable is equivalent to the operation of a very precisely defined device (TM).

Church-Turing Thesis

Church-Turing Thesis

Anything that is computable can be done by a TM.

- This is NOT a Theorem. It CANNOT be proven.
- It is a claim that the informal notion of computable is equivalent to the operation of a very precisely defined device (TM).
- We should convince ourselves that whatever we regard as computable (in common sense) can be done by using a TM.

Church-Turing Thesis

Church-Turing Thesis

Anything that is computable can be done by a TM.

- This is NOT a Theorem. It CANNOT be proven.
- It is a claim that the informal notion of computable is equivalent to the operation of a very precisely defined device (TM).
- We should convince ourselves that whatever we regard as computable (in common sense) can be done by using a TM.
- It is reasonable to show: Whatever that can be done by a RAM can be done by a TM.

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)
- So the CPU of RAM corresponds to the CU of TM.

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)
- So the CPU of RAM corresponds to the CU of TM.
- The memory of RAM corresponds to the tape of the TM.

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)
- So the CPU of RAM corresponds to the CU of TM.
- The memory of RAM corresponds to the tape of the TM.
- Each step of a RAM roughly corresponds to a step of TM.

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)
- So the CPU of RAM corresponds to the CU of TM.
- The memory of RAM corresponds to the tape of the TM.
- Each step of a RAM roughly corresponds to a step of TM.
- The only major difference:
  - For RAM, any memory cell can be accessed in one step.
  - For TM, we can only access the cell that is immediately to the left or right of the current cell.

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)
- So the CPU of RAM corresponds to the CU of TM.
- The memory of RAM corresponds to the tape of the TM.
- Each step of a RAM roughly corresponds to a step of TM.
- The only major difference:
  - For RAM, any memory cell can be accessed in one step.
  - For TM, we can only access the cell that is immediately to the left or right of the current cell.
- However, an algorithm on a RAM with $T(n)$ steps can access at most $T(n)$ memory locations.

Church-Turing Thesis

- The CPU of a RAM is nothing more than a bunch of boolean gates. So CPU can only takes a finite number of states. (The number of states can be 2^{1000}. But that's fine: in the definition of TM, we only require a finite number of states.)
- So the CPU of RAM corresponds to the CU of TM.
- The memory of RAM corresponds to the tape of the TM.
- Each step of a RAM roughly corresponds to a step of TM.
- The only major difference:
  - For RAM, any memory cell can be accessed in one step.
  - For TM, we can only access the cell that is immediately to the left or right of the current cell.
- However, an algorithm on a RAM with $T(n)$ steps can access at most $T(n)$ memory locations.
- So one step of a RAM can be simulated by at most $T(n)$ steps of a TM.

Church-Turing Thesis

Fact

If a problem can be solved in $T(n)$ steps on a RAM, then it can be solved by a TM in at most $(T(n))^2$ steps.

Church-Turing Thesis

Fact

If a problem can be solved in $T(n)$ steps on a RAM, then it can be solved by a TM in at most $(T(n))^2$ steps.

Fact

Whether we define the class $\mathcal{P}$ by RAM or by TM, $\mathcal{P}$ is the same.

Church-Turing Thesis

Fact

If a problem can be solved in $T(n)$ steps on a RAM, then it can be solved by a TM in at most $(T(n))^2$ steps.

Fact

Whether we define the class $\mathcal{P}$ by RAM or by TM, $\mathcal{P}$ is the same.

- This is what we said before: the definition of $\mathcal{P}$ is independent from the model that defines it.

Church-Turing Thesis

Fact

If a problem can be solved in $T(n)$ steps on a RAM, then it can be solved by a TM in at most $(T(n))^2$ steps.

Fact

Whether we define the class $\mathcal{P}$ by RAM or by TM, $\mathcal{P}$ is the same.

- This is what we said before: the definition of $\mathcal{P}$ is independent from the model that defines it.
- Why we use TM as our computation model?

Church-Turing Thesis

Fact

If a problem can be solved in $T(n)$ steps on a RAM, then it can be solved by a TM in at most $(T(n))^2$ steps.

Fact

Whether we define the class $\mathcal{P}$ by RAM or by TM, $\mathcal{P}$ is the same.

- This is what we said before: the definition of $\mathcal{P}$ is independent from the model that defines it.
- Why we use TM as our computation model?
- On one hand, since its operation is very simple, we can argue what a TM can/cannot do.

Church-Turing Thesis

Fact

If a problem can be solved in $T(n)$ steps on a RAM, then it can be solved by a TM in at most $(T(n))^2$ steps.

Fact

Whether we define the class $\mathcal{P}$ by RAM or by TM, $\mathcal{P}$ is the same.

- This is what we said before: the definition of $\mathcal{P}$ is independent from the model that defines it.
- Why we use TM as our computation model?
- On one hand, since its operation is very simple, we can argue what a TM can/cannot do.
- On the other hand, its computation power is the same as any other computation model. So the conclusion we get for TM also applies to other computation models.

Non-Deterministic TM

Non-Deterministic TM

- A non-deterministic step of a TM is a step where the read/write head non-deterministically write 0/1 into a tape cell.

Non-Deterministic TM

Non-Deterministic TM

- A non-deterministic step of a TM is a step where the read/write head non-deterministically write 0/1 into a tape cell.
- A non-deterministic TM is a TM that allows non-deterministic steps.

Non-Deterministic TM

Non-Deterministic TM

- A non-deterministic step of a TM is a step where the read/write head non-deterministically write 0/1 into a tape cell.
- A non-deterministic TM is a TM that allows non-deterministic steps.

The meaning/purpose of non-deterministic TM is the same as the non-deterministic algorithms.

Non-Deterministic TM

Non-Deterministic TM

- A non-deterministic step of a TM is a step where the read/write head non-deterministically write 0/1 into a tape cell.
- A non-deterministic TM is a TM that allows non-deterministic steps.

The meaning/purpose of non-deterministic TM is the same as the non-deterministic algorithms.

Fact

The operation of a non-deterministic algorithm with $T(n)$ steps can be simulated by a non-deterministic TM in $(T(n))^2$ steps.

SAT is $\mathcal{NP}$-hard

We need to show: for ant problem $X \in \mathcal{NP}$, we have $X \leq_{\mathcal{P}}$ SAT.

Outline of the the proof. (It's impossible to mention all details.)

- Pick any problem X in $\mathcal{NP}$.

SAT is $\mathcal{NP}$-hard

We need to show: for ant problem $X \in \mathcal{NP}$, we have $X \leq_{\mathcal{P}}$ SAT.

Outline of the the proof. (It's impossible to mention all details.)

- Pick any problem X in $\mathcal{NP}$.
- This means that we have a non-deterministic algorithm A for solving X with polynomial runtime $T(n)$.

SAT is $\mathcal{NP}$-hard

SAT is $\mathcal{NP}$-hard

We need to show: for ant problem $X \in \mathcal{NP}$, we have $X \leq_{\mathcal{P}}$ SAT.

Outline of the the proof. (It's impossible to mention all details.)

- Pick any problem X in $\mathcal{NP}$.
- This means that we have a non-deterministic algorithm A for solving X with polynomial runtime $T(n)$.
- The operation of A can be simulated by a non-deterministic TM M in $(T(n))^2$ time, still a polynomial in n.

SAT is $\mathcal{NP}$-hard

SAT is $\mathcal{NP}$-hard

We need to show: for ant problem $X \in \mathcal{NP}$, we have $X \leq_{\mathcal{P}}$ SAT.

Outline of the the proof. (It's impossible to mention all details.)

- Pick any problem X in $\mathcal{NP}$.
- This means that we have a non-deterministic algorithm A for solving X with polynomial runtime $T(n)$.
- The operation of A can be simulated by a non-deterministic TM M in $(T(n))^2$ time, still a polynomial in n.
- Since the operation of M is very simple, it can be described by a boolean formula F_1.

SAT is $\mathcal{NP}$-hard

We need to show: for ant problem $X \in \mathcal{NP}$, we have $X \leq_{\mathcal{P}}$ SAT.

Outline of the the proof. (It's impossible to mention all details.)

- Pick any problem X in $\mathcal{NP}$.
- This means that we have a non-deterministic algorithm A for solving X with polynomial runtime $T(n)$.
- The operation of A can be simulated by a non-deterministic TM M in $(T(n))^2$ time, still a polynomial in n.
- Since the operation of M is very simple, it can be described by a boolean formula F_1.
- More precisely, suppose that on the input instance I of X, the non-deterministic steps of M are used to write 0/1 into tape locations $x_1, x_2, \ldots, x_t$, then the operation of M can be fully specified by F_1 where $x_1, \ldots, x_t$ are the only boolean variables.

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance
  - iff there is a sequence of non-deterministic assignments so that algorithm A outputs yes.

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance
  - iff there is a sequence of non-deterministic assignments so that algorithm A outputs yes.
  - iff there is a sequence of non-deterministic steps (that write 0 or 1 into $x_1, \ldots, x_t$) so that the TM M stops at final state q_{yes}.

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance
  - iff there is a sequence of non-deterministic assignments so that algorithm A outputs yes.
  - iff there is a sequence of non-deterministic steps (that write 0 or 1 into $x_1, \ldots, x_t$) so that the TM M stops at final state q_{yes}.
  - iff there is a truth assignment to $x_1, \ldots, x_t$ so that the CNF formula F_2 evaluates true.

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance
  - iff there is a sequence of non-deterministic assignments so that algorithm A outputs yes.
  - iff there is a sequence of non-deterministic steps (that write 0 or 1 into $x_1, \ldots, x_t$) so that the TM M stops at final state q_{yes}.
  - iff there is a truth assignment to $x_1, \ldots, x_t$ so that the CNF formula F_2 evaluates true.
- In other words, I is a yes instance of X iff F_2 is a yes instance of SAT.

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance
  - iff there is a sequence of non-deterministic assignments so that algorithm A outputs yes.
  - iff there is a sequence of non-deterministic steps (that write 0 or 1 into $x_1, \ldots, x_t$) so that the TM M stops at final state q_{yes}.
  - iff there is a truth assignment to $x_1, \ldots, x_t$ so that the CNF formula F_2 evaluates true.
- In other words, I is a yes instance of X iff F_2 is a yes instance of SAT.
- The whole process can be done in polynomial time.

SAT is $\mathcal{NP}$-hard

- F_1 can be converted to an equivalent CNF formula F_2 with boolean variables $x_1, \ldots, x_t$, of polynomial length.
- The construction is such that:
  - Input I of the problem X is a yes instance
  - iff there is a sequence of non-deterministic assignments so that algorithm A outputs yes.
  - iff there is a sequence of non-deterministic steps (that write 0 or 1 into $x_1, \ldots, x_t$) so that the TM M stops at final state q_{yes}.
  - iff there is a truth assignment to $x_1, \ldots, x_t$ so that the CNF formula F_2 evaluates true.
- In other words, I is a yes instance of X iff F_2 is a yes instance of SAT.
- The whole process can be done in polynomial time.
- Hence $X \leq_{\mathcal{P}}$ SAT, as to be shown.

Outline

How to prove a problem is $\mathcal{NP}$-complete?

- It is very hard to find/prove the first $\mathcal{NPC}$ problem.

How to prove a problem is $\mathcal{NP}$-complete?

- It is very hard to find/prove the first $\mathcal{NPC}$ problem.
- Once we have one $\mathcal{NPC}$ problem (SAT), it becomes much easier to show other problems are $\mathcal{NPC}$.

How to prove a problem is $\mathcal{NP}$-complete?

- It is very hard to find/prove the first $\mathcal{NPC}$ problem.
- Once we have one $\mathcal{NPC}$ problem (SAT), it becomes much easier to show other problems are $\mathcal{NPC}$.

Lemma

Let X, Y, Z be three decision problems. If $X \leq_{\mathcal{P}} Y$ and $Y \leq_{\mathcal{P}} Z$, then $X \leq_{\mathcal{P}} Z$.

How to prove a problem is $\mathcal{NP}$-complete?

- It is very hard to find/prove the first $\mathcal{NPC}$ problem.
- Once we have one $\mathcal{NPC}$ problem (SAT), it becomes much easier to show other problems are $\mathcal{NPC}$.

Lemma

Let X, Y, Z be three decision problems. If $X \leq_{\mathcal{P}} Y$ and $Y \leq_{\mathcal{P}} Z$, then $X \leq_{\mathcal{P}} Z$.

Proof: $X \leq_{\mathcal{P}} Y$ means there is an algorithm A:

- A runs in poly-time.
- For any input instance I of X, $J = A(I)$ is an instance of Y.
- I is a yes instance of X iff J is a yes instance of Y.

How to prove a problem is $\mathcal{NP}$-complete?

Similarly, $Y \le_{\mathcal{P}} Z$ means there is an algorithm B:

- B runs in poly-time.
- For any input instance J of Y, $K = B(J)$ is an instance of Z.
- J is a yes instance of X iff J is a yes instance of Z.

How to prove a problem is $\mathcal{NP}$-complete?

Similarly, $Y \leq_{\mathcal{P}} Z$ means there is an algorithm B:

- B runs in poly-time.
- For any input instance J of Y, $K = B(J)$ is an instance of Z.
- J is a yes instance of X iff J is a yes instance of Z.

$C(I)$

1. call $J = A(I)$
2. call $K = B(J)$
3. output K

How to prove a problem is $\mathcal{NP}$-complete?

Similarly, $Y \leq_{\mathcal{P}} Z$ means there is an algorithm B:

- B runs in poly-time.
- For any input instance J of Y, $K = B(J)$ is an instance of Z.
- J is a yes instance of X iff J is a yes instance of Z.

$C(I)$

1. call $J = A(I)$
2. call $K = B(J)$
3. output K

- Given an instance I of X, C outputs an instance K of Z.
- Since both A and B run in poly-time, so is C.
- I is a yes instance of X iff $J = A(I)$ is a yes instance of Y iff $K = B(J) = B(A(I)) = C(I)$ is a yes instance of Z.
- So C is a polynomial time reduction from X to Z.

How to prove a problem is $\mathcal{NP}$-complete?

Lemma 34.8

Let Y and Z be two decision problems. If Y is $\mathcal{NP}$-hard and $Y \leq_{\mathcal{P}} Z$, then Z is $\mathcal{NP}$-hard.

Proof.

- Pick any problem $X \in \mathcal{NP}$.

How to prove a problem is $\mathcal{NP}$-complete?

Lemma 34.8

Let Y and Z be two decision problems. If Y is $\mathcal{NP}$-hard and $Y \leq_{\mathcal{P}} Z$, then Z is $\mathcal{NP}$-hard.

Proof.

- Pick any problem $X \in \mathcal{NP}$.
- Since Y is $\mathcal{NP}$-hard, we have $X \leq_{\mathcal{P}} Y$.

How to prove a problem is $\mathcal{NP}$-complete?

Lemma 34.8

Let Y and Z be two decision problems. If Y is $\mathcal{NP}$-hard and $Y \leq_{\mathcal{P}} Z$, then Z is $\mathcal{NP}$-hard.

Proof.

- Pick any problem $X \in \mathcal{NP}$.
- Since Y is $\mathcal{NP}$-hard, we have $X \leq_{\mathcal{P}} Y$.
- Since $Y \leq_{\mathcal{P}} Z$, we have $X \leq_{\mathcal{P}} Z$ by previous lemma.

$\square$

How to prove a problem is $\mathcal{NP}$-complete?

How to prove a problem Z is $\mathcal{NPC}$?

- Show $Z \in \mathcal{NP}$. We can do this by giving a non-deterministic or verification poly-time algorithm. (This is usually easy.)

How to prove a problem is $\mathcal{NP}$-complete?

How to prove a problem Z is $\mathcal{NPC}$?

- Show $Z \in \mathcal{NP}$. We can do this by giving a non-deterministic or verification poly-time algorithm. (This is usually easy.)
- Pick a $\mathcal{NPC}$ problem Y, and show $Y \leq_{\mathcal{P}} Z$. (By Lemma 34.8, this implies Z is $\mathcal{NP}$-hard).

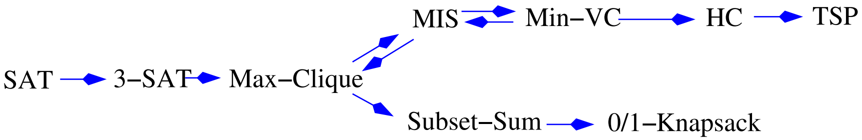
How to prove a problem is $\mathcal{NP}$-complete?

How to prove a problem Z is $\mathcal{NPC}$?

- Show $Z \in \mathcal{NP}$. We can do this by giving a non-deterministic or verification poly-time algorithm. (This is usually easy.)
- Pick a $\mathcal{NPC}$ problem Y, and show $Y \leq_{\mathcal{P}} Z$. (By Lemma 34.8, this implies Z is $\mathcal{NP}$-hard).

Karp's Theorem

He proved the following (among other results), where each $\rightarrow$ is a $\leq_{\mathcal{P}}$.



How to prove a problem is $\mathcal{NP}$-complete?

We will show some of these reductions.

3-SAT

Input: A CNF boolean formula: $F = C_1 \wedge C_2 \cdots C_m$, where each C_i is a clause consisting of EXACTLY 3 literals.
Question: Is F satisfiable?

How to prove a problem is $\mathcal{NP}$-complete?

We will show some of these reductions.

3-SAT

Input: A CNF boolean formula: $F = C_1 \wedge C_2 \cdots C_m$, where each C_i is a clause consisting of EXACTLY 3 literals.
Question: Is F satisfiable?

- This is a special case of SAT.
- It can be shown SAT $\leq_{\mathcal{P}}$ 3-SAT. So 3-SAT is $\mathcal{NP}$-hard.
- The proof needs knowledge in boolean algebra. We omit the proof here. (It's not hard.)

Outline

Max-Clique problem is $\mathcal{NP}$-complete

Max-Clique

Let $G = (V, E)$ be an undirected graph.

- A clique of G is a subset $C \subseteq V$ such that every two vertices in C are adjacent to each other in G.
- A maximum clique of G is a clique C of G with maximum size.

Max-Clique problem is $\mathcal{NP}$-complete

Max-Clique

Let $G = (V, E)$ be an undirected graph.

- A clique of G is a subset $C \subseteq V$ such that every two vertices in C are adjacent to each other in G.
- A maximum clique of G is a clique C of G with maximum size.
- Max-Clique problem: Given G, find a max clique of G,

Max-Clique problem is $\mathcal{NP}$-complete

Max-Clique

Let $G = (V, E)$ be an undirected graph.

- A clique of G is a subset $C \subseteq V$ such that every two vertices in C are adjacent to each other in G.
- A maximum clique of G is a clique C of G with maximum size.
- Max-Clique problem: Given G, find a max clique of G,
- The decision version of Max-Clique: Given G and an integer t, does G have a clique C with size $|C| \geq t$?

Max-Clique problem is $\mathcal{NP}$-complete

Max-Clique

Let $G = (V, E)$ be an undirected graph.

- A clique of G is a subset $C \subseteq V$ such that every two vertices in C are adjacent to each other in G.
- A maximum clique of G is a clique C of G with maximum size.
- Max-Clique problem: Given G, find a max clique of G,
- The decision version of Max-Clique: Given G and an integer t, does G have a clique C with size $|C| \geq t$?

Max-Clique is $\mathcal{NPC}$

We need to show two things:

1. Max-Clique is in $\mathcal{NP}$.
2. Max-Clique is $\mathcal{NP}$-hard.

Max-Clique problem is $\mathcal{NP}$-complete

(1) The following simple non-deterministic algorithm solves this problem.

NP-Max-Clique($G = (V, E), t$)

1. $C = \emptyset$
2. **for** $i = 1$ **to** n **do**
3. $\quad x_i \leftarrow 0/1$
4. $\quad$ **if** $x_i = 1$ put v_i into C
5. check if C is a clique of G or not
6. **if** C is a clique and $|C| \geq t$ **output** yes **else output** no

Max-Clique problem is $\mathcal{NP}$-complete

(1) The following simple non-deterministic algorithm solves this problem.

NP-Max-Clique($G = (V, E), t$)

1. $C = \emptyset$
2. **for** $i = 1$ **to** n **do**
3. $\quad\quad x_i \leftarrow 0/1$
4. $\quad\quad$ **if** $x_i = 1$ put v_i into C
5. check if C is a clique of G or not
6. **if** C is a clique and $|C| \geq t$ **output** yes **else output** no

- First guess a subset C by using non-deterministic assignments.
- Then check if C is a clique and contains at least t vertices. Output yes/no accordingly.
- It solves the Max-Clique problem in poly-time.

Max-Clique problem is $\mathcal{NP}$-complete

(2) We show 3-SAT $\leq_{\mathcal{P}}$ Max-Clique. Since 3-SAT is $\mathcal{NP}$-hard, this implies Max-Clique is $\mathcal{NP}$-hard,

Max-Clique problem is $\mathcal{NP}$-complete

(2) We show 3-SAT $\leq_{\mathcal{P}}$ Max-Clique. Since 3-SAT is $\mathcal{NP}$-hard, this implies Max-Clique is $\mathcal{NP}$-hard,

Given an instance F of 3-SAT:

$$F = C_1 \wedge C_2 \cdots C_k$$

where each $C_i = l_1^i \vee l_2^i \vee l_3^i$ has exactly three literals.
We need to: Construct an instance $\langle G = (V, E), t \rangle$ so that:

- The construction can be done in poly-time.
- F is a yes instance iff $\langle G = (V, E), t \rangle$ is a yes instance.

Max-Clique problem is $\mathcal{NP}$-complete

(2) We show 3-SAT $\leq_{\mathcal{P}}$ Max-Clique. Since 3-SAT is $\mathcal{NP}$-hard, this implies Max-Clique is $\mathcal{NP}$-hard,

Given an instance F of 3-SAT:

$$F = C_1 \wedge C_2 \cdots C_k$$

where each $C_i = l_1^i \vee l_2^i \vee l_3^i$ has exactly three literals.
We need to: Construct an instance $\langle G = (V, E), t \rangle$ so that:

- The construction can be done in poly-time.
- F is a yes instance iff $\langle G = (V, E), t \rangle$ is a yes instance.
- Namely: F is satisfiable iff G has a clique of size at least t.

$G = (V, E)$ is constructed as follows:

Max-Clique problem is $\mathcal{NP}$-complete?

- $V = V_1 \cup V_2 \ldots V_k$ (V_i corresponds to the clause C_i in F).
- $V_i = \{v_1^i, v_2^i, v_3^i\}$ (each vertex in V_i corresponds to a literal in C_i.)
- $(v_s^i, v_t^j) \in E$ if and only if the following hold:
    - $i \neq j$
    - The literals corresponding to v_s^i and v_t^j are not negations of each other.

Max-Clique problem is $\mathcal{NP}$-complete?

- $V = V_1 \cup V_2 \dots V_k$ (V_i corresponds to the clause C_i in F).
- $V_i = \{v_1^i, v_2^i, v_3^i\}$ (each vertex in V_i corresponds to a literal in C_i.)
- $(v_s^i, v_t^j) \in E$ if and only if the following hold:
  - $i \neq j$
  - The literals corresponding to v_s^i and v_t^j are not negations of each other.
- Set $t = k$.

Max-Clique problem is $\mathcal{NP}$-complete?

- $V = V_1 \cup V_2 \ldots V_k$ (V_i corresponds to the clause C_i in F).
- $V_i = \{v_1^i, v_2^i, v_3^i\}$ (each vertex in V_i corresponds to a literal in C_i.)
- $(v_s^i, v_t^j) \in E$ if and only if the following hold:
  - $i \neq j$
  - The literals corresponding to v_s^i and v_t^j are not negations of each other.
- Set $t = k$.
- This completes the construction of the Max-Clique instance.

Max-Clique problem is $\mathcal{NP}$-complete?

- $V = V_1 \cup V_2 \ldots V_k$ (V_i corresponds to the clause C_i in F).
- $V_i = \{v_1^i, v_2^i, v_3^i\}$ (each vertex in V_i corresponds to a literal in C_i.)
- $(v_s^i, v_t^j) \in E$ if and only if the following hold:
    - $i \neq j$
    - The literals corresponding to v_s^i and v_t^j are not negations of each other.
- Set $t = k$.
- This completes the construction of the Max-Clique instance.
- G has $n = 3k$ vertices and at most $n^2/2$ edges.

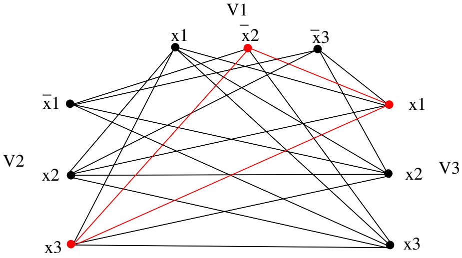
Max-Clique problem is $\mathcal{NP}$-complete?

- $V = V_1 \cup V_2 \dots V_k$ (V_i corresponds to the clause C_i in F).
- $V_i = \{v_1^i, v_2^i, v_3^i\}$ (each vertex in V_i corresponds to a literal in C_i.)
- $(v_s^i, v_t^j) \in E$ if and only if the following hold:
  - $i \neq j$
  - The literals corresponding to v_s^i and v_t^j are not negations of each other.
- Set $t = k$.
- This completes the construction of the Max-Clique instance.
- G has $n = 3k$ vertices and at most $n^2/2$ edges.
- The construction can be easily done by an algorithm in poly-time.

Max-Clique problem is $\mathcal{NP}$-complete?

Example

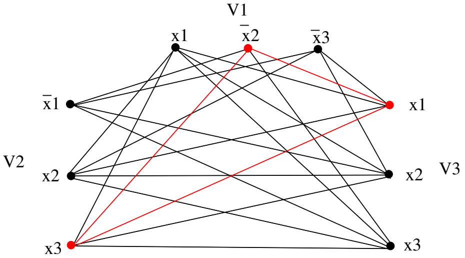
$$F = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



Max-Clique problem is $\mathcal{NP}$-complete?

Example

$$F = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



Note that no two vertices in V_i are adjacent to each other for any V_i. G is a k-partite graph.

It remains to show:

F is satisfiable $\Longleftrightarrow$ G has a clique C of size at least $t = k$.

It remains to show:

F is satisfiable $\Longleftrightarrow$ G has a clique C of size at least $t = k$.

$\Longleftarrow$ Suppose G has a clique C with $|C| \geq k$.

- No two vertices in the same V_i can be in C (they are not adjacent in G.)

It remains to show:

F is satisfiable $\Longleftrightarrow$ G has a clique C of size at least $t = k$.

$\Longleftarrow$ Suppose G has a clique C with $|C| \geq k$.

- No two vertices in the same V_i can be in C (they are not adjacent in G.)
- So C contains exactly k vertices, one vertex from each V_i $(1 \leq i \leq k)$. Say $C = \{v_1, v_2, \ldots, v_k\}$ $(v_i \in V_i$ for $i = 1, 2, \ldots, k)$.

It remains to show:

F is satisfiable $\Longleftrightarrow$ G has a clique C of size at least $t = k$.

$\Longleftarrow$ Suppose G has a clique C with $|C| \geq k$.

- No two vertices in the same V_i can be in C (they are not adjacent in G.)
- So C contains exactly k vertices, one vertex from each V_i $(1 \leq i \leq k)$. Say $C = \{v_1, v_2, \ldots, v_k\}$ $(v_i \in V_i$ for $i = 1, 2, \ldots, k)$.
- For each v_i, assign the corresponding literal the boolean value 1.
- Because C is a clique, any two $v_i, v_j \in C$ are adjacent in G. This implies the corresponding boolean literals are not negation of each other. So this truth assignment is valid.
- For any variable x_i that has not been assigned a value yet, assign a 0/1 value to it arbitrarily.

It remains to show:

F is satisfiable $\Longleftrightarrow$ G has a clique C of size at least $t = k$.

$\Longleftarrow$ Suppose G has a clique C with $|C| \geq k$.

- No two vertices in the same V_i can be in C (they are not adjacent in G.)
- So C contains exactly k vertices, one vertex from each V_i $(1 \leq i \leq k)$. Say $C = \{v_1, v_2, \ldots, v_k\}$ $(v_i \in V_i$ for $i = 1, 2, \ldots, k)$.
- For each v_i, assign the corresponding literal the boolean value 1.
- Because C is a clique, any two $v_i, v_j \in C$ are adjacent in G. This implies the corresponding boolean literals are not negation of each other. So this truth assignment is valid.
- For any variable x_i that has not been assigned a value yet, assign a 0/1 value to it arbitrarily.
- For each $1 \leq i \leq k$, at least one literal in C_i is 1. So C_i evaluates 1.

It remains to show:

F is satisfiable $\iff$ G has a clique C of size at least $t = k$.

$\Longleftarrow$ Suppose G has a clique C with $|C| \geq k$.

- No two vertices in the same V_i can be in C (they are not adjacent in G.)
- So C contains exactly k vertices, one vertex from each V_i $(1 \leq i \leq k)$. Say $C = \{v_1, v_2, \ldots, v_k\}$ $(v_i \in V_i$ for $i = 1, 2, \ldots, k)$.
- For each v_i, assign the corresponding literal the boolean value 1.
- Because C is a clique, any two $v_i, v_j \in C$ are adjacent in G. This implies the corresponding boolean literals are not negation of each other. So this truth assignment is valid.
- For any variable x_i that has not been assigned a value yet, assign a 0/1 value to it arbitrarily.
- For each $1 \leq i \leq k$, at least one literal in C_i is 1. So C_i evaluates 1.
- Since every C_i evaluates 1, $F = C_1 \wedge \cdots \wedge C_k = 1 \wedge 1 \cdots \wedge 1 = 1$. So F is satisfiable.

It remains to show:

It remains to show:

In our example, the red vertices form a clique of size $k = 3$. If we assign $x_1 = 1, \bar{x}_2 = 1$ and $x_3 = 1$, then $F = 1$.

$\Longrightarrow$ Suppose F is satisfiable:

- Consider a 0/1 assignment of boolean variables that makes $F = 1$.

It remains to show:

In our example, the red vertices form a clique of size $k = 3$. If we assign $x_1 = 1, \bar{x}_2 = 1$ and $x_3 = 1$, then $F = 1$.

$\Longrightarrow$ Suppose F is satisfiable:

- Consider a 0/1 assignment of boolean variables that makes $F = 1$.
- As $F = C_1 \wedge \cdots \wedge C_k$, every C_i must be 1.

It remains to show:

In our example, the red vertices form a clique of size $k = 3$. If we assign $x_1 = 1, \bar{x}_2 = 1$ and $x_3 = 1$, then $F = 1$.

$\Longrightarrow$ Suppose F is satisfiable:

- Consider a 0/1 assignment of boolean variables that makes $F = 1$.
- As $F = C_1 \wedge \cdots \wedge C_k$, every C_i must be 1.
- For each C_i, at least one literal in C_i is assigned value 1.

It remains to show:

In our example, the red vertices form a clique of size $k = 3$. If we assign $x_1 = 1, \bar{x}_2 = 1$ and $x_3 = 1$, then $F = 1$.

$\Longrightarrow$ Suppose F is satisfiable:

- Consider a 0/1 assignment of boolean variables that makes $F = 1$.
- As $F = C_1 \wedge \cdots \wedge C_k$, every C_i must be 1.
- For each C_i, at least one literal in C_i is assigned value 1.
- Let C be the subset of vertices of G whose corresponding literals are assigned value 1.
- C contains one vertex from each V_i.

It remains to show:

In our example, the red vertices form a clique of size $k = 3$. If we assign $x_1 = 1, \bar{x}_2 = 1$ and $x_3 = 1$, then $F = 1$.

$\Longrightarrow$ Suppose F is satisfiable:

- Consider a 0/1 assignment of boolean variables that makes $F = 1$.
- As $F = C_1 \wedge \cdots \wedge C_k$, every C_i must be 1.
- For each C_i, at least one literal in C_i is assigned value 1.
- Let C be the subset of vertices of G whose corresponding literals are assigned value 1.
- C contains one vertex from each V_i.
- Since the 0/1 assignment is consistent, no x_i and $\bar{x}_i$ can be both assigned one. So all vertices in C are adjacent to each other.

It remains to show:

In our example, the red vertices form a clique of size $k = 3$. If we assign $x_1 = 1, \bar{x}_2 = 1$ and $x_3 = 1$, then $F = 1$.

$\Longrightarrow$ Suppose F is satisfiable:

- Consider a 0/1 assignment of boolean variables that makes $F = 1$.
- As $F = C_1 \wedge \cdots \wedge C_k$, every C_i must be 1.
- For each C_i, at least one literal in C_i is assigned value 1.
- Let C be the subset of vertices of G whose corresponding literals are assigned value 1.
- C contains one vertex from each V_i.
- Since the 0/1 assignment is consistent, no x_i and $\bar{x}_i$ can be both assigned one. So all vertices in C are adjacent to each other.
- Hence C is a clique of G with size $|C| = k \geq k$.

Maximum Independent Set (MIS) Problem

Let $G = (V, E)$ be an undirected graph.

- An independent set of G is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.
- A MIS of G is an independent set I with maximum size $|I|$.
- The MIS Problem: Given G, find a MIS I of G.
- The decision version of MIS Problem: Given G and t, does G have an independent set I so that $|I| \geq t$?

Maximum Independent Set (MIS) Problem

Maximum Independent Set (MIS) Problem

Let $G = (V, E)$ be an undirected graph.

- An independent set of G is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.
- A MIS of G is an independent set I with maximum size $|I|$.
- The MIS Problem: Given G, find a MIS I of G.
- The decision version of MIS Problem: Given G and t, does G have an independent set I so that $|I| \geq t$?

Theorem: MIS is $\mathcal{NPC}$.

We can easily show MIS $\in \mathcal{NP}$ by describing a non-deterministic algorithm for solving it.
We show MIS is $\mathcal{NP}$-hard by showing Max-Clique $\leq_{\mathcal{P}}$ MIS.

Maximum Independent Set (MIS) Problem

Definition

Let $G = (V, E)$ be a graph. The complement graph of G is $G^c = (V, E^c)$ where

$$E^c = \{(u, v) \mid u \in V, v \in V, u \neq v,\ (u, v) \notin E\}$$

Maximum Independent Set (MIS) Problem

Definition

Let $G = (V, E)$ be a graph. The complement graph of G is $G^c = (V, E^c)$ where

$$E^c = \{(u, v) \mid u \in V, v \in V, u \neq v,\ (u, v) \notin E\}$$

Lemma

A vertex subset C is a clique of $G = (V, E)$ iff C is an independent set in $G^c = (V, E^c)$.

Proof C is a clique of $G \iff$ for any two vertices $u, v \in C$ we have $(u, v) \in E$ $\iff$ for any $u, v \in C$ we have $(u, v) \notin E^c \iff C$ is an independent set of G^c.

Maximum Independent Set (MIS) Problem

Definition

Let $G = (V, E)$ be a graph. The complement graph of G is $G^c = (V, E^c)$ where

$$E^c = \{(u, v) \mid u \in V, v \in V, u \neq v,\ (u, v) \notin E\}$$

Lemma

A vertex subset C is a clique of $G = (V, E)$ iff C is an independent set in $G^c = (V, E^c)$.

Proof C is a clique of $G \iff$ for any two vertices $u, v \in C$ we have $(u, v) \in E$ $\iff$ for any $u, v \in C$ we have $(u, v) \notin E^c \iff C$ is an independent set of G^c.

From this lemma, we can easily show Max-Clique $\leq_{\mathcal{P}}$ MIS:

Maximum Independent Set (MIS) Problem

- Given an instance $\langle G, t \rangle$ of Max-Clique.
- We construct an instance $\langle G^c, t \rangle$ of MIS.
- The construction clearly takes poly-time.
- G has a clique C of size $\geq t \iff G^c$ has an independent set C of size $\geq t$.
- This completes the polynomial time reduction from Max-Clique to MIS.

Minimum Vertex Cover (MVC) Problem

Let $G = (V, E)$ be an undirected graph.

- A vertex cover (VC) of G is a subset $C \subseteq V$ such that for any edge $e = (u, v) \in E$ at least one end vertex of e is in C. (We say "C covers every edge in G".)
- A MVC of G is a VC C with minimum size $|C|$.
- The MVC Problem: Given G, find a MVC C of G.
- The decision version of MVC Problem: Given G and an integer s, does G have a VC C so that $|C| \leq s$?

Minimum Vertex Cover (MVC) Problem

Let $G = (V, E)$ be an undirected graph.

- A vertex cover (VC) of G is a subset $C \subseteq V$ such that for any edge $e = (u, v) \in E$ at least one end vertex of e is in C. (We say "C covers every edge in G".)
- A MVC of G is a VC C with minimum size $|C|$.
- The MVC Problem: Given G, find a MVC C of G.
- The decision version of MVC Problem: Given G and an integer s, does G have a VC C so that $|C| \leq s$?

Theorem: MVC is $\mathcal{NPC}$.

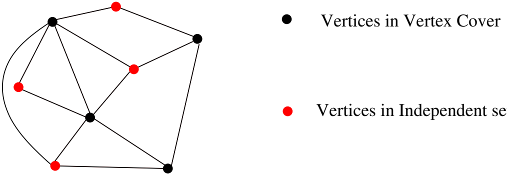
We can easily show MVC $\in \mathcal{NP}$ by describing a non-deterministic algorithm for solving it.
We show MVC is $\mathcal{NP}$-hard by showing MIS $\leq_{\mathcal{P}}$ MVC.

Minimum Vertex Cover (MVC) Problem

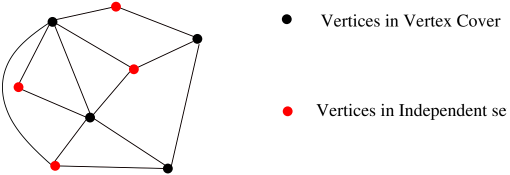
Application

- $G = (V, E)$ represents a communication network.
- Each vertex v is a computer site.
- Each edge $e = (u, v)$ is a communication link between u and v.
- To make sure the network works correctly, we need to monitor each link.
- If we place a monitoring device at a site u, then all links incident to u can be monitored by it.
- The monitors are expensive, we want to use a minimum number of devices to monitor all links.
- How to do this? Find a MVC of G.

Minimum Vertex Cover (MVC) Problem



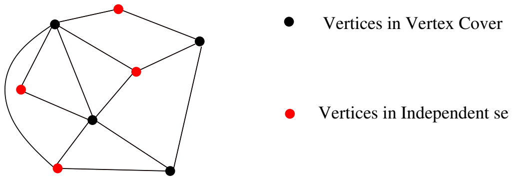
Minimum Vertex Cover (MVC) Problem



Lemma

C is a vertex cover of $G = (V, E) \iff I = V - C$ is an independent set of G.

Minimum Vertex Cover (MVC) Problem



Lemma

C is a vertex cover of $G = (V, E) \iff I = V - C$ is an independent set of G.

Proof: C is a VC of $G \iff$ for any edge $(u, v) \in E$ at least one of u, v is in C $\iff$ for any edge $(u, v) \in E$ not both u and v are in $V - C \iff$ for any edge $(u, v) \in E$ at least one of u and v is not in $I = V - C \iff$ I is an independent set of G.

Minimum Vertex Cover (MVC) Problem

From this lemma, it's easy to show MIS $\leq_{\mathcal{P}}$ MVC:

- Given an instance $\langle G, t \rangle$ of MIS.
- We construct an instance $\langle G, s = n - t \rangle$ of MVC.
- The construction clearly takes poly-time.
- G has an independent set I of size $\geq t \Longleftrightarrow G$ has a vertex cover $C = V - I$ of size $\leq n - t = s$.
- This completes the polynomial time reduction from MIS to MVC.