Intractable Problems

Outline for Today

Intractable Problems

- Definition of P, NP, NP-Hard and NP-Complete (NPC)
- Traveling Salesman Problem
- 0/1 Knapsack, revisited

Background

Defining Efficiency

We have learned many algorithms of different complexities

- Binary search in a sorted list: $O(\log n)$
- RB-tree search, insertion, deletion: $O(\log n)$
- Sequential search in a sorted list: $O(n)$
- Merge Sort: $O(n \log n)$
- Insertion Sort: $O(n^2)$
- Floyd-Warshall: $O(n^3)$

There is something in common: they all have polynomial time complexity

A polynomial function is of the form: $n^a + n^{a-1} + n^{a-2} + \ldots + n^2 + n$
"a" must be a constant, any "n" can be replaced by "logn" since $\log n < n$

Defining Efficiency

What is an efficient algorithm?

An algorithm is efficient iff it runs in polynomial time on a serial computer.

Runtimes of “efficient” algorithms: $O(n)$, $O(n\log(n))$, $O(n^8\log^4(n))$, $O(n^{1,000,000})$.

What are inefficient algorithms?

An algorithm is inefficient if does not run in polynomial time on a serial computer. Usually, they run in exponential time or factorial time.

Runtimes of “inefficient” algorithms: $O(2^n)$, $O(n!)$, $O(1.0000001^n)$.

Why emphasize "serial computer"

Parallelism Some problems can be solved in polynomial time on machines with a polynomial number of processors.

a.k.a. Distributed computing, e.g., distributing the algorithm on many CPUs

Randomization Some algorithms can be solved in expected polynomial time, or have poly-time Monte Carlo algorithms that work with high probability.

Quantum computation Some algorithms can be solved in polynomial time on a quantum computer.

Tractability

A problem is called **tractable** iff there is an efficient (i.e. polynomial time) algorithm that solves it.

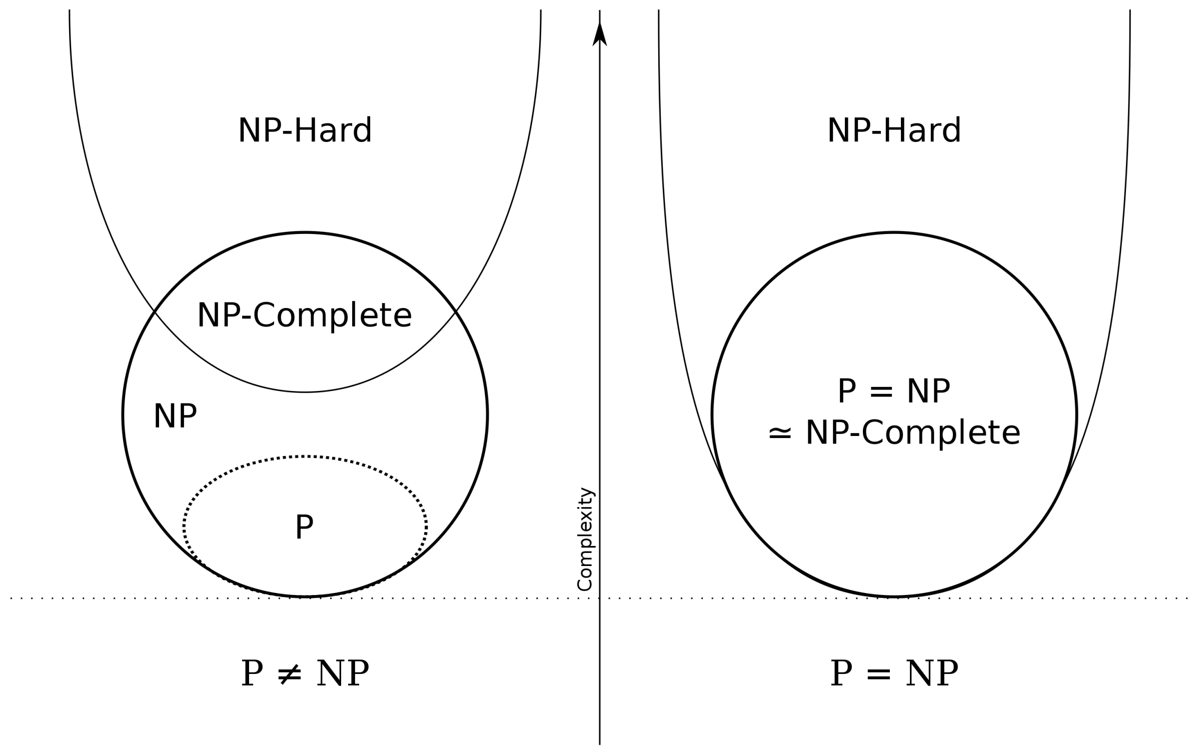
A problem is called **intractable** iff there is no efficient algorithm that solves it.

We can only find exponential or factorial time algorithms for these problems, or even no algorithm at all.

The P Problems

The P problem is the set of all problems that have polynomial time algorithms to solve it.

i.e., the set of tractable problems, “P” means “Polynomial time”.



The NP Problems

The NP problem is set of decision problems that are solvable in polynomial time by a nondeterministic Turing machine.

NP means "Nondeterministic, Polynomial time"

What is a decision problem

A decision problem is a problem with a yes/no answer.

e.g., the 3SAT problem: for the following conjunctive normal form (CNF)

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) = T \ ?$$

Does there exist a valid T/F assignment of variables x_1, x_2 and x_3 so that the equation is T?

The NP Problems

The NP problem is set of decision problems that are solvable in polynomial time by a nondeterministic Turing machine.

NP means “Nondeterministic, Polynomial time”

What is a deterministic Turing machine (DTM)

A (deterministic) Turing machine is a computing machine whose next state is completely determined by its action and the current symbol it sees

What is a nondeterministic Turing machine (NTM)

A nondeterministic Turing machine is a computing machine whose next state is **not** completely determined by its action and the current symbol it sees

NTM is a theoretical model of computation, usually used in thought experiments

The NP Problems

The NP problem is set of decision problems that are solvable in polynomial time by a nondeterministic Turing machine.

NP means "Nondeterministic, Polynomial time"

Solving a decision problem on an NTM, two steps:

Step 1: Guess a solution, in (non)deterministic way

Step 2: Verify the solution, in a deterministic way

e.g., the 3SAT problem

The class **NP** consists of all decision problems where "yes" answers can be verified in polynomial time

NP definition only cares about step 2.

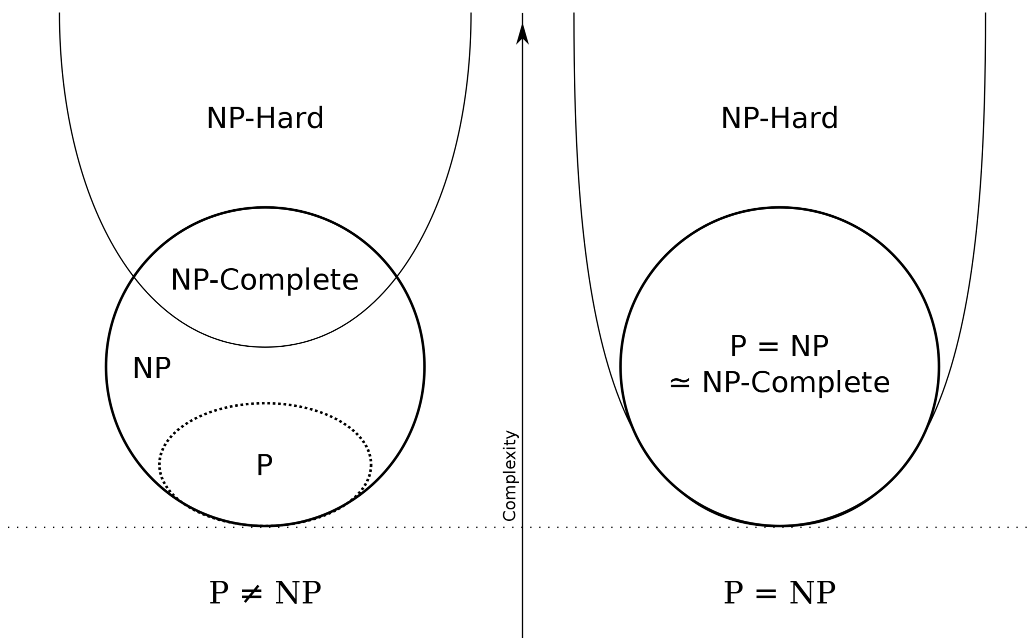
The NP Problems

The NP problem is set of decision problems that are solvable in polynomial time by a nondeterministic Turing machine.

NP means "Nondeterministic, Polynomial time"

All tractable decision problems (P problems) are in **NP**, plus a lot of problems whose difficulty is unknown ($P \subset NP$).

If a problem is P, then we can just solve the problem in polynomial time at step 2



The NPC Problems

The **NP-complete (NPC)** problems are (intuitively) the hardest problems in NP.

Polynomial Reductions:

Consider two decision problems A and B. We say A is polynomially reducible to B if there exists a polynomial time algorithm that can convert each input of A to an input of B such that the answer to B is the answer to A.

Examples:

Problem A: Solve for linear equation: $a x + b = 0$

Problem B: Solve for quadratic equation: $a x^2 + b x + c = 0$

$3 x + 2 = 0$ is reducible to $0 x^2 + 3 x + 2 = 0$

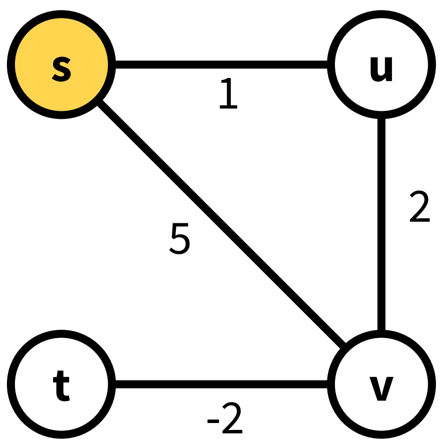
The NPC Problems

The **NP-complete (NPC)** problems are (intuitively) the hardest problems in NP.

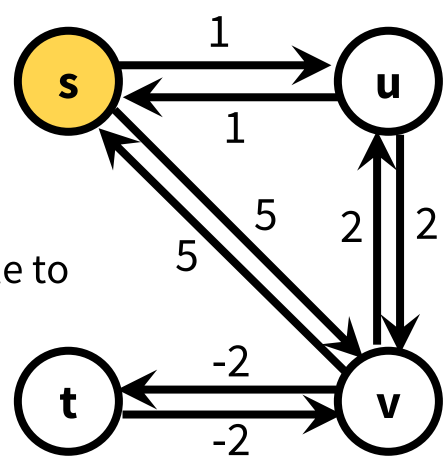
Polynomial Reductions:

Consider two decision problems A and B. We say A is polynomially reducible to B if there exists a polynomial time algorithm that can convert each input of A to an input of B such that the answer to B is the answer to A.

Examples:
Shortest path in undirected graph



is reducible to



The NPC Problems

The **NP-complete (NPC)** problems are (intuitively) the hardest problems in NP.

Polynomial Reductions:

Consider two decision problems A and B. We say A is polynomially reducible to B if there exists a polynomial time algorithm that can convert each input of A to an input of B such that the answer to B is the answer to A.

Transitivity: If $A \prec B$ and $B \prec C$, then $A \prec C$
Complexity: If $A \prec B$, then $O(A) \leq O(B)$

The NPC Problems

Is there any NP problem X such as all other NP problems are reducible to this NP problem X?

The answer is (surprising) YES! And there are more than one such NP problems X.

We call such NP problems X as **NP-complete** problems (NPC).

NPC problems are NP problems (NPC $\subset$ NP)

Intuitively, the NPC problems are the hardest problems in NP.

Some example NPC problems:

3SAT, Hamilton loop problem, Traveling Salesman Problem (TSP)

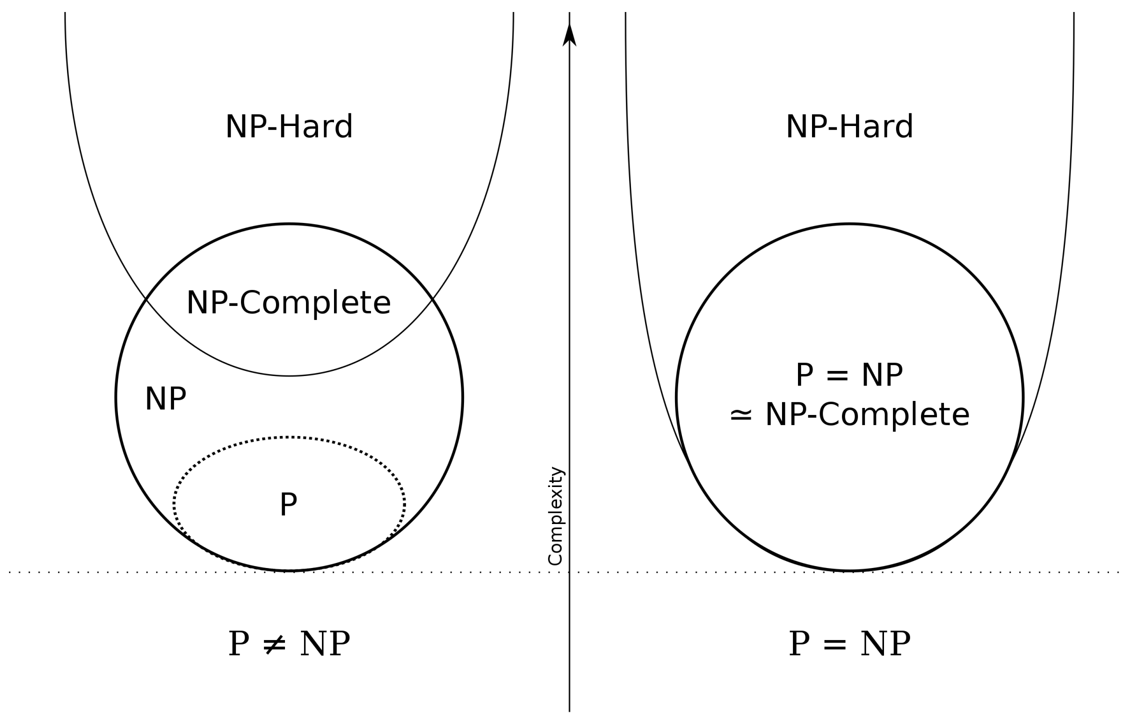
The NPC Problems

A natural question: Is there an efficient (polynomial time) algorithm for an NPC problem?

If the answer is YSE: then all NP problems will be tractable.

If the answer is NO: some NP problems are really intractable.

This is the famous open problem: the P = NP question!



The NPC Problems

A natural question: Is there an efficient (polynomial time) algorithm for an NPC problem?

If the answer is YSE: then all NP problems will be tractable.

If the answer is NO: some NP problems are really intractable.

This is the famous open problem: the P = NP question!

A good (and also surprising) fact: Either all NPC problems are tractable or no NPC problem is tractable.

Because all NPC problems are reducible to each other!

(NP problems are reducible to NPC problems, and NPC problems are NP problems, so NPC problems are reducible to NPC problems.)

By now, there are no known polynomial-time algorithms for any **NPC** problem yet.

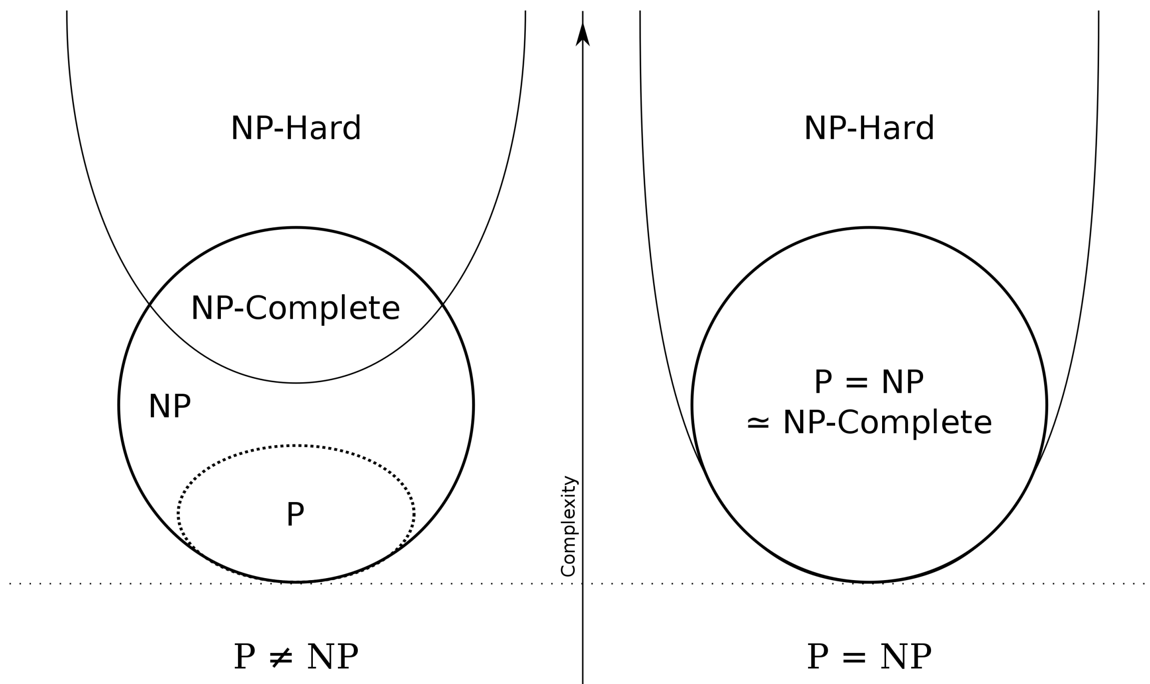
The NPC Problems

By now, people have not found any polynomial-time algorithms for any NPC problem yet.

Example: 3SAT, brute-force search complexity $O(2^n)$

$(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) = T$?

But if polynomial-time algorithm is found for any NPC problem, then we immediately prove P = NP.



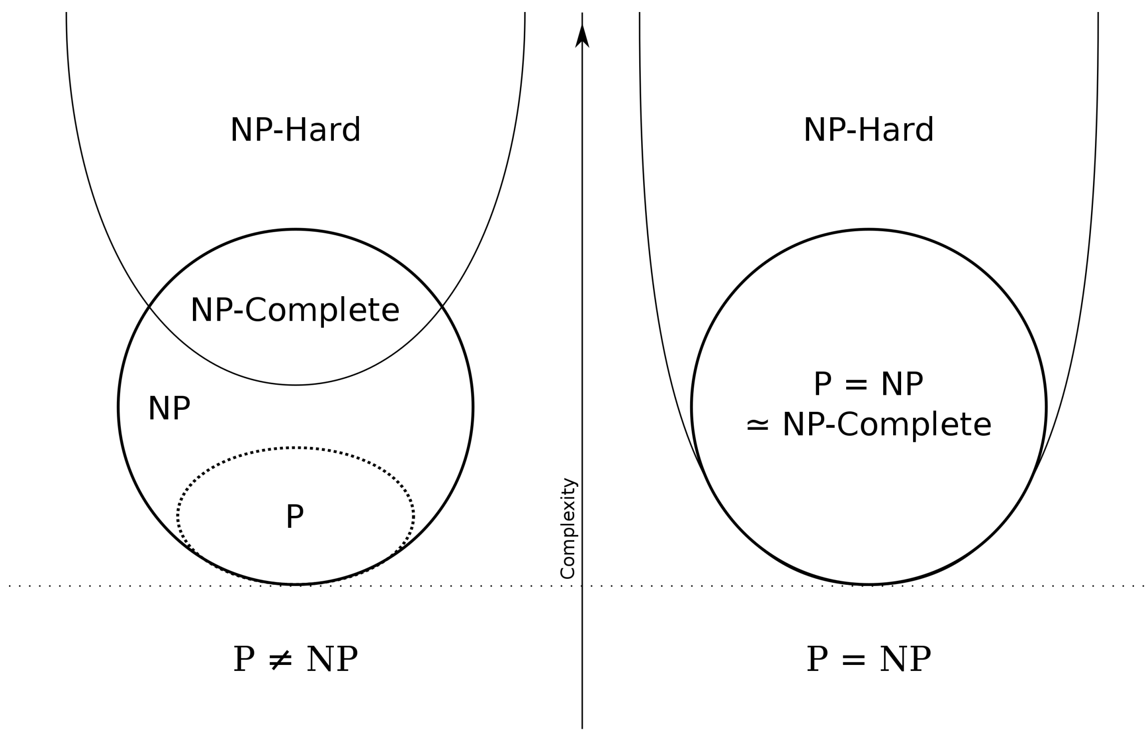
The NPC Problems

To prove P = NP

Try to find a polynomial-time algorithm for an NPC problem.

To prove P != NP

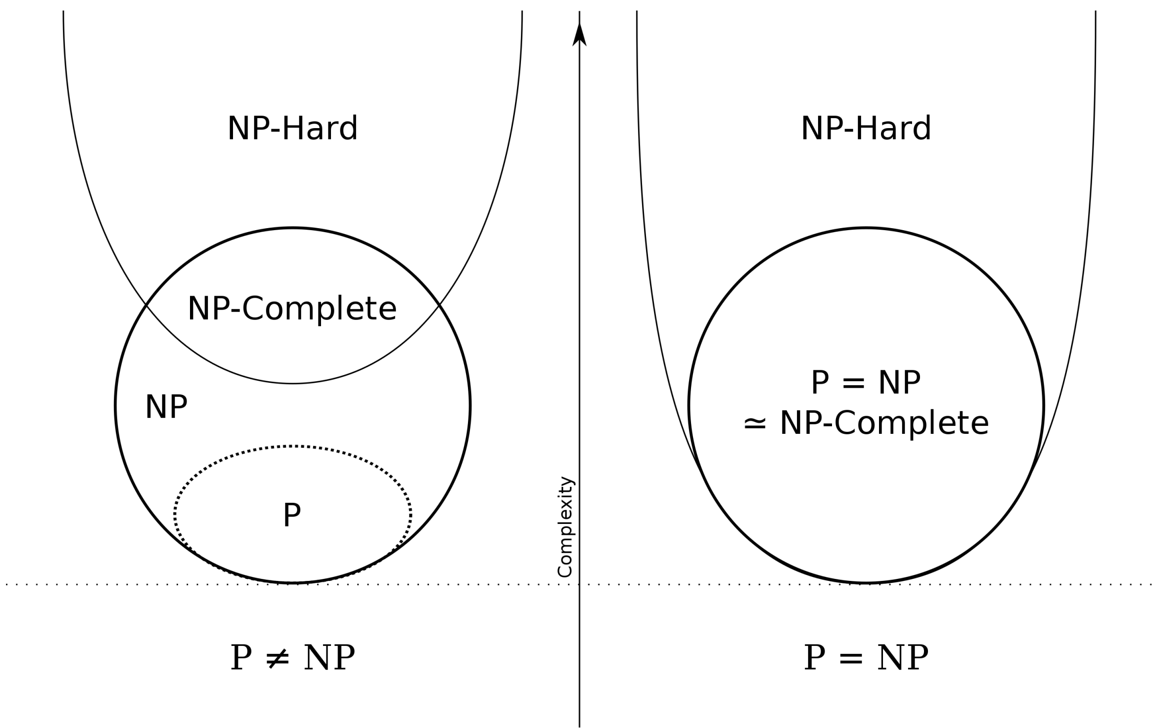
Try to prove that polynomial-time algorithm does not exist for an NPC problem.



NP-Hardness

A problem (which may or may not be a decision problem) is called **NP-hard** if (intuitively) it is at least as hard as every problem in **NP**.

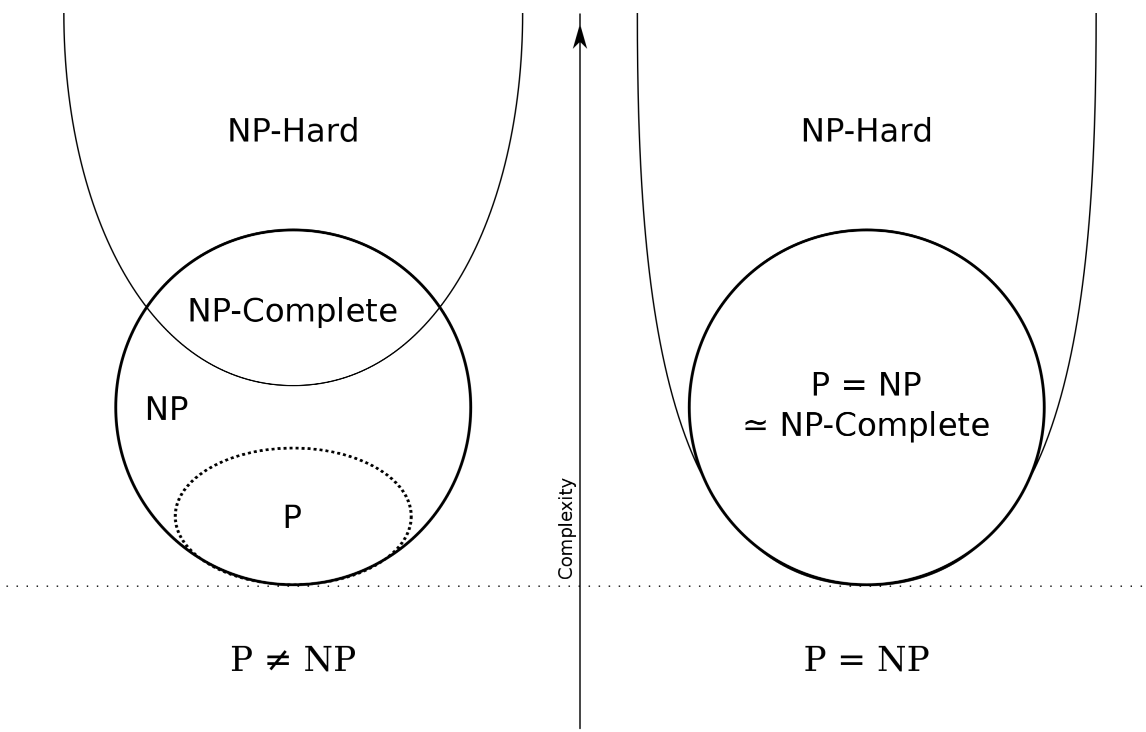
As before: no polynomial-time algorithms are known for any **NP**-hard problem.



NP-Hardness

Assuming that P ≠ NP, all NP-hard problems are intractable.

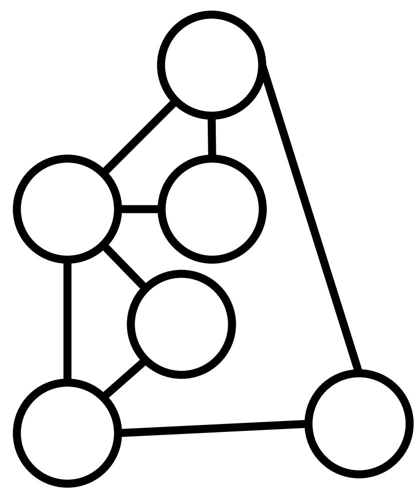
- This does not mean that brute-force algorithms are the only option.
- This does not mean that it is hard to get approximate answers.



Traveling Salesman Problem (TSP)

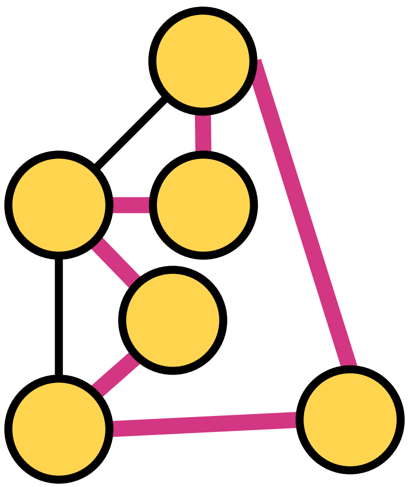
The Hamilton Cycle Problem

A **Hamilton cycle** in an undirected graph G is a cycle that visits each vertex in G exactly once.



The Hamilton Cycle Problem

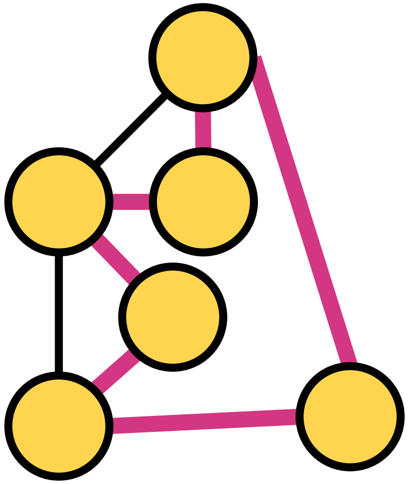
A **Hamilton cycle** in an undirected graph G is a cycle that visits each vertex in G exactly once.



The Hamilton Cycle Problem

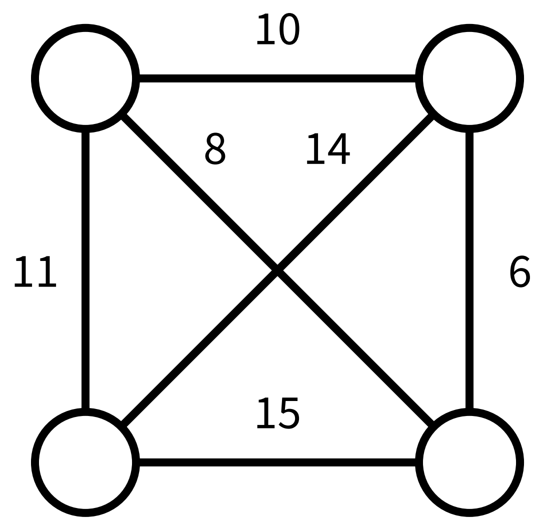
The Hamilton Cycle Problem: Given a graph, decide if a Hamilton cycle exists in the graph (Yes or No).

- This is a decision problem.
- This is an NPC problem (polynomial algorithm exist iff P = NP).



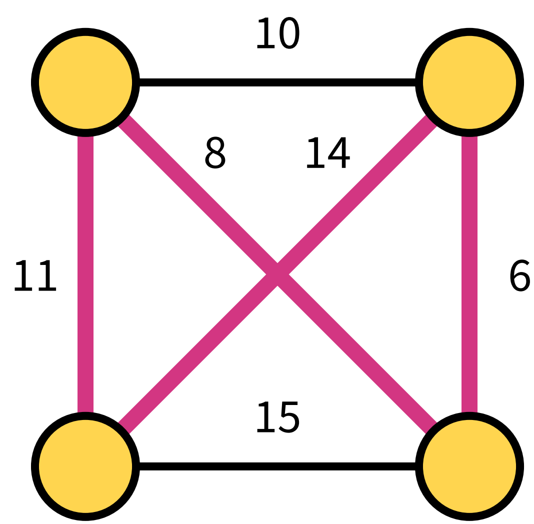
The TSP Problem

Given a complete, undirected weighted graph G, the traveling salesman problem is to find a Hamilton cycle in G of least total cost.



The TSP Problem

Given a complete, undirected weighted graph G, the traveling salesman problem is to find a Hamilton cycle in G of least total cost.

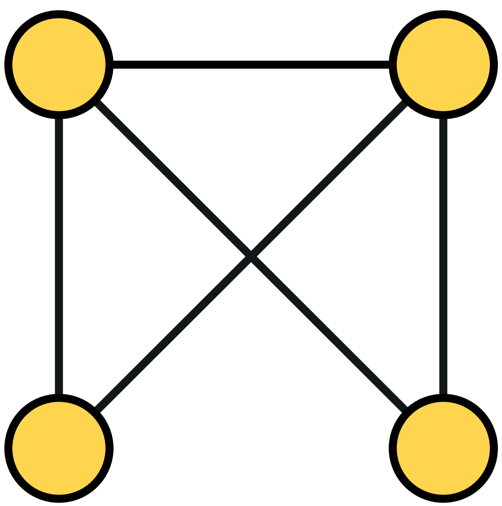


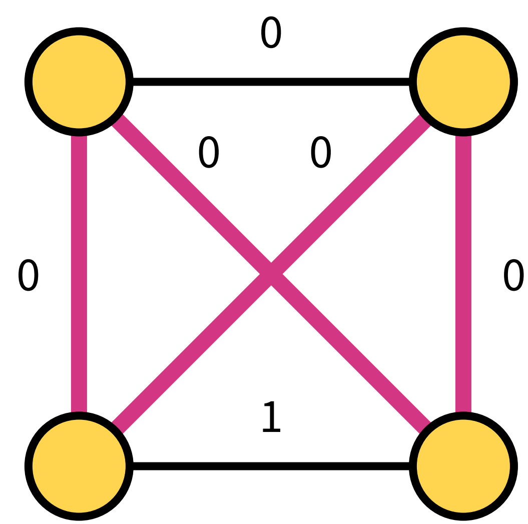
TSP

Given a complete, undirected weighted graph G, the traveling salesman problem is to find a Hamilton cycle in G of least total cost.

Note that since G is complete, there must be at least one Hamiltonian cycle. The challenge is finding the cycle with least cost.

Hamilton Cycle problem is reducible to TSP problem.
TSP problem is known to be **NP**-hard.





TSP

Try all possible Hamiltonian cycles in the graph?

- How many Hamiltonian cycles are there? (n-1)! / 2
- Since each cycle takes O(n)-time, the total time is `O(n!)`.

TSP

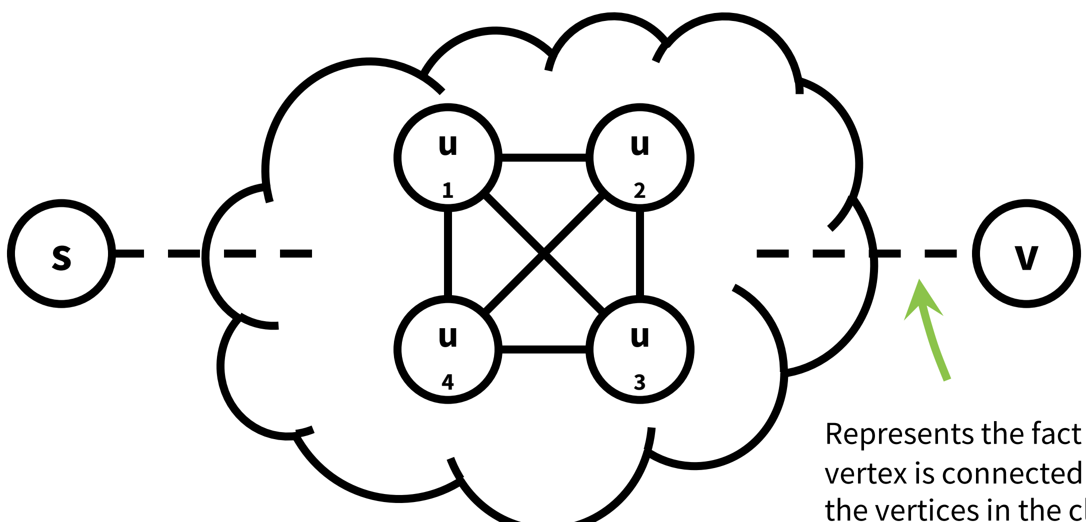
Let OPT(v, S) be the minimum cost of an s - v path that visits exactly the vertices in S. We assume $v \in S$. Let w(u, v) be the weight of the edge (u, v).

Claim OPT(v, S) satisfies the recurrence:

$$\texttt{OPT(v,S)} = \begin{cases} 0 & \text{if } v = s \text{ and } S = \{s\} \\ \infty & \text{if } s \notin S \\ \min_{u \in S - \{v\}} \{\texttt{OPT}(u, S-\{v\}) + w(u,v)\} & \text{otherwise} \end{cases}$$

TSP

$$\text{OPT}(v,S) = \begin{cases} 0 & \text{if } v = s \text{ and } S = \{s\} \\ \infty & \text{if } s \notin S \\ \min_{u \in S-\{v\}} \{\text{OPT}(u,S-\{v\}) + w(u,v)\} & \text{otherwise} \end{cases}$$



Represents the fact that this vertex is connected to all of the vertices in the cloud.

TSP

To solve OPT(v, S), a problem of size |S|, we need to solve subproblems of size |S| - 1.

Idea Evaluate the recurrence on sets of size 1, 2, 3 …, n.

There are 2^n possible subsets of a set S, of which 2^{n-1} contain s.

TSP

```
algorithm tsp(G):
  n = |G.V|
  DP = []  # n × 2^(n-1) table
  s = random vertex from G.V
  DP[s][{s}] = 0
  for k = 2 to n:
    for all sets S ⊆ V where |S| = k and s ∈ S:
      for all v ∈ S - {s}:
        DP[v][S] = min_{u∈S-{v}}{DP[u][S-{v}] + w(u,v)}
  return min_{v≠s}{DP[v][V] + w(v,s)}
```

Runtime: $O(2^n n^2)$

TSP

Each subset of V containing s can be mapped to a unique integer in 0, 1, 2, …, $2^{n-1} - 1$.

- Think of the number as a bitvector where the present elements are 1s and the absent elements are 0s.
- Takes **$O(n)$**-time to compute the above number and index into the table, the cost per subproblem.

$O(2^n n^2)$ total time.

- $O(2^n n)$ total subproblems (cells in the table).
- Solving each subproblem requires us to look at $O(n)$ different subproblems, and $O(1)$-time for each one.
- Map all subsets of V to bitvectors in $O(n)$-time.

TSP

What's the difference between n! and $2^n n^2$?

Compare 20! and $2^{20} 20^2$:

$20! \approx 2.4 \times 10^{18}$

$2^{20} 20^2 \approx 4.2 \times 10^8$

TSP

What's the difference between $n!$ and $2^n n^2$?

- Compare $20!$ and $2^{20}20^2$:
  - $20! \approx 2.4 \times 10^{18}$
  - $2^{20}20^2 \approx 4.2 \times 10^{8}$
- Compare $30!$ and $2^{30}30^2$:
  - $30! \approx 2.6 \times 10^{32}$
  - $2^{30}30^2 \approx 9.7 \times 10^{11}$

TSP

What's the difference between $n!$ and $2^n n^2$?

- Compare $20!$ and $2^{20} 20^2$:
  - $20! \approx 2.4 \times 10^{18}$
  - $2^{20} 20^2 \approx 4.2 \times 10^{8}$
- Compare $30!$ and $2^{30} 30^2$:
  - $30! \approx 2.6 \times 10^{32}$
  - $2^{30} 30^2 \approx 9.7 \times 10^{11}$
- Compare $40!$ and $2^{40} 40^2$:
  - $40! \approx 8.2 \times 10^{47}$
  - $2^{40} 40^2 = 1.8 \times 10^{15}$

TSP

Why this matters?

- Improving upon brute-force (e.g. n!) increases the size of problems that can be solved with exact answers.
- Though there might not exist a poly-time solution, an exponential solution often offers a considerable improvement.

0/1 Knapsack, revisited

Knapsack

0/1 Knapsack

Suppose I only have one copy of each item.

What's the most valuable way to fill the knapsack?

weight	6	2	4	3	11
value	20	8	14	13	35

capacity: 10

Total weight: 9

Total value: 35

Task Find the items to put in a 0/1 knapsack.

0/1 Knapsack

What I didn't say is this problem is known to be **NP**-hard.

- `O(n2ⁿ)` Brute-force solution: try all possible subsets of the items and find the feasible set with the largest total value.
- `O(nlog(n))` Greedy solution: sort items by their "unit value" v_k / w_k.
- `O(nW)` Dynamic programming solution

0/1 Knapsack

Did we just prove P = NP?

A poly-time algorithm is one that runs in time polynomial in the total number of bits required to write out the input to the problem.

Therefore, `O(nW)` is exponential in the number of bits required to write out the input.

Consider W = 1,000,000,000,000. It only takes 40 bits to represent this number, so input size = 40, but the computational runtime uses the factor 1,000,000,000,000 which is $O(2^{40})$. So the runtime is more accurately said to be $O(n\ 2^{\text{bits in W}})$, which is exponential.

The DP runtime of `O(nW)` is better than our brute-force runtime of $O(n2^n)$, provided that $W = o(2^n)$.

That's a little-o, not a big-O.

For any fixed W, this algorithm runs in linear time!

Parameterized Complexity

Parameterized complexity is a branch of complexity theory that studies the hardness of problems with respect to different “parameters” of the input.

- In the case of 0/1 Knapsack, `O(nW)` has two parameters: the number of items (n) and capacity (W).
- Often, **NP**-hard problems aren’t entirely infeasible as long as some parameter of the problem is fixed.

Fixed Parameter Tractability

Suppose that the input to a problem P can be characterized by two parameters, n and k.

P is called fixed-parameter tractable (or psudo-polynomial) iff there is some algorithm that solves P in time `O(f(k)p(n))`.

- f(k) is an arbitrary function and p(n) is a polynomial in n..
- Intuitively, for any fixed k, the algorithm runs in a polynomial in n since that polynomial p(n) does not depend on choice of k.

Fixed Parameter Tractability

Suppose that the input to a problem P can be characterized by two parameters, n and k.

P is called fixed-parameter tractable (or psudo-polynomial) iff there is some algorithm that solves P in time **`O(f(k)p(n))`**.

- f(k) is an arbitrary function and p(n) is a polynomial in n..
- Intuitively, for any fixed k, the algorithm runs in a polynomial in n since that polynomial p(n) does not depend on choice of k.

Acknowledgement: Part of the materials are adapted from Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.