The Halting Problem

Class 38

Grading Programs

- sometimes I teach CS 170 or CS 180
- beginning programmers are always writing infinite loops

```
def main():
  value = int(readline('input.dat'))

  while value != 0:
    print(value)
    value -= 1
```

- fine if the file contains 5
- not so much if it contains -7

Hard to Tell

- note that sometimes it's hard to tell
- some loops definitely end
- some loops are clearly infinite
- but sometimes when a program is running, it is not always obvious whether
    - it is stuck in an infinite loop
    - it is simply taking a long time to run
- nqueens 25

Grading Helper

- it would be so useful if I had a program named *does_end*
- does_end would have two parameters
    1. filename of source code of a program I want to grade
    2. filename of input data for testing the program
- `$ ./does_end beginner.py input.dat`

Grading Helper

- it would be so useful if I had a program named *does_end*
- does_end would have two parameters
  1. filename of source code of a program I want to grade
  2. filename of input data for testing the program
- `$ ./does_end beginner.py input.dat`
- does_end would evaluate *beginner.py* using the contents of *input.dat*
- does_end would output either
  1. true
  2. false
- to tell me whether, if I run beginner.py with the data in input.dat, it would terminate or not

Compilers

- this shouldn't be so hard
- a compiler (or interpreter) is a program whose input is a program
- a compiler analyzes its input, parses it into tokens
- knows all about its variables and call structure

- a debugger is a program whose input is a program
- allows you to step statement-by-statement through a program
- see exactly what values are in a variable, how loops work, etc.

- does_end could be like a compiler plus a debugger
- analyze the program in the context of the input, and determine whether it will terminate or not

The Halting Problem

- however, this cannot work
- consider this program, named paradox.pl
  (written in Perl because Python doesn't have goto)

```
program paradox($filename)
{
  loop:
    result = system("does_end.py($filename, 'input.dat')");
    if (result == 'true')
      goto loop;
}
```

- program paradox goes into an infinite loop if the program "filename" would halt normally
- program paradox halts normally if "filename" would get stuck in an infinite loop

The Halting Problem

- but now what happens with this:

```
paradox("paradox.pl");
```

- if paradox (inner) has an infinite loop, then paradox (outer) halts normally
- if paradox (inner) halts normally, then paradox (outer) has an infinite loop
- but inner and outer paradox are the same program
- a program cannot both halt and be in an infinite loop at the same time
- we have a contradiction
- thus it is impossible to write the program *does_end*

The Halting Problem

- but now what happens with this:

```
paradox("paradox.pl");
```

- if paradox (inner) has an infinite loop, then paradox (outer) halts normally
- if paradox (inner) halts normally, then paradox (outer) has an infinite loop
- but inner and outer paradox are the same program
- a program cannot both halt and be in an infinite loop at the same time
- we have a contradiction
- thus it is impossible to write the program *does_end*

- Alan Turing, 1936

The Halting Problem

- the halting problem is unsolvable by any algorithm
- also called undecidable because it is framed as a decision problem (yes or no)
- another unsolvable problem: arithmetic satisfiability

- given a polynomial equation consisting of an arbitrary number of variables of arbitrary degree, e.g.,

$$x^3yz + 2y^4z^2 - 7xy^5z = 6$$

- are there integer values of the variables that satisfy the equation?
- it is impossible to write an algorithm to find the values

Easy Problems

- most of the semester we have been studying problems that are "easy" once you know the right algorithm and data structures to use
  - binary search: $O(\lg n)$, $\Omega(1)$
  - divide-and-conquer selection: $O(\lg n)$, $\Omega(1)$
  - smart sorting algorithms: $\Theta(n \lg n)$
  - recursive memoized string alignment: $O(mn)$, $\Omega(m + n)$

Hard Problems

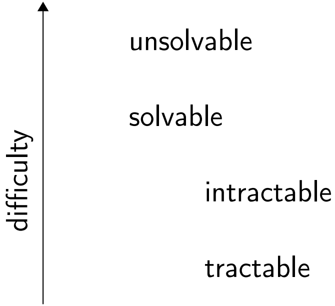
- and some that are truly “hard” regardless of how you code them
    - non-memoized d.p. programs $\in O(n!)$
    - n-queens $\in O(n!)$ or $O(n^n)$
- and some problems that cannot be solved at all
    - halting problem
    - arithmetic satisfiability problem

Classification

- tractable problems are ones that have polynomial-time algorithms
- intractable problems are ones for which no polynomial-time algorithm is known

Status

- so at this point we have



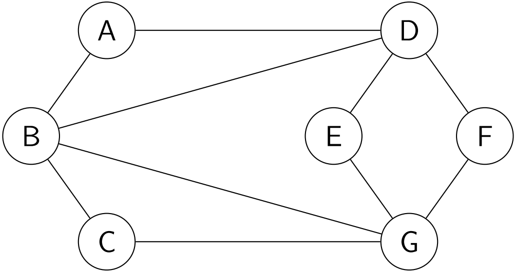
Decision Problems

- most of the problems we have worked with this semester have phrasings such as
    - where is 5 in the array?
    - arrange these elements in order
    - what is the minimum number of coins to make 50 cents, and what are the coin values?
- in computability theory, these types of problems are awkward to work with
- instead, we wish to discuss decision problems
- a decision problem is one that has a true-or-false answer

Decision Problems

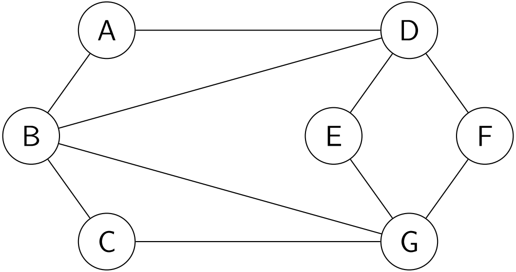
- every problem can be phrased in terms of one or more decision problems
- for example, "where is 5 in the array?"
- can be rephrased as
    1. is 5 at location 0?
    2. is 5 at location 1?
    3. is 5 at location 2?
    4. ...
- which is a series of decision problems
- for computability discussions, we technically must always phrase the problems as decision problems

Independent Set



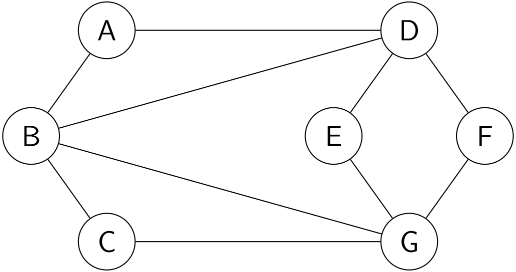
- given a graph
- a set of vertices S is independent if
- no two vertices in S are joined by an edge

Independent Set



- there are many trivial independent sets, e.g., $\{A\}$
- we want to find the largest independent set
- as a decision problem, "does G contain an independent set of size k?"
- what is the largest independent set of this graph?

Vertex Cover



- given a graph G a set of vertices S is a vertex cover if every edge has at least one end in S
- here, we want the smallest vertex cover
- this is similar to a minimum spanning tree, but with the roles of edges and vertices reversed
- what is a vertex cover of this graph?

Two Problems

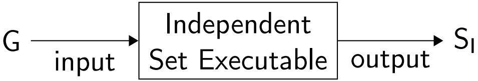
- we have independent set problem I
- and vertex cover problem C
- we don't currently have an algorithm for either one

Two Problems

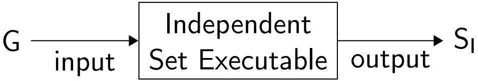
- we have independent set problem I
- and vertex cover problem C
- we don't currently have an algorithm for either one

- imagine that we did have a program for I
- imagine that we don't have source code, just an executable that solves I
- the input for I is a graph G
- the output for I is an independent set S

Independent Set



Independent Set

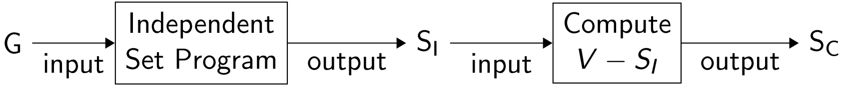


- it turns out there is a theorem in graph theory:

Given graph $G(V, E)$ and independent set S_I, $V - S_I$ is a vertex cover for G.

Vertex Cover

- so we can compute vertex cover this way



Strategy to Solve C

- we have a black-box program that solves one problem (I)
- we want to solve a different problem (C)
- we can
  1. start with C's input
  2. convert it in polynomial time into I's input
  3. solve I using the black box, giving us I's output
  4. convert I's output in polynomial time to C's output
- if we can do this, we say C is polynomially reducible to I
- we write $C \leq_p I$
- and say "I is at least as hard as C"

Polynomial Reducibility

- two very important characteristics of polynomial reducibility
  1. let $Y \leq_p X$. if X can be solved in polynomial time, then Y can be solved in polynomial time
  2. let $Y \leq_p X$. if Y cannot be solved in polynomial time, then X cannot be solved in polynomial time
- if Y is a hard problem, and we can show that $Y \leq_p X$ then Y's "hardness" has spread to X
- to reiterate, we can say that X is at least as hard as Y

Problem Categories

- tractable problems have worst-case polynomial-time solutions
- thus we will call the class of those problems P
- P is the set of all (decision) problems with worst-case polynomial-time solutions

Problem Categories

- tractable problems have worst-case polynomial-time solutions
- thus we will call the class of those problems P
- P is the set of all (decision) problems with worst-case polynomial-time solutions

- there is another big category of problems that have no known polynomial-time solutions
- however, if you propose a potential solution, that solution can be checked in polynomial time

Polynomial-Time Check

- for example, solving *n*-queens is computationally very expensive (definitely not in set P)
- but if I give you this proposed board
- how hard is it for you to check to see if it is a solution or not?
- what is the algorithm?

```
  +---+---+---+---+
4 |   |   | X |   |
  +---+---+---+---+
3 | X |   |   |   |
  +---+---+---+---+
2 |   |   |   | X |
  +---+---+---+---+
1 |   | X |   |   |
  +---+---+---+---+
    a   b   c   d
```

Propose-Then-Check

- in fact, this is a solution strategy
  1. "somehow" guess a possible solution
  2. check (in polynomial time) whether the solution is correct

Propose-Then-Check

- in fact, this is a solution strategy
  1. "somehow" guess a possible solution
  2. check (in polynomial time) whether the solution is correct
- this strategy is called NP
  1. non-deterministically guess a solution
  2. polynomially check whether the solution is correct
- we will denote NP as the class of problems that can be solved by the NP strategy
- what problems are in NP?

NP Problems

- vertex cover
- independent set
- Hamiltonian circuit (does a path exist that touches every vertex except the start and end exactly once)
- clique (find the largest complete subgraph of a graph)
- integer factorization (given integers $m < n$, is there an integer factor f of n such that $1 < f < m$?)
- n-queens

- note that each of these needs to be phrased as a decision problem to accurately state they are NP

k-CNF

a Boolean expression is in conjunctive normal form if it is the conjunction of disjunctive clauses

here is a CNF expression:

$$(x_1 \vee \overline{x_2} \vee x_3 \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

each clause contains literals; each literal is either positive or negative

an expression is in k-CNF if each clause contains exactly k literals
the expression above is not in k-CNF for any k, but the following is in 3-CNF:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_4 \vee \overline{x_1} \vee x_3)$$

Satisfiability

- a k-CNF expression that evaluates to true for some assignment of truth values to each variable in the expression is satisfiable
- a k-CNF expression that is false for all possible assignments to its variables is unsatisfiable

is the following expression satisfiable?

$$(x_0 \vee \overline{x_1} \vee x_2) \wedge (x_0 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_0} \vee \overline{x_1} \vee \overline{x_2})$$

Satisfiability

- a k-CNF expression that evaluates to true for some assignment of truth values to each variable in the expression is satisfiable
- a k-CNF expression that is false for all possible assignments to its variables is unsatisfiable

is the following expression satisfiable?

$$(x_0 \vee \overline{x_1} \vee x_2) \wedge (x_0 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_0} \vee \overline{x_1} \vee \overline{x_2})$$

yes: $x_0 = T, x_1 = F, x_2 = F, x_3 = T$

3SAT

- 3SAT is the problem: given an arbitrary 3-CNF expression, is it satisfiable?
- 3SAT is an NP problem, because if I propose a solution, the solution can easily be checked in polynomial time

P and NP

- note that every P problem is in NP
- recall that the NP algorithm is
  1. non-deterministically guess a solution
  2. polynomially check whether the solution is correct
- for a problem that can be entirely solved with a polynomial algorithm, then clearly step 2 can be done with a polynomial algorithm

Hardest Problem

- we already know that some problems can be used to solve other problems (polynomial reducibility)
- is there some "master" problem that can be used to solve every other problem?
- that would be the hardest possible problem that can be solved
- if we found a master problem X, we would be able to write

$$\text{any problem} \leq_p X$$

- for example

$$\begin{aligned} 3SAT &\leq_p X \\ I &\leq_p X \\ C &\leq_p X \end{aligned}$$

- could such a master problem exist?

Cook

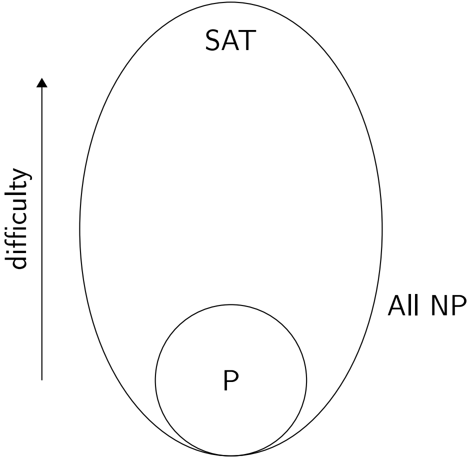
- in 1971, Stephen Cook proved not only could such a problem exist
- he showed that the problem was SAT, the general Boolean satisfiability problem
- SAT is just like 3SAT, but allows each disjunctive clause to contain any number of variables, not just 3
- Cook proved that every NP problem Y can be polynomially reduced to SAT, so we can write

$$\text{Y} \ \leq_p \ \text{SAT}$$

- thus SAT is the hardest possible problem in NP

P, NP, and SAT

- since P problems are the easiest NP problems
- and SAT is the hardest possible NP problem



NPC

- for Cook's SAT problem, we use the term

  NP-Complete

- meaning the complete set of problems in NP can all be modeled by SAT

Unclear

- everyone was happy . . . for about 5 minutes

Unclear

- everyone was happy . . . for about 5 minutes
- Cook's proof had demonstrated

$$\mathrm{I} \ \leq_p \ \mathrm{SAT}$$

Unclear

- everyone was happy ... for about 5 minutes
- Cook's proof had demonstrated

$$\mathsf{I} \ \leq_p \ \mathsf{SAT}$$

- but it's also true that

$$\mathsf{SAT} \ \leq_p \ \mathsf{I}$$

- what does this mean?

Unclear

- everyone was happy ... for about 5 minutes

- Cook's proof had demonstrated

$$\text{I} \ \leq_p \ \text{SAT}$$

- but it's also true that

$$\text{SAT} \ \leq_p \ \text{I}$$

- what does this mean?

- SAT is not harder than I
- rather, they are equally hard
- so NPC is not one problem, but a set of problems

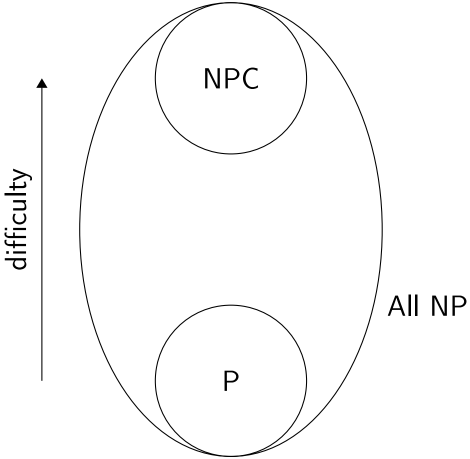
NPC

- immediately several other problems were shown to be in the same set

$$
\begin{aligned}
\text{SAT} &\leq_p \text{C} \\
\text{SAT} &\leq_p \text{TSP} \\
\text{SAT} &\leq_p \text{graph coloring}
\end{aligned}
$$

P, NP, and NPC

- so now the picture is:



NPC

- there are only two ways for a problem X to be classified as NPC
  1. prove it from first principles, as Cook did with SAT
  2. prove there is a polynomial reduction

     $$Y \leq_p X$$

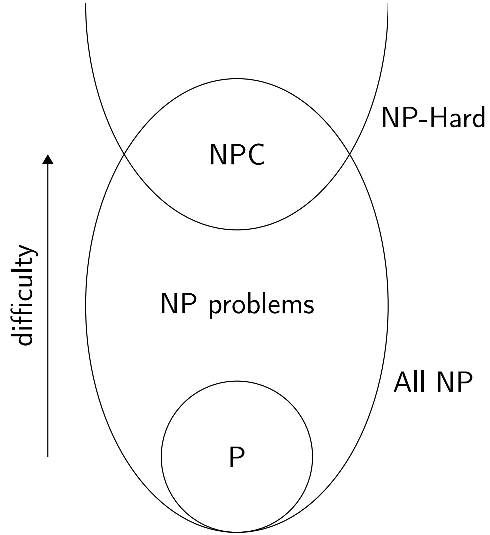
     for some problem Y which has already been proven to be in NPC

NP-Hard

- we have been talking about decision problems
- but some problems are hard to frame as decision problems
  - the chessboard arrangement of queens in the n-queens problem
- we can easily extend our definition a bit to include all problems

- NP-Hard is the set of problems at least as hard as NPC
- but not necessarily decision problems

The Big Picture



Categories

- it is critical to note that

$$P \subset NP$$

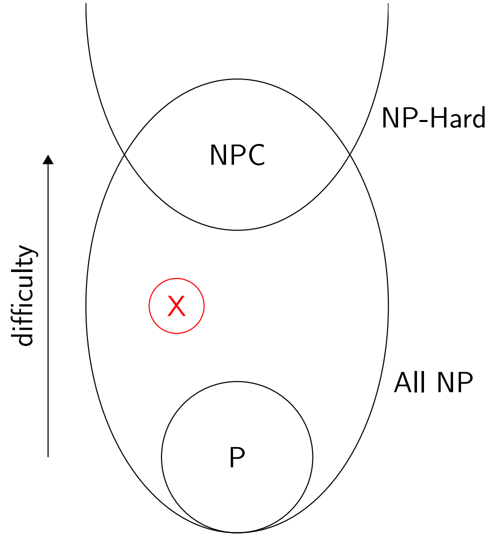
and

$$NPC \subset NP$$

- there are problems in NP that are not in NPC and not in P
- a problem is not in NPC because no one has proved it to be
- a problem is not in P because no one has been clever enough to invent a polynomial-time algorithm to solve it

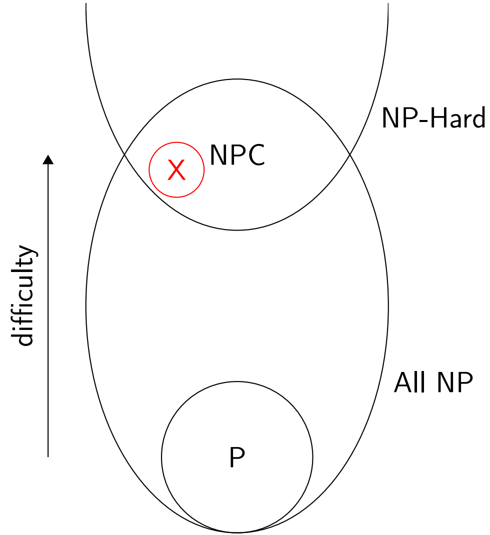
A New Problem

- what happens if problem X is found to be NPC?



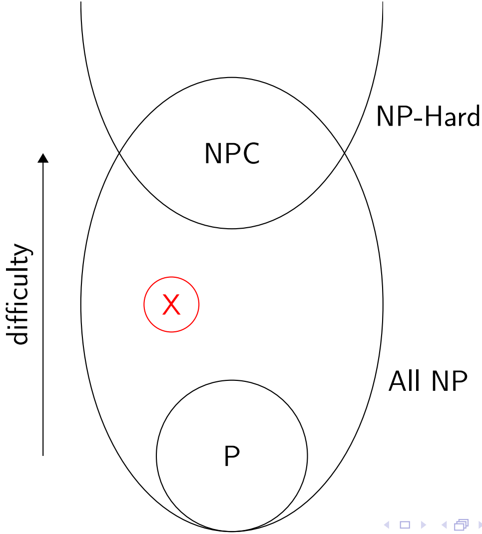
A New Problem

- the picture becomes



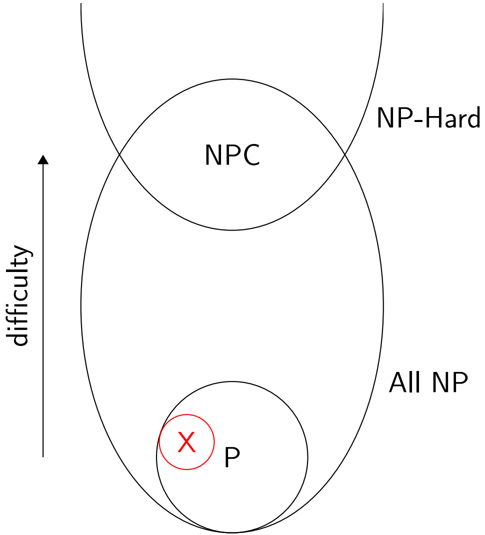
A New Problem

- what happens if problem X is solved with a new polynomial-time algorithm?



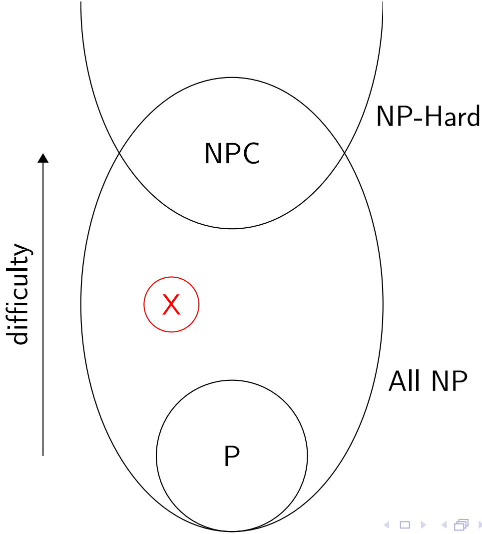
A New Problem

- the picture becomes



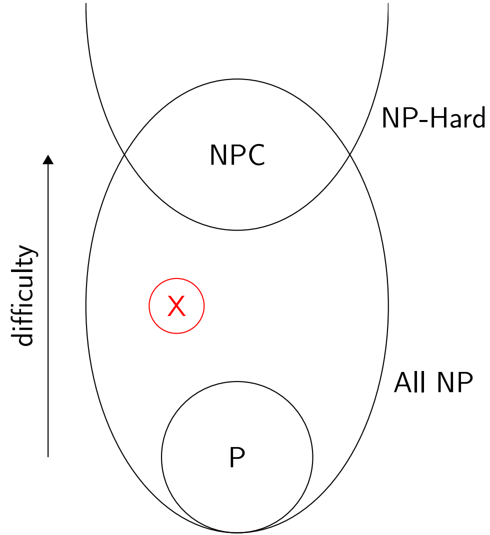
A New Problem

- what happens if problem X is solved with a new factorial-time algorithm?



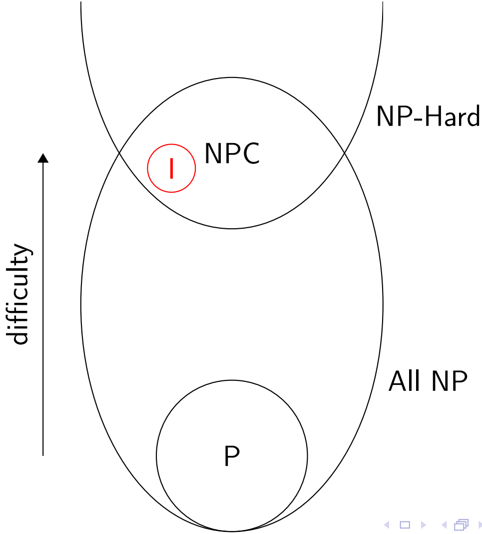
A New Problem

- the picture remains the same



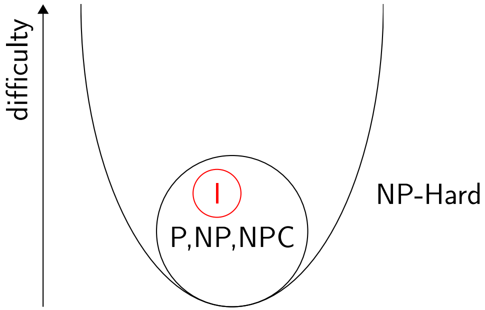
A New Problem

- what happens if I (or any other problem in NPC) is solved with a new polynomial-time algorithm?



A New Problem

- the picture becomes



P = NPC?

- no one has ever proved

$$P \neq NPC$$

or

$$P = NPC$$

- this is the greatest unsolved problem in computer science
- if you demonstrated either one, you would instantly become the most famous computer scientist
- it is one of the Millennium Problems and carries a \$1M prize
- a proof either way would have profound implications for mathematics, cryptography, artificial intelligence, game theory, philosophy, economics, and many other fields