CSE214 – Analysis of Algorithms

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/Analysis-of-Algorithms-2019

Lecture 11

Complexity Classes (P, NP) (Polynomial, Nondeterministic Polynomial, and Beyond)

Based on Andrew Davison's Lecture Notes - Prince of Songkla University (PSU)

Objective

look at the complexity classes P, NP, NP-Complete, NP-Hard, and undecidable problems

Overview

1. A Decision Problem
2. Polynomial Time Solvable (P)
3. Nondeterministic Polynomial (NP)
4. Nondeterminism
5. P = NP : the **BIG** Question
6. NP-Complete
7. The Circuit-SAT Problem
8. NPC and Reducibility
9. NP-Hard
10. Approximation Algorithms
11. Decidable Problems
12. Undecidable Problems
13. Computational Difficulty
14. Non-Technical P and NP

1. A Decision Problem

The problems in the P, NP, NP-Complete complexity classes (sets) are all decision problems

A decision problem is one where the solution is a **yes or no**

e.g. is 29 a prime number?

2. Polynomial Time Solvable (P)

The P set contains problems that are solvable in polynomial-time

- the algorithm has running time $O(n^k)$ for some constant k.
- Polynomial times: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
- Not polynomial: $O(2^n)$, $O(n^n)$, $O(n!)$

e.g. testing if a number is prime runs in O(n) time

P problems are called **tractable** because of their polynomial running time.

P Examples

Problem	Description	Algorithm	Yes	No
MULTIPLE	Is x a multiple of y?	Grade school division	51, 17	51, 16
RELPRIME	Are x and y relatively prime?	Euclid (300 BCE)	34, 39	34, 51
PRIMES	Is x prime?	AKS (2002)	53	51
EDIT-DISTANCE	Is the edit distance between x and y less than 5?	Dynamic programming	niether neither	acgggt ttttta
LSOLVE	Is there a vector x that satisfies Ax = b?	Gauss-Edmonds elimination	$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

3. Nondeterministic Polynomial (NP)

NP is the set of problems that

- can be **checked** by a polynomial time algorithm
- but **can** (or **might)** **<u>not</u> be calculated/solved** in polynomial time

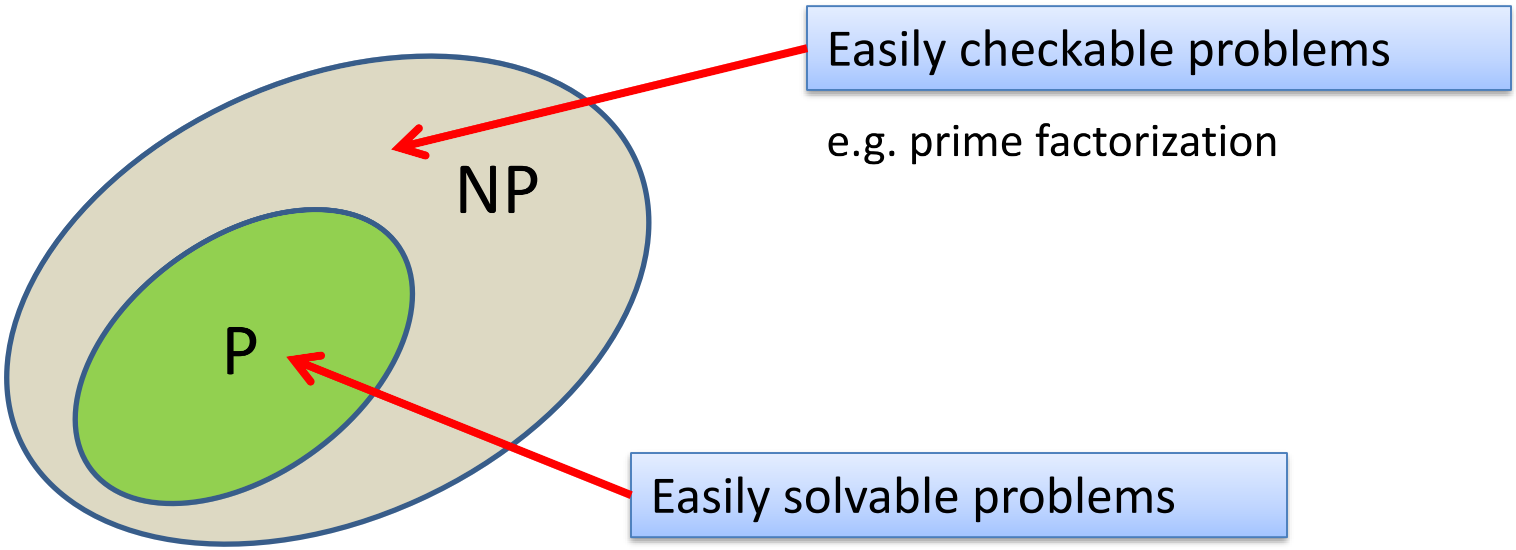
Most people believe that the NP problems are not polynomial time solvable (written as $P \neq NP$)

- this means "cross out the might"

Algorithms to solve NP problems require exponential time, or greater

- they are **intractable**

P and NP as Sets



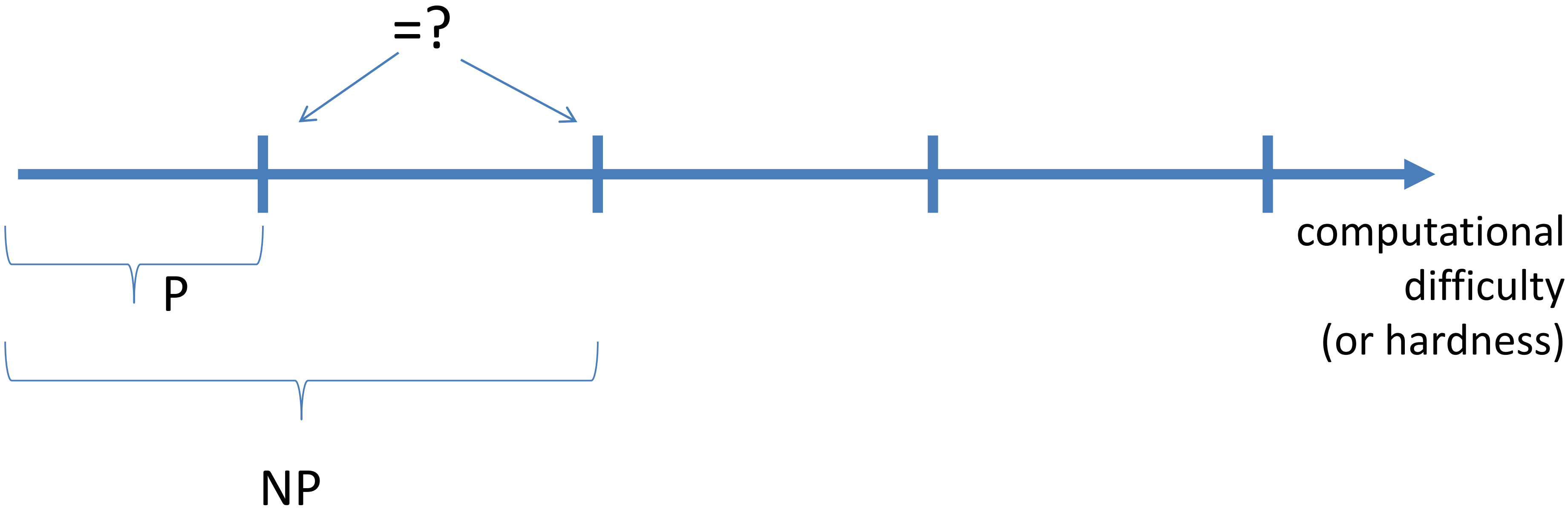
e.g. prime factorization

e.g. multiplication, sorting, prime testing

P = NP

or P ≠ NP

Computational Difficulty Line



3.1. Factorization is in NP

Prime Factorization is the decomposition of a composite number into prime divisors.

$24 = 2 \times 2 \times 2 \times 3$

$91 = 7 \times 13$

This problem is in NP, not P

It is **not** possible to find (**solve**) a prime factorization of a number in **polynomial** time

But given a number and a set of prime numbers, we can **check** if those numbers are a factorization by multiplication, which is a **polynomial** time operation

A Bigger Example

? x ? =

3,107,418,240,490,043,721,350,750,035,888,567,930,037,346,
022,842,727,545,720,161,948,823,206,440,518,081,504,556,
346,829,671,723,286,782,437,916,272,838,033,415,471,073,
108,501,919,548,529,007,337,724,822,783,525,742,386,454,
014,691,736,602,477,652,346,609

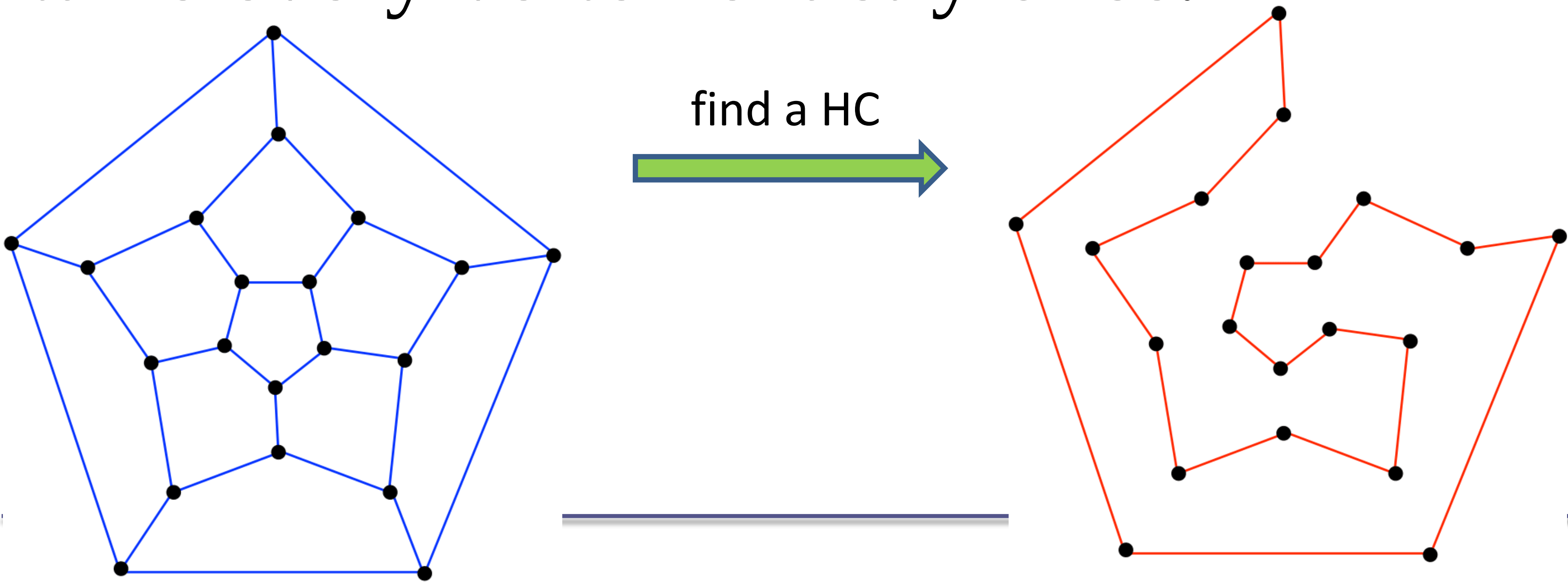
Answer is:

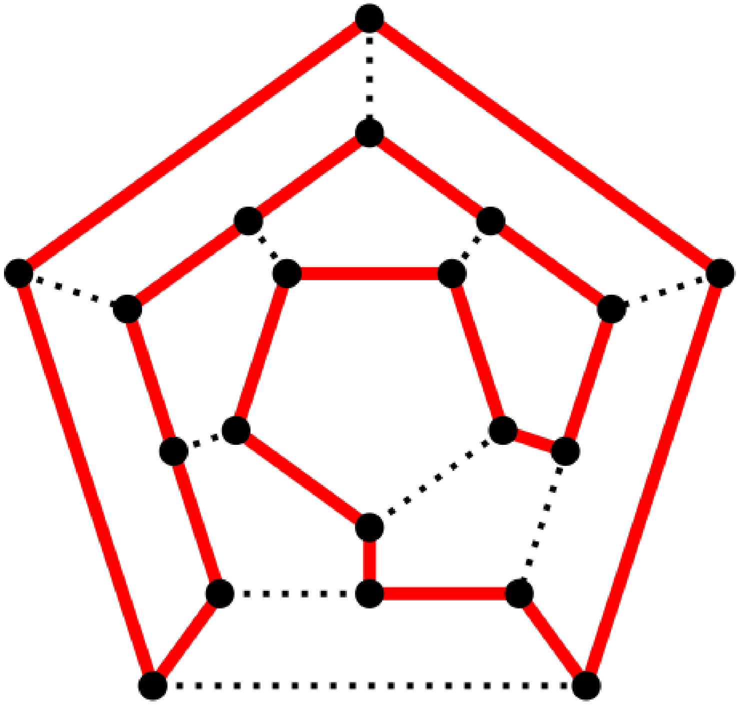
1,634,733,645,809,253,848, 443,133,883,865,090,859, 841,783,670,033,092,312, 181,110,852,389,333,100, 104,508,151,212,118,167, 511,579	X	1,900,871,281,664,822,113, 126,851,573,935,413,975, 471,896,789,968,515,493, 666,638,539,088,027,103, 802,104,498,957,191,261, 465,571

3.2. HC: Another NP Example

Is there a **Hamiltonian Cycle** in a graph G ?

A Hamiltonian cycle of an undirected graph contains every vertex exactly once.





A Hamiltonian cycle is a closed loop on a graph where every node (vertex) is visited exactly once.

A loop is just an edge that joins a node to itself; so a Hamiltonian cycle is a path traveling from a point back to itself, visiting every node en route.

There isn't any equation or general trick to finding out whether a graph has a Hamiltonian cycle; the only way to determine this is to do a complete and exhaustive search, going through all the options.

Finding a HC

How would an algorithm find an Hamiltonian cycle in a graph G?

- One solution is to list all the permutations of G's vertices and check each permutation to see if it is a Hamiltonian cycle.

What is the running time of this algorithm?

- there are **|V|!** possible permutations, which is an exponential running time

So the Hamiltonian cycle problem is **not polynomial time solvable**.

Checking a HC

But what about checking a possible solution?

You are given a set of vertices that are meant to form an Hamiltonian cycle

Check whether the set is a permutation of the graph's vertices and whether each of the consecutive edges along the cycle exists in the graph

- Deciding whether a graph G has a Hamiltonian cycle is **not polynomial time solvable** but is **polynomial time checkable**, so is in NP.

4. Nondeterminism

A deterministic algorithm behaves **predictably**.

When it runs on a particular input, it always produces the same output, and passes through the same sequence of states.

A nondeterministic algorithm has two stages:

- 1. **Guessing** (in **nondeterministic polynomial time** by generating guesses that are **always** correct)
  - The (magic) process that always make the right guess is called an **Oracle**.
- 2. **Verification/checking** (in **deterministic polynomial time**)

Example (Searching)

Is x in the array A: (3, 11, 2, 5, 8, 16, …, 200)?

- Deterministic algorithm

```
for i=1 to n
  if (A[i] == x) then { print i; return true }
return false
```

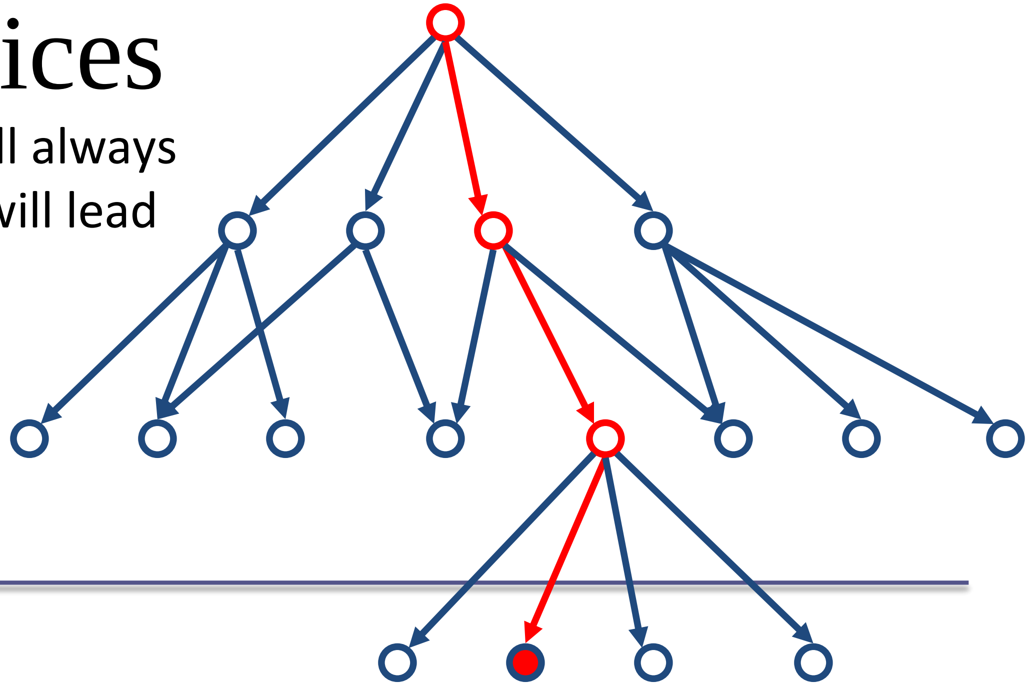
- Nondeterministic algorithm

```
j = choice (1:n)    // choice is always correct
if (A[j] == x) then { print j; return true }
return false
    // since j must be correct if A[] has x
```

Oracle as Path Finder

Another way of thinking of an Oracle is in terms of navigating through a search graph of choices

at each choice point the Oracle will always choose the correct branch which will lead eventually to the correct solution



P and NP and Determinism

P is the set of decision problems that can be solved by a **deterministic** algorithm in **polynomial** time

NP is the set of decision problems that can be solved by a **nondeterministic** algorithm in **polynomial** time

Deterministic algorithms are a 'simple' case of nondeterministic algorithms, and so **P ⊆ NP**

- **simple** because no oracle is needed

No one knows how to write a nondeterministic polynomial algorithm, since it depends on a magical oracle always making correct guesses.

So in the real world, the solving part of every NP problem has exponential running time, not polynomial.

But if someone discovered/implemented an oracle, then NP would becomes polynomial, and **P = NP**

this is **the biggest unsolved question in computing**

5. P = NP : the BIG Question

On balance it is probably **false** (i.e. $P \neq NP$)

Why?

you can't engineer perfect luck

and

generating a solution (along with evidence) is harder than checking a possible solution

One Possible Oracle

One possible way to write an oracle would be to use a parallel computer to create an **infinite** number of processes/threads

- have one thread work on each possible guess at the same time
- the "infinite" part is the problem

Perhaps solvable by using quantum computing or DNA computing (?)

- (but probably not)

Clay Millennium Prize Problems

each one is worth US $1,000,000

P = NP Problem

Riemann Hypothesis

Birch and Swinnerton-Dyer Conjecture

Navier-Stokes Problem

Poincaré Conjecture ☑

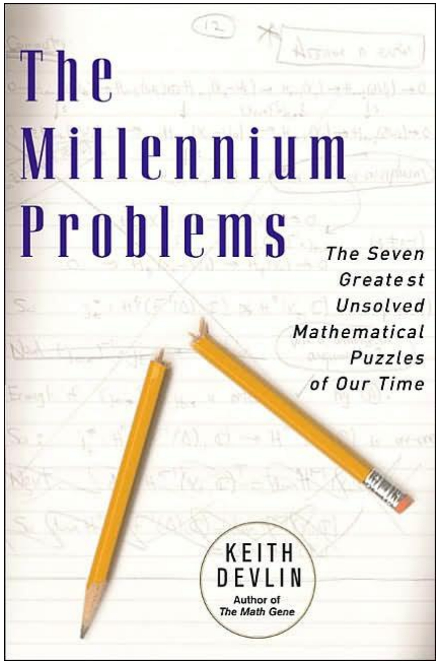
solved by Grigori Perelman; declined the award in 2010

Hodge Conjecture

Yang-Mills Theory

Two Clay Institute videos

http://claymath.msri.org/tate2000.mov
Riemann hypothesis, Birch and Swinnerton-Dyer Conjecture, **P vs NP**
http://claymath.msri.org/atiyah2000.mov
Poincaré conjecture, Hodge conjecture,
Quantum Yang-Mills problem,
Navier-Stokes problem



A popular maths book about the problems:

The Millennium Problem
Keith J. Devlin
Basic Book, 2003
(17 min video): http://profkeithdevlin.com/Movies/MillenniumProblems.mp4

6. NP-Complete

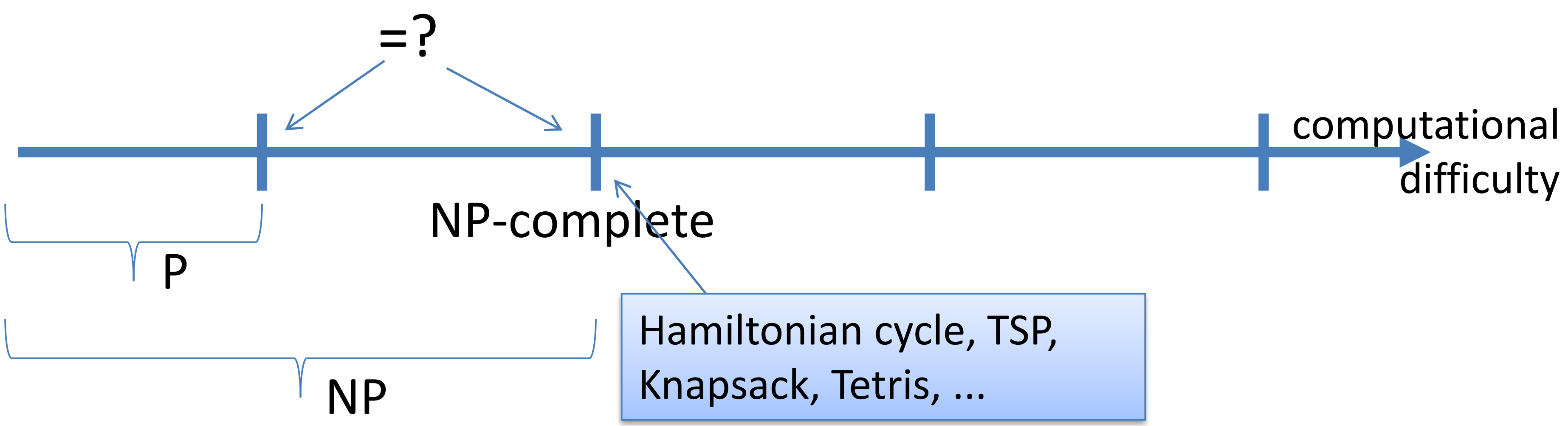
A problem is in the NP-Complete (NPC) set if it is in NP and is **at least as hard as other problems in NP**.

same hardness or harder

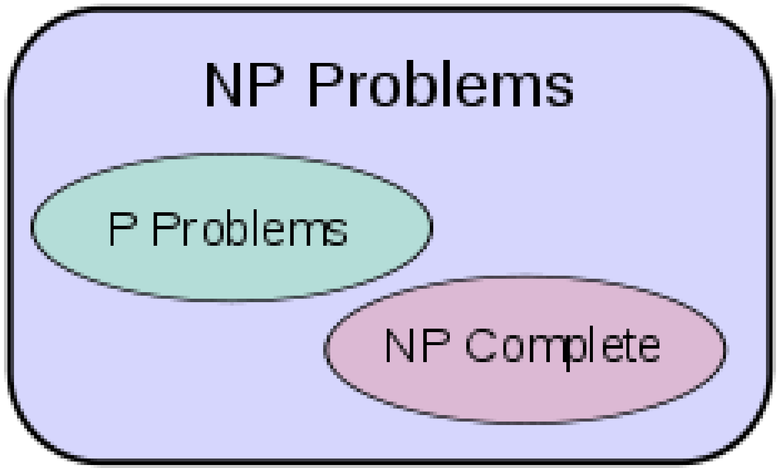
Some NPC problems:

Hamiltonian cycle
Path-finding (Traveling salesman)
Cliques
Map (graph, vertex) coloring
subset sum
Knapsack problem
Scheduling
Many, many more ...

Computational Difficulty



Drawing P, NP, NPC as Sets



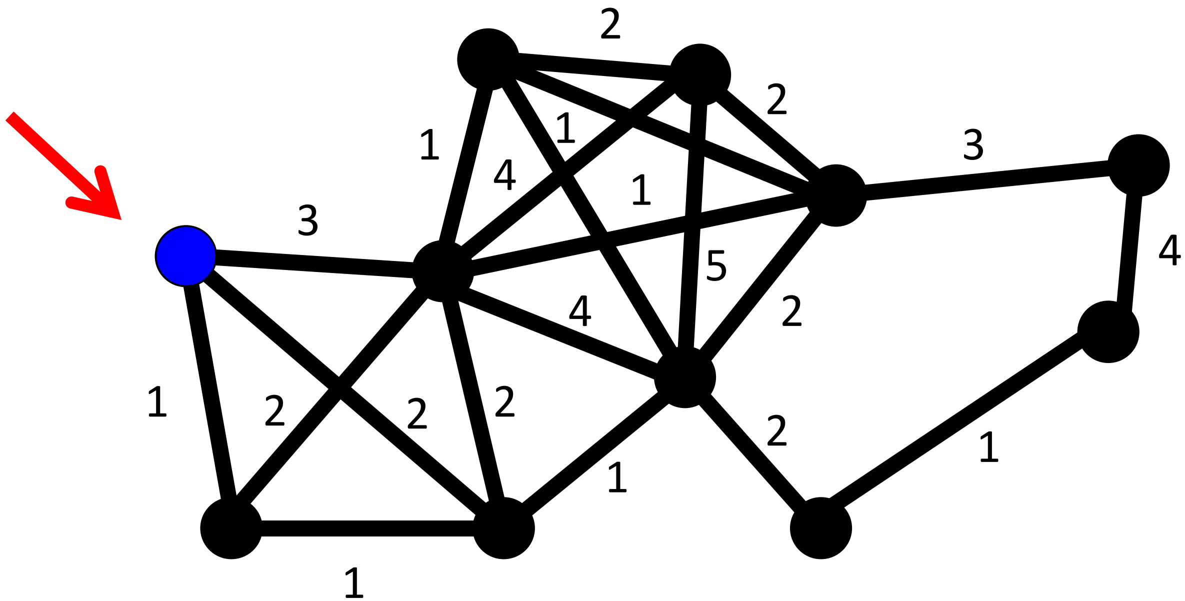
P ≠ NP

or

all problems together
in one P set

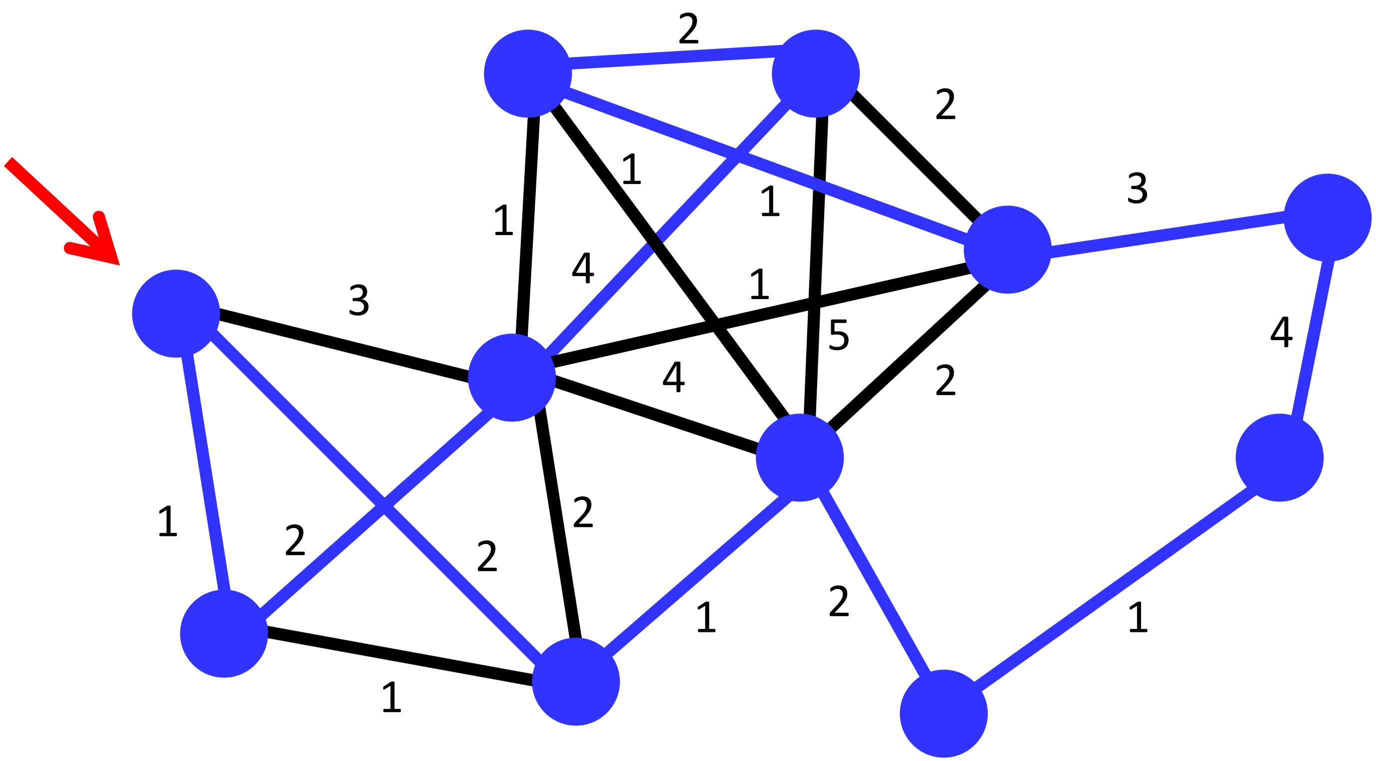
P = NP (= NPC)

6.1. Travelling Salesman (TSP)



The salesperson must make a minimum cost circuit, visiting each city exactly once.

A solution (distance = 23):

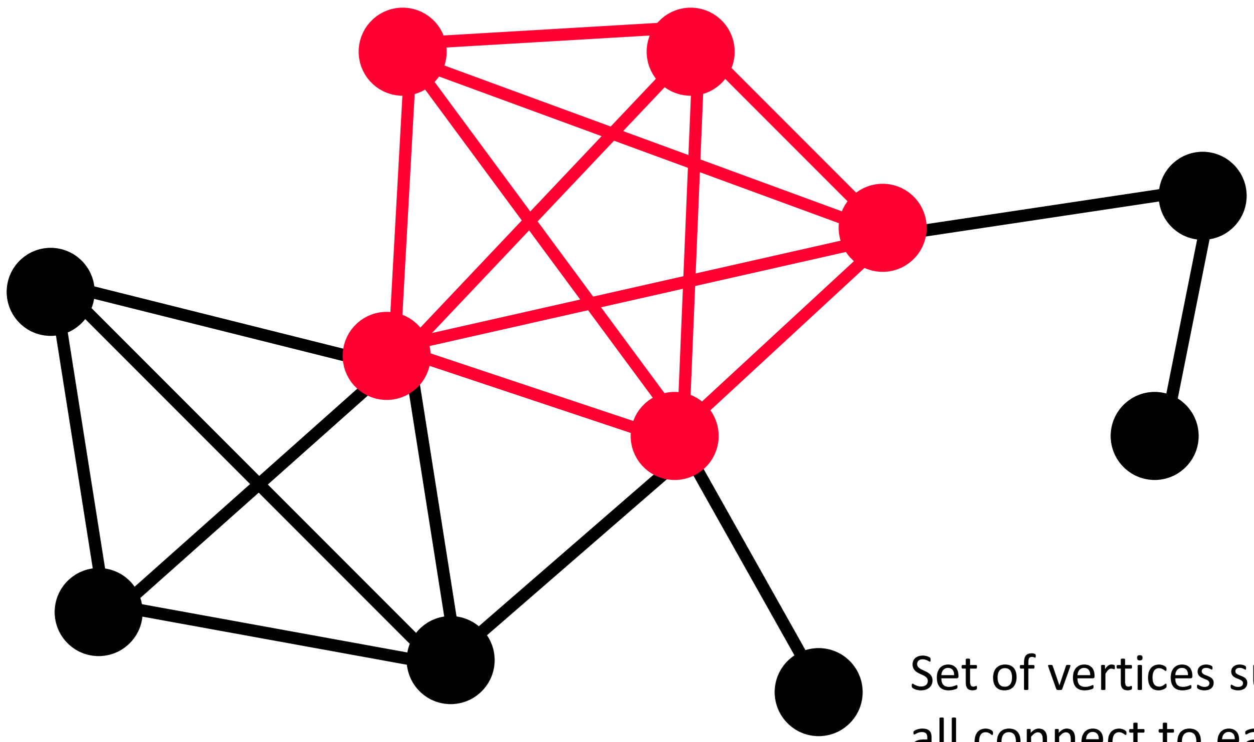


TSPs (and variants) have enormous practical importance

Lots of research into good approximation algorithms

Recently made famous as a DNA computing problem

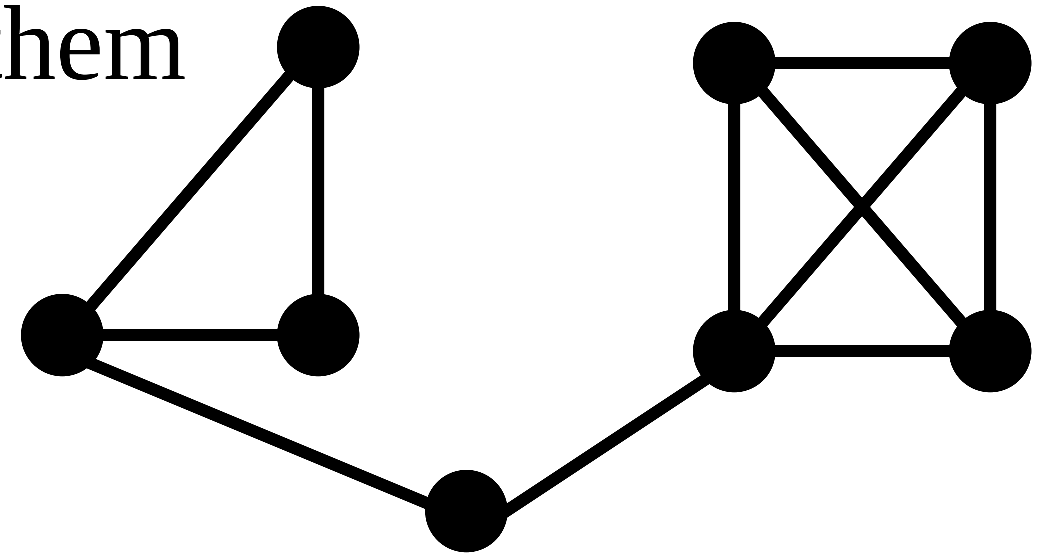
6.2. 5-Clique



Set of vertices such that they all connect to each other.

K-Cliques

A K-clique is a set of K nodes connected by all the possible K(K-1)/2 edges between them



This graph contains a 3-clique and a 4-clique

K-Cliques

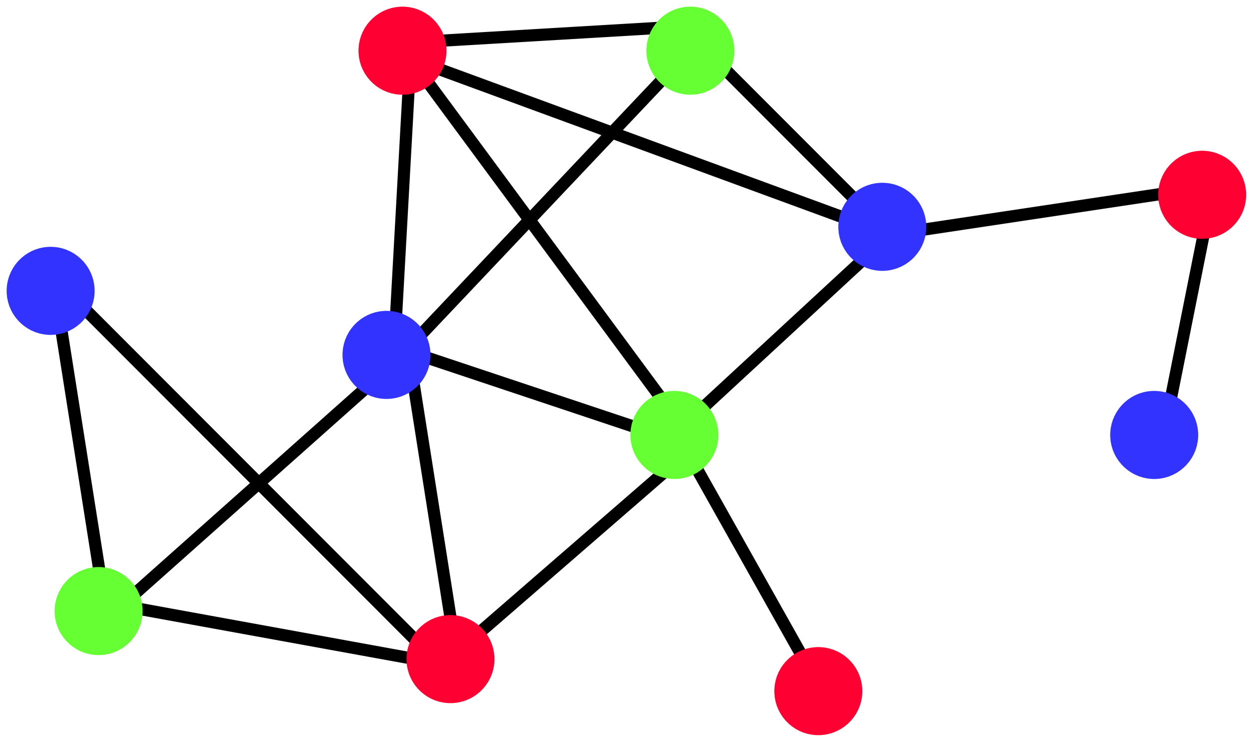
Given: (G, k)

Decision problem: Does G contain a k-clique?

Brute Force

Try out all {n choose k} possible locations for the k clique

6.3. Map Coloring



Can a graph be colored with *k* colors such that no adjacent vertices are the same color?

6.4. Subset Sum

Given a set of integers, does there exist a subset that adds up to some value T?

Example: given the set $\{-7, -3, -2, 5, 8\}$, is there a non-empty subset that adds up to 0.
Yes: $\{-3, -2, 5\}$

Can also be thought of as a special case of the knapsack problem

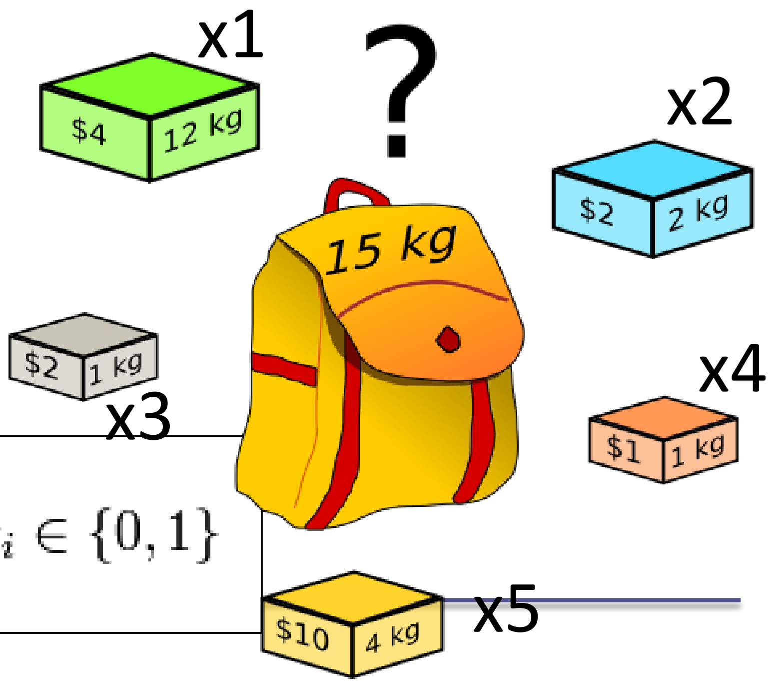
6.5. The Knapsack Problem

The *0-1 knapsack problem*:

A tourist must choose among n items, where the ith item is worth v_i dollars and weighs w_i pounds

Carry at most W kg, but make the value maximum

Example : 5 items: x1 – x5,
values = (4, 2, 2, 1, 10}
weights = {12, 2, 1, 1, 4},
W = 15, maximize total value



- Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leqslant W,\qquad x_i \in \{0,1\}$

Pseudocode

```
def knapsack(items, maxweight):
    best = {}
    bestvalue = 0
    for s in allPossibleSubsets(items):
      value = 0
      weight = 0
      for item in s:
        value += item.value
        weight += item.weight
      if weight <= maxweight:
        if value > bestvalue:
            best = s
            bestvalue = value
    return best
```

n items

2^n subsets

$O(n)$ for each one

Running time is $O(n \cdot 2^n)$

The ***bounded knapsack*** problem removes the restriction that there is only one of each item, but restricts the number of copies of each kind of item

The ***fractional knapsack*** problem:

- the thief can take fractions of items
- e.g. the thief is presented with *buckets* of gold dust, sugar, spices, flour, etc.

6.6. Class Scheduling Problem

We have N teachers with certain time restrictions, and M classes to be scheduled. Can we:

Schedule all the classes?

Make sure that no two teachers teach the same class at the same time?

No teacher is scheduled to teach two classes at once?

6.7. Why study NPC?

Most interesting problems are NPC.

If you can show that a problem is NPC then:

You can spend your time developing an **approximation algorithm** rather than searching for a fast algorithm that solves the problem exactly

or you can look for a special case of the problem, which is polynomial

More NPC Problems

There are over 3000 known NPC problems.

Many of the major ones are listed at:

http://en.wikipedia.org/wiki/List_of_np_complete_problems

The list is divided into several categories:

Graph theory, Network design, Sets and partitions, Storage and retrieval, Sequencing and scheduling, Mathematical programming, Algebra and number theory, Games and puzzles, and Logic

Another way of classifying NPC problems:

Packing problems: SET-PACKING, INDEPENDENT SET

Covering problems: SET-COVER, VERTEX-COVER

Constraint satisfaction problems: SAT, 3-SAT

Sequencing problems: HAMILTONIAN-CYCLE, TSP

Partitioning problems: 3D-MATCHING 3-COLOR

Numerical problems: SUBSET-SUM, KNAPSACK

Hard vs Easy

There's often not much difference in appearance between hard problems and

Hard Problems (NP-Complete)	Easy Problems (in P)
SAT, 3SAT	2SAT
Traveling Salesman Problem	Minimum Spanning Tree
3D Matching	Bipartite Matching
Knapsack	Fractional Knapsack

7. The Circuit-SAT Problem

Almost the first problem to be proved NPC

Cook 1971, Levin 1973

actually Cook proved **SAT** was in NPC, but **Circuit-SAT** is almost the same, and is the example used in CLRS

What did Cook do?

Remember: a problem is in NPC if

1. it is in NP and
2. it is at least as hard as other problems in NP

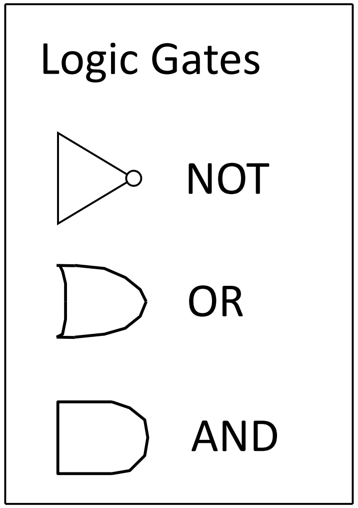
same hardness or harder

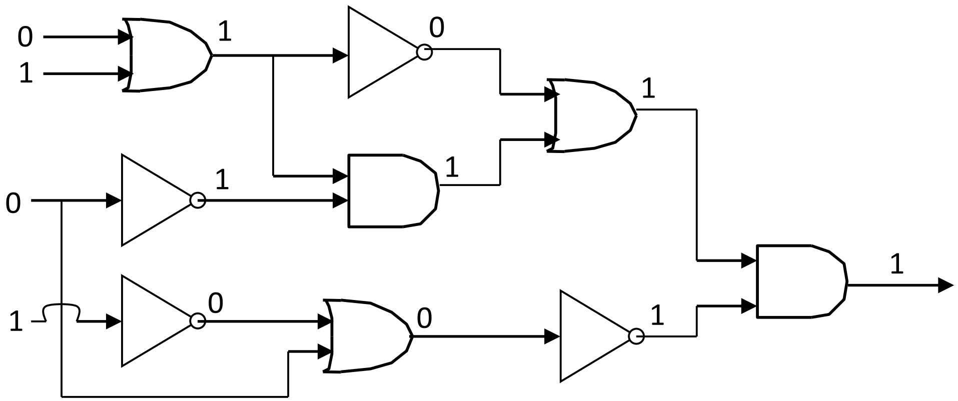
What is Circuit-SAT?

The **Circuit-Satisfiability Problem** (Circuit-SAT) is a circuit composed of AND, OR, and NOT gates.

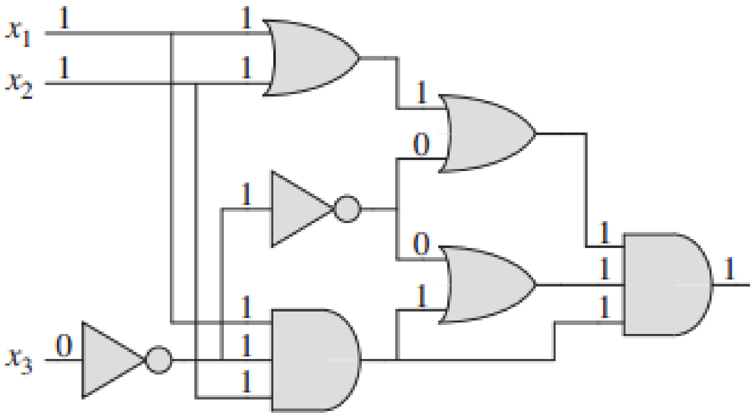
A circuit is satisfiable if there exists a set of boolean input values that makes the output of the circuit equal to 1.

A Circuit

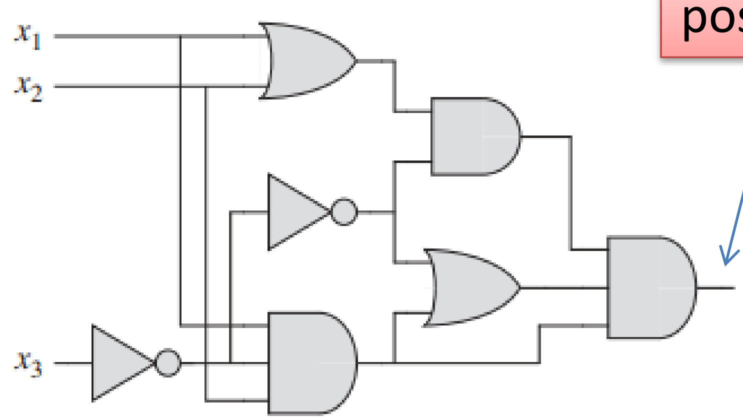




Satisfiablity Examples



(a) Satisfiable



(a) Unsatisfiable

No 1 is possible

Circuit (a) is satisfiable since
<x1, x2, x3> = <1, 1, 0> makes the output 1.

Part 1. Circuit-SAT is NP

If there are n inputs, then there are 2^n possible assignments that need to be tried in order to find a solution

solving is $O(2^n)$, exponential running time

Testing a solution is done by evaluating the circuit's logical expression which can be done in linear running time

testing is $O(n)$, polynomial running time

Part 2. Circuit-SAT is NPC

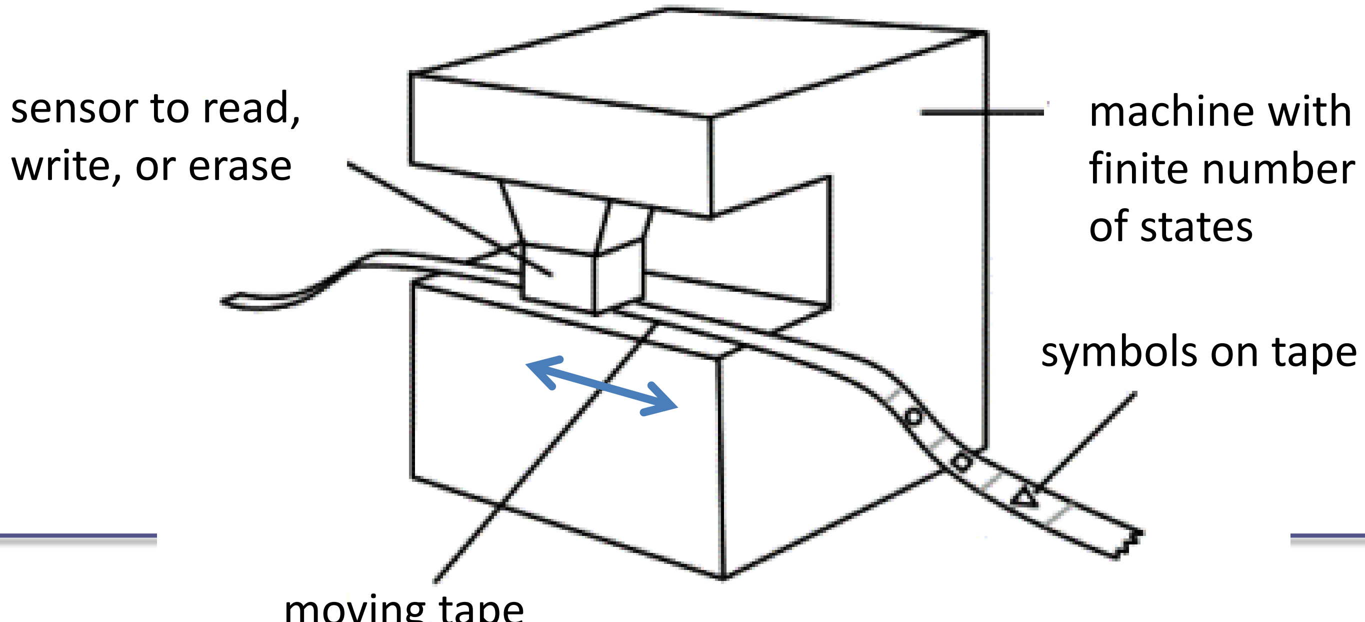
Cook's Theorem

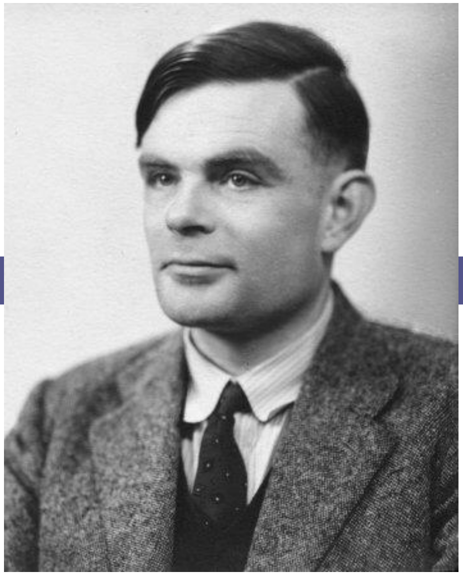
it proves (Circuit-) SAT is NPC from first principles, based on its implementation of the **Turing machine**

All other problems have been shown to be NPC by **reducing** (transforming) Circuit-SAT into those problems (either directly or indirectly)

What's a Turing Machine?

A Turing machine is a theoretical device composed of an infinitely long tape with printed symbols representing instructions.
The tape can move backwards and forwards in the machine, which can read the instructions and write results back onto the tape.





Invented in 1936 by Alan Turing.
Despite its simplicity, a Turing machine can simulate the logic of **any** computing machine or algorithm.

i.e. a function is algorithmically computable if and only if it is computable by a Turing machine
called the **Church–Turing thesis** (hypothesis)

Algorithms & Turing Machines

Any algorithm can be implemented as a Turing machine

this is how "computable" is formally defined

Every problem in NP can be transformed into a program running on a (non-deterministic) Turing machine.

(Church-Turing)

"non-deterministic" means that the finite state machine used by the Turing machine is non-deterministic

Back to Cook's Theorem

Cook's Theorem argues that every feature of the Turing machine, including how instructions are executed, can be written as logical formulas in a satisfiability (SAT) problem. **(cook 1)**

So any a program executing on a Turing machine can be transformed into a logical formula of satisfiability. **(cook 2)**

Combining **Church-Turing** and **cook 2** means that every problem in NP can be transformed into a logical formula of satisfiability. **(cook 3)**

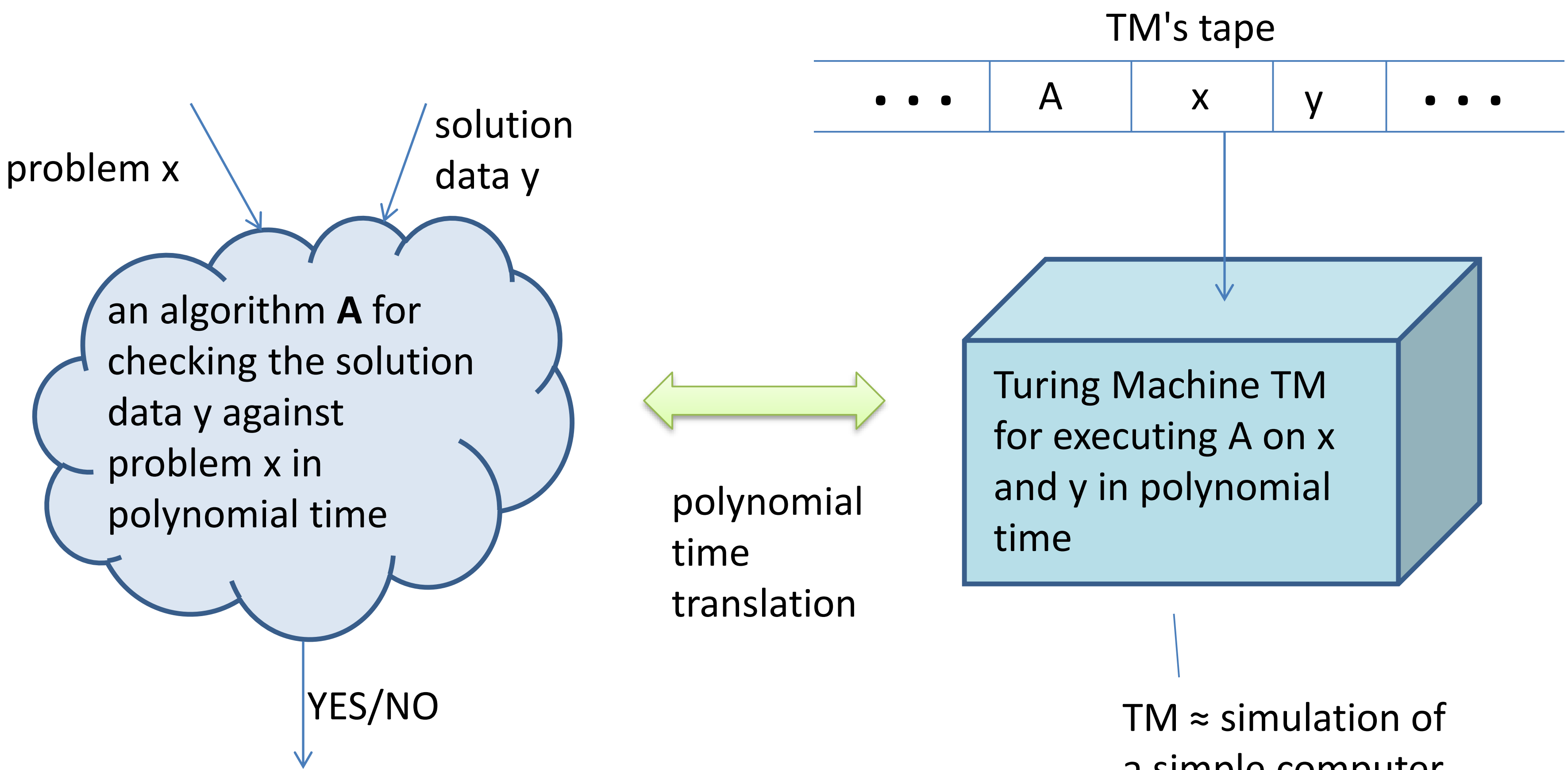
This means that the SAT problem is at least as hard as any problem in NP (**cook 4**)

By definition, a problem is in NPC if it is in NP and is at least as hard as other problems in NP

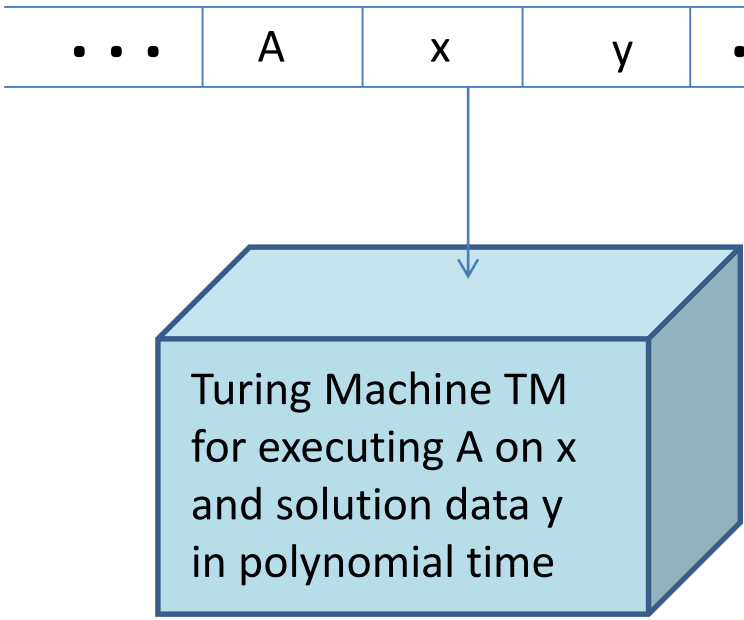
so (Circuit-) SAT is in NPC

same hardness or harder

Cook's Theorem as Pictures



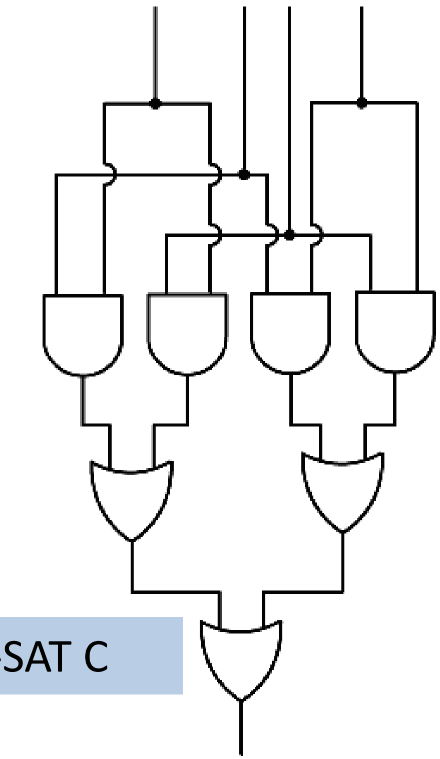
TM's tape



TM ≈ simulation of
a simple computer

polynomial
time
translation

binary inputs (...101011000...)
representing A, x and y



Circuit-SAT C

C ≈ a simple
real computer

output of 1 means A
has checked the y against x

Informal Explanation

The Turing Machine runs the program **A** (run-time software) against the input **x** (program) and the solution data **y** (program's input).

This Turing machine can be implemented as a Circuit-SAT 'computer' made up of AND, OR, and NOT gates.

Since Circuit-SAT can implement any problem (i.e. the A, x, and y) then it is as least as hard (complex) as any other problem, and so is NP-Complete.

8. NPC and Reducability

A formal definition of NPC requires the **reducibility** of one problem into another.

If we can reduce (transform) a problem S into a problem L in polynomial time, then S is **not polynomial factor harder** than L

this is written as : **S ≤p L**

Harder means **"bigger running time"**

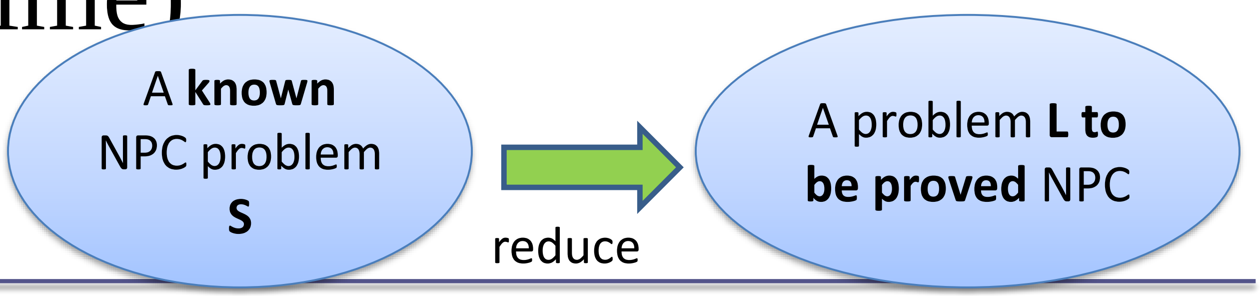
Is a Problem 'L' in NPC?

First show that the problem L is in NP

- show that it can be solved in exponential time
- Show that it can be checked in polynomial time

Show how to reduce (transform) an **existing NPC problem S** into L (in polynomial time)

- S is NPC
- $S \le_p L$
- Then L is NPC



This reduction means that the existing NPC problem (S) is no harder than the new problem L

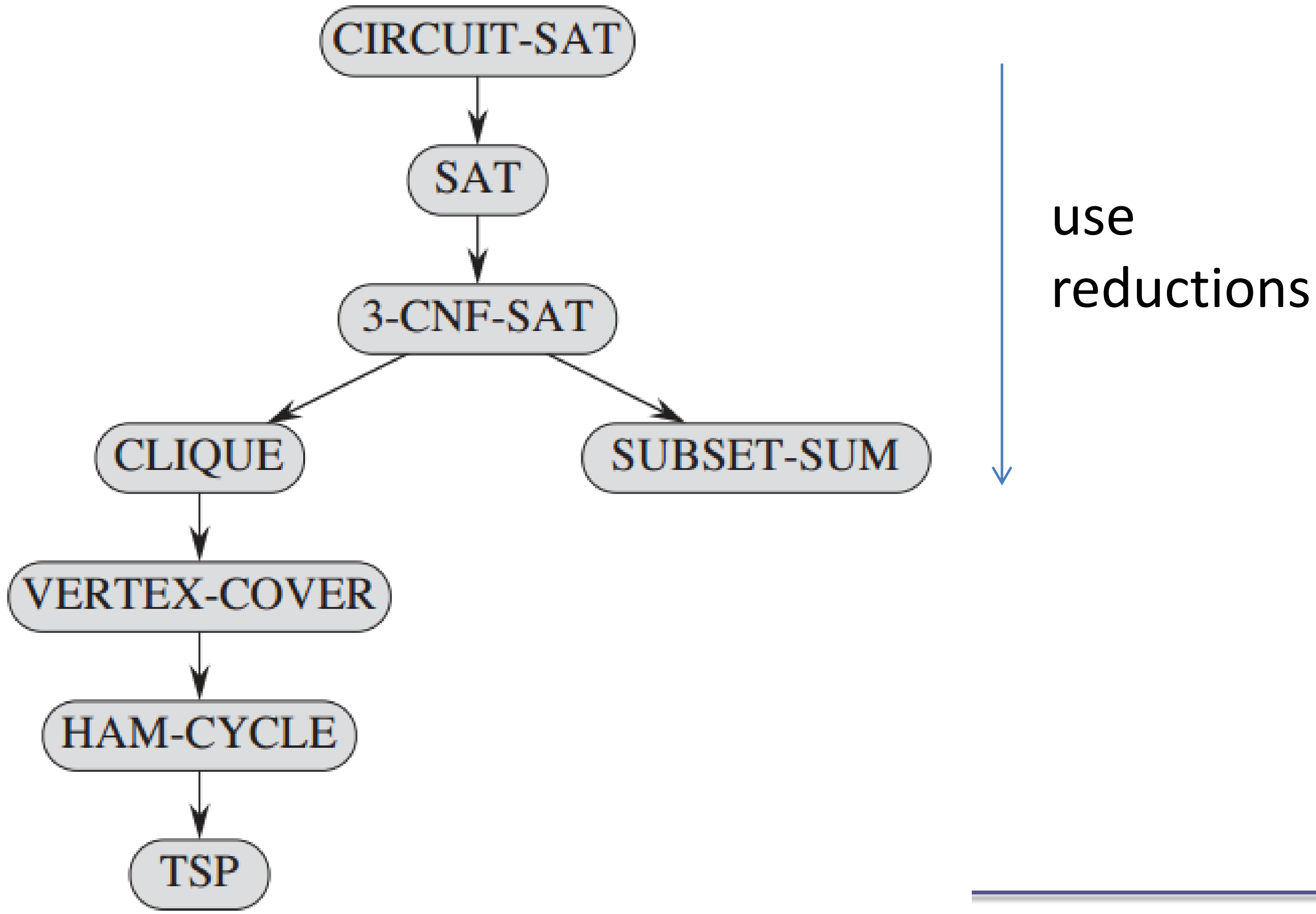
harder means "bigger running time"

This hardness 'link' between NPC problems means that all NPC problems have a similar (or smaller) hardness, and that all NP problems are less hard.

If a researcher proves that **any** NPC problem can be solved in polynomial time, then **all** problems in NPC and NP must also have polynomial running time algorithms.

in other words **P = NP**

Proving Problems are NPC



8.1. The Satisfiability Problem (SAT)

Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?

Example:

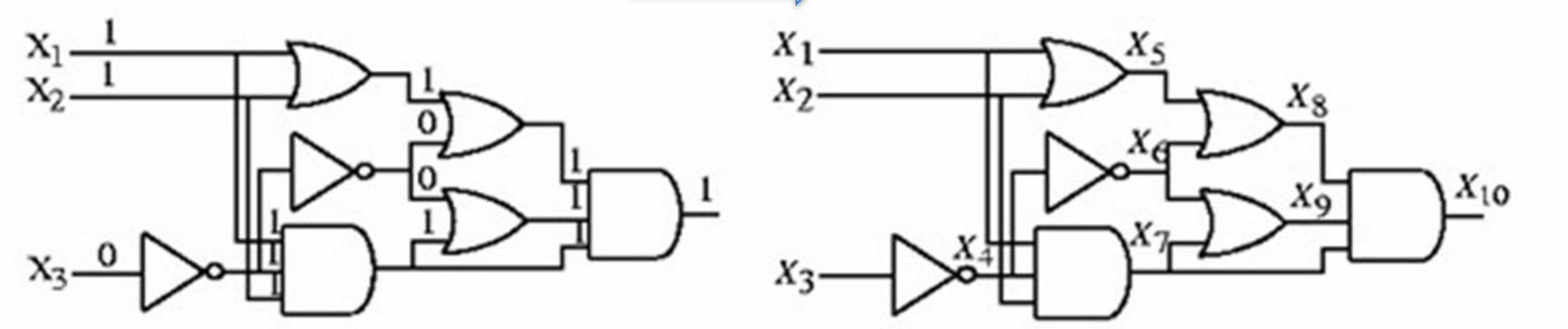
$$(x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

A satisfying truth assignment:

<x1, x2, x3, x4 > = <0, 0, 1, 1>

Reduction Graphically

with additional 'gate' variables



satisfiable circuit

reduction from Circuit–SAT to formula SAT

ϕ = x10 $\wedge$ (x4 $\leftrightarrow \neg$ x3) $\wedge$ (x5 $\leftrightarrow$(x1$\vee$ x2)) $\wedge$ (x6 $\leftrightarrow \neg$ x4) $\wedge$
(x7 $\leftrightarrow$(x1$\wedge$ x2 $\wedge$ x4)) $\wedge$ (x8 $\leftrightarrow$(x5 $\vee$ x6)) $\wedge$(x9 $\leftrightarrow$(x6 $\vee$ x7)) $\wedge$
(x10 $\leftrightarrow$ (x7 $\wedge$ x8 $\wedge$ x9))

Is SAT a NPC Problem?

SAT is an NP problem.

Prove that Circuit-SAT $\leq_P$ SAT

remember the left-hand-side must be an existing NPC problem

8.2. The 3-SAT Problem

3SAT: the Satisfiability of boolean formulas in
3-conjunctive normal form (3-CNF).

3SAT is NP
Show that 3SAT is NPC by **proving SAT $\leq_P$ 3SAT**.

Conjunctive Normal Form (CNF)

A Boolean formula is in conjunctive normal form if it is an AND of clauses, each of which is an OR of literals

e.g. $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$

3-CNF: each clause has exactly 3 distinct literals

e.g. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$

Note: the formula is true if at least one literal in each clause is true

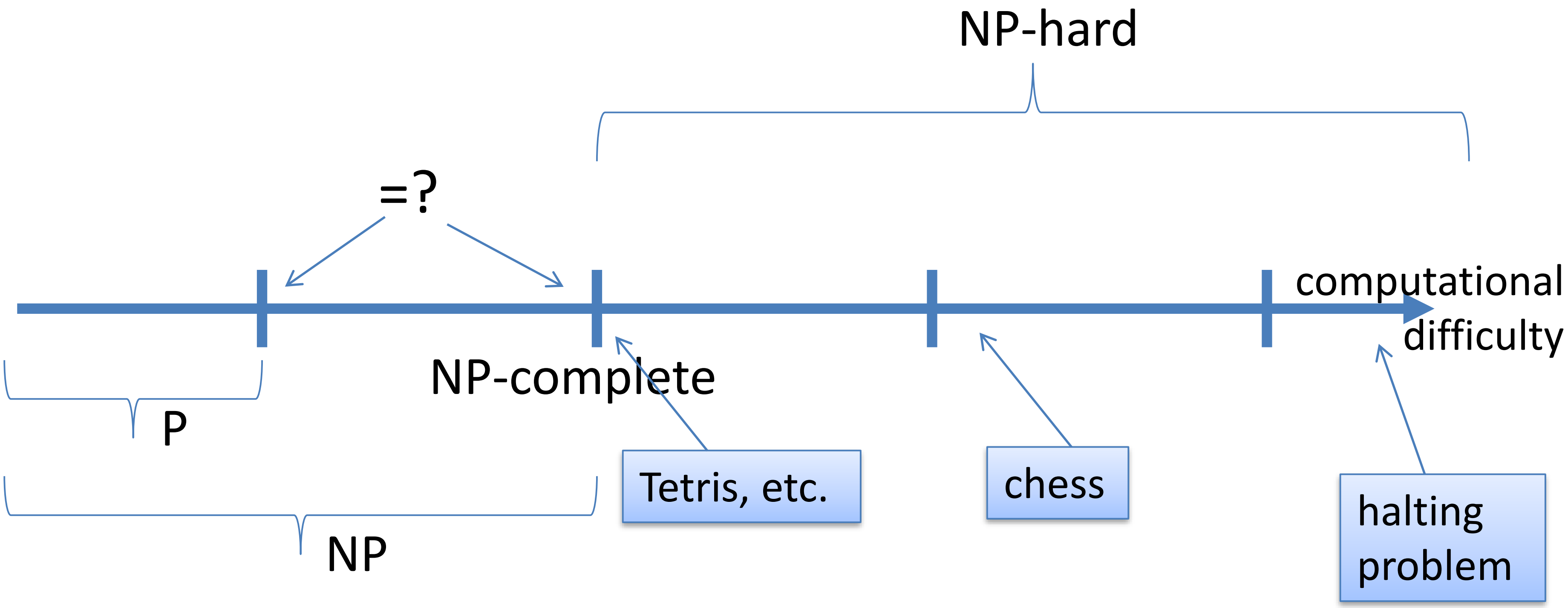
9. NP-Hard

A NP-Hard problem is **at least as hard** as the hardest problems in NP (i.e. NPC problems).

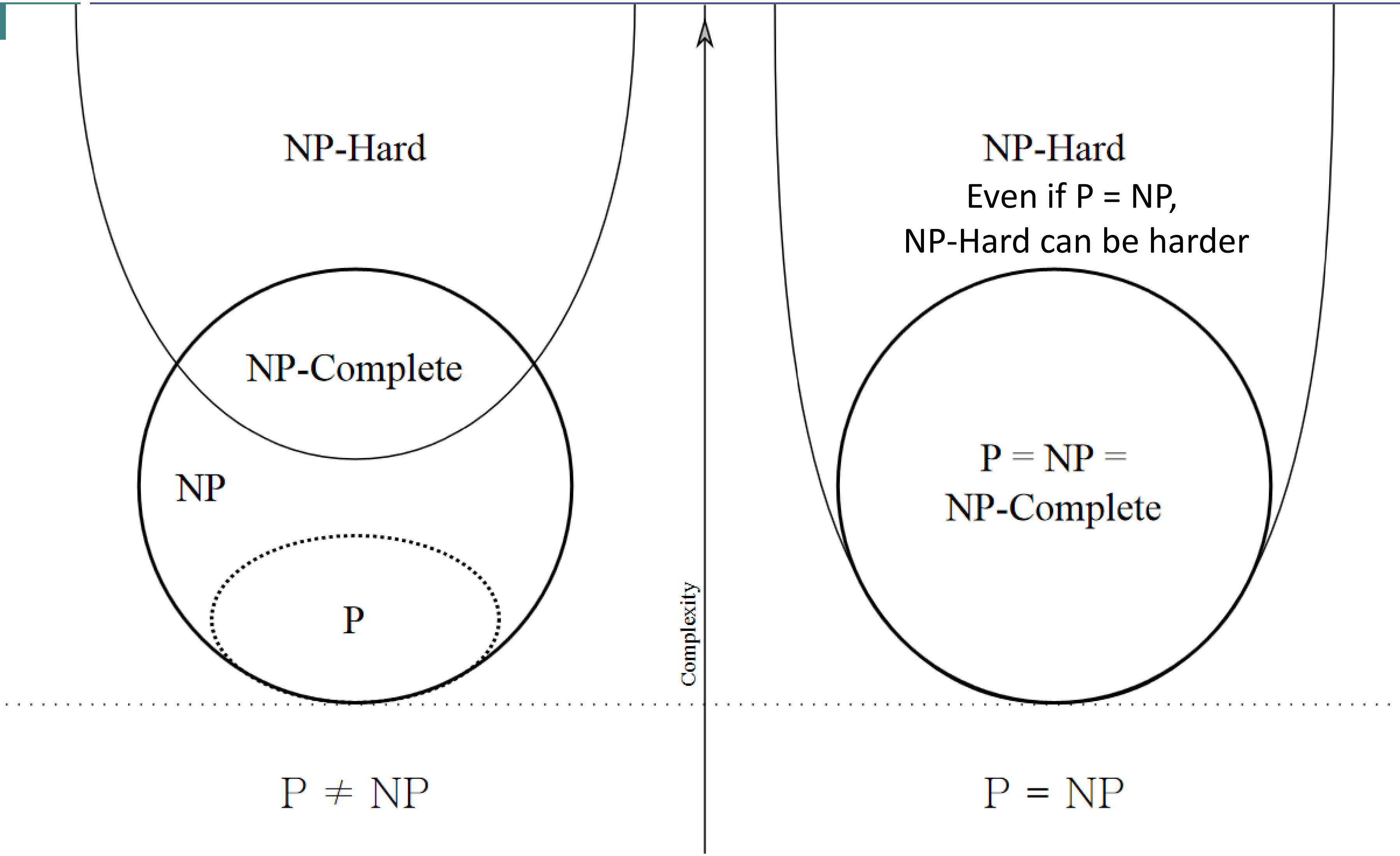
A NP-Hard problem may **not have a polynomial time checking** algorithm

that makes those NP-hard problems harder to solve than NPC problems
e.g. the halting problem: "given a program and its input, will it run forever?"
e.g. chess: "will black or white win from a given board configuration?"

Computational Difficulty



Drawing P, NP, etc. as Sets



Coping with NPC/NP-Hard Problems

Approximation algorithms:
guarantee to be a fixed percentage away from the optimum

Pseudo-polynomial time algorithms:
e.g., dynamic programming for the 0-1 Knapsack problem

Probabilistic algorithms:
assume some probabilistic distribution of the data

Randomized algorithms:
use randomness to get a faster running time, and allow the algorithm to fail with some small probability
e.g. Monte Carlo method, Genetic algorithms

Restriction: work on special cases of the original problem which run better

Exponential algorithms / exhaustive search:

feasible only when the problem size is small.

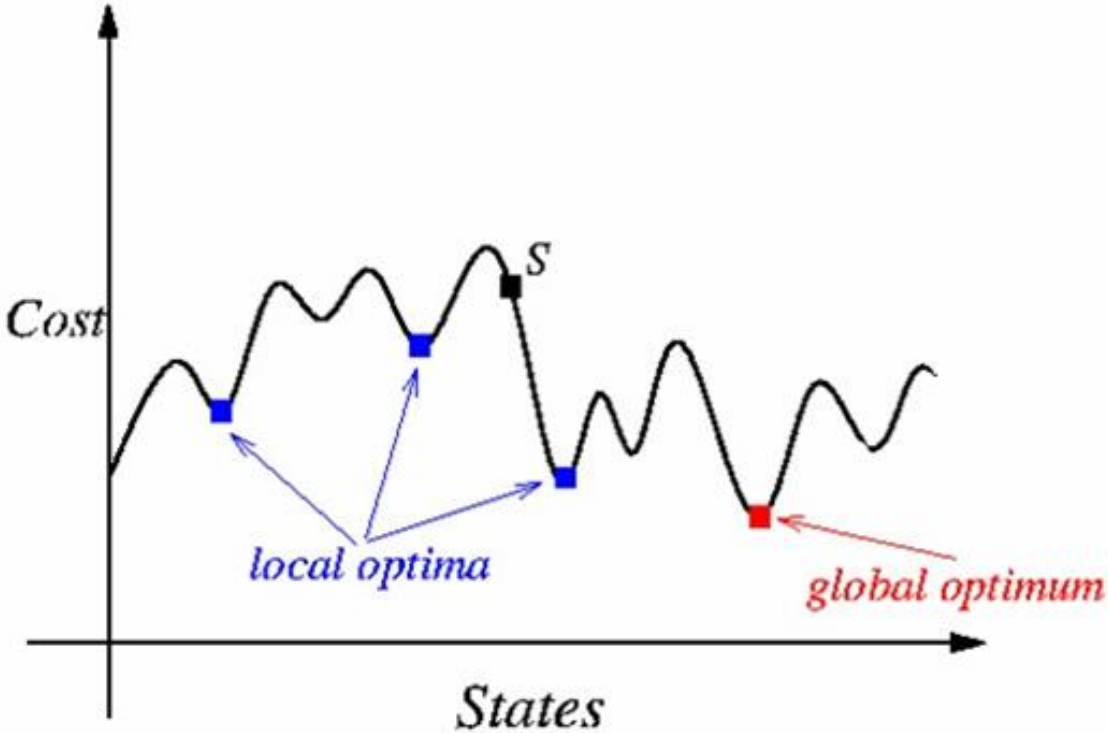
Local search:

simulated annealing (hill climbing)

Heuristics

use "rules of thumb" that have no formal guarantee of performance

Simulated Annealing



Use probabilities to jump around in a large search space looking for a good enough solution called a **local optimum**.
By contrast, the actual best solution is the **global optimum**.

10. Approximation Algorithms

An approximation algorithm is one that returns a near-best solution.

The quality of the approximation can be measured by comparing the 'cost' of the best solution against the approximate one

Cost can be any aspect of the algorithm

e.g. number of nodes, edge distance, running time, space

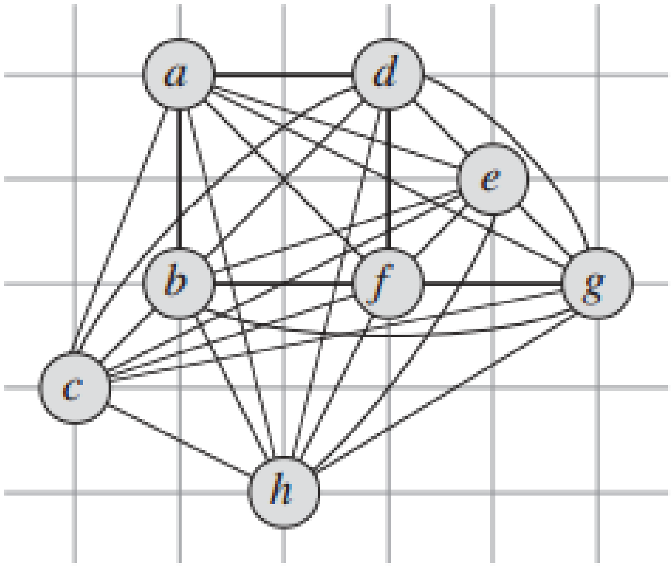
10.1. Approx. Algorithm for TSP

Approx-TSP-Tour(G)

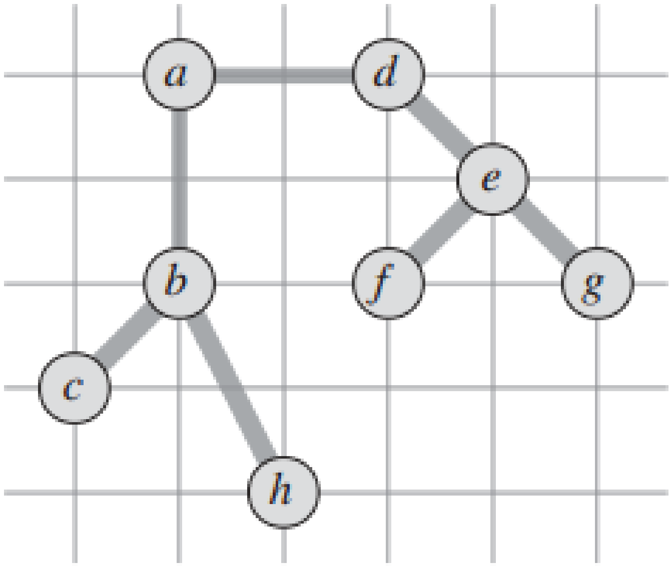
1. select a vertex from G to be a “root”
2. grow a **minimum spanning tree** (MST) for G from root r using Prim()
3. create a list L of vertices visited in a preorder tree walk over the MST
4. return Hamiltonian cycle using L for its order

Approx-TSP Example

an integer unit grid

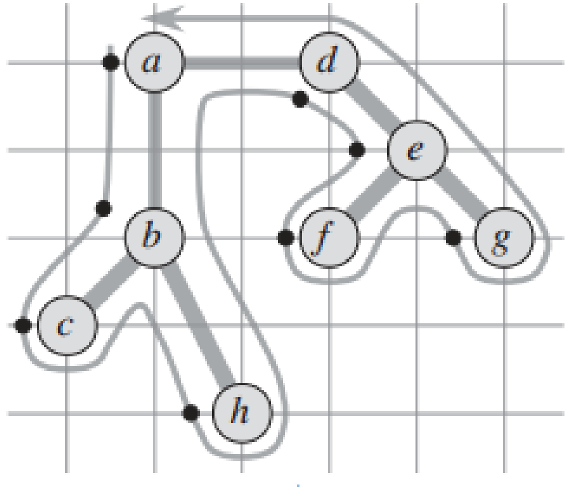


(a) Undirected Graph



(b) MST created with Prim()

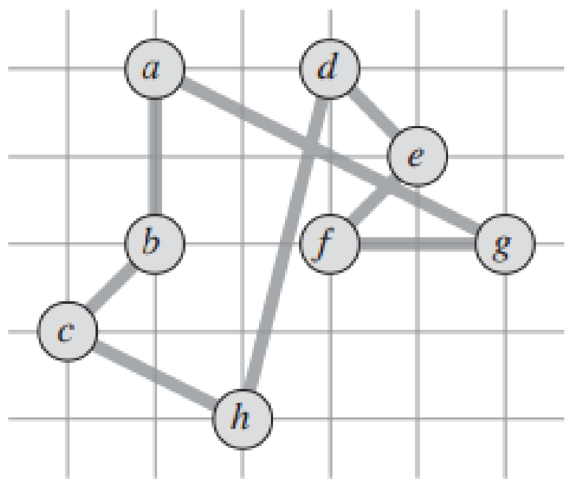
(c) Preorder walk over the MST



Preorder walk: a, b, c, b, h, b, a, d, e, f, e, g, e, d, a

A preorder walk that lists a vertex only when it is first encountered, as indicated by the dot next to each vertex: a, b, c, h, d, e, f, g

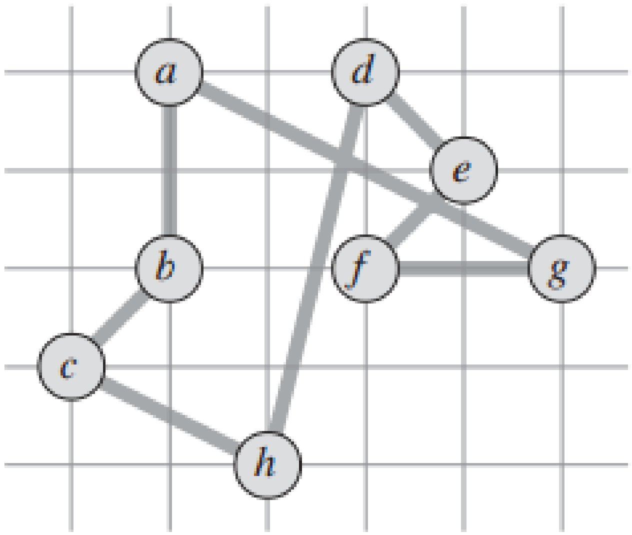
(d) Tour using the preorder MST.

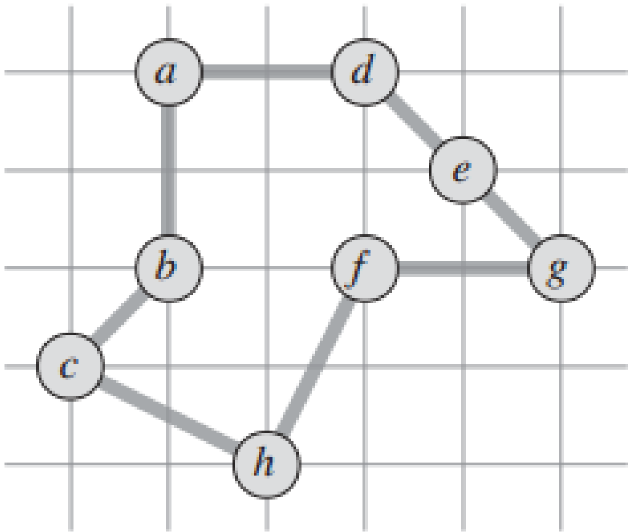


Cost of tour: ~19.074
(by measuring squares)

What are we Comparing?

We want to compare an optimal tour with a tour using a preorder MST. For this example:





Tour using the preorder MST

Cost of tour: ~19.074

Optimal Tour

Cost of tour: ~14.715

11. Decidable Problems

Decidable problems are often split into three categories:

- Tractable problems (polynomial)
- NPC problems
- Intractable problems (exponential)
  - sometimes grouped in the set **EXP**

All of the above have algorithmic solutions, even if they have impractical running times.

- All of these are placed in a group called R
- R = {**set of problems solvable in finite time**}

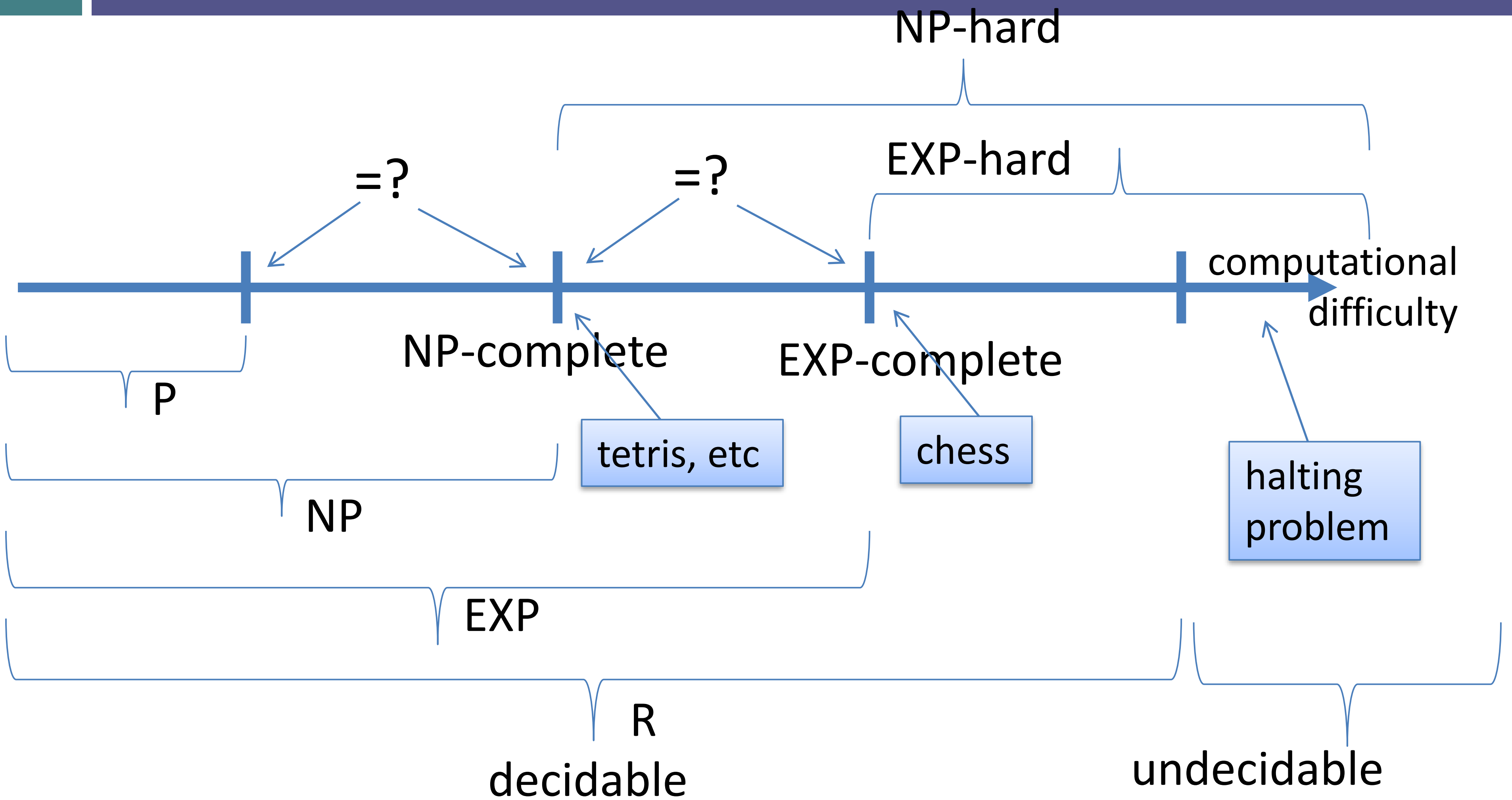
12. Undecidable Problems

An undecidable problem is one where it is impossible to construct an algorithm that gives a correct yes-or-no answer.

Instead:

- the answer may be wrong, or
- no answer may appear, no matter how long you wait

Computational Difficulty



12.1. The Halting Problem

Decide whether a given program with some input finishes, or runs forever.

The halting problem is **undecidable**.

- an algorithm to solve the halting problem for all possible program-input pairs does not exist

Isn't detecting halting easy?

No. It can be very difficult.
e.g. Does this code finish for every possible input?

```
read n;
while(n != 1) {
  if (even(n))
    n = n/2;
  else
    n = 3*n + 1;
  print n;
}
```

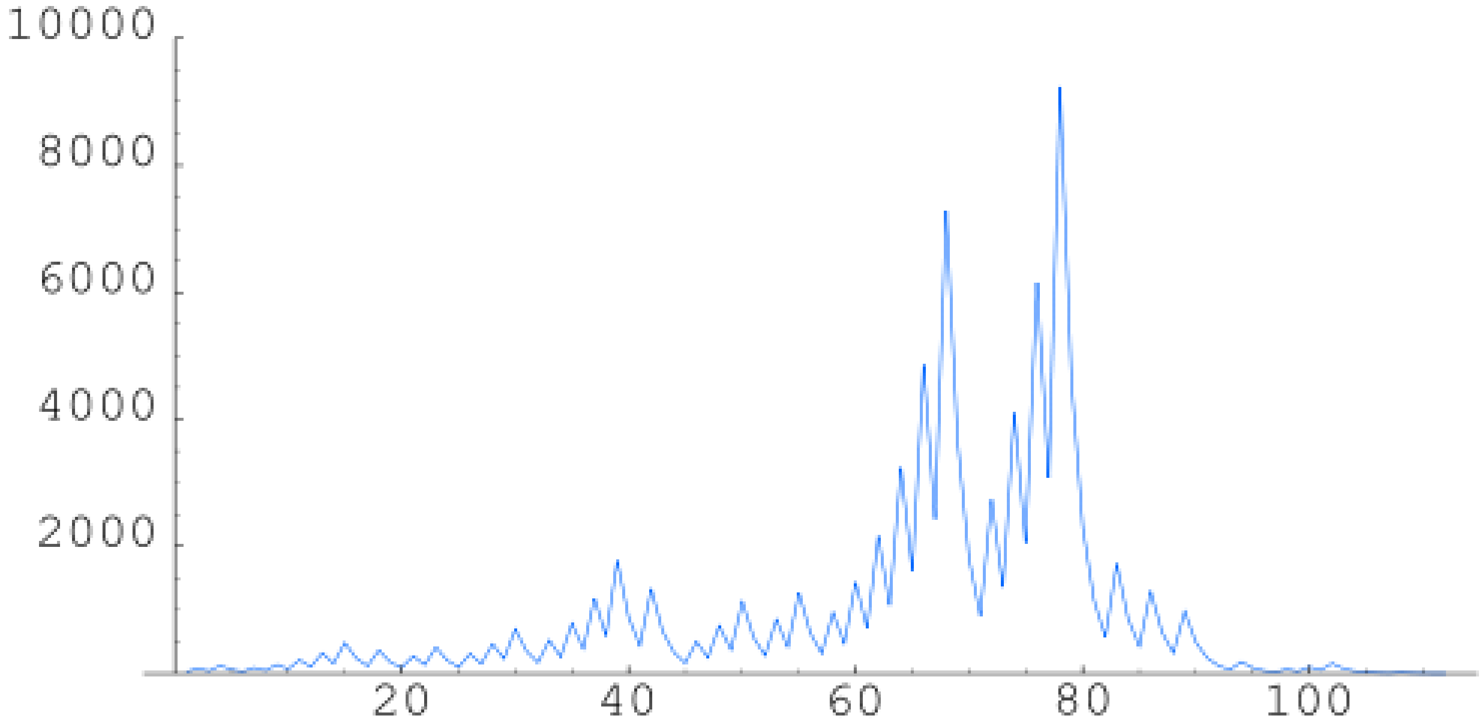
The maths behind this function is known as the **Collatz conjecture** (unsolved problem).

The conjecture: no matter what positive number you start with, the function will eventually reach 1.

Examples

start with n = 6, the sequence is 6, 3, 10, 5, 16, 8, 4, 2, 1.
n = 11, sequence is 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
called **hailstone sequences**

The sequence for n = 27, loops 111 times, climbing to over 9000 before descending to 1

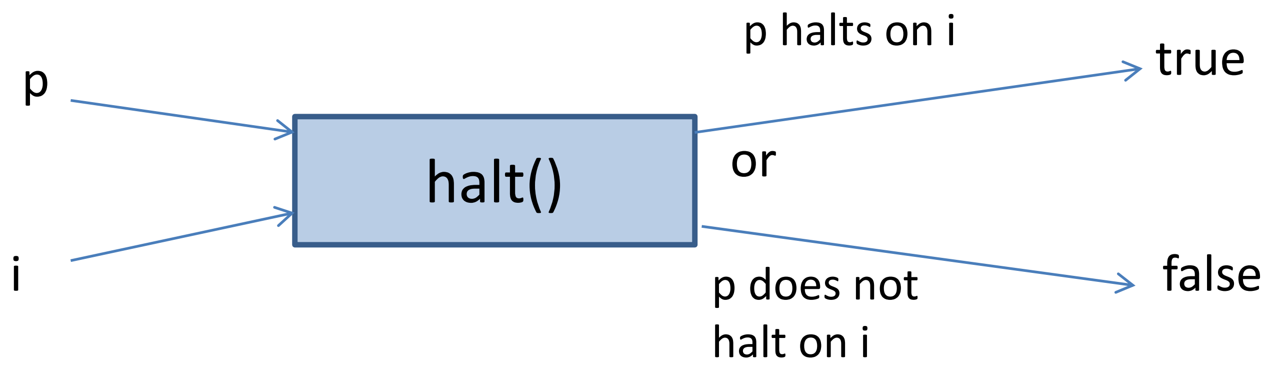


Why the Halting Problem is Undecidable

Let's assume the **halt(String p, String i)** function implements the halting problem.

It takes two inputs, an encoded representation of a program as text (p) and the program's input data (i).

halt() return true or false depending on whether p halts when executing the i input.



Examples

```
halts("def sqr(x): return x * x", "sqr(4)")
```

returns **true**

```
halts("def fac(n):
        if (n == 0) return 1
         else return n * fac(n-1)", "fac(10)")
```

returns **true**

```
halts("def foo(n):
         while (true) { n = n + 1}", "foo(4)")
```

returns **false**

Now for Trouble

Let's create a 2nd function called **trouble(String q)**.

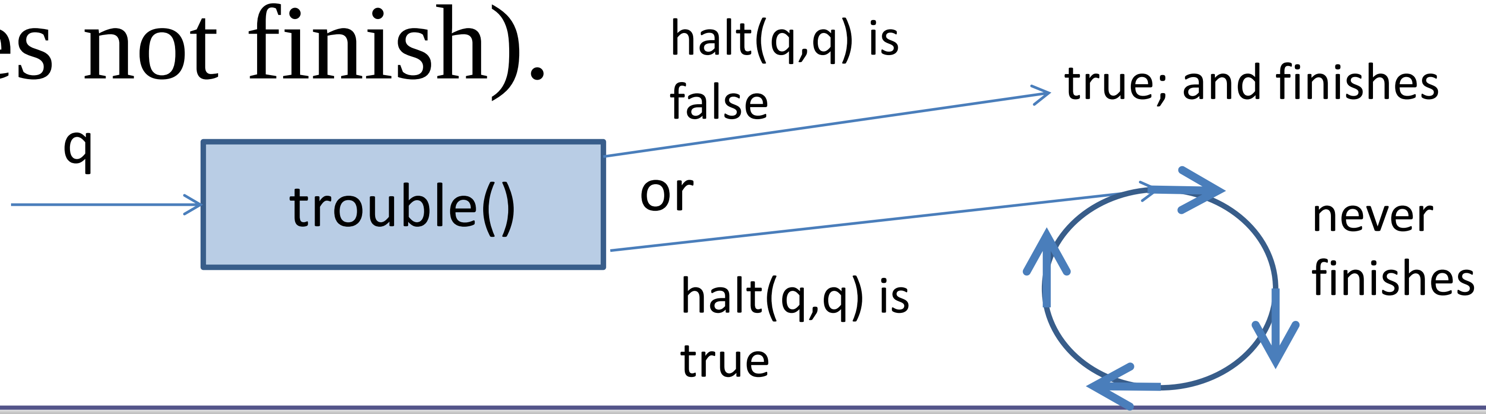
It creates two copies of its input q, and passes them to halt().

trouble(q) calls halt(q, q)

This means that q is both the program and its data in halt()

Implementing trouble()

If halt(q,q) outputs false, then trouble() returns true and finishes.
If halt(q,q) outputs true, then trouble() goes into an unending loop (and thus does not finish).



```
boolean trouble(String q)
{
  if (!halt(q, q))
    return true;
  else {   // halt(q, q) is true
    while (true) {   // loop forever
      // do nothing
    }
    return false;   // execution never
gets here
  }
}
```

Double Trouble

The input string q of trouble() can be anything:

encode the trouble() function as a string called t, and use that as input.

What is the result of **trouble(t)**?

There are two possibilities: either trouble() returns true (i.e. halts), or loops forever (it does not halt). Let's look at both cases.

Two Cases

1. trouble(t) **halts** which means that halt(t,t) returned false.
But halt() is saying that trouble(t) does **not halt**.
Contradiction.

2. trouble(t) does **not halt** which means that halt(t,t) returned true.
But halt() is saying that trouble(t) does **halt**.
Contradiction

Contradictions are Deadly

In both cases, there's a contradiction because of halt().

This means that it is possible to construct an program/input pair for halt() which it processes incorrectly.

This means that there is no halt() algorithm which correctly works for all inputs.

the **Halting Problem is undecidable**

12.2. Wang Tiling

In 1961, Wang conjectured that if a finite set of tiles tile the plane, then they can be combined to form a periodic tiling

- i.e. a repeating pattern

But in 1966, Robert Berger proved that no such combination algorithm exists

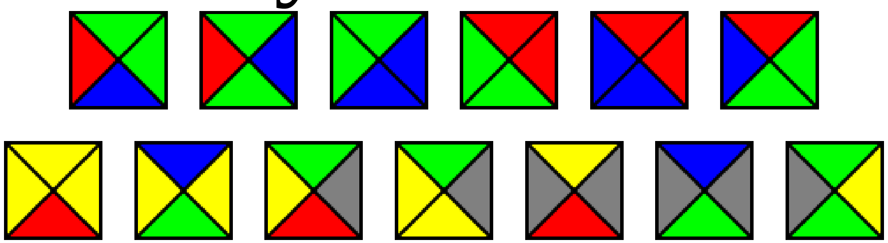
- he used the undecidability of the halting problem to imply the undecidability of this tiling problem

Wang tiles can only tile the plane aperiodically

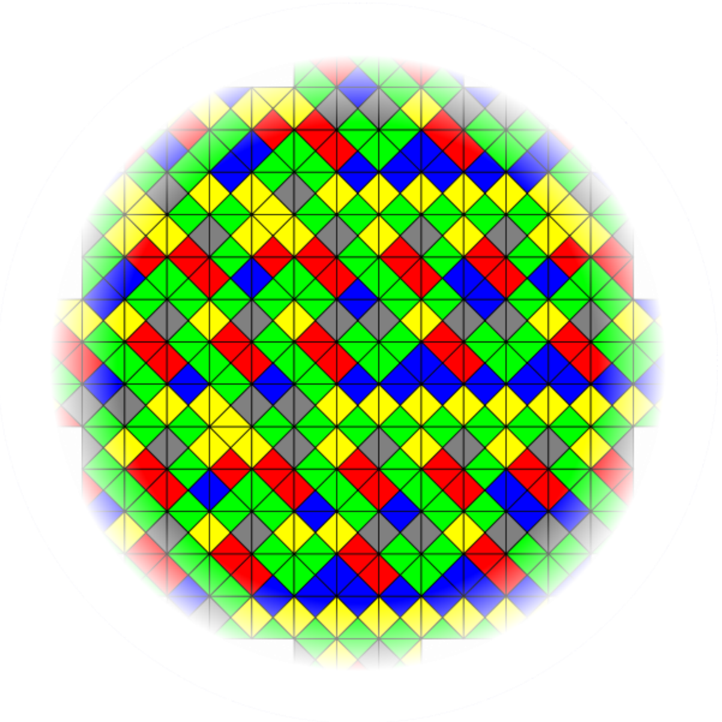
- i.e. a pattern never repeats

Example

13 Wang tiles that can only tile aperiodically:



- Aperiodic example:



Uses

Wang tiles have become a popular tool for procedural synthesis of textures, height fields, and other large, nonrepeating data sets.

e.g. for decorating 3D gaming scenes

12.3. Other Undecidable Problems

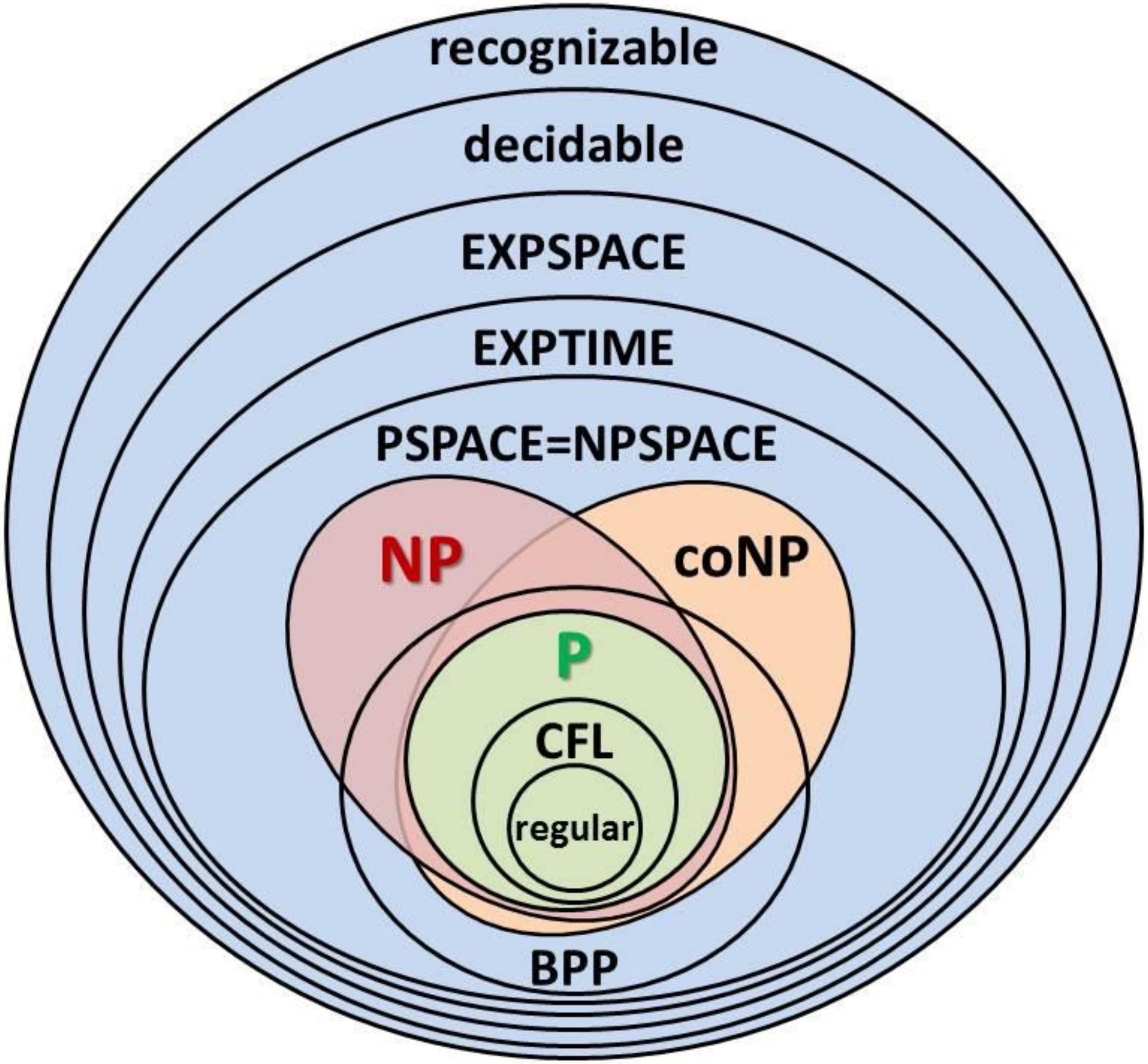
A good list can be found at:

- http://en.wikipedia.org/wiki/List_of_undecidable_problems

The problem categories are mathematical, as you'd expect:

- logic, abstract machines, matrices, combinatorial group theory, topology, analysis, others

13. Computational Difficulty



What!

Adding memory space and randomization to running time creates new complexity classes

PSPACE is the set of all decision problems that can be solved using a **polynomial** amount of **space**
EXPSPACE uses an **exponential** amount of **space**

BPP (bounded-error probabilistic polynomial time)uses a **probabilistic** algorithm in **polynomial time**, with an error probability of at most 1/3 for all instances

NPSPACE is equivalent to **PSPACE**, because a deterministic Turing machine can simulate a nondeterministic Turing machine without needing much more space.

co-NP is the class of problems for which polynomial checking times are possible for ***counterexamples*** of the original NP problem

A counterexample is where the decision problem is negated, and you find a YES/NO answer for that question.

Complexity Classes

P, NP, NPC, NP-Hard, EXP, R, EXP-Hard, ...

PSPACE, NPSPACE, EXPSPACE, ...

quite a few complexity classes (sets)
is that all of them?

No, the "Complexity Zoo" website currently lists **535** classes

https://complexityzoo.uwaterloo.ca/Complexity_Zoo

But the "petting zoo" only lists **16**.

14. Non-Technical P and NP

Video talk "**Beyond Computation: The P vs NP Problem**"

Michael Sipser, Dept. of Maths, MIT

Oct. 2006

http://www.youtube.com/watch?v=msp2y_Y5MLE