Problems, and How Computer Scientists Solve Them

Manas Thakur
PACE Lab, IIT Madras

Content Credits

- *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Hopcroft et al.
- *Introduction to the Theory of Computation*, 2nd edition. Michael Sipser.
- *Algorithms*, TMH edition. Dasgupta et al.
- *https://en.wikipedia.org*
- *https://images.google.com*

Outline

- Computation models
- Solvability
- Complexity
- Coping with difficulties

A Simple Problem

- Design a machine to determine whether a given program P1 prints “Hello World!”.

```
int main() {
    printf(“Hello World!”);
    return 0;
}
```

A Simple Problem (Cont.)

```
int main() {
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; ++x) {
            for (y=1; y<=total-x-1; ++y) {
                z = total-x-y;
                if (exp(x,n)+exp(y,n) == exp(z,n)) {
                    printf("Hello World!");
                }
            }
        }
        ++total;
    }
    return 0;
}
```

```
int exp(int i, int n) {
    int ans, j;
    ans = 1;
    for (j=1; j<=n; ++j) {
        ans *= i;
    }
    return ans;
}
```

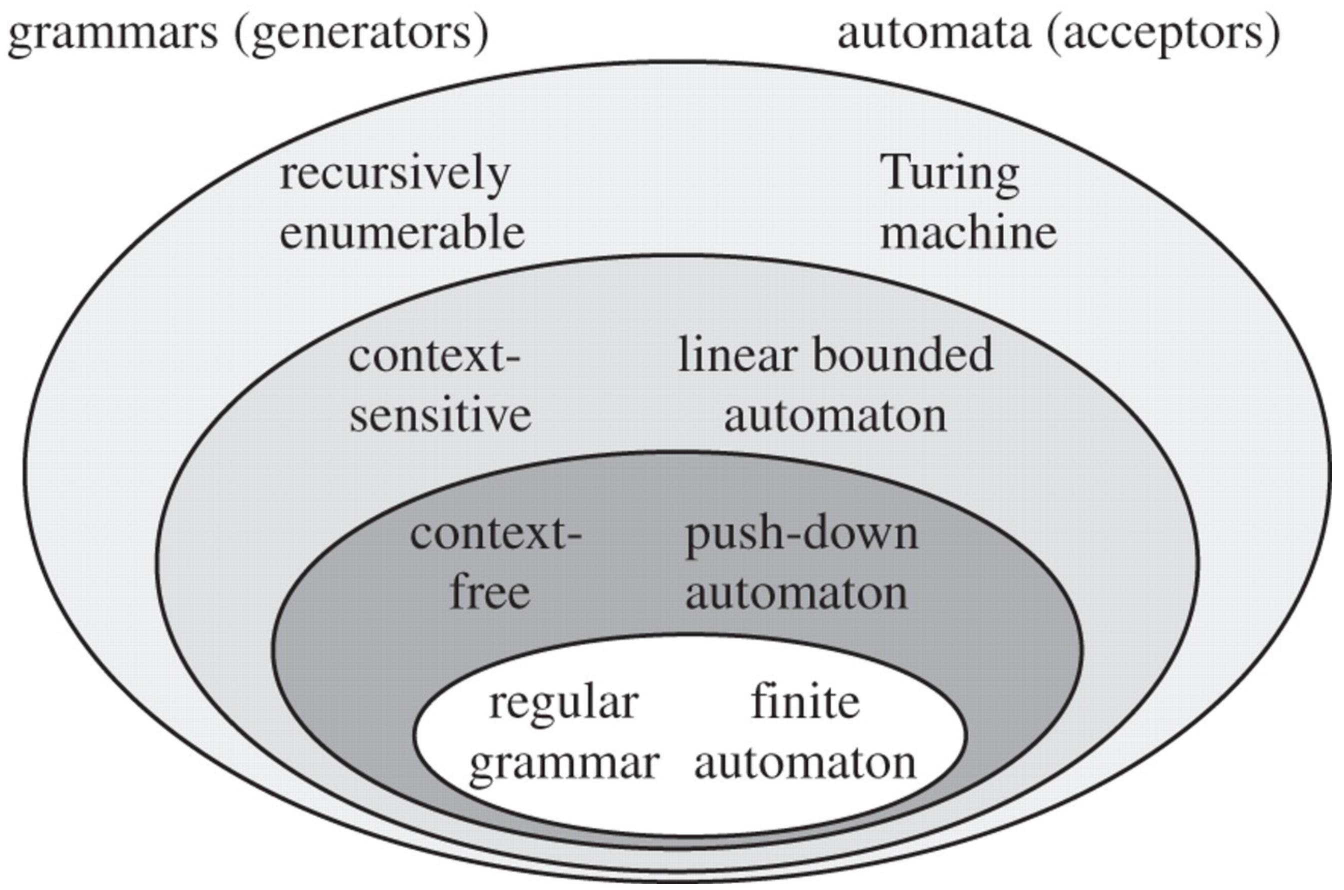
Expressing problems as language-membership tests

- **Step 1:** Represent problem instances as *strings* over a finite alphabet.
  - Our program P1 is essentially a string of characters.
- **Step 2:** Design a machine M1 that accepts *valid* strings.
  - Outputs *yes*, if P1 prints "Hello World!".
  - Outputs *no*, if P1 does not print "Hello World!".
- The language accepted by M1 is:

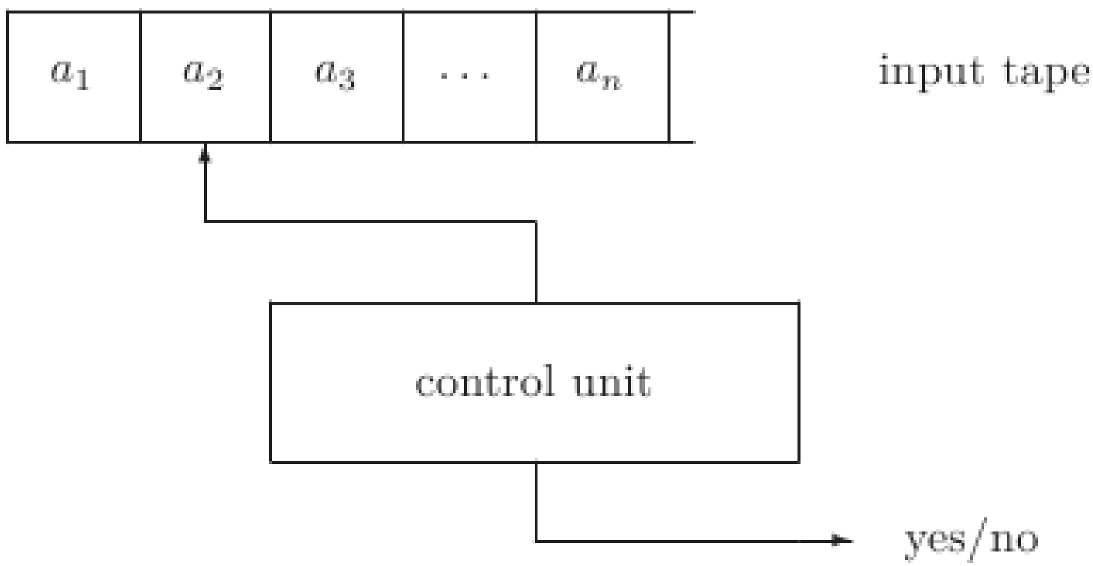
```
L(M1) = { w | w is a program that prints "Hello World!" }
```

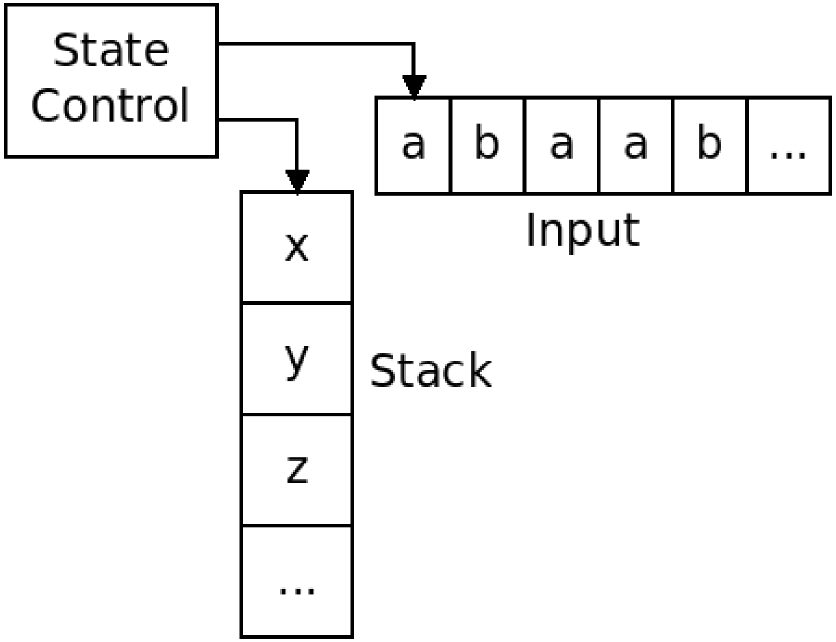
- If M1 always terminates and prints *yes* or *no*, it ***decides*** P1; else it ***recognizes*** P1.

The Chomsky Hierarchy of Languages

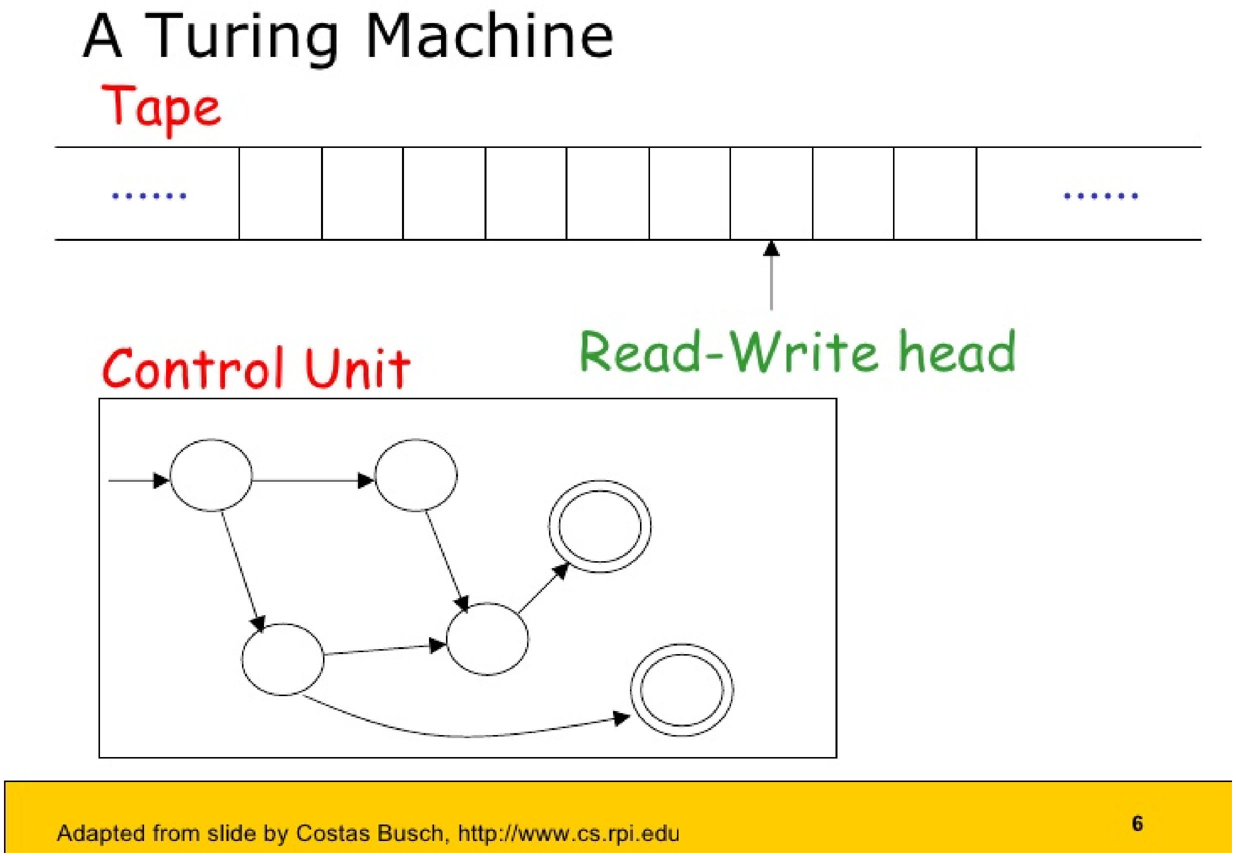


DFA and PDA: A Quick Recap





Turing Machines



Where are we?

- Computation models
- Solvability
- Complexity
- Coping with NP-Completeness

An “undecidable” problem

- Given a TM *M*, and an input *w*, does *M* halt on *w*?
- **Step 1:**

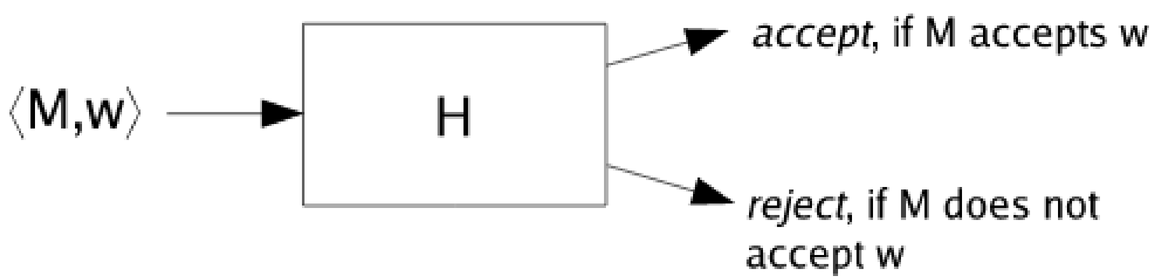
```
L(M) = { <M,w> | M is a TM that halts on w }
```

- **Step 2:**

Our First Undecidability Proof

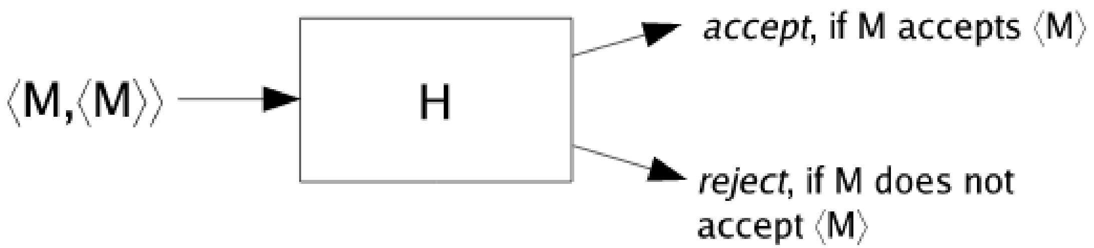
- Prove that $A_{TM} = \{ \langle M,w \rangle \mid M \text{ is a TM that accepts } w \}$ is undecidable.

- Assume that A_{TM} is decidable by the following TM H:

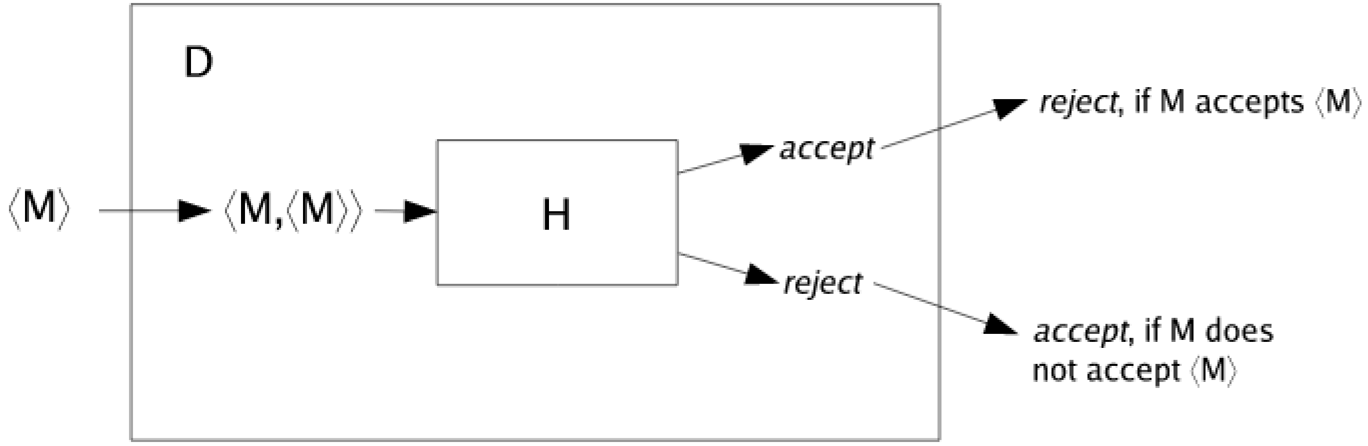


Our First Undecidability Proof

- Give the string representation of M as input to H:

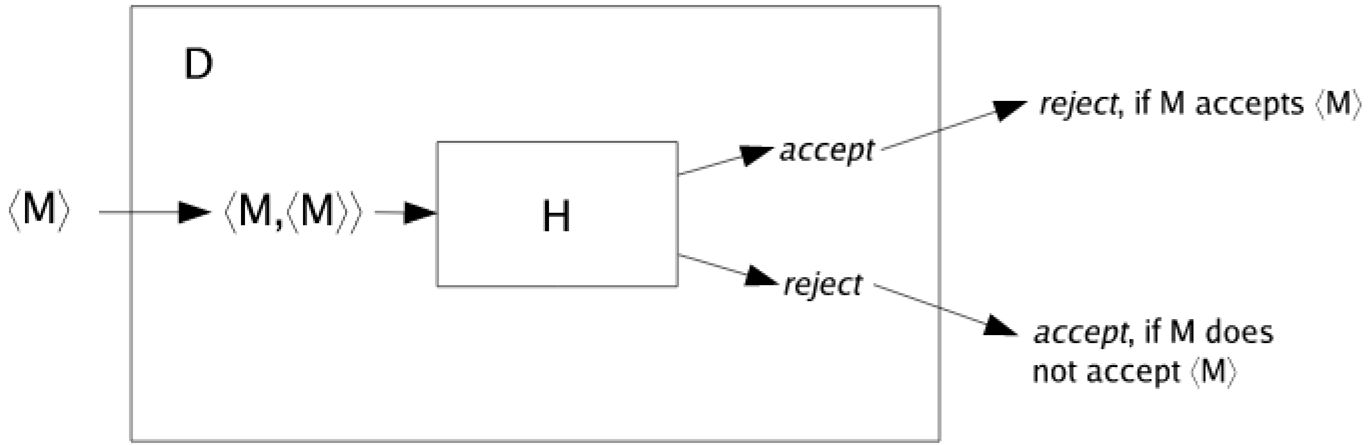


- Construct another TM D as follows:



Our First Undecidability Proof

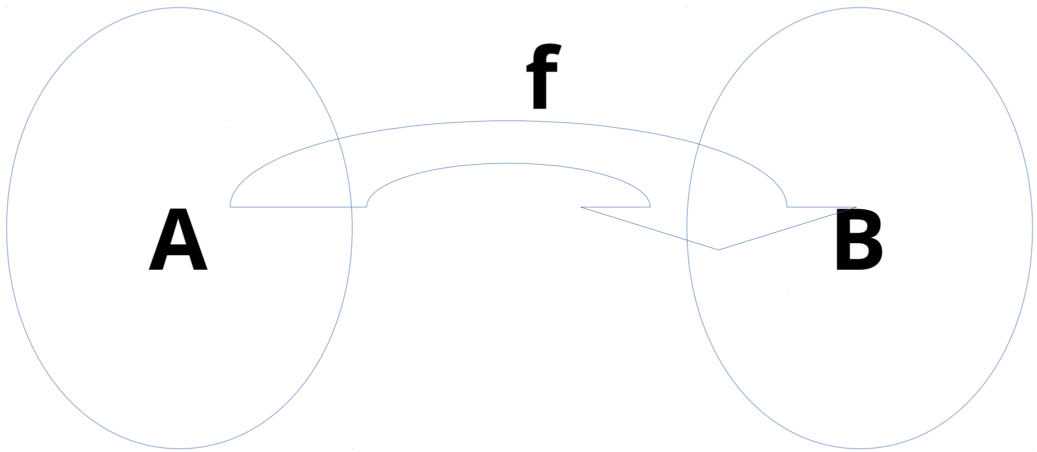
- What does D do on <D> as input?



- *D accepts <D> if D does not accept <D>, and vice-versa.*
- **Contradiction!**
- Hence, H does not exist. Thus, A_{TM} is undecidable!! :-)

Reducibility

- Reduce Problem A to Problem B.



- If B is decidable, so is A.
- If A is undecidable, so is B.

```
~(p implies q) == ~q implies ~p
```

Back to the Halting Problem

- `L(M) = { <M,w> | M is a TM that halts on w }`
- Assume M_H decides L(M).
- Reduce A_{TM} to M_H:

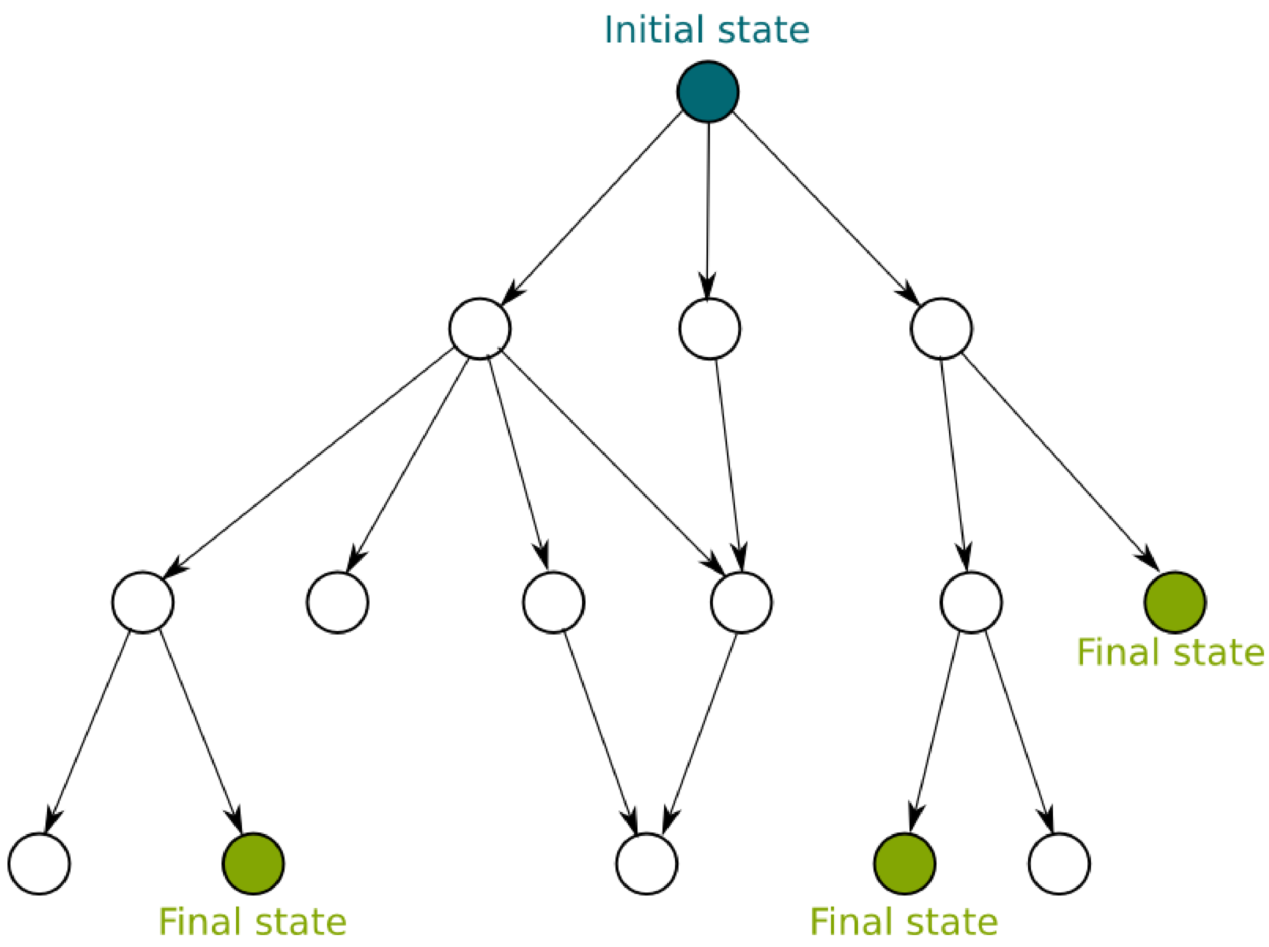
  A_{TM} = `{ <M,w> | M is a TM that accepts w }`

  - Run M_H on `<M,w>`.
  - If M_H rejects (i.e., M does not halt on *w*), then *reject*.
  - If M_H accepts, then simulate M on *w* (guaranteed to stop).
  - *Accept* if M accepts *w*; *reject* if M rejects *w*.
- Thus, if M_H always halts (assumed above), then A_{TM} is decidable.
- **Contradiction!**
- *Note that M_H is Turing-recognizable, though.*

Turing Machines and Algorithms

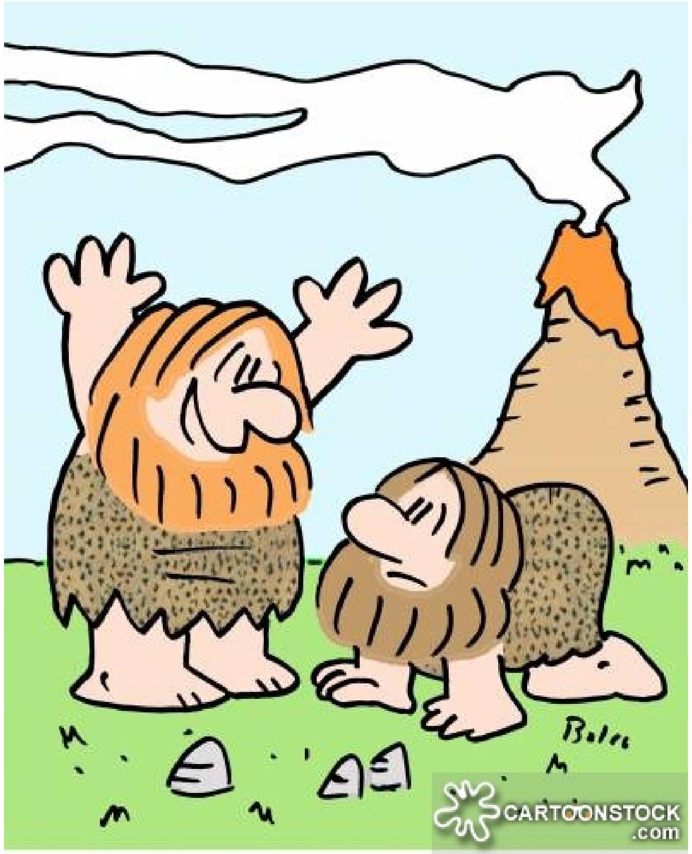
- **Church-Turing Thesis:** Every algorithm can be realized as a Turing Machine.
- A multitape-TM is equivalent to a single-tape TM.
- A TM can simulate a computer.
- A computer with an *infinite tape* can simulate a TM.
- Turing Machines are more powerful than modern day computers!!
- What about Nondeterministic Turing Machines?

Non-determinism: The Power of Guessing



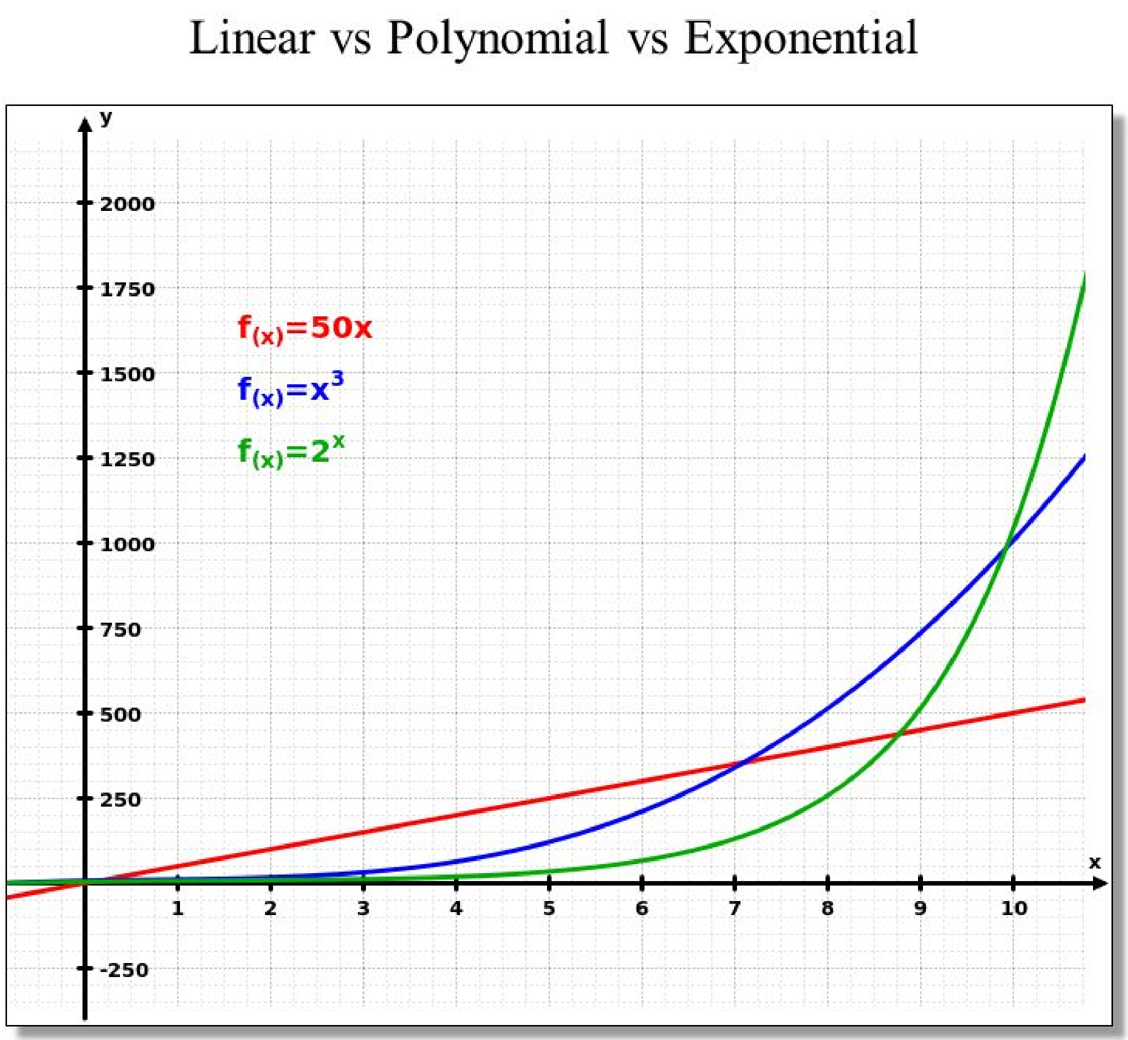
A Shift

- Computation models
- Solvability
- Complexity
- Coping with NP-Completeness



"Man, you've got to try this 'walking upright' stuff! — it's like a total paradigm shift!"

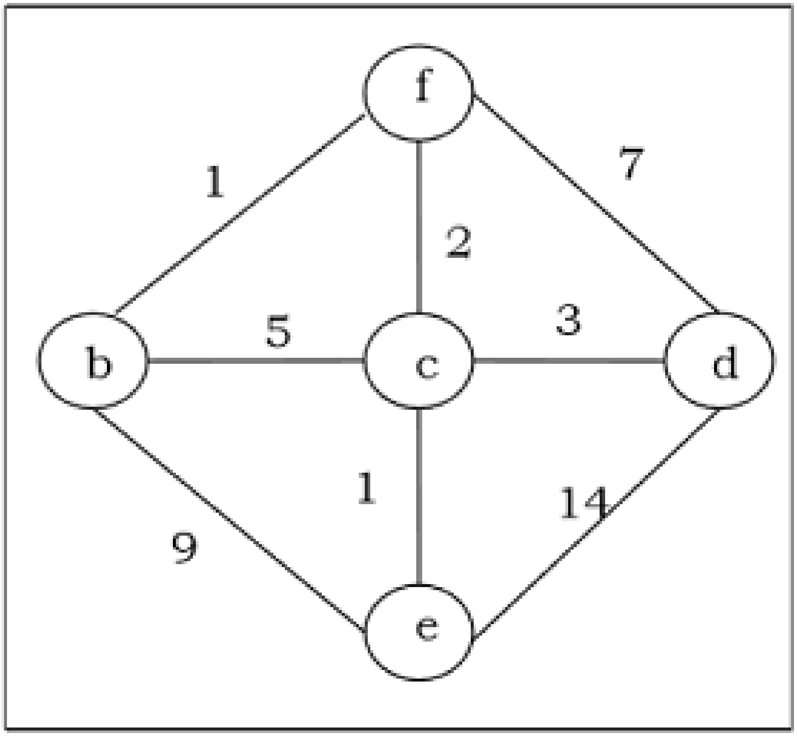
Can a problem be solved in "good-enough" time?

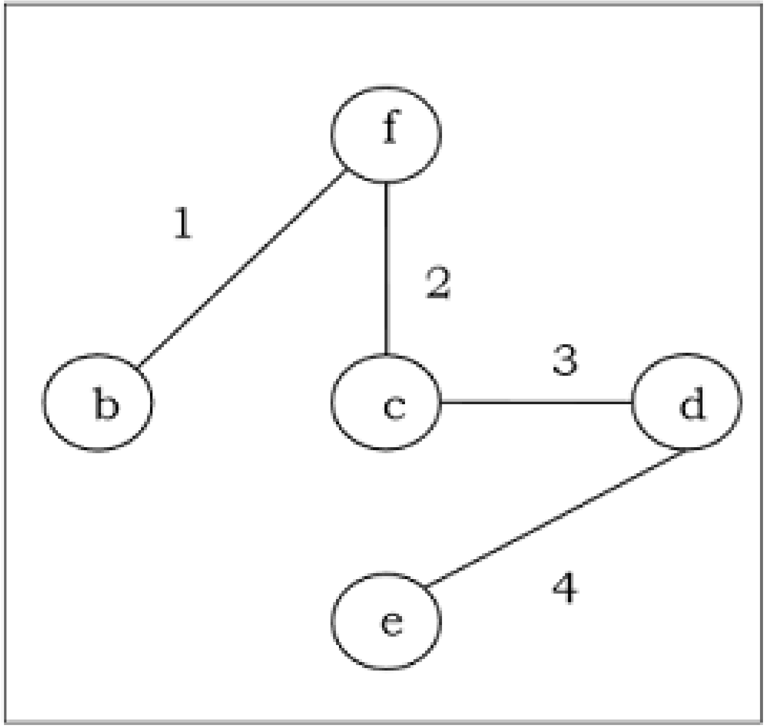


The *P* class of problems

- Problems that can be solved in polynomial time by a Deterministic Turing Machine
- All practical problems that we write algorithms for
- Example: Minimum Spanning Tree

The Minimum Spanning Tree Problem

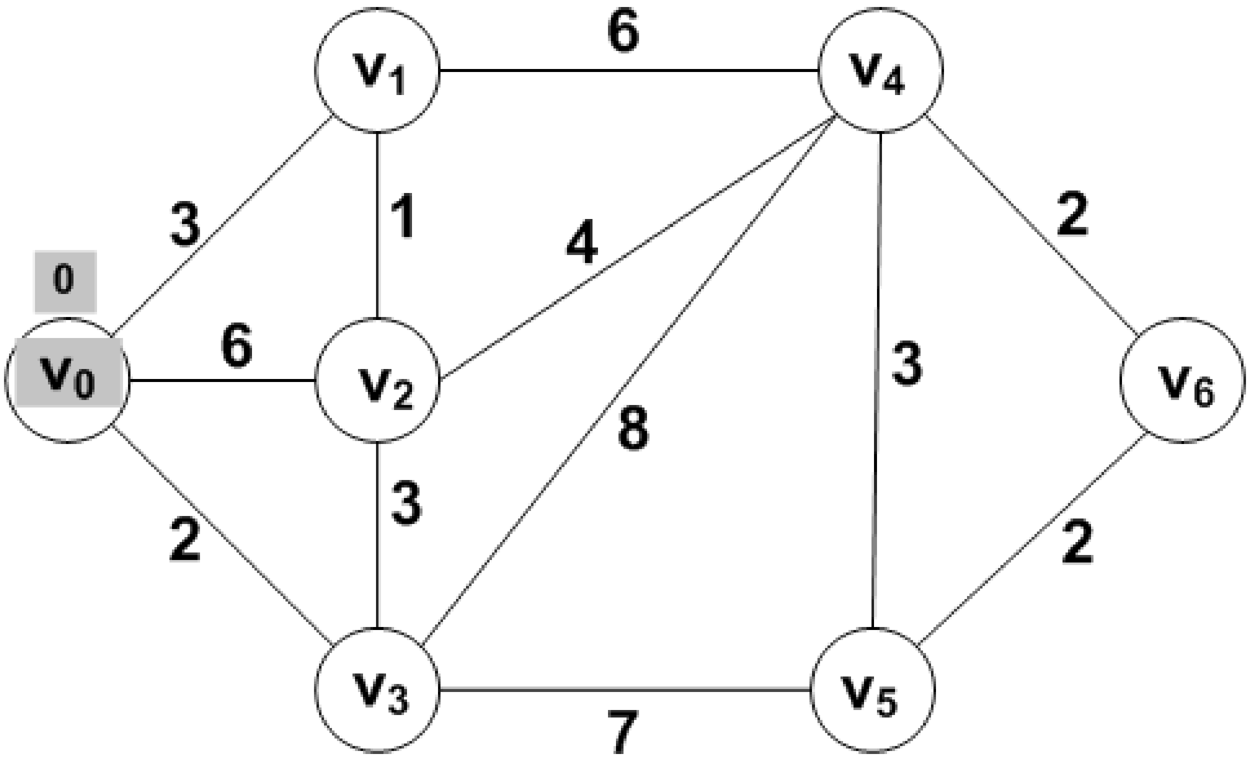




The *NP* class of problems

- Problems that can be solved in polynomial time by a Nondeterministic Turing Machine
  - A given solution can be ***checked*** in polynomial time by a Deterministic Turing Machine
- Even though the power of an NTM is equivalent to that of a DTM, the time requirements of NP may not be in the “good-enough” zone
- Example: Travelling Salesman Problem (decision version)

The Travelling Salesman Problem



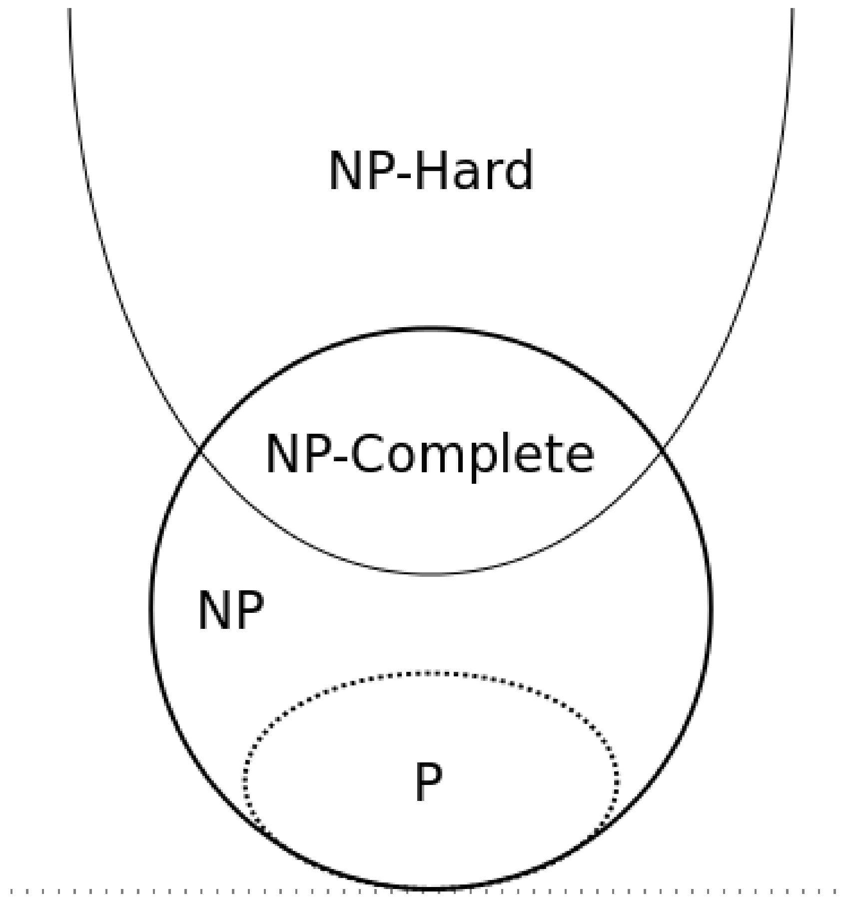
Is *P* = *NP*?

- A problem Q is ***NP-Complete*** if:
  - Q is in *NP*
  - All problems in *NP* can be reduced (in polynomial time) to Q
- A problem R is ***NP-Hard*** if:
  - All problems in *NP* can be reduced (in polynomial time) to R
  - It's not known whether R is in *NP*
- Thus, if even a single NP-Complete problem can be solved by an algorithm in polynomial time, then *P* = *NP.*
- It seems that P != NP; however, there is no proof yet!

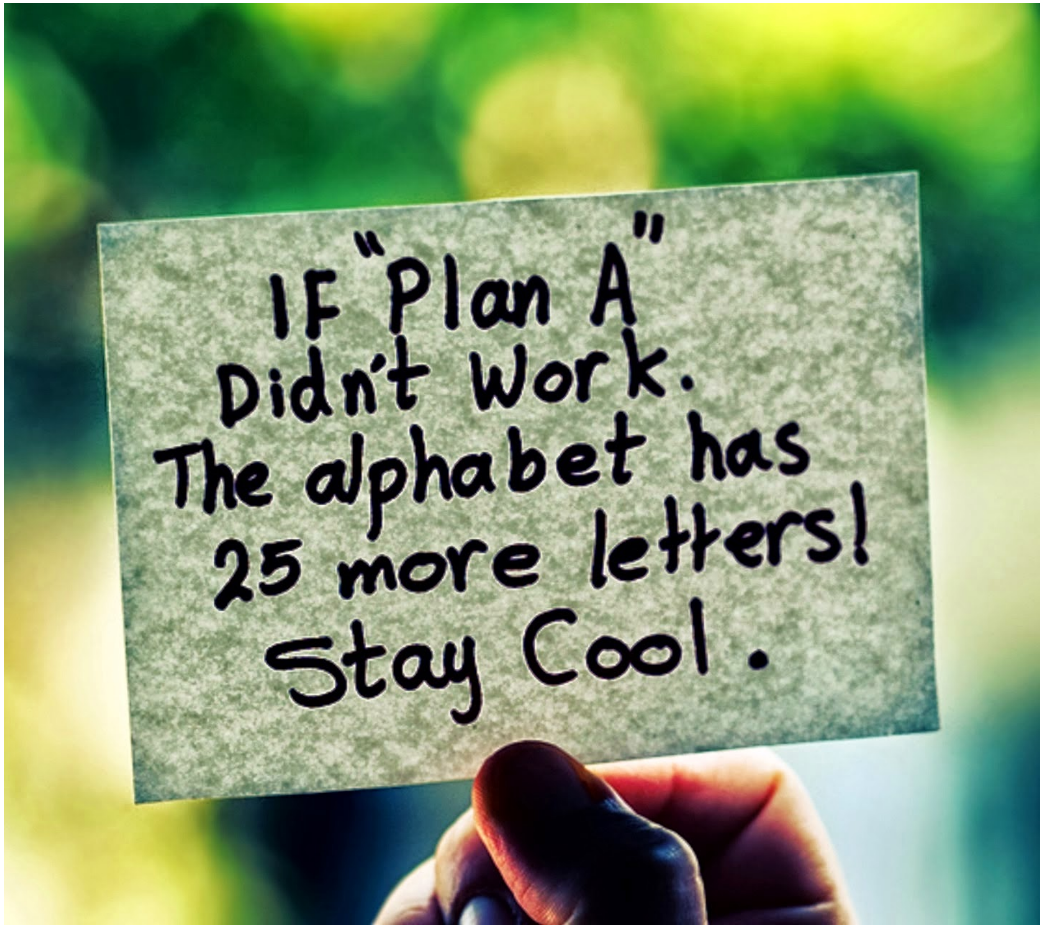
Some popular problems

- NP-Complete:
  - TSP
  - SAT
  - Subset sum
  - Vertex cover
  - Graph coloring
  - Decision version of TSP
- NP-Hard but not NP-Complete:
  - The Halting Problem (undecidable)
  - General TSP

A conclusive picture:



So do we give up?



Never surrender!

- Computation models
- Solvability
- Complexity
- Coping with NP-Completeness

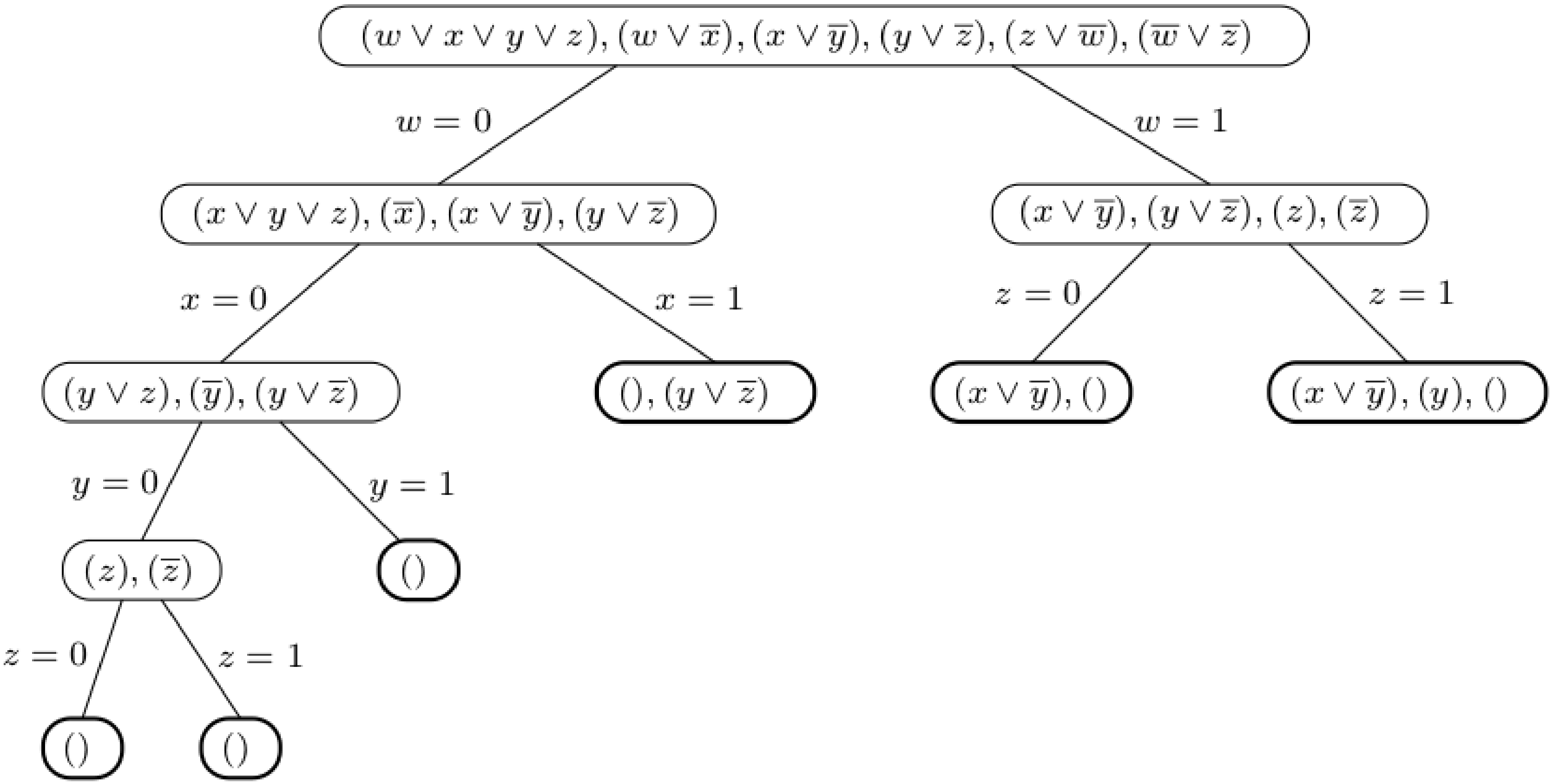
Special Cases

- SAT is *NP-Complete*.
- 2-SAT is in *P*.

- Vertex cover problem is *NP-Complete*.
- Vertex cover problem for bipartite graphs is in *P*.

Intelligent Backtracking

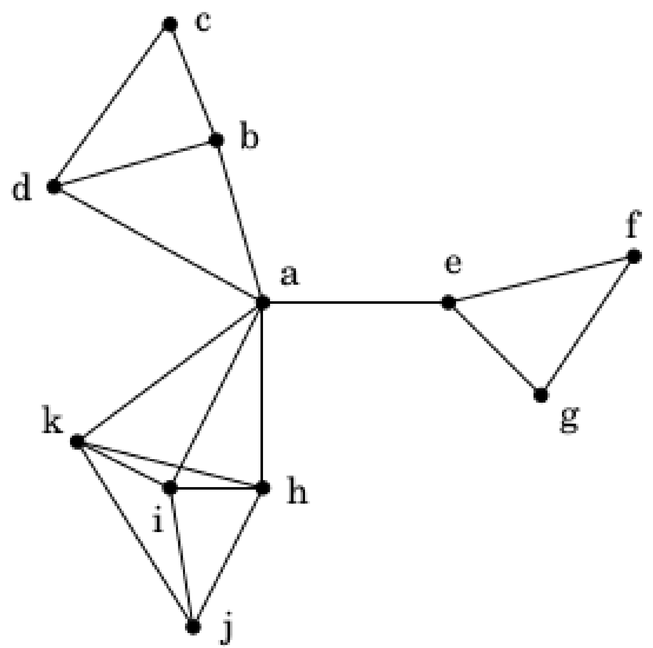
- Useful for exhaustive space-search problems
- Consider the SAT instance:



Approximation

- Obtain a near-optimal solution
- Consider the following problem:

There are 11 towns. According to a government policy, each hospital can cover 30 miles of distance around it. Find the optimal number of hospitals that need to be opened.

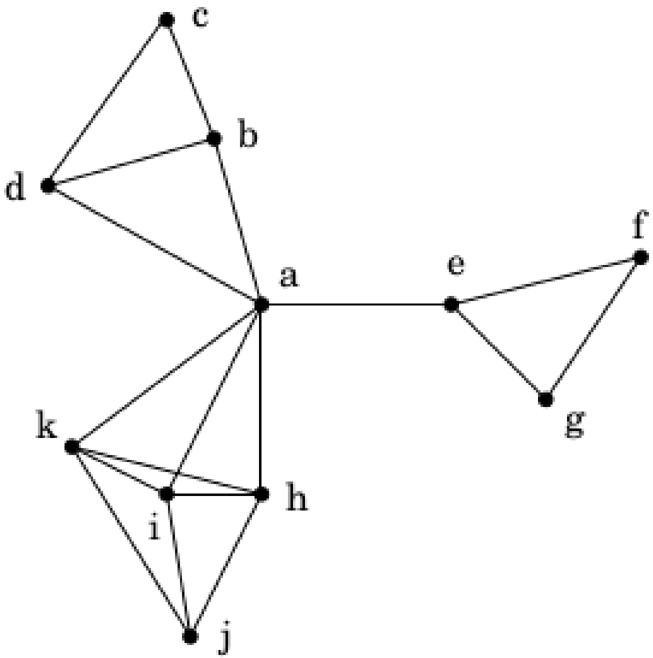


Approximation (Cont.)

- Can be reduced to the Set Cover problem:
  - *Input:* A set of elements
  - *Output:* A selection of S_i whose union is B
  - *Cost:* Number of sets picked

- *Greedy algorithm:* At each step, pick the set S_i with the largest number of uncovered elements
  - `{a, c, j, f}` or `{a, c, j, g}`
- *Optimal:* `{b, e, i}`

- It can be proved that if the optimal set has k elements, the Greedy algorithm generates at max $k.\ln n$ sets.



So how do *YOU* solve problems?

- Ask others for a solution
- Think, re-think, and think more
- Find a best-attempt solution
- Simplify the problem
- Try to generalize the solution
- Prove it unsolvable!

How do *Computer Scientists* solve problems?

- Ask others for a solution
- Think, re-think, and think more
- Find a best-attempt solution
- Simplify the problem
- Try to generalize the solution
- Prove it unsolvable!

- Reduction
- Different algorithms
- Approximation
- Special cases
- Other cases?
- Prove it NP-Complete!

Stay Hungry, Stay Foolish, Stay Connected

www.cse.iitm.ac.in/~manas

github.com/manasthakur
gist.github.com/manasthakur

manasthakur17@gmail.com

manasthakur.wordpress.com

linkedin.com/in/manasthakur

manasthakur.github.io/docs/year3sct.pdf