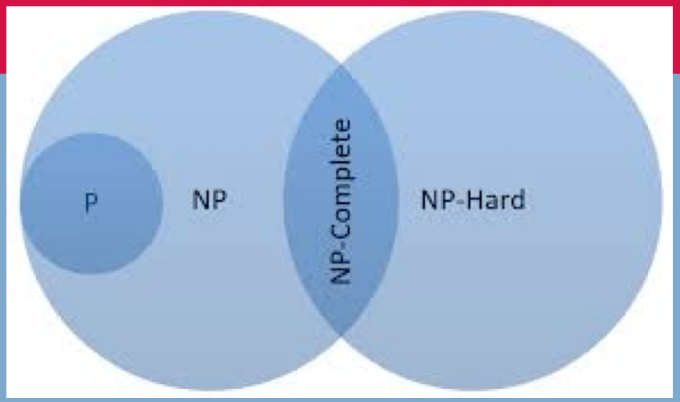
CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

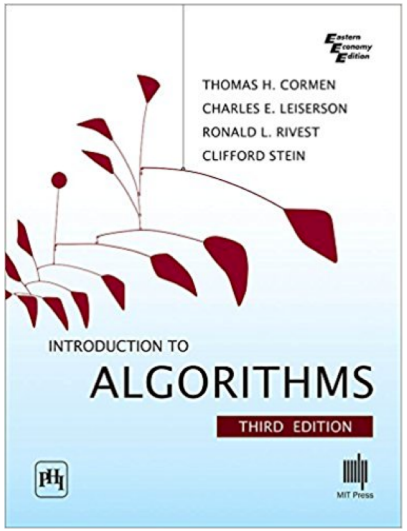
Metaheuristic Optimization

NP Complexity

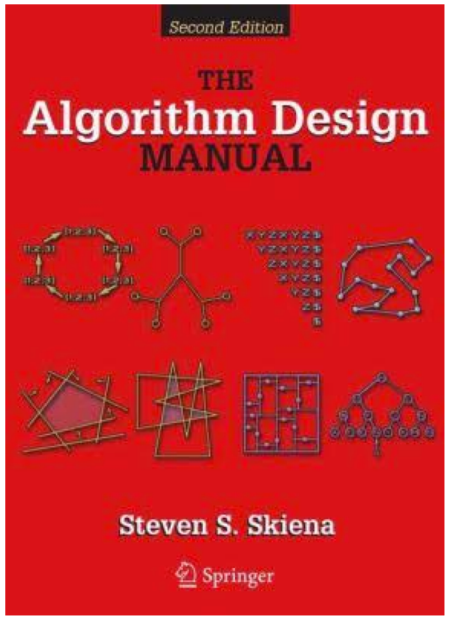
Dr. Diarmuid Grimes



Books



Introduction to Algorithms Complexity
Chapter 34 – NP-completeness



The Algorithm Design Manual
Steven S. Skiena
Chapter 9
Intractable Problems and Approximation Algorithms

Recap: algorithm complexity

- Complexity analysis is a technique to analyze and compare algorithms (not programs).
- It helps to have preliminary back-of-the-envelope estimations of runtime (milliseconds, seconds, minutes, days, years?).
- Worst-case analysis is sometimes overly pessimistic. Average case is also interesting (not covered in this course).
- In many application domains (e.g., big data) quadratic complexity, $O(n^2)$, is not acceptable.

The Boolean Satisfiability Problem (SAT)

The Traveling Salesman Problem (TSP)

The Boolean Satisfiability Problem (SAT)

The Satisfiability Problem

- Definition of the **satisfiability problem:** Given a Boolean formula over a set of variables (x_1,…, x_n), determine whether this formula is satisfiable or not.

- A ***literal***: x_i or $\neg x_i$
- A ***clause***: $x_1 \vee x_2 \vee \neg x_3 \equiv C_i$
- A ***formula***: in *conjunctive normal form* (CNF)

$$C_1 \& C_2 \& \ldots \& C_m$$

Goal: Find an assignment of True/False for all the (Boolean) variables s.t. all clauses are True

The Satisfiability Problem: Example

A logical formula:

$x_1 \vee x_2 \vee x_3$

$\& \ \neg x_1$

$\& \ \neg x_2$

the assignment :

$x_1 \leftarrow F , x_2 \leftarrow F , x_3 \leftarrow T$

will make the above formula true.

($\neg x_1$, $\neg x_2$, x_3) represents $x_1 \leftarrow F , x_2 \leftarrow F , x_3 \leftarrow T$

SAT

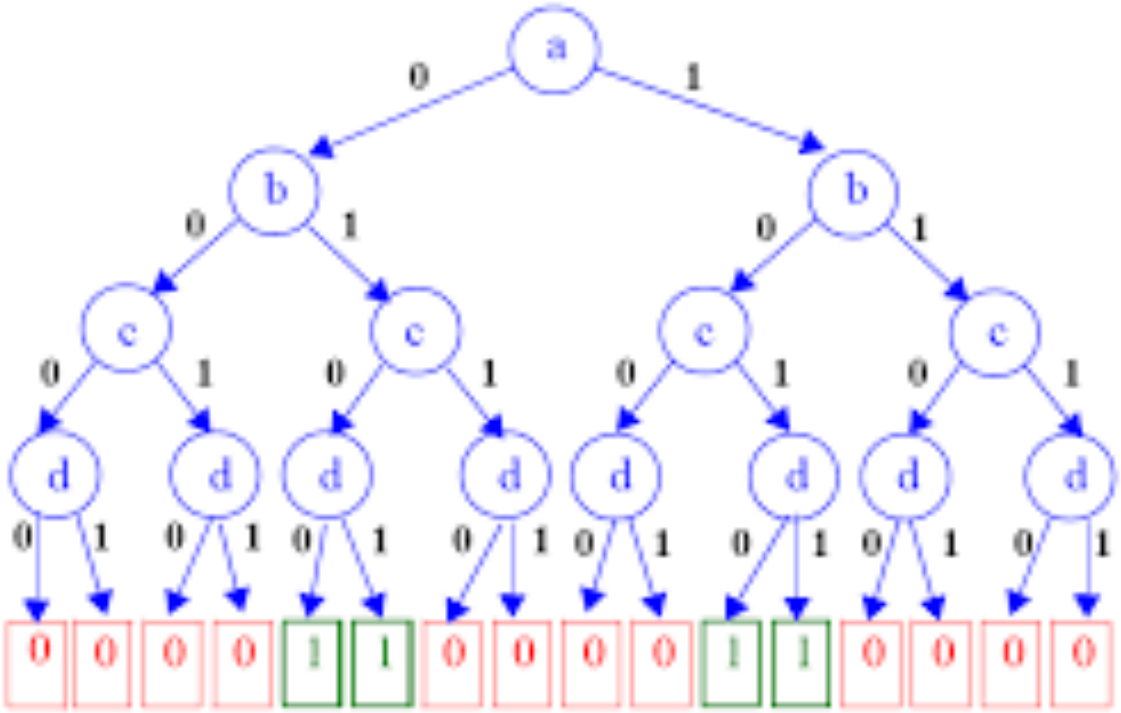
- If there is ***at least one*** assignment which satisfies a formula, then we say that this formula is ***satisfiable***; otherwise, it is ***unsatisfiable***.

- An unsatisfiable formula:

$$x_1 \vee x_2$$
$$\& \; x_1 \vee \neg x_2$$
$$\& \; \neg x_1 \vee x_2$$
$$\& \; \neg x_1 \vee \neg x_2$$

SAT

$$f(a,b,c,d) = (a \vee b \vee c) \cdot (a \vee b \vee \bar{c}) \cdot (\bar{a} \vee c \vee d) \cdot (\bar{a} \vee c \vee \bar{d}) \cdot (\bar{b} \vee \bar{c} \vee d) \cdot (\bar{b} \vee \bar{c} \vee \bar{d})$$



https://www.mdpi.com/2073-8994/2/2/1121

SAT Example: Timetabling

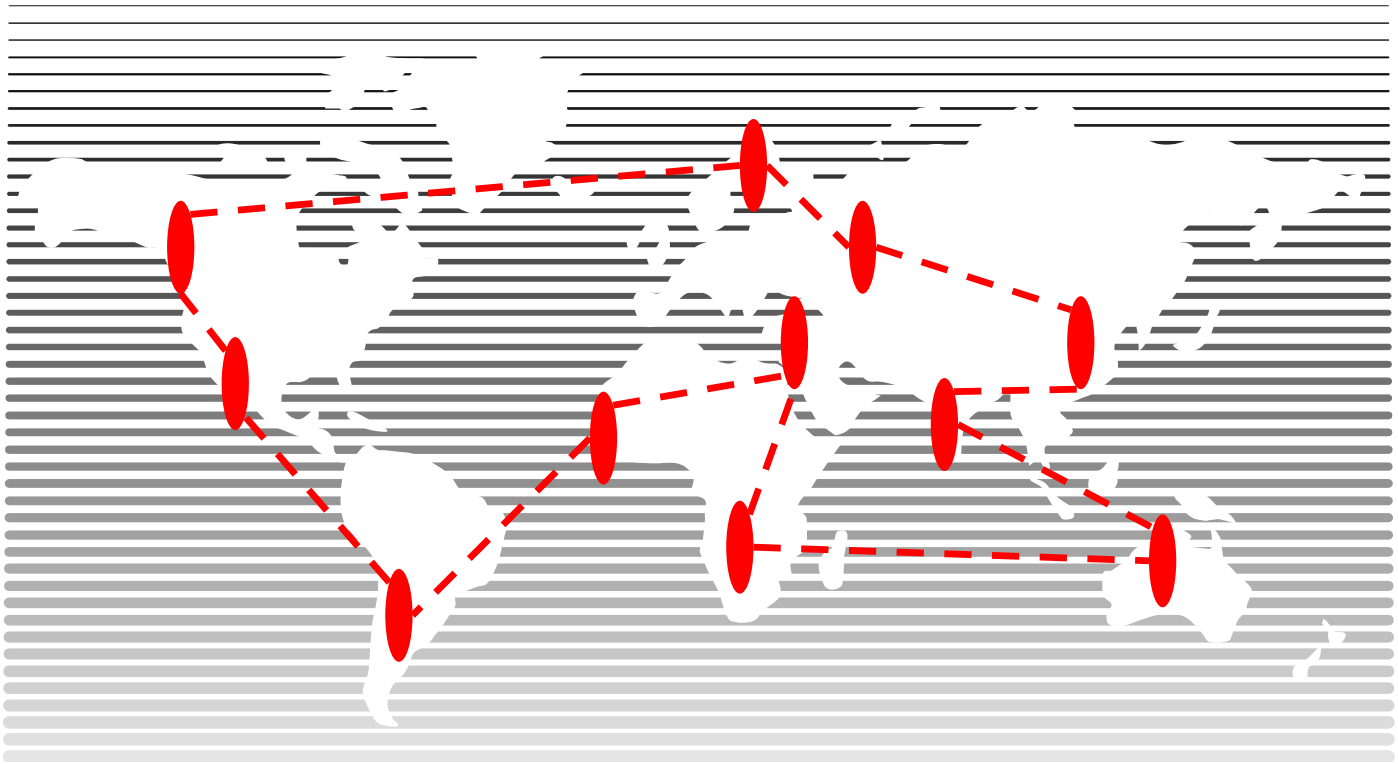
Sample SAT encoding (more details in paper: Achá, Roberto Asín, and Robert Nieuwenhuis. "Curriculum-based course timetabling with SAT and MaxSAT." *Annals of Operations Research* 218, no. 1 (2014): 71-91.)

- Boolean variable for every class,hour pair $X_{c,h}$
- Boolean variable for every class,room pair $X_{c,r}$
- For every class, will have a set of clauses which together enforce that at least 1 of the $X_{c,h}$ variables is true and another set of clauses which together enforce that at most 1 of the $X_{c,h}$ variables is true
- For every pair of classes c,c' that share a student or a lecturer, we have a clause for every hour ($\neg X_{c,h} \vee \neg X_{c',h}$) and similarly for every room. These clauses enforce that both can't take the value True for the same hour/room.
- Room capacity is enforced simply by adding a clause with one literal ($\neg X_{c,r}$) for every room with less capacity than the size of class *c*

Traveling Salesman Person

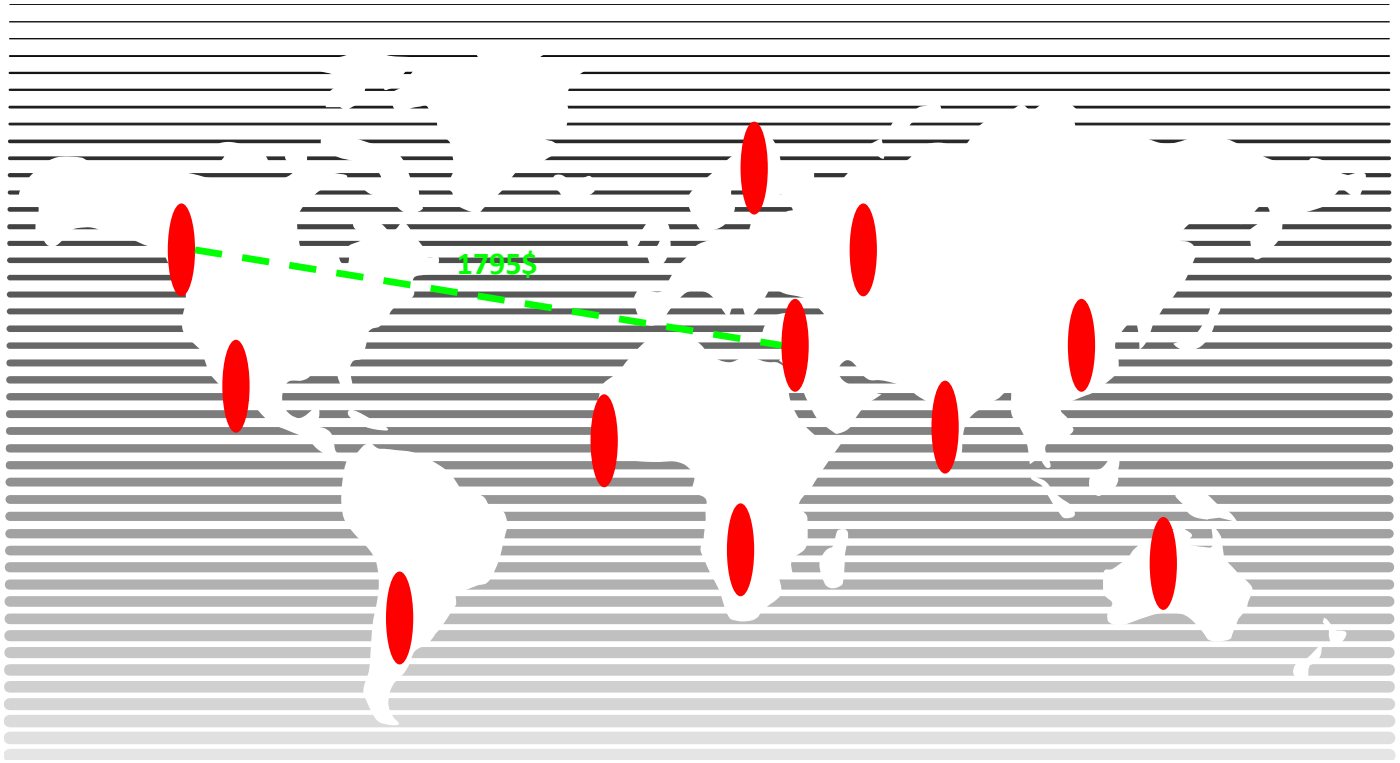
TSP

The Mission: A Tour Around the World



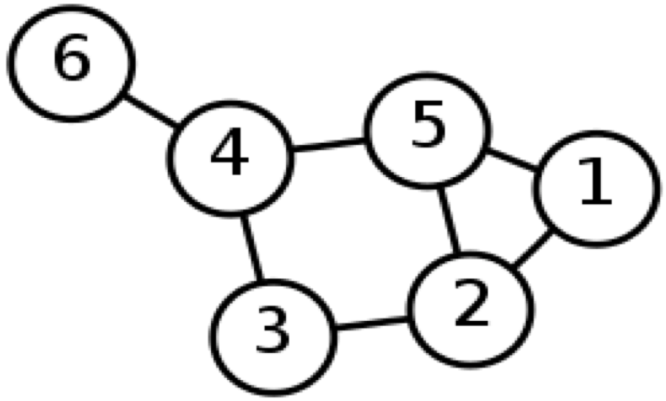
Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

The Problem: Traveling Costs Money



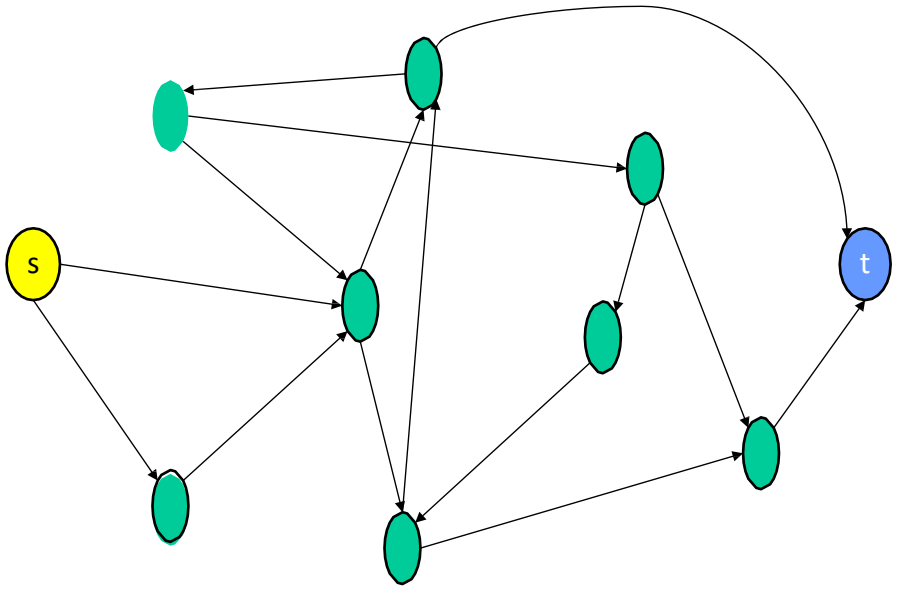
Graphs

*"a **graph** is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge(also called an arc or line).*

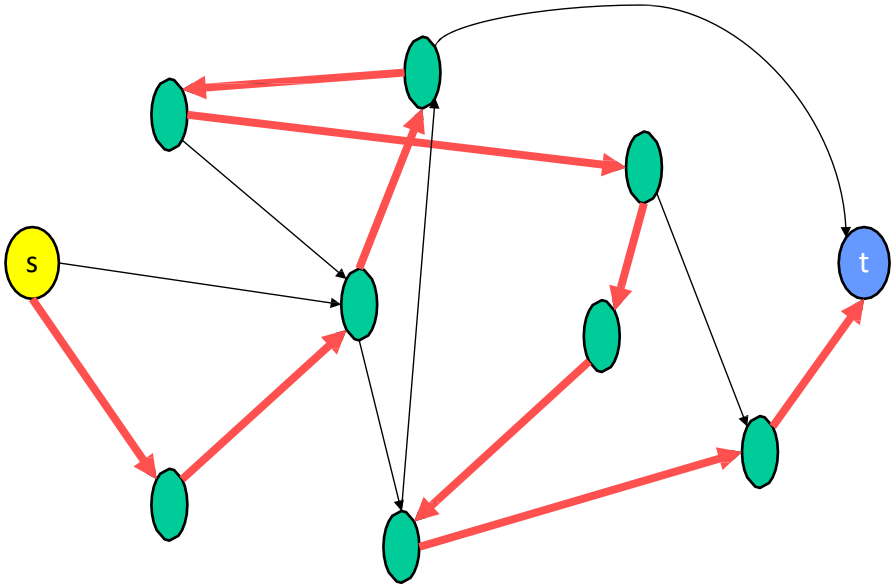


Hamiltonian Path

- **Instance**: a directed Graph. G = (V, E) and two vertices s, t ∈ V
- **Problem**: to decide if there exists a path from **s** to **t**, which goes through each node once



Can You Find One Here?


s
t

SAT vs. Hamiltonian Path vs. TSP

- Finding **a** solution
- Finding the **optimal** solution

First lab discussion: Insertion heuristic for TSP

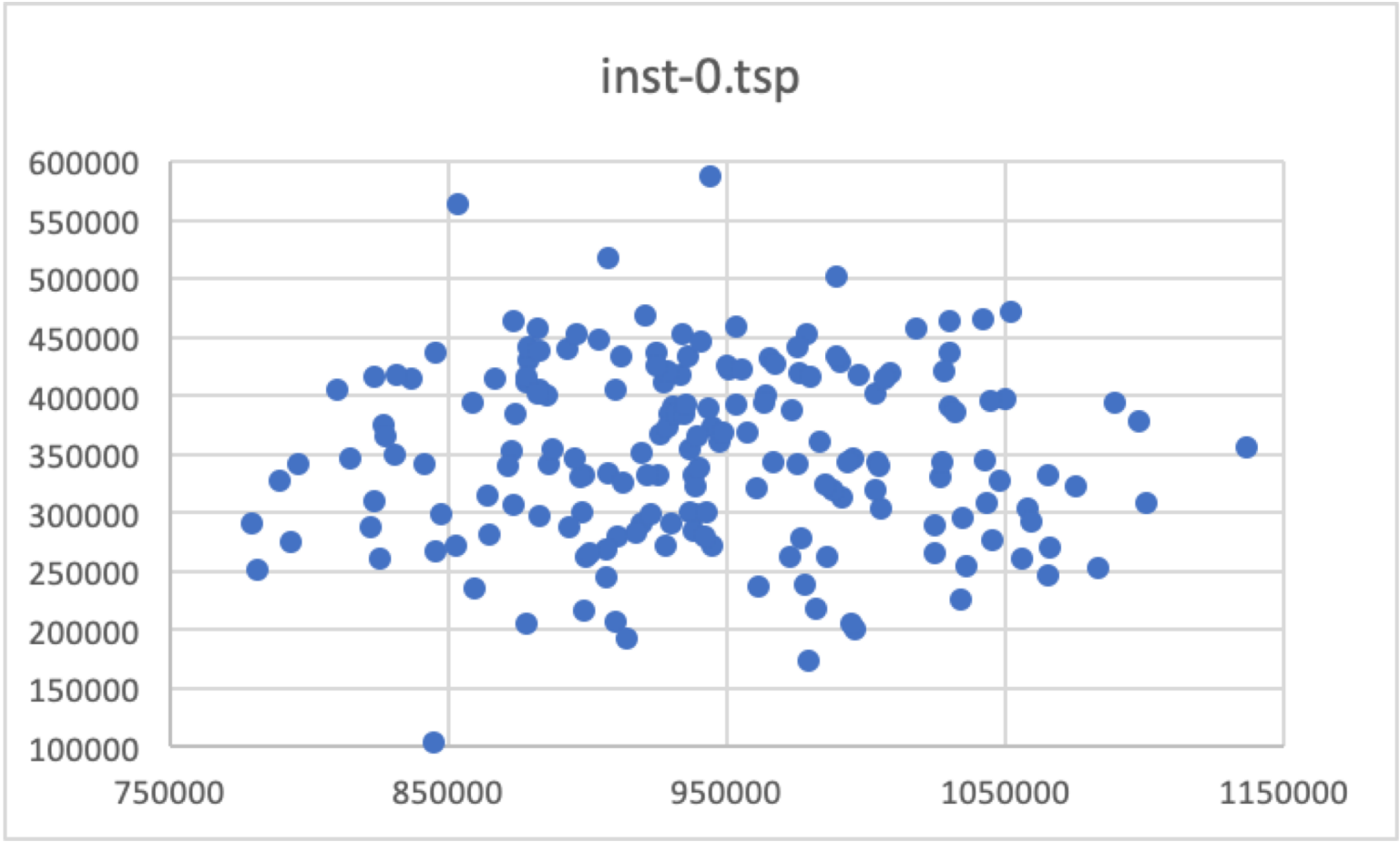
- Distance between pair of cities will be Euclidean distance between x,y coordinates of the pair.

- Solution is route covering all cities, each city visited only once.
  - Should be cycle, i.e. must return to first city at end. So every node should have 2 edges, one from previous city, one to next city.

- Heuristic method: *Nearest-Neighbor.* Choose city at random for start and then repeatedly choose the nearest city to the last chosen city as next city until all cities are on route.

- Alternative: Choose city at random for start and then repeatedly choose city at random and insert it beside nearest city already on route, until all cities are done.

First lab discussion



First lab discussion

- Coding: Read in data. Use whatever data structure, e.g. dictionary with city-id for key and tuple of x,y as element.
- Input format: id and x,y co-ordinates
- Output: First line is the cost of the solution, sum of the distances between consecutive cities in the route, then each subsequent line is the next city on the route

- NOT looking for optimal solution
  - Looking for *good* initial solution and find it fast
- Advise: Create small dummy tsp for debugging

- Python3!
  - Sample IDEs: Spyder, PyCharm, etc.

Problem Complexity

Polynomial Time Algorithms

- Most of the algorithms we have mentioned so far run in time that is upper bounded by a polynomial in the input size
  - sorting: $O(n^2)$, $O(n \log n)$, …
  - matrix multiplication: $O(n^3)$, $O(n^{\log_2 7})$
  - graph algorithms: $O(V+E)$, $O(E \log V)$, …

- In fact, the running time of these algorithms are bounded by small polynomials.

Classifying Problems

Coarse categorization of problems:

1. Problems solvable in polynomial time (i.e., there exists an algorithm solving this problem in polynomial time). We will consider these problems "tractable“.
2. Those not **known to be** solvable in polynomial time

We shall denote by P the class of all polynomial time solvable ***decision*** problems.

Why Polynomial Time?

- It is convenient to define decision problems to be tractable if they belong to the class P, since:
  - the class P is closed under composition.
  - the class P becomes more or less independent of the computational model.

[Typically, computational models can be transformed into each other by polynomial time reductions.]

Decision Problems and the class P

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

We shall denote by P the class of all decision problems that are solvable in polynomial time.

The Class NP

NP is the class of all decision problems for which a candidate solution can be **verified** in polynomial time.

- The class P is a subset of NP.
- NP is short for ***nondeterministic*** polynomial time (since these problems can be solved on a *nondeterministic Turing machine* in polynomial time).
- NP does not stand for not-P! Why?

Verifying a Candidate Solution

- Difference between solving a problem and verifying a candidate solution:

- Solving a problem: is there a path in graph G from node u to node v with at most k edges?

- Verifying a candidate solution: is v_0, v_1, …, v_ℓ a path in graph G from node u to node v with at most k edges?

Verifying a Candidate Solution

- A Hamiltonian cycle in an undirected graph is a cycle that visits every node exactly once.
- Solving a problem: Is there a Hamiltonian cycle in graph G?
- Verifying a candidate solution: Is $v_0, v_1, \ldots, v_\ell$ a Hamiltonian cycle of graph G?

Verifying a Candidate Solution vs. Solving a Problem

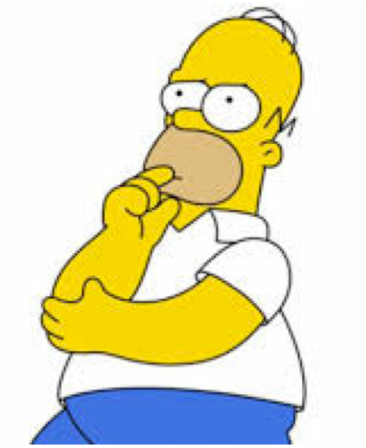
- Intuitively it seems much harder to find a solution to a problem from scratch than to verify if a full assignment is a solution the problem.

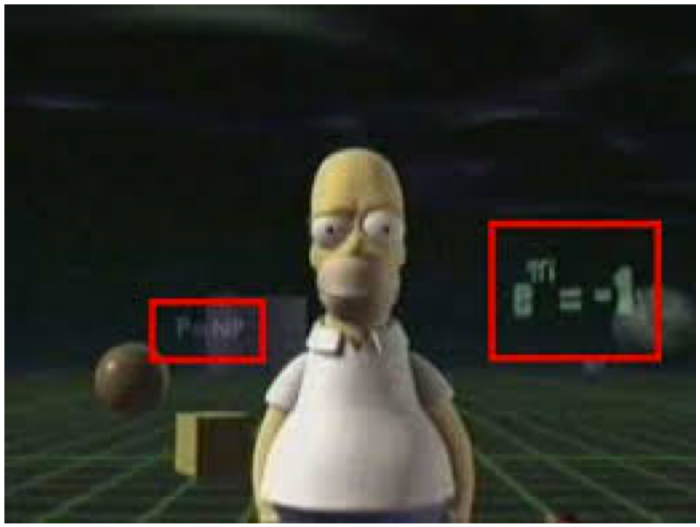
- If there are many candidate solutions to check, then even if each individual one is quick to check, overall it can still take a long time.

Verifying a Candidate Solution

- Many practical problems in computer science, mathematics, operations research, engineering, etc. are **poly-time** verifiable but have no known **poly-time** algorithm.
  - Wikipedia lists problems in computational geometry, graph theory, network design, scheduling, databases, program optimization and more

P versus NP



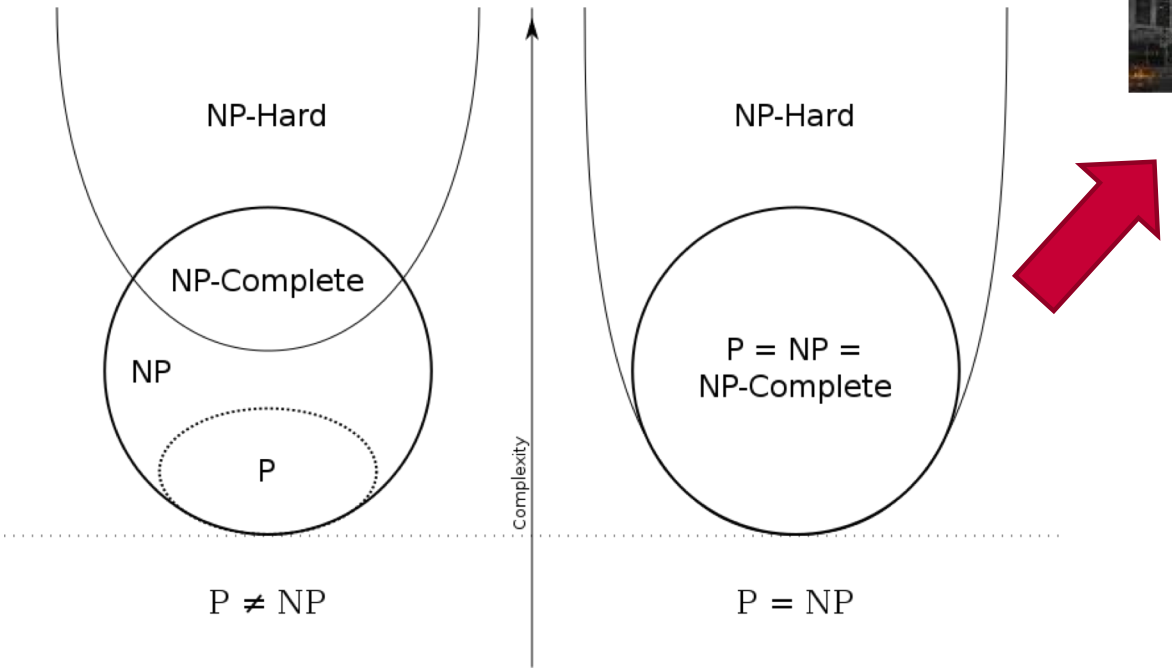


P vs. NP

- Although poly time verifiability ***seems*** like a weaker condition than poly time solvability, no one has been able to prove that it is weaker (i.e., describes a larger class of problems)

- So it is unknown whether P = NP.

P and NP



Current belief

Difficult, but still possible (unlikely)

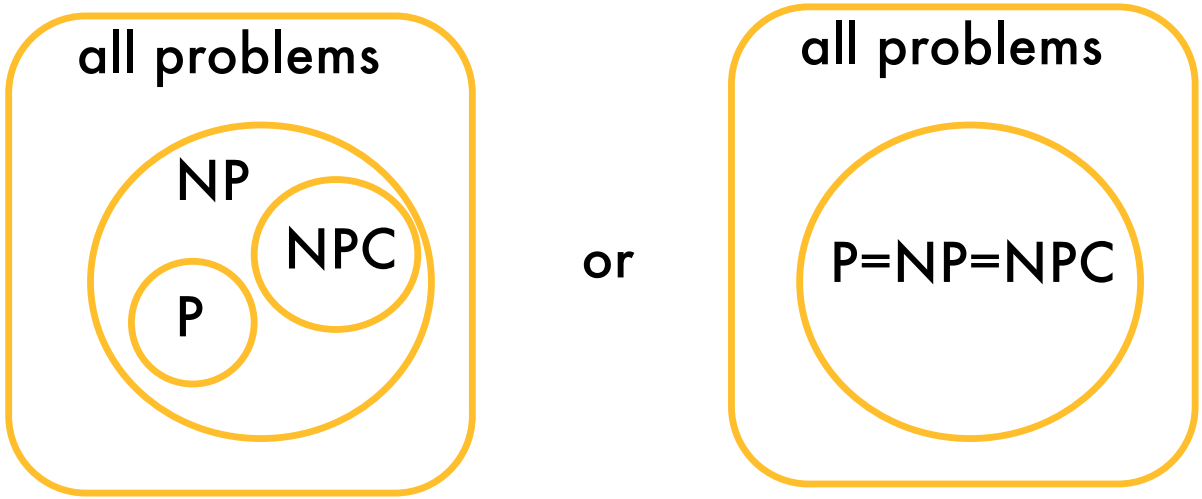
NP-Complete and NP-Hard Problems

- NP-complete problems is class of ***hardest*** problems in NP.
  - Every problem in NP can be reduced to the NP-complete problem in polynomial time (NP-Hard), so we can say that an NP-complete problem is at least as hard as all the problems in NP.
  - NP-hard problems may not necessarily be verifiable in polynomial time
- Reduce in polynomial time? Timetabling problem to SAT, if polytime alg for SAT and polytime reduction to convert Timetabling to SAT, then polytime alg for timetabling!
- If an NP-complete problem can be solved in polynomial time, then all problems in NP can be, and thus P = NP.
- Small subset of problems; the general idea is that if we can reduce a problem X to a problem Y in polynomial time, and problem Y is NP-complete then so must X be.

TSP?

- The TSP is not in NP since an (optimal) solution (the shortest route through all cities) is not verifiable in polynomial time
  - (TSP is an *NP-Hard* problem)
  - The decision variant of the problem is NP-complete: given a limit X is there a solution that is less than this limit, is there a route whose distance is less than this value

Possible Worlds

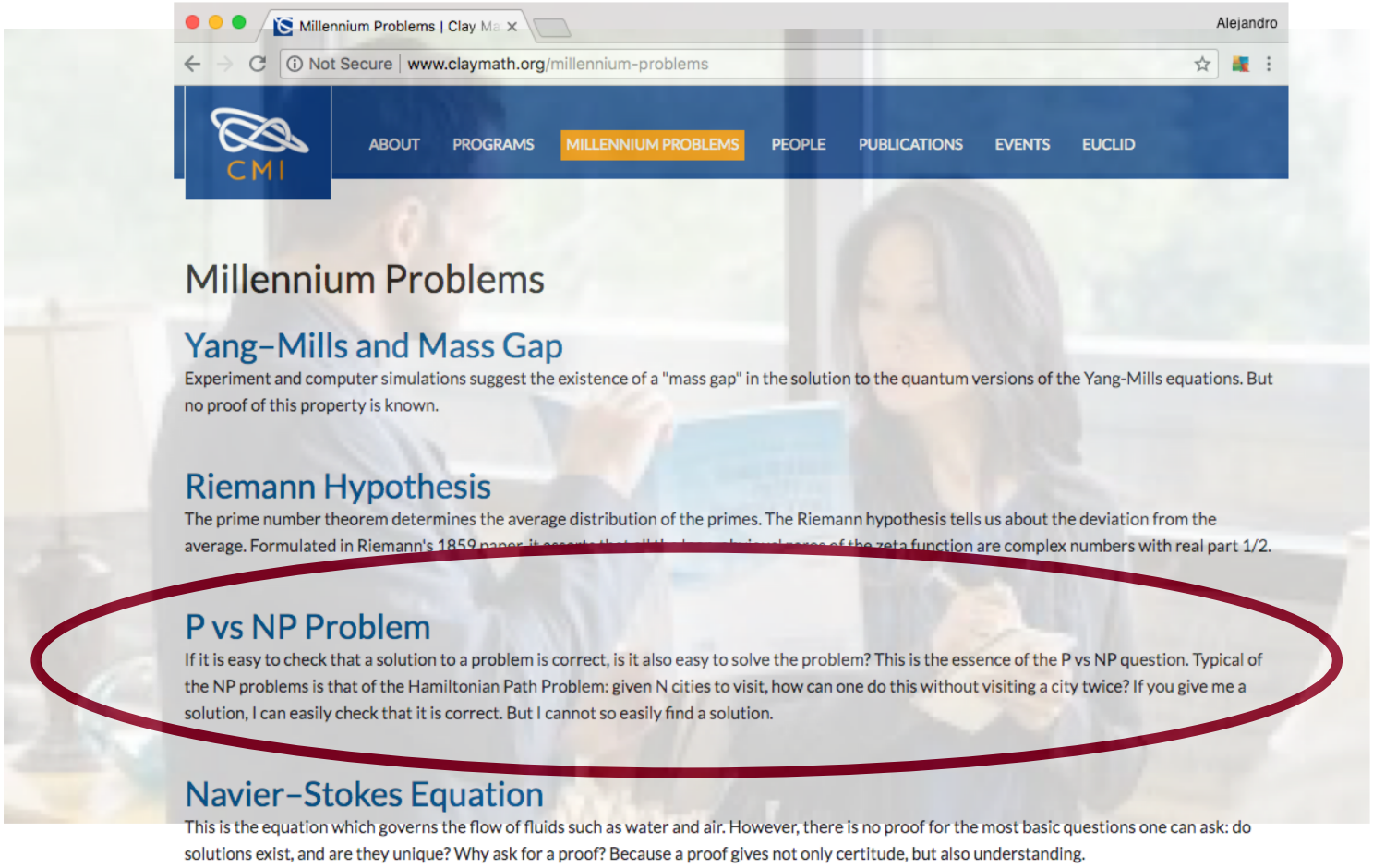


NPC = NP-complete

P = NP Question

- Open question since about 1970
- Great theoretical interest
- Great practical importance:
  - If your problem is NP-complete, then don't waste time looking for an efficient algorithm
  - Instead look for efficient approximations, heuristics, etc.
- Solve for a million dollars!
  - http://www.claymath.org/millennium-problems
  - The Poincare conjecture is solved today

Million dollar problem



Decision Problems

NP-Completeness Theory

As we have already mentioned, the theory is based considering decision problems.

Example:

- Does there exist a path from node u to node v in graph G with at most k edges.
- Instead of: What is the length of the shortest path from u to v? Or even: What is the shortest path from u to v?

Decision Problems

Why focus on decision problems?

- Solving the general problem is at least as hard as solving the decision problem version

- For many natural problems, we only need polynomial additional time to solve the general problem if we already have a solution to the decision problem

Definition of P

- P is the set of all decision problems that can be computed in time $O(n^k)$, where n is the length of the input string and k is a constant

- "Computed" means there is an algorithm that correctly returns YES or NO whether the input string is in the language

Example of a Decision Problem in NP

- Decision problem: Does G have a Hamiltonian cycle?
- Candidate solution: a sequence of nodes $v_0, v_1, \ldots, v_\ell$
- To verify:
  - check if ℓ = number of nodes in G
  - check if $v_0 = v_\ell$ and there are no repeats in $v_0, v_1, \ldots, v_{\ell-1}$
  - check if each (v_i, v_{i+1}) is an edge of G

Going From Verifying to Solving


```
for each candidate solution do
    verify if the candidate really works
    if so then return YES
return NO
```

Difficult to use in practice, though, if number of candidate solutions is large

Number of Candidate Solutions

- *"**Is there a path from u to v in G of length at most k?**"*: more than n! candidate solutions where n is the number of nodes in G

- *"**Does G have a Hamiltonian cycle?**"*: n! candidate solutions

Polynomial Reduction

Polynomial Reduction

Given a problem P which can be solved in polynomial time with algorithm A^*.
Suppose you are given another problem P' that seems similar to P.
How might you solve P'?

- You could try to solve P' from scratch.
- You could try to borrow elements of A^*.
- You could try to find a reduction from P' to P.

A **reduction** of P' to P

1. Transforms inputs to P' into inputs to P to create P''
2. Run A^* on P''
3. Transform the output (solution to P'') into solution to P'.

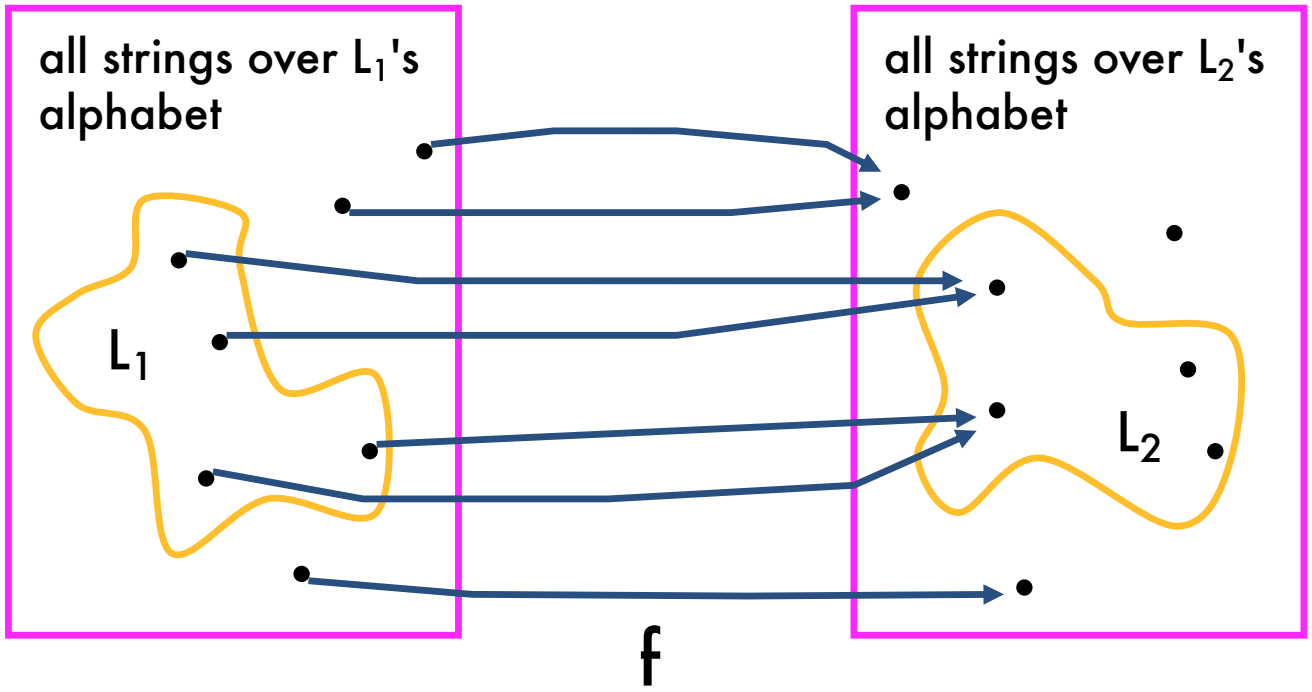
Polynomial Reduction

A ***polynomial reduction*** (or ***transformation***) from language L_1 to language L_2 is a function f from strings over L_1's alphabet to strings over L_2's alphabet such that

(1) f is computable in polynomial time

(2) for all x, x is in L_1 if and only if f(x) is in L_2

Polynomial Reduction



Polynomial Reduction

- YES instances map to YES instances
- NO instances map to NO instances
- computable in polynomial time
- Notation: $L_1 \leq_p L_2$

  Think of it as: L_2 is at least as hard as L_1

Polynomial Reduction Theorem

Theorem: **If** $L_1 \leq_p L_2$ **and** L_2 is in P,

then L_1 is in P.

Proof: Let A_2 be a polynomial time algorithm for L_2. Here is a polynomial time algorithm A_1 for L_1.

input: x

compute f(x)

run A_2 on input f(x)

return whatever A_2 returns

Implications

- Suppose that $L_1 \leq_p L_2$
- If there is a polynomial time algorithm for L_2, then there is a polynomial time algorithm for L_1.
- If there is no polynomial time algorithm for L_1, then there is no polynomial time algorithm for L_2.
- Note the asymmetry!

NP-Completeness

Definition of NP-Complete

L is NP-complete if and only if

(1) L is in NP and

(2) for all L' in NP, $L' \leq_p L$.

In other words, L is at least as hard as every language in NP.

Implication of NP-Completeness

Theorem: Suppose L is NP-complete.

(a) If there is a poly time algorithm for L, then P = NP.

(b) If there is no poly time algorithm for L, then there is no poly time algorithm for any NP-complete language.

Showing NP-Completeness

- How to show that a problem (language) L is NP-complete?

- Direct approach: Show

  (1) L is in NP

  (2) **every** other language in NP is polynomially reducible to L.

- Better approach: once we know some NP-complete problems, we can use reduction to show other problems are also NP-complete. How?

Showing NP-Completeness with a Reduction

To show L is NP-complete:

(1) Show L is in NP.

(2.a) Choose an appropriate known NP-complete language L'.

(2.b) Show $L' \leq_p L$.

Why does this work? By transitivity: Since every language L'' in NP is polynomially reducible to L', L'' is also polynomially reducible to L.