ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Special Topics on Algorithms

Fall 2023

The classes P and NP

Coping with NP-completeness

Vangelis Markakis – George Zois
markakis@gmail.com
georzois@gmail.com

Complexity Class P

P =

All problems Π, such that for every instance I∈Π, there exists an algorithm with worst case complexity O(p(|I|)), for some polynomial p -- denoted also as O(poly(|I|)))

Known Problems in P: finding min/max: O(n)

sorting: O(n log n)

Integer multiplication: $O(n^{1.59})$

GCD(a,b), a>b: O(log a)

Primality testing: $O(\log^{12} n)$ [August 2002]

… many others …

Problems without a known polynomial time algorithm:

SUBSET SUM : O(nB) [recall: pseudo-polynomial]

SAT: $O(2^n)$

… many many others…

Decision and Optimization Problems

Decision problems: problems where the answer is YES or NO

e.g. Primality testing, SUBSET SUM, SAT,...

Optimization problems: maximize/minimize some objective function

TSP (Traveling Salesman Problem)

I: A complete weighted digraph G = (V, E)

Q: Find a minimum weight tour of G

(tour: a cycle visiting each node exactly once)

CLIQUE

I: A graph G = (V,E)

Q: Find the maximum subset $C \subseteq V$ s. t. $\forall$ u,v $\in$ C: (u,v) $\in$ E

(the maximum complete subgraph of G)

Decision and Optimization Problems

An optimization problem has three versions/questions:

Function version: find an optimal feasible solution

Evaluation version: find the cost of an optimal feasible solution

Decision version: Given a bound B, is there a feasible solution of value $\leq$ B (for minimization problems) or of value $\geq$ B (for maximization problems)

<u>TSP</u>

Q1: Find a tour of minimum cost

Q2: Find the actual cost of an optimal tour

Q3: Given a bound B, is there a tour of cost $\leq$ B ?

<u>CLIQUE</u>

Q1: Find the vertices of a maximum clique C

Q2: Find the size of C

Q3: Given a bound B, is there a clique $C \subseteq V$ such that $|C| \geq B$?

For any optimization problem we can state its corresponding decision version

Decision and Optimization Problems

- Complexity theory is mostly built around decision problems
- they are used to define complexity classes
- the decision version of an optimization problem is equivalent to its function and evaluation versions!

Decision and Optimization Problems

For all the problems that we have seen and will see:

Given an algorithm for the decision version of an optimization problem, there exists a polynomial time algorithm to answer both its evaluation and function versions

Example: TSP

(1) decision → evaluation

Apply the question "is there a tour of cost $\leq$ B" for several values of B

For what values of B ? (not for all)

Optimal value upper bounded by the sum of the first n weights

Hence, apply binary search in this range

How many calls needed to the decision version?

O(log(sum of n largest weights)) = O(poly(|I|))

Hence a polynomial “reduction” from decision to evaluation

Decision and Optimization Problems

Example: TSP (cont.)

(2) decision → function

Use (1) to find the cost of an optimal solution, say B*

```
T:= { }   //T will store an optimal tour
For each edge e ∈ E do
    { x :=w(e);
      w(e) := w(e) + M; //M is some positive number > B* , e.g. M = B*+1.
      "is there a tour of cost ≤ B* ? "
      if NO then
          { T=T U {e} ; //e is contained in an optimal tour
            w(e):=x   } //restore the weight of e
      // if YES then e is not needed for finding an optimal tour
      // keep its weight to w(e) + M
```

Complexity $O(|E|) \sim O(n^2)$

The Complexity Class NP

For a decision problem Π, an instance I ∈ Π is a

- yes-instance, if there exists a solution to the question posed by I
- no-instance, otherwise

The class NP - high level definition:

- A problem Π is in NP if we can verify efficiently the validity of a candidate solution
  - i.e., if someone presents to us a candidate solution, we can answer in poly-time if it is indeed a solution to the problem
  - Thus, for a yes instance, there is a way to verify that it is indeed a yes instance

In TSP: a candidate solution (a certificate) = one of the possible tours

- Consider a decision instance of TSP with bound B
- If someone presents to us a candidate solution
  - We can check that it is indeed a tour
  - We can sum up the weights of the tour and check if they exceed B or not
- Hence in poly-time we can verify if the candidate solution is an actual solution

The Complexity Class NP

For a decision problem Π, an instance $I \in \Pi$ is a

- yes-instance, if there exists a solution to the question posed by I
- no-instance, otherwise

The class NP – formal definition:

Definition: A problem Π is in NP if and only if there is a polynomial time verification algorithm A such that: for every yes-instance $I \in \Pi$, there is a certificate x with $|x| \leq poly(|I|)$, such that $A(I, x) = yes$

In complexity theory terms: NP = all problems for which there is a non-deterministic polynomial time algorithm

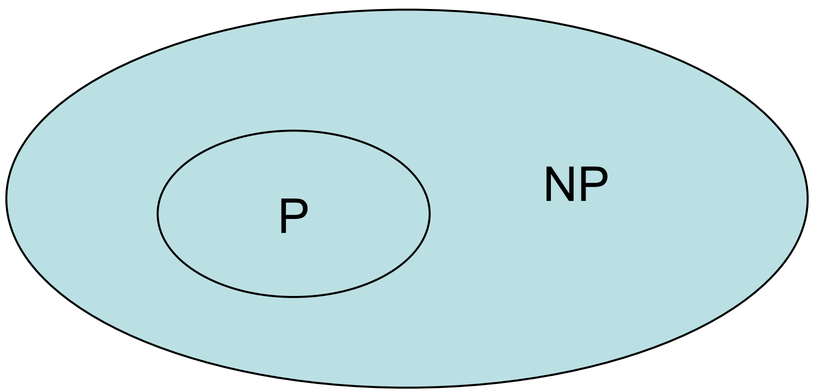
Note: Verifying yes-instances does not imply we can do the same for no-instances

- For TSP: the only way to convince someone for a no-instance would be to check that all tours have cost > B

P versus NP

If for a problem Π, we have a polynomial time algorithm that solves it, then, we can obviously validate a yes-instance in polynomial time

Hence: **P** $\subseteq$ **NP**

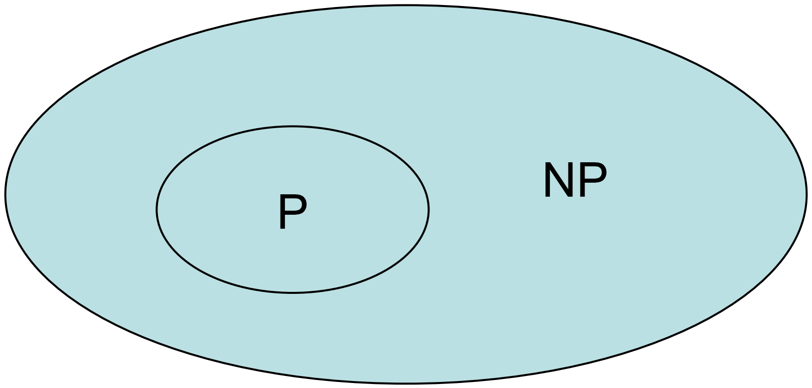


What about the reverse direction? Million dollar question!!
http://www.claymath.org/millennium-problems

P versus NP

If for a problem Π, we have a polynomial time algorithm that solves it, then, we can obviously validate a yes-instance in polynomial time

Hence: **P** $\subseteq$ **NP**



Philosophically: the P≠ NP conjecture supports that it is easier to verify a yes-instance than decide if an instance is yes or no

- Verification is strictly easier than actually solving the problem

The class of NP-complete problems

A problem $\Pi \in NP$ is NP-complete iff all problems in NP polynomially reduce to Π

- Equivalently: for every other problem $\Pi' \in NP$: $\Pi' \leq_p \Pi$, where $\leq_p$ denotes a Karp reduction (as you saw it in the Algorithms course)

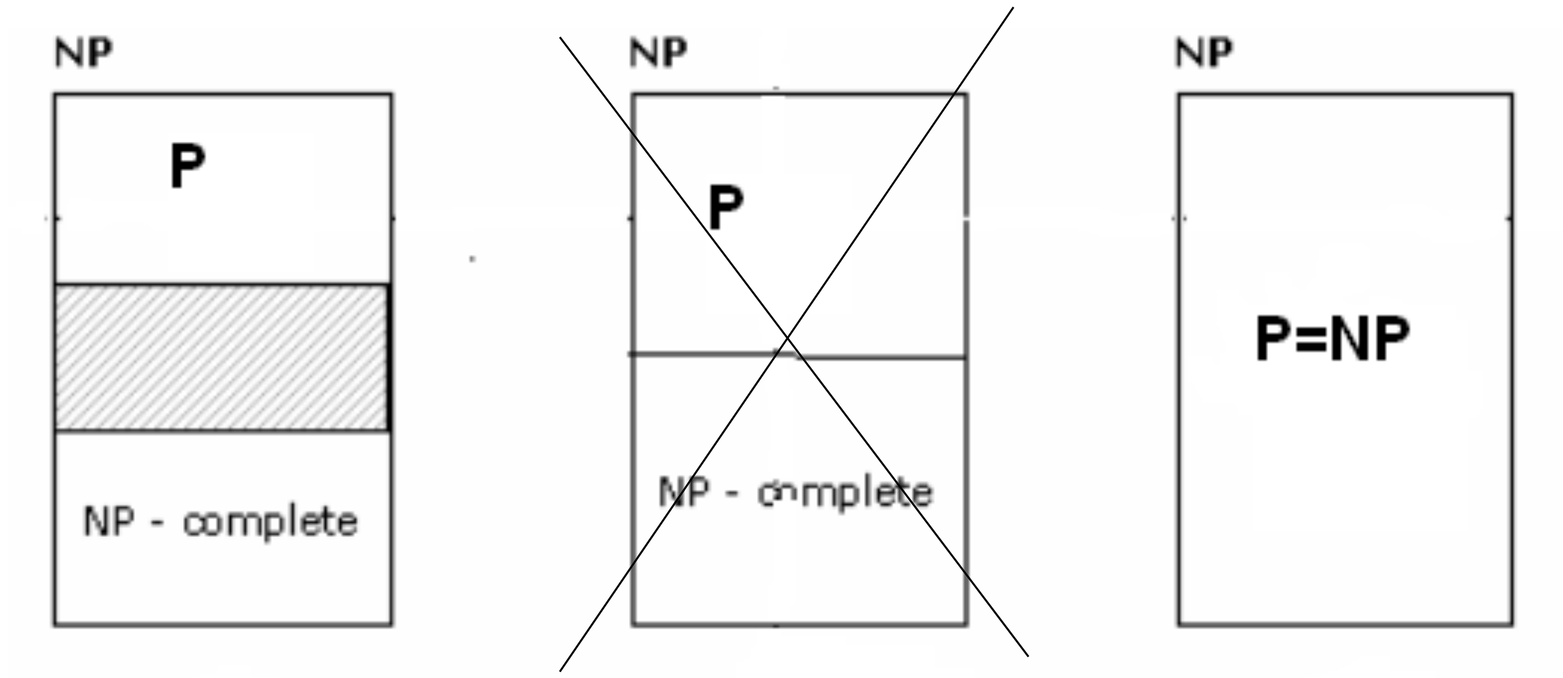
NP-completeness:

- captures the essence and the difficulty of NP
- identifies the most difficult problems in NP

To prove that a problem Π is NP-complete:

1. prove that $\Pi \in NP$
2. prove that $\forall \Pi' \in NP$: $\Pi' \leq_p \Pi$

The class of NP-complete problems



the common belief

Not possible

Not widely believed

Ladner's theorem

Beyond NP: there exist even more difficult problems, in complexity classes that contain NP (not within the scope of this course)

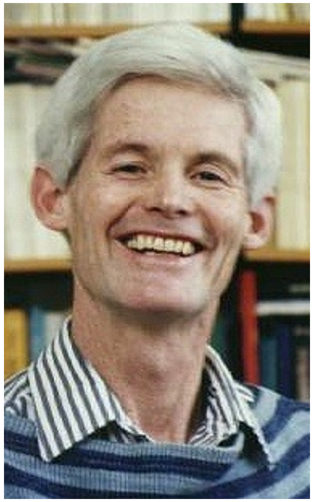
The class of NP-complete problems

But how do we start proving NP-completeness results?

Fortunately we have:

Cook's Theorem [Cook 1970, Levin 1972]:

SAT is NP –Complete (for every $\Pi' \in NP$: $\Pi' \leq_p SAT$)



S. Cook

- Hence we have a starting point!
- We can derive now more NP-complete problems by reducing from SAT
- It suffices to provide a reduction from any known NP-complete problem

L. Levin

More NP-complete problems

<u>CLIQUE</u>:

I: A graph $G = (V,E)$, an integer $B \leq |V|$

Q: Is there $C \subseteq V$ s.t. $\forall$ u,v $\in$ C: (u, v) $\in$ E and $|C| \geq B$?

<u>VERTEX COVER (VC)</u>:

I: A graph $G = (V,E)$, an integer $B \leq |V|$

Q: Is there $S \subseteq V$ s.t. $\forall$ (u, v) $\in$ E either $u \in S$ or $v \in S$ (or both) and $|S| \leq B$?

<u>INDEPENDENT SET (IS)</u>:

I: A graph $G = (V,E)$, an integer $B \leq |V|$

Q: Is there $I \subseteq V$ s.t. $\forall$ u, v $\in$ I: (u, v) $\notin$ E and $|I| \geq B$?

<u>3-GRAPH COLORABILITY (3-GC)</u>:

I: A graph $G = (V,E)$

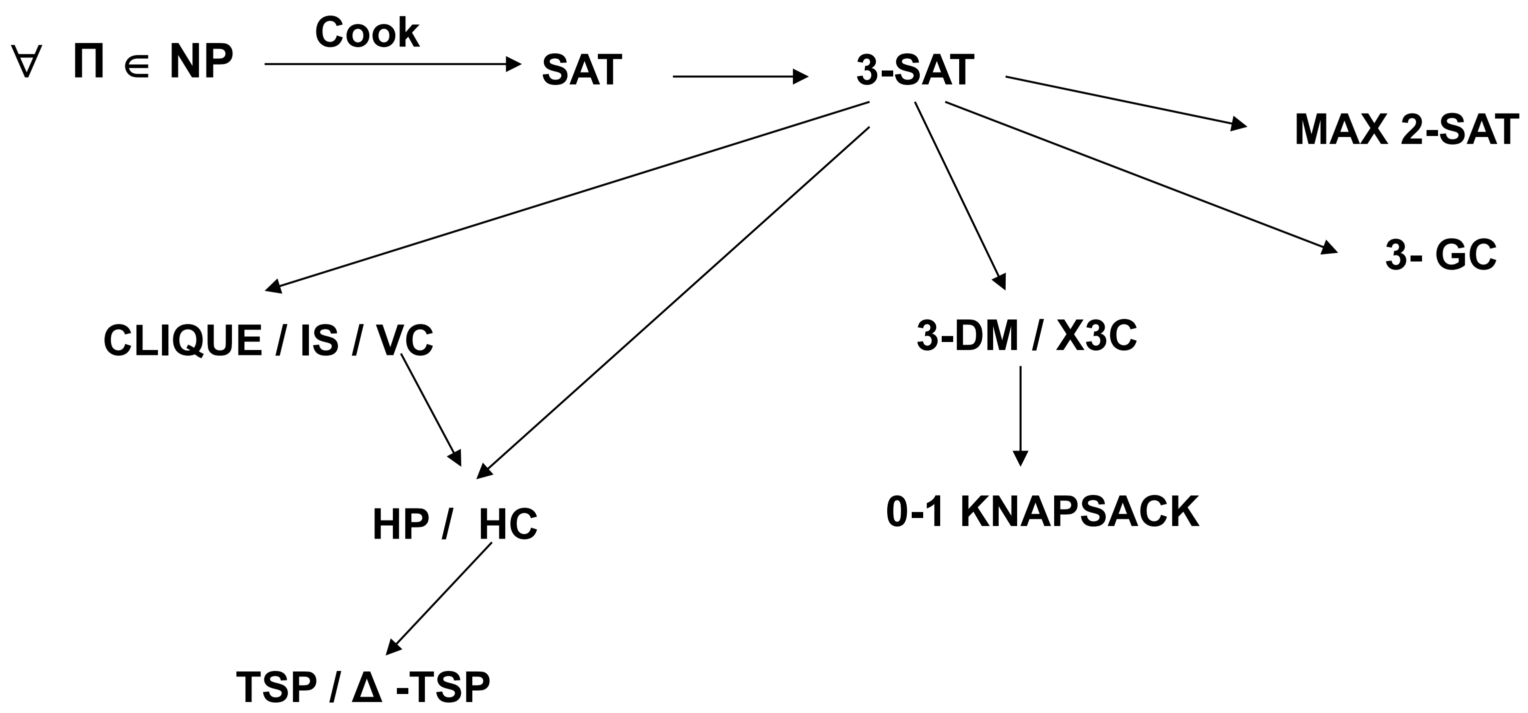
Q: Is there a function $f: V \rightarrow \{1,2,3\}$ s.t. $\forall$ (u, v) $\in$ E: $f(u) \neq f(v)$?

P vs NP

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

No poly-time algorithm is known for an NP-complete problem

A Tree of reductions for some problems



Coping with NP-complete problems

1. Algorithms for small instances
2. Algorithms for special cases
3. Exponential algorithms
4. Approximation algorithms
5. Randomized algorithms
6. Heuristic algorithms

Coping with NP-complete problems

1. Algorithms for small instances

If we want to run an algorithm in small instances only, then an exponential algorithm may be satisfactory

2. Algorithms for special cases

Identify important families of instances where we can have an efficient algorithm, e.g., 2-SAT

- Some times an actual application may only deal with specific types of instances

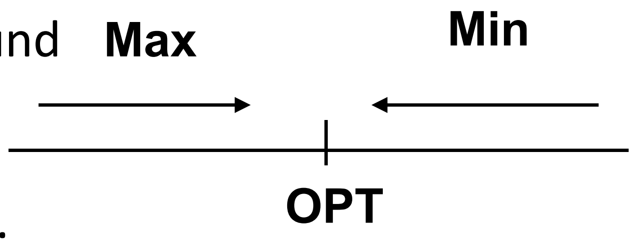
3. Exponential algorithms

Some worst-case exponential algorithms may still be better than brute-force or have a good average-case behavior: Pseudo-polynomial algorithms, Dynamic Programming, Backtracking, Branch-and-Bound

Approximation algorithms

4. Approximation algorithms

algorithms for which we can have a provable bound on the quality of the solution returned



Given an instance I of an optimization problem Π:

•OPT(I) = optimal solution

•C(I) = cost of solution returned by the algorithm under consideration

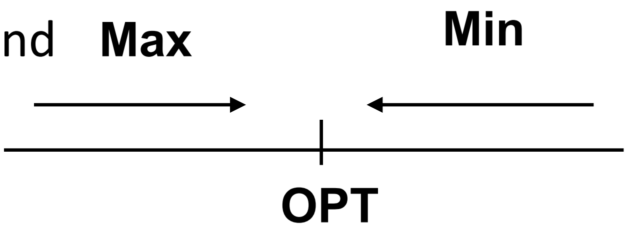
Definition: An algorithm A, for a minimization problem Π, achieves an approximation factor of ρ (ρ ≥ 1), if for **every** instance I of the problem, A returns a solution with:

$$C(I) \leq \rho \, OPT(I)$$

Approximation algorithms

4. Approximation algorithms

algorithms for which we can have a provable bound on the quality of the solution returned



Given an instance I of an optimization problem Π:

•OPT(I) = optimal solution

•C(I) = cost of solution returned by the algorithm under consideration

Definition for maximization: An algorithm A, for a maximization problem Π, achieves an approximation factor of ρ (ρ ≤ 1), if for **every** instance I of the problem, A returns a solution with:

$$C(I) \geq \rho \; OPT(I)$$

Approximations: Good, better, best and more …

Non - constant approximation: $C(I)/OPT(I) \leq f(n)$ for some function that depends on n

Constant (ρ-)approximation: $C(I)/OPT(I) \leq \rho$, where ρ is a constant, e.g. 3/2

Polynomial Time Approximation Schemes (PTAS):

- $C(I)/OPT(I) \leq 1 + \varepsilon$, for any $\varepsilon > 0$ (any constant factor is achievable)
- Complexity should be O(poly (|I|)) and O(exp (1/ε)), e.g. $O(n^{3/\varepsilon})$

Fully Polynomial Time Approximation Schemes (FPTAS):

- $C(I)/OPT(I) \leq 1 + \varepsilon$, for any $\varepsilon > 0$
- Complexity should be O(poly (|I|)), O(poly (1/ ε)) !!!
- e.g. $O((1/\varepsilon)^2 n^3)$, dependence on 1/ε should not be on the exponent

Additive approximation:

- $C(I) \leq OPT(I) + f(n)$ or $C(I) \leq OPT(I) + k$ (a constant), e.g. $C \leq OPT+1$

Coping with NP-complete problems

5. Randomized algorithms

algorithms that use randomization (e.g. flipping coins) and make randomized decisions

Performace: Such algorithms may

- produce a good solution with high probability
- Produce a good expected cost
- Run in expected polynomial time

Power of randomization: for some problems, the only decent algorithms known are randomized!

Coping with NP-complete problems

6. Heuristic algorithms

Algorithms that are typically fast and work well in practice but without a formal guarantee of their performance (e.g., many local search approaches)

- No guarantee on the approximation achieved by the solution returned
- Some times no guarantee that they even terminate in polynomial time