COMP2230 Algorithms

Lecture 10

Professor Ljiljana Brankovic

Lecture Overview

P and NP

Text, Chapter 10

Lecture Overview

- Motivating Examples
- The Complexity of Computations, Algorithms and Problems
- Complexity Classes: P, NP and NP-complete
- $NP-$ completeness proofs

Example 1 - Motivating

Suppose that you were given the following 3 problems to solve:

1. Given a map containing cities and roads between cities, design a route that starts at a given city, traverses each road exactly once and returns to the starting city (if there is such a route).

2. Given a map containing cities, roads between cities, and the length of each road, design a minimum length route that starts at a given city, visits each city exactly once and returns to the starting city.

3. Design an algorithm that takes as input a program and an input to it and determines whether or not the program will halt on that input.

Example 1 – Three Motivating Examples

1. The first problem is known as finding an Eulerian cycle in a graph and can be solved in polynomial time.

2. The second problem is known as Travelling Salesperson Problem (TSP) and there is no known polynomial time algorithm for solving it.

3. The third problem is the so-called Halting problem; this problem is "undecidable", that is, there is no algorithm that can solve this problem.

Intractable Problems

Problems that cannot be solved in polynomial time are called "intractable". Intractable problems are either so difficult that a solution cannot be found in polynomial time, or the solution itself is so extensive that it cannot be described in polynomial time (e.g., list all possible routes a travelling salesman can take – the number of routes is exponential)

We are primarily concerned with problems for which neither there are known polynomial time algorithms, nor it has been proved that such algorithms do not exist.

Decision Problems

We focus our attention on so-called "decision" problems. These are problems which only require "yes" or "no" answer, e.g., "Given a graph G, is there an Eulerian cycle in G?"

All instances of a decision problem can be divided into "yes" instances (those for which the answer is yes) and "no" instances (those for which the answer is no).

Example 2: It is known that a graph has an Eulerian cycle if and only if it is connected and all its vertices have even degrees. Thus in the Eulerian cycle problem, all connected graphs in which every vertex has an even degree correspond to "yes" instances of the problem, and all the other graphs to "no" instances.

Turing machines

In what follows we shall use a Turing machine as a general model of computation.

Recall that a Turing machine has an infinite tape divided into cells, and a read-write head that can read the content of a cell, overwrite the content of a cell, or move left or right one step at the time. A Turing machine has a finite number of states and based on the current state and the content of the current cell it makes decision on what action to take (overwrite the cell and/or move one cell to the left or right) and what new state to go to.

Any problem that can be solved on any Random Access Machine (RAM) in polynomial time, can also be solved on a Turing machine in polynomial time.

Complexity of Computations

- The ***time complexity*** of a computation is the amount of time required to perform the computation.
- The ***space complexity*** of a computation is the space required to perform the computation.
- The time complexity of a Turing machine computation is the number of single transitions executed during the computation.
- The space complexity of a Turing machine computation is the number of tape cells required by the computation.
- If the time complexity of a Turing machine computation is n, then the space complexity of the computation is no more than $n + 1$.

Complexity of Algorithms

A Turing machine can be seen as the implementation of a single algorithm.

In general, the time complexity of a Turing machine computation is a function of the length of the input.

- best-case, worst-case and average-case performance
- average-case :

$$\sum p_i c_i$$

where p_i is the probability of computation with complexity c_i

Complexity of Problems

- The complexity of a problem is not the same as the complexity of a particular algorithm.

- We may say that the complexity of a problem is the complexity of its simplest (most efficient) solution.

- The complexity of a problem is in the class $\Theta(f)$ if it can be solved by an algorithm whose complexity is in the class $\Theta(f)$ and any better solution also has the complexity in $\Theta(f)$.

Definition 1: Let $t: N \to N$ be a function. We define the ***time complexity class***, $TIME(t(n))$, to be

$$TIME(t(n)) = \{ L \mid L \text{ is a problem decided by a } O(t(n)) \text{time Turing machine} \}$$

Complexity of Problems

Example 3:

Consider the language $A = \{0^k 1^k | k \geq 0\}$, and the following algorithm for deciding whether or not a string ω is in the language A:

M_1 = "On input string ω:

- Scan across the tape and reject if a 0 is found to the right of a 1.
- Repeat the following if both 0's and 1's remain on the tape:

  Scan across the tape, crossing off a single 0 and a single 1.
- If 0's still remain after all the 1s have been crossed off, or if 1's still remain after all the 0s have been crossed off, reject; otherwise, accept."

Complexity of Problems

Example 3 (cont'd):

M_1 decides A in time $O(n^2)$, and thus $A \in TIME(n^2)$.

M_2 decides A in time $O(n \log n)$, and thus $A \in TIME(n \log n)$.

M_2 = "On input string ω:

- Scan across the tape and reject if a 0 is found to the right of a 1.
- Repeat the following if both 0's and 1's remain on the tape:
  - Scan across the tape, checking if total number of 0s and 1s remaining is even; if not, reject.
  - Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
- If no 0's and no 1's remain on the tape, accept; otherwise, reject."

Complexity of Problems

Example 3 (cont'd):

M_3 is a two-tape TM that decides A in time $O(n)$.

M_3 = "On input string ω:

- Scan across the tape and reject if a 0 is found to the right of a 1.
- Scan across the 0s on T_1 until the first 1. At the same time, copy the 0s onto T_2.
- Scan across the 1s on T_1 until the end of input. For each 1 on T_1, cross off a 0 on T_2. If all the 0s are crossed off before all 1s are read, reject.
- If all the 0s have been crossed off, accept; otherwise, reject."

Complexity of Problems

Theorem 1: Let $t(n)$ be a function such that $t(n) \geq n$. Then every $t(n)$ time multiple-tape Turing machine has an equivalent $O(t^2(n)$ time single-tape Turing machine.

Definition 2: Let N be a nondeterministic Turing decider. The running time of N is the function $f: N \to N$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

Theorem 2: Let $t(n)$ be a function such that $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

The Class P

Definition 3: The class P is the class of decision problems that are decidable in polynomial time on a deterministic single-tape Turing machine.

$$P = \bigcup_k TIME(n^k)$$

- P is invariant for all computational models that are polynomially equivalent to the deterministic single-tape Turing machine.
- P roughly corresponds to the class of problems that are realistically solvable on a computer.

Example 4: Let G be a directed graph and let s and t be two vertices of G $(s, t \in V(G))$. The PATH problem is to determine whether there exists a directed path from s to t.

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

Theorem 3: $PATH \in P$.

The Class NP

Definition 4: A nondeterministic algorithm can be thoughts of as a two stage algorithm; in the first stage, algorithm ***guesses*** a solution to the problem; in the second stage the algorithm verifies the solution.

A ***verifier*** for a problem A is an algorithm V which takes as input an instance ω and a candidate solution c ("guess" or "witness"), and verifies it.

A ***polynomial time verifier*** runs in polynomial time in length of ω.

Definition 5: The class NP is the class of decision problems (languages) that have polynomial time verifiers.

Theorem 4: A language is in NP if and only if it is decided by some nondeterministic polynomial time Turing machine.

The Class NP

Definition 6:

$$NTIME(t(n)) = \{ L \mid L \text{ is a problem decided by a } O(t(n)) \text{ time nondeterministic Turing machine} \}$$

Corollary: $NP = \bigcup_k NTIME(n^k)$

Example 5: A clique in an undirected graph G is a subgraph of G such that every two vertices in the clique are connected by an edge. A k-clique is a clique that contains k vertices.

$$\text{CLIQUE} = \{\langle G, k\rangle | G \text{ is an undirected graph with a } k-\text{clique}\}$$

The Class NP

Theorem 5: $CLIQUE \in NP$

Proof:

A polynomial time verifier:

V = "On input $\langle \langle G, k \rangle, c \rangle$:

- Test whether c is a set of k vertices in G.
- Test whether every two vertices in c are connected by an edge in G.
- If both answers are yes, accept; otherwise, reject."

The Class NP

Alternative Proof:

A polynomial time NTM:

N = "On input $\langle G, k \rangle$:

- Non-deterministically select a subset c of k nodes of G.
- Test whether every two vertices in c are connected by an edge in G.
- If yes, accept; otherwise, reject."

The Class NP

Example 6:

$SUBSETSUM =$
$\{\langle S, t\rangle \mid S = \{x_1, x_2, \dots, x_k\} \text{ and for some } \{y_1, y_2, \dots, y_s\} \subseteq \{x_1, x_2, \dots, x_k\} \text{ we have } \sum y_i = t\}$

Theorem 6: $SUBSETSUM \in NP$

Proof:

A polynomial time verifier:

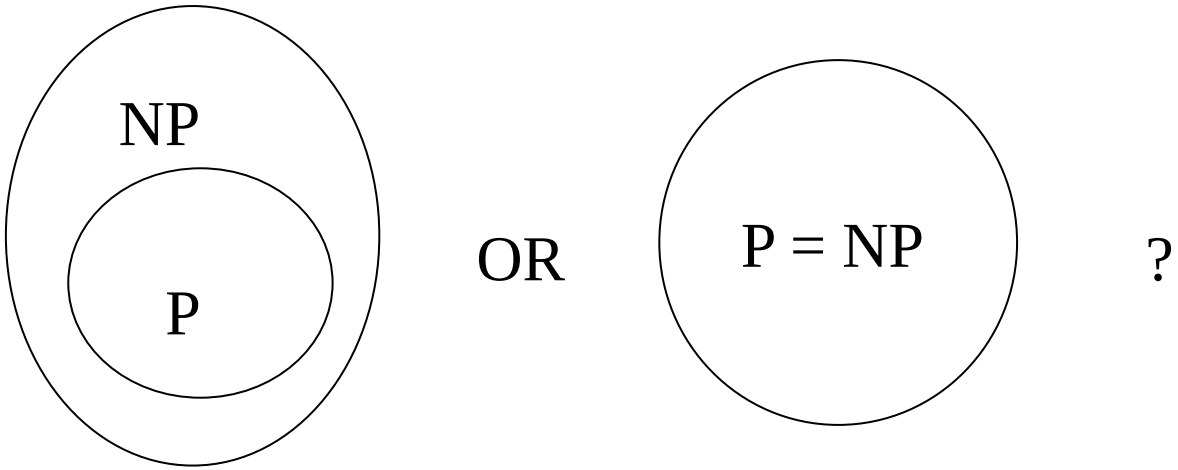
V = "On input $\langle\langle S, t\rangle, c\rangle$:

- Test whether c is a collection of numbers that sum to t.
- Test whether $c \subseteq S$.
- If both answers are yes, accept; otherwise, reject."

P versus NP

- P is the class of decision problems that can be decided in polynomial time.
- NP is the class of decision problems that can be verified in polynomial time.

- The most outstanding unsolved question in computer science: $P \overset{?}{=} NP$



- ***NP*-complete** problems are the problems in NP with the following property: If a polynomial time algorithm exists for any of these problems, then all problems in NP would have a polynomial time algorithm.

Satisfiability Problem

- Boolean variables are variables that can take value TRUE (1) or FALSE (0).
- Boolean operations: AND, OR and NOT.
- A Boolean formula is an expression that involves Boolean variables and operations (evaluates to TRUE or FALSE).
- A Boolean formula is satisfiable if there exists an assignment of 0s and 1s to the variables such that the formula evaluates to 1 (TRUE).

Satisfiability Problem

The ***satisfiability problem (SAT)*** is to test whether a Boolean formula is satisfiable.

A ***literal*** is a Boolean variable or a negated Boolean variable.

A ***clause*** is several literals connected with ORs or ANDs.

A Boolean formula in ***conjunctive normal form*** comprises several clauses connected with AND, where literals in each clause are connected with ORs.

Example 7:

$$(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_3} \vee \overline{x_5} \vee \overline{x_6}) \wedge (\overline{x_3} \vee \overline{x_2} \vee \overline{x_5})$$

NP-Completeness

Cook-Levin theorem:

$$SAT \in P \text{ if and only if } P = NP$$

This theorem was independently proved by Leonid Levin and Stephen Cook in late 60s and early 70s.

Definition 7:

A function $f : \sum^* \rightarrow \sum^*$ is a polynomial time computable function if there exists a polynomial time Turing machine M that halts with $f(\omega)$ on its tape when started on input ω.

NP-Completeness

Definition 8:

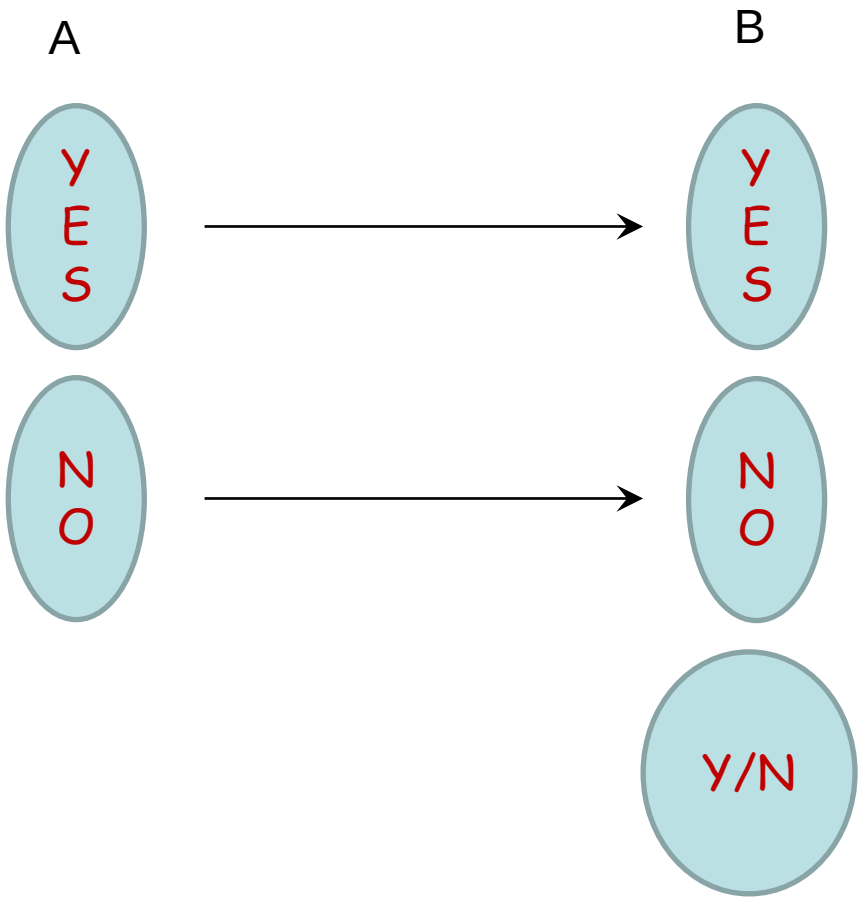
A problem A is polynomial time reducible to problem B, written $A \leq_p B$, if there exists a polynomial time computable function $f : \sum^* \rightarrow \sum^*$ such that for every ω,

$$\omega \in A \Leftrightarrow f(\omega) \in B.$$

The function f is called the polynomial time reduction from A to B.

In other words, the function f is a mapping from instances of problem A to the instances of problem B, such that every "yes" instance of problem A is reduced to a "yes" instance of problem B, and every "no" instance of problem A is reduced to a "no" instance of problem B.

NP-Completeness



NP-Completeness

Theorem 7:

If $A \leq_p B$ and $B \in P$, then $A \in P$.

Proof: Let M be a polynomial time algorithm deciding B and let f be the polynomial time reduction from A to B. Then N is a polynomial time algorithm that decides A.

N = "On input ω:

- Compute $f(\omega)$.
- Run M on input $f(\omega)$ and output what M outputs."

NP-Completeness

$3SAT$ is SAT in 3 conjunctive normal form (all clauses have exactly 3 literals).

Theorem 8: $3SAT$ is polynomial time reducible to Independent set (proof done in workshops).

Definition 9: A language B is NP-complete is it satisfies the following two conditions:

- B is in NP
- every A in NP is polynomial time reducible to B.

Theorem 9: If B is NP-complete and B is in P, then $P = NP$.

Theorem 10: If B is NP-complete and $B \leq_p C$ and C is in NP, then C is NP-complete.

NP-Completeness

Theorem 11 (Levin-Cook): SAT is NP-complete.

Theorem 12: $3SAT$ is NP-complete.

Proof: We use reduction from SAT. We first need to prove that $3SAT$ is in NP. Indeed, if we have a truth assignment for all the variables it is easy to check whether this assignment satisfies all the clauses; we can do this in polynomial time, by checking clause by clause ($O(m)$, where m is the number of clauses).

We next need to show that there exists a polynomial time reduction from SAT to $3SAT$, that takes an arbitrary instance of SAT and maps it into an instance of $3SAT$. Then we need to prove that the reduction can be performed in polynomial time and that the instance of SAT is a "yes" instance if and only if the corresponding instance of $3SAT$ is a "yes" instance.

NP-Completeness

Proof – cont'd:

Reduction: Take an arbitrary instance of SAT. Consider a clause by clause. We distinguish between 4 cases:

1. the clause has one literal
2. The clause has 2 literals
3. The clause has 3 literals
4. The clause has more than 3 literals

NP-Completeness

Proof – cont'd:

1. The clause has one literal.

   Let the original clause be (x); we replace it with the following set of clauses each of which has exactly 3 literals:

   $$(x, y_1, y_2), (x, \overline{y_1}, y_2), (x, y_1, \overline{y_2}), (x, \overline{y_1}, \overline{y_2})$$

   Here y_1 and y_2 are variables that appear in these four clauses and nowhere else. Note that for every truth assignment to y_1 and y_2 there will always be one clause that is not satisfied by y_1 and y_2, so x has to be true in order to satisfy all four clauses.

NP-Completeness

Proof – cont'd:

2. The clause has two literals.

   Let the original clause be (x_1, x_2); we replace it with the following set of clauses each of which has exactly 3 literals:

   $$(x_1, x_2, y), (x_1, x_2, \bar{y})$$

   Again, y is a variable that appear in these two clauses and nowhere else. Note that for every truth assignment to y there will always be one clause that is not satisfied by y, so x_1 or x_2 has to be true in order to satisfy all four clauses.

3. The clause has three literals.

   Let the original clause be (x_1, x_2, x_3); we keep this clause as it is.

NP-Completeness

Proof – cont'd:

4. The clause has more than three literals.

   Let the original clause be $(x_1, \ldots, x_k)$; we replace it with the following set of clauses each of which has exactly 3 literals:

   $$(x_1, x_2, \overline{y_1}), (y_1, x_3, \overline{y_2}), (y_2, x_4, \overline{y_3}), \ldots, (y_{k-3}, x_{k-1}, x_k)$$

   Here $y_1, \ldots, y_{k-3}$ are variables that appear in these $k-2$ clauses and nowhere else. Note that for every truth assignment to $y_1, \ldots, y_{k-3}$ there will always be one clause that is not satisfied by $y_1, \ldots, y_{k-3}$, so one of the $x_1, \ldots, x_k$ has to be true in order to satisfy all the clauses.

NP-Completeness

Proof – cont'd:

The final thing we need to show is that this reduction can be performed in polynomial time. Each clause can be constructed in constant time, since each clause has exactly 3 literals. Therefore we only need to show that the total number of clauses is polynomial in terms of the number n of variables and number m of clauses of the instance of the original SAT problem.

For each original clause with 1 literal we construct 4 clauses; for each clause with 2 literals, 2 clauses; for each clause with 3 literals, 1 clause, and for each clause with k literals, $k \geq 4$, $(k-2)$ clauses; since $k \leq 2n$, we have no more than $2mn$ clauses, therefore the reduction can be performed in polynomial time.

NP-Completeness

Theorem 13: Independent set is NP-complete.

Proof: We already proved that Independent set is in NP, and that $3SAT$ is polynomial time reducible to Independent set.