KUNGL TEKNISKA HÖGSKOLAN

VETENSKAP OCH KONST

International Master Program in System-on-Chip Design

L2: Computational Complexity

Reading material

- de Micheli pp. 42 - 53
- Garey & Johnson “Computers and Intractability: a guide to the theory of NP-completeness”, pp. 1 - 45

Computational complexity

- Computational complexity is an abstract measure of the time and space necessary to execute an algorithm as function of its input size
  - the input is the graph G(V,E)
    - input size = $|V|$ and $|E|$
  - the input is the truth table of an n-variable Boolean function
    - input size = 2^n

Time and space complexity

- Time complexity is expressed in elementary computational steps
  - example: addition (or multiplication, or value assignment etc.) is one step
  - normally, by "most efficient" algorithm we mean the fastest
- Space complexity is expressed in memory locations
  - e.g. in bits, bytes, words

Big-O notation

- $f = O(g)$, if two constants n_0 and K can be found such that for all $n \geq n_0$:

$$f(n) \leq K \cdot g(n)$$

- Examples:

$2n^2 = O(n^2)$

$2n^2 + 3n + 1 = O(n^2)$

Examples of big-O for algorithms

- February 29th birthday problem - O(n)
- Two people with the same birthday problem - $O(n^2)$

Exponential Time Complexity

- An algorithm has an exponential time complexity if its execution time is given by the formula

$$\text{execution time} = k_1 \cdot (k_2)^n$$

where n is the size of the input data and k_1 and k_2 are constants

Exponential Time Complexity

- The execution time grows so fast that even the fastest computers cannot solve problems of practical sizes in a reasonable time
- The problem is called intractable if the best algorithm known to solve this problem requires exponential time
- Many CAD problems are intractable

Time complexity comparison

input size / function	10	20	30	40	50	60
n	.00001s	.00002s	.00003s	.00004s	.00005s	.00006s
n^2	.0001s	.0004s	.0009s	.0016s	.0025s	.0036s
2^n	.001s	1.0s	17.9min	12.7days	35.7years	366centures

Why do we need to know time complexity of the algorithm?

- Suppose you are a chief algorithm designer in some company
- You boss wants you to develop an efficient algorithm for solving some problem
- You are finding out that the problem is intractable

Solution 1

- You are going to your boss and saying:
  "I can't find an efficient algorithm, I guess I am too dump"

Solution 2

- You are going to your boss and saying:
  "I can't find an efficient algorithm, because no such algorithm exists"

Optimization and decision problems

- Optimization problems ask to find a solution which has minimum "cost" among all other solutions
  - e.g. mind a minimal sum-or-product expression for a given function
- Decision problems have only two possible solutions: "yes" or "no"
  - e.g. can a given function be represented as a sum of 3 products?

The satisfiability problem

- PROBLEM DEFINITION:

  Given a product-of-sum Boolean expression C of n variables which consists of m sums, is there a satisfying truth assignment for the variables?
- Example: n=4, m=2

$$C = (x_1 + x_2 + x_4)(x_1 + x_2 + x'_3)$$

the answer is "yes", if (1101) then C = 1

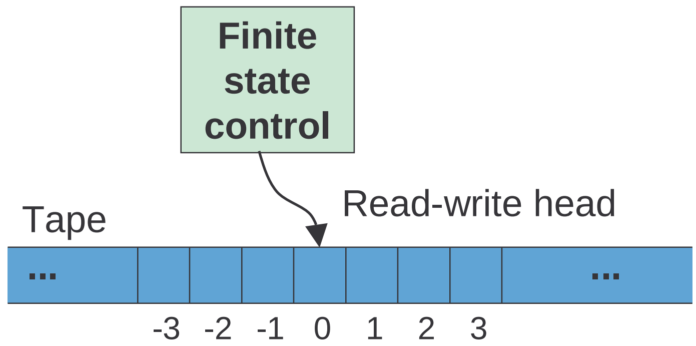
Complexity classes

- Class P contains those problems that can be solved in polynomial time (the number of computation steps necessary can be expressed as a polynomial of the input size n).
- The computer concerned is a deterministic Turing machine

Deterministic Turing machine

- Turing machine is a mathematical model of a universal computer
- any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine
- deterministic means that each step in a computation is predictable

Deterministic one-tape Turing machine

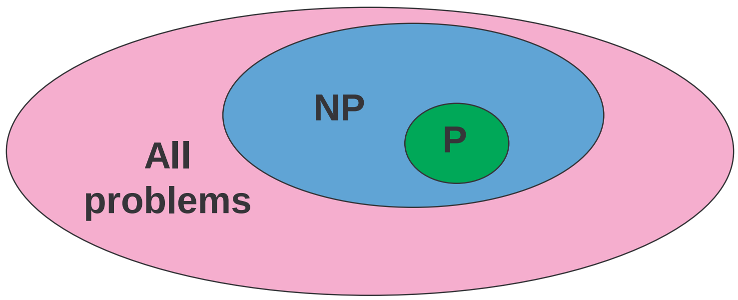


Non-deterministic Turing machine

- If solution checking for some problem can be done in polynomial time on a deterministic machine, then the problem can be solved in polynomial time on a non-deterministic Turing machine
- non-deterministic - 2 stages:
  - make a guess what the solution is
  - check whether the guess is correct

NP-class

- Class NP contains those problems that can be solved in polynomial time on a non-deterministic Turing machine



NP-complete problems

- An question which is still not answered:

$$P \subset NP \text{ or } P \neq NP$$

- There is a strong belief that $P \neq NP$, due to the existence of NP-complete (NPC) problems (NPC)
  - all NPC problems in have the same degree of difficulty: if one of them could be solved in polynomial time, all of them would have a polynomial time solution.

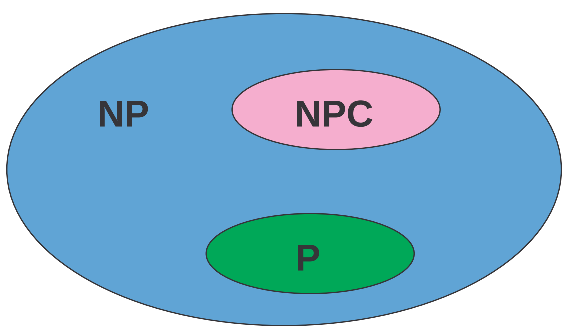
NP-complete problems

- A problem is NP-complete if and only if
  - it is in NP
  - some known NP-complete problem can be transformed to it in polynomial time

Cook's theorem:

SATISFIABILITY is NP-complete

World of NP, assuming P $\neq$ NP



NP-hard problems

- Any decision problem (inside or outside of NP) to which we can transform an NP-complete problem it in polynomial time will have a property that it cannot be solved in polynomial time, unless P = NP
- Such problems are called NP-hard
  - "as hard as the NP-complete problems"

Practical consequences

- Many problems in CAD for VLSI are NP-complete or NP-hard. Therefore:
  - exact solutions to such problems can only be found when the problem size is small.
  - one should otherwise be satisfied with sub-optimal solutions found by:
    - approximation algorithms: they can guarantee a solution within e.g. 20% of the optimum
    - heuristics: nothing can be said a priori about the quality of the solution

Example

- Tractable and intractable problems can be very similar:
  - the SHORTEST-PATH problem for undirected graphs is in P
  - the LONGEST-PATH problem for undirected graphs is NP-complete

Examples of NP complete problems

- Clique:

  instance: Graph G = (V,E), positive integer K ≤ |V|

  question: Does G contain a clique of size K or more?

- Minimum cover

  instance: collection C of subsets of a finite set S, positive integer K ≤ |C|

  question: Does C contain a cover for S of size K or less?

KUNGL TEKNISKA HÖGSKOLAN

International Master Program in System-on-Chip Design

Functions

Reading material

- de Micheli pp. 67 - 95, 288 - 294
- Muzio & Wesselkamper "Multiple-valued switching theory", p. 7 - 19

Binary relation

- Let A and B be sets. A binary relation R between A and B is a subset of the Cartesian product A x B
- Example: If A = {0,1}, B = {0,1,2}, then A x B = {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)}. If we define R as

  $(a,b) \in R$ iff $a = b$

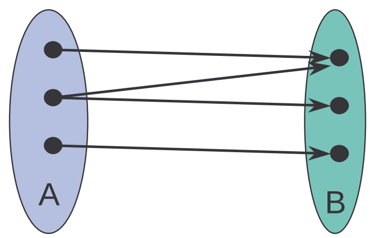
  then, R = {(0,0),(1,1)}

Function

- A function $f: A \rightarrow B$ from A to B is a relation, which has the property that every element $a \in A$ is the first element of exactly one ordered pair (a, b) of the relation
- So, $f: A \rightarrow B$ is a mapping assigning to each element $a \in A$ a unique element $b = f(a)$ in B, called image of a

Examples of functions

- Examples:
  - R_1 = {(0,0),(1,1),(2,2)} is a function
  - R_2 = {(0,0),(0,1),(2,2)} is not function



Some terminology

- A is called the domain of f and B is called the co-domain of f
- The range of f is the set of all images of elements of A (may not be the same as co-domain)
- A function f: A $\rightarrow$ B can be specified by using a rule a $\mapsto$ f(a), assigning to each element a $\in$ A, its image f(a) in B

Example of specifying a function

- Example:
  - R_1 = {(0,0),(1,1),(2,2)} is a function from A = {0,1,2} to B = {0,1,2}, which can be specified as a $\mapsto$ a

Binary operation

- A binary operation • on A is any function of type A x A → A
- So, a binary operation assigns to each ordered pair of elements (a,b) ∈ A x A a uniquely defined third element c = a • b in the same set A
- Example: 2-variable AND and OR are binary operations

Functons used in this course

- Boolean functions f: $B^n \rightarrow B$ on a set $B=\{0,1\}$, where B^n denotes the Cartesian product B x B x … x B
- incompletely specified Boolean functions f: $B^n \rightarrow B \cup \{-\}$, where "-" denotes a don't-care value
- muliple-output Boolean functions f: $B^n \rightarrow B^m$, f: $B^n \rightarrow (B \cup \{-\})^k$

Functons used in this course

- Multiple-valued functions f: $M^n \rightarrow M$ on a set $M = \{0,1,\ldots,m-1\}$
- Multiple-valued input two-valued output functions f: $M^n \rightarrow B$

Some terminology

- We say that $f(x_1, \ldots, x_n)$ is an n-variable function
- Functions $f: M^n \rightarrow M$ are called homogeneous, as opposed to heterogeneous functions, where the variables x_i do not take values in the same set
- There are $m^{(m^n)}$ homogeneous n-variable m-valued functions

VETENSKAP OCH KONST

KUNGL TEKNISKA HÖGSKOLAN

International Master Program in System-on-Chip Design

Boolean Algebra

Reading material

- de Micheli pp. 67 - 68, 288 - 294
- Muzio & Wesselkamper "Multiple-valued switching theory", p. 25 - 28

Boolean Algebra

- Let B be a set , "+" and "·" be binary operations, " ′ " be unary operation
- $\mathcal{B} = \langle B; +, \cdot, {}'; \mathbf{0}, \mathbf{1} \rangle$ is a Boolean algebra is the following set of axioms holds for "+", "·", " ′ " and some distinct elements **0** and **1** of B

Axioms of Boolean algebra

A1: $a,b \in B \;\Rightarrow\; a+b,\ a \cdot b,\ a' \in B$

A2: $\forall a,b \in B,\ a \cdot b = b \cdot a,\ a + b = b + a$

A3: $\forall a,b,c \in B,\ a \cdot (b+c) = a \cdot b + a \cdot c,\ a + b \cdot c = (a+b) \cdot (a+c)$

A4: $\forall a \in B,\ a \cdot \mathbf{1} = a,\ a + \mathbf{0} = a$

A5: $\forall a \in B,\ a \cdot a' = \mathbf{0},\ a + a' = \mathbf{1}$

A6: $\mathbf{0} \neq \mathbf{1}$

0 is called the zero and 1 the unit of B

Properties

- The following properties follow from the axiom set:

P1: $\forall a \in B,\ (a')' = a$

P2: $\forall a,b,c \in B,\ a \cdot (b \cdot c) = (a \cdot b) \cdot c,\ (a + b) + c = a + (b + c)$

P3: $\forall a \in B,\ a \cdot \mathbf{0} = \mathbf{0},\ a + \mathbf{1} = \mathbf{1}$

P4 (De Morgan's laws): $\forall a,b \in B,\ (a+b)' = a' \cdot b',\ (a \cdot b)' = a' + b'$

P5: $\forall a,b \in B,\ a \cdot (a+b) = a,\ a + a \cdot b = a$

P6: $\forall a \in B,\ a \cdot a = a,\ a + a = a$

P7: $\mathbf{0}' = \mathbf{1},\ \mathbf{1}' = \mathbf{0}$

Example 1 of a Boolean algebra

If B = {0,1}

"+" = OR

"·" = AND

" ′ " = NOT

0 = 0 and **1** = 1

then all the axioms are satisfied and ⟨{0,1}; +, ·, ′; 0,1⟩ is a Boolean algebra

Example 2 of a Boolean algebra

If P(S) is the set of all subsets of some non-empty set S

"+" = union ∪

"·" = intersection ∩

" ′ " = complement ¬

0 = ∅ and **1** = S

then all the axioms are satisfied and ⟨P(E); ∪,∩,¬, ∅,S⟩ is a Boolean algebra

Functionally complete sets

- A set of functions is called functionally complete if any other function can be composed from the functions in this set
- {AND, OR, NOT} is functionally complete for Boolean functions $f: \{0,1\}^n \rightarrow \{0,1\}$

Examples of functionally complete sets for $f: \{0,1\}^n \rightarrow \{0,1\}$

- {AND, NOT} is functionally complete
  - follows from de Morgan's law; every "+" can be replaced using "·" and "′" as $a+b = (a' \cdot b')'$
- {OR, NOT} is functionally complete
  - follows from de Morgan's law
- {AND, XOR, 1} is functionally complete