BITS Pilani
Pilani Campus

CS/IS F214
Logic in Computer Science

MODULE: INTRODUCTION

Problems in Logic and Their Complexity

Boolean expressions

- Consider Boolean operations AND, OR, and NOT defined on the set {1,0}:
  - If x and y are both 1,
    - then x AND y is 1; otherwise x AND y is 0
  - If x is 1 , or y is 1, or both of them are 1,
    - then x OR y is 1; otherwise x OR y is 0
  - If x is 1, then NOT x is 0; and if x is 0, then NOT x is 1

- Consider Boolean expressions (or formulas) of the form:
    - e.g. (x AND y) OR ((NOT x) AND z AND w)
  - How do you evaluate such expressions, given values for the variables?

Evaluating Boolean Expressions

- Can you write an algorithm (or a program) for evaluation of Boolean expressions?

  i.e. write an algorithm that
  - takes as input:
    - a Boolean expression ***e***
      - containing variables, say, ***x0***, ***x1***, …
    - and a map ***bm***
      - assigning Boolean values – i.e. 0 or 1 – to each of these variables
  - and evaluate ***e***

Evaluating Boolean Expressions – Time Taken

- What is the time taken by your algorithm to evaluate a Boolean expression?
  - Can you do it faster?
  - Can you argue that it cannot be done faster?
    - i.e. what is the minimum number of operations (or steps) required to evaluate an expression?

Boolean expressions - Satisfiability

- A Boolean expression (or formula) is said to be ***satisfiable***
  - if there is an assignment for which the expression evaluates to 1
    - i.e. if there exists an assignment of values to the variables – *occurring in the expression* – that makes the value of the expression 1.

- For instance, consider:
  - (x AND y) OR ((NOT x) AND z AND y)
    - Is this **satisfiable**? Why or why not?
  - ((NOT x) OR y) AND ((NOT y) AND z AND x)
    - Is this **satisfiable**? Why or why not?

Satisfiability is in NP

- The problem of **satisfiability** of Boolean expressions (*referred to as* **SAT**) is in **NP**:
  - i.e. given an input expression, and an assignment to the variables in the expression,
    - it can be *verified in time that is polynomial* in the length of the expression
      - whether the value of the expression is 1.

Exercise:

- Write a polynomial-time algorithm (or a program)
  - that takes as inputs
    - a Boolean expression and
    - an assignment (i.e. of values to variables)
  - and verifies whether the expression evaluates to 1.

Is SAT in P?

- Is **SAT** in **P**?
  - i.e. is there an algorithm
    - that takes an input Boolean expression, and
    - computes a suitable assignment to make the value of the expression 1, and
    - in time polynomial in the length of the expression?
- There is no known polynomial time algorithm for **SAT** so far:
  - But *no one has proved* that **SAT** is not in **P** either!

- Question:
  - What is the simplest algorithm for **SAT**?

Algorithm for SAT

- Write an algorithm to construct and evaluate a ***truth table***.
  - What is the time complexity of your algorithm?
  - How much space does it cost?
    - Can you eliminate / reduce this cost?
      - i.e. Do you need to store the **truth table**?

Logical Implication

- Logical Implication:
  - Consider the statements:
    - If it rains today, the road will be wet.
    - If I have a billion bucks, then I will stop working
    - If moon is made of cheese, and mars is made of ice then I will give you a billion bucks or I will eat cheese ice cream

Logical Implication

- These statements use “logical” implication:
  - i.e. they are of the form **A *implies* B** (denoted **A --> B**)
- Thus, we can formulate each statement using notation:
  - If it rains today then the road will be wet.
    - **rains_today --> road_wet**
  - If I have a billion bucks then I will stop working
    - **have_billion --> stop_working**
  - If moon is made of cheese and mars is made of ice then I will give you a billion bucks or I will eat cheese ice cream
    - **(cheesy_moon AND icy_mars) --> (give_billion OR eat_cheese_ice-cream)**

Understanding Implication

- A logical implication of the form:
  - **A --> B**

  is TRUE (i.e. evaluates to 1) if **B** is TRUE when(ever) **A** is TRUE.

- Write the truth table for **A --> B**

- Thus **A --> B** can be equated to **(NOT A) OR B**
  - Why?

Horn Clauses

- More generally,
  - **A1 AND A2 AND ... Ak --> B**
- can be translated to
  - **(NOT A1) OR (NOT A2) OR ... (NOT Ak) OR B**
- Such (implication clauses) are referred to as **Horn Clauses**.

HORN-SAT

- Horn Clauses form a subset of Boolean expressions.
- But it turns out that
    - satisfiability of Horn Clauses *can be computed in polynomial time.*
    - i.e. **HORN-SAT** is in **P**.