**Chapter 13: MISCELLANEOUS**

**Topic – 1: Computational Complexity Theory (CCT)**

**Introduction**

* **CCT** is used for studying about **computational resources**.
* Like **time** or **space complexity**.

**P v/s NP Problem**

* It analyses if a problem that can be **solved** in ***polynomial time***, can also be **verified** in ***polynomial time***.
* Class **P** contains problems which all could be **solved in polynomial time**.
* Class **NP** contains problems which all could be verified in **polynomial time**.
* **Solving** in **polynomial time** means **finding solution efficiently**.
* **Verifying** in **polynomial time** means **checking solution efficiently**.

**NP-Complete Problems**

* **NP-complete** problems are **subset** of **NP** problems.
* **NP-complete** problems are considered to be the **hardest problems** in **NP** class.
* Hardest in the sense that they can be **reduced** to **any** other **NP-complete** problem, in **polynomial time**.
* Or in layman’s language, its time complexity **grows exponentially** as the input size **increases**.
* **Completeness** of **NP** is very important in cryptography, optimization or in general for computer science & mathematics.

**Other CCT Concepts**

* **Complexity** of one problem can be compared to another if **polynomial time-reduction** is applied on them.
* There are other **complexity classes** beyond **P** and **NP** like **PSPACE** & **EXPTIME**.
* So, in **CCT** we mainly study about **computational limits** in terms of algorithms.

**Solvable v/s Unsolvable Problems**

* **Solvable problem:** A problem for which there exists an algorithm to solve it in **finite** amount of time.
* **Unsolvable problem:** A problem for which there exists **no** algorithm to solve it in **finite** amount of time.
* For example, ***halting problem*** which checks if a program will halt or run forever.

**Decidable v/s Undecidable Problems**

* **Decidable problem:** A problem whose output is either **Yes** or **No**, determined in **finite** amount of time.
* **Undecidable problem:** A problem whose output can’t be determined as **Yes** or **No** in **finite** amount of time.
* Again, its best example would be the **halting problem**.
* All **unsolvable** problems are **undecidable**, but **not** all **undecidable** problems are **unsolvable**.
* It is useful in understanding limits in **automated decision-making** systems.

**Open Computer Science Problems**

* **P v/s NP problems –** Asks if every problem that can be verified in polynomial time can also be solved in polynomial time.
* **Complexity of some problems –** Travelling salesman, graph isomorphism & factoring large integers etc.
* **Circuit lower bounds –** There is yet no known method to know minimum circuit size to solve a certain problem.
* **De-randomization –** Research on whether or not all randomized algorithm be converted into an equally efficient deterministic algorithm.
* **Quantum complexity theory –** Complexity theory that will be applied in quantum computers.

**Topic – 2: Automata & Game Theory**

**Introduction**

* It’s a field of study where actions of **all players** determine the **outcome**.
* Related to mathematics & economics.
* **Games:** Formal models to simulate game theory.

**Terminologies**

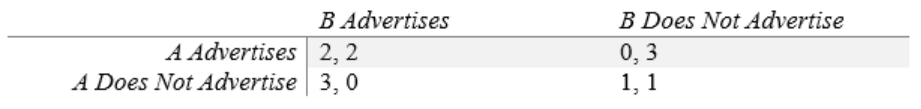
* **Player:** Chooses whatever is in their favour.
* **Strategies:** Course of action chosen by player.
* **Payoffs:** Result of player’s **strategy**, measured in terms of their **satisfaction**.
* **Nash equilibrium:** **Counter-strategy** taken by other players for one’s strategy.
* **Dominant strategy:** Best choice made by a player.
* **Mixed strategy:** Decision having **no fixed** outcome, relying purely of **probability**.

**Type Of Games**

* **Simultaneous games:** Players choose their strategies concurrently, **without** being aware about other player’s strategy.
* For example, prisoner’s dilemma & battle of sexes.
* **Sequential games:** Players choose after others being **aware** about their choices.
* For example, chess & poker.
* **Zero-sum games:** Only **winning player** gains profit from everyone’s payoffs.
* For example, gambling games.
* **Non-zero-sum games:** Players cooperate each other to achieve a **shared goal**.
* For example, prisoner’s dilemma & ultimatum game.
* **Cooperative games:** This type of game also has **legal agreement** among participating teams.
* For example, sports in general.
* **Non-cooperative games:** Has **no** cooperation or agreement among players.
* For example, bidding at auction.

**Nash Equilibrium Example**

* Let’s say there are two companies **A** & **B**.
* The following matrix shows when how much profit is gained by both.



**About Some Games**

* **Chicken game:** Same as **prisoner’s dilemma** but there are two players **driving cars** & turning away gives them profit.
* **Matching pennies:** Same as **prisoner’s dilemma** but with **coin toss**, however there is **no** profit sharing possibility.
* **Ultimatum:** Same as **prisoner’s dilemma** but it is about an **amount of money**.

**Formal Language & Automata Theory**

* **Infinite automata:** Type of automata which can recognize **context-free** & **recursive languages**.
* Game theorists use **finite automata** for **simulation**.
* They are used in sequential games, repeated games & evolutionary games.

**Use Of Pushdown Automata**

* Crucial when making **modelling games** using **CFG**.
* Good for **simulating games** like **chess**, **ludo** etc where actions might be required to be stored **temporarily**.

**Use Of Turing Machine**

* Anything related to algorithm, is directly related to **Turing machine** too.

**Use Of Automata Learning**

**Note!**

**🡪 Automata learning is a field in artificial intelligence that focuses on constructing automata or FSMs using data.**

* **Economics –** Simulating market behaviour, pricing tactics & effect of policies etc.
* **Political science –** Simulating how decisions of government can affect various aspects of nations & elections.
* **Artificial intelligence –** NLPs, image recognition & speech recognition etc.

**Topic – 3: Recursive Descent Parser (RDP)**

**Introduction**

* **RDP** is also known as ***top-down parser***.
* As the name says, it analyses input using **recursive methods**.
* The parser starts from **topmost rule** of grammar & parses **each symbol** as per those rules.
* The output is stored in form of **parse tree**.
* Parsing starts from **terminals**, then proceeds to **non-terminals**.
* Grammar rules are basically used to **check** if syntax is in **right form or not**.

**Grammar Example**

**expression ::= term ('+' term | '-' term )\***

**term ::= factor ('\*' factor | '/' factor )\***

**factor ::= '(' expression ')' | number**

**number ::= [0-9]+**

**Parsing**

* Note that parsing is applied on **non-terminals**.
* These are parsed as per rules they **satisfy**.
* One benefit of **RDP** is that it can **detect problems** in syntax **very efficiently**.
* However, if any **error** is found then it is **backtracked**.
* This creates problem if there are **many possibilities** of picking a rule.

**RDP Summary**

* Basically, syntax is checked **through and through** using **RDP**.
* Syntax is divided into **many parts**, we recursively go inside to **check syntax** from **ground zero**.

**printf("My compiler");**

**printf [Recursion level-0]**

**( [Recursion level-1]**

**" [Recursion level-2]**

**My compiler [Recursion level-3]**

**" [Back to level-2]**

**) [Back to level-1]**

**; [Recursion level-0]**

* Like in above given syntax, we will see **first production rule** which checks if the **function name** is **right or wrong**.
* Then if its **right**, **recursion** happens & we proceed to **second production rule** which checks if **parenthesis** is in **right way or not**.
* After if its right, we check the **inverted commas** & after that we see the **semicolon** etc.
* All of these are expressed in various **production rules**.
* **Recursion** is applied to check the rules.

**Compiler Specific Information**

* **Token:** Various kinds of values entered.
* **Lexer:** Tokenizes & analyses the syntax.
* **Parse:** Determines the result.