Lecture summary for Theory of Computation

Sandeep Sen[1]

January 8, 2015

[1]Department of Computer Science and Engineering, IIT Delhi, New Delhi 110016, India. E-mail:ssen@cse.iitd.ernet.in

Contents

Chapter 1

The notion of computation

What are computable functions ?

Starting from inputs, and using a finite number of well-defined, unambiguous and precise rules we can transform it to the output. Sometimes, we also impose the additional constraint of the individual transformation being *efficient* and the notion of successive transformations is known as an *algorithm*.

It has been elusive and indeed tricky to define computable functions without referring to a specific computing device or computational model.

On the other hand we can define precisely mathematically classes of functions that are computable - the best known is the class of *recursive* [1] functions.

Over the years, we have come to accept *Church-Turing* thesis (it was Kleene who combined the two seemingly different models) that equates computable functions with recursive functions.

There are indeed functions that are non-recursive and therefore non-computable as a consequence of CT thesis. The existence of such functions is proved by clever diagonalisation construction where every recursive function is mapped to an integer and the set of (integral) functions is known to much larger by an argument similar to Russel's paradox.

These results were known much before commercial computers were built and were motivated by Hilbert's quest for automated proofs (i.e. generate proofs using algorithms) and solutions of hard computational problems like *nullenstellansatz* (simultaneous polynomial equations). Göedel settled this problem of automated proofs in the negative by proving his famous *incompleteness theorem* ruling out any such grand pursuits. Later, Matiyasevich showed that even the problem of Diophatine equations is not solvable and it may be surprising that degree 4 and 9 variables suffice for this proof. Between 1931-39, following the works of Gödel, Church, Rosser and Turing, the theory of computability was formally recognized.

Central to establishing equivalence of powers of computational model is the notion of reductions and simulations. These form the core proof techniques in this area. Reductions

[1] not to be confused with recurrence relations or recursive descriptions of functions that we are used to in algorithms

between problems establish a partial ordering of the difficulty (or complexity) of solving problems. Likewise, simulating a model by another also establish relative computational powers of the models.

Chapter 2

Countable sets and cardinality based arguments

Let A, B be infinite sets, then the following are equivalent

- There is a function $f : A \Rightarrow B$ such that f is 1-1.
- There is a function $g : B \Rightarrow A$ such that g is onto.

When the above (equivalent) properties are satisfied then, we define a relation $\#(A) \leq \#(B)$ which intuitively means that the number of elements in A doesn't exceed the number of elements in B.

For example if $A \subset B$ then using f as the identity function $\#(A) \leq \#(B)$. So $\#(\mathbb{Z}) \leq \#(\mathbb{R})$ where $\mathbb{Z}, \mathbb{R}$ represent integers and reals respectively. If the function f is 1-1 and onto, i.e., bijective, then $\#(A) = \#(B)$. Intuitive we can pair up the elements of A, B, so that we can claim that they have the same cardinality.

Further, if $\#(A) \leq \#(B)$ and $\#(A) \neq \#(B)$, then $\#(A) < \#(B)$, i.e., they have different cardinalities.

Theorem 2.1 (Bernstein-Schroeder) *If $\#(A) \leq \#(B)$ and $\#(B) \leq \#(A)$, then $\#(A) = \#(B)$.*

The proof is non-trivial and omitted here.

Exercise 2.1 *Show that $\#(Q) = \#(Z)$ where Q is the set of rationals.*

Exercise 2.2 *If $\#(S_1) \leq \#(\mathbb{Z})$ and $\#(S_2) \leq \#(\mathbb{Z})$ for infinite sets S_1 and S_2 then $\#(S_1) = \#(S_2)$.*

Theorem 2.2 (Russel's paradox) *Let S be any set and let 2^S denote the powerset of S. Then $\#(S) < \#(2^S)$.*

Proof: Since the trivial function $f(x) = \{x\}, x \in S$ is 1-1, $\#(S) \leq \#(2^S)$. We will show that there is no 1-1 function $g : 2^S \Rightarrow S$ by contradiction. Suppose there exists such a g. Then define a subset $T = \{x \in S | x \notin g^{-1}(x)\}$ and $x' = g(T)$. We have now two possibilities

Case 1: $x' \in T$: Then from the earlier definition of T, $x' \notin g^{-1}(x') = T$.

Case 2 $x' \notin T$: Then from the earlier definition $x' \in g^{-1}(x') = T$.

Since both cases are not consistent, our assumption of the existence of g must be flawed. $\square$

Chapter 3

Language, strings, recognisers

Given a finite alphabet Σ, a *string* over Σ is finite ordered sequence of symbols from Σ. A string can be thought of as an element of

$$\Sigma^+ = \Sigma \cup \Sigma \times \Sigma \cup \Sigma \times \Sigma \times \Sigma \cup \ldots \text{infinite union}$$

The number of symbols in a string w is called the length and will be denoted by $|w|$. A special string of length 0 is denoted by ϵ and the set $\Sigma^* = \Sigma^+ \cup \epsilon$.

A language L is a subset of strings from Σ^*. Note that the possible number of languages is the power set of Σ^* that will be denoted by 2^{Σ^*}.

Note that both Σ^* and 2^{Σ^*} are infinite sets but the latter has a larger cardinality (from Russel's paradox). It can be seen that Σ^* is equinumerous with integers by first enumerating the strings in order of their lengths and using lexicographic ordering with the same lengths. Any program (say written in C) can be shown to be an element of Σ^* for an appropriately defined Σ. Any language like L can be also thought of as a function

$$\chi_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{otherwise} \end{cases}$$

which is called a characteristic function of L. Clearly there is a mismatch in cardinality between the possible number of functions and the number of possible C programs. So, there are languages for which we cannot write C programs ! Even for the ones that we can, we try to classify them according to the *difficulty* of writing programmes. In a nutshell this classification is the central problem in Theory of Computation. We will classify languages (functions) in terms of the complexity of the computational models required to recognise them. We will describe a hierarchy of machine models $\mathcal{M}_1, \mathcal{M}_2 \ldots$, such that $\mathcal{M}_i$ is strictly weaker than $\mathcal{M}_{i+1}$ in terms of capability. We associate a language with the appropriate class of machine required to recognise it.

3.1 Machine and States

Imagine a simple computing machine which has just just some minimal number of instructions (actually two suffices) required to write general programmes. It executes a finite length program and uses some space S to store intermediate values during computation. It has an *infinite* array of memory locations each of which can hold a fixed length integer. A program counter keeps track of the instruction being executed.Initially the memory locations $m_1, m_2 \ldots m_n$ store the input of length n and after the termination of the program, the memory location m_0 contains the answer (this function may look somewhat restrictive but it doesn't affect our subsequent arguments). The working space S is mapped stating from m_{n+1}, i.e., the input is not modified. Each step of the computation executes exactly one instruction of the program that could modify some memory location. At any instance, a *snapshot* of the computation can be captured by the contents of some finite number of memory locations[1], and the program counter. A computation is a finite sequence of snapshots where successive snapshots follow from the previous one in accordance with the instruction being executed.

What happens if we put a bound on S, i.e., it is not allowed to scale with the size of the input n ?

Since the program doesn't scale with input either, the possible number of snapshots can be bounded by the possible memory contents of the space S which is also bounded if each location holds a fixed amount of data[2]. Imagine that you enumerate the snapshots in a sequence of computation steps. **Can any snapshot be repeated ?**

Given that the input is not modifiable, we only have a bounded number of snapshots and therefore rather severely constrained. The class of languages that can be recognised using bounded space is called *Finite State Machines* and is one of the most important class of Languages in Computer Science applications. Readers may be warned that *bounded* does not mean some apriori bound but that the space is not a function of the input-size.

3.2 Crossing Sequence based arguments

If the machine is not allowed to modify its tape contents then it is severely restricted in terms of its computational power. This actually helps to rigorously prove limitations in computational power. One common technique is the use of *crossing sequences* that gives a simple way of bounding the number of possible configurations of the machine by some finite number (which can be rather large).

The main observation about the computation is that the *information* carried across the boundary of two cells, say i and $i+1$ can be parameterized by the state. A crossing sequence is a tuple of states $C_i = (l_1, r_1, l_2, r_2 \ldots)$ where $l_j, r_j \in Q$ and l_j (resp. r_j) denotes the state when the head is moving from left to right (and vice versa) over the entire period

[1]Since the computation is a finite process, the number of memory locations must be finite, i.e., S is finite.

[2]This has a technical subtlety as it implies that we cannot even count till n as $\log n$ bits are required for that

of computation. Since there is no change in the tape-cell contents, the crossing sequence characterises the computation that happens to the left and right of this boundary - in fact it is kind of a *fingerprint*. Notice that if two boundaries have identical crossing sequences, say $C_i = C_j$, then the computation will be *identical* - if one were to substitute the first i cells with the first j cells or the block of cells to the right of i with the block of cells to the right of j.

The total number of distinct crossing sequences is finite - Why ? Note that for a terminating computation, none of the l_i's (or r_is) can repeat since that will lead to infinite loops.

The other important observation about crossing sequences is that crossing sequences in adjacent cells should be *compatible* or *matching*. Since the compatibility check can be done locally on the basis of the state transition, one can construct a transition sequence from an initial state for a given input string. If any of the transition sequences leads to acceptance, then the string is accepted.

Chapter 4

Finite Automaton and Non-Determinism

A (deterministic) Finite State Machine M is defined by a 5 tuple

Σ	Finite alphabet
Q	Finite set of states
$\delta : Q \times \Sigma \to Q$	transition function
$q_o \in Q$	A unique initial state
$F \subset Q$	Set of final states

The extension of δ for string is defined inductively as $\delta(q, wa) = \delta(\delta(q, w), a)$ where $w \in \Sigma^*, a \in \Sigma$. A string $w \in \Sigma^*$ is accepted by M, denoted by $w \in L(M)$ iff $\delta(q_o, w) \in F$.

Many useful languages can be recognised by FA, like strings containing one or key words, strings corresponding to valid decimal numbers etc.

An NFA is similar to DFA except that the transition function is more general as $\delta : Q \times \Sigma \to 2^Q$, i.e., it is mapped to a (non-empty) subset of Q. This can be thought of as

> (i) The machine arbitrarily chooses one of the possible states[1]. (ii) The machine makes as many copies as the number of successive states and each of the copies simultaneously follows a unique successor state and continues to process the string. This "spawning" could again happen for any non-deterministic transitions.

The transition function can be extended to strings as follows:

$$\delta(q, wa) = \cup_{r \in \delta(q,w)} \delta(r, a)$$

Therefore, unlike a DFA, an NFA can be simultaneously in a *cloud* of possible states.

The recognition condition of a string w by a machine N is defined as follows:

$w \in L(N)$ iff $\delta(q_o, w) \cap F \neq \phi$, i.e., one of the possible states at the end of the transition is a final state. In other words, as long as one of the possible transition sequence leads to a

[1] not associated with any probability

final state, the string is accepted and not accepted if **all** paths end in non-final states. This makes the NFA somewhat asymmetric in its behavior.

Exercise 4.1 *What is the set of strings accepted by flipping the Final and non-final states of DFA and NFA ?*

4.1 Relation between NFA and DFA

There are certain advantages of designing with an NFA in terms of conceptualisation. For example, think about all strings that must have a "1" at the k-th position from the end. A DFA needs to remember k latest inputs (and hence have about 2^k states) whereas an NFA can simply *guess* on a "1" and verify (or fail) if it is not the k-th "1" from right. This NFA will have about $O(k)$ states, i.e., it exponentially better. However, the physical interpretations of NFA are not practical (either guessing or the spawning paradigm), so to actually build a machine, we have to use the following mechanism.

Theorem 4.1 *For every NFA N, we can construct a DFA M, such that $L(M) = L(N)$.*

The catch in this construction is the *blow-up* in the number of states that is unavoidable in many cases (from the previous example). In fact, the state space of the equivalent DFA is the power set 2^Q where Q is the set of states of the NFA N.

The construction is such that the equivalent DFA simulates the NFA on any given input string. Since the NFA can be a subset of states, say $S \subset Q$, the DFA is in state labelled with $\{S\}$ and the transition of the DFA is defined as $\delta' : 2^Q \to 2^Q$. If $\{S_1\}$ is a state of the DFA then $\delta'(\{S_1\}, a) = \cup_{s \in S_1} \delta(s, a)$[2]

The equivalence of the constructed DFA and the given NFA is based on the following claim

Claim 4.1 *For any state $q \in Q$, $\delta(q, w) = \delta'(\{q\}, w\})$ for all $w \in \Sigma^*$.*

Proof follows by induction on $|w|$.

A similar result can be proved for a slightly version of NFA, namely NFA with ϵ transitions. In this NFA, the transitions are also defined for ϵ, like any regular symbol. An ϵ-closure of a state q is defined as the set of states reachable from q using only ϵ transitions, call it $\epsilon(q)$. This definition also a natural extension to $\epsilon(S)$ where $S \subset Q$. You can argue that given an NFA N with ϵ transitions, we can construct an equivalent DFA M without ϵ transitions by using a construction similar as described above and using the notion ϵ-closure of states. For example, the initial state of M is the ϵ-closure of q_0, the start state of N.

Exercise 4.2 *Fill in the details and argue the equivalence formally.*

[2]There is some abuse of notation since strictly speaking we need to curl and uncurl the set of states.

Chapter 5

Regular Expressions, and properties of regular languages

A regular expression r over Σ is represents a set of strings (possibly infinite) denoted by $L(r)$ is defined as follows

(i) ϵ, ϕ, a are valid r.e. where $a \in \Sigma$.
(ii) If R_1, R_2 are valid r.e., then so are

(a) $R_1 + R_2$ representing $L(R_1) \cup L(R_2)$.
(b) $R_1 \cdot R_2$ representing concatenation.
(c) R_1^* representing $\epsilon \cup L(R_1) \cup L(R_1^2) \cup L(R_2^2) \ldots$.
(d) (R_1) is regular (to limit the scope for use of other operators).

The class of languages that can be represented using (finite length) regular expressions is called *Regular Languages*.

Theorem 5.1 *A language L is regular iff there is a DFA accepting exactly the strings in L.*

The proof is based on showing that for every regular expression r, there is an NFA N such that $L(M) = L(r)$. The proof uses an NFA with ϵ transitions. Similarly, it can be shown using a dynamic programming based construction that the language accepted by a DFA M can be represented using a regular expression.

Note that the membership problem for r.e., namely given a string w and a r.e. r, verify if $w \in L(w)$ can be efficiently solved by reducing it to an acceptance test for the equivalent DFA.

5.1 Properties of Regular languages

The regular languages are closed under

1. **Union**
2. **Complementation**
3. **Intersection** (follows from the previous two)
4. **Kleene closure**
5. **Substitution** : Each symbol a of the alphabet is replaced by L_a which is a regular expression.
6. **Homomorphism and Inverse homomorphism**: It is a special kind of substitution where each symbol is substituted by exactly one string (possibly over a different alphabet).
   The resulting language $L' = \{w' \in \Sigma^* | \text{ for some } w \in \Sigma^*, w' = F(w)\}$ where F is the substitution function.

   Exercise 5.1 *Prove formally that the r.e. obtained by substituting each $a \in \Sigma$ by a string in the r.e. for L is exactly the same language as defined above.*

5.2 How to prove a language is not regular

Given a regular language L, consider any DFA M that accepts L. Suppose the number of states of this DFA is n[1]. If we take a string $z, |z| \geq n$, we can trace the state transitions starting from the initial state, say, $q_0, q_{i_1}, q_{i_2}, \ldots q_{i_n}$ where $q_{i_k} = \delta(q_{i_{k-1}})$. In this sequence of $n+1$ states, there must be at least one repeated state. Consider the first such repeated state, say $q_{i_j} = q_{i_l}$, $j < l$, $\delta(q_{i_j}, v) = q_{i_l}$, where v is a substring of z of length ≥ 1. Suppose $z = u \cdot v \cdot w$, then we can claim that the strings $z_i = u \cdot v^i \cdot w \in L$ as z_i will end in the final state of M for $i \geq 0$. Moreover $|u| + |v| \leq n$.

In other words, we have found an infinite sequence of strings in L **if** L is regular given a string that is *sufficiently long*.

To prove a given L is not regular, say it is the set of strings with equal number of 0'sand 1's, we can use the following proof by contradiction. Suppose L is regular and consider a string 0^n1^n where n is related to the previous proof. From our previous argument, $0^n1^n = u \cdot v \cdot w$ where $v = 0^i$ $i \geq 1$ (recall that $|u \cdot v| \leq n$). Therefore $u \cdot w = 0^{n-i}1^n \in L$ but it does not contain equal number of 0's and 1's. Therefore our assumption that L is regular must be fallacious.

5.3 Myhill Nerode Theorem

For any language L, we define a relation R_L over all strings in Σ^* as follows

[1]Clearly the number of minimum state DFA for L has $\leq n$ states.

For all strings $x, y \in \Sigma^*$, xR_Ly iff for all $z \in \Sigma*$, either
(i) $x \cdot z$ and $y \cdot z$ both are in L or
(ii) $x \cdot z$ and $y \cdot z$ both are not in L

Exercise 5.2 *Prove that R_L is an equivalence relation.*

Theorem 5.2 (Myhill-Nerode) *R_L is finite iff L is regular.*

To prove this, we define a relation R_M for a DFA M that recognises the language L. Two strings $x, y \in \Sigma^*$ are related by R_M iff $\delta(q_0, x) = \delta(q_0, y)$. This is an equivalence relation and has the property that for all $z \in \Sigma^*$, xR_My implies $x \cdot zR_My \cdot z$. This property is called *right extension invariant.*
Claim If xR_My then xR_Ly.

Therefore the partitions induced by R_M is a refinement of the partitions of R_L and hence the number of equivalence classes of R_L is finite. This proves one direction of Myhill-Nerode theorem. For the other direction, let us assume that R_L has finite number of equivalence classes. We define a DFA where the set of states correspond to the equivalence classes of R_L.
Claim For any partition of R_L, either all strings are in L or all the strings are not in L.

Therefore we can define the final states of this machine. For the transition function, we let $[x]$ denote the partition of R_L containing the string x. We define $\delta([x], a) = [x \cdot a]$ for $a \in \Sigma$. Notice that if $[x] = [y]$, then we can prove $[x \cdot a] = y \cdot a]$ as follows. Let $z' = a \cdot z$ for $z \in \Sigma^*$, then if xR_Ly, then $x(az)R_Ly(az)$ or equivalently $x \cdot aR_Ly \cdot a$. By defining the initial state as $[\epsilon]$, any string $w \in L$ takes the machine to a final state $[w]$.

Here are a couple of applications of the MN theorem

- To prove that a language L is non-regular by showing that R_L has infinite partitions.For example for $L = \{0^i1^i | i \geq 1\}$, 0^i and 0^j, $i \neq j$ are indistinct classes and hence there are unbounded number of equivalence classes.

- There is a unique minimum state DFA (upto renaming of states).
  Clearly any DFA for L has at least as many states as the number of partitions of R_L and from previous proof there is a DFA for L defined from the partitions of R_L. The states of any minimum state DFA can be mapped to the partitions of R_L.

Chapter 6

CFL and PDA

A language is Context Free if it can be described using a Context Free Grammar that consists of the following attributes

- Σ : the symbols of the language also called *terminals*
- V : variables also called *non-terminals*
- S : a special variable called the *start symbol*
- P : A set of *production rules* of the following format

$$A \to \alpha$$

  where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

The language generated by the grammar G, denoted by $L(G)$ consists of strings $w \in \Sigma^*$ such that $S \to \alpha_1 \to \alpha_2 \dots w$. Instead we will use the notation $\overset{*}{\to}$ to denote 1 or more applications of the production rules.

The Context Free Language contains Regular Languages as proper subsets, i.e., there are languages like $0^1 1^i$ for which we can write a grammar. For any regular expression, we can write a Context Free Grammar

Any given grammar G can be rewritten such that all production rules are of the form

1. Chomsky Normal Form (CNF)

$$A \to BC|a$$

   where $A, B, C \in V$ and $a \in \Sigma$

2. Greibach Normal Form (GNF)

$$A \to aV^*$$

To accomplish the transformation,

- We eliminate *useless* symbols. These are variables that cannot derive any string or cannot be derived from S.
- Remove ϵ productions, i.e. $A \to \epsilon$.
- Remove unit productions, i.e. $A \to B$.
- Removing left-recursion, i.e., not have productions of the form

  $$A \to A\alpha, \alpha \in (V \cup \Sigma)^*$$

  The first variable on the right hand side cannot be the same as LHS
- Ordering of variables as $X_1, X_2 \ldots$ such that if

  $$X_i \to X_j(V \cup \Sigma)^*$$

  then $j > i$. For the highest numbered variable, all productions must begin with a terminal symbol on the RHS.

The following simple observation is very useful to accomplish the above clean-up steps.

Claim 6.1 *Let $A \to \alpha_1 B \alpha_2, \alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ and let $B \to \beta_1, \beta_2 \ldots \beta_k$ be all productions of B. Then we can replace the production $A \to \alpha_1 B \alpha_2$ with*

$$A \to \alpha_1\beta_1\alpha_2 | \ldots \alpha_1\beta_k\alpha_2$$

Exercise 6.1 *Prove this rigorously, i.e. show that the two grammars (original and the transformed) generate the same set of strings.*

Given a CFG G in CNF, we can design an efficient algorithm for the membership problem, i.e. Does $w \in L(G)$, based on dynamic programming. This algorithm is known as CYK in literature and runs in $O(n^3)$ steps where $n = |w|$. (Without CNF, designing an efficient algorithm can be very challenging).

6.1 Push Down Automaton (PDA)

Intuitively, if we add an unlimited sized stack with a Finite Automaton, then we have memory to recognise certain languages like 0^i1^i by accumulating some tokens corresponding to 0's and tallying with the number of 1's. A Push Down Automaton (PDA) is characterized by

- Σ : a finite input alphabet
- Γ : a finite stack alphabet containing a special bottom stack symbol Z_0.
- Q: a finite set of states, containing a special initial state q_0.

- $\delta : Q \times \{\Sigma \cup \epsilon\} \times \Gamma \rightarrow 2^Q \times \Gamma^*$, i.e. a mapping to a finite subset of $Q \times \Gamma^*$. Here Γ refers to the token on the stack top - it is either popped (results in ϵ) or some tokens are added which is a string of Γ^*. Since there are several possibilities, the machine is inherently non-deterministic. Also note that the machine can change the stack without scanning the input symbol (i.e. on ϵ).

At any instance, the snapshot (also called instantaneous description or ID) of a PDA, can be described in terms of a triple (q, x, β) where $q \in Q$ is the current state, $x \in \Sigma^*$ is prefix of the input that is scanned and $\beta \in \Gamma^*$ is the (entire) contents of the stack. We will use the notation $I_1 \vdash I_2$ to denote a legal move from one ID to the next according to the transition function of the PDA. There are two version of PDA

1. Accepts by final state: The machine M_f should be in a final state after the input is scanned.
   $w \in L(M_f)$ iff $(q_0, w, Z_0) \vdash^* (q_f, \epsilon, \alpha)$

2. Accepts by empty stack: The machine M_e should have no tokens in the stack and the end of the execution.
   $w \in L(M_f)$ iff $(q_0, w, Z_0) \vdash^* (q_i, \epsilon, \epsilon)$

Claim 6.2 *The two versions of the machines are equivalent in their capabilities, i.e., for any language L either both kinds of machines can recognise it or it can't be recognised by either.*

Exercise 6.2 *Establish this rigorously by designing a simulation of one machine by the other primarily with respect to the acceptance criteria.*

6.2 Equivalence of PDA and CFG

The class of languages that can be recognised by PDA is exactly the class of CFL. For this, we will prove the following claim.

Theorem 6.1 *1. Given any CFL L in GNF G , we will design a PDA M_e that accepts using empty stack to recognise L.*
2. Given a PDA M_e that accepts using empty stack, we will describe a CFG G such that $L(G) = L(M_e)$.

For the first construction, we will construct a one state PDA [1] that accepts the same language. The idea is to mimic the left most derivation of any sentence starting from a variable of the CFG. If $A \xrightarrow{*} x\alpha$ where $x \in \Sigma^*$ and $\alpha \in V^*$ then the machine M should be in a configuration where the stack contains α (left most variable is on the top of the stack) and should have scanned x from the input. By ensuring this, it follows as a corollary that $S \xrightarrow{*} w$ iff M accepts w using empty stack.

[1] Therefore all PDAs can be converted to a 1 state PDA

Constructing a grammar from a given PDA exploits the non-deterministic behaviour of the machine crucially. We want to describe a grammar that generates a string (starting from S) exactly when the machine accepts by empty stack. Like in the previous case, we want the grammar to generate a sentence $x\alpha$ by left-most derivation when the machine has scanned x and contains α in the stack. The situation is somewhat more complex since it is not a single state machine. Therefore the final state (as well as the intermediate states) are not known and we should allow for all possibilities. Eventually in the acceptance condition by empty stack it doesn't matter what the final state is as long as it is obtained using legal transitions of the machine.

The variables of the grammar will mimic the moves of the machine and are denoted by the triple $[p, A, q]$ $p, q \in Q$ $A \in \Gamma$. These triples capture the portion of the input string scanned by the PDA starting in state p with A as the top of the stack till the time that A is popped out. This can happen in one or more moves including the case that A is popped in one step or by a sequence of pushes and pops - the state of the PDA at that juncture is q. Since we do not know q, we cover for all possibilities of $q \in Q$. The invariant that we want to ensure is that $[p, A, q] \xrightarrow{*} x$ iff $(p, x, A) \vdash^* (q, \epsilon, \epsilon)$.

The productions are defined on the basis of single moves of the PDA. If the machine is in state q and the top of the stack is $A \in \Gamma$, and the symbol being scanned is $a \in \Sigma \cup \epsilon$, the machine can

- **Pop** : Then it must be the case that (p, a, A) contains (q, ϵ).
  This yields the production $[p, A, q] \rightarrow a$.

- **Push** A is replaced with $B_1, B_2 \ldots B_k$: Then $[p, A, q]$ must be a composition of multiple moves of the machine where $B_1, B_2, \ldots B_k$ must be eventually popped. For popping B_i after $B_1, B_2 \ldots B_{i-1}$ have been popped, some portion of the input must be scanned, say y_i, so that

  $$[p, A, q] \xrightarrow{*} w \text{ iff } \forall i \; [q_i, B_i, q_{i+1}] \xrightarrow{*} y_i \; , \; q_{k+1} = q$$

  where $w = a \cdot y_1 \cdot y_2 \ldots y_k$ and (p, a, A) contains $(q_1, B_1 B_2 \ldots B_k)$. Since we do not know the intermediate states after popping of B_1, B_2 etc., we allow for all possible choices $q_i \in Q$. This is verified by the single pop moves for which we know the state transitions and only those q_i can be reached by composition of the single pop moves.[2]

[2]We do not try to compute explicitly which q_i can be reached and let the leftmost derivation process simulate the PDA moves.

Chapter 7

An alternate view of TM as functions

The behavior of a TM can be interpreted in terms of computing a function in the following manner - the input is the initial contents of the tape and the final content (if it stops) is the end content of the tape. If the index of the TM is k then we denote the mathematical function corresponding to the TM as f_k and it is *defined* only for input x when the TM actually halts on x. f_k is called a *partial recursive function*. If the TM halts on all inputs, then the function is called a *total recursive function*. This notion can be extended to n-ary functions by encoding the input in some appropriate way. For example, a three input function can be encoded as $\underbrace{000\ldots00}_{x_1}1\underbrace{000\ldots00}_{x_2}1\underbrace{000\ldots00}_{x_3}$ for the input (x_1, x_2, x_3). Note that the 1s are used as separators.

In many constructions involving partial recursive functions, we may have to *apply* a function f_i followed by f_j to some input (similar to simulation of Turing machines using Universal TM). Even though f_i, f_j may not be total recursive, we can define a total recursive function *compose* such that

$$f_{compose(i,j)}(x) = f_j(f_i(x)) \forall x \tag{7.0.1}$$

This follows by defining $compose(i, j)$ as the code of a TM that runs TM represented by j on the output of running the TM i on x. This equality holds for those x for which $f_j(f_i(x))$ is defined. If both f_i, f_j are total recursive then the equality holds for all x. In any case $compose(i, j)$ is always defined.

The following result is known as the S_{mn} theorem.

Theorem 7.1 *If $g(x, y)$ is a partial recursive function, then there exists a total recursive function σ such that*

$$f_{\sigma(x)}(y) = g(x, y) \quad \forall x, y$$

Let M be a TM with index k compute $g(x, y)$. Then, we create a TM M_x that first prefixes the input (y) with x and then runs the TM for g. Thus it can be thought of as a composition

of two partial recursive functions and the previous equation can be used to complete the proof.

$$f_k(x,y) = g(\Pi_x(y)) = f_{compose(\Pi_x,k)}(y)$$

where $\Pi_x(y) = (x,y)$. Then $\sigma(x) = compose(\Pi_x, k)$ which is total recursive.

Theorem 7.2 (Recursion theorem) *Let σ be a total recursive function, then there exists an x_o such that $\forall x$, $f_{x_o}(x) = f_{\sigma(x_o)}(x)$.*

Note that this is trivial for $\sigma(x) = x$ but not so for other functions.

Let us define a function $h(i,x) = f_{f_i(i)}(x)$ such that the TM given i, x first applies (simulates) f_i (the function corresponding to the i-th TM) on i. If $f_i(i) = j$ (i.e. if it is defined) then run M_j on x. Note that f_i may not be total recursive. From the S_{mn} theorem, there exists a total recursive function $g(i)$ such that

$$f_{g(i)}(x) = h(i,x) = f_{f_i(i)}(x) \tag{7.0.2}$$

Let t be the code of a TM such that $f_t(x) = \sigma(g(x))$ - t can be thought of as $compose(g, \sigma)$ in Equation 7.0.1. Moreover, f_t is total recursive since it is the composition of total recursive functions σ and g. Let $x* = g(t)$. Then

$$\begin{aligned} f_{x*}(x) &= f_{g(t)}(x) \\ &= f_{f_t(t)}(x) \text{ from equation 7.0.2} \\ &= f_{\sigma(g(t))}(x) \text{ from definition of } f_t(x) \\ &= f_{\sigma(x*)}(x) \end{aligned}$$

A fun application of recursion theorem is that there exists a TM such that when it is started on an empty tape, it prints out its own code. Equivalently, there exists a C-language program [1] that prints itself.

Hint: Use a function $\sigma(i) = print(i)$ where $print(i)$ is the code of a TM that writes out i for any input (including blank tape).

[1] or any other programming language

Chapter 8

Introduction to Computational Complexity

8.1 Definitions of Time and Space Complexity

8.2 Relation between Complexity Classes

8.3 The Reachability Problem

The membership problem for a non-deterministic Turing Machine can be thought of as a graph reachability problem as follows. The vertices of the graph are the distinct IDs[1]. There is a directed edge between v_i and v_j iff there is a legal move (depending on the transition function of the Turing machine) from i-th ID to the j-th ID. Corresponding to a machine M, we will denote this graph by $G(M)$. For a given input w (and the corresponding initial ID I_0), the machine M accepts w iff there is a path in $G(M)$ from v_{I_0} to v_f where f is some final ID. In fact, the transitive closure of $G(M)$ can be used a table lookup for membership problem.

Note that $G(M)$ can be quite large even if finite and it is usually not available explicitly. However, given two vertices v_i, v_j, we can determine if there is an edge between them using the knowledge of the transition function.

Observation 8.1 *A non-deterministic TM can solve the $s-t$ reachability problem using $\log |V|$ space where V is the set of vertices. Here s and t are starting and terminating vertex.*

The algorithm for this is intuitive as the machine guesses a path $P = s = v_o, v_1, \ldots v_k = t$ and verifies that it is a legal path by checking that every consecutive vertices on the path is connected by an edge. The space is used to store the latest vertex in the path, i.e., the path is guessed one vertex at a time.

The less obvious result is the complement of the reachability problem.

[1] If the machine is known to be space or time bounded then the graph is finite

Observation 8.2 *A non-deterministic TM can solve the $s-t$ non-reachability problem using $\log|V|$ space where V is the set of vertices. Here s and t are starting and terminating vertex.*

The certificate of non-reachability is more subtle. For this we solve the restricted problem of non-reachability in i steps, i.e., the shortest path between s and t exceeds i. By repeatedly choosing $i = 1, 2 \ldots |V| - 1$, we can solve the non-reachability problem. The storage for the counter i itself takes $\log|V|$ space. We describe the procedure inductively - We try to determine if $t \notin R(s,i)$, where $v \in R(s,i)$ iff the shortest path from s to v is $\leq i$ edges. Using a NDTM, if we can guess $v \in R(s,i)$ and verify that $t \neq v$ **for all** $v \in R(s,i)$, then we are done. If we explicitly generate $R(s,i)$, this could easily exceed $O(\log|V|)$ space, so we will only keep a count of the set of vertices in $R(s,i)$ that takes $O(\log|V|)$ space.

We generate the sequence of vertices (using a counter) or in some canonical ordering and increment the counter for $R(s,i)$, if the vertex belongs to the set. We stop when we have generated all of them.

But how do we know if we have generated all of them ?

For this we will design a (non-deterministic) algorithm to generate $|R(s,1)|, |R(s,2)| \ldots$ inductively. Suppose we have $n_i = |R(s,i)|$. To generate n_{i+1}, we use the previous algorithm to generate a vertex v in some canonical ordering and verify if it is a neighbour of some vertex in $R(s,i)$[2] - if so, increment n_{i+1}. To check the neighbourhood relation, we must either have the graph available explicitly as input or we should be able to check it on the fly from the implicit representation.

Note that the space required to generate n_{i+1} is that of two counters, n_i, n_{i+1} plus two sequences of vertices generated (in nested loops) in some canonical ordering. Since each of them can be done using $O(\log|V|)$ space, the total space is logarithmic.

The above result is known as Immerman-Szelepcsenyi theorem

Remark Note that, we are avoiding direct computation of $V - R(s,i)$ as there is no simple certificate for non-reachability. Otherwise, we could check every vertex x as the predecessor of t and rejected x, if $x \notin E(s,i)$.

Theorem 8.1 *The reachability problem can be solved in determnistic space $O(\log^2|V|)$.*

This is also known as Savitch's theorem where a clever recursive procedure is used that reuses space for the different phases of the recursion. More specifically $s \rightsquigarrow t$ iff $s \rightsquigarrow x$ and $x \rightsquigarrow t$ for some $x \in V$. Since the two subproblems are symmetric, space can be reused and the best balance is obtained when x is a *middle* vertex in the s-t path. Namely, $s \xrightarrow{\ell} t$ iff $s \xrightarrow{\ell/2} x$ and $x \xrightarrow{\ell/2} t$ for some $x \in V$. Note that $\ell = |V| - 1$. The depth of recursion is at most $\log|V|$ and each entry in the recursion stack is about $O(\log|V|)$ giving us the required bound.

For undirected s-t connectivity problem, there is a simple algorithm based on random walks that takes $O(\log|V|)$ space and recently this was derandomized that showed that s-t connectivity problem is in $DSPACE(\log|V|)$ (due to Reingold).

[2]For this, the same procedure is used and we actually have n_i available from induction

Chapter 9

NP Completeness and Approximation Algorithms

Let $\mathcal{C}()$ be a class of problems defined by some property. We are interested in characterizing the *hardest* problems in the class, so that if we can find an efficient algorithm for these, it would imply fast algorithms for all the problems in $\mathcal{C}$. The class that is of great interest to computer scientists is the class $\mathcal{P}$ that is the set of problems for which we can design polynomial time algorithms. A related class is $\mathcal{NP}$, the class of problems for which non-deterministic[1] polynomial time algorithms can be designed.

More formally,

$$\mathcal{P} = \cup_{i \geq 1} \mathcal{C}(T^{\mathcal{P}}(n^i))$$

where $\mathcal{C}(T^{\mathcal{P}}(n^i))$ denotes problems for which $O(n^i)$ time algorithms can be designed.

$$\mathcal{NP} = \cup_{i \geq 1} \mathcal{C}(T^{\mathcal{NP}}(n^i))$$

where $T^{\mathcal{NP}}()$ represent non-deterministic time. Below we formalize the notion of *hardest* problems and what is known about the hardest problems. It may be noted that the theory developed in the context of $\mathcal{P}$ and $\mathcal{NP}$ is mostly confined to *decision* problems, i.e., those that have a Yes/No answer. So we can think about a problem P as a subset of integers as all inputs can be mapped to integers and hence we are solving the membership problem for a given set.

Exercise 9.1 *Prove the following*
(i) If $P \in \mathcal{P}$ then complement of P is also in $\mathcal{P}$.
(ii) If $P_1, P_2 \in \mathcal{P}$ then $P_1 \cup P_2 \in \mathcal{P}$ and $P_1 \cap P_2 \in \mathcal{P}$.

[1] We will define it more formally later. These algorithms have a choice of more than one possible transitions at any step that does not depend on any deterministic factor.

9.1 Classes and reducibility

The intuitive notion of *reducibility* between two problems is that if we can solve one we can also solve the other. Reducibility is actually an asymmetric relation and also entails some details about the cost of reduction. We will use the notation $P_1 \leq_R P_2$ to denote that problem P_1 is reducible to P_2 using resource (time or space as the case may be) to problem P_2. Note that it is not necessary that $P_2 \leq_R P_1$.

In the context of decision problems, a problem P_1 is *many-one* reducible to P_2 if there is a many-to-one function $g()$ that maps an instance $\mathcal{I}_1 \in P_1$ to an instance $\mathcal{I}_2 \in P_2$ such that the answer to $\mathcal{I}_2$ is *YES* iff the answer to $\mathcal{I}_1$ is *YES*.

In other words, the many-to-one reducibility maps YES instances to YES instances and NO instances to NO instances. Note that the mapping need not be 1-1 and therefore reducibility is not a symmetric relation.

Further, if the mapping function $g()$ can be computed in polynomial time then we say that P_1 is polynomial-time reducible to P_2 and is denoted by $P_1 \leq_{poly} P_2$.

The other important kind of reduction is *logspace* reduction and is denoted by

$$P_1 \leq_{\log} P_2.$$

Claim 9.1 *If $P_1 \leq_{\log} P_2$ then $P_1 \leq_{poly} P_2$.*

This follows from a more general result that any finite computational process that uses space S has a running time bounded by 2^S. A rigorous proof is based on the Turing Machine model with bounded number of states and tape alphabet.

Claim 9.2 *The relation $\leq_{poly}$ is transitive, i.e., if $P_1 \leq_{poly} P_2$ and $P_2 \leq_{poly} P_3$ then $P_1 \leq_{poly} P_3$.*

From the first assertion there must exist polynomial time computable reduction functions, say $g()$ and $g'()$ corresponding to the first and second reductions. So we can define a function $g'(g))$ which is a composition of the two functions and we claim that it satisfies the property of a polynomial time reduction function from P_1 to P_3. Let x be an input to P_1, then $g(x) \in P_2$ [2] iff $x \in P_1$. Similarly $g'(g(x)) \in P_3$ iff $g(x) \in P_2$ implying $g'(g(x)) \in P_3$ iff $x \in P_1$. Moreover the composition of two polynomials is a polynomial, so $g'(g(x))$ is polynomial time computable.

A similar result on transitivity also holds for log-space reduction, although the proof is more subtle since we cannot store the intermediate string and we have to generate this on demand.

[2] This is a short form of saying that $g(x)$ is an YES instance.

Claim 9.3 *If* $\Pi_1 \leq_{poly} \Pi_2$ *then*
(i) If there is a polynomial time algorithm for Π_2 *then there is a polynomial time algorithm for* Π_1*.*
(ii) If there is no polynomial time algorithm for Π_1*, then there cannot be a polynomial time algorithm for* Π_2*.*

Part (ii) is easily proved by contradiction. For part (i), if $p_2(n)$ is the running time of Π_2 and p_1 is the time of the reduction function, then there is an algorithm for P_1 that takes $p_2(p_1(n))$ steps where n is the input length for $\mathcal{P}_1$.

A problem Π is called *NP-hard* under polynomial reduction if for any problem $\Pi' \in \mathcal{NP}$, $\Pi' \leq_{poly} \Pi$.

A problem is Π is *NP-complete* (NPC) if it is NP-hard and $\Pi \in \mathcal{NP}$.

Therefore these are problems that are hardest *within* the class $\mathcal{NP}$.

Exercise 9.2 *If problems* A *and* B *are NPC, then* $A \leq_{poly} B$ *and* $B \leq_{poly} A$*.*

From the previous exercise, these problems form a kind of equivalent class with respect to polynomial time reductions. However, a crucial question that emerges at this juncture is : *Do NPC problems actually exist ?*. A positive answer to this question led to the development of one of the most fascinating areas of Theoretical Computer Science and will be addressed in the next section.

Here is actually a simple proof that NPC problems exist but this problem is somewhat artificial as it uses the machine model explicity.

Let $TMSAT = \{< M, x, 1^n, 1^t > | \exists u, |u| \leq n$ and M accepts (x, u) in t steps $\}$.

Here M is the code of a deterministic Turing Machine that runs on input x with *advice* u. Note that since t is represented in unary, M runs in polynomial time *given* u[3].
If a language L is in $\mathcal{NP}$, there there must exist some u of length n^α and t bounded by n^β where α, β are constants. Therefore $x \in L$ iff there exists an advice string (non-deterministic choices) of length at most n^α such that the Turing Machine M_L accepts it in time n^β. This implies that $< M_L, x, n^\alpha, n^\beta > \in TMSAT$, i.e., any $\mathcal{NP}$ language can be reduced to $TMSAT$.

So far, we have only discussed many-one reducibility that hinges on the existence of a many-one polynomial time reduction function. There is another very useful and perhaps more intuitive notion of reducibility, namely, *Turing reducibility*. The many-to-one reduction may be thought of as using *one* subroutine call of P_2 to solve P_1 (when $P_1 \leq_{poly} P_2$) in polynomial time, if P_2 has a polynomial time algorithm. Clearly, we can afford a polynomial number of subroutine calls to the algorithm for P_2 and still get a polynomial time algorithms for P_1. In other words, we say that P_1 is *Turing-reducible* to P_2 if a polynomial time algorithm for P_2 implies a polynomial time algorithm for P_1. Moreover, we do not require that P_1, P_2 be decision problems. Although, this may seem to be the more natural notion of reducibility, we will rely on the more restrictive definition to derive the results.

[3]How much time does it take to simulate each of the t steps ?

9.2 Cook Levin theorem

Given a boolean formula in boolean variables, the *satisfiability* problem is an assignment of the truth values of the boolean variables that can make the formula evaluate to TRUE (if it is possible). If the formula is in a *conjunctive normal form* (CNF) [4], then the problem is known as CNF Satisfiability. Further, if we restrict the number of variables in each clause to be exactly k then it is known as the k-CNF Satisfiability problem. A remarkable result attributed to Cook and Levin says the following

Theorem 9.1 *The CNF Satisfiability problem is NP Complete under polynomial time reductions.*

To appreciate this result, you must realize that there are potentially infinite number of problems in the class $\mathcal{NP}$, so we cannot explicitly design a reduction function. Other than the definition of $\mathcal{NP}$ we have very little to rely on for a proof of the above result. A detailed technical proof requires that we define the computing model very precisely - it is beyond the scope of this discussion. Instead we sketch an intuition behind the proof.

Given an arbitrary problem $\Pi \in \mathcal{NP}$, we want to show that $\Pi \leq_{poly} CNF - SAT$. In other words, given any instance of Π, say I_Π, we would like to define a boolean formula $B(I_\Pi)$ which has a satisfiable assignment iff I_Π is a YES instance. Moreover the length of $B(I_\Pi)$ should be polynomial time constructable (as a function of the length of I_Π).

A computing machine is a transition system where we have

(i) An initial configuration
(ii) A final configuration that indicates whether or not the input is a YES or a NO instance
(iii) A sequence of intermediate configuration S_i where S_{i+1} follows from S_i using a valid transition. In a non-deterministic system, there can be more than one possible transition from a configuration. A non-deterministic machine *accepts* a given input iff there is some valid sequence of configurations that verifies that the input is a YES instance.

All the above properties can be expressed in propositional logic, i.e., by an unquantified boolean formula in a CNF. Using the fact that the number of transitions is polynomial, we can bound the size of this formula by a polynomial. The details can be quite messy and the interested reader can consult a formal proof in the context of Turing Machine model. Just to give the reader a glimpse of the kind of formalism used, consider a situation where we want to write a propositional formula to assert that a machine is in exactly one of the k states at any given time $1 \leq i \leq T$. Let us use boolean variables $x_{1,i}, x_{2,i} \ldots x_{k,i}$ where $x_{j,i} = 1$ iff the machine is in state j at time i. We must write a formula that will be a conjunction of two two conditions

[4] A formula, that looks like $(x_1 \vee x_2..) \wedge (x_i \vee x_j \vee ..) \wedge \ldots (x_\ell \vee \ldots x_n)$

(i) At least one variable is true at any time i:

$$(x_{1,i} \vee x_{2,i} \ldots x_{k,i})$$

(ii) At most one variable is true :

$$(x_{1,i} \Rightarrow \bar{x}_{2,i} \wedge \bar{x}_{3,i} \ldots \wedge \bar{x}_{k,i}) \wedge (x_{2,i} \Rightarrow \bar{x}_{1,i} \wedge \bar{x}_{3,i} \ldots \wedge \bar{x}_{k,i}) \ldots \wedge (x_{k,i} \Rightarrow \bar{x}_{1,i} \wedge \bar{x}_{2,i} \ldots \wedge \bar{x}_{k-1,i})$$

where the implication $a \Rightarrow b$ is equivalent to $\bar{a} \vee b$.

A conjunction of the above formula over all $1 \leq i \leq T$ has a satisfiable assignment of $x_{j,i}$ iff the machine is in exactly one state (not necessarily the same state) at each of the time instances. The other condition should capture which states can succeed a given state.

We have argued that $CNF - SAT$ is NP-hard. Since we can guess an assignment and verify the truth value of the Boolean formula, in linear time, we can claim that $CNF - SAT$ is in $\mathcal{NP}$.

9.3 Common NP complete problems

To prove that a given problem P is NPC, the standard procedure is to establish that

(i) $P \in \mathcal{NP}$: This is usually the easier part.
(ii) $CNF - SAT \leq_{poly} P$. We already know that any $P' \in \mathcal{NP}$, $P' \leq_{poly} CNF - SAT$. So by transitivity, $P' \leq_{poly} P$ and therefore P is NPC.

The second step can be served by reducing any known NPC to P. Some of the earliest problems that were proved NPC include (besides CNF-SAT)

- 3D Matching
- Three colouring of graphs
- Equal partition of integers
- Maximum Clique /Independent Set
- Hamilton cycle problem
- Minimum set cover

9.3.1 Other important complexity classes

While the classes $\mathcal{P}$ and $\mathcal{NP}$ hogs the maximum limelight in complexity theory, there are many other related classes in their own right.

- $co-\mathcal{NP}$ A problem whose complement is in $\mathcal{NP}$ belongs to this class. If the problem is in $\mathcal{P}$, then the complement of the problem is also in $\mathcal{P}$ and hence in $\mathcal{NP}$. In general we can't say much about the relation between $\mathcal{P}$ and $co-\mathcal{NP}$. In general, we can't even design an NP algorithm for a problem in $co-\mathcal{NP}$, i.e. these problems are not efficiently verifiable. For instance how would you verify that a boolean formula is *unsatisfiable* (all assignments make it false) ?

  Exercise 9.3 *Show that the complement of an NPC problem is complete for the class $co-\mathcal{NP}$ under polynomial time reduction.*

  Exercise 9.4 *What would it imply if an NPC problem and its complement are polynomial time reducible to eachother ?*

- $\mathcal{PSPACE}$ The problems that run in polynomial space (but not necessarily polynomial time). The satisfiability of *Quantified Boolean Formula* (QBF) is a complete problem for this class.

- **Randomized classes** Depending on the type of randomized algorithms (mainly Las Vegas or Monte Carlo) , we have the following important classes

  - $\mathcal{RP}$: Randomized Polynomial class of problems are characterized by (Monte Carlo) randomized algorithms A such that

    If $x \in L \Rightarrow \Pr[A \text{ accepts } x] \geq 1/2$
    If $x \notin L \Rightarrow \Pr[A \text{ accepts } x] = 0$

    These algorithms can err on one side.

  - $\mathcal{BPP}$ When a randomized algorithm is allowed to err on both sides

    If $x \in L \Rightarrow \Pr[A \text{ accepts } x] \geq 1/2 + \varepsilon$
    If $x \notin L \Rightarrow \Pr[A \text{ accepts } x] \leq 1/2 - \varepsilon$

    where ε is a fixed non zero constant.

  - $\mathcal{ZPP}$ Zero Error Probabilistic Polynomial Time : These are the Las Vegas kind that do not have any errors in the answer but the running time is expected polynomial time.

  One of the celebrated problems, involving randomized algorithms is

  $$\mathcal{BPP} \subset \mathcal{NP}?$$

9.4 Combating hardness with approximation

Since the discovery of NPC problems in early 70's , algorithm designers have been wary of spending efforts on designing algorithms for these problems as it is considered to be a rather

hopeless situation without a definite resolution of the $\mathcal{P} = \mathcal{NP}$ question. Unfortunately, a large number of interesting problems fall under this category and so ignoring these problems is also not an acceptable attitude. Many researchers have pursued non-exact methods based on heuristics to tackle these problems based on heuristics and empirical results [5]. Some of the well known heuristics are *simulated annealing*, *neural network* based learning methods , *genetic algorithms*. You will have to be an optimist to use these techniques for any critical application.

The accepted paradigm over the last decade has been to design polynomial time algorithms that guarantee *near-optimal* solution to an optimization problem. For a maximization problem, we would like to obtain a solution that is at least $f \cdot OPT$ where OPT is the value of the optimal solution and $f \leq 1$ is the *approximation factor* for the *worst case* input. Likewise, for minimization problem we would like a solution no more than a factor $f \geq 1$ larger than OPT. Clearly the closer f is to 1, the better is the algorithm. Such algorithm are referred to as *Approximation* algorithms and there exists a complexity theory of approximation. It is mainly about the extent of approximation attainable for a certain problem.

For example, if $f = 1 + \varepsilon$ where ε is any user defined constant, then we way that the problem has a *Polynomial Time Approximable Scheme* (PTAS). Further, if the algorithm is polynomial in $1/\varepsilon$ then it is called FPTAS (Fully PTAS). The theory of *hardness of approximation* has yielded lower bounds (for minimization and upper bounds for maximization problems) on the approximations factors for many important optimization problems. A typical kind of result is that *Unless $\mathcal{P} = \mathcal{NP}$ we cannot approximate the set cover problem better than* $\log n$ *in polynomial time.*

In this section, we give several illustrative approximation algorithms. One of the main challenges in the analysis is that even without the explicit knowledge of the optimum solutions, we can still prove guarantees about the quality of the solution of the algorithm.

9.4.1 Equal partition

Given n integers $S = \{z_1, z_2 \ldots z_n\}$, we want to find a partition $S_1, S - S_1$, such that $|\left(\sum_{x \in S_1} x\right) - \left(\sum_{x \in S - S_1} x\right)|$ is minimized. A partition is *balanced* if the above difference is zero.

Let $B = \sum_i z_i$ and consider the following a generalization of the problem, namely, the subset sum problem. For a given integer $K \leq B$, is there a subset $R \subset S$ such that the elements in R sum up to K.

Let $S(j, r)$ denote a subset of $\{z_1, z_2 \ldots z_j\}$ that sums to r - if no such subset exists then we define it as ϕ (empty subset). We can write the following recurrence

$S(j, r) = S(j-1, r-z_j) \cup z_j$ if z_j is included or $S(j-1, r)$ if z_j is not included or ϕ not possible

Using the above dynamic programming formulation we can compute $S(j, r)$ for $1 \leq j \leq n$

[5] The reader must realize that our inability to compute the actual solutions makes it difficult to evaluate these methods in in a general situation.

and $r \leq B$. You can easily argue that the running time is $O(n \cdot B)$ which may not be polynomial as B can be very large.

Suppose, we are given an approximation factor ε and let $A = \lceil \frac{n}{\varepsilon} \rceil$ so that $\frac{1}{A} \leq \varepsilon/n$. Then we define a new scaled problem with the integers scaled as $z'_i = \lfloor \frac{z_i}{z/A} \rfloor$ and let $r' = \lfloor \frac{r}{z/A} \rfloor$ where z is the maximum value of an integer that can participate in the solution [6].

Let us solve the problem for $\{z'_1, z'_2 \ldots z'_n\}$ and r' using the previous dynamic programming strategy and let S'_o denote the optimal solution for the scaled problem and let S_o be the solution for the original problem. Further let C and C' denote the cost function for the original and the scaled problems respectively. The running time of the algorithm is $O(n \cdot r')$ which is $O(\frac{1}{\varepsilon}n^2)$. We would like to show that the cost of $C(S'_o)$ is $\geq (1-\varepsilon)C(S_o)$. For any $S'' \subset S$

$$C(S'') \cdot \frac{n}{\varepsilon z} \geq C'(S'') \geq C(S'') \cdot \frac{n}{\varepsilon z} - |S''|$$

So

$$C(S'_o) \geq C'(S'_o)\frac{\varepsilon z}{n} \geq C'(S_o)\frac{\varepsilon z}{n} \geq \left(C(S_o)\frac{n}{\varepsilon z} - |S_o|\right)\frac{\varepsilon z}{n} = C(S_o) - |S_o|\frac{\varepsilon z}{n}$$

The first and the third inequality follows from the previous bound and the second inequality follows from the optimality of S'_o wrt C'. Since $C(S_o) \geq \frac{z|S_o|}{n}$ and so $C(S'_o) \geq (1-\varepsilon)C(S_o)$

9.4.2 Greedy set cover

Given a ground set $S = \{x_1, x_2 \ldots x_n\}$ and a family of subsets $S_1, S_2 \ldots S_m \quad S_i \subset S$, we want to find a minimum number of subsets from the family that covers all elements of S. If S_i have associated weights $C()$, then we try to minimize the total weight of the set-cover.

In the greedy algorithm, we pick up a subset that is most *cost-effective* in terms of the cost per unchosen element. The cost-effectiveness of a set U is defined by $\frac{C(U)}{U-V}$ where $V \subset S$ is the set of elements already covered. We do this repeatedly till all elements are covered.

Let us number the elements of S in the order they were covered by the greedy algorithm (wlog, we can renumber such that they are $x_1, x_2 \ldots$). We will apportion the cost of covering an element $e \in S$ as $w(e) = \frac{C(U)}{U-V}$ where e is covered for the first time by U. The total cost of the cover is $= \sum_i w(x_i)$.

Claim 9.4

$$w(x_i) \leq \frac{C_o}{n-i+1}$$

where C_o is the cost of an optimum cover.

In iteration i, the greedy choice is more cost effective than any left over set of the optimal cover. Suppose the cost-effectiveness of the best set in the optimal cover is C'/U', i.e. $C'/U' = \min\left\{\frac{C(S_{i_1})}{S_{i_1}-S}, \frac{C(S_{i_2})}{S_{i_2}-S} \ldots \frac{C(S_{i_k})}{S_{i_k}-S}\right\}$ where $S_{i_1}, S_{i_2} \ldots S_{i_k}$ forms a minimum set cover. It follows that

$$C'/U' \leq \frac{C(S_{i_1}) + C(S_{i_1}) + \ldots C(S_{i_1})}{(S_{i_1}-S) + (S_{i_2}-S) + \ldots (S_{i_k}-S)} \leq \frac{C_o}{n-i+1}$$

[6] It is a lower bound to the optimal solution - for the balanced partition, it is the maximum integer less than $B/2$.

So $w(x_i) \leq \frac{C_o}{n-i+1}$.

Thus the cost of the greedy cover is $\sum_i \frac{C_o}{n-i+1}$ which is bounded by $C_o \cdot H_n$. Here $H_n = \frac{1}{n} + \frac{1}{n-1} + \ldots 1$.

Exercise 9.5 *Formulate the Vertex cover problem as an instance of set cover problem. Analyze the approximation factor achieved by the following algorithm. Construct a maximal matching of the given graph and consider the union C of the end-points of the matched edges. Prove that C is a vertex cover and the size of the optimal cover is at least $C/2$. So the approximation factor achieved is better than the general set cover.*

9.4.3 The metric TSP problem

If the edges of the graph satisfies triangle inequality, i.e., for any three vertices u, v, w $C(u, v) \leq C(u, w) + C(w, v)$, then we can design an approximation algorithm for the TSP problem as follows.

Metric TSP on graphs
Input: A graph G = (V,E) with weights on edges that satisfy triangle inequality.

1. Find a Minimum Spanning Tree T of G.
2. Double every edge - call the resulting graph E' and construct an Euler tour $\mathcal{T}$.
3. In this tour, try to take shortcuts if we have visited a vertex before.

Claim 9.5 *The length of this tour no more than twice that of the optimal tour.*

$MST \leq TSP$, therefore $2 \cdot MST \leq 2 \cdot TSP$. Since shortcuts can only decrease the tour length (because of the triangle inequality), the tour length is no more than twice that of the optimal tour.

9.4.4 Three colouring

We will rely on the following simple observation. If a graph is three colourable, its neighbourhood must be triangle free (or else the graph will contain a four-clique) i,e,, it must be 2 colourable,

Observation 9.1 *A graph that has maximum degree Δ can be coloured using $\Delta+1$ colours using a greedy strategy.*

Use a colour that is different from its already coloured neighbours.

Given a 3-colourable graph $G = (V, E)$, separate out vertices that have degrees $\geq \sqrt{n}$ - call this set H. Remove the set H and its incident edges and denote this graph by $G' = (V', E')$. Note that G' is 3-colourable and all vertices have degrees less than $\sqrt{n}$

and so from our previous observation, we can easily colour using $\sqrt{n}+1$ colours. Now, reinsert the the vertices of H and use an extra $|H|$ colours to complete the colouring. Since $|H| \leq \sqrt{n}$, we have used at most $2\sqrt{n}$ colours.

It is a rather poor approximation since we have used significantly more colours than three. However, it is known that unless $\mathcal{P} = \mathcal{NP}$, any polynomial time colouring algorithm will use $\Omega(n^{\epsilon})$ colours for some fixed $\epsilon > 0$.

9.4.5 Maxcut

Problem Given a graph $G = (V, E)$, we want to partition the vertices into sets $U, V - U$ such that the number of edges across U and $V - U$ is maximized. There is a corresponding weighted version for a weighted graph with a weight function $w : E \to \mathbb{R}$.

We have designed a polynomial time algorithm for mincut but the maxcut is an NP-hard problem. Let us explore a simple idea of randomly assigning the vertices to one of the partitions. For any fixed edge $(u, v) \in E$, it either belongs to the optimal maxcut or not depending on whether u, v belong to different partitions of the optimal maxcut M_o. The probability that we have chosen the the right partitions is at least half. Let X_e be a random 0-1 variable (also called indicator random variables) that is 1 iff the algorithm choses it consistently with the maxcut. The expected size of the cut produced by the algorithm is

$$E[\sum_e w(e) \cdot X_e] = \sum_e w(e) \cdot E[X_e] \geq M_o/2$$

Therefore we have a simple randomized algorithm that attains a $\frac{1}{2}$ apprximation.

Exercise 9.6 *For an unweighted graph show that a simple greedy strategy leads to a $\frac{1}{2}$ approximation algorithm.*