Lecture Notes For

Language Hierarchies and Algorithmic Complexity

Gordon J. Pace

Department of Computer Science & A.I.
Faculty of Science
University of Malta

February 1999

Contents

Part I

In the beginning . . .

Chapter 1

Introduction

This course is a continuation of the first year course 'Introduction to Formal Languages and Automata'. That course dealt mainly with the properties of regular languages and context free grammars, the most commonly used languages in computer science. This course starts off where the previous one left. We will briefly review some of the important results of the previous course from a slightly different perspective and prove further results now more accessible to you, a (hopefully) more mathematically mature audience. We will be looking at the two language classifications already examined as part of a larger hierarchy of languages — the Chomsky hierarchy.

Needless to say, the more general the language class, the more complex is a machine which recognizes the language. This was already evident when we had to discard finite state automata in favour of non-deterministic pushdown automata as context free language recognizers. We will see that for the most general class of languages we will need the full power of a Turing machine. This relation is, however, not single way. The most general grammars are in fact as expressive as general Turing machines, providing us with another equivalence between computational devices, further strengthening evidence for Church's thesis.

Equivalence between Turing machines and grammars leads naturally to proofs using the famous halting problem (there cannot exist a Turing machine capable of divining, in general, whether a Turing machine with a given input will terminate). Unsolvability is a powerful result, but at times can be far too powerful. As we will see certain problems are solvable in theory, but would be completely inherently impractical to use, since they yield to no known efficient algorithm. Imagine if you were to come up with a complete, foolproof algorithm, which can predict, to any desired accuracy the movements of the global financial market. Have your dreams of piles of gold coins and casual strolls down exotic beaches reading mathematical texts finally come true[1]? Not if your algorithm can only give results for the next few minutes after deliberating for a few thousand times over the age of the universe and cannot be made inherently more efficiently. This is the gist behind intractability, which will then round up the course.

The main text books used for this course are listed below together with a brief comment:

G.J. Pace, *Lecture Notes For 'Language Hierarchies and Algorithmic Complexity'*, 1998.

The set of notes you are reading. These should suffice for most of the course content, although I strongly advice reading through at least one major textbook suggested below.

G.J. Pace, *Lecture Notes for 'An Introduction to Formal Languages and Automata'*, 1998.

This is a set of note I have prepared for the first year course about formal languages. It may be found useful to have a copy to take a look at more detailed proofs and examples which we will only gloss over during the first part of the course.

V.J. Rayward-Smith, *A First Course in Formal Language Theory*, McGraw-Hill Computer Science Series, 1995.

The contents of this will briefly be covered during the first part of the course. Your first year notes, together with these notes, however, should suffice.

[1]No, actually this is not my fantasy, but D.E. Knuth's. For a most exotic mathematical exposition, may I suggest you read his book 'Surreal Numbers', where two castaways discover the beauty of Conway's Surreal numbers on a desert island!

V.J. Rayward-Smith, *A First Course in Computability*, McGraw-Hill Computer Science Series, 1995.

A substantial part of this book will be covered during the course.

M.R. Garey & D.S. Johnson, *Computers and Intractability — A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

The final part of this course coincides with the first part of this book.

M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

A recently published book, useful from all aspects of the course. The approach is slightly different from the one we use, but should be readily accessible.

J.E. Hopcroft & J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Series in Computer Science, 1979.

A good all-rounder. This book should cover all aspects of the course, although not always using the same approach as will be used during the lectures. You may see this as a bonus (alternative point of view, different approach may be more to your liking) or a not (more effort needed to follow text).

This list could obviously go on and on. I believe that the notes I am providing should be sufficient for an overall understanding of the subject. However, this is not enough. I suggest that you take a look at some of the above books and read through the more useful parts. Due to the structure of the syllabus, we will be mainly be following the two Rayward-Smith texts and finish with the Garey and Johnson one. I think that you would also benefit from reading through one of the last two books to get a feeling of alternative sequences and methods of presentation.

Part II

The Chomsky Language Hierarchy

Chapter 2

Language Classification

2.1 Languages and Grammars

This chapter presents the Chomsky classification of languages. Since you are assumed to be familiar with the concepts of alphabets, strings, languages and grammars, this chapter will only mention them in passing. If you do not remember the notation, definitions or basic results, I suggest that you take a look at one of the textbooks or the set of notes which I have prepared for the first year course: 'Introduction to Formal Languages and Automata'.

Recall that a language is simply a, possibly infinite, set of strings over a finite alphabet Σ. How can we specify an infinite set using a finite notation? One way is to use set comprehension:

$$AB \stackrel{def}{=} \{a^n b^n c^n \mid n \in \mathbb{N}\}$$

However, this is not always so convenient to use, since it quickly becomes large and difficult to understand. Various other solutions exist. The one which we will be starting off from is that of a grammar. A grammar is simply a set of rules used to produce a language.

Definition: A *phrase structure grammar* is a 4-tuple $\langle \Sigma,\ N,\ S,\ P\rangle$ where:

Σ is the alphabet over which the grammar generates strings.
N is a set of non-terminal symbols.
S is one particular non-terminal symbol.
P is a relation of type $(\Sigma \cup N)^+ \times (\Sigma \cup N)^*$.

It is assumed that $\Sigma \cap N = \emptyset$.

Variables for non-terminals will be represented by uppercase letters and a mixture of terminal and non-terminal symbols will usually be represented by greek letters. G is usually used as a variable ranging over grammars.

We still have to formalise what we mean by a particular string being generated by a certain grammar.

Definition: A string β is said to derive immediately from a string α in grammar G, written $\alpha \Rightarrow_G \beta$, if we can apply a production rule of G on a substring of α obtaining β. Formally:

$$\begin{array}{rcl} \alpha \Rightarrow_G \beta & \stackrel{def}{=} & \exists \alpha_1, \alpha_2, \alpha_3, \gamma \cdot \\ & & \quad \alpha = \alpha_1 \alpha_2 \alpha_3 \quad \wedge \\ & & \quad \beta = \alpha_1 \gamma \alpha_3 \quad \wedge \\ & & \quad \alpha_2 \rightarrow \gamma \in P \end{array}$$

But single step derivations are not the only thing we would like to perform. With this in mind, we define the following relational closures of $\Rightarrow_G$:

$$\begin{array}{rcl} \alpha \stackrel{0}{\Rightarrow}_G \beta & \stackrel{def}{=} & \alpha = \beta \\ \alpha \stackrel{n+1}{\Rightarrow}_G \beta & \stackrel{def}{=} & \exists \gamma \cdot \alpha \Rightarrow_G \gamma \wedge \gamma \stackrel{n}{\Rightarrow}_G \beta \\ \alpha \stackrel{*}{\Rightarrow}_G \beta & \stackrel{def}{=} & \exists n \geq 0 \cdot \alpha \stackrel{n}{\Rightarrow}_G \beta \\ \alpha \stackrel{+}{\Rightarrow}_G \beta & \stackrel{def}{=} & \exists n \geq 1 \cdot \alpha \stackrel{n}{\Rightarrow}_G \beta \end{array}$$

Definition: A string $\alpha \in (N \cup \Sigma)^*$ is said to be in sentential form in grammar G if it can be derived from S, the start symbol of G. $\mathcal{S}(G)$ is the set of all sentential forms in G:

$$\mathcal{S}(G) \stackrel{def}{=} \{\alpha : (N \cup \Sigma)^* \mid S \stackrel{*}{\Rightarrow}_G \alpha\}$$

Definition: Strings in sentential form built solely from terminal symbols are called *sentences*.

These definitions indicate clearly what we mean by the language generated by a grammar G. It is simply the set of all strings of terminal symbols which can be derived from the start symbol in any number of steps.

Definition: The language generated by grammar G, written as $\mathcal{L}(G)$ is the set of all sentences in G:

$$\mathcal{L}(G) \stackrel{def}{=} \{x : \Sigma^* \mid S \stackrel{+}{\Rightarrow}_G x\}$$

Example: Thus, to define the initial example language using a grammar, we can use $G = \langle \Sigma,\ N,\ S,\ P \rangle$, where:

$$\begin{array}{rcl} \Sigma & = & \{a,\ b\} \\ N & = & \{S\} \\ P & = & \{\ \ S \to aSb, \\ & & \quad\ \ S \to \varepsilon \qquad \} \end{array}$$

Recall, that for brevity, we will combine together multiple production rules with the same left hand side as follows: $\alpha \to \beta_1$ to $\alpha \to \beta_n$ will be written as $\alpha \to \beta_1 | \beta_2 \ldots \beta_n$. Note that this notation does *not* allow the introduction of infinite production rules by using trailing dots.

The above production rules can thus be written as $S \to aSb|\varepsilon$.

Proving properties of languages is usually done via induction over the length of string or derivation. For example, to prove that $AB = \mathcal{L}(G)$ (with AB and G as defined above), we would start by proving that the set of sentential forms over G, forms the set $\{a^nSb^n,\ a^nb^n \mid n \in \mathbb{N}\}$. It then follows that, since $\mathcal{L}(G) = \mathcal{S}(G) \cap \Sigma^*$, and only the second type of string has no non-terminals, $\mathcal{L}(G) = AB$.

To prove that $\mathcal{S}(G) = \{a^nSb^n,\ a^nb^n \mid n \in \mathbb{N}\}$, we split the proof into two parts:

Part 1: We first prove that $\mathcal{S}(G) \subseteq \{a^nSb^n,\ a^nb^n \mid n \in \mathbb{N}\}$. Let $x \in \mathcal{S}(G)$. We now prove, by induction on the length of the derivation of x that x must have the structure a^nSb^n or a^nb^n.

For $n = 1$, we get $S \Rightarrow_G x$. Thus, x is either ε $(= a^0 b^0)$ or aSb, satisfying the requirement in both cases.

Assume that the argument holds for derivations of length k. Now consider a derivation of length $k+1$. $S \overset{k+1}{\Rightarrow}_G$ implies that, for some α, $S \overset{k}{\Rightarrow}_G \alpha \Rightarrow_G x$. But, by the inductive hypothesis, α is either of the form $a^n b^n$ or $a^n S b^n$. Clearly, it cannot be in the first form (no further productions can take place), hence we have:

$$S \overset{k}{\Rightarrow}_G a^n S b^n \Rightarrow_G x$$

Clearly, by definition of $\Rightarrow_G$, we get that $x = a^{n+1} S b^{n+1}$ or $x = a^{n+1} b^{n+1}$, hence satisfying the requirement that $x \in \{a^n S b^n,\ a^n b^n \mid n \in \mathbb{N}\}$.

This completes the induction.

Part 2: Secondly, we prove that $\{a^n S b^n,\ a^n b^n \mid n \in \mathbb{N}\} \subseteq \mathcal{S}(G)$.

The proof is easy (just use induction on n) and is thus omitted.

$\square$

Now consider the grammar $G' = \langle \Sigma',\ N',\ S',\ P' \rangle$, where:

$$\begin{array}{rcl} \Sigma' & = & \{a,\ b\} \\ N' & = & \{S', A, B\} \\ P' & = & \{\ \ S' \rightarrow aA \mid Bb \mid \varepsilon, \\ & & \quad\ A \rightarrow S'b \\ & & \quad\ B \rightarrow aS' \qquad\qquad \} \end{array}$$

It should be clear that the language generated by G' is also identical to the set AB. Thus, it can be said that grammar G and grammar G' are equivalent despite the different sets of production rules.

This is the motivation behind the following definition:

Definition: Two grammars G_1 and G_2 are said to be *equivalent* if they produce the same language: $\mathcal{L}(G_1) = \mathcal{L}(G_2)$.

2.2 The Chomsky Hierarchy

Clearly, not all languages are of the same complexity. Certain languages are more difficult than others to recognise. But how can we measure this complexity? Grammars provide a straightforward way of classifying languages, depending on the complexity of the production rules.

We start by defining a hierarchy of grammars, in differing complexity of their production rules, and then we can define the 'complexity level' of a language to be the level of the simplest grammar which generates it. This is precisely what the Chomsky hierarchy is all about.

Definition: A grammar $G = \langle \Sigma,\ N,\ S,\ P \rangle$ is said to be of type i, if it satisfies the corresponding conditions given in the following table.

Type	Conditions
0	No conditions
1	All productions $\alpha \to \beta$ must satisfy $\lvert\alpha\rvert \leq \lvert\beta\rvert$.
2	All productions are of the form $A \to \alpha$, where $A \in N$.
3	All productions are of the form $A \to aB$ or $A \to a$, where $A, B \in N$ and $a \in \Sigma$.

A language is of type i, if there is a grammar of type i which generates it. We define $\mathcal{L}_i$ to be the class of all languages of type i.

Certain books give a different restriction for type 1 grammars: all productions must be of the form $\alpha A \gamma \to \alpha\beta\gamma$, where $A \in N$ and $\beta \neq \varepsilon$ (except that we allow the rule $S \to \varepsilon$, as long as S does not then appear on the right hand side of any rules). It can be shown that these two restrictions are, in fact, equivalent — that is for every grammar satisfying the original restriction, there is an equivalent grammar satisfying the alternative ones, and vice-versa). Note that in the alternative definition, α and γ, give a context to the non-terminal A, specifying when and how it can be used.

To avoid confusion we also assign a name to each class of grammars (and thus, languages). Type 0 grammars are called phrase structure grammars; type 1 are called context sensitive grammars, due to the alternative restrictions; type 2 are called context free grammars, since the way non-terminals can evolve is independent of their context; finally, type 3 grammars are called regular grammars.

2.3 Properties of Languages

Why does a mathematician abstract over concrete instances? The aim is always to have a simpler, more general model, which can then provide results which extend over a whole family of instances. However, abstraction needs to be done over the right property. Consider the gravitation laws as developed by Newton. The equation (rightly) differentiates between objects of different mass but not between ones of different colour. In other words, the theory abstracts colour away and partitions objects by their mass. Clearly, Newton could have abstracted away mass and left colour distinction. However, he would have quickly found that two 'identical' objects — a red pencil and the red planet Mars, have different behaviour within the realms of our interest. This is precisely what we do not wish to happen. Is the language classification meaningful? Does it have any actual relevance to what we would intuitively like to call 'the complexity of a language'?

The results given in the first year course seem to indicate that, at least as far as types 2 and 3 are concerned, there seems to be an increase in the computational power necessary to parse a language of type 2 (as opposed to one of type 3). This is the kind of result we now desire to show to be the case of all the language classes. Below are enlisted a number of results, we will show throughout the course:

Hierarchy: The classification is in fact a hierarchy.

> **Inclusion:** As you may have noticed, a grammar can be of more than one type. In particular, any grammar is of type 0. Can we show that any grammar of type $i+1$ is necessarily of type i?
>
> $$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0$$
>
> **Discrimination:** It starts off with a wide population of languages at its base and gradually narrows down. In other words, at every step, we are discarding languages, and we do not have any case where $\mathcal{L}_i = \mathcal{L}_{i+1}$.

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$$

Closure: A number of operations on languages (in particular union, catenation and Kleene closure) are closed with respect to the classification. In other words, the union of two languages of type i is itself of type i. Similarly for catenation and Kleene closure.

Computational complexity: The higher the type of a language, the simpler is the machine necessary to recognise it. This kind of analysis has already been done in the first year course where it was shown that $\mathcal{L}_3$ is in fact exactly the set of languages recognisable by deterministic finite state automata and $\mathcal{L}_2$ is the class of languages recognisable by non-deterministic pushdown automata. What about classes $\mathcal{L}_1$ and $\mathcal{L}_0$?

We start by proving the inclusion property of the hierarchy and will then prove the remaining results in the coming chapters.

Theorem: The Chomsky hierarchy is inclusive.

$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0$$

Proof:

- $\mathcal{L}_3 \subseteq \mathcal{L}_2$. Let L be a language of type 3. Then there is a grammar G ($G = \langle \Sigma,\ N,\ S,\ P \rangle$) of type 3 such that $\mathcal{L}(G) = L$. Since the grammar is of type 3, all the productions in P are of the form $A \to aB$ or $A \to a$ ($A, B \in N$ and $a \in \Sigma$). They are thus in the form $A \to \alpha$ ($A \in N$ and $\alpha \in (N \cup \Sigma)^+$) satisfying the conditions of type 2. G is thus a type 2 grammar. Therefore $L \in \mathcal{L}_2$.

- $\mathcal{L}_2 \subseteq \mathcal{L}_1$. As before, let L be a language of type 2. Then there is a grammar G ($G = \langle \Sigma,\ N,\ S,\ P \rangle$) of type 2 such that $\mathcal{L}(G) = L$. Since G is of type 2, all its production rules are of the form $\alpha \to \beta$. where $\alpha \in N$ and $\beta \in (N \cup \Sigma)^+$. But $|\alpha| = 1 \leq |\beta|$ implying that all the productions conform to class 1 restrictions.

- $\mathcal{L}_1 \subseteq \mathcal{L}_0$. Since any grammar is of type 0 this is trivially true.

$\square$

2.4 Exercises

The following results are be quite shocking. Prove that:

1. the class of all languages is uncountable
2. the class of all languages of type 0 is countable

Does this indicate that grammars are not, in fact, a very clever way of specifying languages? Can we somehow express languages in a more effective way? One main result of this course is to show that there is no language inexpressible as a grammar for which we can write a computer program to recognise. But more of this later ...

Chapter 3

Regular Languages

3.1 Definition

Recall the definition of a regular (type 3) grammar:

Definition: A grammar $G = \langle N, \Sigma, S, P \rangle$ is said to be regular if all the productions in P are of the form $A \to aB$, $A \to a$ or $A \to \varepsilon$, where $A, B \in N$ and $a \in \Sigma$.

A regular language is a language for which there is a regular grammar which generates it.

This class of languages is also sometimes referred to as right-linear languages. Left-linear languages are the class of languages generated by grammars with productions of the form $A \to a$ or $A \to Ba$ ($A, B \in N$ and $a \in \Sigma$). It is possible to show that the two classes are in fact equivalent.

Note that it is simple to add or remove the empty string from a regular language.

- Removing ε simply involves removing $S \to \varepsilon$.
- Adding ε is slightly more involved. We first add a new start symbol S' and production rule $S' \to \varepsilon$. For every rule $S \to \alpha$ we also add the rule $S' \to \alpha$.

  Give an example of a regular grammar where simply adding $S \to \varepsilon$ would give th wrong result.

3.2 ε-Productions

Definition: A grammar is said to be ε-*free* is none of its production rules is of the form $A \to \varepsilon$ except possibly $S \to \varepsilon$, in which case S (the start symbol) does not appear anywhere on the right hand side of production rules.

Proposition: For every regular grammar there exists an equivalent ε-free regular grammar.

Strategy: The proof proceeds by construction:

1. For every rule of the form $A \to aB$, such that $B \to \varepsilon$ is also a rule, add the production $A \to a$.
2. Remove rules of the form $A \to \varepsilon$.
3. If $S \to \varepsilon$ was one of the productions we add ε to the language using the procedure described earlier.

The proof will not be taken into any further detail here.

□

Note that when proving closure properties, this proposition allows us to assume that the grammars are ε-free.

3.3 Closure

Regular languages are closed under union, catenation and Kleene closure.

We simply mention how these can be constructed without going into proof details. The construction of a regular grammar generating the union, catenation or Kleene closure of regular languages guarantees closure of these operations over the class $\mathcal{L}_3$.

Union: To simplify the problem, we consider only regular languages which do not include ε. The previous methods to remove and add ε can then be used to generalize the result.

Given two regular grammars G_1 and G_2 with disjoint non-terminal alphabet, we would like to construct an alternative grammar with a new start symbol S

and two new production rules $S \to S_1|S_2$. However, these new rules are not acceptable in type 3 grammars. The solution is similar to that for the empty string: for all productions $S_1 \to \alpha$ we make a copy $S \to \alpha$, and similarly for productions from S_2. Thus, the new production rule set will be:

$$P_1 \cup P_2 \cup \{S \to \alpha \mid S_1 \to \alpha \in P_1 \text{ or } S_2 \to \alpha \in P_2\}$$

Catenation: Consider two regular languages L_1 and L_2. Assume $\varepsilon \notin L_i$ for either i. If regular grammars G_1 and G_2 generate the two languages (where $G_i = \langle N_i, \Sigma, S_i, P_i \rangle$) with discrete non-terminal symbols, we can construct a regular grammar to accept exactly $L_1 L_2$ by *replacing* all productions $A \to a$ ($A \in N$ and $a \in \Sigma$) in P_1 by $A \to aS_2$:

$$P_1 \cup P_2 \cup \{A \to aS_2 \mid A \to a \in P_1\}$$

This procedure works for ε-free grammars which do not include ε. If ε is in either of the two languages, we can still obtain the desired result since:

- If $\varepsilon \in L_1 \cap L_2$, then $L_1 L_2 = \{\varepsilon\} \cup L_1 \cup L_2 \cup (L_1 \setminus \{\varepsilon\})(L_2 \setminus \{\varepsilon\})$.
- If $\varepsilon \in L_1 \setminus L_2$, then $L_1 L_2 = L_2 \cup (L_1 \setminus \{\varepsilon\}) L_2$.
- If $\varepsilon \in L_2 \setminus L_1$, then $L_1 L_2 = L_1 \cup L_1 (L_2 \setminus \{\varepsilon\})$.

Note that in all cases, we know how to perform the desired operations in the equivalences (which are also easily verifiable).

Kleene Closure: Again, we start by removing the empty string from the language (if it is there). Then, for every rule $A \to a$ ($A \in N$ and $a \in \Sigma$), we add a new rule $A \to aS$ to allow for arbitrary repeats of strings from the language.

$$P \cup \{A \to aS \mid A \to a \in P\}$$

Since $\varepsilon \in L^*$ for any language L, we finally add the empty string as already described earlier.

3.4 Recognition

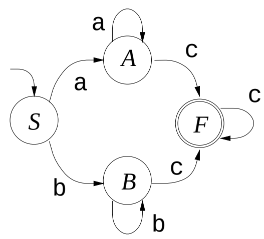
Regular languages can be recognized by a very simple form of automaton, a deterministic, finite state automaton. Basically, this is a machine with a finite number of states and which communicates with the outside world via an input tape but which has no form of 'memory'.

The behaviour of a deterministic finite state automaton (DFSA) can be described as follows:

1. Start off in a defined initial state.
2. If the current state is a final one, the machine may terminate.
3. Read a symbol off the input tape.
4. Depending on the current state and input, choose the next possible state (using a fixed internal transition table).
5. Jump back to instruction 2.

If a state-input combination occurs for which there is no defined next state, the machine 'breaks' and cannot continue computing. The relevance of the final states, is that they are used to decide which strings the machine accepts. If, upon input w the machine reaches a final state (when the whole input has been read and used), we say that the machine accepts w. If the machine breaks or ends up in a non-final state, we say that the string is rejected.

We usually draw finite state automata, as shown below:



Every labeled circle is a *state* of the machine, where the label is the name of the state. If the internal transition table says that from a state A and with input a, the machine goes to state B, this is represented by an arrow labeled a going from the circle labeled A to the circle labeled B. The initial state from where the automaton originally starts is marked by an unlabeled incoming arrow. Final states are drawn using two concentric circles rather than just one.

Imagine that the machine shown in the previous diagram were to be given the string `aac`. The machine starts off in state S with input `a`. This sends the machine to state A, reading input `a`. Again, this sends the machine to state A, this time reading input `c`, which sends it to state F. The input string has finished, and the automaton has ended in a final state. This means that the string has been accepted.

Similarly, with input string `bcc` the automaton visits these states in order: S, B, F, F. Since after finishing with the string, the machine has ended in a terminal state, we conclude that `bcc` is also accepted.

Now consider the input `a`. Starting from S the machine goes to A, where the input string finishes. Since A is not a terminal state `a` is not an accepted string.

Alternatively consider any string starting with `c`. From state S, there is no outgoing arrow labeled `c`. The machine thus 'breaks' and any string starting in `c` is not accepted.

Finally consider the string `aca`. The automaton goes from S (with input `a`) to A (with input `c`) to F (with input `a`). Here, the machine 'breaks' and the string is rejected. Note that even though the machine broke in a terminal state the string is not accepted.

Definition: A deterministic finite state automaton is a 5-tuple:

$$M = \langle K,\ T,\ t,\ k_1,\ F\rangle$$

K	is a finite set of states
T	is the finite input alphabet
t	is the partial transition function which, given a state and an input, determines the new state: $K \times T \to K$
k_1	is the initial state ($k_1 \in K$)
F	is the set of final (terminal) states ($F \subseteq K$)

Note that t is partial, in that it is not defined for certain state, input combinations.

Example: The automaton already depicted is thus formally represented by the 5-tuple $\langle K,\ T,\ t,\ S,\ E\rangle$, where:

$$K \stackrel{def}{=} \{S, A, B, F\}$$

$$
\begin{array}{rcl}
T & \stackrel{def}{=} & \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \\
t & \stackrel{def}{=} & \{ \quad (S, a) \to A, \\
 & & \qquad (S, b) \to B, \\
 & & \qquad (A, a) \to A, \\
 & & \qquad (A, c) \to F, \\
 & & \qquad (B, b) \to B, \\
 & & \qquad (B, c) \to F, \\
 & & \qquad (F, c) \to F \quad \} \\
E & \stackrel{def}{=} & \{F\}
\end{array}
$$

We can extend the definition of t to t_* to work on whole strings, rather than individual symbols:

Definition: Given a transition function t (type $K \times T \to K$), we define its string closure t_* (type $K \times T^* \to K$) as follows:

$$
\begin{array}{rcl}
t_*(k,\ \varepsilon) & \stackrel{def}{=} & k \\
t_*(k,\ as) & \stackrel{def}{=} & t_*(t(k, a),\ s) \text{ where } a \in T \text{ and } s \in T^*
\end{array}
$$

We can now easily define the language accepted by a deterministic finite state automaton:

Definition: The set of strings (language) accepted by the deterministic finite state automaton M is denoted by $\mathcal{T}(M)$ and is the set of terminal strings x, which, when M is started from state k_1 with input x, it finishes in one of the final states. Formally it is defined by:

$$\mathcal{T}(M) \stackrel{def}{=} \{x \mid x \in T^*,\ t_*(k_1,\ x) \in F\}$$

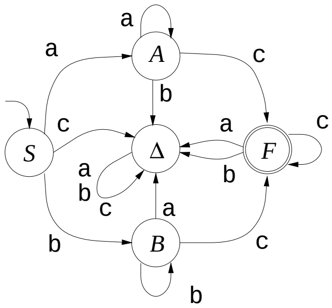
Definition: Two deterministic finite state automata M_1 and M_2 are said to be equivalent if they accept the same language: $\mathcal{T}(M_1) = \mathcal{T}(M_2)$.

Example: In the example automaton M we have been using, we can prove that $\mathcal{T}(M) = \{a^n c^m \mid n, m > 0\} \cup \{b^n c^m \mid n, m > 0\}$.

Proposition: For any deterministic finite state automaton, there is an equivalent one with a total transition function.

Construction: Simply add a new (non-final) state Δ and map any undefined state, input combinations to Δ.

Example: The automaton we have been using for an example can be made total as follows:



Theorem: The class of languages recognizable by deterministic, finite state automata, is no larger than the class of regular languages:

$$\{\mathcal{T}(M) \mid M \text{ is a DFSA }\} \subseteq \mathcal{L}_3$$

Construction: Given a DFSA, we want to generate a regular grammar which recognizes exactly the same language. We first create a non-terminal symbol for each state and set the start state to be the start symbol in the grammar. For every transition $t(A, a) = B$, we add the production $A \to aB$. If B is terminal, we also add $A \to a$. Thus, given a DFSA $\langle K, T, t, k_1, F\rangle$, we claim that $\mathcal{L}(G) = \mathcal{T}(M)$, where G is defined below:

$$\begin{array}{rcl} G & = & \langle T, K, P, k_1\rangle \text{ where} \\ P & = & \begin{array}{ll} [t] & \{A \to aB \mid t(A, a) = B\} \\ \cup & \{A \to \varepsilon \mid A \in F\} \end{array} \end{array}$$

Note that rather than using the second set, most texts use the set $\{A \to a \mid t(A, a) = B \text{ and } B \in F\}$. In this case we would also need to check whether $S \in F$ and if so add the production $S \to \varepsilon$.

Example: The regular grammar recognizing the language of the example DFSA we have been using is thus:

$$G = \langle\{a, b, c\}, \{S, A, B, F\}, P, S\rangle \text{ where}$$

$$\begin{array}{rcl} P & = & \{ \; S \rightarrow aA \mid bB, \\ & & \phantom{\{ \;} A \rightarrow aA \mid cF, \\ & & \phantom{\{ \;} B \rightarrow bB \mid cF, \\ & & \phantom{\{ \;} F \rightarrow cF \mid \varepsilon \end{array}$$

To prove that the class of regular languages $\mathcal{L}_3$ is included in the class of languages recognized by DFSA, we first extend our definition of finite state automata. A non-deterministic finite state automaton (NFSA) is similar to a DFSA except that we allow for multiple similarly labeled outgoing arrows from the same state. t is now no longer a function, but a relation from $K \times N$ to K.

Note that a relation between two sets A and B can be expressed as a function f from A to $\mathbb{P}B$, such that element $a \in A$ is related to $b \in B$ if and only if $b \in f(a)$. This may simplify the presentation of certain proofs considerably.

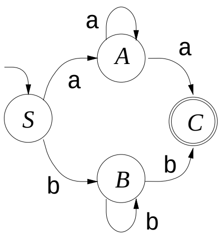
Definition: A non-deterministic finite state automaton M is a 5-tuple $\langle K,\ T,\ t,\ k_1,\ F \rangle$, where:

K	is a finite set of states in which the automaton can reside
T	is the input alphabet
$k_1 \in K$	is the initial state
$F \subseteq K$	is the set of final states

t is a total function from state and input to a set of states. $(T \times K \rightarrow \mathbb{P}T)$.

The set of states $t(A, \mathtt{a})$ defines all possible new states if the machine reads an input $\mathtt{a}$ in state A. If no transition is possible, for such a state, input combination $t(A, \mathtt{a}) = \emptyset$.

Example: Consider the non-deterministic automaton depicted below:



This would formally be encoded as non-deterministic finite state automaton $M = \langle \{A, B, C\},\ \{\mathtt{a}, \mathtt{b}\},\ t,\ S,\ \{C\} \rangle$, where t is:

$$
\begin{aligned}
t \quad = \quad \{ \quad & (S, \mathtt{a}) \rightarrow \{A\}, \\
& (S, \mathtt{b}) \rightarrow \{B\}, \\
& (A, \mathtt{a}) \rightarrow \{A,\ C\}, \\
& (A, \mathtt{b}) \rightarrow \emptyset, \\
& (B, \mathtt{a}) \rightarrow \emptyset, \\
& (B, \mathtt{b}) \rightarrow \{B,\ C\}, \\
& (C, \mathtt{a}) \rightarrow \emptyset, \\
& (C, \mathtt{b}) \rightarrow \emptyset \qquad\qquad \}
\end{aligned}
$$

As before, we would like to extend the definition of t to work on strings. However note that whereas before we defined $t_*(aw, A) = t_*(w, t(a, A))$, the result of t is now not compatible with its input (it takes a single state and outputs a set of states). We first have to define $\bar{t}$, a function from sets of states to sets of states.

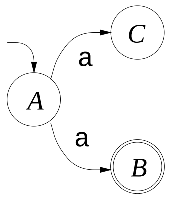
Definition: The extension of a transition function to take sets of states $\bar{t} : \mathbb{P}T \times K \rightarrow \mathbb{P}T$, is defined by:

$$\bar{t}(S, \mathtt{a}) \quad \stackrel{def}{=} \quad \bigcup_{k \in S} t(k, \mathtt{a})$$

Definition: We can now define t_*, the string closure of $\bar{t}$:

$$
\begin{aligned}
t_*(S,\ \varepsilon) \quad & \stackrel{def}{=} \quad S \\
t_*(S,\ as) \quad & \stackrel{def}{=} \quad t_*(\bar{t}(S, a),\ s) \text{ where } a \in T \text{ and } s \in T^*
\end{aligned}
$$

Using t_* we can now define the set of strings accepted by a NFSA. However, there still is an issue to resolve. Consider the following NFSA:



Should we consider string `a` to be accepted by the automaton? Clearly, it may end up in a terminal state (B) and thus, if we consider a string to be accepted if there is at least one path which leads to a terminal state, then `a` is accepted. On the other hand, if the machine is assumed to run through the states once, with no predictability power, then it may reach a non-terminal state (C) and is thus not accepted. The first option is chosen. One reason is that it is closer to how we chose to define acceptance of a string by a grammar (there is at least one derivation which produces the desired string).

With this choice, we can view non-determinism to be doing one of the following:

- The automaton somehow tries all possible paths for a given input and is satisfied if it hits a terminal state at the end of any of the paths.
- At every choice, the machine is replicated, with each machine taking a different choice. If any machine ends up in a terminal state, the input is considered accepted.
- The machine has an 'oracle' or crystal-ball, which makes it take the right choice such that it finally ends up in a terminal state (if at all possible).

These choices are more of an 'explanation' of how such a machine may be seen to be implemented.

Since we chose to accept a string if it is capable of reaching a terminal state, we simply apply t_* on the input string and the initial state, and check whether the set of final states includes at least a terminal one:

Definition: A non-deterministic finite state automaton M is said to accept a terminal string s, if, starting from the initial state with input s, it may reach a final state: $t_*(k_1,\ s) \cap F \neq \emptyset$.

Definition: The language accepted by a non-deterministic finite state automaton M, is the set of all strings (in T^*) accepted by M:

$$\mathcal{T}(M) \quad \stackrel{def}{=} \quad \{s \mid s \in T^*,\ t_*(k_1,\ s) \cap F \neq \emptyset\}$$

NFSA have the extra power necessary to be able to invert the construction of a regular grammar from a DFSA.

Theorem: The class of languages recognizable by non-deterministic finite state automata is at least as large as $\mathcal{L}_3$, the class of regular languages.

$$\mathcal{L}_3 \subseteq \{\mathcal{T}(M) \mid M \text{ is a NFSA }\}$$

Construction: Given a regular language L, there is an ε-free regular grammar G which recognizes it:

$$G = \langle \Sigma, N, P, S \rangle$$

We now create a NFSA M which has a state for every non-terminal in N, plus a new one $\#$ which is a terminal state. S will also be a terminal state if the grammar includes the rule $S \to \varepsilon$. Obviously, M will have input alphabet Σ. The initial state of M is S, and it is allowed to go from state A to B upon input a if $A \to aB$ is a production rule in P. M is also allowed to go from state A to $\#$ upon input a if $A \to a$ is a production in P:

$$M = \langle N \cup \{\#\}, \Sigma, t, S, F \rangle$$

where the transition function t is defined by:

$$\begin{array}{rcl} t(A, a) \stackrel{def}{=} & & \{B \mid A \to aB\} \\ & \cup & \{\# \mid A \to a\} \end{array}$$

The set of final states F is defined as $\{\#, S\}$ if $S \to \varepsilon$ is a production rule and $\{\#\}$ otherwise.

We have thus proved that:

$$\{\mathcal{T}(M) \mid M \text{ is a DFSA }\} \subseteq \mathcal{L}_3 \subseteq \{\mathcal{T}(M) \mid M \text{ is a NFSA }\}$$

If we prove that every language recognizable by NFSA is also recognizable by a DFSA, we have thus proved that NFSA, DFSA and regular grammars are all equally expressive.

Theorem: The class of languages recognizable by NFSA is not larger than that of languages recognizable by DFSA.

$$\{\mathcal{T}(M) \mid M \text{ is a NFSA }\} \subseteq \{\mathcal{T}(M) \mid M \text{ is a DFSA }\}$$

Construction: The idea is to create a new automaton with every combination of states in the original represented in the new automaton by a state. Thus, if the original automaton had states A and B, we now create an automaton with 4 states labeled $\emptyset$, $\{A\}$, $\{B\}$ and $\{A, B\}$. The transitions would then correspond directly to $\bar{t}$. Thus, if $\bar{t}(\{A\}, a) = \{A, B\}$ in the original automaton, we would have a transition labeled a from state $\{A\}$ to state $\{A, B\}$. The equivalence of the languages is then natural.

Given a DFSA $M = \langle K,\ T,\ t,\ k_1,\ F\rangle$, we now define a NFSA $M'' = \langle K',\ T,\ t',\ k_1',\ F'\rangle$, where :

$$\begin{aligned} K' &= \mathbb{P}K \\ t' &= \bar{t} \\ k_1' &= \{k_1\} \\ F' &= \{S \mid S \subseteq K,\ S \cap F \neq \emptyset\} \end{aligned}$$

Theorem: The following statements are equivalent:

1. L is a regular language ($L \in \mathcal{L}_3$).
2. L is accepted by a non-deterministic finite state automaton.
3. L is accepted by a deterministic finite state automaton.

A number of conclusions and observations follow from this theorem.

- If, for a given total DFSA M, we construct a similar DFSA M', but where a state is terminal in M' if and only if it is not in M, we have constructed a DFSA which recognizes exactly the complement of the language recognized by M: $\mathcal{T}(M') = \overline{\mathcal{T}(M)}$. (prove!).

  Thanks to the last theorem, regular languages are thus also closed under set complement.

- Notice that $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Since regular languages are closed under set complement and union, they are also closed under set intersection.

- Notice that in a DFSA we only had to perform an amount of work linear with respect to the length of the string to decide whether it is recognized or not. In other words, an algorithm recognizing whether $w \in L$ can be constructed such that it is of order $O(n)$ (where $n = |w|$).

- Now consider the same problem but on a NFSA. If we have a non-deterministic computer with an in-built, infallible oracle, the problem is obviously still of order $O(n)$. What about using a conventional machine? In the worst case of full branching, we have to look down a tree of depth $|w|$ and width $|K|$. If we are really unlucky, we may have to look through $|K|^{|w|}$ possible paths. This is of exponential order, which, as you may already have heard from other courses, is not very efficient (to put it mildly).

- The last two statements, together with the equivalence of DFSA and NFSA) seem to indicate some form of contradiction. However, we simply note that when translating a NFSA into a DFSA, we bypass the exponential time problem but generate an exponential size problem. You may have noticed that, given a NFSA with n states, the equivalent DFSA resulting from our construction has 2^n states (since $|\mathbb{P}S| = 2^{|S|}$).

  This is the nature of the last part of this course which deals with algorithmic complexity.

3.5 The Pumping Lemma for Regular Languages

There is still one property which we would like to prove about regular languages — that they are a proper subset of context free grammars. In other words, there are type 2 languages which are not also of type 3.

We start by giving an informal argument on a language to prove that no finite state automaton can recognize. Since type 3 grammars are not more expressive than finite state automata, this means that this language is not of type 3. To conclude the argument, we then give a type 2 grammar which recognizes this language.

Proposition: No deterministic finite state automaton can recognize exactly the following language:

$$L = \{a^n b^n \mid n > 0\}$$

Argument: Let $M = \langle K, T, t, S, F\rangle$ be a total DFSA such that $\mathcal{T}(M) = L$.

Consider the set of strings $\{a^i \mid n > 0\}$. Clearly this set is infinite. Now, $t_*(S, a^i) \in K$, where K is the *finite* set of states of M. Hence, there must be some different strings a^i and a^j such that $t_*(S, a^i) = t_*(S, a^j)$ despite that $i \neq j$.

Now consider $w = a^i b^i$. Since $w \in L$, we know that $t_*(S, w) \in F$. Hence, using a property of t_*, we get: $t_*(t_*(S, a^i), b^i) \in F$. But $t_*(S, a^i) = t_*(S, a^j)$ and thus:

$$t_*(t_*(S, a^j), b^i) \in F$$

This implies that $t_*(S, a^j b^i) \in F$ and thus $a^j b^i \in L$ (with $i \neq j$). This contradicts the definition of L, and hence no total DFSA with the desired property can exist. Also, since for any partial DFSA we can construct an equivalent total DFSA, we conclude that no DFSA recognizing L can exist.

$\square$

What about a type 2 grammar which recognizes this language? The solution is quite simple:

$$G = \langle \{a, b\}, \{S\}, \{S \to ab \mid aSb\}, S\rangle$$

This is enough to show the desired property that regular languages are limited in their expressibility and that there are type 2 languages which are not also of type 3. However, we would like to analyze this result slightly further and generalize it before going on to context free grammars.

The lack of memory in finite state automata is the limiting factor of type 3 languages. The only information they can store is the current state, and since the number of states is finite it can only remember at most one of $|K|$ distinct pieces of information (where K is the set of states of the automaton). Obviously this does not mean that type 3 languages only include finite languages, but that any infinite language must necessarily use cycles in the related DFSA. This is the result given by the pumping lemma.

Theorem: The Pumping Lemma: If L is a regular language, then there is a number n (called the pumping length) where, for any string w in L at least n symbols long, then w can be split into 3 pieces $w = xyz$ such that:

- $|y| > 0$
- $|xy| \leq n$
- for any $i \in \mathbb{N}$, $xy^iz \in L$

At first sight it may seem quite confusing and probably even intimidating. However, if we go through it step by step, we find that it is not so difficult to follow.

If we are given a regular language L, we know, by the previous informal discussion, that beyond a certain length, the automaton must start to cycle. Thus, any such language L must have an associated constant n, such that strings longer than n must include a cycle. Now, this means that any string of this length (or longer) can be split into 3 parts xyz. x takes the DFSA acceptor to the beginning of a cycle for the first time, y is the cycle carried out and z then takes us to a terminal state. However, y can be repeated any number of times (including zero) and thus, all strings of the form xy^iz must be in L.

The pumping lemma *cannot* be used to prove that a language is regular, but rather that it is not. Such proofs take the following form:

1. Assume that L is regular.
2. Since L is regular, we are guaranteed that there is a constant n — the pumping length of L.
3. We then take a string w in L at least n symbols long, and ...
4. Take a general decomposition of w into xyz such that $|y| > 0$ and $|xy| \leq n$.
5. We prove that for some number i, xy^iz is not in L.
6. Hence we have reached a contradiction and thus the assumption in 1 must be incorrect.

The choice of n and the splitting of w into xyz is not within our control, which makes these arguments look like a game:

- We play a language L.
- The opponent plays a number n.
- We play a string of length at least n in L.
- Our opponent gives us back the string decomposed into 3 parts x, y and z, satisfying the desired properties.
- We play back a number i such that $xy^iz \notin L$.

The following is a proof that $L = \{a^ib^i \mid i \in \mathbb{N}\}$ is not regular using the pumping lemma:

Proposition: $L = \{a^ib^i \mid i \in \mathbb{N}\}$ is not regular.

Proof:

- Assume L is regular.
- Then, by the pumping lemma, there exists a pumping length n.
- Consider string a^nb^n. Clearly $|a^nb^n| \geq n$ and $a^nb^n \in L$.
- Take x, y and z satisfying $xyz = a^nb^n$, where $|y| > 0$ and $|xy| \leq n$.
- Since $|xy| \leq n$, $xy = a^i$ and $z = a^{n-i}b^n$. Also $|y| > 0$ implies that $y = a^j$ with $j > 0$.
- But $xy^0z = a^{i-j}a^{n-i}b^n = a^{n-j}b^n$ which is not in L (since $j > 0$). But by the pumping lemma, $xy^0z \in L$.
- Hence, L cannot be regular.

$\square$

Proposition: $L = \{a^ib^j \mid i < j\}$ is not regular.

Proof:

- Assume L is regular.
- Then, by the pumping lemma, there exists a pumping length n.
- Consider string $a^n b^{n+1}$. Clearly $|a^n b^{n+1}| \geq n$ and $a^n b^{n+1} \in L$.
- Take x, y and z satisfying $xyz = a^n b^{n+1}$, where $|y| > 0$ and $|xy| \leq n$.
- Since $|xy| \leq n$, $xy = a^i$ and $z = a^{n-i} b^{n+1}$. Also $|y| > 0$ implies that $y = a^j$ with $j > 0$.
- But $xy^2 z = a^i a^j a^{n-i} b^{n+1} = a^{n+j} b^{n+1}$ which is not in L (since $j > 0$ and thus $n + j \geq n + 1$). But by the pumping lemma, $xy^2 z \in L$.
- Hence, L cannot be regular.

$\square$

Proposition: The language which includes exactly all palindromes over $\{a, b\}$, $L = \{ww^R \mid w \in \{a, b\}^*\}$ is not regular.

Proof:

- Assume L is regular.
- Then, by the pumping lemma, there exists a pumping length n.
- Consider string $a^n b a^n$. Clearly $|a^n b a^n| \geq n$ and is in L.
- Take x, y and z satisfying $xyz = a^n b a^n$, where $|y| > 0$ and $|xy| \leq n$.
- Since $|xy| \leq n$, $xy = a^i$ and $z = a^{n-i} b a^n$. Also $|y| > 0$ implies that $y = a^j$ with $j > 0$.
- But $xy^2 z = a^i a^j a^{n-i} b a^n = a^{n+j} b a^n$ which is not in L (since $j > 0$). But by the pumping lemma, $xy^2 z \in L$.
- Hence, L cannot be regular.

$\square$

Proposition: The language over $\{a\}$ which includes exactly all strings of length n, where n is a prime number is not regular.

Proof:

- Assume L is regular.
- Then, by the pumping lemma, there exists a pumping length n.
- Consider string a^p where p is the first prime larger than $n + 1$. Since the number of prime numbers is infinite, such a number exists.
- Take x, y and z satisfying $xyz = a^p$, where $|y| > 0$ and $|xy| \leq n$. Since $p > n + 1$, $|z| > 1$.
- Now consider $w = xy^{|xz|}z$. Clearly, $|w| = |x| + |xz||y| + |z|$ which can be simplified to $|xz|(|y| + 1)$. Since both values are larger than 1, $|w|$ is composite. But by the pumping lemma, $w \in L$.
- Hence, L cannot be regular.

$\square$

Obviously, we have only proved that these languages are not regular. To show that they are of type 2, we would need to find a type 2 grammar which recognizes them.

Finally, let us prove the pumping lemma for regular languages. We need a mathematical principle called the 'pigeon hole principle' to complete the proof. It states that if we have n objects to put in m holes, where $n > m$, one of the holes must end up with more than one object. More formally, we write that if $a_i \in A$ for $1 \leq i \leq n$ and $|A| < n$, then there are at least two distinct i and j such that $a_i \neq a_j$.

The Pumping Lemma: If L is a regular language, then there is a number n (called the pumping length) where, for any string w in L at least n symbols long, then w can be split into 3 pieces $w = xyz$ such that:

- $|y| > 0$
- $|xy| \leq n$
- for any $i \in \mathbb{N}$, $xy^iz \in L$

Proof: Let $M = \langle K, T, t, q_1, F\rangle$ be a DFSA such that $\mathcal{T}(M) = L$. Let n be defined to be the number of states $|K|$.

Consider a string $w \in L$ such that $|w| > n$. Let $w = a_1 a_2 \dots a_m$. Now consider the sequence of states that M passes through while parsing w: $k_1, k_2 \dots k_{m+1}$, where $k_{i+1} = t_*(k_1, a_1 a_2 \dots a_i)$. Since $w \in L$, $k_{m+1} \in F$.

Since $m+1 > m > n$, we have more states in the sequence traveled through than the number of states n. Consider the first $n+1$ states. By the pigeon-hole principle, there are two equal states: $k_i = k_j$ for some $i < j \leq n+1$.

Now, decompose w into xyz as follows:

$$\begin{aligned} x &\stackrel{def}{=} a_1 \dots a_{i-1} \\ y &\stackrel{def}{=} a_i \dots a_{j-1} \\ z &\stackrel{def}{=} a_j \dots a_m \end{aligned}$$

Since $i \neq j$, we can conclude that $|y| > 0$. Also, since $j \leq n+1$, $|xy| \leq n$. This gives the desired bounds on the decomposition.

By the definition of the sequence of states:

$$\begin{aligned} k_i &= t_*(k_1, x) \\ k_j &= t_*(k_i, y) \\ k_{m+1} &= t_*(k_j, z) \end{aligned}$$

We can easily prove that for all $r \geq 0$, $t_*(k_i, y^r) = k_i$ by induction on r.

Hence, for all $r \geq 0$:

$$\begin{aligned} & t_*(k_1, xy^r z) \\ = \; & t_*(t_*(k_1, x), y^r z) \\ = \; & t_*(k_j, y^r z) \\ = \; & t_*(k_i, y^r z) \\ = \; & t_*(k_i, z) \\ = \; & k_{m+1} \in F \end{aligned}$$

$\square$

3.6 Exercises:

1. Prove that all finite languages are regular.
2. Would NFSA be any more expressive if they were to start from one of an arbitrary number of states? Formalize this extension and prove your statement.
3. Is $\mathcal{L}_3$ closed under set difference?
4. Given a regular language L, prove that its reverse L^R is also a regular language.

   Hint: Use the fact that there is a NFSA which recognizes L.
5. Given a left-linear regular grammar which recognizes a language L, construct a right-linear grammar which recognizes L^R. Outline a proof. Argue why the construction may also be used in the other direction.
6. Use the previous two proofs to conclude that the class of languages generated by left-linear grammars is equivalent to the class of languages generated by right-linear grammars.
7. Is the class of languages recognizable by grammars which use a mixture of left and right linear productions still $\mathcal{L}_3$?
8. Is the language over $\{a, b\}$ in which all strings have an equal number of occurrences of a and b, regular?
9. Is the set of strings a^n, where composite numbers (numbers which can be decomposed into the product of two numbers both larger than 1) a regular language?
10. Prove that $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a regular language. Can you find a context free grammar to generate the language?
11. Is the language over $\{a, b\}$ in which all strings have an equal number of occurrences of ab and ba, regular?

Chapter 4

Context Free Languages

4.1 Definition

We no switch our attention to the next class of languages — type 2, or context free languages. Recall the definition of this class of languages:

Definition: A phrase structure grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$ is said to be a *context free grammar* if all the productions in P are in the form: $A \to \alpha$, where $A \in N$ and $\alpha \in (\Sigma \cup N)^*$.

Definition: A language L is said to be a *context free language* (or type 2 language) if there is a context free grammar G such that $\mathcal{L}(G) = L$.

4.2 Closure

As with type 3 languages, type 2 languages are closed under set union, catenation and Kleene closure.

The constructions are simpler than those for regular grammars, since the restrictions placed on the production rules are less pedantic. At the same time we still have strong enough restrictions which make certain techniques usable.

Theorem: $\mathcal{L}_2$ is closed under set union.

Construction: Given two type 2 languages, L_1 and L_2, there are two context free grammars G_1 and G_2 such that $\mathcal{L}(G_i) = L_i$. Let $G_i = \langle \Sigma_i, N_i, P_i, S_i \rangle$. Assume that $N_1 \cap N_2 = \emptyset$.

We claim that grammar G defined as follows generates $L_1 \cup L_2$. Let S be a new non-terminal symbol not in N_1 or N_2.

$$G = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S\}, P, S \rangle$$

where $P = P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}$.

Note that production must start from non-terminal symbols. If productions from terminal symbols were allowed, this construction would not work.

Theorem: $\mathcal{L}_2$ is closed under catenation.

Construction: Given two type 2 languages, L_1 and L_2, there are two context free grammars G_1 and G_2 such that $\mathcal{L}(G_i) = L_i$. Let $G_i = \langle \Sigma_i, N_i, P_i, S_i \rangle$. Assume that $N_1 \cap N_2 = \emptyset$.

We now construct grammar G to generate $L_1 L_2$. Let S be a new non-terminal symbol not in N_1 or N_2.

$$G = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S\}, P, S \rangle$$

where $P = P_1 \cup P_2 \cup \{S \to S_1 S_2\}$.

Again note that this is possible since we can only have non-terminals on the left hand side of productions. Otherwise, problems could have arisen where terminal symbols produced by S_1 could have activated rules from P_2 and vice-versa.

Theorem: $\mathcal{L}_2$ is closed under Kleene Closure.

Construction: Given a type language, L, there is a context free grammar G, such that $\mathcal{L}(G) = L$. Let $G = \langle \Sigma, N, P, S \rangle$.

We now construct grammar G' to generate L^*.

$$G = \langle \Sigma, N, P \cup \{S \to SS\}, S \rangle$$

As before, this construct is dependent on the format of allowed rules in type 2 grammars.

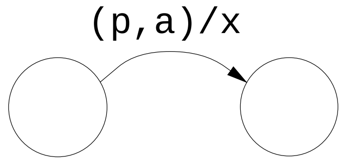
4.3 Recognition

The main problem with finite state automata, is that they had no way of 'remembering'. Pushdown automata are basically an extension of finite state automata, with an allowance for memory. The extra information is carried around in the form of a stack.

Recall that a stack is a last-in first-out data structure which has two defined operations *push* (adds a new object on the top of the stack) and *pop* (which retrieves the object on the top of the stack, and also returns a modified stack with the top object removed). We will use strings to represent stacks. ε is obviously the empty stack, whereas any string over Σ is a possible configuration of a stack which stores objects in Σ. The stack functions are defined by:

$$
\begin{aligned}
push(s, a) &\stackrel{def}{=} sa \\
pop(as) &\stackrel{def}{=} (a, s)
\end{aligned}
$$

Upon initialization, the stack always starts off with a particular value being pushed (normally used to act as a marker that the stack is about to empty). Every transition now depends, not only on the input, but also on the value on the top of the stack. Upon performing a transition, a number of values may also be pushed on the stack. In diagrams, transitions are now marked as shown in the figure below:

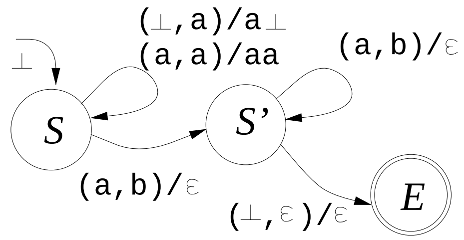


The label $(\mathtt{p}, \mathtt{a})/x$ is interpreted as follows: pop a value off the stack, if the symbol just popped is `p` and the input is `a`, then perform the transition and push onto the stack the string x. If the transition cannot take place, try another using the value just popped. The machine 'crashes' if it tries to pop a value off an empty stack.

The arrow marking the initial state is marked by a symbol which is pushed onto the stack at initialization.

As in the case of FSA, we also have a number of final states which are used to determine which strings are accepted. A string is accepted if, starting the machine from the initial state, it terminates in one of the final states.

Example: Consider the PDA below:



Initially, the value $\perp$ is pushed onto the stack. This will be called the 'bottom of stack marker'. While we are at the initial state S, read an input. If it is a push back to the stack whatever has just been popped, together with an extra a. In other words, the stack now contains as many as as have been accepted.

When a b is read, the state now changes to S'. The top value of the stack (a) is also discarded. While in S' the PDA continues accepting bs as long as there are as on the stack. When finally, the bottom of stack marker is encountered, the machine goes to the final state E.

Hence, the machine is accepting all strings in the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$. Note that this is also the language of the context free grammar with production rules: $S \rightarrow \mathtt{ab} \mid \mathtt{a}S\mathtt{b}$.

Recall that it was earlier stated that this language cannot be accepted by a regular grammar. This seems to indicate that we are on the right track and that pushdown automata accept certain languages for which no finite state automaton can be constructed to accept.

4.4 Non-deterministic Pushdown Automata

Definition: A non-deterministic pushdown automaton M is a 7-tuple:

$$M \stackrel{def}{=} \langle K, T, V, P, k_1, A_1, F \rangle$$

$$
\begin{array}{lcl}
K & = & \text{a finite set of states} \\
T & = & \text{the input alphabet of } M \\
V & = & \text{the stack alphabet} \\
P & = & \text{transition function of } M \text{ with type:} \\
 & & (K \times V \times (T \cup \{\varepsilon\}) \to \mathbb{P}(K \times V^*) \\
k_1 & \in & K \text{ is the start state} \\
A_1 & \in & V \text{ is the initial symbol placed on the stack} \\
F & \subseteq & K \text{ is the set of final states}
\end{array}
$$

Of these, the production rules (P) may need a little more explaining. P is a total function such that $P(k, v, i) = \{(k'_1, s_1), \ldots (k'_n, s_n)\}$, means that: in state k, with input i (possibly ε meaning that the input tape is not inspected) and with symbol v on top of the stack (which is popped away), M can evolve to any of a number of states k'_1 to k'_n. Upon transition to k'_i, the string s_i is pushed onto the stack.

Example: The PDA already depicted is formally expressed by:

$$M = \langle \{S, S', E\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \perp\}, P, S, \perp, \{E\} \rangle$$

where P is defined as follows:

$$
\begin{array}{rcl}
P(S, \perp, \mathtt{a}) & = & \{(S, \mathtt{a}\perp)\} \\
P(S, \mathtt{a}, \mathtt{a}) & = & \{(S, \mathtt{aa})\} \\
P(S, \mathtt{a}, \mathtt{b}) & = & \{(S', \varepsilon)\} \\
P(S', \mathtt{a}, \mathtt{b}) & = & \{(S', \varepsilon)\} \\
P(S, \perp, \varepsilon) & = & \{(E, \varepsilon)\} \\
P(X, x, y) & = & \emptyset \text{ otherwise}
\end{array}
$$

Note that this NPDA has no non-deterministic transitions, since every state, input, stack value triple has at most one possible transition.

Whereas before, the current state described all the information needed about the machine to be able to deduce what it can do, we now also need the current state of the stack. This information is called the *configuration* of the machine.

Definition: The set of configurations of M is the set of all pairs (state, string): $K \times V^*$.

We would now like to define which configurations are reachable from which configurations. If there is a final state in the set of configurations reachable from (k_1, A_1) with input s, s would then be a string accepted by M.

Definition: The transition function of a PDA M, written t_M, is a function which, given a configuration and an input symbol, returns the set of states in which the machine may terminate with that input.

$$
\begin{aligned}
t_M \quad &: \quad (K \times V^*) \times (T \cup \{\varepsilon\}) \to \mathbb{P}(K \times V^*) \\
t_M((k, xX), a) \quad &\stackrel{def}{=} \quad \{(k', yX) \mid (k', y) \in P(k, x, a)\}
\end{aligned}
$$

Definition: The string closure transition function of a PDA M, written t^*_M is the function, which given an initial configuration and input string, returns the set of configurations in which the automaton may end up in after using up the input string.

$$
t^*_M \quad : \quad (K \times V^*) \times (T \cup \{\varepsilon\}) \to \mathbb{P}(K \times V^*)
$$

We start by defining the function for an empty stack:

$$
\begin{aligned}
t^*_M((k, \varepsilon), \varepsilon) \quad &\stackrel{def}{=} \quad \{(k, \varepsilon)\} \\
t^*_M((k, \varepsilon), a) \quad &\stackrel{def}{=} \quad \emptyset
\end{aligned}
$$

For a non-empty stack, and input $a_1 \dots a_n$, (where each $a_i \in T \cup \{\varepsilon\}$, we need to find

$$
\begin{aligned}
t^*_M((k, X), a) \quad \stackrel{def}{=} \quad & \{(k', X') \mid \\
& \exists a_1, \dots a_n : T \cup \{\varepsilon\} \cdot a = a_1 \dots a_n \\
& \quad \exists c_1, \dots c_n : K \times V^* \cdot \\
& \quad\quad c_1 \in t_M((k, X), a_1) \\
& \quad\quad c_{i+1} \in t_M(c_i, a_i) \text{ for all } i \\
& \quad\quad (k', X') \in t_M(c_n, a_n)
\end{aligned}
$$

We say that $c' = (k', X')$ is reachable from $c = (k, X)$ by a if $c' \in t^*_M(c, a)$.

Definition: The *language* recognized by a non-deterministic pushdown automaton $M = \langle K, T, V, P, k_1, A_1, F \rangle$, written as $\mathcal{T}(M)$ is defined by:

$$\mathcal{T}(M) \quad \stackrel{def}{=} \quad \{x \mid x \in T^*,\ (F \times V^*) \cap t^*_M((k_1, A_1), x) \neq \emptyset\}$$

Informally, there is at least one configuration with a final state reachable from the initial configuration with x.

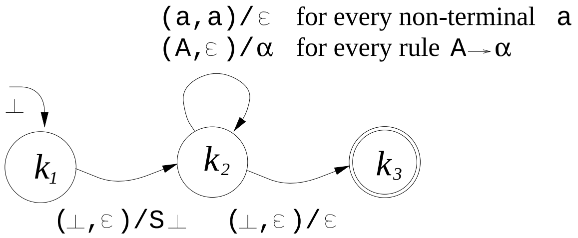
Theorem: For every context free language L there is a non-deterministic pushdown automaton M such that $\mathcal{T}(M) = L$.

Proof Outline: Consider the acceptance of a string in a type 2 grammar. If we always perform productions on the leftmost non-terminal, we can follow the following procedure:

1. Start with the string S (the start symbol).
2. Get the leftmost symbol.
3. Is it a terminal? If so the input must start with that symbol. Otherwise use a production rule to open up the non-terminal, and perform the replacement in the current string.
4. If the current string is empty and the input has also finished, then the input has been accepted.
5. Otherwise, jump back to 2.

This strategy is possible since none of the rules depend on the terminal symbols already read.

To construct a NPDA which performs the above procedure, we will simply keep the current string on the stack. If a terminal symbol is on top try to match it with the input, otherwise use any of the relevant production rules to open it up. The whole automaton will be made up of just 3 states, as shown in the diagram below:



Let $G = \langle \Sigma, N, P, S \rangle$ be a context free grammar such that $\mathcal{L}(G) = L$.

We now construct a NPDA:

$$
\begin{aligned}
M \quad = \langle \quad & \{k_1, k_2, k_3\}, \\
& \Sigma, \\
& \Sigma \cup N \cup \{\perp\}, \\
& P_M, \\
& k_1, \\
& \perp, \\
& \{k_3\} \rangle
\end{aligned}
$$

where P_M is:

- From the initial state k_1 we push S and go to k_2 immediately.

$$P((k_1, \perp), \varepsilon) \quad = \quad (k_2, S\perp)$$

- Once we reach the bottom of stack marker in k_2 we can terminate.

$$P((k_2, \perp), \varepsilon) \quad = \quad (k_3, \varepsilon)$$

- For every non-terminal A in N with at least one production rule in P of the form $A \to \alpha$ we add the rule:

$$P((k_2, A), \varepsilon) \quad = \quad \{(k_2, \alpha) \mid A \to \alpha \in P\}$$

- For every terminal symbol a in Σ we also add:

$$P((k_2, a), a) \quad = \quad \{(k_2, \varepsilon)\}$$

We can prove that $\mathcal{T}(M) = L$.

We would now like to prove the inverse: that languages produced by NPDA are all context free languages. Before we prove the result, we prove a lemma which we will use in the proof.

Lemma: For every NPDA M, there is an equivalent NPDA M' such that:

1. M' has only one final state
2. the final state is the only one where the state can be clear
3. the automaton clears the stack upon termination

Strategy: The idea is to add a new bottom of stack marker which is pushed onto the stack before M starts. Whenever M terminates we go to a new state which can clear the stack.

Proof: Let $M = \langle K, T, V, P, k_1, A_1, F\rangle$.

Now construct $M' = \langle K', T, V', P', k_1', \perp, F'\rangle$:

$$\begin{array}{rcl}
K' & = & K \cup \{k_1', k_2'\} \\
V' & = & K \cup \{\perp\} \\
P' & = & \quad P \\
 & & \cup \;\; \{((k_1', \perp), \varepsilon) \rightarrow \{(k_1, A_1\perp)\} \\
 & & \cup \;\; \{((k, A), \varepsilon) \rightarrow \{(k_2', \varepsilon)\} \mid k \in F, A \in V'\} \\
 & & \cup \;\; \{((k_2', A), \varepsilon) \rightarrow \{(k_2', \varepsilon)\} \mid A \in V'\} \\
F' & = & \{k_2'\}
\end{array}$$

Note that throughout the execution of M, the stack can never be empty, since there will be at least $\perp$ left on the stack.

The proof will not be taken any further here. Check in the textbooks if you are interested in how the proof is then completed.

□

Theorem: For any NPDA M, $\mathcal{T}(M)$ is a context free language.

Strategy: The idea is to construct a grammar, where, for each $A \in V$ (the stack alphabet), we have a family of non-terminals A^{ij}. Each A^{ij} generates the input strings which take M from state k_i to k_j, and remove A from the top of the stack. A_1^{1n} is the non-terminal which takes M from k_1 to k_n removing A_1 off the stack, which is exactly what is desired.

Assume that M satisfies the conditions of the previous lemma (otherwise obtain a NPDA equivalent to M with these properties, as described earlier). Label the states k_1 to k_n, where k_1 is the initial state, and k_n is the (only) final one. Then:

$$M = \langle \{k_1, \ldots k_n\}, T, V, P, k_1, A_1, \{k_n\} \rangle$$

We now construct a context-free grammar G such that $\mathcal{L}(G) = \mathcal{T}(M)$.

Let $G = \langle \Sigma, N, R, S \rangle$.

$$\begin{array}{rcl}
N & = & \{A^{ij} \mid A \in V,\ 1 \leq i, j \leq n\} \\
S & = & A_1^{1n} \\
R & = & \{A^{in_m} \rightarrow a B_1^{jn_1} B_2^{n_1 n_2} \ldots B_m^{n_{m-1} n_m} \mid \\
& & \quad (k_j, B_1 B_2 \ldots B_m) \in t((k_i, A), a),\ 1 \leq n_1, n_2, \ldots n_m \leq n\} \cup \\
& & \{A^{ij} \rightarrow a \mid (k_j, \varepsilon) \in t((k_i, A), a)\}
\end{array}$$

The construction of R is best described as follows: Whenever M can evolve from state k_i (with A on the stack) and input a to k_j (with $B_1 \ldots B_m$ on the stack), we add rules of the form:

$$A^{in_m} \rightarrow a B_1^{jn_1} B_2^{n_1 n_2} \ldots B_m^{n_{m-1} n_m}$$

(with arbitrary n_1 to n_m). This says that M may evolve from k_i to k_{n_m} by reading a and thus evolving to k_j with $B_1 \ldots B_m$ on the stack,m after which it is allowed to travel around the states as long as it removes all the Bs and ends in state k_m.

For every transition from k_i to k_j (removing A off the stack) and with input a, we also add the production rule $A^{ij} \to a$.

If you do not remember this construction from the first year course, I suggest you look it up and try it out on a couple of examples.

Theorem: The class of languages recognizable by NPDA and the class $\mathcal{L}_2$ are equivalent.

This result concludes this section. Clearly, we see a jump between the automata used to recognize regular languages and those used to recognize context free languages.

4.5 The Pumping Lemma

We have proved all the desired properties of context free grammars. However, we would like to confirm our suspicion that context free grammar have certain limitations which can be overcome by going one step further up in the Chomsky hierarchy of languages.

The technique we use is similar to the one used for regular languages. We prove a 'pumping lemma' which, as in the previous case, says that for each context free language, there is a special value, for which any string at least as long as this value in the language, can be 'pumped' to produce more strings in the language. The concept of pumping in this case is more involved, and we split the string into five parts, the second and fourth of which can be pumped together.

Theorem: The pumping lemma for context free languages: For a context free language L, there is a a number n, called the pumping length, which, if $w \in L$ and $|w| \geq n$ then there is a decomposition of $w = uvxyz$ such that:

- $vy \neq \varepsilon$
- $|vxy| \leq n$
- For any $i \geq 0$, $uv^ixy^iz \in L$

As before, we use this lemma to prove that a language is not a context free one. Below are a couple of examples demonstrating this technique.

Proposition: $L = \{a^i b^i c^i \mid i \in \mathbb{N}\}$ is not in $\mathcal{L}_2$.

Proof:

- Assume L is a context free language (in $\mathcal{L}_2$).
- By the pumping lemma, there is a pumping length for L, namely n.
- Now consider the string $w = a^n b^n c^n$. Clearly, $w \in L$ and $|w| \geq n$.
- Now decompose the string w into five parts $uvxyz$ satisfying the conditions placed by the pumping lemma, namely $|vxy| \leq n$ (hence vxy can include at most two distinct symbols) and $vy \neq \varepsilon$.
- Now, by the pumping lemma, $uv^2xy^2z \in L$. But, since vxy can include at most two distinct symbols, we have increased the length of at most two symbols. Also, at least one of v and y must be non-empty, and thus we have effectively increased the number of one or two of the symbols in w. Hence the number of as, bs and cs in uv^2xy^2z cannot be equal.
- Hence L cannot be of type 2.

$\square$

Proposition: $L = \{a^i \mid i \text{ is prime }\}$ is not in $\mathcal{L}_2$.

Proof:

- Assume L is a context free language (in $\mathcal{L}_2$).
- By the pumping lemma, there is a pumping length for L, namely n.
- Now consider the string $w = a^p$, where p is the first prime larger than $n + 1$. Clearly, $w \in L$ and $|w| \geq n$. Also, thanks to the infinitude of primes, such a p must exist.
- Now decompose the string w into five parts $uvxyz$ satisfying the conditions placed by the pumping lemma, namely $|vxy| \leq n$ (hence $|uz| > 1$ and therefore $|uxz| > 1$) and $vy \neq \varepsilon$.

- Now, by the pumping lemma, $uv^m xy^m z \in L$. But, since $|uv^m xy^m z| = |uxz| + m|vy|$. If we take $m = |uxz|$, we get that $|uv^m xy^m z| = |uxz|(1 + |vy|)$. But $|uxz| > 1$ and $|vy| > 0$. Hence, we have proved that there is a string $a^c \in L$ where c is a composite number.

- Hence L cannot be of type 2.

□

Therefore, there are languages which are not context free. But are any of these of type 1. In other words, can we show that $\mathcal{L}_1 \subset \mathcal{L}_2$?

Proposition: $\mathcal{L}_1 \subset \mathcal{L}_2$

Proof: It has already been shown that the language $L = \{a^i b^i c^i \mid i \in \mathbb{N}\}$ is not of type 2. We now show that there is a type 1 grammar which generates L.

Consider $G = \langle \{a, b, c\}, \{S, B, C\}, P, S \rangle$ where P is the set of productions shown below:

$$
\begin{array}{rcl}
S & \rightarrow & aSBC \mid \varepsilon \\
CB & \rightarrow & BC \\
cB & \rightarrow & Bc \\
aB & \rightarrow & ab \\
bB & \rightarrow & bb \\
bC & \rightarrow & bc \\
cC & \rightarrow & cc
\end{array}
$$

Clearly, the grammar is of type 1. It can also be proved that $\mathcal{L}(G) = L$. (Try using the fact that every sentential form is $a^n S\beta$ or $a^n\beta$, where the count of b and B in β is n and similarly the count of c and C. Furthermore, $\beta = b^m\gamma$, where γ has no occurrences of b).

Thus, $\mathcal{L}_1 \not\subseteq \mathcal{L}_2$ which implies that $\mathcal{L}_1 \neq \mathcal{L}_2$. But $\mathcal{L}_2 \subseteq \mathcal{L}_1$ and therefore $\mathcal{L}_2 \subset \mathcal{L}_1$.

□

We conclude by proving the pumping lemma.

The pumping lemma: For a context free language L, there is a number n, called the pumping length, which, if $w \in L$ and $|w| \geq n$ then there is a decomposition of $w = uvxyz$ such that:

- $vy \neq \varepsilon$
- $|vxy| \leq n$
- For any $i \geq 0$, $uv^ixy^iz \in L$

Proof: The proof is based on the idea that for long enough strings, there is a derivation from S to uRz to $uvRyz$ to $uvxyz$. Hence, $R \stackrel{*}{\Rightarrow}_G v^iRy^i$ for any $i \in \mathbb{N}$. Hence $S \stackrel{+}{\Rightarrow}_G uRz \stackrel{*}{\Rightarrow}_G uv^iRy^iz \stackrel{+}{\Rightarrow}_G uv^ixy^iz$. This will complete the proof. More formally ...

Consider $L \in \mathcal{L}_2$. Let $G = \langle \Sigma, N, P, S \rangle$ be a type 2 grammar such that $\mathcal{L}(G) = L$. G exists by definition of type 2 languages. Let b be the maximum number of symbols on the right hand side of production rules (assumed to be at least 2).

Clearly, in the parse tree using this grammar, at any node we have at most b children. Thus, for a derivation of length h, the maximum length of the string generated is b^h.

Let the pumping length for this language be $n = b^{|N|+2}$. Consider a string $w \in L$ such that $|w| \geq n$. Since $b > 1$, we know that $n > b^{|N|+1}$, and thus we need a parse tree of depth $|N| + 2$ or more to generate w.

Let T be the minimal parse tree for w (the one with the smallest number of nodes, if the grammar is ambiguous and it has more than one derivation tree). Since the longest path in T is at least $|N| + 2$ (the minimum depth), and every non-leaf node is labeled by a non-terminal symbol, we have $|N| + 1$ non-terminal symbols along this path. By the pigeon-hole principle, at least one non-terminal symbol must repeat. Let R be the symbol which is repeated lowest in the derivation tree.

Now, we have a derivation $S \stackrel{*}{\Rightarrow}_G uRz$ (the first occurrence of R) and $R \stackrel{+}{\Rightarrow}_G vRy$ (the second occurrence of R) and finally $R \stackrel{+}{\Rightarrow}_G x$. Note that we are taking the lowest two occurrence of R in the tree (if more than one repetition is present).

Clearly, by induction on i, we can prove that for any $i \geq 0$, $R \stackrel{*}{\Rightarrow}_G v^i R y^i$. Hence, $S \stackrel{*}{\Rightarrow}_G uRz \stackrel{*}{\Rightarrow}_G uv^i R y^i z \stackrel{+}{\Rightarrow}_G uv^i x y^i z$. This implies that for any $i \geq 0$, $uv^i x y^i z \in L$ satisfying the last condition of the theorem.

What about the first two conditions? For $vy \neq \varepsilon$, we require that at least one of v and y is non-empty. If both are empty, the part of derivation tree giving $R \stackrel{+}{\Rightarrow}_G vRy$ can be left out, thus giving a smaller derivation tree for w, which contradicts the assumption.

What about $|vxy| \leq n$? If $|vxy| > n$, by the argument we have already gone through, there must be a repeating occurrence of a non-terminal symbol. But we have taken the lowest repeating non-terminal which makes this impossible. Hence this condition is also satisfied.

□

4.6 Exercises

1. It was noted in the constructions to show closure of type 2 languages, that the given constructions worked only thanks to the restrictions placed on type 2 grammars. Give type 1 grammars for which the constructions for Kleene closure and catenation do not work.

2. Is $\{1^n + 1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$ in $\mathcal{L}_2$? If so give a type 2 grammar which generates it (and accompanying proof), otherwise prove that it is not possible.

3. Prove that $\{w^2 \mid w \in \{a, b\}^*\}$ is not a type 2 language. Can you give a grammar to generate this language?

4. Notice that the pumping lemma assumed that the maximum length of the right hand side of production exceeded 1. Prove the pumping lemma for grammars not satisfying this condition.

Part III

Turing Machines as Language Acceptors

Chapter 5

Turing Machines

5.1 Introduction

In the previous chapters we have established a relation between language classes and different machines. We still haven't worked out which machines are capable of computing type 1 and type 0 languages, and this is what we now set out to find out. However, we will take a different approach to the previous chapters. Whereas before the exploratory process started from the grammar definitions and went out in search of an appropriate machine, now we will do the opposite.

You should have already encountered Turing machines in your course. It is interesting to analyze what classes of languages Turing machines can recognize under different constraints. Eventually we will relate this to phrase structure grammars and thus complete the analysis of Chomsky's taxonomy of languages.

Before we start, we will however briefly go through Turing machines, and their basic properties and limitations. This will be followed by an overview of how a Turing machine can be seen as a language acceptor (rather than as a function calculator as you have already encountered it).

5.2 Definitions

A Turing machine is a (theoretical) automaton which is basically an extended version of a pushdown automaton. As with pushdown automata, a Turing machine has a central 'processing' part which can be in one of a finite number of states. As for storage, Turing machines use an infinite tape which they can read from and write to. Unlike pushdown automata, they control the head which reads and writes on the tape by moving it to the right or left. Also, Turing machines receive their input on the same tape which they use for storage (even though their capabilities are not affected if they use a separate input source, like a pushdown automaton).

Definition: A Turing machine is a 6-tuple $M = \langle K, \Sigma, T, P, q_1, F \rangle$, where:

- K is a finite set of states.
- Σ is a finite set of symbols which can appear on the machine tape. Amongst these is the special blank symbol $\square$.
- $T \subseteq (\Sigma \setminus \{\square\})$ is the set of input symbols which may appear on the tape at the beginning as the input of the computation.
- P is the 'program' of the machine. For some non-terminal state, tape symbol combinations, P tells us what the next state will be, what to write on the tape at the current position, and whether the tape head is to be moved to the right, the left or left where it is.

$$P : (K \setminus F) \times \Sigma \to K \times \Sigma \times \{R, L, S\}$$

- $q_1 \in K$ is the initial state the machine always starts off in.
- $F \subseteq K$ is a finite set of final states.

Note that various other definitions of Turing machines may be found in the literature. Some use two special states q_y and q_n as the only final states. Depending on which state the machine terminates, it can be seen to be answering *yes* or *no*. In our variant, the yes/no answer will be written to the tape using a particular convention which we will define later. Usually, the possibility to leave the head in the current position (S) is not given.

However, it is easy to show that this does not change the computing powers of a Turing machine, so we opt for the slightly more convenient version.

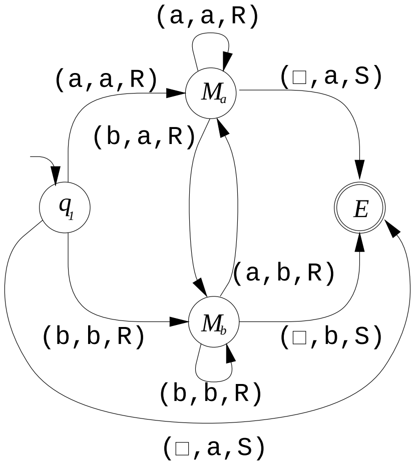
When describing a Turing machine, it is not very helpful just to list the program P. Thus, we usually use a graph to depict the machine, where every node represents a state, and directed edges are labeled with a triple (current symbol on tape, symbol to write on tape, direction in which to move) representing the program contents. Thus, there is a connecting arrow from state q to q' labeled (a, b, δ) if and only if $P(q, a) = (q', b, \delta)$. As usual, we mark the initial state by an incoming arrow, and final states by two concentric circles.

Example: Design a Turing machine to add a symbol a to the beginning of the input string (ranging over $\{a, b\}^*$).

We simply copy the input, shifted one space to the right and writing an initial a. q_1 is the initial state, where we write an a and start copying symbols to the right. States M_a and M_b represent, respectively, the state in which the machine is remembering that the previous symbol read was an a or a b. E is the only terminal state in which the machine terminates.

State	Read	Write	Next State	Move
q_1	a	a	M_a	R
q_1	b	a	M_b	R
q_1	$\Box$	a	E	S
M_a	a	a	M_a	R
M_a	b	a	M_b	R
M_a	$\Box$	a	E	S
M_b	a	b	M_a	R
M_b	b	b	M_b	R
M_b	$\Box$	b	E	S

Graphically, this can be expressed as:



Note that this machine can be made more efficient by removing state q_1 and making state M_a the initial state.

We have thus informally shown what it means for a Turing machine to compute a function. However, before we can prove anything about them, we need to formalize this notion.

To completely describe the state in which a Turing machine is, we need to specify what is written on the tape, where the tape head currently lies and the state of the automaton. For convenience, we will record the tape contents as three separate pieces of information: the symbol currently pointed to, a string representing the tape contents to the left of the tape head, and another string for the symbols to the right of the tape head. To avoid handling infinite strings, we only consider the symbols where the input has been written or the tape head has passed over.

Definition: The configuration of a Turing machine, is a 5-tuple (q, i, α, a, β), where:

- $q \in K$ describes the state of the Turing machine.
- $i \in \mathbb{Z}$ describes the position of the tape head.
- $\alpha \in \Sigma^*$ is the string to the left of the tape head.
- $a \in \Sigma$ is the symbol the tape head is currently pointing at.
- $\beta \in \Sigma^*$ is the string to the right of the tape head.

Initially, the input is written on the tape and the tape head is placed on the first symbol of the input.

Definition: The *initial configuration* of a Turing machine M with input $x \in T^*$, written as $C_0^{M,x}$ (or simply C_0^x if the Turing machine we are referring to is obvious):

- If $x = \varepsilon$, we define $C_0^x \stackrel{def}{=} (q_1, 1, \varepsilon, \Box, \varepsilon)$.
- Otherwise, if $x \neq \varepsilon$, we define $C_0^x \stackrel{def}{=} (q_1, 1, \varepsilon, head(x), tail(x))$.

Recall that we used automata configurations to define the life cycle of a configuration:

Definition: A configuration $C = (q, i, \alpha, a, \beta)$ is said to evolve in one step to $C' = (q', i', \alpha', a', \beta')$ in a Turing machine $M = \langle K, \Sigma, T, P, q_1, F\rangle$, written as $C \vdash_M C'$ (or $C \vdash C'$ if we are obviously referring to M), if:

- $P(q, a) = (q', a', S)$ and $\alpha' = \alpha$, $\beta' = \beta$ and $i' = i$.
- $P(q, a) = (q', b, R)$ and $\alpha' = \alpha b$, $\beta' = tail(\beta)$, $a' = head(\beta)$ and $i' = i + 1$.
- $P(q, a) = (q', b, L)$ and $\alpha' = front(\alpha)$, $\beta' = b\beta$, $a' = last(\alpha)$ and $i' = i - 1$.

Note that the $front$ and $last$ functions are similar to the $head$ and $tail$, but act on the rear rather than head of the list.

Another string function we will find useful is *initial* which returns the initial part of string s until the first blank character appears. A functional programming-like definition of this function would look like:

$$\begin{array}{lcll} initial([]) & = & [] & \\ initial(x : xs) & = & [] & \text{if } x = \Box \\ & & x : initial(xs) & \text{otherwise} \end{array}$$

Definition: The *output* of a configuration $C = (q, i, \alpha, a, \beta)$, is the initial part of the tape, starting from position 1:

$$output(C) \stackrel{def}{=} initial(tape)$$

where $tape = tail^{-i}(\beta)$ if $i \leq 0$, and $tape = tail^{|\alpha|-i+1}(\alpha a \beta)$ otherwise.

Definition: A configuration C is said to evolve to C' in Turing machine M ($C \vdash_M^* C'$) if either:

- $C = C'$ or ...
- There is a configuration C'' such that $C \vdash_M C''$ and $C'' \vdash_M^* C'$.

Definition: A configuration $C = (q, i, \alpha, a, \beta)$ of Turing machine M is said to be a *halting* one if there is no configuration C' such that $C \vdash_M C'$.

A halting configuration is said to *successful* if $q \in F$, but otherwise it is said to *fail*.

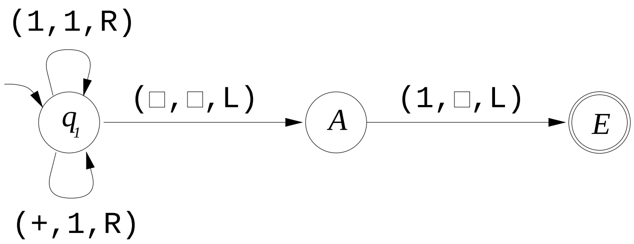
Definition: The function defined by a Turing machine M, usually denoted by f_M is defined by: $f_M(x) = y$ if and only if there is a successful halting configuration C' such that $C_0^x \vdash^* C'$ and $output(C') = y$.

Definition: A function f is said to be Turing computable, if there is a Turing machine M such that $f_M = f$.

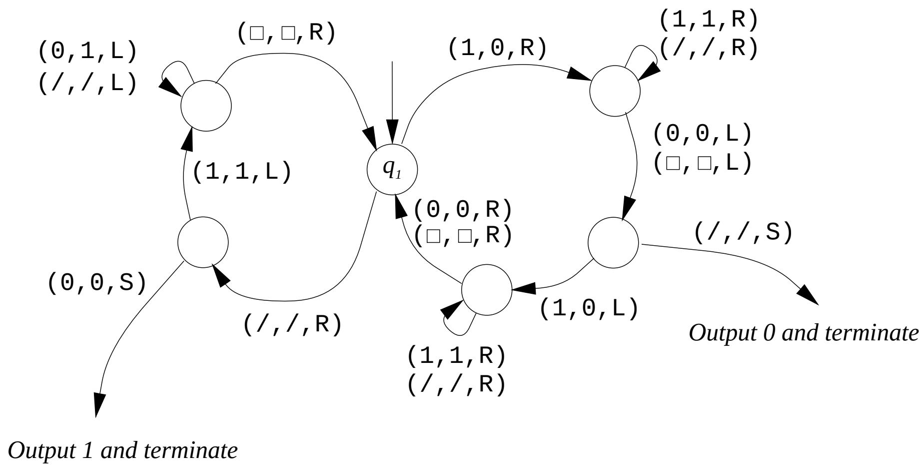
Example: Addition is Turing computable.

Let us represent numbers using a unary notation. In other words, we will represent $n \in \mathbb{N}$ as 1^n. The input of the machine will thus be $1^n + 1^m$ and we expect the output to be 1^{n+m}.

The Turing machine which performs this operation is rather simple. It scans through the input until it finds a +, replaces it with a 1 and continues scanning to the right. Once a blank is found, we move back to the left and remove the last 1.



Example: What about a Turing machine which, given two positive numbers separated by / returns 1 if the second is divisable by the first, but 0 otherwise?



At this point, it is interesting to note the next results:

Theorem: For a given input alphabet (T), there is only, at most, a countably infinite number of different Turing machines.

Proof: Consider a Turing machine M. Enumerate the states $q_1, \ldots q_n$, and the symbols $s_1, \ldots s_m$. Similarly we can encode the directions as d_1, d_2 and d_3. To specify M, we need to encode the program 5-tuples and the final states.

Each program 5-tuple is of the form $(q_i, a_j, q_k, a_l, d_m)$. Using Gödel numbering, this can be represented as a number:

$$2^i 3^j 5^k 7^l 11^m$$

Hence, a finite sequence of 5-tuples, can be represented as a finite sequence of numbers $n_1, n_2, \ldots n_m$. This, in turn can also be encoded as an integer:

$$2^{n_1} 3^{n_2} \ldots p_m^{n_m}$$

where p_m is the mth prime number.

Similarly, the set of final states can be encoded as an integer. Thus, the program and final states of M can be encoded as a positive integer, such that no two distinct Turing machines map to the same integer. Hence we have established a one-to-one encoding from Turing machines to $\mathbb{N}$, implying that the there is at most a countably infinitude of Turing machines.

□

Now consider the family of functions $\{f_L \mid L \subseteq \Sigma^*\}$, such that $f_L(x)$ is 1 if $x \in L$, 0 otherwise. But for non-empty Σ, we have an uncountable number of possible values of L, and thus an uncountable number of such functions. Hence, we can conclude that:

Corollary: There is an uncountable number of languages over non-empty alphabet Σ which are not recognizable by a Turing machine.

Note that we are using the phrase *L is recognizable by M* to mean that f_M is the function which returns 1 if $x \in L$, 0 otherwise.

This result is suspiciously similar to the result already given, that phrase structure grammars can only describe an infinitesimal (countably infinite) portion of possible languages over a non-empty alphabet Σ (which are uncountable). The question naturally arises: are Turing machines and phrase structure grammars equally expressive?

This leads to Church's thesis: every effectively computable function is Turing-computable. According to this, there ought to be no mechanical means of establishing whether a string is in a language or not. This means, that if we can use a phrase structure grammar to algorithmically compute whether a string lies in a language or not, then phrase structure grammars cannot be more expressive than Turing machines. We will discuss this in more detail later.

5.3 Equivalent Turing Machines

Before proceeding any further, we will list a number of modifications which may be performed to the Turing machine model we are using which do not affect the computing power of the resulting machine. We will not prove any of the results since you should have already encountered them in a different course.

5.3.1 Extensions

According to Church's thesis, no extension to a Turing machine should increase its computing power.

Synchronized multi-tape Turing machines: These machines are basically a Turing machine but with more than one tape under its control. It is called synchronized, since all tape heads move together. Initially, the input is written on tape 1 and eventually it is also read off tape 1.

Multi-tape Turing machines: A multi-tape Turing machine is a Turing machine which has under its control a (finite) number of infinite tapes. Program instructions now contain, not just one head movement instruction, but as many as there are tapes. Initially, the input is placed on tape 1 and all heads start off in the first position of all tapes. The output is also read off tape 1.

Off-line Turing machines: These machines have two tapes: one is read-only and used to give the input, while the other is read-write and is used to write the output.

Multi-dimensional Turing machines: These have a multi-dimensional tape. In an n-dimensional machine, there are $2n$ possible head motions. One particular direction is chosen in which the input and output are to appear (starting from the initial head position). Thus, for example, $n = 1$ is a normal Turing machine, whereas $n = 2$ is a machine with an infinite plane covered in squares on which to write. The tape head can now move in 4 directions (N, S, E, W). In three dimensional space, we can move the head in any of 8 directions (N, S, E, W, U, D).

Non-deterministic Turing machines: The program is no longer a function but a relation. Thus, for a state, read symbol pair, the machine may go to different states or write different symbols on the tape or move in different directions (or a combination of these).

5.3.2 Restrictions

The definition of a Turing machine we have given is more complicated than necessary. Certain features can be cut down or removed altogether without affecting the computing power of the machine.

Semi-infinite tape: This variant has a tape which is only infinite in the right direction. The input is placed on the extreme left of the tape, where the head starts off from.

Forced head movement: The S option to keep the tape head at the same position may be done away with.

Binary Turing machine: The tape alphabet is restricted to 1 or 0.

Two-state Turing machine: The machine can only be in either of two possible states.

The set of computable functions in all these cases remains exactly the same as the set of Turing computable functions.

5.4 Limitations of Turing machines

To close this brief overview of Turing machines, we will examine in more detail what the limitations of Turing machines are. Although most of the results we will quote are not necessary for the remainder of the course, the contents are necessary for a fuller appreciation of the complete course.

5.4.1 Universal Turing Machines

Church's thesis states that any function computable by a computer program can also be computed by a Turing machine. The main gap people usually fail to connect between Turing machines and computers is that whereas computers can run different programs, a Turing machine has one set of instructions (P) which it can execute. To see that there is, in fact, no paradox resulting from the two execution styles, we may take either of two alternative viewpoints.

A computer also has a small number of instructions hardwired into it which it executes forever without changing. The underlying circuit which fetches an instruction from memory and executes it is the underlying automaton program and the actual machine code instructions are simply the input. This is no different from a Turing machine.

On the other hand, recall that a Turing machine can be encoded as a single number, which gives the program instructions and initial and final states. The encoding/decoding process is clearly Turing computable. Thus, we can construct a Turing machine which takes a Turing machine's instructions and its input $(E(M) * x)$ as input and emulates it. This is called a Universal Turing machine.

The existence of such a machine is proof of the computing power possible from Turing machines. It also raises a number of interesting questions.

Consider a universal Turing machine U with input $E(M) * x$. Clearly, it terminates and fails if M terminates and fails with input x, terminates and succeeds with output o if M terminates and succeeds (with output o) when given input x, and loops forever if M fails to terminate when given input x. Can we go one step further, and build a super-universal Turing machine which also terminates with an appropriate output when given input $E(M)*x$, where M does not terminate when given input x?

In fact, we cannot construct such a machine. A simpler problem — the Halting Problem — can be shown to be unsolvable, from which we can then deduce that super-universal Turing machines cannot be built.

Definition: The halting problem is a function defined as follows:

$$f(E(M) * x) = \begin{cases} 1, & \text{if } M \text{ terminates with input } x \\ 0, & \text{otherwise} \end{cases}$$

Note that, if we can build a super-universal Turing machine, then the halting problem is solvable. Equivalently, if the halting problem is unsolvable, then there is no super-universal Turing machine.

Theorem: The halting problem is not Turing computable.

Proof: Assume it is solvable by a Turing machine M, which given input $E(T) * x$ returns 1 if T terminates with input x, 0 otherwise.

We can clearly build a Turing machine T, which, if the input is 1 loops forever but terminates if it is 0. We can also build a copying Turing machine C, which given an input x outputs $x * x$.

Now we can construct a Turing machine P, which upon input x, starts by executing C, then the execution is 'transferred' to M (the solution to the halting problem) and finally acts like T.

What does P do when given input $E(P)$? After the first part, we end up with $E(P) * E(P)$. We now have two cases to consider:

- P terminates with input $E(P)$: Hence, M leaves 1 in the tape, which T interprets to loop forever. Hence it does not terminate.

- P does not terminate with input $E(P)$: This time round, M leaves 0 on the tape, which T computes as 'terminate'. Hence it terminates.

Therefore, the construction of M is impossible.

□

Recall that this is but one of the uncountable set of non-Turing computable functions. Other problems may not be as easy to prove, so we try to use our knowledge about the halting problem to show that certain other functions are non-computable. The basic idea is that if we can show that a Turing machine to solve a problem Π can be used to solve the halting problem, then so such Turing machine is possible, and Π is not Turing computable.

We will be only discuss decision problems — which can have only a yes/no answer. Essentially, a decision problem is a way of partitioning the set of strings over Σ (a finite alphabet) in the domain of the problem into two: those returning yes, and those returning no. For a decision problem Π, let D_Π be the domain of the problem — the set of strings over Σ which are meaningful as inputs to Π. D_Π is partitioned into two Y_Π and N_Π, the set of inputs returning yes and those returning no respectively.

We say that a problem is solvable, if there is a Turing machine which terminates successfully when given any input $I \in D_\Pi$, outputting 1 if $I \in Y_\Pi$, 0 otherwise.

A problem Π is said to be reducible to Π' ($\Pi \preceq \Pi'$) if there is a Turing machine which, when given an encoding $I \in D_\Pi$, produces an encoding $I' \in D_{\Pi'}$ such that $I \in Y_\Pi \Leftrightarrow I' \in Y_{\Pi'}$.

Theorem: Let $\Pi \preceq \Pi'$. Then Π' is solvable implies that Π is solvable, or equivalently, Π is unsolvable implies that Π' is unsolvable.

Example: Consider the decision problem: Does a given Turing machine halt when given input ε?

If we refer to the halting problem HP and the empty string halting problem as εHP, we show that $HP \preceq \varepsilon HP$. Consider an instance of HP. We are given a Turing machine M and input x and we have to decide whether M halts with x.

If there is a Turing machine solving εHP then we can give it as input a Turing machine M_x, which initially writes x on the tape and then starts behaving like M. Thus, given an instance of HP we are constructing an instance of εHP. Clearly M_x terminates on empty input if M terminates with input x. Similarly, if M terminates with input x, then M_x terminates when given empty input. Hence $M \in Y_{HP}$ if and only if $M_x \in Y_{\varepsilon HP}$.

Therefore $HP \preceq \varepsilon HP$, and hence εHP is unsolvable.

We will not go any further with computability issues. This is the stuff other courses are made of. The only reason for their inclusion in this course is that they follow naturally from the previous discussion and serve as a reminder since the approach used for unsolvability issues will later reappear when we discuss intractability.

5.5 Exercises

1. Discuss how you could modify the Turing machine example calculating whether a number is divisible by another to decide whether a given number is prime or not.

2. Build a Turing machine which computes the following function:

$$f(x) \quad = \quad \begin{cases} 1, & \text{if } x = a^i b^i c^i \text{ for some } i \in \mathbb{N} \\ 0, & \text{otherwise} \end{cases}$$

   What are the implications of this regarding pushdown automata?

3. Design a Turing machine which outputs 1 or 0 depending on whether the input x is generated by the following regular grammar:

$$\begin{array}{lcl} G & \stackrel{def}{=} & \langle \{a, b, c\},\ \{S, A, B\},\ P,\ S \rangle \\ P & = & \{\ \ S \to aA \mid bB, \\ & & \phantom{\{}\ \ A \to aA \mid cB \mid b \\ & & \phantom{\{}\ \ B \to bB \mid cA \mid a \quad \} \end{array}$$

4. Design a Turing machine which outputs 1 or 0 depending on whether the input x is generated by the following regular grammar:

$$
\begin{array}{rcl}
G & \stackrel{def}{=} & \langle\{a,b\},\ \{S,A,B\},\ P,\ S\rangle \\
P & = & \{\ \ S \to bA \mid aB, \\
 & & \quad\ A \to aB \mid a, \\
 & & \quad\ B \to bA \mid b \qquad \}
\end{array}
$$

5. Design a Turing machine which outputs 1 or 0 depending on whether the input x is generated by the following regular grammar:

$$
\begin{array}{rcl}
G & \stackrel{def}{=} & \langle\{a,b,c\},\ \{S,A,B\},\ P,\ S\rangle \\
P & = & \{\ \ S \to cA \mid cB, \\
 & & \quad\ A \to aB \mid c, \\
 & & \quad\ B \to bA \mid c \qquad \}
\end{array}
$$

6. Generalize your solutions to work with general regular grammars.

Chapter 6

Recursive and Recursively Enumerable Languages

6.1 Introduction

Turing machines can be used to accept languages. We have already used an informal definition for the language accepted by a Turing machine, namely that a language L is accepted by Turing machine M, if the function computed by M is the inclusion function of L ($f(x)$ is 1 if $x \in L$, 0 otherwise). However, there are other alternative means of accepting languages using Turing machines which we will investigate. Recall that a Turing machine does not necessarily terminate and if it does terminate, it may fail and return no output. Two other possible ways to define classes of languages are:

- there exists a Turing machine which given an input, halts (successfully) and returns 1 if it is in the language L, but fails (by terminating unsuccessfully) otherwise.

- there exists a Turing machine which terminates successfully and returns 1 if the input is in the language L.

This is the kind of material we will be investigating in this part of the course.

6.2 Recursive Languages

6.2.1 Definition

We start off by defining the class of languages we have been experimenting with in the previous chapter.

Definition: A language $L \subseteq T^*$ is said to be *recursive* if there exists a Turing machine which computes the characteristic function of L, χ_L:

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

We will use $\mathcal{L}_R$ to represent the class of all recursive languages.

Note that some textbooks use the term *decidable* or *Turing-decidable* to describe this class of languages. It is important to realize that the given Turing machine must terminate successfully for all inputs, leaving either 1 or 0 as output.

However, this can be somehow relaxed without affecting the class of languages recognized, as the following theorem shows.

Theorem: A language L is recursive if and only if there exists a Turing machine which terminates and succeeds when the input is in L, but terminates and fails if it is not.

Proof: Let L be a recursive language. Then there exists a Turing machine which computes the characteristic function of L. Since Turing machines with a semi-infinite tape are equivalent to ones with an infinite tape, we can assume that the Turing machine uses only the tape to the right of the initial position. We enhance this machine such that it initially writes a special symbol one position to the left and returns back to the initial position. At the end, we scan the tape to the left until we encounter the special symbol, and check what is written on the tape just to the right of this symbol. If it is 1 (the input string is a member of the language) we go to a terminal state, but we do not provide an action for the case where a 0 lies on the tape. Hence, if the string was not in L, it terminates and fails.

Conversely, assume that for a language L, there is a Turing machine which always terminates, but succeeds if and only if the input was a string in L. As before, assume that the Turing machine has a semi-infinite tape. Again, use a special symbol to mark the tape. Extend the machine, such that from any of the states which were final, it finds the special symbol, writes 1 to the tape and terminates. What about the failing cases? The machine fails due to the partiality of the program function. Thus, simply make the program function total, sending any new transitions to a new state. From this new state, search back for the special symbol, write 0 and terminate.

□

6.2.2 Closure

The class of recursive languages is closed under catenation, set union, set intersection and complement.

Theorem: $\mathcal{L}_R$ is closed under catenation.

Proof: Consider the 3 tape Turing machine, which takes two Turing machines and a string as input. For a two-part decomposition of the input string $x = x_1x_2$ (there are $|x|$ of them), it writes x_1 to the first tape and x_2 to the second tape. It then emulates the first machine using the first tape, and the second machine using the second tape. Upon termination, the outputs are compared. If they are both 1, then the Turing machine outputs 1 and terminates, but it tries another decomposition if either of them returned 0. If all decompositions have been tried, output 0 and terminate.

Clearly, since both M_1 and M_2 always terminate, and there is a finite number of ways in which the input can be decomposed, this machine always terminates. Furthermore, it always outputs 1 or 0 upon termination. When does it return 1?

It returns 1 if and only if for any one of the decompositions both M_1 and M_2 return 1 when given x_1 and x_2 respectively. Hence, the Turing machine returns 1 iff $\exists x_1, x_2 \cdot x = x_1x_2 \wedge f_{M_1}(x_1) = 1 \wedge f_{M_2}(x_2) = 1$.

But $F_{M_i}(x_i) = 1$ if and only if $x_i \in L_i$.

Hence, it returns 1 iff $\exists x_1, x_2 \cdot x = x_1x_2 \wedge x_1 \in L_1 \wedge x_2 \in L_2$, which is equivalent to $x \in L_1L_2$.

□

Theorem: $\mathcal{L}_R$ is closed under set complement.

Proof: If $L \in \mathcal{L}_R$, we can produce a Turing machine which calculates the characteristic function of L using only the positive (right) direction of the tape. Modify this Turing machine such that it initially marks the tape with a special symbol, and at the end of the computation, move back to the left and replace the output from 1 to 0 or vice-versa. The new Turing machine will be computing:

$$f(x) = \begin{cases} 1, & \text{if } x \notin L \\ 0, & \text{otherwise} \end{cases}$$

But $f = \chi_{T^* \setminus L}$. Hence, we have constructed a Turing machine which computes the characteristic function of $T^* \setminus L$, implying that $T^* \setminus L$ is recursive.

□

Theorem: $\mathcal{L}_R$ is closed under set union and intersection.

Theorem: If $L_1, L_2 \in \mathcal{L}_R$, then there are two terminating Turing machines M_1 and M_2 which calculate their characteristic functions. Since a universal Turing machine is possible, we can extend it to emulate more than one Turing machine in sequence. We append an extra module to this universal Turing machine, such that upon termination of both executions, it will look at the outputs of the two machines and write its output accordingly:

- For union, write 1 if either of the two outputs is 1, 0 otherwise. the characteristic function f of the resultant machine returns 1 if and only if $f_{M_1}(x) = 1$ or $f_{M_2}(x) = 1$. But $f_{M_i}(x) = 1$ if and only if $x \in L_i$. Hence, $f(x) = 1$ iff $x \in L_1$ or $x \in L_2$ which is equivalent to $x \in L_1 \cup L_2$.

- For intersection, write 0 if either of the two outputs is 0, 1 otherwise. The reasoning is similar to that used in the previous case.

□

6.2.3 Comparison to Chomsky's Language Hierarchy

The next problem which we will now tackle is placing the class of recursive languages within Chomsky's language hierarchy. We have already shown (in the exercises at the end of the last chapter) that there are non-context free languages which are recursive. The languages we have shown to lie in this gap were $\{a^p \mid p \text{ is prime }\}$ and $\{a^ib^ic^i \mid i \in \mathbb{N}\}$. In both cases, we have shown that these languages are not context free using the pumping lemma, but that they are recursive by constructing a Turing machine which computes their characteristic function. But Turing machines, as it has already been noted, are just a jazzed up version of pushdown automata. In fact, given a pushdown automaton, it is possible to construct a Turing machine which emulates it. Hence we seem to be able to guarantee that, at least, $\mathcal{L}_2 \subset \mathcal{L}_R$.

We also know that $\mathcal{L}_2 \subset \mathcal{L}_1 \subseteq \mathcal{L}_0$. So can we go further and place $\mathcal{L}_R$ with respect to $\mathcal{L}_0$ and $\mathcal{L}_1$?

Theorem: Every type 1 language is recursive.

$$\mathcal{L}_1 \subseteq \mathcal{L}_R$$

Proof: Consider a type 1 language L. By definition, there is a context sensitive grammar $G = \langle \Sigma, N, P, S \rangle$ which generates L: $\mathcal{L}(G) = L$. Recall that all production rules in a context sensitive grammar are of the form $\alpha \to \beta$, where $|\alpha| \leq |\beta|$.

Given G, we would like to construct a decidable algorithm which, given input w, always terminates with a 1 or 0, depending on whether $w \in L$.

Let $|w| = n$. Assume $n > 0$ (if not, we can easily check whether the empty string is in the language, since it must appear in a rule $S \to \varepsilon$, where S is the start symbol).

Now define the chain of sets T_0^n, T_1^n, ...

T_i^n is the set of all sentential forms derivable in G, starting from S, in at most i derivations. There is a simple algorithm we can use to calculate any of these sets:

$$\begin{aligned} T_0^n &\stackrel{def}{=} \{S\} \\ T_{i+1}^n &\stackrel{def}{=} T_i^n \cup \{\beta \mid \exists \alpha \in T_i^n \cdot \alpha \Rightarrow_G \beta \text{ and } |\beta| \leq n\} \end{aligned}$$

It can also be easily seen that this is, in fact, a chain. In other words:

- for any i, $T_i^n \subseteq T_{i+1}^n$ and,
- if $T_i^n = T_{i+1}^n$, then

$$T_i^n = T_{i+1}^n = T_{i+2}^n = T_{i+3}^n = \dots$$

But it is also obvious that, since the maximum length of the strings in T_i^n is n, and there are $|\Sigma \cup N|$ symbols to choose from, $|T_i^n| \leq |\Sigma \cup N|^n$. This implies that the chain must stabilize (reach its maximum) in at most $|\Sigma \cup N|^n$ steps.

We can thus construct a Turing machine which, given a input string x of length n, will construct the chain of T_i^n until it starts repeating. This is guaranteed to terminate. The Turing machine, then simply checks whether x is in the final set.

$\square$

Hence, $\mathcal{L}_R$ is at least as large as $\mathcal{L}_1$. But is it larger? Are there any recursive languages which are not context sensitive?

Theorem: Some recursive languages are not context free.

Proof: The proof is basically a diagonalisation proof but with a refreshing twist.

Consider an alphabet $\Sigma = \{a, b\}$. The set of all context sensitive grammars over this (terminal) alphabet can be enumerated and encoded/decoded using a Turing machine. We will give the details of a possible encoding at the end of the proof. Thanks to this encoding, we thus have an infinite sequence of context free grammars G_1, G_2, ...

The set of strings over Σ can also be enumerated thus giving strings w_1, w_2, ... One possible way of enumerating the strings is by giving shortest strings first, (and for strings of the same length use alphabetical ordering).

Now consider the language L over Σ, defined by $\{w_i \mid w_i \notin \mathcal{L}(G_i)\}$. Clearly, this cannot be a context free grammar. However, (and here comes the twist), given a string $x \in \Sigma^*$ a Turing machine can find i, such that $x = w_i$, construct grammar G_i and check whether $w_i \in \mathcal{L}(G_i)$ using the algorithm given in the previous proof. The output is then negated (since $w_i \in L$ if and only if $w_i \notin calL(G_i)$).

Now, for an enumeration of context sensitive grammars. Clearly, given a grammar we can freely rename the non-terminals (as long as it is done consistently) without affecting the language generated. We rename all the non-terminal states to be in the set $\{A_1,\ A_2,\ \ldots\}$ and in particular, we will rename the initial state to A_1. We will now encode context sensitive grammars into a string of 0s and 1s using the following table:

Symbol	Encoding
a	00
b	001
A_i	$(01)^i$
$\rightarrow$	0011
,	00111

Any grammar over Σ can be encoded, using this scheme, as a string of 0s and 1s representing the production rules separated by commas. Not all strings over 0, 1 are a valid encoding of a grammar, but clearly, given such an encoding, it is easy to check whether it is a valid encoding of a context sensitive grammar and decode it back to the original grammar. Hence, to build a Turing machine which enumerates these grammars and, given an input n outputs G_n, we simply build a Turing machine which generates the strings over 0 and 1 in some sequence. For each, it checks whether it is a valid encoding of a context sensitive grammar, and if so decrements the original input n. It does this until the input is reduced to zero, whereupon the last grammar generated is outputted.

$\square$

Corollary: $\mathcal{L}_1 \subset \mathcal{L}_R$.

Proof: Obvious from previous two theorems.

$\square$

But beyond type 1 languages, we only have the class of languages generated by free structure grammars. The next class of languages we will examine — recursively enumerable languages — will help us show the relationship between recursive and type 0 languages.

6.3 Recursively Enumerable languages

6.3.1 Definition

In the last chapter, we have proved that the halting problem is, in general unsolvable. This indicates that we may be able to describe a larger subset of languages using Turing machines, if we only insist that they terminate and return 1 if the input is in the language, but does not terminate otherwise.

Definition: A language L is said to be a *recursively enumerable* if there is a Turing machine which computes the partial characteristic function of L:

$$\chi'_L(x) = \begin{cases} 1, & \text{if } x \in L \\ \text{undefined}, & \text{otherwise} \end{cases}$$

The class of recursively enumerable languages is denoted by $\mathcal{L}_E$. The definition for recursively enumerable languages may be somewhat weakened:

Theorem: L is recursively enumerable if and only if there is a Turing machine which always terminates successfully outputting 1 whenever the input is a string in L. If the input is not in L, the Turing machine is allowed to behave arbitrarily, except that it may not terminate successfully, returning 1.

Proof: The result is quite similar to that given in the first theorem about recursive languages, and the proof is also quite similar.

If L is recursively enumerable, there is a Turing machine which returns 1 if the input is in L, but fails to terminate otherwise. This already satisfies the desired format of the Turing machine.

Conversely, assume that there is a language L for which there exists a Turing machine which upon input $x \in L$ returns 1, but upon an input not in L, it may do any of the following:

- terminate and fail, or
- terminate and return a value other than 1, or
- loop forever

We can modify this Turing machine, making it total, by sending all undefined combinations in the domain of P to a new state, writing some particular symbol: if $P(q, a)$ is undefined, define it to be $(q', \Box, S)$. We also add the instruction $P(q', \Box) = (q', \Box, S)$. This makes sure that all failing terminations are changed into infinite loops.

What about terminating runs which return values other than 1? The solution is similar to the one given in the previous chapter. We modify the Turing machine to use only the right side of the tape and at the beginning mark the initial position on the tape. At the end of an execution, check the output and, if the result is not 1, loop forever.

$\Box$

6.3.2 Relation to Recursive Languages

By the definition of recursively enumerable languages, it seems, quite naturally to include the whole class of recursive languages.

Theorem: Every recursive language is recursively enumerable.

$$\mathcal{L}_R \subseteq \mathcal{L}_E$$

Proof: Consider a recursive language L. By definition, there is a Turing machine which always terminates successfully, returning 1 if the input was in L, 0 otherwise. By the last theorem we have proved, L is recursively enumerable.

$\Box$

But are we talking about the same class of languages? Is $\mathcal{L}_R = \mathcal{L}_E$? The result we proved in the previous chapter, that the halting problem is unsolvable, seems to indicate otherwise. But can we prove it?

Theorem: Not all recursively enumerable languages are recursive.

Proof: As we have already discussed, we can enumerate the set of all Turing machines. By the theorem where we relaxed the constraints for a language to be recursively enumerable, each one of these machines represents a language. We hence end up with an infinite sequence: L_1, L_2, ... of languages.

Strings over a particular finite alphabet can also be easily enumerated. We thus also have a sequence x_1, x_2, ... of strings.

Define language L_D to be $\{x_i \mid x_i \in L_i\}$.

Claim: L_D is recursively enumerable.

The idea, which we will only outline, is that we can build a Turing machine which, given input w first determine the value of i for which $w = x_i$. It then constructs Turing machine M_i and executes it with input x_i. It returns 1 if and only if $x_i \in L_i$ which is true exactly when $x_i = w \in L_D$.

Hence $L_D \in \mathcal{L}_E$.

Claim: $\bar{L}_D$ is not recursively enumerable.

Assume there is a Turing machine which recognizes (in the recursively enumerable way) $\bar{L}_D$. Clearly, the language it defines must be one of the enumerated ones. Hence, $\bar{L}_D = L_i$ for some i. But:

$$x_i \in L_D \Leftrightarrow x_i \in L_i \Leftrightarrow x_i \in \bar{L}_D \Leftrightarrow x_i \notin L_D$$

Hence $\bar{L}_D$ cannot be recursively enumerable.

But we have proved that recursive languages are closed under set complementation. Hence L_D cannot be recursive.

$\square$

Corollary: $\mathcal{L}_R \subset \mathcal{L}_E$.

6.3.3 Closure

As we have already seen in the previous section, $\mathcal{L}_E$ is not closed under set complement. However, it is closed under set union, intersection and catenation.

Theorem: $\mathcal{L}_E$ is closed under intersection.

Proof: Consider L_1 and L_2, recursively enumerable languages. Let M_1 and M_2 be Turing machines which compute the partial characteristic functions of L_1 and L_2 respectively.

Now we can construct a multi-tape universal Turing machine M which, given an input x executes M_1 with input x on one tape and, upon termination, executes M_2 with input x on another tape. Now compare the results outputted on the tape. If both of them are 1, output 1 and terminate.

If $x \notin L_1 \cap L_2$, then $x \notin L_1$ or $x \notin L_2$. Hence either M_1 or M_2 will fail to terminate with input x, and so, so does M. But if $x \in L_1 \cap L_2$, then $x \in L_1$ and $x \in L_2$. Thus both M_1 and M_2 terminate and return 1, and therefore, so will M.

Therefore, M computes the partial characteristic function of $L_1 \cap L_2$.

□

Theorem: $\mathcal{L}_E$ is closed under union.

Proof outline: Consider L_1 and L_2, both recursively enumerable languages.

We construct a universal Turing machine M which given input x emulates the two Turing machines which recognize L_1 and L_2 step by step, both with input x. In other words, we do not run through either machine at one go, but alternately perform the transitions on them. When either of them terminates, check whether the output is 1 and output 1 if this is so, but continue emulating the other machine otherwise.

If $x \in L_i$ for $i = 1$ or 2, M_i (the Turing machine recognizing L_i) will return 1 in a finite number of transitions and hence so will M. If neither of them is in the required language, neither M_i will terminate and thus neither will M.

But this is a Turing machine which returns 1 if $x \in L_1 \cup L_2$ and fails to terminate otherwise, and is thus computing the partial characteristic function of $L_1 \cup L_2$.

Hence, $L_1 \cup L_2$ is recursively enumerable.

□

Theorem: $\mathcal{L}_E$ is closed under catenation.

Proof outline: Consider $L_1, L_2 \in \mathcal{L}_E$. Let M_1 and M_2 be Turing machines computing their respective partial characteristic functions.

Note that $x \in L_1 L_2$ if and only if $\exists x_1, x_2 \cdot x = x_1 x_2 \wedge x_1 \in L_1 \wedge x_2 \in L_2$.

But there are exactly $|x| + 1$ possible decompositions of x into two parts. Imagine a *dynamic* universal Turing machine M, which, given an input x, will emulate $|x| + 1$ copies of M_1 and $|x| + 1$ copies of M_2 in parallel. By this we mean that rather than pick a single machine and emulate it until it terminates, we alternate between the machines executing one transition at a time.

The ith copy of M_1 is given the first i characters of x as input, whereas the corresponding ith copy of M_2 is given the last $|x| - i$ characters. When a corresponding pair of machines has terminated we check their input and if both have returned 1, M outputs 1 and exits. Otherwise it continues looping.

Now, if $x \in L_1L_2$, there is a decomposition $x = x_1x_2$ such that $x_1 \in L_1$ and $x_2 \in L_2$. Thus, there is a decomposition for which M_1 with input x_1 and M_2 with input x_2 both terminate, outputting 1 on their tape. Hence, M will eventually terminate and output 1 on its tape.

On the other hand, if $x \notin L_1L_2$, then for every decomposition $x = x_1x_2$, either $x_1 \notin L_1$ or $x_2 \notin L_2$. Hence, one of each corresponding pair will fail to terminate, and thus so will M.

$\square$

6.3.4 Free Grammars

Recall that the class of recursive languages is larger that the class of type 1 grammars, but smaller than the class of recursively enumerable languages. Hence $\mathcal{L}_1 \subset \mathcal{L}_E$. But how is $\mathcal{L}_E$ related to $\mathcal{L}_0$, the class of languages accepted by general phrase structure grammars?

Theorem: Languages generated by phrase structure grammars are recursively enumerable.

$$\mathcal{L}_0 \subseteq \mathcal{L}_E$$

Proof: Consider $L \in \mathcal{L}_0$. If we can find an algorithmic way of enumerating the strings in L (which, if you recall, must be countably infinite) as $\langle w_1, \ , w_2 \ \ldots \rangle$, we can build a Turing machine which produces these strings in sequence, and compares each produced string with the input. If it matches, output 0 and terminate, but otherwise continue producing strings.

If the input $x \in L$, it must be equal to w_i, for some i. Hence, the Turing machine will eventually terminate and return 1. However, if $x \notin L$, it matches no w_i and will thus never terminate. It is therefore computing the partial characteristic function of L.

But how do we algorithmically enumerate the strings in L? The easiest way is to list them in shortest derivation first order. Define the sequence of sets of strings over $(\Sigma \cup N)^*$:

$$N_0 \stackrel{def}{=} \{S\}$$
$$N_{i+1} \stackrel{def}{=} \{\beta \mid \exists \alpha \in N_i \cdot \alpha \Rightarrow_G \beta\}$$

Each of these sets is finite (since N_0 is a finite set of finite strings, and we have only a finite number of production rules) and, if $x \in \mathcal{L}(G)$ then $x \in N_n$ for some value of n. Clearly, there is a simple algorithm to generate the sets N_i. Also, if we sort the terminal strings in N_i (say alphabetically), we get an enumeration for every string in $\mathcal{L}(G)$.

□

Theorem: For every recursively enumerable language, there is a type 0 grammar which generates it.

$$\mathcal{L}_E \subseteq \mathcal{L}_0$$

Proof: Let M be a semi-infinite tape Turing machine which accepts L. We can modify M such that it never writes a blank symbol on the tape, by adding a new tape symbol $\diamond$, replacing all program instructions $P(q, a) = (q', \Box, \delta)$ by $P(q, a) = (q', \diamond, \delta)$, and for every program instruction $P(q, \Box) = (q', a, \delta)$ we add $P(q, \diamond) = (q', a, \delta)$. In plain language, we make sure that the machine writes a new symbol rather than blank and upon reading the new blank character, it will behave just as if it has read a normal blank character. Call this modified Turing machine M'. Clearly, the transitions of M' are just like those of M except that the output may be different because of the new blank symbols. Thus M terminates exactly when M' terminates. But M terminates if and only if its input is in L. Hence M' terminates if and only if the input it receives is in L.

Let $M' = \langle K, \Sigma, T, P, q_1, F\rangle$. We now construct a grammar G which emulates the behaviour of M', but in reverse. Thus $S \stackrel{*}{\Rightarrow}_G x$ if and only if M' reaches a final state when receiving input x.

The trick we use is to start off with two copies of x and the initial state of M'. However, we will only allow one copy of the string to be modified. If the state eventually reaches a terminal one, we will clear away all rubbish off the string and leave just x.

The non-terminal symbols of G will consist of the states Q of M', pairs of symbols (the first is the terminal symbol originally appearing on the tape and the other is any tape symbol) ($T \cup \{\varepsilon\} \times \Sigma$) and three extra new symbols S, A and B. S is the start symbol of G. We have three collections of productions:

- Initialization: Initially, we want to set up a string of the form:

  $$q_1(a_1, a_1)(a_2, a_2)\ldots(a_n, a_n)(\varepsilon, \Box)\ldots(\varepsilon, \Box)$$

  The initial state q_1 represents, not just the state of the Turing machine, but also the position of the head. As already noted, we will start off with two copies of the input string, hence the list of (a_i, a_i). Finally, the Turing machine will use a finite amount of tape space which the extra $(\varepsilon, \Box)$ are used for.

  The production rules needed to construct this structure are:

  $$\begin{array}{l} S \to q_1 A \\ A \to (a, a)A \mid B \quad \text{(for every } a \in T\text{)} \\ B \to (\varepsilon, \Box)B \mid \varepsilon \end{array}$$

  We would like to prove that for every string $a \in T^*$, where $a = a_1 a_2 \ldots a_n$, and for every number m:

  $$S \overset{+}{\Rightarrow}_G q_1(a_1, a_1)(a_2, a_2)\ldots(a_n, a_n)(\varepsilon, \Box)^m$$

  The proof is easy and straightforward.

- Emulation: The emulation production rules are translations of the program instructions:

  - For every program rule of the form $P(q, x) = (q', x', S)$ we add a family of production rules:

    $$\{q(a, x) \to q'(a, x') \mid a \in T \cup \{\varepsilon\}\}$$

  - For every program rule of the form $P(q, x) = (q', x', R)$ we add the family of productions:

    $$\{q(a, x) \to (a, x')q' \mid a \in T \cup \{\varepsilon\}\}$$

  - Finally, or every program rule of the form $P(q, x) = (q', x', L)$ we add the family of productions:

    $$\{(b, y)q(a, x) \to q'(b, y)(a, x') \mid a, b \in T \cup \{\varepsilon\}, y \in \Sigma\}$$

At this stage we would like to prove that:

$$\begin{array}{lll}\exists m & \cdot & q_1(a_1,a_1)(a_2,a_2)\dots(a_n,a_n)(\varepsilon,\Box)^m \stackrel{*}{\Rightarrow}_G \\ & & \quad (a_1,X_1)\dots(a_i,X_i)q(a_{i+1},X_{i+1})\dots(a_{m+n},X_{m+n})\end{array}$$

if and only if Turing machine M' with input $a_1a_2\dots a_n$ can reach the configuration:

$$(q, i+1, X_1\dots X_i, \mathit{initial}(X_{i+1}\dots X_{m+n}))$$

This can be proved by induction on the length of the derivation.

- Final productions: Eventually, when a final state is reached, we would like to clear away all the garbage. We do this by propagating the final state and remove the redundant second part of the pairs. For every final state $q_f \in F$, we add the following productions:

$$\begin{array}{l}\{(a,x)q_f \rightarrow q_f a q_f \mid a \in T,\ x \in \Sigma\} \\ \{q_f(a,x) \rightarrow q_f a q_f \mid a \in T,\ x \in \Sigma\} \\ q_f \rightarrow \varepsilon\end{array}$$

We can prove that $S \stackrel{+}{\Rightarrow}_G a \Leftrightarrow a \in L$. Hence, $\mathcal{L}(G) = L$ and thus $L \in \mathcal{L}_0$.

$\Box$

Corollary: $\mathcal{L}_0 = \mathcal{L}_E$

6.4 Conclusions

In this chapter, we have used Turing machines to define two classes of languages: recursive ($\mathcal{L}_R$) and recursively enumerable ($\mathcal{L}_E$) languages. These are related by a proper inclusion:

$$\mathcal{L}_R \subset \mathcal{L}_E$$

Furthermore, we have shown that recursively enumerable languages correspond exactly to type 0 languages in the Chomsky hierarchy of languages and the class of recursive languages is larger than the class of type 1 languages. The major results of the classification analysis we have discussed up to now are:

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_R \subset \mathcal{L}_0 = \mathcal{L}_E$$

Language Class	Automaton
Type 0 (recursively enumerable)	Turing machines
Recursive languages	Terminating Turing machines
Type 1 (context sensitive)	?
Type 2 (context free)	NPDA
Type 3 (regular)	DFSA

6.5 Exercises

1. The proof that $\mathcal{L}_1 \subseteq \mathcal{L}_R$ glossed over a couple of details, which may be useful to expand and formalize better. Give a proof of the following statements, and where they should appear in the proof:
   - The sets T_i^n are all finite.
   - $x \in L$ if and only if x eventually appears in the set $T_i^{|x|}$.
2. In the proof given that some recursive languages are not context free, point out where the proof fails for general phrase structure grammars.
3. For finite state automata and pushdown automata, we said that *language L is accepted by machine M* if and only if L is the set of strings which take machine M to a final state. What is the class of languages accepted by Turing machines under this definition?
4. Prove that the membership problem for general phrase structure grammars (type 0) is unsolvable.

Chapter 7

Context Sensitive Languages and General Phrase Structure Grammars

7.1 Context Sensitive Languages

In the previous chapter, we have shown that recursive languages are a proper superset of context sensitive languages. We have not, as yet, proved any other properties of the language, except that the class of such languages $\mathcal{L}_1$ is a proper superset of the class of context free languages $\mathcal{L}_2$. We will now prove some properties of this class of languages.

Theorem: The class of context sensitive languages is a proper subset of the class of free (type 0) languages.

$$\mathcal{L}_1 \subset \mathcal{L}_0$$

Proof: We have already proved that $\mathcal{L}_1 \subseteq \mathcal{L}_0$. Assume that these two classes are, in fact equal. But $\mathcal{L}_0 = \mathcal{L}_E$, implying that $\mathcal{L}_1 = \mathcal{L}_E$. We have also proved that $\mathcal{L}_1 \subset \mathcal{L}_R$ and that $\mathcal{L}_R \subset \mathcal{L}_E$. Hence, $\mathcal{L}_1 \subset \mathcal{L}_E$ which contradicts the previous conclusion that $\mathcal{L}_1 = \mathcal{L}_E$. Hence $\mathcal{L}_1 \neq \mathcal{L}_0$, and thus $\mathcal{L}_1 \subset \mathcal{L}_0$.

□

7.1.1 Closure

Under what operations is the class $\mathcal{L}_1$ closed?

Theorem: $\mathcal{L}_1$ is closed under set union.

Proof: Given $L_1, L_2 \in \mathcal{L}_1$, there exist type 1 grammars G_1 and G_2 (where $G_i = \langle \Sigma_i, N_i, S_i, P_i \rangle$). Assume that $N_1 \cap N_2 = \emptyset$ (otherwise rename the common non-terminals). Also assume that neither language contains the empty string (look at the solution used in context free grammars to see how we would handle the empty string).

If we add a new start symbol S and the production rules $S \to S_1 \mid S_2$, we have a grammar:

$$G = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S\} \cup N', S, P_1' \cup P_2' \cup \{S \to S_1 \mid S_2\} \cup P' \rangle$$

where N' is the set $\{N_{a,i} \mid i \in \{1,2\},\ a \in \Sigma_i\}$, P_i' is the same rules as P_i but with all occurances of terminal symbols a with the non-terminal $N_{a,i}$ and P' is the set of productions of the form $N_{a,i} \to a$.

It is not difficult to prove that $\mathcal{L}(G) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. Also, the new productions obey the restrictions for type 1 grammars. Hence, we have constructed a context sensitive grammar which recognizes $L_1 \cup L_2$. Hence, $L_1 \cup L_2 \in \mathcal{L}_1$ and thus $\mathcal{L}_1$ is closed under set union.

□

Note that the simple alternative of adding $S \to S_1 \mid S_2$ to the sets of productions does not always work. Consider, for example the following sets of productions:

$$\begin{aligned} P_1 &= \{S_1 \to a\} \\ P_2 &= \{S_2 \to b,\ a \to c\} \end{aligned}$$

Using the simpler construction we would get the following derivation:

$$S \Rightarrow_G S_1 \Rightarrow_G a \Rightarrow_G c$$

But c is in neither language. The more complex construction used in the notes gives the correct result:

$$
\begin{array}{ll}
\{ & S \to S_1 \mid S_2, \\
& S_1 \to N_{a,1}, \\
& S_2 \to N_{b,2}, \\
& N_{a,2} \to N_{c,2}, \\
& N_{a,1} \to a, \\
& N_{a,2} \to a, \\
& N_{b,2} \to b, \\
& N_{c,2} \to c \qquad \}
\end{array}
$$

What about catenation of languages? Initial attempts may not be very fruitful. If we just add a new start symbol and a production $S \to S_1S_2$ (where S is the new start symbol and S_1 and S_2 are the old start symbols), adjacent symbols derived from the different start symbols may react together. For example, consider language L which has a grammar with the the following productions:

$$\{S \to A \mid B,\ A \to aA \mid a,\ B \to b \mid bB,\ AB \to c\}$$

where S is the start symbol, clearly, none of the strings accepted by this grammar include c. But now consider if we add $S_n \to SS$. This gives rise to the following derivation:

$$S_n \Rightarrow_G SS \Rightarrow_G AS \Rightarrow_G AB \Rightarrow_G c$$

But $c \notin LL$.

The next possible solution examined would be that of making an extra copy of the symbols (both terminal and non-terminal) where the symbols in the copy appear with a prime symbol (A', a') and are all non-terminal. Similarly, the production rules are copied renaming all occurrences of the symbols. Finally a new non-terminal symbol X is used as the start symbol, with the related productions $X \to S_1S_2'$. Thus, no adjacent non-terminals can react together. But we also have to add rules of the form $a' \to a$ which can be performed prematurely and give problems due to reacting terminals, as is possible in the following case:

$$\{S \to A \mid B,\ A \to aA \mid a,\ B \to b \mid bB,\ ab \to c\}$$

Again, c does not appear in any of the strings in this language, but the following derivation is possible in the grammar derived by the above procedure:

$$
\begin{aligned}
X \Rightarrow'_G \quad SS' &\Rightarrow'_G AS' \Rightarrow'_G AB' \Rightarrow'_G \\
aB' &\Rightarrow'_G ab' \Rightarrow'_G ab \Rightarrow'_G c
\end{aligned}
$$

So, can it be done? The solution is to go one step further, as outlined in the following theorem:

Theorem: $\mathcal{L}_1$ is closed under catenation.

Proof: Given $L_1, L_2 \in \mathcal{L}_1$, there exist type 1 grammars G_1 and G_2 (where $G_i = \langle \Sigma_i, N_i, S_i, P_i \rangle$). Assume that $N_1 \cap N_2 = \emptyset$ (otherwise rename the common non-terminals).

Define the prime operation on strings: $a' = a$ and $(as)' = a's'$. Extend this operation to work on productions $(\alpha \rightarrow \beta)' = \alpha' \rightarrow \beta'$ and on sets of strings or productions: $A' = \{x' \mid x \in A\}$.

Now define a new grammar $G = \langle \Sigma, N, S, P \rangle$, where:

$$
\begin{aligned}
\Sigma &\stackrel{def}{=} \Sigma_1 \cup \Sigma_2 \\
N &\stackrel{def}{=} N_1' \cup N_2'' \cup \Sigma_1' \cup \Sigma_2'' \cup \{S\} \\
P &\stackrel{def}{=} P_1' \cup P_2'' \cup \{S \rightarrow S_1' S_2''\} \cup \{a' \rightarrow a \mid a \in \Sigma_1'\} \cup \{a'' \rightarrow a \mid a \in \Sigma_2''\}
\end{aligned}
$$

We can prove by induction that the language generated by this new (type 1) grammar is, in fact, $L_1 L_2$.

Hence $L_1 L_2$ is also of type 1.

$\square$

Theorem: $\mathcal{L}_1$ is closed under Kleene closure.

Proof: Let $L \in \mathcal{L}_1$. By definition, there exists a context sensitive grammar $G = \langle \Sigma, N, P, S \rangle$ such that $\mathcal{L}(G) = L$.

The trick is to use the same solution as for catenation, and alternate between using the primed symbols and the doubly primed symbols.

Define a new grammar with start symbol X (a new symbol), terminal symbols Σ and non-terminal symbols the set: $N' \cup \Sigma' \cup N'' \cup \Sigma'' \cup \{X\}$. The set of production rules allow us to produce arbitrary repetitions of alternating (old) start symbols S' and S'': $X \to S' \mid S'S'' \mid S'S''X$. We need to derive strings in every occurrences and thus also add $P' \cup P''$. Note that up to this point we can derive strings of the form $x_1'y_1''x_2'y_2'' \ldots x_n'y_n''$ or $x_1'y_1''x_2'y_2'' \ldots x_n'$ where each x_i and y_i is in L. Finally, we would like to remove those extra prime symbols: $\{a' \to a \mid a \in \Sigma\}$.

We can prove by induction that the language generated by this new grammar is, in fact, L^*. Also note that the new grammar is also of type 1.

Hence L^* is also of type 1.

□

7.1.2 Recognition

As we have already seen, Turing machines are far too powerful when compared to type 1 grammars. Even restricting Turing machines to terminating ones (which must return 0 if the string is not in the language) is still too powerful. And yet, pushdown automata are too weak to describe certain context sensitive languages. Can we find a class of machines, compromising between NPDA and Turing machines, to recognize exactly the class of type 1 languages?

The way we do it is by limiting the power of Turing machines. For convenience we will use non-deterministic Turing machines. Such a Turing machine is said to accept a string x, if when given input x, there is at least one sequence of computations in which it will eventually terminate successfully. Otherwise, we say that the Turing machine rejects the string. Recall that the computational power of these machines is the same as that of a normal Turing machine. So how do we limit its power? The solution is quite simple. We just limit the length of tape it is allowed to use. Non-deterministic linear bounded automata are simply non-deterministic Turing machines which receive their input between two special symbols $\ll$, $\gg$ and may only use the tape between these two symbols.

Definition: A *non-deterministic linear bounded automaton* (LBA) is a non-deterministic Turing machine, $\langle K, \Sigma, T, P, \ll, \gg, q_1, F\rangle$ which is given input between two special symbols $\ll$, $\gg$, with the head starting off on $\ll$. The machine is not allowed to go to the left of $\ll$ or to the right of $\gg$. Neither is it allowed to overwrite them:

$$\begin{array}{rcl}(q', a, \delta) \in P(q, \ll) & \Rightarrow & a = \ll,\ \delta \neq L \\ (q', a, \delta) \in P(q, \gg) & \Rightarrow & a = \gg,\ \delta \neq R\end{array}$$

Linear bounded automata are powerful enough to recognize any context sensitive language. Since the productions are non-contracting (they cannot reduce the length of the derived string), any intermediate string in the derivation cannot exceed the length of the string we are trying to derive — the input.

Theorem: For any context sensitive language, there exists a linear bounded automaton which recognizes it.

$$\mathcal{L}_1 \subseteq \{L \mid\ L \text{ is recognized by a LBA }\}$$

Proof: Let $L \in \mathcal{L}_1$. Then there is a type 1 grammar G which recognizes L. Consider a LBA M which has, in its tape alphabet the terminal symbols of G and also pairs of symbols (both terminal and non-terminal) in G. If $\varepsilon \in L$, we first perform a simple check on the input. If there is no input ($\ll\gg$) we make the LBA terminate successfully if $\varepsilon \in L$, otherwise leave the action undefined, hence rejecting it. Otherwise ...

Initially, the input is relegated to the first bank (first elements of the pairs) and the machine writes S at the beginning of the second bank. Then, for every production rule in P, the machine searches through the tape and performs any of the possible replacements (including shifting of the string if necessary), repeating until the two banks become equal. If this happens, the LBA terminates successfully. If no production rules are possible it fails.

Because of the nature of the production rules of type 1 grammars, if $\alpha \stackrel{*}{\Rightarrow}_G \beta$, it has to be the case that $|\beta| \geq |\alpha|$. Hence, if $x \in L$ then $S \stackrel{+}{\Rightarrow}_G x$ and all the intermediate sentential forms are at most $|x|$ symbols long. Hence, the machine will terminate successfully returning 1 and thus $x \in \mathcal{T}(M)$.

If $x \notin L$, there is no derivation of x from S and thus, M will never terminate successfully. Hence $x \notin \mathcal{T}(M)$.

Hence we have constructed a LBA M such that $\mathcal{T}(M) = L$.

$\square$

Theorem: For every LBA M, $\mathcal{T}(M) \in \mathcal{L}_1$.

Proof: The proof is very similar to the proof that every Turing machine has an equivalent phrase structure grammar. If we take exactly the same approach, we have a slight problem. At the end of the simulation, when we clean up the string, we will need to remove the left and right symbols and the machine state from the string. But this reduces the length of the sentential form which is not possible in a type 1 grammar. To solve this problem, we include these symbols *on* the tape into one symbol.

Let $M = \langle K, \Sigma, T, P, \ll, \gg, q_1, F\rangle$.

We will now construct grammar $G = \langle T, N, S, R\rangle$. The set of non-terminal states will consist of 2 new symbols S and A and the set of symbols made up of:

$$N_1 = \{(a, X) \mid a \in T,\ X \in \Sigma\}$$
$$N_2 = \{(a, qX) \mid (a, X) \in N_1,\ q \in K\}$$

These are just like the ones used in the previous proof, but we include the state in the non-terminal if it happens to be on that symbol.

$$\begin{array}{rcl} N_3 & = & \{(a, \ll X) \mid a \in T,\ X \in \Sigma\} \\ N_4 & = & \{(a, X \gg) \mid a \in T,\ X \in \Sigma\} \\ N_5 & = & \{(a, qXY) \mid q \in K,\ (a, XY) \in N_2 \cup N_3\} \\ N_6 & = & \{(a, XqY) \mid q \in K,\ (a, XY) \in N_2 \cup N_3\} \end{array}$$

To accommodate the left and right symbols in the same non-terminal.

$$N \quad \stackrel{def}{=} \quad \{S, A\} \cup \bigcup_{i=1}^{6} N_i$$

As before, we will have three collections of productions, one dealing with the initialization, one with the emulation, and one dealing with the garbage collection.

- Initialization: Initially we want to be able to build any non-empty string within the left and right symbols and with the head on the left symbol:

$$
\begin{aligned}
S &\to (a, q_1 \ll a)A && \text{for all } a \in \Sigma \setminus \{\ll, \gg\} \\
S &\to (a, q_1 \ll a \gg) && \text{for all } a \in \Sigma \setminus \{\ll, \gg\} \\
A &\to (a, a)A && \text{for all } a \in \Sigma \setminus \{\ll, \gg\} \\
A &\to (a, a \gg) && \text{for all } a \in \Sigma \setminus \{\ll, \gg\}
\end{aligned}
$$

- Simulation of LBA: Simulation of the machine is now more complicated due to the extra non-terminals.

  For every program instruction $(q', a', S) \in P(q, a)$, we have three cases.

  1. If $a = \ll$, we add the productions:
     $$\{(b, q \ll c) \to (b, q'a'c) \mid b, c \in \Sigma\}$$
  2. If $a = \gg$, we add the productions:
     $$\{(b, cq \ll) \to (b, cq'a') \mid b, c \in \Sigma\}$$
  3. If a is neither $\ll$ nor $\gg$, we add the productions:

     $$
     \begin{aligned}
     & \{(b, qa) \to (b, q'a') \mid b \in \Sigma\} \\
     \cup\ & \{(b, \ll qa) \to (b, \ll q'a') \mid b \in \Sigma\} \\
     \cup\ & \{(b, qa \gg) \to (b, \ll q'a' \gg) \mid b \in \Sigma\}
     \end{aligned}
     $$

  Similarly, we construct production rules for left and right movement.

  We must also allow the grammar to reach a terminal string once a final state is reached:

  $$
  \begin{aligned}
  & \{(a, qb) \to a \mid a \in T, b \in \Sigma, q \in F\} \\
  \cup\ & \{(a, bqc) \to a \mid a \in T, b, c \in \Sigma, q \in F\} \\
  \cup\ & \{(a, qbc) \to a \mid a \in T, b, c \in \Sigma, q \in F\}
  \end{aligned}
  $$

- Garbage collection: Finally, once at least one symbol becomes terminal, so do all the others:

$$\begin{array}{rl} & \{(a,X)b \to ab \mid X \in \Sigma^*,\ a,b \in T\} \\ \cup & \{a(b,X) \to ab \mid X \in \Sigma^*,\ a,b \in T\} \end{array}$$

Clearly, G is of type 1. Also, it is clear (although we do not prove) that G can generate any string, if and only if it is accepted by M. Hence, all languages which can be accepted by a LBA, can be generated by a context sensitive grammar.

□

Corollary: $\mathcal{L}_1 = \{L \mid \exists M \in LBA \cdot \mathcal{T}(M) = L\}$

7.2 General Phrase Structure Grammars

While examining the properties of recursively enumerable languages, we proved that the class $\mathcal{L}_E$ is, in fact, equivalent to the class of type 0 languages — languages which can be generated by some phrase structure grammar. Hence, all the properties of class $\mathcal{L}_E$, are also properties $\mathcal{L}_0$. The properties we have proved are:

- closed under intersection
- closed under union
- closed under catenation
- not closed under complementation

Furthermore, since $\mathcal{L}_E$ is the class of languages accepted by a Turing machine, Church's thesis says that if we define a larger class of languages, we cannot construct a machine which accepts the extra languages (otherwise the machine is more powerful than a Turing machine). Hence, our faith in phrase structure grammars used to describe any meaningful language is well-founded.

One remaining closure question which we have not yet answered is whether $\mathcal{L}_0$ is closed under Kleene closure. The proof follows just like the one for $\mathcal{L}_1$ given earlier in this chapter.

Theorem: $\mathcal{L}_0$ is closed under Kleene closure.

Proof: Precisely the same construction as used to prove closure of $\mathcal{L}_1$ under Kleene closure is used.

□

This ends our discussion of type 0 languages here, since we have already proved the properties we may need to use. As a closing note, it is interesting to note that Chomsky and Turing were working independently when they defined their grammar hierarchy and machines respectively. The equivalence of the most general concept of computation was not limited to just these, however. Other work done by Church on the Lambda calculus, Post on production systems, Smullyan on formal systems and Kleene in recursive function theory at approximately the same time also led to the same conclusion. In other words, despite the various formalizations of a computation one may use, the resulting system was always most as powerful as a Turing machine.

As an aside, I would like to point out similar results in other fields also first encountered this past century:

- Quantum physics
- Gödel's Incompleteness Theorem
- Chaotic systems
- Inherent unfairness of any voting system

Not surprisingly, the philosophers of science have been kept very busy this past century and in most cases, a lot of work still needs to be done before they catch up. The only other such 'disappointing' result (in the sense that they are all negative results which show that the way we have been looking at the world needs to be thoroughly revised, and there is no hope in removing some inherent undesirable properties) was the realization that there are numbers (such as the square root of 2) which cannot be written as a fraction. But this is straying too far away from the original content of the course, and I will thus end this aside comment here.

7.3 Conclusions

We close this section by constructing a table with the properties of the language classes we have analyzed.

Language Class	Recognizing Automaton	Closure $L \cup M$	$L \cap M$	$\bar{L}$	LM	L^*
Free languages	TM	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
Recursive	Terminating TM	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Context sensitive	LBA	$\surd$			$\surd$	$\surd$
Context free	NPDA	$\surd$			$\surd$	$\surd$
Regular	DFSA	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

7.4 Exercises

1. Where does the proof that for every LBA there is an equivalent context sensitive language, fail if we try to extend it to general Turing machines?

2. Discuss how one can directly construct a Turing machine which accepts the Kleene closure of a recursively enumerable language. *Hint:* Recall how we proved that $\mathcal{L}_E$ is closed under catenation and how one enumerates $\mathbb{N} \times \mathbb{N}$.

Part IV

Algorithmic Complexity

Chapter 8

Algorithm Classification

In the previous chapters, we have encountered the concept of unsolvability. This provides a useful classification of problems — we thus have two classes of problems, solvable and unsolvable.

However, this classification is far too general. Consider the class of solvable problems. Within this equivalence class, some of the problems are more equal than others. Although some problems may lie in this class, it is not guaranteed that they are solvable in practice. Recall that a Turing machine has unlimited storage space and unlimited time in which to run. However, we live in a universe of limited resources. An algorithm is as good as useless if it need 10 times the current age of the universe to execute, or needs more storage cells than there are basic particles in the universe. Still, if we were to draw an arbitrary line dividing useful from non-useful algorithms based on the physical size or length of time it requires, it would be a subjective matter[1].

There is also another problem. As computers get faster, we would have to admit more and more algorithms as useful. This all goes to show that placing a fixed limit on resources is not a very elegant way of solving the problem of classifying further the class of solvable problems.

[1]One may classify algorithms taking longer than a century to execute, but this didn't stop an alien race from building a computer which took 5 million years to execute its program to find the answer to life, the universe and everything in Douglas Adams' *Hitchhiker's Guide To The Galaxy*.

The solution is to analyze the inherent complexity of algorithms. You have already encountered complexity in previous courses. You may remember that whereas adding a new element to the head of a linked list takes only a constant amount of time (irrespective of the size of the list), sorting the list needs on the order of $n \log(n)$ time units (where n is the length of the list). So which problems of which complexity will be classified as inherently too complex?

Consider a problem with complexity of the order n, where n is the size of the problem. By doubling computer speed, in the same period of time, we can solve a problem of twice the previous size. What about a problem of size n^2. By doubling the speed, we can solve problems of up to 1.4 times the original size. But how about a problem of size 2^n? By doubling speed, we are only increasing the maximum size of the problem solvable in fixed time by 1. This shows an inherent difference between problems solvable in polynomial time to those solvable in exponential time: whereas by multiplying the speed of the underlying machine, the size of solvable polynomial problems increases geometrically, this only grows arithmetically for exponential problems.

There still is a question of what model of computing we are allowed to use. Consider the following problem:

Traveling Salesman (TS)

INSTANCE: Given a set of n cities C, a partial cost function of traveling from one city to another *cost* (such that $cost(c_1, c_2)$ is defined if there is a direct means of traveling from city c_1 to city c_2, and if defined is the cost of the traveling) and an amount of money m.

QUESTION: Is there a way of traveling through all the cities exactly once, returning back to the initial city, and in the process not spending more than the amount of money m?

Consider a non-deterministic Turing machine, which will simply fan-out trying all possible paths. Clearly, in at most n steps, each path will either succeed or fail. Hence, we will get a result in at most n steps. Hence, it seems that this problem is one of polynomial complexity. But non-determinism, involves, if you recall, an oracle function which tells us which path (if any) will yield results, or a machine with the ability to replicate itself for any number of times. This is not a very realistic means of computation. So what if we try to solve the problem on a deterministic Turing machine? If we enlist all the paths and try them one by one, in the worst case, we will need to examine all possible paths. This goes beyond the realms of polynomial complexity. You may try to find an algorithm to solve the problem on a deterministic time, but it is very improbable that you will manage to do so.

Hence, the rudimentaries of a hierarchy start to become apparent. We will define the class of problems solvable in polynomial time on a deterministic Turing machine to be P. The class of problems solvable in polynomial time on a non-deterministic Turing machine will be called NP (non-deterministically polynomial). Beyond this, lie the problems which need exponential time even on a non-deterministic Turing machine. There exist further classifications of this last class, but it is beyond the scope of this short part of the course to deal with these divisions.

Note that a number of problems have been shown to require exponential time on a deterministic Turing machine. We refer to such problems as *intractable* because of the difficulties we encounter as soon as we try to apply these algorithms to modest sized instance of the problem.

Before concluding this brief introduction, a couple of questions arise from the above text. First of all, does there exist a solution of the traveling salesman problem which uses only polynomial time on a deterministic machine? And secondly, why do we include this material together with language hierarchies in a single unit?

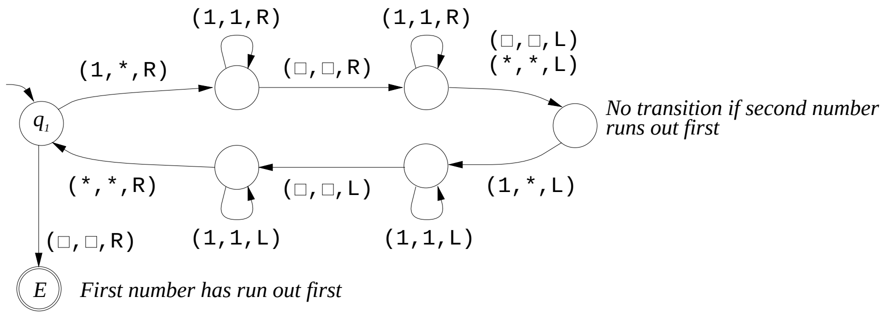
Let us start with the initial question. Before, we consider the traveling salesman, consider the following problem:

Less than or equal to (LTE)

INSTANCE: Given two numbers n_1 and n_2.

QUESTION: Is $n_1 \leq n_2$?

Consider the deterministic Turing machine which performs this operation. It will terminate and succeed if the answer to the question is positive, but terminate and fail otherwise. One such machine overwrites the numbers (one at a time) with a new symbol. If, the second runs out before the first, fail but terminate successfully otherwise:



Each right-left trip will take about $2(n_1+2)$ steps. This will have to be done for n_1 times. Hence, the machine terminates after at most $\alpha n_1^2 + \beta n_1 + \gamma$ steps. Since the machine is a deterministic one, and it takes only a polynomial amount of time (relative to the size of the input), this problem is in P. Clearly, every deterministic Turing machine can be seen as a non-deterministic one and thus, it can also be solved by a nondeterministic Turing machine in polynomial time and is hence also in *NP*.

How about the traveling salesman's problem? We have already given a rough idea of how to construct a non-deterministic Turing machine which computes the answer in polynomial time. Hence it is in *NP*. But is it also in P? This is probably one of the most sought after answers in mathematics today. Nobody has come up with an algorithm showing that it is in P, but, on the other hand, nobody has come up with a proof that this is impossible. The mystery is even more intriguing than this. There is no problem known to be in *NP* but not P.

In 1971, an important paper identified a problem, the satisfiability problem, and showed that it is the hardest problem in *NP*. Hence, if it is shown to be in P, so must all the other problems in *NP*, and $P =$*NP*. Further work done in these past 30 years has identified a number of other such hardest problems in *NP*. Amongst these is the traveling salesman problem.

The general consensus, is that $NP \neq P$ and hence the traveling salesman problem is intractable and not in P, but this still has to be proved.

As for the inclusion of this material together with language hierarchies, we note that we are basically using the Turing machines as language recognizers. We have a decision problem Π which, divides the set of possible inputs into two: the instances answered positively and those answered negatively, which are complements of each other. Hence, our questions are all dealing with the complexity of the language in question. Note that we are only dealing with recursive languages, since we assume that the Turing machine terminates with a yes or no answer. This provides us with yet another hierarchy of languages. Unfortunately, due to lack of time, we will not be able to compare this hierarchy with the Chomsky one.

8.1 Exercises

1. Show that the following problem is in P:

   Triangle (TRI)

   INSTANCE: Given three numbers.

   QUESTION: Can these three numbers be the lengths of the three sides of a triangle?

2. Show that the traveling salesman's problem but with distinct start and terminal cities is also in NP. We have, in passing, mentioned that the traveling salesman problem is the hardest problem in NP. In effect, if we show that the traveling salesman is in P, then so are all the other members of NP. Can you outline, how to show that the modified problem given here is also, in the same sense, the hardest NP problem?

Chapter 9

Definitions and Basic Results

9.1 Background

We will now start formally defining what we mean by the complexity of a problem. We will only be treating problems with a yes/no answer and for which there exists a terminating algorithm which gives the answer. The following definitions regarding complexity are based on languages. This may seem strange, but the concept of a yes/no decision problem is intimately related to languages.

Consider a problem Π. There is the set D_Π of strings which the problem accepts as input. Of these, the strings in the set Y_Π are those which give a positive answer whereas N_Π are those which give a negative answer. Clearly a Turing Machine which computes the answer to this problem is basically deciding whether the given input string in D_Π is in the language Y_Π. Hence, if we define the complexity of deciding whether a string lies in a particular language or not we can apply these results to decision problems in general.

The format in which we will present decision problems is as follows:

Primality test (PRT)

Instance: Given a number n.

Question: Is n a prime number?

The first line states the name of a decision problem (in this case, primality test) and a short version of the name which will be used as an abbreviation

(PRT). We then specify what an *instance* of the problem consists of. In this case it is a number n. This can be seen as the formal parametrisation of the input of the problem. Finally, we ask a yes/no question regarding the instance.

Our main task is to decide how much work has to go into showing that a particular instantiation of the input is in the language of positive answers or not. Obviously, certain inputs inherently require more work than others. For example, consider the problem of checking whether a given word is in a given dictionary. The problem becomes more 'complex' as the size of the dictionary increases. The complexity of the problem is thus to be given as a function of the size of the dictionary. This is not very convenient, since for every decision problem we must first isolate the inputs on which the problem complexity depends. However, there is a simpler solution which we will adopt. Given a reasonable encoding of the inputs, we will calculate the complexity of the problem as a function which, given the length of an input, gives the complexity of calculating the result. Consider the dictionary problem once more. We are now giving the complexity of computing the result in terms of, not just the dictionary size, but also the size of the word we are searching for. This makes sense, since every time we check whether a particular dictionary entry matches the searched word, we need to look through the symbols of the word to check for equality, and hence, the longer the word, the more work needs to be done.

Now consider the complexity of a particular algorithm to solve the dictionary problem. As you should already be aware, there is a variety of complexity (execution length) measures we can take. The ones we are usually interested in are the average and worst case length of execution. In this course we will be interested exclusively in the worst case analysis.

To classify a problem we use the big-O notation. Since it is difficult to calculate the exact number of execution steps of an algorithm (and it is dependent on the amount of effort put into making it more efficient), the big-O notation allows us to approximate an upperbound to a function. The formal definition is the following:

Definition: We say that a function $f(n)$ ($f : \mathbb{N} \to \mathbb{R}^+$) is $O(g(n))$, sometimes written as $f(n) = O(g(n))$ if there is a constant c, such that, beyond a particular value of n, $c\, g(n) \geq f(n)$:

$$\exists c : \mathbb{R}^+,\ n_0 : \mathbb{N} \cdot \forall n : \mathbb{N} \cdot (n \geq n_0 \Rightarrow c\, g(n) \geq f(n))$$

For example, you may recall that $7n^2 + n - 4$ is $O(n^2)$, since for non-zero n ($n_0 = 0$), $8n^2 \geq 7n^2 + n - 4$ ($c = 8$).

Similarly, $2^n + n^{100}$ is $O(2^n)$, since for any $n \geq 1000$, $2 \cdot 2^n \geq 2^n + n^{100}$.

Finally, there is still an ambiguity in the description we have given. We have stated that the input is reasonably encoded. What exactly do we mean by this?

Some encodings are obviously unreasonable. Consider a Turing machine which checks whether two given strings are equal. There is a simple algorithm which goes back and forth to check the symbols for equality. Assume that the first string is m symbols long. Clearly, it needs, at most, $m + 1$ runs to check each of the m symbols, and in each run it moves the head $m + 1$ spaces to the left and then m symbols to the right. Hence, we have a machine taking about $m(m+1+m)$. Since the total length of the input $n \geq m$ symbols long, the execution length is not larger than $n(2n + 1)$, which is $O(n^2)$. But now consider the encoding of the input w_1 and w_2 as $(w_1 \Box w_2 \Box)^{|w_1 w_2|}$. Using precisely the same algorithm but ignoring the extra symbols which we will not use, the analysis now yields a worst case taking $O(n)$. This is because we are using an unreasonable encoding.

Reverting back to Turing machines as language acceptors bypasses the problem very neatly. It is now clear that the two machines are accepting considerably different languages. The first machine accepted exactly those strings in $\{w \Box w \mid w \in \Sigma^*\}$ and rejected all those in $\{w \Box w' \mid w, w' \in \Sigma^*,\ w \neq w'\}$. On the other hand, the second machine is accepting strings in:

$$\{(w \Box w \Box)^{2|w|} \mid w \in \Sigma^*\}$$

but rejected strings of the form:

$$\{(w \Box w' \Box)^{|ww'|} \mid w, w' \in \Sigma^*,\ w \neq w'\}$$

Note that we *assume* that the input lies in the domain of the language.

But we eventually want to apply this knowledge to actual problems. There is no hard and fast way of defining what a reasonable encoding is. It is usually used to denote a mixture of conciseness (without unnecessary padding as was used in the example) and decodability (the ability to decode the the input into a usable format using an algorithm of $O(n^\alpha)$, where α is a constant).

9.2 Machine Complexity

Until now, we have only looked at Turing machines as a means of production of an output. We have not analyzed the length of such executions. This is what we set out to do in this section. The basic machine we will be looking at is a single (two-way) infinite tape machine. Furthermore, we will be looking at terminating Turing machines used to give a yes/no answer depending whether they terminated successfully or not (hence we are ignoring the output).

Definition: A terminating deterministic Turing machine is said to have complexity t (where t is a function from natural numbers to natural numbers) if, when given input x, the length of the derivation is not more than $t(|x|)$.

Note that we are looking at the worst case. Consider the list of all single character inputs x_1, x_2, $\ldots x_n$. Assume that they respectively take l_1, l_2,$\ldots l_n$ transitions to terminate. Then, the definition states that $t(1) \geq \max\{l_1,\ l_2, \ldots l_n\}$.

Here, we are talking about single tape Turing machines. Usually it is much more convenient to be able to use, at least, multiple tape machines in our constructions. However, the extra tapes may allow us to perform certain problems faster than on a single tape machines, and our results for the construction we give would not be comparable to the real complexity as defined above. To get around this problem, we prove a theorem which relates the complexity of a k-tape Turing machine with that of a single tape machine.

Theorem: Provided that $t(n) \geq n$, a k-tape Turing machine of complexity $t(x)$ can be simulated on a single tape Turing machine with complexity $O(t^2(x))$.

Proof: We give the construction of a single tape Turing machine which can simulate a multi-tape Turing machine, and then we analyze the complexity of the resultant machine.

The emulator will separate the contents of the k tapes by a special symbol $\star$ and will have extra symbols, such that the contents of the location of the heads will be marked by the symbol with a bar over it. Hence, the tape contents $\star a\bar{b}aa \star \square \star \bar{a} \star$ represents a 3 tape machine, where the first tape has the contents $abaa$ and the head lies on the b, the second is still unused and the third contains a single symbol over which the head resides.

The emulator, upon input $w = av$, will follow this program:

1. Write on the tape the initial configuration: $\star\bar{a}v \star (\Box\star)^{k-1}$.

2. To simulate a single move, the machine scans the input from the first $\star$ to the $(k+1)$th to note what is under the heads. Then a second pass is made to update the tapes accordingly.

3. If, at any point the machine tries to move a virtual head onto a $\star$, it will replace the position by a blank and shift the symbols in the required direction.

Clearly, the initial stage takes $O(n)$ steps. The second and third stage requires two scans and up to k shifts. Each scan and shift requires a number of steps of the order of the length of the strings stored on the tape. Clearly, these cannot be any longer than the length of the complete execution of the k-tape machine. Hence, every time the second and third stages are performed, they take $O(t(n))$. However, these will be performed up to $t(n)$ times, and hence the total complexity is:

$O(n) + t(n)O(t(n))$

Since we assumed that $t(n) \geq n$, this reduces to $O(t^2(n))$.

$\Box$

We could do the same with other extensions of Turing machines, but k-tape Turing machines are generally sufficient to allow clear descriptions of algorithms. However, in certain problems, such as the traveling salesman problem as defined in the previous chapter, we will be better off using non-deterministic Turing machines. The importance of these machines will be realized later, as we discuss such algorithms into more detail. For now, we will just give a definition of what we mean by the complexity of a non-deterministic Turing machine (the complexity of a multi-tape Turing machine followed naturally from that of a single tape machine, which is why we did not give it).

Definition: A terminating non-deterministic Turing machine is said to have complexity t (where t is a function from natural numbers to natural numbers) if, when given input x, the length of the longest branch of execution (whether it terminates successfully or not) is $t(|x|)$.

Since deterministic Turing machines are all-powerful, they can also simulate non-deterministic Turing machines. This is obviously done at a price. The following theorem relates the complexity of a non-deterministic Turing machine with that of a deterministic one emulating it.

Theorem: Provided that $t(n) \geq n$, a non-deterministic single tape Turing machine of complexity $t(n)$ can be simulated on a single tape deterministic Turing machine with complexity $2^{O(t(n))}$.

Proof: As before, we construct an emulator and then analyze its complexity. To emulate a non-deterministic Turing machine, we use a 4-tape Turing machine. The deterministic machine proceeds to enumerate all possible computations and executes then in sequence. The four tapes are used as follows:

1. The first tape stores the input string which it does not modify in any way.
2. The second tape is used as a working tape when emulating a particular instance of the non-deterministic machine.
3. The third tape is used to store the possible computations which we have already tried.
4. The last tape is used to hold a flag to remember whether there was a computation of the length currently being investigated which has not terminated.

The main problem is that of enumerating all possible computation paths. Consider the largest size of the sets in the range of P. This corresponds to the maximum number of possibilities we have at any point in the computation. Let this number be α. Clearly, if we enumerate the contents of the sets in the range of P, we can describe any computation with input w as a sequence of numbers in 1, 2, $\ldots \alpha$. At every stage, if $P(q, a)$ is the set of possibilities from p_1 to p_m, the next number i in the list will determine which p_i to follow. Obviously, not all strings over this alphabet are valid computations.

In particular, note that if a string is rejected, there is a maximum length beyond which all computation sequence strings over $\{1, \ldots \alpha\}$ will either reject the input (not necessarily using up all the computation sequence string), or give an invalid computation (in that a choice of a non-existent path is made). This number is the maximum depth at which the rejection occurs.

Now consider the following algorithm:

1. Copy the contents of tape 1 to tape 2.

2. Simulate the non-deterministic machine using the string on tape 3 to take decisions as to which path to follow. A number of possibilities can occur:

   If a decision is to be made about which path to follow but no more symbols remain on tape 3 jump to step 3.

   If an invalid computation occurs (the next symbol on tape 3 is larger than the set of next possibilities) or a rejecting configuration is encountered jump to step 4.

   If, however, an accepting configuration is encountered, accept the input and terminate.

3. Write a symbol $\star$ on tape 4.

4. Calculate the next string over $\{1, \ldots \alpha\}$ to be written on tape 3. These are to be produced in order of the length of the string (and alphabetically for strings of the same length).

   If it is longer than the previous string then check tape 4. If there is a blank symbol (no $\star$ written on the tape), then all the paths were either invalid, or produced rejecting computations. Terminate and reject. If there is a $\star$, then there must have been some, as yet unterminated computations and we need to check computations of a longer length. Clear tape 4.

   Jump to instruction 1.

It should be clear (although not necessarily easily provable) that this emulator terminates and accepts exactly when the original machine did so, and terminates and rejects when the non-deterministic machine did so.

For an input of length n, the maximum depth of the computation tree is $t(n)$. At every stage, there are at most α siblings of a particular node. The total number of nodes, thus cannot exceed $\alpha^{t(n)}$ and is thus $O(\alpha^{t(n)})$. The time taken to travel down the tree path to a node does not take more than $O(t(n))$ steps. Hence, the running time of the resultant machine is $O(t(n))O(\alpha^{t(n)})$. This is $2^{O(t(n))}$. But the machine we used had 4 tapes. By the previous lemma, this can be simulated on a single tape machine with complexity $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

$\square$

Note that this theorem does not say that it is impossible for a deterministic Turing machine to emulate a non-deterministic one in less than exponential time, but that we know of an algorithm which needs this amount of time. Hence, we have established an upperbound for the complexity of this simulation.

9.3 Complexity Classes

9.3.1 Class P

Clearly, we can classify languages by the complexity of a Turing machine needed to decide the language. As already indicated before, we will classify together all languages which need a polynomial complexity single tape deterministic Turing machine to decide them. This is the class of languages P.

Definition: P is the class of languages decidable in polynomial time on a deterministic single tape Turing machine:

$$P \stackrel{def}{=} \{L \mid L \text{ decidable by deterministic } T \text{ such that complexity of } T \text{ is } O(n^k) \text{ for some } k \in \mathbb{N} \}$$

At this point it is worth reviewing the definition of what it means for a language L to be decidable by Turing machine M. It means that M terminates on all inputs, successfully if the input is in L, but failing if it is not. It is clear, thus, that we are restricting our analysis to recursive languages.

We have already shown, for example, that checking whether a given word is in a given dictionary is a problem in P. We will give another example before going on to further classes.

Example: Consider the following problem:

Directed path (PATH)

INSTANCE: Given a directed graph G and two nodes n_1 and n_2.

QUESTION: Is there a path from n_1 to n_2?

We will show that $PATH \in P$, by constructing a polynomial time algorithm working on a deterministic Turing machine which solves the problem.

1. Mark node n_1.
2. For every edge (a, b) such that a is marked, mark also b.
3. If some new nodes have been marked, go back to 2.
4. Accept if n_2 is marked, reject otherwise.

The first and last steps are executed just once (and take only, at most $O(n)$ steps to locate n_1 and n_2). Step 2 scans every edge, checks the first node and may mark the second. This can be easily done in polynomial time on a deterministic machine (scanning every edge takes $O(n)$ and in every case checking the first node and possibly marking the second will only take $O(n)$, hence taking $O(n^2)$). Finally step 2 is repeated for a number of times. Consider the count of marked nodes which starts at 1 and increases every time step 2 is executed. Clearly, this cannot go on more than the number of nodes we have. Hence, the maximum number of repetitions is $O(n)$. Hence, the total complexity is:

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

Hence, $PATH \in P$.

$\square$

Theorem: The class of languages recognizable by k-tape Turing machines in polynomial time is P.

Proof: Let M be a k-tape Turing machine such that it is decidable in $O(n^\alpha)$ ($\alpha \in \mathbb{N}$) steps. By the theorem relating multi-tape Turing machines with single tape ones, we have that it is decidable in $O(n^{2\alpha})$ on a single tape machine. Hence, it is in P. Note that in the case of $\alpha = 0$, if $f(n) = O(1)$ then $f(n) = O(n)$. Hence we can still apply the emulation theorem.

Conversely, if $L \in P$ then there is a single tape deterministic Turing machine which decides L. But we can run M on a multi-tape machine which ignores all but the first tape. This emulation takes exactly as many steps as the original machine. Hence, L is decidable in polynomial time on a k-tape machine.

$\square$

9.3.2 Class *NP*

The theorem which constructed a non-deterministic Turing machine emulator running on a deterministic Turing machine, stated that we can perform this emulation approximately in the exponent of the original time taken by a non-deterministic machine. It does not state that there is not a more efficient emulation. However, if there is one, we do not yet know about it. This means that the class of languages decidable on a non-deterministic Turing machine in polynomial time is not necessarily the same as P. With this in mind, we define the class *NP*:

Definition: *NP* is the class of languages decidable in polynomial time on a non-deterministic single tape Turing machine:

$$\begin{aligned} NP \stackrel{def}{=} \{L \quad | \quad & L \text{ decidable by non-deterministic } T \text{ such that} \\ & \text{complexity of } T \text{ is } O(n^k) \text{ for some } k \in \mathbb{N}\ \} \end{aligned}$$

NP stands for *non-deterministically polynomial*.

The first thing we note is that class P is contained in class *NP*:

Theorem: $P \subseteq NP$

Proof: The proof is trivial. Every deterministic Turing machine can be seen as a non-deterministic Turing machine with no choices. Hence, the time complexity of a deterministic Turing machine is the same as when interpreted as a non-deterministic Turing machine. Thus, all polynomial time deterministic machines run in polynomial time on a non-deterministic machine and therefore $P \subseteq NP$.

□

Example: Let us look at a problem encountered on a multiprocessor system:

Scheduler (SCH)

INSTANCE: Given a set T of tasks, a time cost function c associating a positive cost with every task, a number of processors p and a deadline D

QUESTION: Can the tasks be scheduled in the given deadline? Or more precisely, is there a partition of T, $T_1 \ldots T_n$ such that:

$$D \geq \max\{\sum_{t \in T_i} c(t) \mid 1 \leq i \leq n\}$$

We prove that $SCH \in NP$. Consider the non-deterministic Turing machine which goes through all the tasks assigning each one to a processor non-deterministically. At the end, the total time taken is calculated and checked against D. If it larger than D, the machine terminates and fails, otherwise it terminates and succeeds. Clearly, the algorithm always terminates. If there is any assignment which will work it will be tried and the machine will succeed and accept the input, but if there are none, the machine fails and rejects the input. The assignment of tasks to processors takes $O(n)$ steps. Adding two numbers can be done in polynomial time and thus so can the adding of $O(n)$ numbers. Finding the maximum of a set of p elements is also $O(n)$ as is comparison of numbers. Hence, the total time taken by any path is $O(n^\alpha)$ for some particular value of α. Hence, we have constructed a polynomial time non-deterministic Turing machine which decides *SCH*. Hence $SCH \in NP$.

□

There is a particular property you may have noticed, which this problem enjoys. If we find a solution, it is very easy to show that it is, in fact correct. As we have already said, we can calculate whether a given assignment of tasks to processors satisfies the deadline constraint in polynomial time on a deterministic machine. Even in the traveling salesman case, we can count the values on the claimed paths and compare the result to the upper bound in polynomial time on a deterministic machine. This is the concept of polynomial verifiability. Basically, a problem is polynomial verifiable if, given an answer, we can easily verify its correctness. This can be formalized into:

Definition: An algorithm M is said to be a verifier of a language L when $w \in L$ if and only if there exists a string c, such that M accepts (w, c).

c is sometimes called a certificate, and it is left out in the analysis of the complexity of M.

The idea is that M accepts strings if and only if they are followed by an instance proving their membership in L. In the case of the traveling salesman, a verifier V would be an algorithm which, for every instance of the problem, V accepts the instance exactly when we can add extra information to the input.

A Turing machine solving the problem from scratch is a special case of a verifier where $c = \varepsilon$ in all cases. However, we started discussing verifiers because we know that the solution we gave for the traveling salesman problem is not very efficient, even though we know that we can verify an answer quickly. This can be generalized into a surprising result:

Theorem: A language L is in *NP* if and only if it has a verifier in P.

Proof: Assume $L \in$ *NP*. We need to show that it has a polynomial time verifier (in P). Consider the following algorithm V working on input (w, c):

1. Simulate the non-deterministic machine with input w and using c as the non-deterministic path to take (as in the deterministic emulation of non-deterministic machines).

2. If this path of the computation accepts, accept the input. Otherwise, reject.

Clearly, there is a string c such that (w, c) is accepted if and only if $w \in L$. Also, the machine runs in polynomial time. Hence L has a verifier in P.

Conversely, let L have a verifier in P, namely V. Assume that V has time complexity n^α. Now consider the following non-deterministic machine M, working on input w:

1. Branch (non-deterministically) to all strings c of length n^α.

2. Run V on (w, c).

3. If V accepts, accept, otherwise reject.

Obviously, V can never look at strings longer than n^α. Hence, if $w \in L$ then there is a string c such that $|c| \leq n^\alpha$ such that V accepts (w, c). Clearly such a path would exist on M and thus the string would be accepted. On the other hand, if $w \notin L$, there is no c such that V accepts (w, c). Thus there is no path in M accepting w, and hence it is rejected.

Also, since V has polynomial time, so has M, and thus $L \in NP$.

$\square$

Thus, whenever we want to show that a given decision problem is in NP we now have two approaches we can take. We can either stick to the definition and show that there is a non-deterministic machine which computes the answer in polynomial time, or we may show that it is easy to check whether a given answer is correct or not. In most cases, both approaches are quite easy to use. The result is more important from a conceptual point of view. The class NP can be characterized in two alternative ways, neither of which clearly indicates whether there are problems in NP which are not in P. This is the question we will be tackling for the rest of this course, but for which, unfortunately, will not be able to furnish an answer.

9.4 Reducibility

A psychologist analyzing the mathematical mind asked the following question to a mathematician:

You are walking down a street where you see a burning house, a fire hydrant, a box of matches and a hosepipe. What do you do?

After careful thought, the mathematician answered:

I simply connect the pipe to the hydrant, turn on the water and extinguish the fire.

Satisfied by the answer, the psychologist asked another question:

You are now walking down a street where you see a house, a fire hydrant, a box of matches and a hosepipe. What do you do?

Without even pausing for thought, the mathematician answered:

I light a match, set the house alight, and it has now been reduced to a previously solved problem.

In mathematics, the concept of reducing a problem into a special case of an already known one is a very important technique. The area of algorithm complexity is no exception and in this section we will define what it means for a problem to be reducible to another and how this technique can be used effectively.

Consider the following problem:

Transitive acquaintance (TA)

INSTANCE: Given set of people, a list of who knows whom and two particular persons p_1 and p_2.

QUESTION: Is p_1 a transitive acquaintance of p_2?

We say that p_1 is a transitive acquaintance of p_2 if, either p_1 is a acquaintance of p_2 or there is an intermediate person p such that p_1 is a acquaintance of p and p is a transitive acquaintance of p_2. Is this problem in P?

With a little bit of insight we realize that this problem is simply a rewording of the *PATH* problem. If we label the nodes by people's names, and make two edges (a, b) and (b, a) for every pair of people a and b who know each other (assume that acquaintance is mutual), we can run the *PATH* algorithm and solve the problem in polynomial time.

Since we can build the graph in polynomial time, and $PATH \in P$, we immediately know that $MA \in P$. How can we formalize this concept of a problem being reducible to another?

Definition: A language L_1 is said to be polynomially reducible to another language L_2, written as $L_1 \preceq_P L_2$, if there is a total function $f : L_1 \rightarrow L_2$ which is computable in polynomial time on a deterministic Turing machine, such that

$$w \in L_1 \Leftrightarrow f(w) \in L_2$$

To show that $TA \preceq_P PATH$ we need to define function f, which given an instance of *TA* gives an instance of *PATH*. This function, is the one already defined: $f(\langle P, A\rangle) = \langle P, A \cup A^{-1}\rangle$ (where $\langle P, A\rangle$ is an instance of the acquaintance problem — P is the set of people and A the set of acquaintances, and $\langle N, E\rangle$ is an instance of *PATH* if N is the set of nodes and E the set of edges). This function can clearly be calculated in polynomial time since it requires only copying.

Since $PATH \in P$ and $TA \preceq_P PATH$ we expect to be able to conclude that $TA \in P$. This is proved in the following theorem:

Theorem: If $L_1 \preceq_P L_2$ and $L_2 \in P$ then $L_1 \in P$.

Proof: Simply compute function f, taking $O(n^\alpha)$ steps, then execute the algorithm deciding L_2, taking $O(n^\beta)$. Now $w \in L_1$ if and only if $f(w) \in L_2$. Hence this algorithm decides L_1. Its complexity is $O(n^\alpha) + (n^\beta) = O(n^{\max(\alpha,\beta)})$ and L_1 is hence in P.

$\square$

Note that the function need not be as simple as the one for *TA*. Consider the following extension of the problem:

Country acquaintance (CA)

INSTANCE: Given set of people and their nationality, a list of who knows whom and two distinct countries c_1 and c_2.

QUESTION: Is there a person from c_1 who is a transitive acquaintance of a person from c_2?

We will show that $CA \preceq_P TA$, by giving a function which maps any instance of *CA* into an instance of *TA*.

We start by building a graph as we did in *TA*. Now we add a person p_1 such that p_1 knows all people from c_1 and nobody else. Similarly, we add p_2 for c_2. Clearly, this means $O(n)$ more edges. We now ask the *TR* question on p_1 and p_2. This function is computable in polynomial time. Now, if some person q_1 in c_1 transitively knows some person q_2 in c_2, then p_1 knows q_1 who transitively knows q_2 who knows p_2. Hence p_1 knows p_2. Conversely, if p_1 knows p_2, then p_1 knows somebody q_1 (who must be from c_1) who transitively knows p_2. But p_2 knows only people of c_2 (which is different from c_1) and hence there must be a person q_2 from c_2 who knows p_2 and such that q_1 transitively knows q_2. Hence there exist q_1 from c_1 and q_2 from c_2 who are transitive acquaintances. Hence, $w \in CA \Leftrightarrow f(w) \in TA$ and therefore $CA \preceq_P TA$. But $TA \in P$ and hence $CA \in P$.

Before we conclude this section we give a few properties of the reducible relation.

Proposition: The reducible relation ($\preceq_P$) is reflexive and transitive. Also $P \times NP \subseteq \preceq_P$.

9.5 *NP*-completeness

The main use of reducibility, is however not in P, but in NP. The following theorem can easily be proved:

Theorem: If $L_1 \preceq_P L_2$ and $L_2 \in NP$ then $L_1 \in NP$.

Reducibility is basically a measure of how hard a problem is. When we say that $L_1 \preceq_P L_2$, we are basically saying that (ignoring any 'insignificant' polynomial delays) deciding L_2 is at least as hard as deciding L_1. Is there a problem which is the hardest in NP?

Assume there is, and let us call the problem X. Thus, for any language $L \in NP$, $L \preceq_P X$. Now assume that somebody manages to show that $X \in P$. By the theorem we have proved earlier, $L \preceq_P X$ and $X \in P$ implies that $L \in P$. Hence, the whole of NP would collapse to P. Therefore, if such a problem exists, researchers can concentrate exclusively on whether $X \in P$. There turns out to be not just one, but a whole family of such problems. We call this family of functions NP-complete and it is defined as follows:

Definition: A language L is said to be NP-complete if:

- $L \in NP$
- For every language $L' \in NP$, $L' \preceq_P L$.

The property that if we find an NP-complete problem to lie in P then the whole of NP would collapse to P, can now be proved.

Theorem: If NP-complete $\cap\, P \neq \emptyset$ then $P = NP$.

Proof: Recall that if $A \preceq_P B$ and $B \in P$ then $A \in P$.

Assume NP-complete $\cap\, P \neq \emptyset$ and let X be one element from this set. Consider any $Y \in NP$. By the definition of NP-completeness, $Y \preceq_P X$. But $X \in P$. Hence $Y \in P$. Therefore $NP \subseteq P$. We also proved that $P \subseteq NP$ and therefore $P = NP$.

$\square$

Assume we know at least one NP-complete problem X. If we find an NP hard problem Y such that X is reducible to Y, then Y is at least as hard as

X and Y should therefore also classify as 'the hardest problem in *NP*'. The following theorem proves this result.

Theorem: If L is *NP*-complete and $L \preceq_P L'$ (where $L' \in NP$) then L' is also *NP*-complete.

Proof: To prove that L' is *NP*-complete, we need to prove two properties: that it is in *NP* and that all *NP* hard problems are reducible to L'. The first is given in the hypothesis. Now consider any *NP* hard problem X. Since L is *NP*-complete, we know that $X \preceq_P L$. But we also know that $L \preceq_P L'$. Hence, by transitivity of $\preceq_P$, we get $X \preceq_P L'$.

$\square$

We will now end this chapter by the grand proof of this part of the course. Originally given in a pioneering paper by Cook in 1971, this theorem gives us a first *NP*-complete problem which we can then use to prove that other problems are *NP*-complete.

9.6 Cook's Theorem

Before we state the theorem and proof, we need some background information about Boolean logic, which is the domain of the problem given in Cook's theorem.

Given a set of Boolean variables V, we call a total function $s : V \to \{T, F\}$ a *truth assignment*. We say that variable v is true under s if $s(v) = T$. Otherwise we say that v is false under s.

Literals can be either a variable, or a variable with a bar over it $\bar{v}$ (corresponding to *not* v). We can extend s to give the truth value of literals in the following manner: given a variable v, $s(\bar{v}) = T$ if $s(v) = F$, otherwise it is equal to F.

A set of literals is called a *clause* (corresponding to disjunction). The interpretation of s can be extended once more to get:

$$s(C) = \begin{cases} T & \text{if } \exists c \in C \cdot s(c) = T \\ F & \text{otherwise} \end{cases}$$

Finally, a set of clauses is said to be satisfiable under interpretation s if for every clause c, $s(c) = T$.

We can now state Cook's theorem.

Theorem: $SAT \in NP$-complete, where SAT is defined as:

Satisfiability (SAT)

INSTANCE: Given a set of variables V and a set of clauses C.

QUESTION: Is there a truth assignment s satisfying C?

Comment: If you recall Boolean logic, a collection of clauses is effectively an expression in disjunctive normal form. Also, all boolean expressions are expressible in disjunctive normal form. Hence, we are effectively showing that checking whether a general boolean expression can be satisfied is *NP*-complete.

Proof: That $SAT \in NP$ is not hard to prove. Consider the problem of deciding whether a given interpretation satisfies C. For every variable $v \in V$ we pass over C to replace literals v and $\bar{v}$ by T or F. Another pass over C replaces clauses by T or F. Finally a last pass is needed to check whether each clause has been satisfied or not. Clearly, this takes polynomial time. Effectively, we have constructed a polynomial verifier for SAT. This guarantees that $SAT \in NP$. (alternatively we could have gone on to construct a non-deterministic Turing machine which tries all possible truth assignments at once.

We now need to prove that all problems in *NP* can be reduced to SAT. Consider $L \in NP$. We need to prove that $L \preceq_P SAT$. What do we know about L? The property defining *NP* guarantees that there is non-deterministic Turing machine which decides L in polynomial time. What we thus try to do, is transform a non-deterministic Turing machine into an instance of SAT. If this can be done in polynomial time, we can transform L into an instance of SAT via the non-deterministic Turing machine which recognizes L, whatever L might be.

Let M be a terminating non-deterministic Turing machine such that $M = \langle K, \Sigma, T, q_1, P, F \rangle$. We now set up a set of Boolean clauses such that they can be satisfiable if and only if an input string is accepted by M. We then show that this construction is possible in polynomial time.

What boolean variables do we need? We need to remember the following information:

- In what state is M at time t? If we enumerate the states starting from q_1 up to $q_{|K|}$, we can have variables ${}_tQ_i$ which is true if and only if M is in state q_i at time step t.
- Which tape square is the head scanning at time t? We will use ${}_tH_i$ which is set to true if and only if the head is scanning square i at time t.
- What are the contents of the tape? If we enumerate the tape symbols as σ_1 to $\sigma_{|\Sigma|}$, we can use variables ${}_tS[i]_j$ which is true if and only if square i has symbol j at time t.

But we have to place bounds on the subscripts of the above variables. Recall that M has polynomial complexity $p(n)$. In this time it cannot use any tape square not in $S[-p(n)] \ldots S[p(n)]$. Similarly, the time variables cannot exceed $p(n)$. The result is that each subscript has $O(p(n))$ possible values. Hence, we have $O(p^2(n)) + O(p^2(n)) + O(p^3(n))$ variables which is $O(p^3(n))$ — a polynomial in n.

Now, we will give a set of clauses C which can be satisfied if and only if M accepts the input in at most $p(n)$ steps. Hence:

$$\begin{array}{ll} & w \in L \\ \Leftrightarrow & M \text{ accepts } x \\ \Leftrightarrow & M \text{ accepts } x \text{ in at most } p(n) \text{ steps} \\ \Leftrightarrow & C \text{ can be satisfied} \end{array}$$

We now set out to define the clauses. We will have four sets of clauses. Each will serve a specific function:

1. Machine invariants, split into three subclasses:

   (a) M can only be in one state at a time.

(b) The head can only be at one position at a time.

(c) Each square can only contain one piece of information at a time.

2. Initialization of the machine.

3. Application of P to calculate the next state, position and tape contents.

4. Termination of the machine.

We now define the clauses.

1. Machine invariants:

   (a) One state at a time: The machine has to be in at least one state at any time t. For t ranging over: $0 \leq t \leq p(n) \ldots$

   $$\{_tQ_1,\ _tQ_2 \ldots _t Q_{|K|}\}$$

   But there may be no more than one state which is true. This is expressible as $\neg(_tQ_i \wedge_t Q_j)$ where $i \neq j$. In disjunctive form, this appears as:

   $$\{_t\bar{Q}_i,\ _t\bar{Q}_j\}$$

   where $0 \leq t \leq p(n)$ and $1 \leq i < j \leq |K|$.

   Note that we $O(|K|^2)$ of these clauses. Hence, the encoding of the clauses of this type is polynomial in size.

   (b) The head is at one place at a time: This is almost identical to the previous case:

   $$\{_tH_{-p(n)}, \ldots _t H_{p(n)}\}$$

   where $0 \leq t \leq p(n)$ and $\ldots$

   $$\{_t\bar{H}_i,\ _t\bar{H}_j\}$$

   where $0 \leq t \leq p(n)$ and $-p(n) \leq i < j \leq p(n)$.

   As before, the encoding of these clauses only takes $O(p^2(n))$, which is a polynomial.

(c) Tape squares can contain only one piece of information at a time:

$$\{{}_tQ[s]_{-p(n)}, \ldots {}_t\, Q[s]_{p(n)}\}$$

where $0 \leq t \leq p(n)$ and $1 \leq s \leq |\Sigma|$ and ...

$$\{{}_t\bar{Q}[s]_i,\ {}_t\bar{Q}[s]_j\}$$

where $0 \leq t \leq p(n)$, $1 \leq s \leq |\Sigma|$ and $-p(n) \leq i < j \leq p(n)$.

As before, the encoding of these clauses only takes polynomial space.

2. Initialization: Initially, the head is on the 0th position, M is in the initial state q_1 and the tape contains the input string:

$$\begin{array}{rl} \{{}_0H_0\} & \\ \{{}_0Q_1\} & \\ \{{}_0S[-p(n)]_1\} & \quad (\sigma_1 = \Box) \\ \vdots & \\ \{{}_0S[-1]_1\} & \\ \{{}_0S[0]_i\} & \quad \text{where the first symbol of the input is } \sigma_i \\ \vdots & \\ \{{}_0S[n-1]_j\} & \quad \text{where the last symbol of the input is } \sigma_j \\ \{{}_0S[n]_1\} & \\ \vdots & \\ \{{}_0S[p(n)]_1\} & \end{array}$$

Again, notice that we have $O(p(n))$ clauses.

3. Calculation of next state: Consider a transition instruction $(q_j, \sigma_m, S) \in P(q_i, \sigma_l)$. In boolean logic terms we want to say that: for any time t, if the state is q_i and the information at the current head position is σ_l, then we want to leave the head where it is, but change the information at the current position to σ_m and change the state to q_j.

$$({}_tH_s \wedge_t S[s]_l \wedge_t Q_i) \Rightarrow$$
$$({}_{t+1}H_s \wedge {}_{t+1}Q_j \wedge {}_{t+1}S[s]_m)$$

With some simple manipulation, we can get them into the right format:

$$\{{}_t\bar{H}_s,\ {}_t\bar{S}[s]_l,\ {}_t\bar{Q}_i,\ {}_{t+1}H_s\}$$
$$\{{}_t\bar{H}_s,\ {}_t\bar{S}[s]_l,\ {}_t\bar{Q}_i,\ {}_{t+1}Q_j\}$$
$$\{{}_t\bar{H}_s,\ {}_t\bar{S}[s]_l,\ {}_t\bar{Q}_i,\ {}_{t+1}S[s]_m\}$$

where $0 \leq t \leq p(n)$, $-p(n) \leq s \leq p(n)$. We can construct similar triplets for instructions which move to the left or to the right.

The rest of the tape contents must not change.

$$({}_t\bar{H}_s \wedge_t S[s]_i) \Rightarrow_{t+1} S[s]_i$$

This can be converted to:

$$\{{}_t\bar{S}[s]_i,\ {}_tH_s,\ {}_{t+1}S[s]_i\}$$

where $0 \leq t \leq p(n)$, $-p(n) \leq s \leq p(n)$ and $1 \leq i \leq |\Sigma|$.

If we count the number of clauses, we still have a number proportional to a polynomial of the input size.

4. Termination: Once, a program terminates (before $p(n)$ steps), we must keep the same state and leave the current position unchanged. If $P(q_i, \sigma_j) = \emptyset$, we add:

$$({}_tQ_i \wedge_t H_s \wedge_t S[s]_j) \quad \Rightarrow \quad ({}_{t+1}H_s \wedge_{t+1} S[s]_j \wedge_{t+1} Q_i)$$

This can be written as:

$$\begin{array}{r}\{{}_t\bar{Q}_i,\ {}_t\bar{H}_s,\ {}_t\bar{S}[s]_j,\ {}_{t+1}H_s\}\\ \{{}_t\bar{Q}_i,\ {}_t\bar{H}_s,\ {}_t\bar{S}[s]_j,\ {}_{t+1}S[s]_j\}\\ \{{}_t\bar{Q}_i,\ {}_t\bar{H}_s,\ {}_t\bar{S}[s]_j,\ {}_{t+1}Q_i\}\end{array}$$

where $0 \leq t \leq p(n)$, $-p(n) \leq s \leq p(n)$.

Finally we want the set of clauses to be satisfiable only if we end up in a final state. Hence for every $q_i \in F$, we add:

$$\{{}_{p(n)}Q_i\}$$

The size of these clause again does not exceed a polynomial function of the input size.

We now have constructed a collection of clauses which, upon input w, can be satisfied if and only if M accepts w. Hence, for any language $L \in NP$ we have defined a function f_L which transforms the language (or rather the non-deterministic machine accepting the language in polynomial time) into an instance of *SAT*. Also $w \in L \Leftrightarrow f(w) \in SAT$. Hence, if we can show that f_L can be computed in polynomial time, we have shown that for any $L \in NP$, $L \preceq_P SAT$ and thus *SAT* is *NP*-complete.

Note that once we know L (and hence M), constructing f_L consists of nothing but replacing values into the long formulae given above. It is therefore polynomial as long as we do not have to produce output which is more than polynomial in length. But we have seen that each subdivision of clauses in at most a polynomial of n in length. Hence, so is their sum. Therefore f_L has polynomial complexity.

$\square$

9.7 Exercises

1. Show that the following problem is in P:

   Shortest path (SP)

   INSTANCE: Given a directed graph G, cost function c associating with each edge a numeric value (> 0), two nodes n_1 and n_2 and a bound $B \in \mathbb{N}$.

   QUESTION: Is there a path in G from n_1 to n_2 costing less than B?

   Use the answer to prove that CkA, as defined below, is also in P.

   Given an acquaintance relation A and a number k, we say that p is a k-acquaintance of q if $(p, q) \in \bigcup_{i=1}^{k} A^k$. In other words, there is a list of up to $k - 1$ persons r_1 to r_i such that p knows r_1 who knows $r_2 \ldots$ who knows r_i who knows q.

   Country k-acquaintance (CkA)

   INSTANCE: Given set of people and their nationality, a list of who knows whom, two distinct countries c_1 and c_2 and a bound k.

   QUESTION: Is there a person from c_1 who is a k-acquaintance of a person from c_2?

   As an aside, it is interesting to note that the minimum k such that everybody on Earth is k-acquainted to everybody else is very low — possibly as low as 10. This is because of the hierarchical way our society is organized. Everybody in the world knows somebody important in their village who knows somebody important in their region who know somebody important in the country who knows the official head of state. Since the heads of states have met quite a few other heads of states (and are hence acquaintances), you can then follow down any other person on earth through their head of state!

2. Show that the following problem is in NP:

   Hamiltonian Path (HAM)

INSTANCE: Given a graph G.

QUESTION: Is there a circular path going through all the nodes in G but without going through any more than once?

3. Consider the following single player game: For a particular value of $\alpha \in \mathbb{N}$, the game is played on a square board made up of $n \times n$ squares (where $n^2 > \alpha$). Initially α pieces are placed on the board. The player makes moves, each of which consists of a piece jumping over an adjacent piece vertically or horizontally onto an empty square. The piece jumped over is removed. The aim of the game is to end up with only one piece.

   α-Solitaire (αSOL)

   INSTANCE: Given a value n such that $n > \alpha$ and an initial position I.

   QUESTION: Is the game winnable from position I?

   Prove that, for a constant α, $\alpha SOL \in P$.

   Hint: You can show that $\alpha SOL \preceq_P PATH$. Start by informally showing that checking whether B' is a 'valid next position' of B can be calculated in polynomial time.

4. Cook's theorem points out one problem which can be labeled as 'the hardest problem in *NP*'. However, the definition of the classes P and NP themselves state one other problem which obviously enjoys the same status as the satisfiability problem. Can you point out what the problem is, and give a simple proof showing that it is so?

   Hint: Take a look at the note after the theorem relating deterministic with non-deterministic Turing machine complexity.

Chapter 10

More *NP*-Complete Problems

If we were to use a technique similar to the one used in Cook's theorem for the satisfiability problem, there probably would not be far too many results showing problems to be *NP*-complete. As it happens, we have already identified a technique which we can use to prove that other problems are *NP*-complete. Given a problem Π, we can show that it is *NP*-complete if we can show that it has two properties:

- $\Pi \in NP$.
- $X \preceq_P \Pi$ (where X has already been shown to be *NP*-complete).

In other words we have to show that a polynomial time non-deterministic Turing machine can solve Π and that for any instance of X, we can transform it into an instance of Π in polynomial time, such that one is true if and only if the other is true. We are then justified in claiming that Π is *NP*-complete. This result has already been proved in the previous chapter. Since, at this point we only know of SAT to be *NP*-complete, we have no choice for X, but as we gather more and more *NP*-complete problems, we have a wider repertoire which we can use to show that a new problem is *NP*-complete.

This chapter proves the *NP*-completeness of a number of important standard problems. Some of these results are followed up by a number of minor problems which can be tackled using the main result. The idea is to get you acquainted with a number of standard results and how they have been proved and also to give you the skill needed to prove that other problems are also *NP*-complete.

10.1 3-Satisfiability

A restricted version of the satisfiability problem is the case when all the clauses are exactly 3 literals long. We will show that this problem is also *NP*-complete by reducing *SAT* into it.

3-Satisfiability (3SAT)

INSTANCE: Given a set of variables V and a set of clauses C each of which is 3 literals long.

QUESTION: Is there an assignment of V which satisfies C?

Theorem: *3SAT* is *NP*-complete.

Proof: We show that $3SAT \in NP$ and that $SAT \preceq_P 3SAT$ to conclude the desired result.

- $3SAT \in NP$: We have already shown that *SAT* is in *NP*. Hence there is a non-deterministic polynomial Turing machine which decides *SAT*. But *3SAT* is just a restricted version of *SAT* and we can thus still use the same machine. Since such a machine exists, we conclude that $3SAT \in NP$.

- $SAT \preceq_P 3SAT$: We have to show that every instance of *SAT* can be reduced into an instance of *3SAT*.

  Consider a clause which is only two literals long: $\{\alpha, \beta\}$ (corresponding to $\alpha \vee \beta$). Clearly, this can be satisfied exactly when $\{\alpha, \beta, z\}$ can be satisfied whatever the value of z. How can we express this?

  $$\{\ \{\alpha, \beta, z\},\ \{\alpha, \beta, \bar{z}\}\ \}$$

  Using the definition of satisfaction, we can easily prove the equivalence of the two sets of clauses.

  In the case of 1 literal-long clauses, we use a similar trick to transform $\{\alpha\}$ into

  $$\{\ \{\alpha, z, z'\},\ \{\alpha, \bar{z}, z'\},\ \{\alpha, z, \bar{z}'\},\ \{\alpha, \bar{z}, \bar{z}'\}\ \}$$

  What about clauses with more than three literals? Let us first consider the case with four literals. We note that the clause $a \vee b \vee c \vee d$ can be satisfied exactly when $(a \vee b \vee z) \wedge (\bar{z} \vee c \vee d)$ can be satisfied. This can be shown by proving that:

$$\begin{array}{rl} & \exists\, a, b, c, d \cdot a \vee b \vee c \vee d \\ \Leftrightarrow & \exists\, a, b, c, d, z \cdot (a \vee b \vee z) \wedge (\bar{z} \vee c \vee d) \end{array}$$

Hence, we can transform $\{a, b, c, d\}$ into $\{\ \{a, b, z\},\ \{\bar{z}, c, d\}\ \}$.

But what about longer clauses? The answer is to apply the rule for 4 literals repeatedly. Thus, in the case of 5 literals we have: $a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5$ which can be satisfied exactly when $(a_1 \vee a_2 \vee z_1) \wedge (\bar{z} \vee a_3 \vee a_4 \vee a_5)$ can be satisfied. We can now apply the 4-literal clause rule again to the second clause and get that the original 5-literal clause is equivalent to:

$$\begin{array}{rl} & (a_1 \vee a_2 \vee z_1) \\ \wedge & (\bar{z}_1 \vee a_3 \vee z_2) \\ \wedge & (\bar{z}_2 \vee a_4 \vee a_5) \end{array}$$

In general we get $\{a_1, \ldots a_n\}$ transformed into:

$$\{\ \{a_1, a_2, z_1\},\ \{\bar{z}_{n-3}, a_{n-1}, a_n\}\ \}\ \ \cup\ \ \{\{\bar{z}_i, a_{i+2}, z_{i+1}\} \mid 1 \leq i \leq n-4\}$$

We have thus defined a function f, which, given an instance in SAT, produces an equivalent instance of *3SAT*. The informal reasoning above should convince you that an instance x of SAT can be satisfied if and only if $f(x)$ can be satisfied in *3SAT*, or $x \in SAT \Leftrightarrow f(x) \in 3SAT$.

But is the transformation f polynomial? As in the case of SAT we notice that an algorithm that performs the replacements is trivial and can be designed to work in polynomial time, provided it does not have to produce an exponential amount of output. In the cases of single or double literal clauses, we are only adding a constant number of variables and new clauses per such clause. Hence, the total increase in size is only $O(n)$. What about longer clauses? For every such clause, we are adding $O(n)$ new variables, and $O(n)$ new clauses (all of fixed length). It should thus be clear that the output is $O(n)$ — making f polynomial and thus $SAT \preceq_P 3SAT$.

$\square$

10.2 Clique

Clique (CLIQUE)

INSTANCE: Given a directed graph G and a positive integer k.

QUESTION: Does G contain a clique of size k?

Note that a *clique* is a set of nodes which are all interconnected.

Theorem: *CLIQUE* is *NP*-complete.

Proof: We reduce *3SAT* to *CLIQUE* to show that it is *NP*-complete.

- First, we must show that *CLIQUE* is in *NP*.

  A polynomial time non-deterministic algorithm to decide *CLIQUE* is the following:

  1. Choose, non-deterministically, a subset of k nodes from G.
  2. Check whether the chosen subset is a clique.
  3. Accept the input if it is, reject otherwise.

  Note that to check whether a given subset is a clique we only need $O(n^2)$ operations to check every element with every other element. Hence $CLIQUE \in$ NP.

- We now want to show that $3SAT \preceq_P CLIQUE$.

  Given a set of 3-literal clauses, we want to construct a graph and a constant k such that the transformation f takes polynomial time and a set of clauses C can be satisfied ($C \in$*3SAT*) if and only if the graph has a clique of size k ($f(C) \in$*CLIQUE*).

Consider a set of clauses C:

$$\{\ \{\alpha_1, \beta_1, \gamma_1\},\ \ldots \{\alpha_m, \beta_m, \gamma_m\},\ \}$$

Now consider the graph which has $3m$ nodes — one for every α_i, β_i and γ_i. As for edges, two nodes are connected if:

– they have a different subscript (come from different clauses) and

– are not contradictory (they are not x and $\bar{x}$, where x is a variable).

Thus, for example, the set of clauses $\{\{x, y, \bar{y}\}, \{\bar{x}, x, y\}\}$ has 6 nodes n_1 to n_6 corresponding to the 6 literals in the set of clauses respectively (we have renamed them to avoid having to deal with two nodes named x). The set of edges is:

$$\{(n_1, n_5), (n_1, n_6), (n_2, n_4), (n_2, n_5), (n_2, n_6), (n_3, n_4), (n_3, n_5)\}$$

The constant k is set to the number of clauses m.

Now we need to show that $C \in$*3SAT* $\Leftrightarrow$ *f(C)*$\in$*CLIQUE*. Assume that C is in *3SAT* (and can thus be satisfied). Then there is at least one member of every clause which is true, such that none of these are contradictory. The construction guarantees that these form a clique of size k. On the other hand, assume that the graph has a clique of size k. No two of these can be nodes from the same clause (since otherwise, they would not be connected). Also there are no contradictory nodes in this clique (again, by the definition of the construction). Hence, there is at least one assignment of variables (as appear in the node labels) which makes sure that each one of the clauses is true (since each clause has a node in the clique). Hence, the original set of 3-literal clauses is satisfiable and thus in *3SAT*.

But can the construction be done in polynomial time? As usual, this construction is simple to construct and we only need to check that it is of polynomial size. Note that, given an instance of *3SAT*, we construct a graph with $O(n)$ nodes, and $O(n^2)$ edges. Hence, the size of $f(C)$ is polynomially related to the size of C.

This ensures that *3SAT*$\preceq_P$*CLIQUE*.

Using the standard theorem proved in the previous chapter, and the earlier result that *3SAT* is *NP*-complete, we conclude that so is *CLIQUE*.

$\square$

10.2.1 Vertex Covering

Vertex cover (VC)

INSTANCE: Given a graph $G = (V, E)$ and a positive integer k

QUESTION: Is there a vertex cover of size k in G?

A vertex cover is a set of nodes $V' \subseteq V$ such that for every edge $(a, b) \in E$, at least one of a and b is in V'.

To prove that *VC* is *NP*-complete we first show that it is in *NP* and then prove that *CLIQUE*$\preceq_P$*VC*.

The first part is easy. Now consider an instance of *CLIQUE* which we want to express as an instance of *VC*. We are given a graph $G = (V, E)$ and a constant k. If we construct $G' = (V, (V \times V) \setminus E)$ and $k' = |V| - k$, we claim that G and k are in *CLIQUE* if and only if G' and k' are in *VC*.

Assume that G has a clique of size k. Now consider the remaining $|V| - k = k'$ nodes. We claim that these nodes form a vertex cover in G'. Consider an edge (a, b) of G'. If neither is in the vertex cover than they must both lie in the rest of the nodes — the clique of G. But the fact that there is an edge between a and b in G' implies they are not connected in G. Hence they cannot be both in a single clique. Hence, we have reached a contradiction, implying that at least one of a and b must be in the claimed vertex cover.

Conversely, assume that G' has a vertex cover of size k'. Consider the remaining $|V| - k' = k$ nodes between which there is no edge in G' (otherwise, it is not a vertex cover). Since there is no edge in G' if and only if there an edge in G, then there is a set of k nodes which are all interconnected in G. This is a clique of size k in G.

The transformation defined is clearly polynomial, and hence *VC* is *NP*-complete.

10.3 Hamiltonian Path

Hamiltonian Path (HAM)

INSTANCE: Given a directed graph G and two distinct nodes n_s and n_f

QUESTION: Is there a Hamiltonian path in G from n_s to n_f?

A Hamiltonian path is one which visits all vertices exactly once.

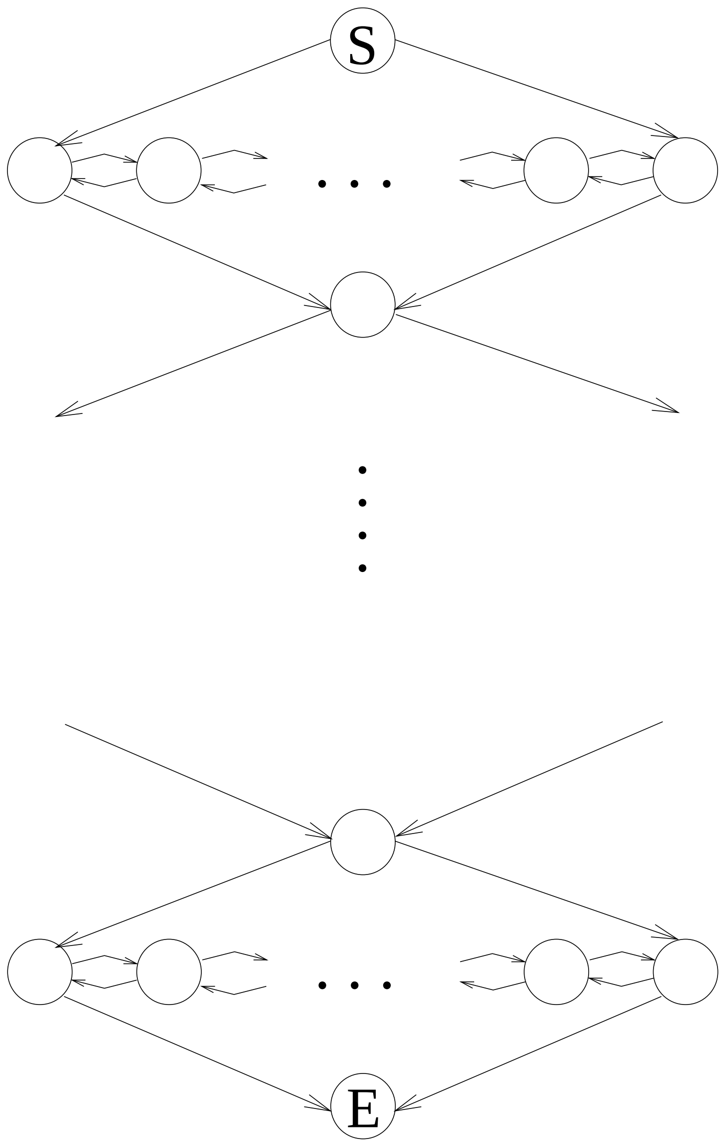
Theorem: *HAM* is *NP*-complete.

Proof outline: We use the standard technique with the problem *3SAT*.

Proving that *HAM* is in *NP* is easy and it is left up to you to do.

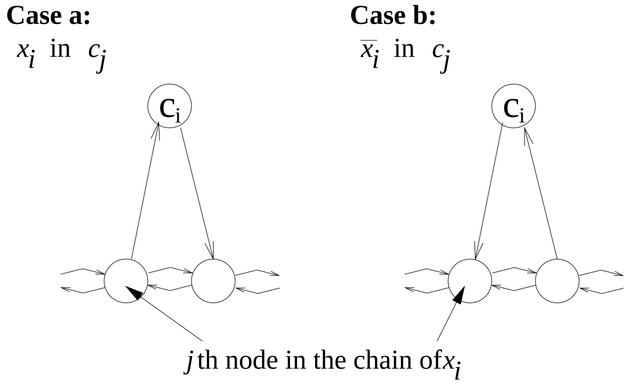
We would like to show that $3SAT \preceq_P HAM$. Consider an instance of *3SAT* — a set C of 3-literal clauses over variables V. How can we construct an instance of *HAM* — a directed graph G and two distinct nodes n_s and n_f — such that C can be satisfied if and only if there is a Hamiltonian path from n_s to n_f in G?

We start off by building a skeleton structure onto which we will build further bits and pieces. Below is the basic graph we will start off with.



For every variable, we will have a diamond shaped structure. Across each diamond we have $|C| + 1$ nodes. The start and end nodes in the graph are marked in the graph as S and F respectively.

To this we add an extra node for each clause. Every such node will be connected to three diamonds of the three literals it uses as shown in the figure below:



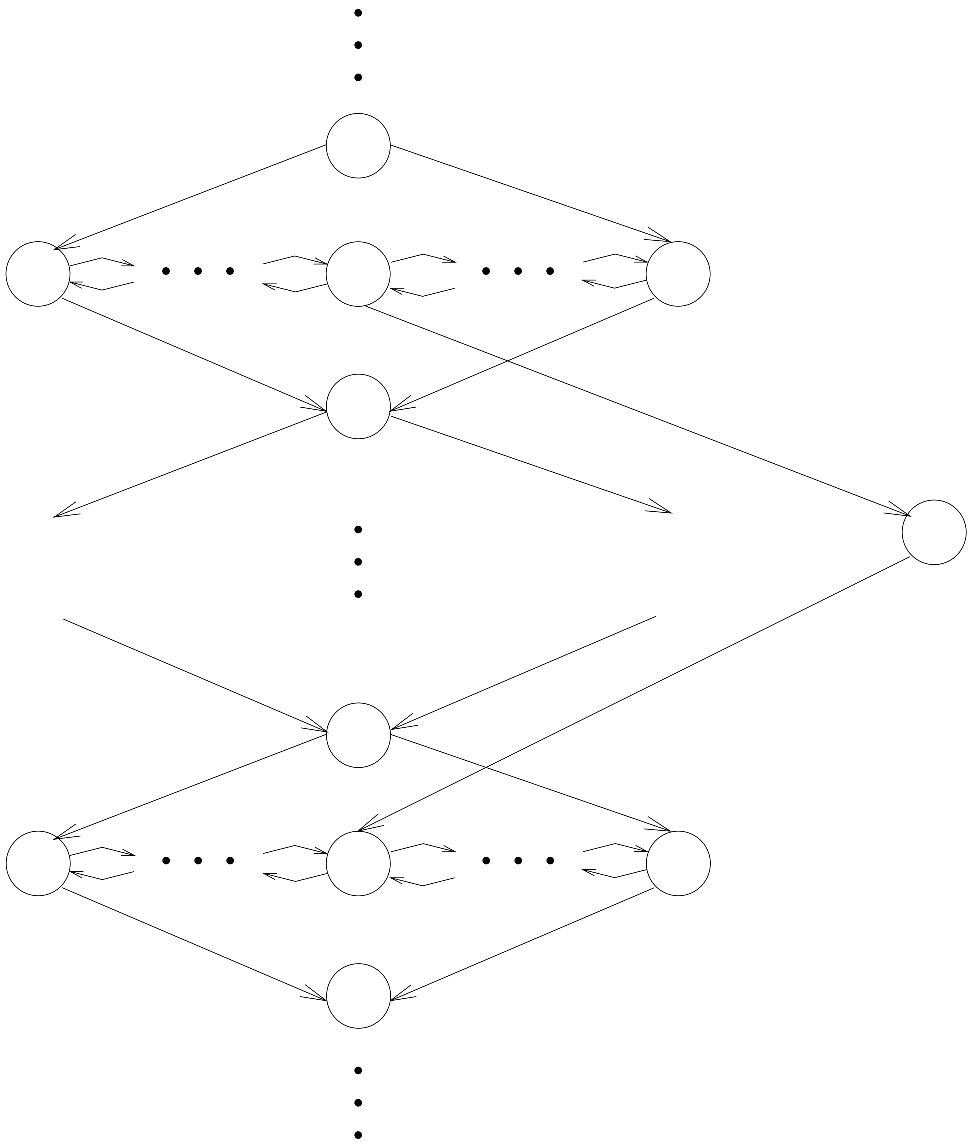
Now assume that C can be satisfied. Hence, there is an assignment of V with which every clause is true. For this assignment, there is a literal in each element of C which is true.

Now consider the following path down the graph: At the top node of the diamond of the ith variable we pick the left path if v_i is true under the particular assignment, but the right if v_i is assigned to false. We then travel across the horizontal before moving down to the next diamond.

What about the nodes representing the clauses? For each clause c_i, we pick the first satisfied literal v_j or $\bar{v}_j$ and we visit the node representing c_i from the jth diamond. Note that this is possible since we are traveling from left to right if v_j is satisfied (and thus c_i has v_j), but right to left otherwise.

Hence, there is a Hamiltonian path from S to F.

Conversely, assume there is a Hamiltonian path in the graph. We claim that this path should describe an assignment in the sense already discussed. By inspection it is clear that the only problem we can have is the case when we enter a clause node from a diamond and exit into another. But how could we then finish the remaining nodes of the first diamond? We cannot reach them normally (since there are only arrows going down), we cannot go back into that node and going in the right direction from a clause node (since no other clause node can use those pair of nodes) and neither can we enter that horizontal but going in the opposite direction (since there is nowhere to go once we hit the used node). Hence the only solution is for there to be no such abnormal connection. Since the path now follows an assignment which goes through, and hence satisfies, all clauses, C is satisfiable.



Also note that the construction can be checked to be polynomial.

Hopefully, this extremely informal presentation should convince you that *3SAT*$\preceq_P$*HAM* and hence *HAM* is *NP*-complete.

□

10.3.1 Hamiltonian Cycle

Hamiltonian Cycle (HAMC)

INSTANCE: Given a directed graph G.

QUESTION: Is there a Hamiltonian cycle in G?

A Hamiltonian cycle is a path which goes through all nodes exactly once, but returning back to the initial city at the end.

We will show that $HAM \preceq_P HAMC$.

Consider an instance of HAM. We are given a graph G and two nodes n_s and n_f. We construct G' which is the same as G, but removes all incoming edges into n_s and outgoing edges from n_f and adds a new edge from n_f to n_s.

Now assume there is a Hamiltonian path from n_s to n_f in G. Since the nodes are visited exactly once, n_s and n_f appear nowhere else in the path except for the endpoints. Hence, the Hamiltonian path is still possible in G'. Furthermore, the extra edge in G' from n_f to n_s ensures that there is a Hamiltonian cycle.

Conversely assume there is a Hamiltonian cycle in G'. Clearly, n_f must have a successor node. But the only edge leaving n_f is the one going to n_s. Removing this edge we conclude that in G', we have a Hamiltonian path from n_s to n_f. But this only uses the edges which were already in G. Hence there is a Hamiltonian path in G from n_s to n_f.

Note that the construction is simply $O(n)$ and hence polynomial.

This argument, together with a proof that $HAMC$ is in NP suffices to show that $HAMC$ is NP-complete.

10.3.2 Traveling Salesman

Traveling salesman (TS)

INSTANCE: Given a set of n cities C, a partial cost function of traveling from one city to another $cost$ (such that $cost(c_1, c_2)$ is defined if there is a direct means of traveling from city c_1 to city c_2, and if defined is the cost of the traveling) and an initial amount of money m.

QUESTION: Is there a way of traveling through all the cities exactly once, returning back to the initial city, and in the process not spending more than the initial amount of money m?

We can show that the traveling salesman problem is *NP*-complete by proving that it is in *NP* and that $HAMC \preceq_P TC$.

Consider an instance of $HAMC$. We are given a graph $G = (V, E)$. Now copy all the nodes as cities ($C = V$) and define the *cost* function only on pairs of nodes which are joined by an edge. The cost is always defined to be of 1. Set m to be $|V|$.

Now if there is a Hamiltonian cycle, the salesman can travel along the same path to produce a cycle costing exactly m. Hence the traveling salesman problem also has a solution. Conversely, if the salesman can find a valid cycle costing up to m, then clearly this cycle has to be a Hamiltonian one.

10.4 Exercises

1. Extend the instance of the traveling salesman problem such that the cost function is total. Show that this problem is also *NP*-complete.

2. Prove that $HSET$ is *NP*-complete:

   Hitting Set (HSET)

   INSTANCE: Given a collection C of subsets of S and an integer k.

   QUESTION: Is there a subset of S of not more than k elements which has elements in common with every set in C?

3. Prove that the following problems are *NP*-complete.

   Longest Path (LPATH)

   INSTANCE: Given a graph G and an integer k.

   QUESTION: Is there a path in G, of length k, and which does not visit any node more than once?

   Dominating Set (DS)

INSTANCE: Given a graph $G = (V, E)$ and number k.

QUESTION: Is there a subset V' of V, of size k, such that every element of $V - V'$ is connected to some element in V by an edge in E?

Set Splitting (SS)

INSTANCE: Given a collection C of subsets of S.

QUESTION: Is it possible to partition S into two such that every element of C has elements in both partitions?

Part V

. . . which concludes this course.

Chapter 11

Conclusion

By now you should have a better understanding of the role of formal languages in computer science. In the related first year course, the main emphasis was on the application of newly acquired mathematical techniques, and on the development of a number of results and techniques which greatly simplify the process of writing a parser for such languages.

This course started with a better analysis of a classification of languages based on the complexity of a means of describing the language, namely grammars. We showed that this increase in complexity reflected an increase in the computing power required to recognize the language. In fact, for the most general (and hence most complex) class of languages nothing less than the full power of a Turing machine can be used to recognize them.

This led to an interest in the classification of languages on a different criterion, namely that of the complexity of the Turing machine required to recognize it. In fact, we only partially analyzed this other hierarchy from a time point of view. An alternative point of view which would have followed this analysis had we had the time would have been from that of space. We also lack the time to compare the two alternative hierarchies we constructed.

But the application of formal languages to prove the complexity of decision problems is not simply interesting from a theoretical point of view. It is quite feasible that at some point in your career you may be asked to program or specify a system which can be proved to be intractable or to be *NP*-complete. A good engineer, when asked to build a bridge in glass should know the limitations he or she is working in, be able to point out the problem and possibly scan the literature to find some techniques developed to make the project more feasible. You should be able to do the same when it comes to designing and building a computer system. The knowledge acquired in this course does not simply allow you to recognize difficult problems (even though this is very important in itself). Once a problem has been proved to be *NP*-complete by reducing it to another well-known problem, it is possible to search the literature to find techniques which can be used to increase the efficiency of the algorithm. Despite having to remain in an exponential time domain, you may find results which allow you to execute your algorithm within (generous) physical time bounds you have set.