BSc (Hons) Computer Science
University of Portsmouth
Third Year

Theoretical Computer Science

M21276
Semester 1

Hugh Baldwin
hugh.baldwin@myport.ac.uk

Contents

Part I

Part A

Lecture - Induction Lecture

13:00 | 25/09/24 | Janka Chlebikova

Contact Time

- Lectures
  - Delivered by Janka
  - Slides and videos from Covid times available weekly on Moodle
- Tutorials
  - Delivered by either Janka or Dr Paolo Serafino
  - Worksheets available with solutions weekly on Moodle
  - 5 Groups
  - Work through the worksheet solutions
  - Kahoot quiz for revision
  - ‘Tutorial 0’ available as revision of concepts from DMAFP needed for this module

Assessments

- 50%, 90 minute In-Class Test covering Part A - November 20th, 2pm
- 50%, 90 minute Exam covering Part B (Only) - TB1 Assessment Period (January)
- Deferred assessment covers both Part A & B, so will be harder than the two separate exams

There are two past papers (paper-based, but still relevant to the computer-based test) available for each of the assessments. As with DMAFP, the solutions are supplied in the form of a crowd-sourced document that Janka will check over before the exam.

Resources

Lecture slides, videos and tutorial papers are released every week. All of the books on the reading list are available online through the links on Moodle.

Lecture - A1: Introduction to Languages

13:00 03/10/24 Janka Chlebikova

Languages

In this context, a **language** is a set of symbols which can be combined to create a list of acceptable **strings**. There are then also rules which tell us how to combine these strings together, known as **grammars**. The combination of an alphabet, list of valid strings and grammars is known as a language. There will be a formal definition later on.

Definition

An **Alphabet** is a finite, non-empty set of symbols.

For example, in the English language we would define the alphabet, A, as $A = \{a, b, c, d, \ldots, x, y, z\}$. These symbols can then be combined to create a **string**.

Definition

A **String** is a finite sequence of symbols from the alphabet of a language.

With the alphabet A, we could have strings such as 'cat', 'dog', '$antidisestablishmentarianism$', etc *over* the alphabet. A string with no symbols, and therefore a length of zero, is known as the **empty string**, and is denoted by Λ (capital lambda).

Definition

A **Language** over an alphabet (e.g. English over A) is a set of strings – including Λ – made up of symbols from A which are considered 'valid'. They could be valid as per a set of rules, or could be arbitrary as in most spoken languages.

If we have an alphabet, Σ, then Σ^* denotes the infinite set of all strings made up of symbols in Σ – including Λ. Therefore, a language over Σ is any subset of Σ^*.

Example

If $\Sigma = \{a, b\}$, then $\Sigma^* = \{\Lambda, a, b, aa, ab, ba, bb, \ldots\}$
There are many languages which could be defined over this alphabet, but a few simple one are

- $\emptyset$ (The empty set)
- $\{\Lambda\}$ (The set containing an empty string)
- $\{a\}$ (The set containing a)
- The infinite set $\Sigma^* = \{\Lambda, a, aa, aaa, \ldots\}$

Combining Languages

There are three common ways of combining two languages to create a new language – union, intersection and product.

Union and Intersection

Since languages are sets of strings, they can be combined as you usually would for any other set with the usual operations, union and intersection.

Example

If $L = \{aa, bb, cc\}$ and $M = \{cc, dd, ee\}$, then
$L \cap M = \{cc\}$, and
$L \cup M = \{aa, bb, cc, dd, ee\}$

Product

To combine two languages L and M, we form the set of all **concatenations** of strings in L with strings In M. This is known as the **product** of the two languages.

Definition

To **Concatenate** two strings is to juxtapose them such that a new string is made by appending the second string to the end of the first.

Example

If we concatenate the strings ab and ba, the result would be $abba$.
This can be represented using the function cat, such as $\mathrm{cat}(ab, ba) = abba$

With this definition of concatenation, we can say that the product of two languages, L and M would be $L \cdot M$, where $L \cdot M = \{\mathrm{cat}(s, t) : s \in L \text{ and } t \in M\}$

Example

If $L = \{a, b, c\}$ and $M = \{c, d, e\}$, then the product of the two languages, $L \cdot M$, is the language
$L \cdot M = \{ac, ad, ae, bc, bd, be, cc, cd, ce\}$

Further on Products

It is simple to see that for any language, L, the following simple properties are true:

$$L \cdot \{\Lambda\} = \{\Lambda\} \cdot L = L$$
$$L \cdot \emptyset = \emptyset \cdot L = \emptyset$$

Commutativity

Aside from the above properties, the product of any two languages is **not** commutative, and therefore

$$L \cdot M \neq M \cdot L$$

Example

If $L = \{a, b\}$ and $M = \{b, c\}$, then the product $L \cdot M$ is the language

$$L \cdot M = \{ab, ac, bb, bc\}$$

but the product $M \cdot L$ is the language

$$M \cdot L = \{ba, bb, ca, cb\}$$

which are clearly not the same language, as the only common string is bb

Associativity

However, the product of any two languages is associative. This means that if we had any three languages, L, M and N, then

$$L \cdot (M \cdot N) = (L \cdot M) \cdot N$$

Example

If we have the three languages $L = \{a, b\}$, $M = \{b, c\}$ and $N = \{c, d\}$, then

$$\begin{aligned} L \cdot (M \cdot N) &= L \cdot \{bc, bd, cc, cd\} \\ &= \{abc, abd, acc, acd, bbc, bbd, bcc, bcd\} \end{aligned}$$

which is the same as

$$\begin{aligned} (L \cdot M) \cdot N &= \{ab, ac, bb, bc\} \cdot N \\ &= \{abc, abd, acc, acd, bbc, bbd, bcc, bcd\} \end{aligned}$$

Powers of Languages

If we have the language L, then the product $L \cdot L$ of the language is denoted by L^2. The product L^n for every $n \in \mathbb{N}$ is defined as

$$\begin{aligned} L^0 &= \{\Lambda\} \\ L^n &= L \cdot L^{n-1}, \text{if} n > 0 \end{aligned}$$

Example

If $L = \{a, b\}$ then the first few powers of L are

$$\begin{aligned} L^0 &= \{\Lambda\} \\ L^1 &= L = \{a, b\} \\ L^2 &= L \cdot L = \{aa, ab, ba, bb\} \\ L^3 &= L \cdot L^2 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\} \end{aligned}$$

The Closure of a Language

If L is a language over Σ, then the **closure** of L is the language denoted by L^*, and the **positive closure** is language denoted by L^+.

Definition

The **Closure** of the language L, L^* is defined as

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$$

and so, if $L = \{a\}$ then

$$\begin{aligned} L^* &= \{\Lambda\} \cup \{a\} \cup \{aa\} \cup \{aaa\} \cup \dots \\ &= \{\Lambda, a, aa, aaa, \dots\} \end{aligned}$$

Definition

The **Positive Closure** the language L, L^+ is defined as

$$L^+ = L^1 \cup L^2 \cup L^3 \cup L^4 \cup \dots$$

and so, if $L = \{a\}$ then

$$\begin{aligned} L^+ &= \{a\} \cup \{aa\} \cup \{aaa\} \cup \{aaaa\} \cup \dots \\ &= \{a, aa, aaa, aaaa, \dots\} \end{aligned}$$

It then follows that $L^* = L^+ \cup \{\Lambda\}$, but it's not necessarily true that $L^+ = L^* - \{\Lambda\}$.

Example

If our alphabet is $\Sigma = \{a\}$ and our language is $L = \{\Lambda, a\}$, then

$$L^+ = L^*$$

Properties of Closures

Based upon these definitions, you can derive some interesting properties of closures.

Example

If L and M are languages over the alphabet Σ, then

$$\begin{gathered} \{\Lambda\}^* = \emptyset^* = \{\Lambda\} \\ L^* = L^* \cdot L^* = (L^*)^* \\ \Lambda \in L \text{ if and only if } L^+ = L^* \\ (L^* \cdot M^*)^* = (L^* \cup M^*)^* = (L \cup M)^* \\ L \cdot (M \cdot L)^* = (L \cdot M)^* \cdot L \end{gathered}$$

These will be explored more during the tutorial session for this week

The Closure of an Alphabet

Going back to the definition of Σ^* of the alphabet Σ, it lines up perfectly with the definition of a closure such that Σ^* is the set of all strings over Σ. This means that there is a nice way to represent Σ^* as follows

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$$

From this, we can also see that Σ^k denotes the set of all strings over Σ whose length is k.

Lecture - A2: Grammars

13:00 03/10/24 Janka Chlebikova

As discussed previously, you can define a language over an alphabet as a set of arbitrary strings that are considered 'valid', but you can also define a language using a **grammar**.

Definition

A **Grammar** is a set of rules used to define a language over an alphabet, by specifying the structure of valid strings in said language.

To describe the grammar of a language, two sets of symbols (alphabets) are required, terminals and non-terminals

Definition

Terminal symbols are those from which the actual strings which make up a language are derived, like the symbols discussed previously for a manually specified language. In the case of a spoken language, these would usually be lowercase letters, such as the latin alphabet used in English.

Definition

Non-Terminal symbols are 'temporary' symbols (fully disjoint from the set of terminal symbols) used to define the grammar's replacement rules. These must all be replaced by terminals before a production can be considered a valid string in the language. These are usually represented by uppercase letters, or letters from an alphabet other than that of the language.

Productions

Further to this, a grammar for the language L over Σ would typically consist of a set of productions (grammar rules) which allow you to produce the set of strings which make up L.

Definition

A **Production** is a rule which allows you to produce a string for a language. They are typically in the form

$$\alpha \to \beta$$

where α is a string of symbols from the set of terminals (Σ) and β is a string of symbols from the set of non-terminals.

You can read these rules in several ways – $\alpha \to \beta$ could be read as;

- replace α by β
- α produces β
- α rewrites as β
- α reduces to β

A Formal Definition

Formally, a grammar is defined by the following properties

1. An alphabet T of terminal symbols (This is the same as the alphabet of the language defined by this grammar)
2. An alphabet N of non-terminal symbols
3. A specific non-terminal symbol known as the **start symbol**, which is often S
4. A finite set of productions in the form $\alpha \to \beta$ where α and β are strings over the alphabet $N \cup T$

Every grammar must have the special non-terminal symbol known as a start symbol, and it must also have at least one production in which the left side consists of only the start symbol.

Example

Let G be a grammar defined by
- The set of terminals $T = \{a, b\}$
- A single non-terminal being the start symbol S
- The set of productions

$$S \to \Lambda,\ S \to aSb$$

or for shorthand,

$$S \to \Lambda \mid aSb$$

To get the full set of strings which are valid for this grammar, we need to perform a **derivation**

Derivations

Starting from any production where the left side consists of only the start symbol, we can go through a step by step process to generate all strings belonging to the language described by a grammar.

Example

Using the grammar G from the previous example, we could start with either of the two productions, since they both have only the start symbol S on the left-hand side. This means that the first step of our process would give us Λ and aSb.

In this example, the string aSb is a **sentential form** of the terminals $\{a, b\}$ and non-terminal S. From this point, to extend or generate any other strings, we need to perform a derivation.

Definition

If x and y are sentential forms, and $\alpha \to \beta$ is a production, then to replace α by β in $x\alpha y$ is to **Derive** a new string. This is denoted by writing

$$x\alpha y \Rightarrow x\beta y$$

Example

Going back to the grammar G again, since it contains the production $S \to aSb$, we could take our first step to generate the string aSb, then we can go another step further and derive $aaSbb$ from aSb, which means

$$aSb \Rightarrow aaSbb$$

Since we can use this production again and again, we could derive that

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow \dots$$

To represent this multi-step process, we have the following symbols to show how the derivation was constructed

- $\Rightarrow$ derives in a single step
- $\Rightarrow^+$ derives in one or more steps
- $\Rightarrow^*$ derives in zero or more steps

Example

Using the grammar G, we can show multiple derivations using these new symbols

$$S \Rightarrow \Lambda, S \Rightarrow aSb, \therefore S \Rightarrow^* ab$$
$$S \Rightarrow^* aaabbb$$
$$S \Rightarrow^* aaaaaaSbbbbbb$$

What is $L(G)$?

Definition

If G is a grammar with the start symbol S and the set of terminals T, then the language generated by G is the set

$$L(G) = \{s \mid s \in T^* \text{ and } S \Rightarrow^+ s\}$$

That is to say that it's the set of all strings containing only terminal symbols which can be derived from the start symbol with one or more steps.

Example

With the grammar G,

$$L(G) = \{\Lambda, ab, aabb, aaabbb, aaaabbbb, \dots\}$$

Infinite Languages

Within an infinite language, there is no limit on the length of strings, and therefore also no limit on the number of steps needed to derive a string. If the grammar of a language has n productions, then any derivation with $> n + 1$ steps must use at least one production twice. Therefore, if a language is infinite, then a production or sequence of productions must be used repeatedly to construct the derivations of the grammar.

Example

The infinite language $\{a^n b \mid n \geq 0\}$ can be described by a grammar with the production $S \to b \mid aS$. To derive the string $a^n b$, you would apply the production $S \to aS$ n times over and then finish the derivation with the production $S \to b$.

With the production $S \to aS$, we can also say that "If S derives w, then it also derives aw"

Recursion and Indirect Recursion

Definition

A production is **Recursive** if its entire left side is contained within the right side.

Example

The production $S \to aSb$ is recursive, since the sentential form it produces contains itself.

Definition

A production is **Indirectly Recursive** if it is possible to derive a sentential form in two or more steps which contains the entire left side of the production.

Example

If a grammar contains the rules $S \to b \mid aA$ and $A \to c \mid bS$, then both productions are indirectly recursive, since

$$S \Rightarrow aA \Rightarrow abS$$
$$A \Rightarrow bS \Rightarrow baA$$

Recursive Grammars

Definition

A grammar is **Recursive** if it contains either a recursive or indirectly recursive production. *In order for a grammar to be infinite, it must first be recursive.*

Constructing Grammars

We've seen previously how to derive a language from the definition of a grammar, but we also need to be able to construct a grammar to produce a language. It is often very difficult and sometimes impossible to write down a grammar for a given language, and there are often multiple grammars which form the exact same language.

Finite Languages

In the case of a finite language, it is always possible to find a grammar which produces the language. This is because it is possible to simply make a grammar which contains a production for every string of the language, where $S \to w$ for every string w in the language.

Example

The finite language $\{a, b, c, abc\}$ over the alphabet $\{a, b, c\}$ can be described by the grammar

$$S \to a|b|c|abc$$

Infinite Languages

However, in the case of an infinite language, there is no universal method that will always produce a valid grammar. This means it is much harder to find a valid grammar, and often requires **combining grammars**.

Example

A simple example of an infinite language would be $\{\Lambda, a, aa, aaa, \ldots\}$ which we can also write in the form $\{a^n : n \in \mathbb{N}\}$. We can then see that one grammar which describes this language is as follows;

- The set of terminals, $T = \{a\}$
- The non-terminal start symbol, S
- The productions $S \to \Lambda, S \to aS$

Combining Grammars

Suppose we have two languages, L and M for which there are known grammars. There are a few simple rules we can use to create the grammars which describe the languages $L \cup M$, $L \cdot M$ and L^*.

To create this 'composite' grammar, we can describe L and M with disjoint sets of non-terminals. If we have the start symbols of L and M as A and B respectively, i.e. $L : A \to \ldots$, $M : B \to \ldots$, we can then combine these as productions in another grammar to create the language.

Unions

If we need to produce the union of two languages, $L \cup M$, we start with the two productions $S \to A \mid B$, followed by the productions which make up L and M, using A and B as their respective start symbols.

Example

If we wanted to write the grammar that produces the language

$$K = \{\Lambda, a, b, aa, bb, aaa, bbb, \ldots, a^n, b^n\}$$

we can realise that K is the union of two different languages,

$$L = \{a^n \mid n \in \mathbb{N}\}$$
$$M = \{b^n \mid n \in \mathbb{N}\}$$

and so we can write a grammar for K as

- $S \to A \mid B$ (Union rule)
- $A \to \Lambda \mid aA$ (Ls grammar)
- $B \to \Lambda \mid bB$ (Ms grammar)

Products

Similarly, if we need to produce the the product of two languages, $L \cdot M$, we start with the production $S \to AB$, followed by the productions which make up L and M, using A and B as their respective start symbols.

Example

If we wanted to write the grammar that produces the language

$$K = \{\Lambda, a, b, aa, ab, aaa, bb, \ldots\}$$

we can realise that K is the product of the two languages,

$$L = \{a^n \mid n \in \mathbb{N}\}$$
$$M = \{b^n \mid n \in \mathbb{N}\}$$

and so we can write a grammar for K as
- $S \to AB$ (Product rule)
- $A \to \Lambda \mid aA$ (Ls grammar)
- $B \to \Lambda \mid bB$ (Ms grammar)

Closures

And finally, if we need to produce the closure of a language, L^*, we start with the production $S \to AS \mid \Lambda$ followed by the productions which make up L, using A as its start symbol.

Example

If we wanted to write the grammar that produces the language L^* where

$$L = \{aa, bb\}$$

and so

$$L^* = \{\Lambda, aa, bb, aaaa, aabb, bbbb, bbaa, \ldots\}$$

we can write the grammar for L^* as
- $S \to AS \mid \Lambda$ (Closure rule)
- $A \to aa \mid bb$ (Ls grammar)

Equivalent Grammars

Any given language could have many different grammars which produce the exact same thing. This means that grammars are not unique. We can also use this to simplify grammars.

Example

Take the grammar from the previous example,

$$S \to AS \mid \Lambda$$
$$A \to aa \mid bb$$

If we replace A in S with the right side of A, aa, we get the production $S \to aaS$. We can then do the same with the other production to get $S \to bbS$. Therefore, we can write this grammar in a simplified form as

$$S \to aaS \mid bbS \mid \Lambda$$

Lecture - A3: Regular Languages

13:00 10/10/24 Janka Chlebikova

A regular language is a simple sort of language that meets a few basic requirements. They are often used for pattern matching and in lexical analysers.

Definition

There are several ways of describing a regular language
- Languages inductively formed by combining simple languages
- Languages described by a regular expression
- Languages produced by a grammar with a special and restricted form
- Languages that can be accepted by a finite automata

Inductively Formed Regular Languages

We start with a very simple language or set thereof and build more complex ones by combining them in particular ways–

- **Basis** - $\emptyset$, $\{\Lambda\}$ and $\{a\}$ are all regular languages for all $a \in \Sigma$
- **Induction** - If L and M are regular languages, then $L \cup M$, $L \cdot M$ and L^* are also regular languages

The basis of this definition gives us the following four regular languages over the alphabet $\Sigma = \{a, b\}$:

$$\emptyset, \{\Lambda\}, \{a\}, \{b\}$$

All regular languages over Σ can be constructed by combining these four languages in various ways by recursively applying the union, product and closure operations.

Example

Is the language $\{a, ab\}$ regular?
Yes, since it can be constructed from the four basic regular languages using a product and union operation–

$$\begin{aligned} \{a, ab\} &= \{a\} \cdot \{\Lambda, b\} \\ &= \{a\} \cdot (\{\Lambda\} \cup \{b\}) \end{aligned}$$

We can also see from this that it is possible to construct any finite language in this way, and therefore **all finite languages are regular**. There are also many infinite languages which are regular.

Regular Expressions

If you wanted to find an algorithm which can determine if a string belongs to a particular regular language, you would most likely use a **regular expression**

Definition

A **Regular Expression** is basically a short-hand way of showing how a regular language is constructed from a base set of regular languages.
For each regular expression E, there is a regular language $L(E)$

Like the regular languages discussed previously, regular expressions can be manipulated inductively to form new regular expressions.

- **Basis** - Λ, $\emptyset$ and a are regular expressions for all $a \in \Sigma$
- **Induction** - If R and E are also regular expressions, then (R), $R + E$, $R \cdot E$ and R^* are also regular expressions

Example

A few of the regular expressions over the alphabet $\Sigma = \{a, b\}$ are

$$\Lambda, \emptyset, a, b$$
$$\Lambda + b, b^*, a + (b \cdot a)$$
$$(a + b) \cdot a, a \cdot b^*, a^* \cdot b^*$$

So that we don't have to include lots of parentheses, the operations have the following hierarchy–

- $*$ (Highest)
- $\cdot$
- $+$ (Lowest)

Example

The regular expression

$$a + b \cdot a^*$$

is the same as

$$(a + (b \cdot (a^*)))$$

We can also juxtapose languages rather than using $\cdot$ where there is only one possible interpretation

Example

The regular expression

$$a + b \cdot a^*$$

is the same as

$$a + ba^*$$

Operators

In regular expressions, there are two binary operations ($+$ and $\cdot$), and one unary operation ($*$). These are closely related to the union, product, and closure operations over corresponding languages.

Example

The regular expression

$$a + bc^*$$

is more-or-less shorthand for the regular language

$$\{a\} \cup (\{b\} \cdot (\{c\}^*))$$

From Expressions to Languages

There is a very simple substitution method to find the language described by a regular expression.

Example

Find the language described by the regular expression $a + bc^*$–

$$\begin{aligned} L(a + bc^*) &= L(a) \cup L(bc^*) \\ &= L(a) \cup (L(b) \cdot L(c^*)) \\ &= L(a) \cup (L(b) \cdot L(c)^*) \\ &= \{a\} \cup (\{b\} \cdot \{c\}^*) \\ &= \{a\} \cup (\{b\} \cdot \{\Lambda, c, c^2, c^3, \ldots\}) \\ &= \{a\} \cup \{b, bc, bc^2, bc^3, \ldots\} \\ &= \{a, b, bc, bc^2, bc^3, \ldots\} \end{aligned}$$

This approach can also be used to prove that a given language is regular.

Many infinite languages can easily be proven to be regular, by finding a regular expression that describes it. Not all infinite languages are regular however.

Regular expressions are also not necessarily unique, as they may represent the same language as another regular expression, such as $a + b$ and $b + a$ being different regular expressions, but both represent the language $\{a, b\}$

Definition

Regular expressions are said to be **Equal** if their languages are the same. If R and E are regular expressions where $L(R) = L(E)$, then they are equal and can be written as $R = E$.

Properties of Regular Expressions

There are many general equalities for regular expressions, which hold for any regular expressions R, E, and F, and can be proven by using the basic properties of languages and sets.

- Additive Properties

$$\begin{aligned} R + E &= E + R \\ R + \emptyset &= \emptyset + R = R \\ R + R &= R \\ (R + E) + F &= R + (E + F) \end{aligned}$$

- Product Properties

$$R\emptyset = \emptyset R = \emptyset$$
$$R\Lambda = \Lambda R = R$$
$$(RE)F = R(EF)$$

- Distributive Properties

$$R(E+F) = RE + RF$$
$$(R+E)F = RF + EF$$

- Closure Properties

$$\emptyset^* = \Lambda^* = \Lambda$$
$$R^* = R^*R^* = (R^*)^* = R + R^*$$
$$R^* = \Lambda + RR^* = (\Lambda + R)R^*$$
$$RR^* = R^*R$$
$$R(ER)^* = (RE)^*R$$
$$(R+E)^* = (R^*E^*)^* = (R^* + E^*)^* = R^*(ER^*)^*$$

Regular Grammars

There are several ways to characterize grammars which describe regular languages.

Definition

A **Regular Grammar** is one where all of the productions take one of the following forms

$$B \to \Lambda$$
$$B \to w$$
$$B \to A$$
$$B \to wA$$

Where A and B are non-terminals and w is a non-empty string of terminals

Only one non-terminal can appear on the right hand side of any production. Non-terminals must appear on the right end of the right hand side. Therefore the productions $A \to aBc$ and $S \to TU$ are not productions in a regular grammar, but $A \to abcA$ is.

For any regular language, we can find a regular grammar which describes it, but there may also be non-regular grammars which produce it.

Lecture - A4: Finite Automata

13:00 10/10/24 Janka Chlebikova

There are several different 'models of computation', such as finite automata, push-down automata and turing machines. They are all abstract models of computers which are capable of different things. They all have an input tape with a single string of an infinite length and can accept or reject the input string.

Finite Automata

Finite automata are the most basic models of a computer.

Definition

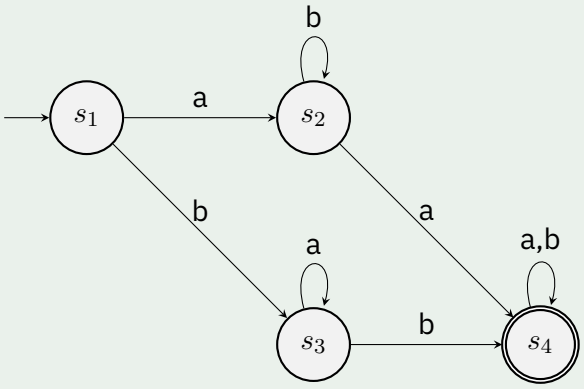
A **Finite Automaton** has three components–

- An input tape, which contains an input string over the alphabet Σ
- A head, which reads the input string one symbol at a time
- Memory, which is realised as a finite set Q of states, of which the automaton can only be in one at any time

The 'program' of the automaton defines how the read symbol changes the current state.

To better understand an automaton, they are typically represented as a transition graph, which is a form of directed graph.

Example



This is a finite automata which takes the alphabet $\Sigma = \{a, b\}$, and accepts strings such as $abba$, $baab$, etc.

Each automaton has one initial state, denoted by the arrow entering s_1, and at least one final or accepting state, denoted by the double circle such as s_4. Once the whole input string is read into the automaton, if the current state is a final or accepting state, then the string is accepted. Otherwise, the string is rejected. The language of an automaton is the set of strings it accepts.

A Formal Definition

Definition

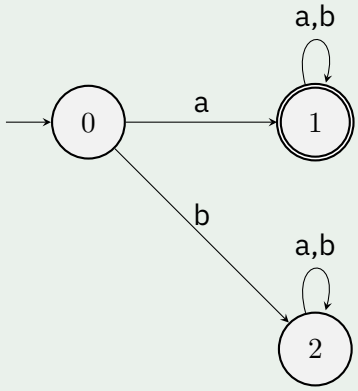
A finite automaton is defined by
- A set of states, Q
- A start state, $s \in Q$
- A set of accepting states, $Q_a \subset Q$
- A set of transition functions $T : Q \times \Sigma \to Q$

Example

A very simple finite automaton could be defined by

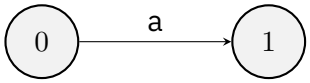
$$Q = \{0, 1, 2\}$$
$$s = 0$$
$$Q_a = \{1\}$$
$$T(0, a) = 1, T(0, b) = 2$$
$$T(1, a) = T(1, b) = 1$$
$$T(2, a) = T(2, b) = 2$$

And as a transition graph,



State Transition Functions

A state transition function, $T : Q \times \Sigma \to Q$, of the form



is represented by $T(0, a) = 1$, where $0, 1 \in Q$ and $a \in \Sigma$.

Deterministic Finite Automata

In the previous examples, there is always exactly one transition function for every state and symbol – every node has exactly one edge for each possible input symbol. Such automata are known as **deterministic** finite automata. That means that for any input string, we always know which unique state we are in at any symbol in the string.

Definition

A **DFA** accepts a string w over Σ^* if and only if there is a path from the initial state to an accepting state, such that w is the concatenation of the labels on the edges of the path. Otherwise, the DFA rejects w. For a FA to be deterministic, there must also be **exactly** one edge from each state for each symbol in Σ.
The set of all strings over Σ accepted by the DFA M is known as the language of M, and is represented by $L(M)$.

It has been proven that "the class of regular languages is exactly the same as the class of languages accepted by DFAs". This means that for any given regular language, we can find at least one DFA which recognises it, and vice versa.

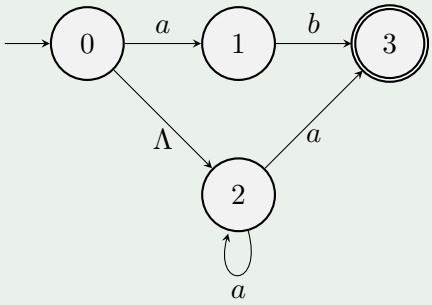
Non-Deterministic Finite Automata

With a DFA, you always know exactly which state it will be in after a given input string. This is not the case with an NFA, since in any given state there may be zero or more transitions for each symbol in the alphabet. Since more than one transition is possible for each state, it is impossible to know which state it will be in given an input string. If the NFA reaches an accepting state after reading the string, it is accepted. Otherwise it is rejected. Since there could be multiple paths for any given strings, as long as one or more path reaches an accepting state, it is accepted. Also, if there are no edges that can be traversed from a state using the current input symbol, the input string is immediately rejected.

Additionally, an edge may be labelled with Λ, which means that one can make the transition without consuming any symbols from the input string.

The non-determinism of these FA also means that they are impossible to run on a traditional computer, at least without using tricks that make them much less efficient. With the advent of quantum computers, it may become possible to use NFAs for much more efficient parsing and pattern matching.

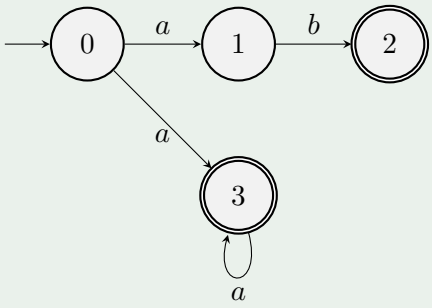
Example



This is an NFA which corresponds to the regular expression $ab + a^*a$.

Since we are able to have multiple edges with the same symbol from a node, we could also use this different NFA which recognises the language of the same regular expression.

Example



NFA Transition Functions

Since NFAs are non-deterministic, their transition functions instead take the form $T : Q \times \Sigma \to P(Q)$. This allows for the result of any given transition function to be a set of states which could be transitioned to.

Example

Taking the first example of an NFA, the transition function might look as below–

$$T : Q \times \Sigma \to P(Q)$$
$$T(0, a) = \{1\}, T(0, \Lambda) = \{2\}$$
$$T(1, b) = \{3\}$$
$$T(2, a) = \{2, 3\}$$

and for the second example–

$$T : Q \times \Sigma \to P(Q)$$
$$T(0, a) = \{1, 3\}$$
$$T(1, b) = \{2\}$$
$$T(3, a) = \{3\}$$

DFAs vs NFAs

On the surface it may seem like there aren't many differences between NFAs and DFAs, and in fact, DFAs are a subset of NFAs. It has also been proven that "the class of languages accepted by NFAs is also exactly the class of regular languages", just as for DFAs. This also means that for every NFA there must be an equivalent DFA, or at least one that accepts the same language.

Generally, NFAs are easier to construct for any given regular expression, and tend to be more simple, with fewer states and transitions. However, DFAs are much easier to execute, especially on the deterministic computers we currently use. Since they recognise the same set of languages, it is always possible to find equivalents, though the DFA will usually be much larger and more complex.

Finding an Equivalent DFA

Generally, the DFA will have more states than the equivalent NFA. If the DFA has n states, the DFA may have as many as 2^n states. To actually do this conversion, you can follow the algorithm below (shown here converting the NFA in the figure below)

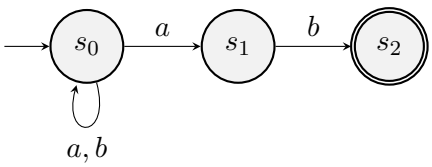


Figure 5.1: The NFA which accepts the language for the regular expression $(a + b)^* ab$

Step 1 – Begin with the NFA start state. If it's connected to any other states by Λ, make the initial state of the DFA a set containing the NFA's initial state and those others.

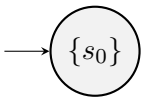


Figure 5.2: The resulting DFA after step 1

Step 2 – For each symbol, determine the set of possible NFA states you may be in after reading it. This set becomes the label of the state, and is connected to the previous state by that symbol. Only create a new state if one with that label does not already exist.

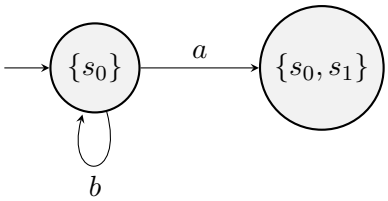


Figure 5.3: The resulting DFA after step 2

Step 3 – Repeat Step 2 for each new DFA state, until the system is closed. DFA accepting states are those that include one or more NFA accepting states. If there is no transition for a symbol in the NFA, create a new state in the DFA with the label $\emptyset$, and add loops for all symbols (known as a non-final trap state).

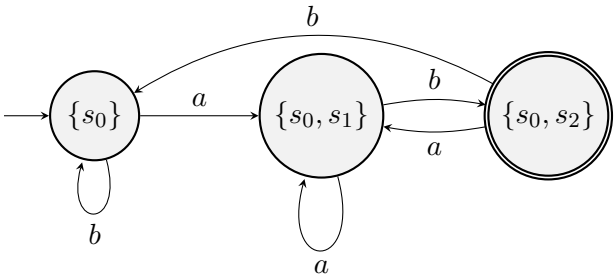


Figure 5.4: The final DFA

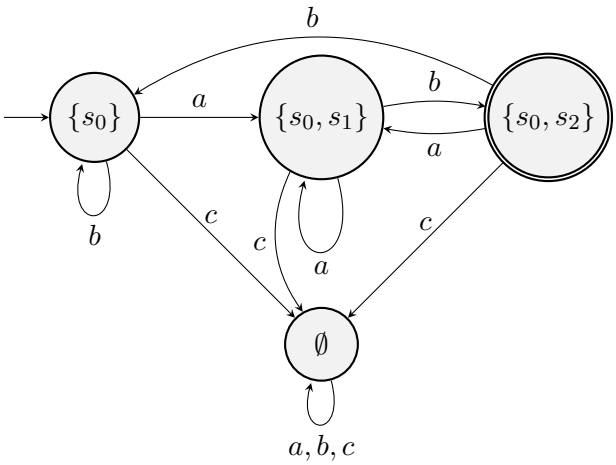


Figure 5.5: The same DFA, assuming that the language was $\{a, b, c\}$ rather than $\{a, b\}$

Lecture - A5: Finite Automata & Regular Languages

13:00 17/10/24 Janka Chlebikova

Automata and Regular Languages

How can we prove that the set of languages accepted by automata is exactly the set of regular languages? Well it takes multiple steps.

Step 1– We need to show that for any regular expression, we can find an NFA that recognises it, thus proving

$$L(\text{regular expressions}) \subseteq L(\text{NFA})$$

Step 2– Then we take a finite automaton and find a regular expression which describes the language of the automaton, thus proving that

$$L(\text{NFA}) \subseteq L(\text{regular expressions})$$

Step 3– We then combine the previous results to get that

$$L(\text{NFA}) = L(\text{regular expressions})$$

Regular Expressions to Finite Automata

Given any regular expression, we need to be able to construct a finite automaton (either NFA or DFA) which recognises its language. Given that all regular expressions are built up using union, product and closure operations, we can use a set of rules to construct an FA for a given regular expression.

Rule 0– Start the algorithm with a simple FA that has a start state, a single accepting state and an edge labelled with the given regular expression.

Rule 1– If an edge is labelled with $\emptyset$, erase the edge.

Rule 2– Transform any edge of the form $R + S$ into two edges, labelled R and S.

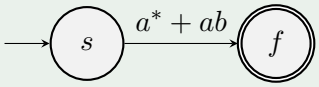
Rule 3– Transform any edge of the form $R \cdot S$ into a new intermediary state, with an edge labelled R going into it from the previous state, and an edge labelled S going into the next state.

Rule 4– Transform any edge of the form R^* into a new intermediary state with a looping edge labelled R, and two edges labelled Λ, one entering and one exiting the new state.

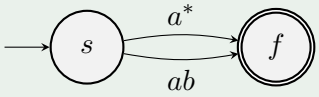
You must apply these rules until no more of the edges can be broken up any further using the rules.

Example

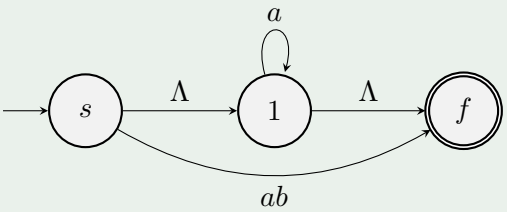
For the regular expression $a^* + ab$, we would follow the steps below–
Start with rule 0–



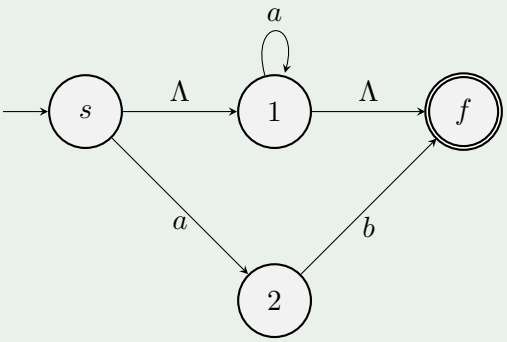
Apply rule 2–



Apply rule 4 to a^*–



Apply rule 3 to ab–



Thus, we have proven that we can produce a finite automata for any regular expression, and given a set of rules to do so.

Finite Automata to Regular Expressions

Now, to do the same thing in reverse, we need a new algorithm, as follows–

Step 1– Create a new start state s, and draw an edge labelled Λ to the original start state.

Step 2– Create a new accepting state f, and draw edges labelled Λ from all original accepting states.

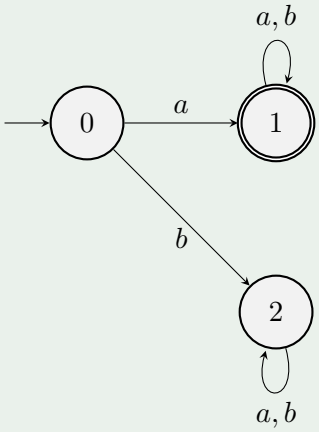
Step 3– For each pair of states i and j with more than one edge from i to j, replace all edges with a single edge labelled with the regular expression formed by the sum of labels on each edge from i to j.

Step 4– Step-by-step eliminate any states, changing the labels on corresponding edges until the only remaining states are s and f. When deleting a state, you must replace any possible transitions with a regular expression which matches all of the possible transitions.

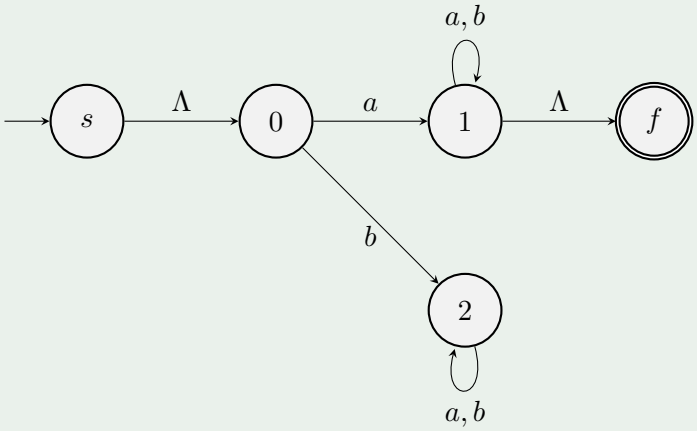
You should end up with an FA with only two states, and a single edge connecting them which is labelled with the desired regular expression.

Example

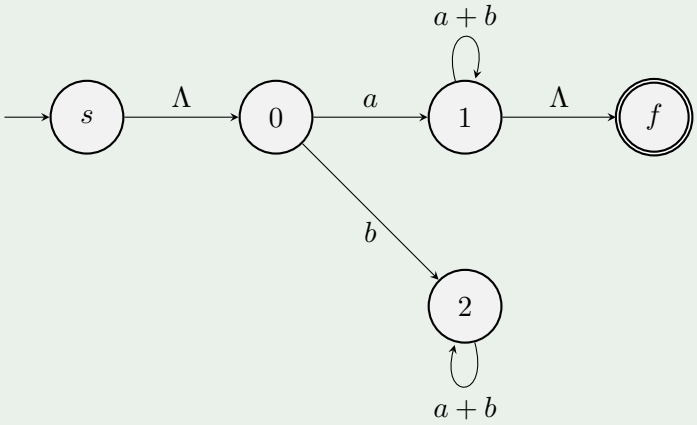
Initial DFA–



Steps 1 and 2: add new initial and final states–

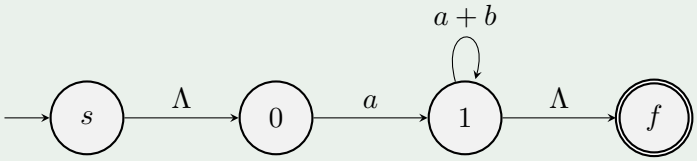


Apply step 3–

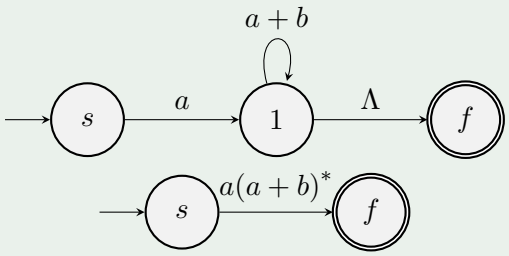


Example

We can then eliminate state 2, since there is no way for a string to be accepted from it–



Then we can start to eliminate states to get our final regular expression–



And this then leaves us with the final regular expression $a(a+b)^*$.

And now, we have proven that we can transform any regular expression into an NFA, and vice versa, and that we can transform any NFA into a DFA, and vice versa.

$$\text{regular expressions} \Leftrightarrow \text{NFA} \Leftrightarrow \text{DFA}$$

Simplifying DFAs

A lot of the time, we will end up with a complicated DFA, with far more states than is absolutely necessary. To get a simplified DFA, we need to transform it into a unique DFA with the minimum number of states which recognises the same language.

We can find this DFA in two parts– Finding all pairs of equivalent states, and combining the equivalent states into a single state, with modified transition functions.

Equivalent States

We can say that two states s and t are equivalent if for all possible strings w left to consume (including Λ), the DFA will finish in the same **type** of state (i.e. accepting or non-accepting) after consuming w. Therefore, once you arrive at s or t, the same string will always produce the same result (accepting or rejecting the string).

Two states are not equivalent if there exists a string w such that, after consuming w, we would end up accepting the string if we started from s and rejecting it if we started from t, or vice versa.

Lecture - A6: Beyond Regular Languages

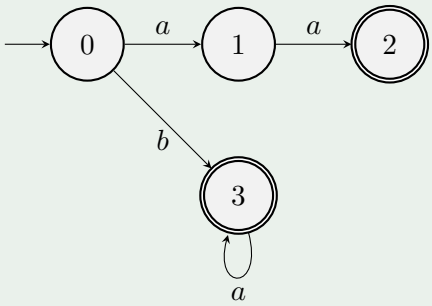
13:00 17/10/24 Janka Chlebikova

Non-deterministic Finite Automata to Regular Grammars

Every NFA can be converted into a corresponding regular grammar. Each state of the NFA is associated with a non-terminal symbol of the grammar, and the initial state is associated with the start symbol. Each transition is associated with a grammar production, and every final state has the additional production which produces Λ.

Example

For the NFA–



–we can apply the rules defined above to get the following productions, supposing that $S = 0$, $A = 1$, $B = 2$ and $C = 3$

$$S \to aA \mid aC$$
$$A \to bB$$
$$B \to \Lambda$$
$$C \to aC \mid \Lambda$$

This may not be the simplest set of productions, but it does work.

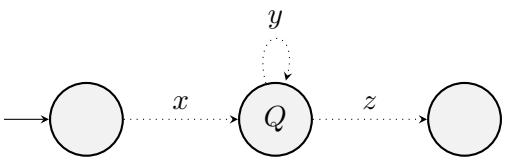
Testing for Regular Languages

Given a language, we need to be able to determine if it is regular. This would also allow us to prove that a given language cannot be recognised by any finite automaton.

One possible method is by using the **pumping lemma**, which applies for infinite languages. Since all finite languages are regular, this is enough to determine if a language is regular or not.

If the input string is long enough (e.g. greater than the number of states in the minimum state DFA for the language), then there must be at least one state Q which is visited more than once. Therefore, there must be at least one closed loop, which begins and ends at the state Q, and a particular string y, which corresponds to this loop.

We can represent this situation as the FA–



Where each dotted edge represents a path that may contain other states and transitions of the DFA.

- x is a string of symbols which the automaton consumes to transition from the start state, to get to the state Q
- y is the string of symbols to transition around the closed loop
- z is the string of symbols to transition from Q to an accepting state

Thus, we know that the string xyz is accepted, but also that the DFA must accept the strings $xy, xyz, xyyz, \ldots, xy^kz$, and that the central string y is 'pumped'.

Formally, the pumping lemma is defined as follows

Definition

Let L be an infinite regular language accepted by a DFA with m states. Then any string w in L with at least m symbols can be decomposed as $w = xyz$ with $|xy| \leq m$ and $|y| \geq 1$ such that

$$w_i = xy^iz$$

is also in L for all $i = 0, 1, 2, 3, \ldots$

This is necessary, but not sufficient for a language to be regular. If a language does not satisfy the pumping lemma, then it cannot be regular. But a language that satisfies the pumping lemma may not be regular. Therefore, you can use the lemma to prove that a language is **not** regular, but not that it is regular.

Lecture - A7: Pushdown Automata

13:00 24/10/24 Janka Chlebikova

There are many context-free languages that cannot be recognised by a finite automaton, especially non-regular languages. This is because they typically require an infinite memory to be recognised, such as the language $\{a^n b^n | n \geq 0\}$. A more powerful model of computing is known as Non-deterministic Pushdown Automata, or NDPA.

Non-Deterministic Pushdown Automata

An NDPA is like a finite automata, except that they also have memory in the form of a stack, where they can store an infinite amount of information. They are made up of the input tape, an infinite sequence of symbols from the input alphabet, a finite control automaton which is similar to an NFA, and the memory. Since the memory is modelled as a stack, it works in a Last In, First Out way, and there are 3 stack operations which can be performed at each step.

- pop– Reads the top symbol and removes it from the stack
- push– Writes a new symbol on to the top of the stack
- nop– No operation, has no effect on the stack

The symbols contained in the stack are different to the alphabet of the language the NDPA recognises. The initial state of the automaton is that the stack only contains the initial stack symbol, and that the control automaton is in it's initial state.

Transition Functions

At each step, the current state, input symbol and symbol at the top of the stack determine the transition to the next state. Each transition must change the state, may read a symbol from and advance the input tape, and change the stack using one of the above stack operations.

Each transition function has three inputs

- The current state, p
- The input character, a or Λ
- The stack character, A

and two outputs

- The new state
- A stack operation

NDPA Formally

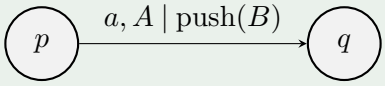
Definition

An NDPA can be formally described by

- A finite set Q of states, the initial state and the set of accepting states
- A finite set Σ of symbols making up the input alphabet
- A finite set Γ of symbols making up the stack alphabet, and the initial stack symbol, often $\$$
- A finite set of transition instructions, or a transition function T of the form $T : Q \times \Sigma \cup \{\Lambda\} \times \Gamma \to \Gamma^* \times Q$

Example

The transition



can also be written as the function

$$T(p, a, A) = (\text{push}(B), q)$$

or as the transition instruction

$$(p, a, A, \text{push}(B), q)$$

Instantaneous Descriptions of NDPAs

During computation, you need to be able to describe the NDPA after any given set of inputs. Since it also modifies the stack at each step, the entire stack needs to be included in the description. Formally, this is known as the **instantaneous description** of an NDPA.

Definition

The instantaneous description consists of

- The current state
- The unconsumed input characters
- The contents of the stack

and is written as

$$(\text{current state}, \text{unconsumed input}, \text{stack contents})$$

In this case, the top of the stack is on the left, and bottom on the right.

Example

$$(0, abba, YZ\$)$$

Accepted Languages

A string is accepted by an NDPA if there is some path from the initial state to an accepting state which consumes the entire input string. Otherwise, the string is rejected by the NPDA. The language is then the set of all strings that it accepts.

Specifically, an input string is rejected if it finishes reading the string before reaching an accepting state, if the current state has no transition for the current input or stack symbol, or finally if it attempts to pop an empty stack.

Example

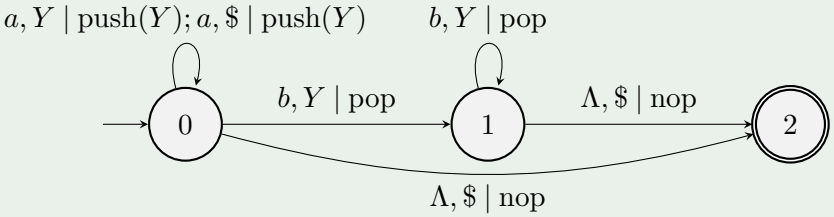
If we wanted to build an NDPA that recognises the language $\{a^n b^n | n \geq 0\}$, we could use the following plan.

- Read the string, and for each a that's read, push Y onto the stack
- Once the first b is read, change state and start removing one Y from the stack for each b read
- If the input string ends and the stack has just emptied, the string is accepted
- Otherwise, the string should be rejected

We can create this NDPA as the automaton with

- $Q = \{0, 1, 2\}$ where 0 is the initial state, and 2 accepting
- $\Sigma = \{a, b\}$
- $\Gamma = \{Y, \$\}$

and drawn as



NDPAs and Context-Free Languages

The class of all languages accepted by NDPAs is exactly the same as the class of context-free languages. To prove this, we need to show that

1. With any given NDPA, we can find a context-free grammar which produces the same language
2. With any given context-free language,we can find an NDPA which accepts the given language

Example

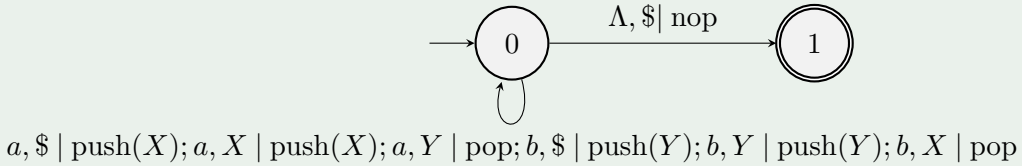
Show that the language containing all strings over $\{a, b\}$ with exactly the same number of as and bs is context-free. Using the theorem, all we need to do is find an NDPA which recognises this language.

Since we need to keep track of the number of as and bs, we can change the stack accordingly. If we've seen more as, add an X, and if we've seen more bs, add a Y.

We can then represent this NDPA as

- $Q = \{0, 1\}$ where 0 is the initial state, and 1 accepting
- $\Sigma = \{a, b\}$
- $\Gamma = \{X, Y, \$\}$

with the transition diagram



Determinism

Similarly to finite automata, push-down automata can be either deterministic or non-deterministic. However, a non- deterministic push-down automata isn't very useful, since it can never make a choice at each state. Unlike NFAs and DFAs, deterministic push-down automata cannot recognise the entire family of

context-free languages.

Lecture - A8: Applications of Context-Free Grammars

13:00 24/10/24 Janka Chlebikova

The original purpose of context-free languages was to describe the grammar of English, in terms of it's block structure, built up recursively from smaller phrases. However, since phrases with the exact same structure can have one of several meanings, we need a way to show which of the possible derivations was used to produce it. This can be shown using a parse (or derivation) tree, which starts with the initial symbol and works down until all of the leaves are terminal symbols (usually words in a spoken language).

- The start symbol of the grammar becomes the root of the tree
- For each production $X \to Y_1 \dots Y_n$, add $Y_1 \dots Y_n$ as children of the node X
- Every leaf must end up as a terminal, and any internal nodes must be non-terminals

If there are multiple possible parse trees for a given string in a language, how do we know which one is correct? Well, all of them are. This means that for any string with a non-unique parse tree, there are several correct meanings.

Ambiguous Grammars

If a string in a grammar has a non-unique parse tree, then it is known as **ambiguous**. This is because there would be no way for a compiler or other computer program to know which of the options is correct. As a human, you can usually figure it out based upon other context, but that is not a luxury afforded to computers.

Conversely, if each string of a language has only one parse tree (or alternatively, if there is only one left-most (or right-most) derivation for each string), then the language is unambiguous.

Since there are no general techniques for handling ambiguity, it is impossible to convert an ambiguous grammar into an unambiguous one. In some cases, a grammar can be modified such that it produces the same language, but is unambiguous, however this can only be done on a case-by-case basis. An example of this is adding parenthesis in mathematical expressions to make sure that the correct order of operations is used.

Parsing

Parsing is one of the main steps of a compiler, and is a process in which

- It is determined whether the input string has correct syntax
- A parse tree is built, which represents the internal structure of a string, and how it was derived from the grammar

Parsing can be done in one of two ways– top-down parsing and bottom-up parsing.

Top-down parsing constructs the parse tree by starting from the grammar's start symbol, and replacing non-terminals, working it's way towards the string. If, after following every logical set of non-terminals, it is not possible to produce the input string, then the string cannot be parsed and it is not in the language of the parser.

Bottom-up parsing starts with the string, and works backward to get to the start symbol, replacing sets of terminals with a grammar rule which produces them. Once again, if all combinations fail then the string cannot be parsed and is not in the language of the parser.

When working with a top-down parser, there is often a need to backtrack through the parse tree, if it gets stuck at some point and cannot progress any further towards the final string, which means that the whole state at each step needs to be stored until the parse is completed. Some grammars are backtrack-free, and it is sometimes possible to design a grammar purposefully like that.

When bottom-up parsing, the internal state also needs to be stored, but there is usually less backtracking required, depending on the grammar. They can also handle a larger class of grammars than top-down parsers.

Computer Languages

Since a parser is one of the most important parts of a compiler, it is desirable to design the programming language in such a way that it is easy to construct a parser for it. This means that a lot of programming languages take the form of a context-free grammar, but more specifically there are categories of CFG that are easier for either a top-down or bottom-up parser to parse.

The best candidates for these easier languages are either $LL(k)$- or $LR(k)$-grammars, for any $k \geq 0$.

- $LL(k)$-grammars are based on $LL(k)$-parsers, and are better for top-down parsing
- $LR(k)$-grammars are based on $LR(k)$-parsers, and are better for bottom-up parsing

$LL(k)$-grammars

What does $LL(k)$ actually mean?

- L– The input string is parsed from left-to-right
- L– Only left-most derivations are considered at each step
- k– The number of look-ahead symbols needed to decide which derivation to use

Example

The grammar $S \to aSc|b$ for the language $LL(1) = \{a^n bc^n, n \geq 0\}$ only requires one symbol of look-ahead, since it is always possible to know which production needs to be used to continue parsing any given string.

Example

The grammar $S \to AB, A \to aA|a, B \to bB|c$ requires two symbols of look-ahead, since there are multiple possible productions for any given symbol. If we had the input symbol a, we could have used either of the two productions of A.

There are some languages that cannot be represented by an $LL(k)$ grammar, such as if any of the productions start with a non-terminal, since you would need to know the length of the full string. That means that it is impossible to do a partial derivation, knowing only how the string starts.

All $LL(k)$ grammars are unambiguous, and are a subset of deterministic context-free languages. $LL(1)$ languages are very popular, as they are much easier to parse than other languages. Some languages however are too complex to be described by an $LL(1)$ grammar, such as the C language, however these can be parsed using higher order grammars.

$LR(k)$-grammars

Languages based upon $LR(k)$ grammars are parsed bottom up, scanning the string from left-to-right, as with $LL(k)$ grammars, but produce a right-most derivation in reverse.

- L– The input string is parsed from left-to-right
- R– Only right-most derivations are considered at each step
- k– The number of look-ahead symbols needed to parse the grammar

Donald Knuth proved that all $LR(1)$ languages are exactly $LR(k)$ languages, which are exactly deterministic context- free languages (DCFL). DCFGs are a proper subset of the context-free grammars, and can be recognised by DPDAs. They are also always unambiguous.

They are of particular interest since they can be parsed in linear time, and a parser can be automatically generated from the grammar using a parser generator.

To derive an $LR(k)$ grammar, you start from the bottom up, essentially scanning from left-to-right, until you find the first right-hand side of a production. You then replace the symbols in the string with the left-hand side of the production, and continue the process on the new string. This continues until you eventually cannot find any more productions (in which case, the string does not belong to the language), or you reach the start symbol. You can then invert the order of derivations to find the original derivation of the string.

Lecture - A9: Turing Machines

13:00 07/11/24 Janka Chlebikova

The main difference between finite and pushdown automata is that PDA have an infinite stack of memory. This does make them more powerful than FA, but since they still can only move the reading head to the right at each step, there is a class of languages they cannot parse. For example, NDPAs can parse the language $\{a^n b^n, n \geq 0\}$, but not $\{a^n b^n c^n, n \geq 0\}$.

If the main differentiating factor is the type of storage available, then we can say an automaton with no storage is an FA, if the storage is a stack, it's a PDA, but what if we had 2 or 3 stacks? They would certainly be a more powerful form of automaton, and theoretically able to parse more complex languages.

Turing Machines

Turing machines are the most powerful class of automaton, since they are essentially the same model of computing as modern computers. The main improvement over PDAs is that the memory is modelled as an infinite 'tape' of data, which can be read from and written to, but all of it is accessible at once. The tape 'head' can move left or right, or not change position on each read or write operation.

The tape is divided into cells, with an infinite number in each direction from the starting cell. Each cell contains either a symbol from the alphabet, or a blank symbol. Since this is based more in reality than previous automatons, the tape must have a finite number of non-blank symbols written on it. The head can only ever read or write to the current cell.

Based upon the current state and symbol in the current cell, the machine may

- Change the state
- Move the head in either direction (or not at all)
- Rewrite the current symbol, or leave it unchanged

Again, since these are more based in reality, the 'standard' type of Turing machine is deterministic, since it would have to be to exist in the physical world.

Instructions

Each movement step is represented by a letter– move to the left is L, move to the right is R, and stay still/ do nothing is S.

Definition

Each instruction for a Turing machine consists of five parts–

- The current state of the machine (over the set of all states, Q)
- The symbol read from the current cell of the tape (from Γ, or the blank symbol)
- A symbol to write to the tape (from Γ, or the blank symbol)
- The state to transition to (from Q)
- The direction for the tape head to move

Each instruction is of the form

$$T : Q \times \Gamma \to \Gamma \times Q \times \{L, R, S\}$$

And the instruction can be written as an instruction such as

$$(i, a, b, L, j)$$

Or in a diagram in the form

$$\frac{a}{b, L}$$

Values on the Tape

An input string is represented by placing the letters from the string into adjacent cells on the tape. All other cells in the tape initially contain the blank symbol, which is denoted by $\square$. Typically, the head starts over the leftmost cell of the input string (the leftmost non-blank cell).

Starting and Stopping

As with FAs and PDAs, there must be a single start state, which must be specified. In the case of a Turing machine, there is also usually only one final state, usually known as 'Halt'. A Turing machine will halt when it either enters the halt state, or enters a state from which there is no valid move.

Recognising Languages

A Turing machine T recognises a string over Σ if and only when

- T starts in the initial position, with the string x written on the tape
- T halts in a final state

T is said to recognise the language A if x is recognised by T, if and only if x belongs to A. A string is rejected by T if it never halts, or it halts in a state which is not final.

Instantaneous Description

To describe a Turing machine at any instant in time, we need to know what's on the tape, where the tape head is located, and what state the control automaton is in. We can represent this as follows:

$$\text{State } i : \square aa\mathbf{b}ab\square$$

where the current position of the tape head is in bold.

Add example 2 here

Lecture - A10: Computing with TMs & Alternative Definitions

13:00 07/11/24 Janka Chlebikova

We've shown so far that a Turing machine is able to take an input string, and return a boolean value, representing if the string was accepted by the language of the machine. But they can also do much more, such as transform a string, and write the result onto the tape. These machines are sometimes known as transducers.

Functions with Turing Machines

The input of the function can be the input string x, and the output can be the string y which is written to the tape at the point the machine halts. We can define a partial function $T(x) = y$ for all strings x for which the machine halts. We can also represent non-negative integers in other ways, such as in unary or binary.

Machine for adding 2 example

Non-Deterministic Turing Machines

A non-deterministic Turing machines is similar to a deterministic one, but with a finite number of choices at each combination of current state and symbol. A non-deterministic TM accepts the input string if there is at least one computation which reaches the halt state for the input.

Surprisingly, there is no difference in the set of languages which deterministic and non-deterministic Turing machines can accept. This means that they accept the same family of languages, which are of the unrestricted grammar also known as recursive enumerable languages.

Lecture - A11: More About Turing Machines

13:00 14/11/24 Janka Chlebikova

Some Turing Machines have inputs for which they will never halt. There are 3 possible outcomes for a given string– halting, finishing in a non-halted state, or not halting at all. Of these, only if the TM halts does it accept the input strings. A TM that never halts for any input string can be called a 'dancing' Turing Machine.

Recursive and Recursive Enumerable Languages

Definition

A language L is recursive if L is the set of strings accepted by some TM which halts for every given input.
A language L is recursively enumerable if L is the set of strings accepted by some TM.

We can then say that if L is a recursive language, then

- If $w \in L$ then a TM halts in a final state
- If $w \notin L$ then a TM halts in a non-final state

if L is instead a recursive enumerable language, then

- If $w \in L$ then a TM halts in a final state
- If $w \notin L$ then a TM halts in a non-final state **or** loops forever

Every recursive language is therefore also recursive enumerable, but not the other way. There are plenty of Languages which are not recursive enumerable, and as such cannot be described by a grammar. Every regular language is recursive, as there must be a DFA to represent it and a DFA must always halt.

The Final Chomsky Hierarchy

add info from slide 11

The Universal Turing Machine

The Universal Turing Machine (UTM) is a turing machine that can perform the job of any other turing machine. Given an arbitrary TM M and an input w, then U simulates the operations of M on w.

- U must halt on an input if and only if M halts
- If M accepts a string, then U must accept it
- If M rejects a string, then U must reject it

We can imagine this UTM as a TM with three tapes, where

- Tape 1 corresponds to M's tape
- Tape 2 contains M's program, which U executes
- Tape 3 contains the encoding of the state M is in at any point during the simulation

A UTM acts as a general-purpose computer, and can store and execute any arbitrary program from it's tapes.

Part II

Part B

Lecture - B1: Computability and Equivalent Models

13:00 21/11/24 Janka Chlebikova

Computability

Something is computable if– there is some computation which computes it, and if it can be described by an algorithm. Therefore, we can say that a computation is the execution of an algorithm.

Models of Computation

There are many different models of computation, some of which are more 'powerful' than others. There are several models which are all equivalent, and the most powerful models. Anything that is intuitively computable can be computed by a Turing machine (Church-Turing Thesis).

This means that no one has invented a computational model more powerful than a Turing machine, but there are several other models which are equivalent to a Turing machine.

Simple Programming Language

The **simple language** is a small imperative language introduced by Stepherdson and Sturgis in 1963. it has the same power as a Turing machine, so they can solve the same problems as each other.

Informal Description:

- Variables which take the values in the set of $\mathbb{N}$ natural numbers
- While statement of the form 'while $\mathrm{var} \neq 0$ do **statement** od.'
- An assignment, which can only assign 0, increment or decrement the variable
- A statement is either a while statement, an assignment or a sequence of statements separated by semicolons
- A simple program is a statement

Markov Algorithms

Markov algorithms were introduced by Markov in 1954. They are also equivalent in power to Turing machines. A markov algorithm over an alphabet Σ is a finite ordered sequence of productions $x \to y$ where $x, y \in \Sigma^*$.

Some productions may be labelled with halt, but it is not required. If there is a production $x \to y$ such that x occurs as a substring of w, then the leftmost occurrence of x in w is replaced by y. The algorithm transforms one string over Σ^* into another string over Σ^*, and so it computes a function from Σ^* to Σ^*.

add execution

Post Algorithms

A **post algorithm** is another string processing model, which is also equivalent in power to Turing machines.

Lecture - B2: Computability and Equivalent Models II

13:00 21/11/24 Janka Chlebikova

Recursive Functions

The basic concept of a computable function is that there must be a finite procedure to follow in order to compute the value of the function for any given input.

Mathematicians created a formal definition of a class of function whose values can be calculated using recursion, partial recursive functions. Typically, the functions of the following forms are computable– $f(x) = 0$, $g(x) = x + 1$ and $h(x, y, z) = x$. Functions such as these can be combined together using simple rules to construct all computable functions.

Primitive Recursive Functions

If you consider the function $\exp(x, y) = x^y$, then you can say that $x^0 = 1$, $x^1 = x$, $x^2 = x \times x$, ..., $x^y = x \times x \times \ldots$ (y occurrences of x). Two 'rewriting rules' are then enough to define the function,

$$x^0 = 1$$
$$x^{y+1} = x \times x^y$$

If you then consider multiplication–

$$x \times 0 = 0$$
$$x \times (y + 1) = x + x \times y$$

and addition–

$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$

This is effectively rewriting the function as two rules, one for $y = 0$ and one for $y > 0$, where y acts as a 'countdown' for the number of remaining computation steps. There are then three basis functions– successor, zero and projections– and two ways of building new primitive recursive functions from old ones– composition and primitive recursion.

- The zero function– $\text{zero} : \mathbb{N} \to \mathbb{N}, \text{zero}(x) = 0$
- The successor function– $\text{succ}(x) = x + 1$
- The projection function– $\text{project}_i(x_1, \ldots, x_k) = x_i, i \in \{1, \ldots, k\}$

composition

Not all functions are primitive recursive, such as the Ackermann Function

Minimisation

Recursive Functions

A function is partial recursive if it can be built using the base functions, and composition, primitive recursion and minimisation. A function is computable by a Turing machine if and only if it is partial recursive.

Lecture - B3: Diagonalisation and the Halting Problem

13:00 28/11/24 Janka Chlebikova

There are limits to computation, and not all problems are algorithmically solvable. This also supports the Church-Turing thesis, since it shows that if a problem cannot be solved by a turing machine, it likely cannot be solved by any form of computer.

These problems which cannot be solved by a computer are known as 'undecidable problems'.

Diagonalisation

The initial proofs for undecidability used a technique known as self-reference or diagonalisation. Any computational model that is powerful enough to allow self-reference will cause problems. To demonstrate this, we will use the Barber paradox.

Barber Paradox

Suppose there is a small town in which

- There is a single barber
- Every man is clean-shaven, either by shaving themselves or by going to the barber
- The barber obeys the rule 'Shave all and only those men in town who do not shave themselves'

This then presents a problem if we need to know if the barber shaves himself.

- If the barber shaves himself, then according to the rule he does not shave himself
- If the barber does not shave himself, then according to the rule he does shave himself

There is no solution to this problem, and so it is a paradox.

Countable Sets

How can we prove that two sets are the same size, without counting the elements? If the sets are both finite, we can pair one item from each set, and if we can pair each item then they are the same size. If a set is infinite, does the set have the same size as $\mathbb{N}$, or is it larger?

Example

The set of all even numbers is the same size as $\mathbb{N}$, since they can be paired as

$$1-2$$
$$2-4$$
$$3-6$$
$$4-8$$
$$\dots$$

A set is known as **countably infinite** if it is the same size as $\mathbb{N}$. A set is countable if it is finite or countably infinite, meaning that you can construct a numbered list of all it's elements. The set of integers, the set of

odd numbers and the set of rational numbers are all countable.

However, the size of the power set of $\mathbb{N}$ is not countable, which we can prove using a method known as diagonalisation.

Example

To prove that the set $P(\mathbb{N})$ is not countable, we can use proof by contradiction.
Suppose that the set is countable, so we can write down a list of all the subsets of $\mathbb{N}$. It might look like

$$1 - \{2, 3\}, 2 - \{4, 7\}, 3 - \{2, 4, 6, 8\}, \ldots$$

So, we have a function $f : \mathbb{N} \to P(\mathbb{N})$ that maps the numbers to sets such that every set appears in the list.
If we now define a set T such as: add i to the set T when $i \notin f(i)$, e.g. $1 \in T$ because $1 \notin f(1)$, $3 \in T$ because $3 \notin f(3)$, etc.
T cannot be in $P(\mathbb{N})$ because it is different from every set, but T is a subset of $\mathbb{N}$. This is a contradiction, and so $P(\mathbb{N})$ is not countable.

Decision Problems

A decision problem is any which asks a question with a yes or no answer. Examples

A decision problem can be viewed as a language where the members of the language are instances whose answer is yes, and non-members are instances whose output is no. A decision problem is decidable if there is an algorithm which for every input instance of the problem, halts with a correct answer, either yes or no. If no such algorithm exists, then the problem is undecidable.

Decidable problems correspond to recursive languages. They can be recognised by Turing machines which halt for every input.

Partial Decision Problems

Definition

An undecidable problem is partially decidable if there is an algorithm which
- Halts with the answer yes for instances that have the answer yes, but
- May run forever for instances which have the answer no

(Or the other way around in some cases)

Partial decidable problems correspond to the recursive enumerable languages, which can be recognised by Turing machines.

Existence of an undecidable problem

How many Turing machines exist? If we let S be a countable set of symbols, then any TM can be coded as a finite string of symbols over S, with all transition functions one after the other. This means there are a countable number of finite strings over S, and therefore a function which maps a unique number to each TM. This means that the set of all TMs is countable.

How many languages exist? A language is a subset of a countable set of strings. We have shown that the size of the set of all subsets of $\mathbb{N}$ is not countable. This means that the set of language is uncountable!

So, we've shown that the set of all TMs is countable, and sot the set of languages accepted by TMs is also countable, but that the set of all languages is uncountable. Therefore, there is a language which cannot be recognised by any TM, and therefore is an undecidable problem.

The Halting problem is famous because it was one of the first problems to be proven to be algorithmically undecidable.

Lecture - B4: Undecidable Problems

13:00 28/11/24 Janka Chlebikova

The Halting Problem

Algorithms may contain loops which may be finite or infinite. The amount of work done by an algorithm usually depends on the input data.

The halting problem asks the question 'Is there an algorithm which can decide whether the execution of an arbitrary program halts on an arbitrary input?'

You might initially think that you can simply run the program with the given input. This works if the program stops, since it then clearly halted. But if the program doesn't stop within a reasonable length of time, we have no way to know if it will halt at some point, and if we've not waited long enough, or if it will simply run forever.

The halting problem is undecidable, or more accurately it is partially decidable, since it will always eventually halt if the answer is yes and we just run the program. Therefore, there is no algorithm which could solve it.

Acceptance Formulation

Define the set $A = \{< M, w >: M \text{ is a TM that accepts } w$, where $< M, w >$ is a unique coding. This is another formulation as the halting problem; is there a TM which will recognise the set A? No.

This proof is once again done by contradiction.

Suppose there is a machine **Solver** for which on every input w and every TM M, would tell us if M accepts w. If we build another TM **Opposer** which does the following

- Take the input w and determines the TM $< w >$ which w encodes
- Ask Solver for the answer, e.g. 'does the TM $< w >$ accept w'
- If Solver accepts, reject
- If Solver rejects, accept

Opposer must be a valid TM, since Solver always halts.

But what if the input to Opposer is the encoding of Opposer itself?

- Opposer asks Solver for an answer for itself
- If Solver claims that Opposer accepts, then Opposer rejects
- If Solver claims that Opposer rejects, then Opposer accepts

This is a paradox, assuming that Solver exists, and so Solver cannot exist!

Lecture - B5: Introduction to Computational Complexity

13:00 05/12/24 Janka Chlebikova

Theoretically, any decidable problems are solvable by Turing machines and therefore modern computers. However, it is not always practical to solve these problems due to limited time, memory, storage, etc.

Developing a Solution

Developing a solution to a problem typically involves at least 4 steps–

- Designing an algorithm or procedure for solving the problem
- Analysing the correctness and efficiency of the algorithm
- Implementing the algorithm in some programming language
- Testing the implementation

Time Complexity

Informally, one may describe a program or algorithm as being 'fast' or 'slow', but the actual execution time of an algorithm depends on many different factors. This includes– the algorithm itself, the programming language used to implement the algorithm, the quality of the algorithm, the computer actually running the code, and the size of the input to the algorithm.

Formally, when analysing the efficiency of an algorithm you would focus on the 'speed' (or complexity) of the algorithm as a function of the size of the input upon which it is run. The time-complexity function $T(n)$ of an algorithm expresses the number of operations which the algorithm needs to execute to get the result for an input of size n. You almost always refer to the worst-case time complexity of an algorithm, as then the algorithm must always finish in that time or less, for all inputs of size n.

Example

With the array sum problem (given a list of n integers, return their sum), how does the time taken scale with the size of the input?

- Problem– Add all n integers in the array S
- Inputs– A positive integer n, and an array of numbers S indexed from 1 to n
- Outputs– An integer, the sum of the integers in S

We may chose to implement this using a for-loop over the array which adds the current integer to a running total at each step. If we implement it using this method, we would have 4 main steps, each of which taking a different length of time– Initialising the function (t_1), setting up the loop (t_2), iterating the loop and adding to the total (t_3), finalising the function and returning the result (t_4).

- Operation– Adding the current item to the running total is the most expensive (in terms of time) operation in the implementation
- Input size– The size of the input is proportional to n, the number of items in the array
- Time complexity function– The time required is dominated by the addition operation t_3, which is called n times and so, $T(n) \approx t_1 + t_2 + t_3 \times n + t_4$

We can then simplify $T(n)$ by removing the less significant times, in this case anything that requires constant time, and get $T(n) = A \times n$ where A is a suitable constant corresponding to the number of primitive operations needed to perform the addition.

Example

Add key searching example

As in the previous example, there may be multiple perfectly valid solutions to a problem, but there is often only one which is optimal, with the lowest time-complexity. It is almost always desirable to reduce the time-complexity of an algorithm, especially if the algorithm is going to be run repeatedly or on a slow computer.

The Travelling Salesman Problem

Problem– There are n cities labelled $C_1, C_2, \ldots, c_n$ for which the distance $d_{i,j}$ between any two cities C_i and C_j is known. Find the shortest possible path that visits each city exactly once.

Brute-Force

One possible solution to solve this problem would be to brute-force every possible routing, and pick the one of the shortest length. But, this may be inefficient depending upon how many possible routes there are. Clearly, there are

- $n - 1$ ways to select the first city
- $n - 2$ ways to select the second city
- And so on, until there is only one city left to visit

That means that the number of possible routes would be

$$\frac{(n-1) \times (n-2) \times \cdots \times 1}{2} = \frac{(n-1)!}{2}$$

which is clearly a factorial function, which grows rapidly.

If we were to implement this brute-force function, we would see that each possible route requires n additions, and there are $(n-1)!$ routes, so it would require $n \times (n-1)!$ additions. The time-complexity of the algorithm would therefore be $T(n) = A \times n!$. This is obviously a very inefficient algorithm, and would quickly become infeasible to run on a computer with a relatively small number of cities.

Other Algorithms

There are various other algorithms which can solve the travelling salesman problem, but they all have a limit to the number of cities before they two become infeasible.

- Various branch-and-bound algorithms can be used with up to 86000 cities
- Progressive improvement algorithms can work well for up to 200 cities
- An exact solution was found for 15112 cities using a solution based on linear programming

Lecture - B6: Asymptotic Growth

13:00 05/12/24 Janka Chlebikova

There are usually several algorithms which can solve any given problem. Ideally, we want to use the algorithm which has the lowest time-complexity, and uses the least memory.

add time-complexity graph

There are two factors two consider when comparing the time-complexity of two algorithms–

- It is usually important to know how fast the time taken to run an algorithm grows with the size of the input to the function
- Counting the basic steps in a time-complexity function does not give a completely accurate picture, since it depends heavily on the programming language, compiler, and computer used to implement the algorithm, but the difference is at most a constant factor

We can see from this that comparing the time complexity of the algorithms is equivalent to comparing the asymptotic growth of the time-complexity functions.

Asymptotic Analysis

If we have two algorithms with the time complexities $T_A(n) = n + 10$ and $T_B(n) = n$, then the growth of the functions is the same, and for large values of n, $n \approx n + 10$.

If the two algorithms instead had the time-complexities $T_A(n) = 4n^2 + 3n + 10$ and $T_B(n) = 2n^2$, then the growth is once again the same, as for large values of n, $4n^2 + 3n + 10 \approx 2n^2 \approx n^2$.

If we wanted to formally express the rate of growth of a function, we want to keep the dominant term with respect to n , but ignore any constants around it. We also want to formalise that an $n \log 10n$ algorithm is better than an n^2 algorithm.

Big-O

One way of formalising this is the Big-O notation. $O(\dots)$ is known as an asymptotic upper bound of a function.

Informally, when f, g are two non-negative functions, then $f(n)$ is $O(g(n))$ if f grows at most as fast as g. Formally, $f(n) = O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \leq c \times g(n)$.

We write $f(n) = O(g(n))$ or $f(n) \in O(g(n))$ and read this as '$f(n)$ is big O of $g(n)$'.

Example

$2n^2 + 10 = O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that $c \times g(n) \geq 2n^2 + 10$ for all $n \geq n_0$.
Therefore, $2n^2 + 10 = O(n^2)$ since $3n^2 \geq 2n^2 + 10$ for all $n \geq 4$, hence $c = 3$ and $n_0 = 4$. But we could also use $c = 100$ and $n_0 = 1$, since this still holds the rule.

So, Big-O gives an upper-bound on the growth of f, but not necessarily a tight one.

Big Omega (or Ω-notation)

Big Omega is known as an asymptotic lower bound of a function.

Informally, when f, g are two functions, then $f(n)$ is $\Omega(g(n))$ if f grows at least as fast as g. Formally, $f(n) = \Omega(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \geq c \times g(n)$.

We write $f(n) = \Omega(g(n))$ or $f(n) \in \Omega(g(n))$ and read this as '$f(n)$ is big omega of $g(n)$'.

Example

$4n^2 - 10 = \Omega(n^2)$ if there exists $c, n_0 \in \mathbb{R}^+$ such that $c \times g(n) \leq 4n^2 - 10$ for all $n \geq n_0$.
Therefore, $4n^2 - 10 = \Omega(n^2)$ since $n^2 \leq 4n^2 - 10$ for all $n \geq 2$, hence $c = 1$ and $n_0 = 2$.

Big Theta (or Θ-notation)

Big Theta is known as the asymptotic tight bound of a function.

Informally, when f, g are two functions, then $f(n)$ is $\Theta(g(n))$ if f is essentially the same as g, to within a constant multiple. Formally, $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

We write $f(n) = \Theta(g(n))$ or $f(n) \in \Theta(g(n))$ and read this as $f(n)$ is big theta of $g(n)$

Example

add big theta example

finish off from slide 17

Lecture - B7: Analysis of Algorithms

13:00 12/12/24 Janka Chlebikova

Sorting Algorithms

- **Problem**– Sort n integers in ascending order
- **Inputs**– Positive integer n, array S of integers indexed from 1 to n
- **Output**– The array S containing the integers sorted in ascending order

There are many algorithms which can be used to sort arrays, each of which has a different time-complexity. Each algorithm is not explained, as they have been covered in previous modules. Any new algorithms are explained fully.

Bubble Sort

The algorithm makes use of two nested for-loops, one of which repeats $n-1$ times, and the other repeats $n-i$ times for every loop of the other for-loop, i.e. $\sum_{i=1}^{n-1}(n-i)$.

If you then simplify this, it works out as

$$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

and so, $T(n) = \Theta(n^2)$.

Exchange Sort

The first unsorted element is compared to every subsequent element, and if they need to be, they are swapped. This is repeated until no swaps are needed, which indicates that the list is sorted. After each iteration, the next smallest item is moved to the correct position.

Once again, the algorithm makes use of two nested for-loops, one of which repeats $n-1$ times, and the other repeats $n-i$ times for every loop of the other for-loop, i.e. $\sum_{i=1}^{n-1}(n-i)$.

If you then simplify this, it also works out as

$$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

and so, $T(n) = \Theta(n^2)$.

Insertion Sort

This algorithm makes use of a for-loop with a nested while-loop. This once again works out such that $T(n) = \Theta(n^2)$, since it uses nested loops. Any algorithm with nested loops, each iteration of which takes constant time, will end up running in exponential time, as each loop runs in linear time, which are effectively multiplied by being nested.

Merge Sort

If the size of the array is already 1, then the algorithm completes in constant time, as the array is already sorted. The time taken to sort an array of size n is roughly $2T(\frac{n}{2})$, since it splits the array into two smaller arrays of size $\frac{n}{2}$.

We then need to merge the two arrays together, which requires n comparisons between the items in each of the arrays, and so runs in linear time. Since each level of recursion spits the problem in half, the number of recursions needed is the logarithm to the base 2 of n, $\log_2 n$.

Since the two steps are linear and logarithmic time, when we multiply the two together, we end up with the final $T(n) = \Theta(n \log_2 n)$.

Comparison

Since $n \log_2 n$ grows slower than n^2 asymptotically, the merge sort algorithm is more efficient than the others, at least in the worst case.

Towers of Hanoi

It is not immediately obvious that there is a general solution for n disks, but it can actually be solved recursively. In general, $T(n) \leq 2 \times T(n-1) + 1$, e.g. $T(3) \leq 2 \times T(2) + 1$. The lower bound for this recursion is when $n = 1$, since only 1 move is needed to move the single disk. Therefore, we can say that the number of steps would be

$$T(1) = 1$$
$$T(n) = 2 \times T(n-1) + 1$$

Lecture - B8: Problem Complexity and Classification of Problems

13:00 12/12/24 Janka Chlebikova

As well as the complexity of the algorithm, it is sometimes useful to be able to talk about the complexity of a problem. There is often no way to give a definitive complexity, but we can at least find the upper and lower bounds.

Upper Bounds

Upper bounds are typically defined by the most efficient known algorithm, and each time a more efficient algorithm is found, the upper bound decreases to that point. For example, if the first known algorithm A to solve a problem Q has a time-complexity of $O(n^3)$, then the upper bound for Q would also be $O(n^3)$. Later, an algorithm is discovered which can solve Q in $O(n^2)$ time. The upper bound for Q therefore becomes $O(n^2)$. Each successive algorithm improvement moves the upper bound downwards.

Lower Bounds

Lower bounds are typically defined by what can be proven about the problem. For example, if a problem Q is proven to not be solvable in less than linear time $\Omega(n)$, then the problem complexity cannot be lower than $\Omega(n)$. It is later proven that the problem cannot be solved in less than $\Omega(n \log_2 n)$ time, and as such, the lower bound for the complexity becomes $\Omega(n \log_2 n)$. Each successive proof moves the lower bound upwards.

Finding these proofs is typically very difficult, as it must be generalised for all possible algorithms that can solve the problem. As the lower bound approaches the upper bound, it becomes harder and harder to prove.

Open Complexity

The complexity of most problems is not fully known. In some cases, the upper and lower bound reach the same complexity, in which case that is exactly the problem complexity. In the vast majority of cases, there's a difference between the upper and lower bounds, meaning that more experimentation and research needs to be done into the topic to be certain.

There are some simple problems which we do know definitively the complexities for–

- Searching an unordered list of items– Upper and lower bound are linear, so problem complexity is $\Theta(n)$
- Searching an ordered list of items– Upper and lower bound are logarithmic, so problem complexity is $\Theta(\log n)$
- Sorting an arbitrary array– Upper and lower bound are both logarithmic, so problem complexity is $\Theta(n \log n)$

From this, we can also see that the merge sort algorithm is already as efficient as possible at sorting an arbitrary array, as it's time-complexity is $\Theta(n \log n)$.

Decision Trees

Decision trees can be used to prove a lower bound as being logarithmic. Since we can often use a tree to represent the decision process that takes place in an algorithm, it is a useful tool to see how many decisions are actually required to solve a problem. If each step has only two outcomes, then it is a binary decision tree. We may need a tree with more possible outcomes, in which case we can use a tree with a higher degree, where there are at most k outcomes at each step, for some constant k.

The depth of a decision tree often corresponds to the lower bound for the complexity of a problem, as the worst-case scenario for the run time is just the maximum depth. The depth is certainly a lower bound on the actual running time, which is typically good enough to prove the lower bound for the complexity of a problem.

Why Logarithms?

If a problem has n different outputs, then any decision tree for the problem must have at least n leaves. Since the number of leaves in a binary tree of height h is at most 2^h, $2^h \geq n$, and so the height of the decision tree must be at least $\lceil \log_2 n \rceil$ or $\Omega(\log n)$.

Intractable Problems

We've seen decidable and undecidable problems so far, but some problems are partially decidable, meaning there is an algorithm which may return yes, but may never return no for any given input.

Decidable problems can also be split into three categories–

1. (Proven) Intractable– Solvable, but impractical
2. (Apparently) Intractable– Problems which *appear* to be intractable but which have not been proven so
3. Tractable– Practically solvable

Proven Intractable Problems

There are two types of problems in this category, those which require a non-polynomial amount of time to solve, such as the travelling salesman problem and the towers of Hanoi. Both are proven to be solvable, but require unreasonable amounts of time, requiring $(n-1)!$ and $2^n - 1$ steps to solve, respectively.

There are also problems which **do not** require a non-polynomial amount of time, but which we can prove cannot be solved in polynomial time. Only a few of these problems are known to exist, and all of them were created with the sole intention of proving their existence, not to solve a real problem.

Apparently Intractable Problems

In the case of these problems, no polynomial-time algorithms have been discovered, but it has also not been proven to not exist. Examples of this include the Hamiltonian cycle and travelling salesman problems.

Tractable Problems

Tractable problems are those that already have a polynomial-time algorithm to solve them.

Decision Problems

All previous discussion includes any types of problem, including decision problems, but there are some unique features of decision problems. They are all those problems whose outputs are simply yes or no. Many problems can also be re-written to become decision problems. For example, we could re-write the

travelling salesman problem as– Given n cities, the pairwise distances between them and a constant $d > 0$, is it possible to find a round trip with a total length less than d?

Complexity Classes

Decision problems can be separated into so-called complexity classes, P, NP and $NP - \text{complete}$.

P

The class P is the set of all decision problems that can be solved by polynomial-time algorithms. That means that it is also just the set of tractable decision problems. It does not contain proven intractable or undecidable problems, as are proven to not be solvable in polynomial time.

NP

The class NP is the set of all non-deterministically polynomial problems. That means that there is a non-deterministic algorithm which can solve the problem in polynomial time. A non-deterministic algorithm allows at every possible step multiple continuations. These algorithms accept an input if there exists a sequence of choices for which the algorithm returns yes. What this really means is that the algorithm must be able to produce a yes within polynomial time, but may take longer to produce a no.

An NP solution can be thought of as having two stages–

- Guessing Stage– Make a guess at a solution, using any means
- Verification Stage– A deterministic algorithm checks if the solution completes in polynomial time, definitely halting in every case where the answer is yes

One example of this is the decision version of the travelling salesman problem, as we only need to be able to decide if the solution is correct in polynomial, not actually come up with the solution.

Lecture - B9: P, NP and NP-Complete Problems

13:00 19/12/24 Janka Chlebikova

P and NP

To prove a problem is in P, you must write an algorithm which can solve it in polynomial time. To prove a problem is in NP, you mist write an algorithm which can check a given yes-solution in polynomial time.

Everything in P is in NP, such as $P \subseteq NP$, since to be in P the problem must be solvable in polynomial time with a deterministic Turing machine, which is also clearly a non-deterministic Turing machine.

So far, no one has found a problem in NP which is definitely not in P, but also it has not been proven that all problems in NP are also in P. The consensus so far is that $P \neq NP$, but it is still as yet unproven.

NP-complete

Within the NP problems, some are more 'difficult' than others. In 1979, Cook discovered that there are a number of problems which are the hardest in NP, known as the NP-complete problems. The first problem shown to be NP-complete was the satisfiability problem.

Satisfiability Problem– Given a boolean expression written using only AND, OR, NOT, variables and parentheses, is there an assignment of True/False values to the variables which makes the entire expression true? Every such boolean expression is equivalent to the one in conjunctive normal form.

The base problem is in NP, as the checking stage can be performed in $O(n)$ time. A restricted form of the problem, the 3-satisfiability problem, is NP-complete.

NP-complete problems are all equivalent in the sense that if any one of them is in P, then all of them are. If any NP-complete problem is shown to have a polynomial time solution, we can deduce that $P = NP$.

Any NP problem can be solved by an exponential algorithm, but for none of them a polynomial algorithm is known. Further, no-one has been able to prove that no polynomial algorithm exists for any of those problems.

Definition

A decision problem, B is NP-complete if
- $B \in NP$
- $A \leq B$ for all problems where $A \in NP$, i.e. NP-complete problems are problems in NP to which all other NP problems can be reduced in polynomial time

To prove that a problem is NP-complete–
1. Prove that the problem is in NP
2. Find an NP-complete problem A which can be polynomially reduced to the problem

Polynomially Reducible

Given two problems, a polynomial-time reduction is an algorithm which runs in polynomial time and reduces one problem to the other. Suppose that we know how to solve problem B and want to know how to

solve problem A. What we need to find is a polynomial reduction which transforms the an input I_A of A to the input of I_B of B in such a way that B's answer to I_B is the same as A's answer to I_A.

Then, if we have an input I_A to A, we can transform I_A to an input I_B for B, and use the algorithm for B to get the desired output. Therefore, if we had a polynomial algorithm for B, we would have a polynomial algorithm for A as well.

add example?

NP-hard

A problem is NP-hard if all NP problems can be polynomially reduced to it. An NP-hard problem does not need to be in NP, and it does not need to be a decision problem. For example, the original TSP is NP-hard, and the decision version of the TSP is NP-complete.

Lecture - B10: Tackling NP-complete & NP-hard Problems

13:00 12/12/24 Janka Chlebikova

If the problem is known to be NP-complete or NP-hard, then we know there is no point looking for an optimal solution. Instead, it may be useful to look for an algorithm which comes close to the real answer, or which can get the correct answer for a subset of inputs.

Heuristic Solutions

An algorithm which works 'reasonably well' for many instances, but for which there is no proof that the algorithm is always fast or always produces a good solution. This also contains genetic algorithms.

Approximation Algorithms

An algorithm which finds a solution that is 'close' to optimal in polynomial time, for every instance. There must be a proof that these properties are true. For any given problem, 'close' may have a different meaning, either due to the nature of the problem, or the needed accuracy of the solution. The 'closeness' could mean anything arbitrarily close to the optimal solution, such as $1+\epsilon$ for any ϵ, or a constant factor, or even worse.

Restricting the Input

Some NP-hard and NP-complete problems can be in P, if we solve them only for a subset of the inputs. For example, the 3-satisfiability problem is NP-complete but the 1- and 2-satisfiability problems are in P.

Parametrisation

There are often fast algorithms to solve problems, if certain parameters are fixed, or have a small value.

Probabilistic Solutions

There are some algorithms which are usually correct, but give an incorrect answer in a small number of cases.