THEORETICAL COMPUTER SCIENCE FOR IT SECURITY

LIBFEXDLMCSETCSITS01_E

LIBF

LEARNING OBJECTIVES

On completion of this unit, you will be able to

- understand limitations of data structures, algorithms and computation in general
- use formal languages and automata to solve security problems
- use machine learning techniques in data analysis
- use logic and knowledge representation
- understand the principles of program analysis and verification

UNIT 1

ALGORITHMS AND DATA STRUCTURES

STUDY GOALS

On completion of this unit, you will be able to ...

- understand different data structures
- know what are algorithms and general types of them
- understand constitutive elements of graphs and trees
- understand algorithms for sorting and searching
- understand how to analyze algorithms

1. ALGORITHMS AND DATA STRUCTURES

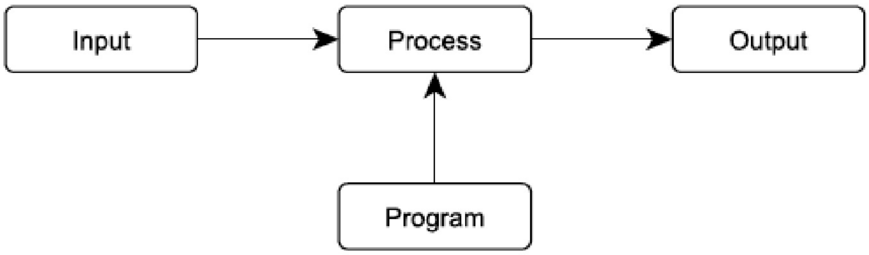
Introduction/Case study

Computer science deals with the processing of information with the help of a digital computer. A digital computer differs from an analog computer by the encoding of the processed information using discrete states. In contrast, an analog computer represents the information to be processed by analog quantities (e.g. by voltages). Another key feature of digital computers is their programmability. In addition to the incoming information, a digital computer also requires a program that determines how the information is processed. In summary, their exists a relationship between the processed information and the program.

Figure 1.1 shows an abstract model of a computer as Input-Process-Output (IPO). The data is flowing from left to right. The input data is available in a chosen format (data structure). This data is processed, dependent to a program that is written in a specific programming language (e.g. Java).

That programming language is defined by a formal language. After finishing the process step, the output is produced.

Figure 1: Figure 1.1: IPO Model



The topics of computer science are oriented to basic structures of an information-processing system. Technical computer science deals with the functional structure of computers (hardware). On the other hand, practical computer science concentrates on the development of programs (software).

Therefore, general methods for program development are considered (software engineering), as well as special solutions for standard problems of information processing. Fundamental questions of computer science such as the classification of tasks with respect to their degree of difficulty and the limits of information processing are dealt within the context of theoretical computer science.

Outline

Section 1.1 defines terms like algorithm and introduces different data structures.

In section 1.2 are introduced directed and undirected graphs with their structures of representation. Additionally, specific graphs called trees are introduced. Section 1.3 describes different algorithms for sorting e.g. numbers and searching patterns in texts. Section 1.4 explains, how algorithms can be analyzed concerning time and space complexity and defines different notations for that.

1.1 Algorithms, Programming Languages and Data Structures

Algorithms and data structures are related in computer science. Data structures are managing data using basic data types. Whereas, algorithms define the operations for processing the managed data. For e.g. solving a problem both are “working” together. The questions are which data structures exist and how to decide which are appropriate for a given problem? Afterwards, similar questions have to be answered for algorithm(s). The questions are which algorithms exist and which one is used for what data structure resp. (class of) problem? An additional question is: which algorithm or data structure is the “best”? Before answering that, criteria like time and space are required. Therefore, this question is skipped for now (cf. section 3). Lets take a look at the term algorithm.

DEFINITION: ALGORITHM

An algorithm is a step-by-step procedure for solving a problem respectively a class of problems in a finite number of steps. It takes a finite amount of initial input(s), processes them unambiguously at each operation, before returning its outputs within a finite amount of time. (Webster, 2021) and (Vault, 2021)

Some parts of this definition reveal that an algorithm is a kind of general recipe, consists of steps/operations, is executed step-by-step, is solving a problem or class of problems and returns within finite amount of time.

Thus, an algorithm defines an accurate and finite processing specification and realizes an input to output relationship. Additionally, correctness and efficiency are important properties for algorithms. Another definition of the term algorithm is:

DEFINITION: ALGORITHM

In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. Algorithms are always unambiguous and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks. (Wikipedia, 2021)

This definition explicitly adds tasks. Thus, an algorithm can also be referenced to a specific task like calculating the greatest common divisor of two natural numbers by euclidean algorithm. Algorithm 1 shows an divisionbased implementation expressed in pseudocode. The input parameters for the algorithm in function euclid(a, b) are two natural numbers a and b. In line 1, the remainder of the division a and b is calculated and assigned to r. If $r = 0$ the loop (lines 2 to 6) is skipped and b is returned as greatest common divisor. Otherwise, the loop is processed until this condition is fulfilled (line 2). In lines 3 and 4, b is assigned to a and r to b. In line 5, the remainder of a divided by b is calculated and then assigned to r. Finally, the algorithm returns the calculated greatest common divisor b (line 7).

Algorithm 1 Euclidean Algorithm - Iterative euclid(a, b)

```
Input: a, b {Two natural numbers a and b}
1: r ← a mod b
2: while r /= 0 do
3:   a ← b
4:   b ← r
5:   r ← a mod b
6: end while
7: return b
Output: {Greatest common divisor of a and b}
```

iterative
Iterative describes a repeated process to approximating solution.

An example for calculating the greatest common divisor is given by the function call euclid(42,56). The **iterative** implementation is illustrated in table 1.1. The step-by-step calculation terminates with greatest common divisor 14 for natural numbers 42 and 56 (cf. table 1.1, bold number 14).

Table 1: Table 1.1: Iterative Calculation with Euclidean Algorithm

iteration	r	a	b
0	42 (42 mod 56)	42	56
1	14 (56 mod 42)	56	42
2	0 (42 mod 14)	42	14

The algorithm 1 is an iterative implementation of euclidean algorithm.

Whereas, algorithm 2 calculates the greatest common divisor in a **recursive** way. Lines 1 and 2 declare the “stop criterion” for the recursive function calls of euclid(a, b). Line 4 defines the recursive call of euclid(a, b) with parameters b as a and $amodb$as b. Iterative and recursive algorithms are equivalent. This means here that both can be used to solve the problem “greatest common divisor”. It is mentionable that recursion offers the possibility to describe an infinite number of computations by a finite recursive algorithm without using loops.

recursive
Recursive means that a function is called within their own definition.

Algorithm 2 Euclidean Algorithm - Recursive euclid(a, b)

```
Input: a, b {Two natural numbers a and b}
1: if b = 0 then
2: return a
3: else
4: return euclid(b, a mod b)
5: end if
Output: {Greatest common divisor of a and b}
```

The mentioned example is also used for the recursive calculation of the greatest common divisor of 42 and 56. This is depicted in table 1.2.

Table 2: Table 1.2: Recursive Calculation with Euclidean Algorithm

euclid(42,56)	= euclid(56,42)
	= euclid(42,14)
	= euclid(14,0)
	= **14**

After the illustration by example, main questions for a given algorithm to solve a single problem or class or problems are

- What is the intention of the algorithm? ⟶ specification
- Does the algorithm really do what it is expected to do? ⟶ verification
- How about the efficiency of the algorithm? ⟶ algorithm analysis

The specification can be taken into account for formalizing the problem that the algorithm solves. Therefore, specifications can be informal as text or formal (e.g. using mathematics). Formal specification are appropriate if correctness proofs are executed to verify the satisfaction of the algorithm regarding the specification. Thus, evidence is provided that the algorithm do what he is intended to do. The last question refers to the consumption of time (how long will the algorithm run) and space (how much memory is used), the algorithm requires. This depends also on the selected data structure the algorithm is processed on.

As mentioned, algorithms operate on data. This data can be structured or unstructured. The focus is here on structured data. Therefore, data structures are required that are appropriate for a given problem. A definition of the term data structure is

DEFINITION: DATA STRUCTURE

A data structure is a systematic way of organizing and accessing data. (cf. (Goodrich, Tamassia, & Goldwasser, 2014, p. 150))

Arrays
An Array is an indexed collection of arbitrary data.

In the beginning, fundamental data structures involving array and linked list are described. **Arrays** sequentially store a collection of items. An array a e.g. of numbers is described as

$$\begin{array}{rl} i: & \;\;0 \;\;\; 1 \;\;\; 2 \;\;\; 3 \;\;\; 4 \\ a = & [3,\ 5,\ 2,\ 17,\ 1] \end{array}$$

In general, arrays store values at different positions and an array has the size of n representing the number of stored entries. These entries can be accessed via the index i with $i = 0, \ldots, n-1$. Thus, a value at index i is accessed by $a[i]$. In the example, the array has five items, consequently, the size of the array is $n = 5$. Consequently, the range of the index $i = 0, \ldots, 4$ and $a[0] = 3, \quad a[1] = 5, \ldots, \quad a[4] = 1$. Arrays store arbitrary values like integers, characters, objects, and so on.

Additionally, two-dimensional arrays are introduced. These arrays are also called matrices. Before an array, $a[]$ is like a vector, whereas arrays $a[][]$ are like matrices. $a[i][j]$ depicts the value at point of intersection of row index i and column index j. The size of the two-dimensional array is calculated by multiplying the number of rows and columns. The following example shows a 3×5 array with natural numbers.

$i \backslash j$	0	1	2	3	4
0	3	5	2	17	1
1	9	45	276	7	4
2	7	15	65	11	8

The two-dimensional array stores the dimensions $3 \cdot 5 = 15$ entries/values.

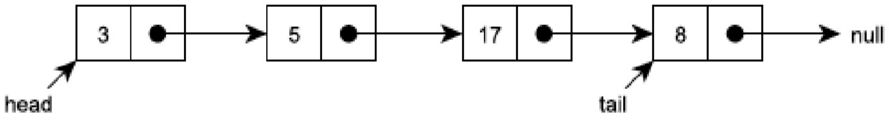
Thus, values are $a[0][0] = 3, \quad a[0][1] = 5, \quad a[0][2] = 2, \ldots, a[2][4] = 8$. For example, the value in 3rd row ($i = 2$) and 2nd column ($j = 1$) is $a[2][1] = 15$.

Arrays are the right choice for storing data in a certain order. The drawbacks of arrays are that the size of the array has to be defined at creation, and deleting or inserting data at interior positions can be time-consuming because of shift operations. Therefore, the linked list will be introduced.

linked list
Linked lists are a linear sequence of nodes with a reference from one node to the next node.

The **linked list** is a collection of nodes storing data and simultaneously a reference to the next node. The reference can also be “null” if it exists no successor node. Figure 1.2 illustrates a linked list using natural numbers.

Figure 2: Figure 1.2: Linked List with Natural Numbers



The linked list starts with a reference to the head of the list, here the node with value 3. This node has the value 3 and a reference to the next node with value 5. This node has a reference to node with value 17 and this to the node with value 8. This node is the last node of the linked list and so called tail. The tail refers to "null" meaning that no further nodes are present.

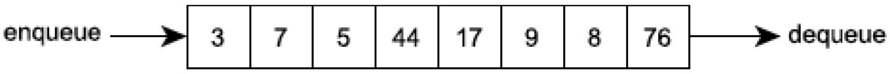
As in arrays, linked lists can also store arbitrary data in the nodes. The benefit in linked list is that the size has not be defined at creation. The list can grow or shrink at runtime. If a new entry is added, this entry can be attached as node at the beginning or at the end. The insertion at the start of the list means becoming the new head, whereas at the end means becoming the new tail. In case of deletion, the references have to be changed. If the head is deleted, the next node becomes the head. If the tail is deleted, the node before has to reference to "null". For deleting an intermediate node, the example in figure 1.2 is used. Lets assume the node with value 5 is deleted. Thus, the reference from node with 5 to node with 17 has to be shifted to node with 3. Ending in referring from node with 3 to node with 17. Afterwards, the node with value 5 can be deleted.

More advanced data structures are queue, stack and heap. Graphs and trees are excluded but are described in section 1.2. **Queues** can be used to implement memory structures that aim to store data in a certain order until it is processed further. The stored data is read out again in the same order in which it was stored. This is called FIFO (First-In-First-Out).

Queues
Queues are data structures processing in FIFO principle.

Figure 1.3 shows a queue with natural numbers.

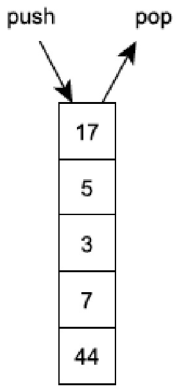
Figure 3: Figure 1.3: Queue with Natural Numbers



New entries can be enqueued by inserting entries (from logically left). Following FIFO, the first enqueued entry will be dequeued next (from logically right). In the example, a new entry can be enqueued before the value 3.

If an algorithm wants to processes the next element of a queue, the 76 is dequeued. Thus, a queue is appropriate e.g. for a web server responding to requests. The requests will be enqueued and one after another processed by dequeue the queue.

Figure 4: Figure 1.4: Representation of a Stack



stack
A stack is a linear list implementing the LIFO principle.

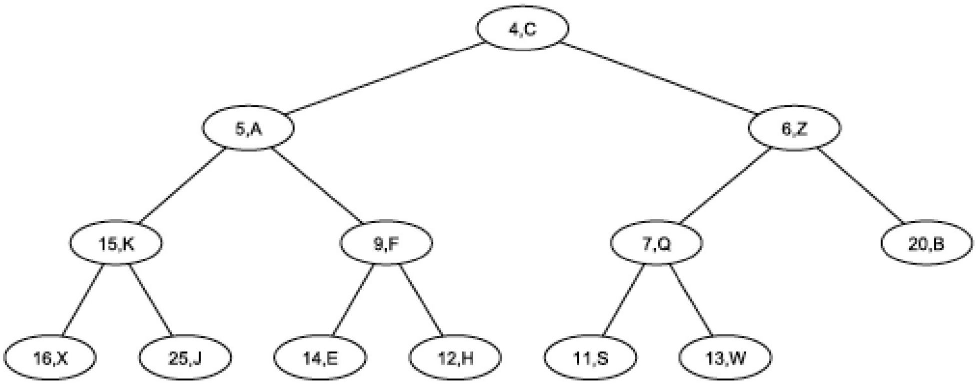
Another data structure is a **stack**. On a stack, data is stored by simply stacking on top of one another. This concept is used e.g. in microprocessors and determines the processing sequence of instructions fed to the processor. Data is stored and read out again according to the LIFO principle (Last-In-First-Out). This concept of data storage is frequently used in automata theory. Another special feature of the stack is that it is also used in most microprocessors with its special memory form, since this enables information to be managed close to the hardware.

Figure 1.4 depicts a stack to explain the principle of operation. At this example, on the stack are stored natural numbers. A new entry can be inserted via push to the top of the stack, whereas pop releases the topmost entry (here 17) from the stack. Reading the entry 44 means that every single entry on the stack has to be released (pop) before.

heap
A heap is an ordered binary tree structure.

Finally, the data structure **heap** is introduced. A heap is a data structure that usually resembles a binary tree. Binary means that each node has not more than two successors. Data elements can be placed in a hierarchical structure and read out accordingly. Basically, the heap follows two properties: the heap-order and the shape of the heap (binary tree) itself, cf. (Goodrich et al., 2014, p. 370). The heap-order defines that “in a heap, for every position p other than the root, the key stored at p is greater or equal to the key stored at p’s parent” (Goodrich et al., 2014, p. 370). This is true for min-heaps. In max-heaps the keys are ordered the other way around. Thus, in max-heaps the root has the maximal key value. Figure 1.5 shows an example of a min-heap as binary tree.

Figure 5: Figure 1.5: Binary Heap with $key, \quad value$ (min-heap)



The example, shows in each node a tuple $(key, \quad value)$ with the key as integer and a character representing the $value$ (data). Further deep dive concerning the structure of trees is given in section 1.2.

The section concludes with an overview regarding **programming languages**. Programming languages differ between imperative and declarative paradigms. Imperative programming use instructions that change the program/data. In contrast, declarative programming describes what the program should achieve without instructions how to do that. The focus in theoretical computer science is not on different programming languages like Java, Python and so on. It focuses on formal languages to define the syntax and semantics for specific programming languages. Therefore, it is not dependent to the mentioned paradigms.

programming languages
Programming languages are based on their formal language.

1.2 Graphs and Trees

The description of real world problems or scenarios as graph gives the opportunity to deal with many different challenges. Before some scenarios are described, the basic notations are introduced. In general, graphs differ between directed and undirected graphs. A directed graph is a tuple $G = (V, R, \alpha, \omega)$ with

- V is an not empty set of vertices resp. nodes.
- R is a set of relations resp. directed edges with $R \subseteq V \times V$.
- $V \cap R = \emptyset$
- $\alpha: R \rightarrow V$ and $\omega: R \rightarrow V$ ($\alpha(r)$: head (vertex), $\omega(r)$: tail (vertex))
- $g^{+}(v) = |\{r \in R: \alpha(r) = v\}|$ is the number of outgoing directed edges of vertex v
- $g^{-}(v) = |\{r \in R: \omega(r) = v\}|$ is the number of incoming directed edges of vertex v

Additionally, two vertices v, u are adjacent if a directed edge $r \in R$ exists so that $\alpha(r) = u$ and $\omega(r) = v$ or $\alpha(r) = v$ and $\omega(r) = u$. A directed edge can also be written as tuple $r = (u, v)$ meaning that head is u and tail is v. Furthermore, a **subgraph** is defined as $G_s = (V_s, R_s, \alpha_s, \omega_s)$ of graph $G = (V, R, \alpha, \omega)$ if $V_s \subseteq V$ and $R_s \subseteq R$.

subgraph
A subgraph is a partial graph of a graph.

Figure 6: Figure 1.6: Directed Graph

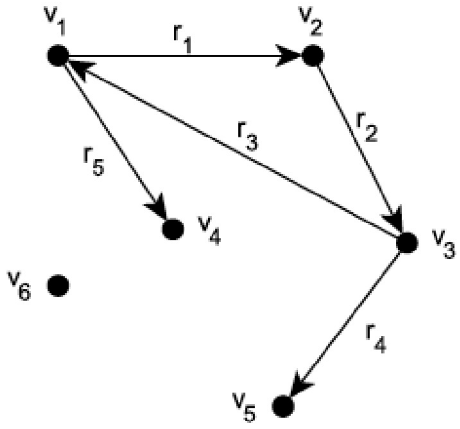


Figure 1.6 shows a directed graph with vertices $V = \{v_1, v_2, \ldots, v_6\}$ and directed edges $R = \{r_1, r_2, \ldots, r_5\}$. The directed edge e.g. r_3 has $\alpha(r_3) = v_3$ and $\omega(r_3) = v_1$. An example of a subgraph is $G_s = (V_s, R_s, \alpha_s, \omega_s)$ with $V_s = \{v_1, v_2, v_4\}$ and $R_s = \{r_1, r_5\}$. Note: Based on the definition of a graph, even a single vertex without any directed edges, such as v_6, is also a subgraph.

For representing and storing graphs, suitable data structures are the incidence and the adjacency matrix or adjacency list. Thereon, algorithms process for different tasks. The **incidence matrix** I for fig. 1.6 has dimension $|V| \times |R|$ (here 6×5) consisting of

incidence matrix
An incidence matrix represents a graph with vertices as rows and edges as columns.

$$\begin{array}{c|ccccc} v_i \backslash r_j & r_1 & r_2 & r_3 & r_4 & r_5 \\ \hline v_1 & 1 & 0 & -1 & 0 & 1 \\ v_2 & -1 & 1 & 0 & 0 & 0 \\ v_3 & 0 & -1 & 1 & 1 & 0 \\ v_4 & 0 & 0 & 0 & 0 & -1 \\ v_5 & 0 & 0 & 0 & -1 & 0 \\ v_6 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The incidence matrix I contains an 1 at each intercept point of v_i and r_j if $\alpha(r_j) = v_i$, an -1 if $\omega(r_j) = v_i$ and otherwise 0. Graphs with loops ($\alpha(r) = \omega(r)$) are not considered for simplification. In a directed graph, the sum of the values in a specific column has to be 0. This means that each directed edge has a head and tail referring vertices. As mentioned, the size of an incidence matrix depends on the number of vertices (rows) and directed edges (columns) $|V| \times |R|$. If the graph is dense (many directed edges) this structure grows for fixed number of vertices.

Whereas, the **adjacency matrix** A for a graph (cf. fig. 1.6) has dimension $|V| \times |V|$ (here 6×6) consisting of

adjacency matrix
An adjacency matrix is a symmetric matrix with vertices as rows and columns.

$$\begin{array}{c|cccccc}
v_i \backslash v_j & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\
\hline
v_1 & 0 & 1 & 0 & 1 & 0 & 0 \\
v_2 & 0 & 0 & 1 & 0 & 0 & 0 \\
v_3 & 1 & 0 & 0 & 0 & 1 & 0 \\
v_4 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_5 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_6 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

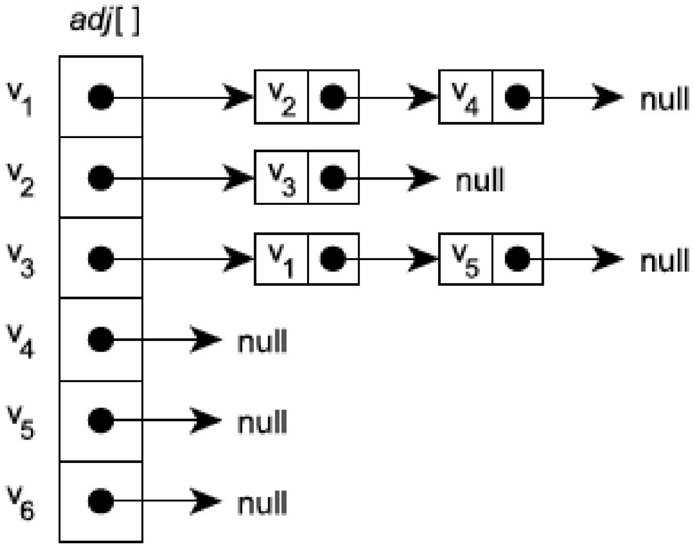
The adjacency matrix A contains at each intercept point a_{ij} the number of directed edges with $a_{ij} = \left|\left\{r \in R{:}\, \alpha\left(r\right) = v_i \wedge \omega\left(r\right) = v_j\right\}\right|$. In other words, an entry a_{ij} in the matrix shows the number of directed edges originating in vertex v_i and ending in vertex v_j. For example, the entry 1 intercepting row v_3 and column v_1 represents the directed edge $r_3 = \left(v_3, v_1\right)$ in figure 1.6. As mentioned, the size of an adjacency matrix is $|V| \times |V|$. Thus, the matrix is independent of the number of directed edges. The adjacency matrix does not grow in case of a dense graph, i.e. a graph containing many edges compared to the number of vertices.

Another representation of a graph is an **adjacency list**. An adjacency list is composed of an array $adj[v_i]$ of size $|V|$ containing one entry for each vertex v_i and a linked list for each entry. The overall number of entries in the linked lists is $|R|$. The array entries are references (pointers) to the corresponding linked list. Figure 1.7 illustrates the example graph for fig. 1.6. In this example, the linked list for vertex v_1 has two entries v_2, v_4 representing both directly reachable vertices. Thus, only entries are stored in the linked list of vertex v if this vertex has outgoing edges $g^{-}(v) > 0$.

adjacency list
An adjacency list is a combination of an array and linked lists.

For example, the array elements for v_4, v_5, v_6 have no linked list and thus waste no space.

Figure 7: Figure 1.7: Adjacency List for Directed Graph



In summary, the data structures representing graphs are incidence and adjacency matrix as well as adjacency lists. Table 1.3 summarizes the required space for the structures. Notations for Θ and O will be introduced in section 3.

Table 3: Table 1.3: Summary of Representation Structures for Graphs

<table>
<tr><th>Structure</th><th>Space</th><th>(v,w) ∈ R?</th></tr>
<tr><td>Incidence Matrix</td><td>Θ(|V | · |R|)</td><td>O(|R|)</td></tr>
<tr><td>Adjacency Matrix</td><td>Θ(|V |2)</td><td>O(1)</td></tr>
<tr><td>Adjacency List</td><td>Θ(|V | + |R|)</td><td>O(g+(v))</td></tr>
</table>

This table helps for finding the appropriate representation structure. The required space is only one criterion. Further criteria have to be considered like the task (i.a. path finding, are two vertices connected?) has to be considered.

undirected graphs Undirected graphs consist of vertices and edges but edges have no head and tail.

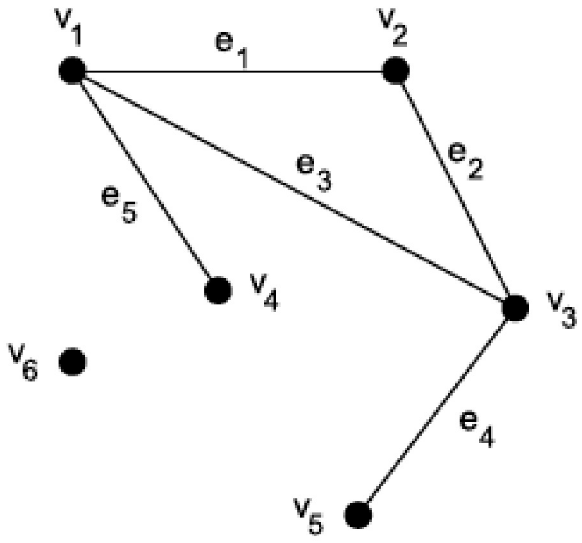
After introducing directed graphs, **undirected graphs** are described. An undirected graph is a tuple $G = (V, E, \gamma)$ with

$$\gamma : E \to \{X : X \subseteq V \text{ with } 1 \leq |X| \leq 2\}.$$

Thus, $\gamma(e) \in E$ gives the end vertices of edge $e \in E$. Additionally, undirected graphs have no head and tail. The example of the directed graph is changed to an undirected graph. This undirected graph is depicted in figure 1.8. In undirected graphs, following edges can be processed in both directions. In directed graphs, the direction is restricted from head to tail.

This is important for example at evaluation paths, if a vertex is reachable from another one.

Figure 8: Figure 1.8: Undirected Graph



The structures representing graphs are shown to compare directed with undirected graphs. The incidence matrix for undirected graph of figure 1.8 is described in the following matrix. The size of the matrix is identical to directed graphs. The difference is that the incoming edges (-1) for directed graphs are changed to 1. As mentioned, the direction for edges in undirected graphs is not given.

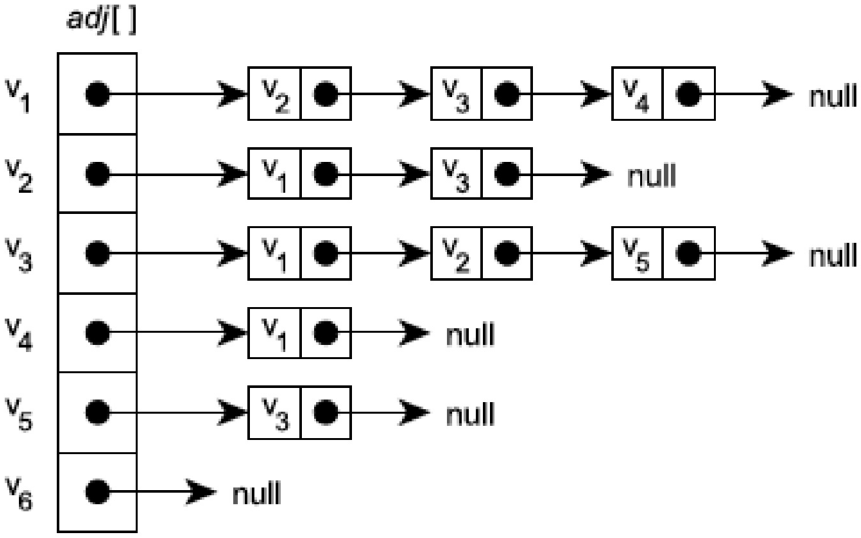
$v_i \backslash e_j$	e_1	e_2	e_3	e_4	e_5
v_1	1	0	1	0	1
v_2	1	1	0	0	0
v_3	0	1	1	1	0
v_4	0	0	0	0	1
v_5	0	0	0	1	0
v_6	0	0	0	0	0

The adjacency matrix for undirected graphs using the example of figure 1.8 is shown below. The matrix for undirected graphs is symmetric to the diagonal line from top left to bottom right. Thus, for different tasks, only the upper or lower triangle has to be processed. The size of the matrix is also identical to directed graphs.

$v_i \backslash v_j$	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	1	1	0	0
v_2	1	0	1	0	0	0
v_3	1	1	0	0	1	0
v_4	1	0	0	0	0	0
v_5	0	0	1	0	0	0
v_6	0	0	0	0	0	0

Taking a look at the adjacency list $adj[v_i]$, the size of the list grows for undirected graphs in comparison to directed graphs. The adjacency list for the graph (cf. fig. 1.8) is depicted in figure 1.9.

Figure 9: Figure 1.9: Adjacency List for Undirected Graph



Note that each edge e with $\gamma(e) = \{v, w\}$ and $v \neq w$ is represented twice, once for $w \in adj[v]$ and once for $v \in adj[w]$. For example, lets take a look at the edge e_2 connecting vertices v_2, v_3. This edge is located at $adj[v_2]$ with entry $v_3$in the corresponding linked list as well as at $adj[v_3]$ with entry v_2 in the corresponding linked list. Thus, the needed space for storing undirected graphs in an adjacency list is double compared to directed graphs.

The summary depicted in table 1.3 is also true for undirected graphs. The number of edges is then $|E|$ instead of $|R|$ and $g^+(v)$ has to be interpreted as $g^+(v) = g^-(v)$. Thus, differing between outgoing and incoming is not needed. The space in the adjacency list changes from $\varTheta(|V| + |R|)$ for directed graphs to $\varTheta(|V| + 2|E|)$ for undirected graphs.

Additionally, we define weights of edges using the mappings $w: E \to \mathbb{R}$ for undirected graphs, and $w: R \to \mathbb{R}$ for directed graphs. The weight of an edge may represent information such as bandwidth, distance, and so on. It is also possible to change the mapping to represent text instead of numbers. For example, the text represents used protocols or ports. This kind of graphs are called **weighted graph**.

weighted graph
A weighted graph has numbers or text assigned to the edges.

Furthermore, a tree T is defined as a directed graph with properties

- the graph has one vertex v with $g^-(v) = 0$ (root vertex)
- all vertices u, except root vertex, have $g^-(u) = 1$
- the graph has no cycles (or loops)

The **tree** is a binary tree if each vertex v has $g^+(v) \leq 2$. In case of an undirected graph, the graph is a tree if exactly one path exists from each vertex v to each other vertex u. Trees of undirected graphs have no defined root.

tree
A tree is a graph with specific properties.

Figure 10: Figure 1.10: Trees of Directed and Undirected Graphs

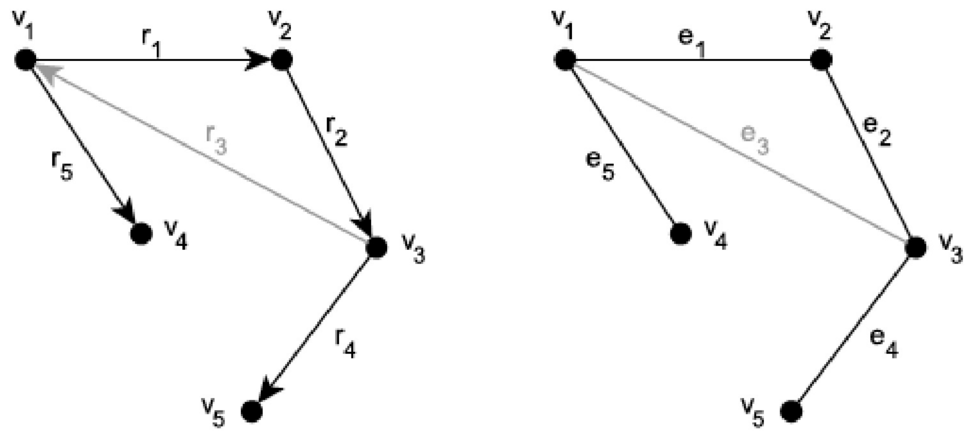
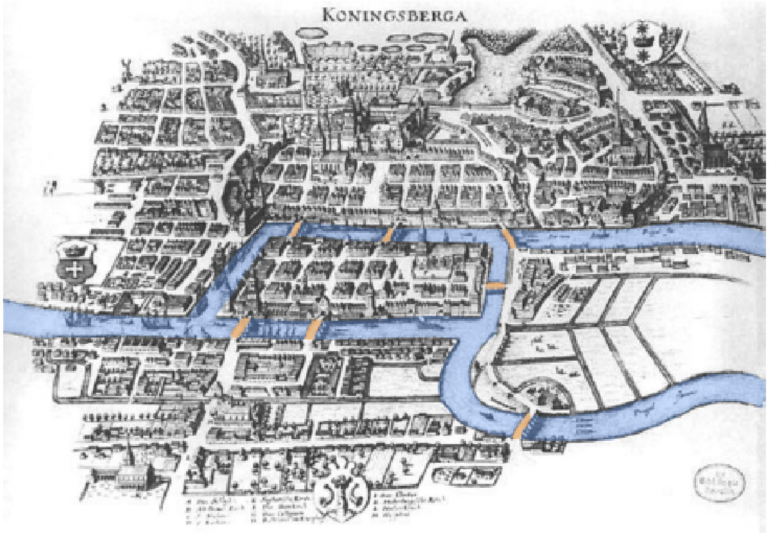


Figure 1.10 depicts trees of directed and undirected graphs. At both graphs the light grey edges (r_3, e_3) are not included. They are only visible to see the changes in the graphs depicted in figures 1.6 and 1.8. On the left side is a tree of a directed graph. The defined properties of trees in directed graphs are fulfilled by the root v_1 with no incoming edges, all other vertices u have one outgoing edge $g^-(u) = 1$ and the graph has no cycles. Hint: The tree is a binary tree because each vertex has less or equal two outgoing edges. On the right side a tree of a undirected graph is illustrated. All paths are distinct from each vertex to each other vertex.

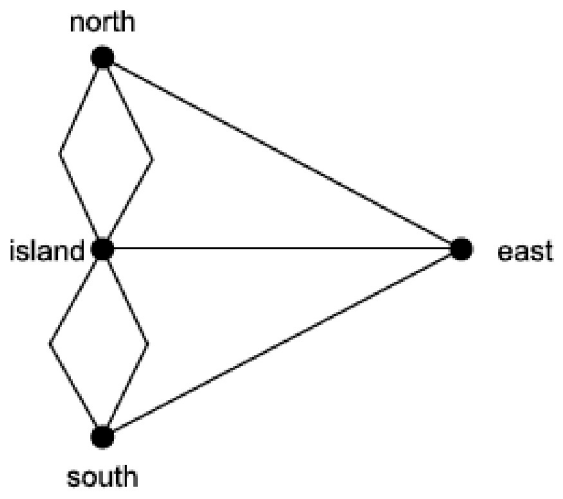
A well-known example using graphs is Seven Bridges of Königsberg which originates from Euler 1736. The river Pregel is crossed by seven bridges, and the question was whether it is possible to arrange a walk so that each bridge is used exactly once (cf. figure 1.11). Of course, one wants to be back at the starting point at the end.

Figure 11: Figure 1.11: Seven Bridges of Königsberg based on (Merian-Erben, 2019)



This problem can be modelled as a graph with parts (north, south, east and island) divided by the river as vertices and the bridges as undirected edges (cf. figure 1.12). The edges are undirected because it is not important in which direction bridges are passed. It can be proven that such an euler's circle is not possible.

Figure 12: Figure 1.12: Seven Bridges of Königsberg as Graph



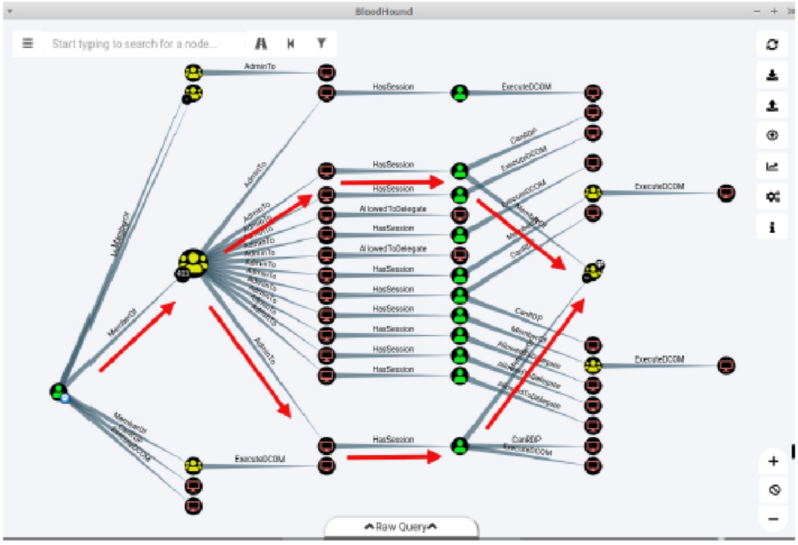
Beside this many use cases of graph theory are in the field of IT security.

For example, an attacker wants to find out highly used systems in a company.

Thus, tools using graphs illustrate systems with vertices and these systems at the same network are interconnected by edges. After analyzing the network traffic, attackers can find the most frequently requested systems. If their goal is to affect the availability of such systems whole business processes are interrupted.

Additionally, applications like BloodHound are used to visualize environments with Active Directory involved. The collected data about the network is visualized by a graph database. The application helps to identify attack paths regarding the Active Directory. This includes access control lists, users, groups, trust relationships and unique AD objects (PenTestPartners, 2021). Figure 1.13 visualizes collected network traffic and results in two possible paths to privilege escalation (red arrows).

Figure 13: Figure 1.13: Visualization with Bloodhound (Roberts, 2018)



Graphs are also used to display the distribution of malware (e.g. wormable ransomware). This can be used for simulating different attacks in penetration tests. Results can be that vulnerabilities can be closed by patches or the network has to be segmented. By the way, if not the whole network is affected the graph is a subgraph of the network. Directed graphs are used for example to visualize systems and network protocols. The graph has protocols as “weights” (labels) on directed edges and vertices for systems.

Firewall rules restrict network traffic in protocol and direction. Thus, directed edges show the traffic incoming and outgoing. Graph algorithms find anomalies if a new system is attached, unknown protocols is used, two systems use new protocols and so on. A new system in the network could be an attacker. A new protocol is an anomaly if for example traffic is sent encrypted from a system to the internet.

The examples show that graphs are a powerful approach to simulate attacks from a penetration test perspective as well as from an attacker perspective.

Let’s continue with the mentioned example of wormable malware. For example, one system has not the current security patches installed. A successful attack of that system with a wormable malware can result in many affected systems. A penetration test but also a real attack results in enormous impact. The problem can be mapped to finding a spanning tree with all infected systems. The root is the first infected system and afterwards other systems are infected in the kill chain.

1.3 Sorting and Searching

In practice, it is necessary to sort numerical or alphanumerical elements.

Sorting can be in ascending or descending order. For this purpose, some algorithms have been developed which can perform this task efficiently. The following **sorting algorithms** are selected examples with their possible implementation in pseudocode.

sorting algorithms
A sorting algorithm orders unsorted elements in ascending or descending order.

The first sorting algorithm is the naive selection sort algorithm. The algorithm iterates over the unsorted array a and has the sorted array as output.

The selection sort is defined in algorithm 3.

Algorithm 3 Selection Sort

```
Input: Array a {Unsorted array of integers}
n ← sizeof(a)
for (i ← 0; i < n; i ← i + 1) do
    minPos ← i
    for (j ← i + 1; j < n; j ← j + 1) do
        if a[j] < a[minPos] then
            minPos ← j
        end if
    end for
    swap(a[i], a[minPos])
end for
Output: Sorted Array a
```

The time complexity of the selection sort algorithm for worst-case, bestcase and average performance is given by $O(n^2)$. An algorithm, which is used for the determination of a minimum of n elements, must execute at least $n-1$ comparisons in each case. Thus, this algorithm is straight forward to implement, but it is inefficient compared to other sorting algorithms.

Another sorting algorithm is sorting by insertion. Here, n elements to be sorted are considered in sequence and inserted into the respective already sorted subsequence (initially empty) at the correct position. The insertion sort is defined in algorithm 4.

Algorithm 4 Insertion Sort

```
Input: Array a {Unsorted array of integers}
n ← sizeof(a)
for (i ← 1; i < n; i ← i + 1) do
    for (j ← i; j > 0 ⊠ a[j - 1] > a[j]; j ← j - 1) do
        swap(a[j], a[j - 1])
    end for
end for
Output: Sorted Array a
```

The insertion sort algorithm is similar to the selection sort algorithm. The time complexity for insertion sort is $O(n^2)$in average and worst-case. In best-case the algorithm requires $O(n)$ comparisons. Thus, in best-case the number of comparisons grows linear instead of growing polynomial.

Next sorting method is Bubble Sort. It is based on the idea of restricting swaps to only adjacent elements during sorting. Bubble sort is defined in algorithm 5.

Algorithm 5 Bubble Sort

```
Input: Array a {Unsorted array with integers}
n ← sizeof(a)
repeat
    swapped ← false
    for (i ← 1; i < n; i ← i + 1) do
        if a[i - 1] > a[i] then
            swap(a[i - 1], a[i])
            swapped ← true
        end if
    end for
until NOT swapped
Output: Sorted Array a
```

Bubble sort algorithm has time complexity in best-case with $O(n)$ comparisons.

Similar to the insertion sort algorithm the worst-case and average is $O(n^2)$.

divide and conquer algorithm
A divide and conquer algorithm recursively breaks down a problem into sub-problems until these can be solved directly.

Quick sort is an **divide and conquer algorithm** that is recursively implemented. Quick sort is defined in algorithm 6.

Algorithm 6 Quick Sort

```
Input: Unsorted array a, low (l) and high (h) indices referring to a
algorithm quicksort (a, l, h)
    if l < h then
        p ←partition(a, l, h)
        quicksort(a, l, p - 1)
        quicksort(a, p + 1, h)
    end if
algorithm partition (a, l, h)
    pivot ← a[h]
    i ← l
    for (j ← l; j ≤ h; j ← j + 1) do
        if a[j] < pivot then
            swap(a[i], a[j])
            i ← i + 1
        end if
    end for
    swap(a[i], a[h])
    return i
Output: Sorted Array a
```

The algorithm has lower time complexity because of the "smaller" inner loop. In worst-case the time complexity is $O(n^2)$but for best-case and average case the time complexity is $O(n \cdot log(n))$.

After introducing sorting algorithms different **searching algorithms** are described and their time complexity is mentioned.

searching algorithms
Search algorithms find elements in data structures.

One way to perform a search, which also does not require any deeper knowledge or prerequisites regarding the quantity to be searched, is the sequential or linear search. In this case, the available data is run through systematically from the beginning to the end and the values are compared in each case with the value to be found. Algorithm 7 defines the linear search.

Algorithm 7 Linear Search

```
Input: Search element s, Unsorted/sorted Array a
n ← sizeof(a)
for (i ← 0; i < n; i ← i + 1) do
    if a[i] = s then
        return i
    end if
end for
return -1
Output: Index i of searched element s in array a or -1 if not in array
```

The linear search has time complexity of $O(n)$ in worst-case, $O\left(\frac{n}{2}\right)$ in average and $O(1)$ in best-case.

In contrast to the linear search, which works for any unsorted arrays, the binary search requires a sorted array as starting point. It is based on a divide and conquer approach, where the list to be searched is first split into two parts and then only the relevant part is searched further. Algorithm 8 defines the binary search in pseudocode to search an element s in array a with values sorted in ascending order.

Algorithm 8 Binary Search

```
Input: Search element s, ascending sorted Array a
n ← sizeof(a), left ← 0, right ← n - 1
while (left ≤ right) do
    mid ← left + ((right - left)/2)
    if a[mid] = s then
        return mid
    else
        if a[mid] > s then
            right ← mid - 1
        else
            left ← mid + 1
        end if
    end if
end while
return -1
Output: Index mid of searched element s in array a or -1 if not in array
```

The binary search has time complexity in best-case $O(1)$, and in worst-case and average case $O(log_2(n))$.

Beside search algorithms for numbers also string matching algorithms are present. String matching generally tries to find a so-called “pattern” in a string or text. It extends the search for a number or a letter in a character string to the search for text segments. For this purpose, three well-known algorithms are briefly described.

The sequential string matching finds the occurrence of a pattern in the text by means of the naive procedure. First the pattern is put on, beginning with the first character of the text, before character by character from left to right a comparison takes place. In order to resolve a correspondence between pattern and text. This is done sequentially for each substring of length of the pattern until the pattern is found in the text or the end of the text is reached. If no match results, this is referred to as a "mismatch".

The sequential search is described in algorithm 9.

Algorithm 9 Sequential String Matching

```
Input: Pattern p with characters p1, ..., pm, text t with t1, ..., tn
for (i ← 1; i ≤ n -m+ 1; i ← i + 1) do
    found ← true
    for (j ← 1; j ≤ m; j ← j + 1) do
        if ti+j-1 ≠ pj then
            found ← false
        end if
    end for
    if found then
        return start index i to end index i +m- 1
    end if
end for
return -1
Output: Start and end indices of pattern in text or -1 if not in text
```

The sequential string matching algorithm shows that the pattern p must be applied $(n - m + 1)$-times to the original text t and is passed through completely in each case. This leads accordingly to $(n - m + 1) \cdot m$ comparisons.

Furthermore, the naive procedure is memoryless. This means that if necessary the same text passage is compared several times, since the procedure does not remember which characters of the text have already matched the pattern to be compared. The time complexity of the algorithm is $O((n - m + 1) \cdot m)$.

While the naive string matching procedure was still memoryless, this limitation is improved by the Knuth-Morris-Pratt algorithm. Here, we first assume that a mismatch between pattern and text occurs at j-th position.

In this case, previously considered characters $j-1$ are matching. This knowledge represent a shift of the pattern not only one position to the right (as before in the naive method), but as far as possible. The maximum possible shift is determined by taking advantage of which characters have previously shown a match before the first mismatch occurred. Algorithm 10 describes the Knuth-Morris-Pratt string matching algorithm.

The sequence of the Knuth-Morris-Pratt algorithm can be stated as follows:

1. Determination of an “end piece” with length l from the “initial piece” of the pattern with length $j-1$. The end piece is thereby also an initial piece of the pattern. Next considered position $next[j]$ is position $l+1$.
   Comparison with i-th character of the text.
2. If no mismatch occurs, the pattern is shifted one position to the right and position $i+1$ is compared with $next[j]+1$. In general, the function of $next[j]$ can be described as $next[j] = 1$ + length of the longest end piece of the first $j-1$ characters, which is also an initial piece of the pattern.
3. If a mismatch occurs, the determination of the length l' of the longest end piece of the initial piece with length $next[j]-1$ (also initial piece of the pattern) and comparison of i-th character of the text with $l'+1 = next[next[j]]$. If a mismatch occurs again, this step is iteratively executed until a match is found again.

The Knuth-Morris-Pratt algorithm consists of two steps, the prefix analysis resulting in maximal shift positions for the pattern (cf. $next[j]$) and the search itself (cf. algorithm 10). The prefix analysis has the time complexity $O(m)$ whereas the search has $O(n)$. The overall time complexity is so $O(m+n)$.

```
Algorithm 10 Knuth-Morris-Pratt Algorithm
Input: Pattern p with characters p1, ..., pm, text t with t1, ..., tn
i ← 1, j ← 1
repeat
    if ti = pj OR j = 0 then
        i ← i + 1
        j ← j + 1
    else
        j ← next[j]
    end if
until j > m OR i > n
if j > m then
    return i -m
else
    return -1
end if
Output: Index i -m of searched pattern in text or -1 if not in text
```

The Boyer-Moore algorithm no longer compares a reference text with an existing pattern from left to right, but now from right to left. In this method, the character comparison takes place starting with the last character of the pattern. If a mismatch occurs here, the pattern is shifted to the right by as many characters as necessary until a match can be found.

Algorithm 11 describes the method for Boyer-Moore.

Algorithm 11 Boyer-Moore Algorithm

```
Input: Pattern p with characters p1, ..., pm, text t with t1, ..., tn
i ← m- 1, j ← m- 1
repeat
    if ti = pj then
        if j = 0 then
            return i
        else
            i ← i - 1
            j ← j - 1
        end if
    else
        i ← i +m-Min(j, 1 + last[[ti]])
        j ← m- 1
    end if
until i > n - 1
return -1
Output: Index i of searched pattern in text or -1 if not in text
```

The Boyer-Moore algorithm works most efficiently when it finds a character in the text that does not occur in the search pattern. So the “bad character rule” kicks in. This is most likely with a relatively small pattern and a large alphabet, which makes it particularly suitable for such a case. In this case, the algorithm operates with an average efficiency (time complexity) of $O\left(\frac{n}{m}\right)$. If the algorithm searches the first occurrence of the pattern in the text the worst-case is $O(n+m)$. If all matches have to be found the worstcase is $O(n \cdot m)$. The preparation for Boyer-Moore is $O(m)$ to calculate the shifting for a given mismatch character.

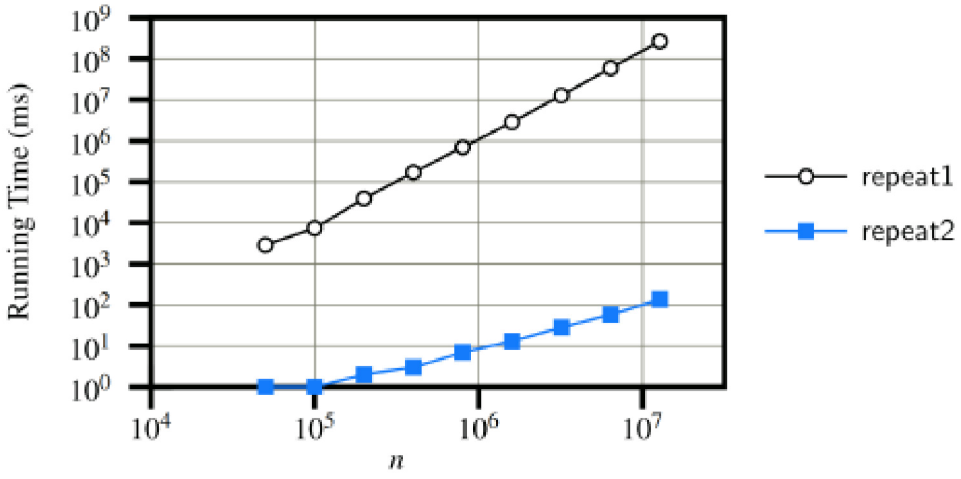
1.4 Algorithm Analysis

Algorithm analysis deals with the correctness and efficiency of algorithms.

The correctness will be explained in another section. Thus, the question arises: what is efficiency with regard to algorithms? Efficiency has two main aspects, the demand for computing time and for storage space. In general, efficiency means the rate of growth of for example the aspect computing time with increasing amount of input elements. This is called time complexity. For example, in unit 1.3 the selection sort algorithm (cf. algorithm

3) requires n^2 steps for n input elements. This leads to the general question: How to determine the number of steps? or How to estimate the time (complexity)? In this section, the focus is on time complexity whereas space complexity is similar. A naive method for analyzing the required time of an algorithm is the use of experimental studies. A program will be measured from start to termination. This elapsed time reflects the algorithm efficiency. For the analysis, different input sizes (n) are used for the same data structure to get a trend curve after a statistical analysis. The experiments should be independent and based on randomly chosen inputs. Afterwards, the measured data points with the input size n (abscissa) and the required run-time (ordinate) can be visualized in a plot (cf. (Goodrich et al., 2014, p. 151)). The naive method ("measuring") will vary from computer to computer. The CPU of a computer is shared across different processes on that computer. Thus, the analysis is dependent to other processes and the result of the experiment can vary "immense" (cf. figure 1.14).

Figure 14: Figure 1.14: Visualization of a plot for two experiments (Goodrich et al., 2014, p. 153)



The results from experimental studies are helpful to optimize productionquality code but there are limitations to their use for algorithm analysis according to (Goodrich et al., 2014, p. 153):

- Experiments of two algorithms are not directly comparable if not measured in the same soft- and hardware.
- Experiments are limited to the set of test inputs. Thus, trend curve and other important information for inputs not tested are not available.
- Experiments can only be performed on fully implemented algorithms.
  They have to be analyzed in run-time which is not possible if not implemented.

The goal for an independent way of analyzing algorithms has to fulfill the following properties (cf. (Goodrich et al., 2014, p. 154)):

1. Evaluation of the efficiency of any two algorithms independent of soft- and hardware.
2. The analysis can be done in a high-level description of the algorithm.
   The concrete implementation is not required.
3. Includes every possible input for the algorithm.

For this approach of analyzing algorithms, a couple of regulations have to be introduced. **Primitive operations** are for example assigning a value to a variable, following an object reference, an arithmetic operation, comparing two numbers, accessing an element in an array, calling a method and return from a method (cf. (Goodrich et al., 2014, p. 154)). For comparing algorithm run-times (time complexity), a function $f(n)$ is introduced.

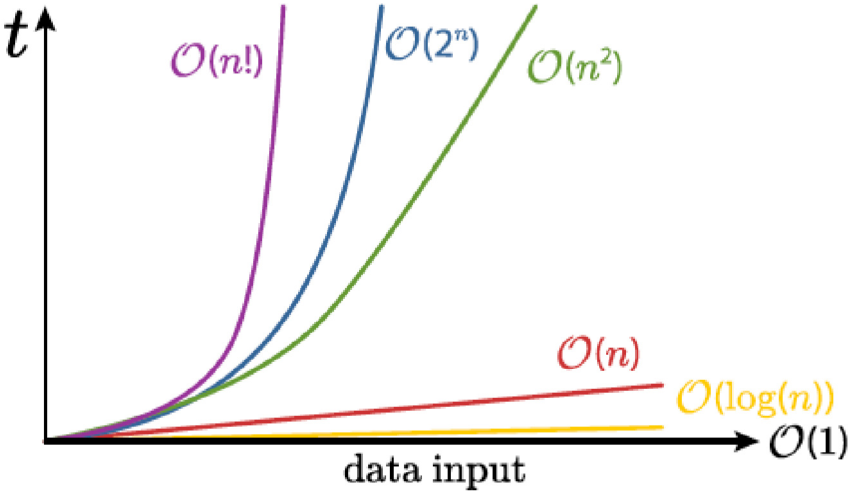
Primitive operations
Primitive operations are instructions with a constant run-time.

This function reflects the number of operations that are required for input size n.

An algorithm analysis applied to data structures and algorithms can result in different outputs for input size n. The constant function $f(n) = c$ is independent of the input size n. It will always be equal to the value c(e.g. $c = 3, c = 42$ or $c = 4^7$). The most fundamental constant function is $g(n) = 1$, so that any constant function can be written as $f(n) = c \cdot g(n)$. The constant function is used for the number of steps needed for performing a primitive operation. The logarithm function $f(n) = \log(n)$ in most cases in computer science has base of 2 (i.a. binary trees, binary values). Another function is the linear function $f(n) = n$ representing e.g. a single primitive operation for each of n elements. The $n \cdot log(n)$ function grows faster than the linear function but not so fast as the quadratic function $f(n) = n^2$. Two nested loops are an example for the quadratic function with $n \cdot n = n^2$ operations in the algorithm. The cubic function $f(n) = n^3$ is present for example using three nested loops. The class of functions for linear, quadratic and cubic functions is the polynomial function. Polynomial functions $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ . \ . \ . + a_k n^k$ consist of coefficients $a_0, a_1, \ . \ . \ . a_k$ and with $a_k \neq 0$ the degree of the polynomial is k. The exponential function $f(n) = b^n$ with $b > 0$ grows faster than aforementioned functions. The factorial function $f(n) = n$! grows very fast and is the worst run-time until now.

In general, the functions $f(n)$are ascending ordered by their growth rate c $c < log(n) < n < n \cdot log(n) < n^2 < n^3 < \ . \ . \ . < b^n < n!$. The figure 1.15 shows different functions for rate of growth in conjunction with the O-notation (e.g. $O(n)$).

Figure 15: Figure 1.15: Rate of Growth with data input n (Fufaev, 2020)



In algorithm analysis, the run-time of an algorithm grows proportionally to n is often adequate to know (cf. (Goodrich et al., 2014, p. 164)).

Therefore, the asymptotic analysis is the right choice. The number of primitive operations are represented by a constant factor in that analysis.

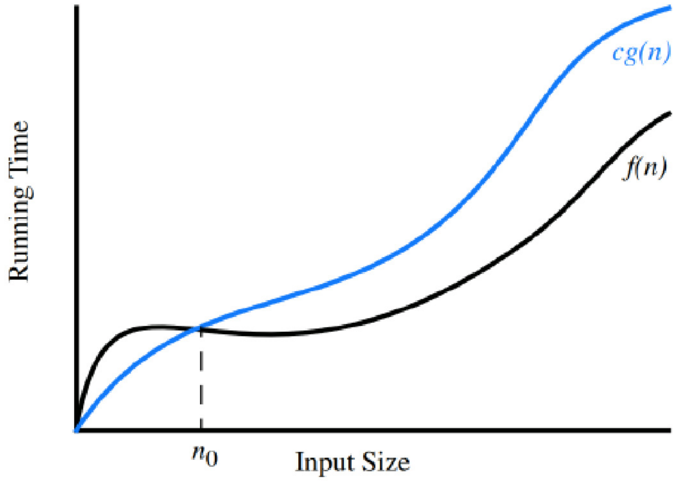
This leads to the O-notation for algorithm analysis that is not dependent to soft- and hardware.

Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We define $f(n) = O(g(n))$if there exists a real constant $c > 0$ and integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), n \geq n_0$$

This definition is depicted in figure 1.16. With growing input size starting at n_0, the function $c \cdot g(n)$ is above $f(n)$ and therefore acts as an upper boundary for $f(n)$. The O-notation defines in the asymptotic sense for $n \mapsto \infty$ that $f(n)$ is less than or equal to another function $g(n)$ times a constant c when $n \geq n_0$. It is from a mathematical perspective incorrect but in computer science it is $f(n) = O(g(n))$. The statement $f(n) \in O(g(n))$ implies that $f(n)$ denotes to a whole collection of functions $g(n)$.

Figure 16: Figure 1.16: Function $f(n)$ is $O(g(n))$ since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$ (Goodrich et al., 2014, p. 164)



Let us take a look at an example: $32n + 2 \in O(n)$? The statement is true.

We can calculate that by using the definition for the O-notation:

$$32n + 2 \overset{?}{\leq} c_1 n + c_2 \text{ with } c_1 \geq 32, c_2 \geq 2$$

Another example is $3^n \in 2^{\backslash O(n)}$?

$$\begin{aligned} &3^n \overset{?}{\leq} 2^{c_1 n + c_2} \\ &e^{ln(3) \cdot n} \overset{?}{\leq} e^{ln(2) \cdot (c_1 n + c_2)} \\ &ln\Big(3\Big) \cdot n \leq ln\Big(2\Big) \cdot \Big(c_1 n + c_2\Big) \text{ with } c_1 \geq \frac{ln(3)}{ln(2)} \end{aligned}$$

Hint: If an algorithm consists of different elements, the “most substantial” element is decisive. For example $2^n + 6n^6 + log(n) \in O(2^n)$ because the most substantial respectively fastest growing function is 2^n (cf. figure 1.15).

The algorithm analysis can be performed by using the O-notation. The algorithm 12 is used to explain the calculation for the time complexity.

```
Algorithm 12 Example Algorithm
for i = 0; i < 100; i ← i + 1 do
    Primitive Operation
end for
for i = 0; i < n; i ← i + 1 do
```

```
5:     Primitive Operation
6: end for
7: for i = 0; i < n; i ← 2i do
8:     Primitive Operation
9: end for
10: for i = 0; i < n; i ← i + 1 do
11:     for j = 0; j < n; j ← j + 1 do
12:         Primitive Operation
13:     end for
14: end for
```

The lines 1 to 3 represent a loop over a primitive operation $(O(1))$ that is executed 100 times. Thus, from line 1 to 3 the time complexity is exactly 100 that means $O(1)$. The loop from lines 4 to 6 is executed n times, consequently $O(n)$. The loop from lines 7 to 9 is executed $log_2(n)$-times because i growth exponential $1, 2, 4, 8, 16, 32, \ldots (2n)$ until the condition $i < n$ is violated. Thus, the loop is $O(\log(n))$ (the base 2 is not important for O). The lines 10 to 14 describe two nested loops. The outer loop is executed n-times $(O(n))$ and the inner loop is also executed n-times $(O(n))$. Thus, the nested loops together are $O(n^2)$. Nested loops have to be multiplied for calculating the time complexity and successive loops have to be added. Thus, the whole algorithm has exactly the time complexity of $100 + n + log_2(n) + n^2$ that results in $O(n^2)$ because of the polynomial function.

Beside the O-notation exists also the Ω- and Θ-notation. As mentioned before, O is referring to the upper boundary for time complexity of an algorithm. The Ω is the lower boundary for time complexity of an algorithm.

This means that $f(n) \in \Omega(g(n))$ when $c \cdot g(n) \leq f(n)$. The Θ-notation can be used if you want to describe the exact boundary. This means that $f(n) \in \Theta(g(n))$ when $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$. In most of the cases, the O-notation is used because it expresses that the time complexity of an algorithm is not worse than denoted.

SUMMARY

In theoretical computer science, the important term algorithm is defined and paradigms like iterative and recursive are described.

Furthermore the intention and efficiency is mentioned. Algorithms process on data structures. Thus, data structures like arrays, linked lists, queues, stacks and heaps are defined.

Furthermore, structures like directed and undirected graphs and special graphs, trees, are described. Therefore, different representation formats like incidence and adjacency matrices and the adjacency list is described and evaluated.

Sorting and searching algorithms for numbers and texts are shown in pseudocode and their efficiency is mentioned to compare and discuss them.

Algorithm analysis introduces two possibilities to analyse the performance in time and space of algorithms. Measuring the elapsed execution time of an algorithm is dependent to soft- and hardware whereas the analysis using e.g. the O-notation is independent and an important for computer science.

UNIT 2

FORMAL LANGUAGES AND AUTOMATA THEORY

STUDY GOALS

On completion of this unit, you will be able to ...

- understand differences between formal languages
- understand the interdependence between formal languages and automata
- know constitutive elements of formal languages
- understand difference between syntax and semantics

2. FORMAL LANGUAGES AND AUTOMATA THEORY

Introduction/Case study

Formal languages in conjunction with automata are the basis for computer systems. A computer system can process data and information based on formal languages. Automata are mathematical representations of computers to i.a. indicate if a formal language expression (e.g. computer program) is syntactically correct or not. Thus, an automaton that is suitable for a formal language has the possibility to decide if a word or "longer text" is part of a formal language or not. An human can easily decide if the addition of two numbers is syntactically correct. 3+4 is syntactically correct whereas 3+cat is not. We will learn how a computer or more precisely formal languages and automata can be used for that.
Outline
Section 2.1 introduces the concept of languages and grammars to generate language expressions. Additionally, the Chomsky hierarchy to classify formal grammars are introduced. In section 2.2 are introduced regular languages and related finite state machines. Section 2.3 describes context-free languages and pushdown automata. Section 2.4 explains context-sensitive languages and turing machines.

2.1 Languages and Grammars

alphabet
An alphabet is a finite set of symbols.

formal language
A formal language consists of words that are build using the alphabet.

In general, formal languages are used to analyse, classify and construct words based on an **alphabet** Σ. The alphabet is the set of symbols that can be used for a word ω. In conclusion, a word is defined as $\omega \in \Sigma*$ with meta-symbol $*$ means 0 to many symbols can be used. The set $\Sigma*$ is called Kleene closure and summarizes all finite sequences of symbols including the empty word ε. Thus, each subset $L \subseteq \Sigma*$ is a **formal language** using alphabet Σ.

The following language expressions are constructed by expanding the example for addition with the alphabet $\Sigma = \{0,1,2,3,4,5,6,7,8,9,+,=\}$.

1. 3+4=7
2. 3+4=10
3. 4+t=7
4. 4+3+3=2
5. 7=3=4

The listed examples (1., 2., 4., 5.) are part of the formal language because the used symbols are included in the alphabet. Thus, only example (3.) is not part of the formal language because symbol t is not included in the alphabet. At this moment, the decision of word ω is part of the formal language L is only relying on the alphabet without further rules. From a human perspective, only example (1.) is correct depending on syntax and semantics but that is only based on experience from school. So let us dive into that in more detail.

The generation of formal language expressions resp. words can be based on structured production rules. These rules are summarized by a grammar G for generating words of a formal language $L(G)$. Let us reuse the addition example with the following production rules[1][2]:

Table 1

equation	$\longrightarrow$	addition = result
addition	$\longrightarrow$	number+ number
result	$\longrightarrow$	number+
number	$\longrightarrow$	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Using this grammar for the example, only examples (1., 2.) are part of the formal language. Examples (4., 5.) are violating the production rules and example (3.) has still a symbol that is not included in the alphabet.

Additionally, it is important that the semantics are excluded so far. Only the syntax can be checked using the production rules and grammar. Thus, **grammar** $G = (V, \Sigma, P, S)$ is defined as

grammar
A grammar consists of variables, terminal symbols, production rules and a start variable.

- V is the finite set of variables/nonterminal symbols,
- Σ is the finite set of the terminal alphabet,
- P is the finite set of the production rules with each production rule is based on $l \longrightarrow r$ with $l \in (V \cup \Sigma)^+$ and $r \in (V \cup \Sigma)$*,
- S is the start variable with $S \in V$.

The meta-symbol + is similar to the * but means 1 to many instead of 0 to many. In reference to the addition example, the grammar G is based on

- $V = \{equation, addition, result, number\}$,
- $\Sigma = \{\,=\,,\,+\,, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
- P: $\begin{array}{ccccc} equation & \longrightarrow & addition & = & result \\ addition & \longrightarrow & number & + & number \\ result & \longrightarrow & & number^{+} & \\ number & \longrightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 & & \end{array}$
- $S = equation.$

[1] Italic text indicates variables/nonterminals whereas “normal” text indicates terminals. The meta-symbol “|” is used for alternatives.

[2] The meta-symbol + means 1 to many.

In conjunction with formal languages, i.a. the following questions arise: Is word $\omega \in \Sigma*$ part of formal language L with $\omega \in L$? Are formal language expressions part of the formal language? For answering these questions, an analytical way for checking the syntax is required. The production rules P enable a structured way (e.g. via syntax tree) to proof if a word is part of the formal language $L(G)$. This is based on the derivation relation ($\Rightarrow$). Additionally, $\Rightarrow*$ is the reflexive-transitive closure of the derivation relation. This means a word ω is part of a formal language $L(G)$ if $L(G) = \{\omega \in \Sigma* \mid S \Rightarrow* \omega\}$. Thus, it has to be possible to derive the word ω from the start variable S.

Let us derive the addition example (3+4=7) with that knowledge, starting with start variable $S = equation$:

Table 2

equation	⇒	addition = result
	⇒	number + number = result
	⇒	3 + number = result
	⇒	3 + 4 = result
	⇒	3 + 4 = number+
	⇒	3 + 4 = 7

It is proven that “3+4=7” is syntactically correct and thus part of formal language $L(G)$ which can be briefly written $equation \Rightarrow* 3 + 4 = 7$. Let us try the derivation with example (4+3+3=2):

Table 3

equation	⇒	addition = result
	⇒	number + number = result
	⇒	4 + number = result
	⇒	4 + 3 = result

There is no possible derivation based on the production rules of grammar G to end in “4+3+3=2”. Thus, the expression is not part of the formal language $L(G)$ and in this context syntactically incorrect. Keep in mind that each derivation can be translated into a syntax tree with the start variable as root, the inner nodes as nonterminals/variables and the terminals as leafs. The syntactical analysis, independent of using syntax trees or the derivation relation, can be done also in other sequencing. As shown in both examples, a left derivation is used but it is also possible to do a right derivation or mixed approach.

The following example describes the example (3+4=7) again but now with right derivation:

Table 4

equation	$\Rightarrow$	addition = result
	$\Rightarrow$	addition = number+
	$\Rightarrow$	addition = 7
	$\Rightarrow$	number + number = 7
	$\Rightarrow$	number + 4 = 7
	$\Rightarrow$	3 + 4 = 7

The derivation from start variable to the word is a proof if the word is part of the formal language. It is also possible to process from bottom (terminals, e.g. 3+4=7) to the top (start variable, e.g. $equation$).

Formal grammars are powerful for generating languages. They can be used to generate languages from “easy” til “complex” ones. The most important factor are the production rules of the grammar. The scientist Noam Chomsky defines therefore four different classes that form the **Chomsky hierarchy**.

Chomsky hierarchy
Chomsky hierarchy categorizes the set of formal languages based on their grammars into four classes.

- Regular grammars (type-3-grammars) are context-free and their right side of the production rules consists of the empty word ε or a terminal followed by a nonterminal. Thus, the production rules are $l \longrightarrow r$ with $l \in V$ and $r \in \{\varepsilon\} \cup \Sigma V$.
- Context-free grammars (type-2-grammars) are characterized by the left side of the production rule that has only one variable. Thus, the production rules are $l \longrightarrow r$ with $l \in V$.
- Context-sensitive grammars (type-1-grammars) are grammars with the property of production rules that $l \longrightarrow r$ with $|r| \geq |l|$. Thus, a production rule can only extend the derived output
- Unrestricted grammars (type-0-grammars) include all formal grammars.
  Each grammar is per definition always also a type-0-grammar.
  Thus, the production rules are characterized by the initial definition $l \longrightarrow r$ with $l \in (V \cup \Sigma)^+$ and $r \in (V \cup \Sigma)*$.

In conclusion, a type-n-grammar (G_n) generates a type-n-language ($L_n(G_n)$).

Thus, between the corresponding languages exist the inclusion relation $L_0 \supset L_1 \supset L_2 \supset L_3$. This means that it exists in each class of the languages a language L so that L is in L_n but not in L_{n+1}.

Some abstract examples for the different classes of L_1, L_2 and L_3 are

- $L_3 = \left\{(ab)^n \mid n \in \mathbb{N}^+\right\}$ is a type-3-language.
- $L_2 = \left\{a^n b^n \mid n \in \mathbb{N}^+\right\}$ is a type-2-language but no type-3-language.
- $L_1 = \left\{a^n b^n c^n \mid n \in \mathbb{N}^+\right\}$ is a type-1-language but no type-2-language.

In the next units, we will go into more detail with these examples and why they are falling into the mentioned language classes.

2.2 Regular Languages and Finite State Machines

The regular languages (type-3-languages) are the “smallest” class referring to the Chomsky hierarchy. Many data formats are regular and also search patterns are based on regular expressions. As mentioned, regular languages based on regular grammars are using production rules that have one nonterminal on the left side and the right side consists of the empty word ε, a terminal symbol or a terminal symbol followed by a nonterminal symbol which is called right-linear language[3]. This is defined by $l \rightarrow r$ with $l \in V$ and $r \in \{\varepsilon\} \cup \Sigma V$.

Let us take a deeper look at the regular language $L_3 = \left\{(ab)^n \mid n \in \mathbb{N}^+\right\}$.

A corresponding regular grammar $G_3 = (\{S, B, C\}, \{a, b\}, P, S)$ has the production rules P

Table 5

S	$\rightarrow$	aB
V	$\rightarrow$	bC
C	$\rightarrow$	ε|aB

syntax tree
A syntax tree illustrates the derivation from the start variable to the word consisting of terminals.

Thus, the regular language $L_3(G_3) = \{ab, abab, ababab, \ldots\}$ is generated by this grammar G_3. The derivation using the production rules results in a **syntax tree** that forms a linear chain if a word of this language is generated (cf. fig. 2.1). This kind of grammar is also called right-linear grammar because it “grows” to the right side by substituting a nonterminal with a terminal or combination of terminal and nonterminal. Let us take a look at the generation of the word “abab” using the grammar G_3.

The corresponding syntax tree for this sequence of derivations with the production rules is depicted in figure 2.1. The left side depicts the syntax

S ⇒ aB ⇒ abC ⇒ abaB ⇒ ababC ⇒ abab

Table 6

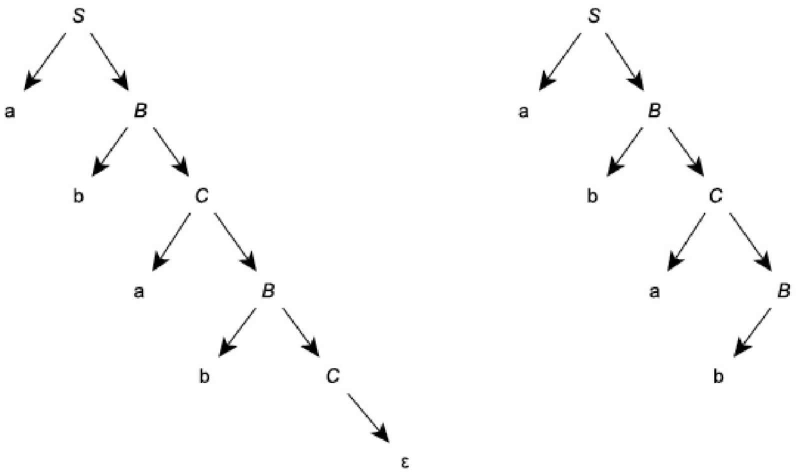
S	$\rightarrow\rightarrow$	aB

[3] Additionally, a nonterminal symbol followed by a terminal symbol is possible and is called a left-linear language.

	$\rightarrow$	ab$C$
	$\rightarrow$	aba$B$
	$\rightarrow$	abab$C$
	$\rightarrow$	abab

tree with the epsilon-rule and the right side without that rule. The epsilonrule $C \rightarrow \varepsilon$ can be eliminated by expanding the production rules with $B \rightarrow$ b$C|$b. The elimination of epsilon-rules is possible by allowing rules with the right side only consists of a terminal. Thus, in our example the syntax tree ends in the generation process without an additional step.

Figure 17: Figure 2.1: Syntax Tree for Generating "abab"



We have based our decision referring to the syntactical correctness of a word ω on "manually elaborate" the production rules. Finite state machines are a way to formally conceive and systematically analyse regular languages. A finite state machine can be used as acceptor. It can process a language expression/word ω resulting in a decision between word is part of a language or not. The set of all possible words that are decided by the automaton A as correct forms the accepted language $L(A)$. In general, finite state machines differ between deterministic (DFA) and non-deterministic finite state machines (NFA). A deterministic finite state machine (DFA)

$A = (S, \Sigma, \delta, E, s_0)$is defined as

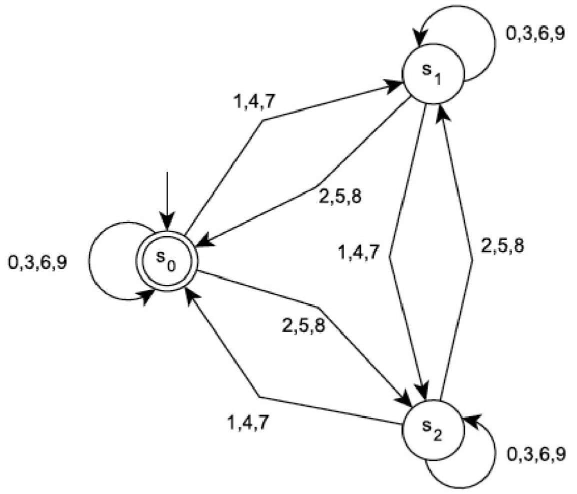
- S is the finite set of states,
- Σ is the finite set of the input alphabet,
- δ is the transition function with $\delta\colon S \times \Sigma \rightarrow S$,
- E is the set of final states with $E \subseteq S$,
- s_0 is the start state with $s_0 \in S$.

Let us dive into a concrete example for a deterministic finite state machine to understand the formal definition and introduce a way to design a DFA as a directed graph. The input is accepted from the DFA if the integer is divisible by 3 without a remainder in the following example.

$$
\begin{aligned}
S &= \left\{ s_0, s_1, s_2 \right\}, \\
\mathit{\Sigma} &= \left\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \right\}, \\
\delta\left(s_0, x \right) &= \begin{cases} s_0 \text{with } x \in \{0, 3, 6, 9\} \\ s_1 \text{with } x \in \{1, 4, 7\} \\ s_2 \text{with } x \in \{2, 5, 8\} \end{cases}, \\
\delta\left(s_1, x \right) &= \begin{cases} s_1 \text{with } x \in \{0, 3, 6, 9\} \\ s_2 \text{with } x \in \{1, 4, 7\} \\ s_0 \text{with } x \in \{2, 5, 8\} \end{cases}, \\
\delta\left(s_2, x \right) &= \begin{cases} s_2 \text{with } x \in \{0, 3, 6, 9 \\ s_0 \text{with } x \in \{1, 4, 7\} \\ s_1 \text{with } x \in \{2, 5, 8\} \end{cases}, \\
E &= \left\{ s_0 \right\}.
\end{aligned}
$$

As mentioned, this DFA can also be drawn as a directed graph depicted in figure 2.2. The start state is marked with an arrow without beginning in a state (cf. node/state s_0). The end state is marked with a double circle (cf. node/state s_0, in the example same as start state). A word ω is accepted by an automaton A if the automaton ends after the processing in the/an end state. This is identical to word ω is part of the language $L(A)$.

Figure 18: Figure 2.2: Deterministic Finite State Machine for the Division by 3 without Remainder



In the following, the more formal as well as the graph-based approach will be addressed. For example, the integer 147 is used as input/word ω for the DFA. The word ω can be structured like $\omega = \lambda_0\lambda_1 \ldots \lambda_n$. Thus, using the definition of DFA, it is

$$L(A) = \{\lambda_0\lambda_1 \ldots \lambda_n \in \Sigma^* \,|\, \delta(\ldots\delta(\delta(s_0, \lambda_0), \lambda_1), \ldots, \lambda_n) \in E\}$$

This means that in each state transition a part of the word is "processed" and leads to the subsequent state. The DFA is **deterministic** so that not more than one path from start state to an intermediate or end state is possible for a given input. Referring the example, the start state is $s_0$and the first part of the word ω is 1 $\lambda_0 = 1$. Thus, the processing results in a transition to state s_1 following the arrow in the graph-based DFA $(s_0, 147) \to_A (s_1, 47)$. The same can be described by the formal approach $s_1 = \delta(s_0, 1)$. In state s_1, the partial input $\lambda_1 = 4$ is processed and so a transition to state s_2 is made, $(s_1, 47) \to_A (s_2, 7)$. The remaining $\lambda_2 = 7$ is processed by the transition to s_0 following the arrow $(s_2, 7) \to_A s_0$. Thus, 147 is divisible by 3 without a remainder (ending in end state s_0). This means also that the word $\omega = 147$ is part of the formal language $L(A)$.

deterministic
Deterministic means that each subsequent state is uniquely determined by the read input character.

Summarized, an accepted formal language $L(A)$ is characterized by

$$L(A) = \{\omega \in \Sigma^* | \text{ for a } s_e \in E{:}\, (s_0, \omega) \to_A^* s_e\}$$

In contrast to the deterministic finite state machine (DFA), the non-deterministic finite state machine (NFA) $A = (S, \Sigma, \delta, E, s_0)$ is defined as

- S is the finite set of states,
- Σ is the finite set of the input alphabet,
- δ is the transition function with $\delta{:}\, S \times \Sigma \to S$,

- E is the set of final states with $E \subseteq S$,
- s_0 is the start state with $s_0 \in S$.

The definitions for DFA and NFA differ in the transition function δ. The $\delta: S \times \Sigma \longrightarrow S$ in DFA maps to one element of the set of states S whereas $\delta: S \times \Sigma \longrightarrow 2^S$ in NFA maps to the power set 2^S. This means in the NFA is more than one subsequent state possible. Keep in mind that with that power set also $\delta(s, \lambda) = \emptyset$ is feasible. Thus, the NFA stops with processing independent if more partial input is available. The word ωwill not be accepted by the automaton independently if it is currently in an end state or not.

The following example describes a NFA that accepts binary input having a 0 at the third from last. Figure 2.3 depicts a NFA accepting such an input.

Same as the DFA, the NFA processes character by character starting in the start state s_0. Let us take a look at the possible sequences of processing input $\omega = 000$. There are four paths[4] possible

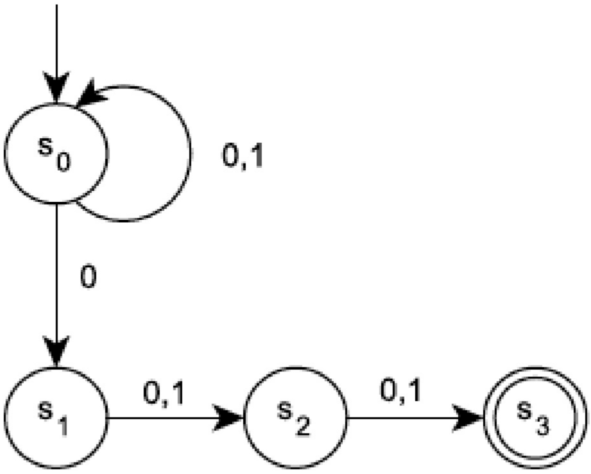
1. $s_0 \longrightarrow s_0 \longrightarrow s_0 \longrightarrow s_0$
2. $s_0 \longrightarrow s_0 \longrightarrow s_0 \longrightarrow s_1$
3. $s_0 \longrightarrow s_0 \longrightarrow s_1 \longrightarrow s_2$
4. $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3$

A word ω will be accepted by the automaton if at least one sequence ends in a final state. It seems that the automaton can guess such a successful sequence by choosing a path at each decision point to end in a final state.

In the example, the fourth path ends in a final state (s_3) so that the word is accepted.

[4] The meta-symbol →A is here interchangeable for simplification with →.

Figure 19: Figure 2.3: Non-deterministic Finite State Machine Having a 0 at Third from Last



Some languages can be much easier described by a NFA instead of a DFA.

An example is depicted in the DFA and NFA automaton for having a 0 at third from last, cf. figure 2.5.

The question is if the set of accepted languages of NFAs is greater than DFAs or vice versa. Rabin and Scott had specified that for each NFA exist a DFA (and the other way around) that accepts the same formal language. Thus, NFA and DFA are equally powerful in sense of accepting the same formal language. This is called the clause of Rabin and Scott with L(DFA) = L(NFA). Additionally, each DFA can be transformed to a NFA and vice versa.

Let us focus on the formal grammar, the translation to the corresponding DFA as well as transforming from NFA to DFA. For the grammar to DFA translation, we use the abstract example $L_3 = \{(ab)^n \mid n \in \mathbb{N}^+\}$ which is a type-3-language. The corresponding grammar $G_3 = (\{S, B, C\}, \{a, b\}, P, S)$ has the production rules P

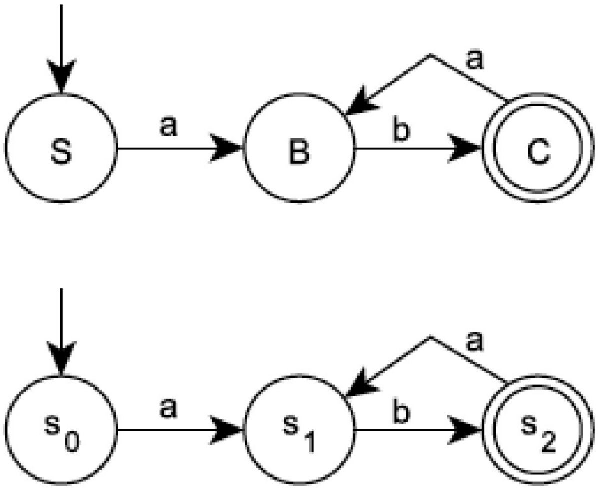
The result of the translation of this grammar to a DFA can be depicted in figure 2.4. The translation from a formal grammar follows the systematic steps

Table 7

S	$\rightarrow$	aB
B	$\rightarrow$	bC
C	$\rightarrow$	ε|aB

1. the states of the automaton are the nonterminals from the grammar.
2. the input alphabet of the automaton is identical to terminal alphabet of the grammar.
3. the start state of the automaton equals the start variable of the grammar.
4. the final states of the automaton are equal to the production rules with ε.
5. for each production rule like $B \longrightarrow bC$ (nonterminal $\longrightarrow$ terminal followed by nonterminal), an arrow starting in left-side nonterminal (here B) to nonterminal at the right-side (here C) with the terminal (here b) will be drawn.

Figure 20: Figure 2.4: Translation from Grammar to Deterministic Finite State Machine for $L_3 = \{(ab)^n \mid n \in \mathbb{N}^+\}$



Both presented automata are equivalent, cf. figure 2.4. Just for clarification reasons, the automaton directly following the steps from grammar to DFA is translated in a form that is more familiar. This refers to the substitution of nonterminals from the grammar to more popular states of the automaton. Hint: The other format (letting nonterminals as states)

is of course also valid. The vice versa translation from automaton to the grammar is analog to the presented way.

After the translation of a grammar of a regular language, we learn about the transformation of a NFA to DFA. Therefore, we use the aforementioned example having a 0 at third from last, cf. fig. 2.3. The result of the transformation of this NFA to a DFA can be depicted in figure 2.5. The transformation follows the (shortened) systematic steps

1. the input alphabet of the NFA is identical to DFA.
2. start with the set of start state(s) of the NFA as start state of the DFA.
3. follow each outgoing arrows with assigned input terminal(s) and cluster the possible particular subsets as new states in DFA. Hint: the overall number of states in the DFA can be 2S of the NFA states in maximum, cf. definition of NFA.
4. mark each clustered subset state in DFA including a final state of the NFA as final state of the DFA.

Let us develop the transformation of the NFA (cf. fig. 2.5 left automaton) to the DFA (cf. fig. 2.5 right automaton) step by step.

The start state of the NFA s_0 is also start state of the DFA s_0. This start state (NFA) has for input terminal 0 arrows ending in s_0 and s1. Thus, the resulting subset consists of s_0, s1 as new state in the DFA for input 0.

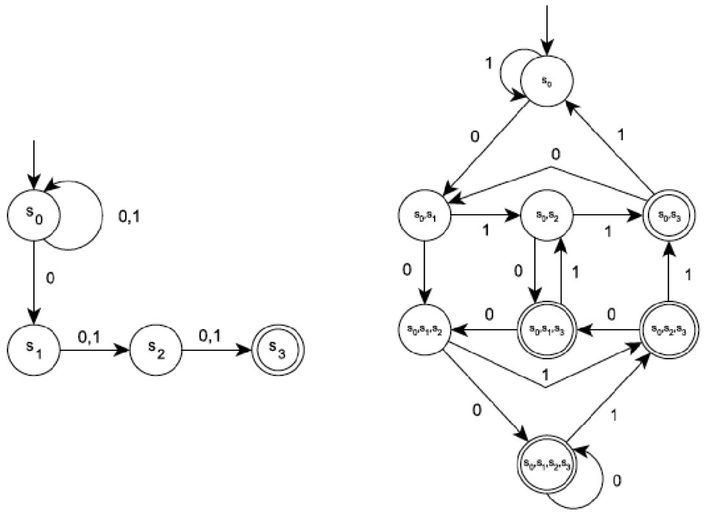
Secondly, we take care of input terminal 1 for the pinned start state of the NFA. The arrow ends ins_0. Thus, the resulting subset consists ofs_0. This state is already in the DFA so that we draw a self-reference to s_0 for input 1.

In the next step, state s_0, s_1 (DFA) is considered. Firstly, state $s_0$of the NFA is pinned. For input 0, the arrows end in s_0 and s_1. Secondly, state s_1 of NFA is considered for input 0. The arrow ends in s_2. Thus, the resulting subset for input 0 consists of s_0,s_1, s_2 as new state in the DFA. Now, the analog consideration has to be done for input 1. Firstly, state s_0 of the NFA is considered for input 1, the arrow ends in s_0. Secondly, state s_1 of NFA is considered for input 1. The arrow ends in s_2. Thus, the resulting subset for input 1 consists of s_0, s_2 as new state in the DFA.

This procedure (checking for input 0 and 1) has to be conducted for each (new) states in the DFA analogously.

Finally, all states in the DFA which have included final states of the NFA (here s_3) are marked as final states of the DFA.

Figure 21: Figure 2.5: Non-deterministic and Deterministic Finite State Machine for Having a 0 at Third from Last



In summary, NFA can be less complex in their representation and easier to design compared to a DFA. DFA is more efficient than NFA because the path for an input is unique. NFA has to choose a possible path because of more than one possible alternative. Additionally, each terminal symbol can be forgotten after processing using a DFA. In a DFA, the automaton immediately results in “syntactically incorrect” if an input terminal can not be processed - no outgoing arrow with appropriate terminal in current state. NFA has to check each possible path to do so.

2.3 Context-free Languages and Pushdown Automata

In comparison to regular languages, context-free languages (type-2-languages) are an extension. The production rules of a context-free grammar are defined as $l \longrightarrow r$ with $l \in V$ and $r \in (\Sigma \cup V)*$. For the left side one nonterminal is allowed, the same as in grammars for regular languages. The right side consists of an arbitrary sequence of terminal and nonterminal symbols.

This is in contrast to regular grammars that are more restrictive for the right side (cf. $r \in \{\varepsilon\} \cup \Sigma V$).

Let us take a deeper look at the context-free language $L_2 = \left\{a^n b^n \mid n \in \mathbb{N}^+\right\}$. A corresponding context-free grammar $G_2 = (\{S\}, \{a, b\}, P, S)$ has the production rules P

Table 8

S	$\longrightarrow$	aSb|ab

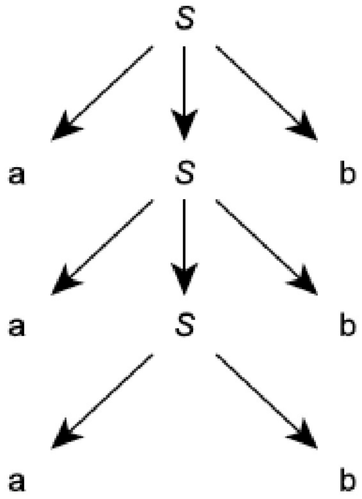
Thus, the context-free language $L_2(G_2) = \{ab, aabb, aaabbb, \ldots\}$ is generated by this grammar G_2. The derivation using the production rules results in a syntax tree with a “pumped” middle part if a word of this language is generated (cf. fig. 2.6). For a deep dive into the formal proof if a language is regular or context-free the Pumping-Lemma can be used. For the course, the Pumping-Lemma is out of scope. Let us take a look at the generation of the word “aaabbb” using this grammar G_2.

Table 9

S	$\Rightarrow$	aSb
	$\Rightarrow$	aa$S$bb
	$\Rightarrow$	aabbb

Firstly, the nonterminal start symbol $S$is derived to aSb with two terminals and a nonterminal in between. In the next step, this S in the middle is derived to aa$S$bb by substituting the S again with aSb. The last derivation substitutes the S with ab ending in the word aaabbb. Thus, the word is part of the context-free language $L_2(G_2)$.

Figure 22: Figure 2.6: Syntax Tree for Generating "aaabbb"



The production rules of context-free grammars can be simplified by using the **Chomsky Normal Form (CNF)** . The "normal" context-free grammars and CNF are equivalent. A grammar $G = (V, \Sigma, P, S)$ is in CNF if all production rules are like $S \rightarrow \varepsilon, A \rightarrow \sigma$ or $S \rightarrow \varepsilon, A \rightarrow \sigma$ with $A \in V, \quad B, C \in V \setminus \{S\}$ and $\sigma \in \Sigma$. Thus, the transition of a context-free grammar to CNF can be done by the following steps:

Chomsky Normal Form (CNF)
The Chomsky Normal Form consists of production rules with terminals or two nonterminals on the right side.

1. Elimination of ε-rules: All production rules $A \rightarrow \varepsilon$are eliminated by adapting all rules that are involved in a derivation to ε. Thus, the empty word will be removed from the language $L(G)$: $\varepsilon \notin L(G)$.
2. Elimination of chain rules: Each production rule $A \rightarrow B$ with $A, B \in V$ is a chain rule and does not contribute to the production of terminals. Thus, these rules are analogously eliminated similar to step 1 by substituting the right side.
3. Separation of terminals: Each terminal σ that occurs in combination with nonterminals is substituted by a new nonterminal $V\sigma$ and a rule is added as $V_\sigma \rightarrow \sigma$ to the set of production rules.
4. Elimination of multi-elemental nonterminal chains: All production rules like $A \rightarrow B_1 B_2 \ldots B_n$ are changed to $A \rightarrow A_{n-1} B_n, A_{n-1} \rightarrow A_{n-2} B_{n-1}, \ldots, A_2 \rightarrow B_1 B_2$. After this substitution, all multi-elemental chains are changed and the CNF is constructed.

The described steps are applied to the context-free grammar $G = (\{S, A, B\},$, $\{a, b\}, P, S)$ generating $\left\{a^i b^j a^k \mid n \in \mathbb{N}^+, j, k \in \mathbb{N}\right\}$ with P

Table 10

S	$\rightarrow$	$AB \mid ABA$
A	$\rightarrow$	aA |a

B	$\longrightarrow$	Bb$\vert\varepsilon$

The sequence for the transformation of this context-free grammar to the CNF is depicted in table 2.1.

Table 4: Table 2.1: Successive Generation of the Chomsky Normal Form

Start	Step 1	Step 2	Step 3	Step 4
$S \longrightarrow AB$	$S \longrightarrow AB$	$S \longrightarrow AB$	$S \longrightarrow AB$	$S \longrightarrow AB$
$S \longrightarrow ABA$	$S \longrightarrow A$	$S \longrightarrow$aA	$S \longrightarrow V_aA$	$S \longrightarrow V_aA$
$A \longrightarrow$aA	$S \longrightarrow ABA$	$S \longrightarrow$a	$S \longrightarrow$a	$S \longrightarrow$a
$A \longrightarrow$a	$S \longrightarrow AA$	$S \longrightarrow ABA$	$S \longrightarrow ABA$	$S \longrightarrow S_2A$
$B \longrightarrow B$b	$A \longrightarrow$aA	$S \longrightarrow AA$	$S \longrightarrow AA$	$S_2 \longrightarrow AB$
$B \longrightarrow \varepsilon$	$A \longrightarrow$a	$A \longrightarrow$aA	$A \longrightarrow V_aA$	$S \longrightarrow AA$
	$B \longrightarrow B$b	$A \longrightarrow$a	$A \longrightarrow$a	$A \longrightarrow V_aA$
	$B \longrightarrow$b	$B \longrightarrow B$b	$B \longrightarrow BV_b$	$A \longrightarrow$a
		$B \longrightarrow$b	$B \longrightarrow$b	$B \longrightarrow BV_b$
			$V_a \longrightarrow$a	$B \longrightarrow$b
			$V_b \longrightarrow$b	$V_a \longrightarrow$a
				$V_b \longrightarrow$b

In step 1, the possible production rules ending in $\varepsilon$are considered. Obviously, $B \longrightarrow \varepsilon$ is such a rule. Next production rules like $A \longrightarrow * \varepsilon$, starting in an arbitrary nonterminal and ending in ε are considered[5]. Thus, the nonterminals for ending in ε are only B. From the rules $A \longrightarrow$ aA |a, it exists no path to ε because for each specific rule immediately a terminal is produced (here a). The rules $S \longrightarrow AB \mid ABA$ have involved Awhich has no possibility to produce ε. Thus, these rules can also be skipped. In summary, the rule $B \longrightarrow$b is added because the B can be derived to ε in rule$B \longrightarrow B$b. Analogously, $S \longrightarrow A$ and $S \longrightarrow AA$ are added.

In step 2, all production rules from step 1 are considered consisting of one nonterminal on the left as well as right side. The rules $S \longrightarrow$aA and $S \longrightarrow$a substitute the chain rule $S \longrightarrow A$. No more such chain rules are present so step 2 is finished.

[5] →∗ ε means that it is possible by using production rules to end in ε.

In step 3, the considered rules with a nonterminal and terminal from step 2 are $S \rightarrow$aA, $A \rightarrow$aA and $B \rightarrow B$b. The aA is substituted with $V_a A$ and Bb with BV_b. Additionally, the rules $V_a \rightarrow$a and $V_b \rightarrow$b are added.

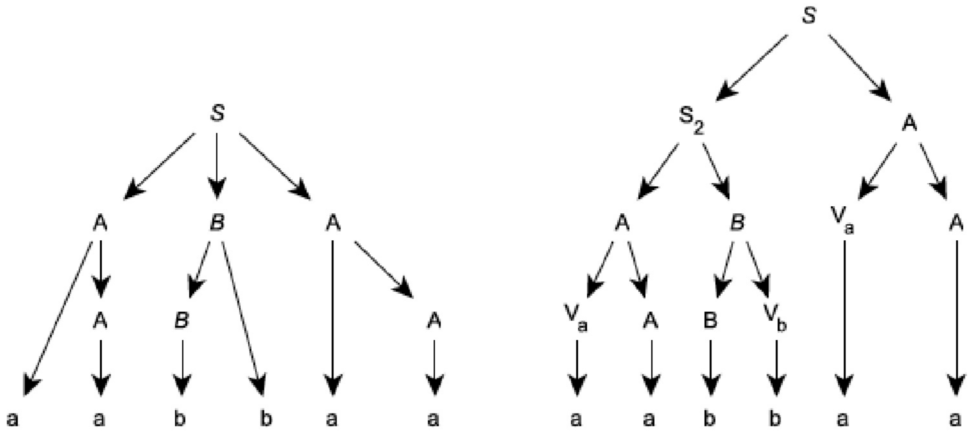
In step 4, rules from step 3 with multi-elemental nonterminals (more than two nonterminals) are considered. The rule $S \rightarrow ABA$ is substituted with $S \rightarrow S_2 A$, and $S_2 \rightarrow AB$ is added to the production rules. Thus, the Chomsky Normal Form is constructed after step 4.

Figure 2.7 depicts on the left side a syntax tree for the above-mentioned grammar (not in CNF). The syntax tree consists of three arrows/successors starting in S for the production rule $S \rightarrow ABA$. The syntax tree can generally have arbitrary successors starting from a nonterminal.

In contrast, the grammar in CNF depicts a syntax tree as **binary tree** on the right side of this figure. Each grammar in CNF results in such a binary tree. Considering the production rules, each nonterminal (left side of the rule) is substituted with a terminal or two nonterminals. Thus, there is a connection between the depth of the resulting binary syntax tree and the amount of leaves. The amount of leaves is 2^h with h being the depth of the tree if the binary tree is complete. Thus, in general, a binary tree has 2^h leaves at maximum.

binary tree
A binary tree is a tree that has at most two successors.

Figure 23: Figure 2.7: Syntax Tree of Context-free Grammar in Chomsky Normal Form for "aabbaa"



The CNF allows a statement about the required derivation steps from the start variable to a given word. The production rules in a CNF are defined as either $A \rightarrow$ a or $A \rightarrow AB$. Thus, for the word "aabbba" 11 steps are required. For simplification, $S \rightarrow AABBBA$ requires 5 steps (e.g. $S \rightarrow AA \rightarrow AAB \rightarrow AABB \rightarrow AABBB \rightarrow AABBBA$). Afterwards, each nonterminal is directly derived to a terminal (6 steps) using rules like $A \rightarrow$ a.

Most programming languages can be described using a context-free language.

The formal specification of the syntax can be defined with Backus- Naur Form (BNF). The syntax of the BNF and already known production rules differ in using the derivation meta-symbol $::=$ instead of $\rightarrow$. The extended BNF includes angular and curved brackets for fragments like $A::=a_1[a_2]a_3$ and $A::=a_1\{a_2\}a_3$. The angular brackets means that a_2 is optional whereas the curved brackets means that a_2 can be arbitrarily repeated. The BNF is another form for describing context-free languages but not have less or greater expressiveness.

Finite state machines are a way to formally conceive and systematically analyse regular languages. For context-free languages, the pushdown automaton (PDA) is the used approach. It is an extension of finite state machines, with the addition of a stack to allow for more complex pattern matching. PDAs are a type of automaton that use a stack to recognize context-free languages. They work by reading symbols from an input string and pushing and popping symbols onto a stack to keep track of the context of the input. PDAs can recognize context-free languages by using the stack to keep track of nested structures, such as matching parentheses or nested loops.

The connection between context-free languages and PDAs is significant because many programming languages can be described by context-free grammars. As a result, PDAs are a fundamental tool for understanding and analyzing the behavior of many different types of languages and systems.

A pushdown automaton (PDA) $A=(S,\Sigma,\Gamma,\delta,s_0)$ is defined as

- S is the finite set of states,
- Σ is the finite set of the input alphabet with $\varepsilon\notin\Sigma$,
- $\Gamma$is the finite set of the stack alphabet with $\perp$ as initial stack symbol,
- δ is the transition function with $\delta : S\times\left(\Sigma\cup\left\{\varepsilon\right\}\right)\times\Gamma\rightarrow 2^{S\times\Gamma^*}$ with $|\delta(s,\omega,\gamma)|<\infty$ for all s,ω,γ,
- $s_0$is the start state with $s_0\in S$.

The transition function takes as input the current state, the current input symbol, and the top symbol of the stack, and outputs the next state, the symbol to be pushed onto the stack, and whether to pop a symbol off the stack. The PDA accepts the input string if it ends up in an accepting state when it has processed the entire input string.

Let us take a look at the context-free language $L_2=\left\{a^nb^n\mid n\in\mathbb{N}^+\right\}$ using a PDA. The production rules P of the context-free grammar are $S\rightarrow$ aSb | ab. Here is an example of how a PDA would process these production rules for the concrete word “aabb”.

First, we need to define the PDA based on the grammar: $S=\{s_0,s_1\},\Sigma=\{a,b\},\Gamma=\{A,\perp\}$ and the transition function δ is defined as follows:

$\delta(s_0,a,\perp)=_1\{(s_0,A\perp)\}$,

$\delta(s_0,a,A)=_2\{(s_0,AA)\}$,

$\delta(s_0, b, A) =_3 \{(s_1, \varepsilon)\}$

$\delta(s_1, b, A) =_4 \{(s_1, \varepsilon)\},$

$\delta(s_1, \varepsilon, \perp) =_5 \{(s_1, \varepsilon)\}.$

Now, let’s see how the PDA processes the input string "aabb":

1. The PDA starts in state s_0 with the initial stack symbol $\perp$ on the stack.
2. The PDA reads the first symbol a from the input string, has $\perp$ on the stack and pushes Aonto the stack and remains in s_0. The stack now contains $A \perp$ (transition $=_1$).
3. The PDA reads the second symbol a from the input string, has $A \perp$ on the stack and pushes A onto the stack and remains in s_0. The stack now contains $AA \perp$ (transition $=_2$).
4. The PDA reads the third symbol b from the input string and pops A from the stack, transitioning to state s_1 with A remaining on the stack (transition $=_3$). The stack now contains $A \perp$.
5. The PDA reads the fourth symbol b from the input string and pops A from the stack, remaining in state s_1 with $\perp$ on the stack (transition $= 4$).
6. The PDA reads ε from the input string and pops $\perp$ from the stack, remaining in state s_1 with ε on the stack (transition $=_5$).

Since the PDA has reached the end and the stack contains only the empty symbol ε, the input string “aabb” is accepted by the PDA and recognized as a valid sentence in the context-free language generated by the grammar.

2.4 Context-sensitive Languages and Turing Machines

Context-sensitive grammars are an extension to context-free grammars. In contrast to context-free grammars, the left side of the production rule can consist of an arbitrary combination of terminals and nonterminals. Thus, it is possible to ensure the term “context” is given. The substitution of a nonterminal can be bound to nature of its surroundings. The contextsensitive grammars have one restriction concerning the length of the left and right side of the production rules. It has to be ensured $|l| \leq |r|$ for each production rule $l \rightarrow r$. Thus, each derivation step can not shorten the resulting derivation.

The context-sensitive language (type-1-language) $L_1 = \left\{a^n b^n c^n \mid n \in \mathbb{N}^+\right\}$ can be generated by such a grammar. The context-sensitive grammar of that language can be processed by the following steps:

1. The nonterminals A, B and C are introduced beside the start variable S. These nonterminals are representatives for the terminals a,b and c. Additionally, the production rules require a rule to arbitrarily generate a,b and c with the mentioned nonterminals A, B and C. It has to be ensured that a,b and c are generated in equal number (cf. production rules $S \longrightarrow abc$ and $S \longrightarrow SABC$).
2. The production rules generate the required number of A, B and C but they are unordered. The right order is reached by adding the next six production rules.
3. Finally, nonterminals have to be substituted with terminals. This substitution can only be conducted if the nonterminals are in the right order. In context-sensitive grammars, terminals can occur also at the left side so that this is ensured. These production rules are the remaining three of the grammar.

Thus, the context-sensitive language $L_1(G_1) = \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$ is generated by the corresponding grammar $G_1 = (\{S, A, B, C\}, \{a, b, c\}, P, S)$ with the production rules P

S → SABC S → abc CA → AC CB → BC BA → AB cA → Ac cB → Bc bA → Ab aA → aa bB → bb cC → cc

Table 11

S	$\longrightarrow$	$SABC$
S	$\longrightarrow$	abc
CA	$\longrightarrow$	AC
CB	$\longrightarrow$	BC
BA	$\longrightarrow$	AB
cA	$\longrightarrow$	Ac
cB	$\longrightarrow$	Bc
bA	$\longrightarrow$	Ab
aA	$\longrightarrow$	aa
bB	$\longrightarrow$	bb
cC	$\longrightarrow$	cc

Let us take a look at the generation of the word “aaabbbccc” using the grammar G_1.

Table 12

S	$\longrightarrow$	$SABC$	$\longrightarrow$	ab$AABB$cCC
	$\longrightarrow$	$SABCABC$	$\longrightarrow$	aAbABBcCC
	$\longrightarrow$	abc$ABCABC$	$\longrightarrow$	aAAbBBcCC
	$\longrightarrow$	abc$ABACBC$	$\longrightarrow$	aa$A$bBBcCC

	$\rightarrow$	abc$AABCBC$	$\rightarrow$	aaabBBcCC
	$\rightarrow$	abcAABBCC	$\rightarrow$	aaabb$B$cCC
	$\rightarrow$	ab$AABBCC$	$\rightarrow$	aaabbbcCC
	$\rightarrow$	abAAc$BBCC$	$\rightarrow$	aaabbbcc$C$
	$\rightarrow$	abAABcBCC	$\rightarrow$	aaabbbccc

The Turing machine is a way to formally conceive and systematically analyse context-sensitive languages. A Turing machine is a mathematical abstraction of a computer. The Turing machine consists of

- Tape: A linear, one-dimensional storage medium consisting of an infinite sequence of cells. Each cell can contain a symbol from a finite alphabet, which may include blank symbols.
- Read/Write Head: A mechanism that can read the symbol on the current cell of the tape, write a new symbol to the cell, or move the tape one cell to the left or right.
- Control Unit: A finite state machine that determines what action the read/write head should take based on the current state and the symbol on the tape. The control unit has access to a set of states, a transition function that maps the current state and symbol to a new state and action, and a set of accepting states.
- States: A finite set of states that the Turing machine can be in at any given time. The control unit starts in an initial state and transitions between states according to the transition function.
- Transition Function: A function that maps the current state and symbol to a new state, a new symbol to be written to the tape, and a direction for the read/write head to move. The transition function is used by the control unit to determine the next action of the Turing machine.
- Accepting States: A subset of the states that are designated as accepting states. If the Turing machine enters an accepting state, it halts and accepts the input string. If it enters a non-accepting state, it halts and rejects the input string.
- Input: An input string that is written on the tape before the computation begins. The input is read by the Turing machine as it moves the read/write head along the tape.

The tape is infinite in both directions, but only a finite portion of the tape is ever used during the computation. The read/write head always starts at the leftmost cell of the tape, and the control unit starts in an initial state.

At each step of the computation, the control unit reads the symbol on the current cell of the tape and decides what action to take next based on the current state and the symbol. The action can be one of three types: write a new symbol to the current cell, move the read/write head one cell to the left or right, or transition to a new state.

The set of states and the transition function determine the behavior of the Turing machine. A Turing machine is said to accept a string if it halts in an accepting state when the input string is written on the tape. Otherwise, it rejects the string.

One of the most important properties of Turing machines is their universality.

That is, any algorithm that can be implemented on a computer can be implemented on a Turing machine. This property makes Turing machines a fundamental concept in theoretical computer science and the basis for the study of computability and complexity theory.

For completeness, the remaining type-0-language $L_0(G_0)$ consists of an unrestricted grammar $G_0 = (V, \Sigma, P, S)$ with the production rules P. P is the finite set of the production rules that are based on $l \rightarrow r$ with $l \in (V \cup \Sigma)^+$ and $r \in (V \cup \Sigma)*$. This means that the left and right side of production rules can consist of an arbitrary sequence of terminals and nonterminals.

Unrestricted languages, also known as recursive enumerable languages, are a class of formal languages that can be recognized by a type of automaton known as a Turing machine. One of the defining properties of Turing machines is their ability to simulate any other computing device, which allows them to recognize a wide range of languages, including unrestricted languages.

SUMMARY

Formal languages and automata theory deals with the study of formal languages, which are sets of sequences of symbols that have some predetermined properties, and automata, which are abstract machines that process these sequences of symbols. This field has applications in many areas of computer science, such as compiler design, programming languages, natural language processing, and artificial intelligence.

Formal languages are studied through the use of formal grammars, which are rules that describe the structure of the language. These grammars are classified into different types, such as regular, contextfree, context-sensitive, and unrestricted/recursively enumerable.

Each type of grammar corresponds to a class of languages with different properties and complexities.

In conjunction, automata theory deals with the study of abstract machines, which can be used to recognize and manipulate sequences of symbols according to a set of rules. These machines include deterministic and non-deterministic finite state machines, pushdown automata, and Turing machines, each with different computational capabilities and complexity classes.

The combination of formal languages and automata theory provides a foundation for the study of the fundamental properties and limitations of computation. The field is also important in the development of algorithms and programming languages, and in the design and analysis of computer systems.

UNIT 3

COMPUTABILITY, DECIDABILITY AND COMPLEXITY

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the different complexity classes
- know and understand decidable and non-decidable problems
- know different decision problems
- understand quantum mechanics and the impact for IT-Security

3. COMPUTABILITY, DECIDABILITY AND COMPLEXITY

Introduction/Case study

Computability, decidability and decision problems, complexity theory, and quantum computing are all important topics in theoretical computer science that deal with the limits of computation and the complexity of solving computational problems.

Computability is concerned with understanding what can and cannot be computed, and how efficiently it can be computed. It explores the concept of algorithmic solvability and the limits of computation.

Decidability and decision problems are related to the concept of computability, but focus on whether or not a given problem can be solved algorithmically. Decision problems ask if a problem has a yes or no answer, and if it can be computed algorithmically.

Complexity theory studies the efficiency of algorithms and the amount of resources, such as time and memory, required to solve computational problems. It classifies problems based on their computational complexity and studies the trade-offs between time and space resources.

Quantum computing is a relatively new area of research that studies the use of quantum mechanics to perform computations. It has the potential to revolutionize computing by enabling the solution of problems that are intractable for classical computers.

Outline
Section 3.1 defines computability, Halting problem and unsolvable problems and the Church-Turing thesis. In section 3.2 are decision problems and their computational complexity as well as applications of decision problems in computer science. Section 3.3 describes the big-O notation, complexity classes, including P, NP, and NP-complete and the relationship between complexity classes. Section 3.4 explains quantum qubits, quantum algorithms, such as Shor's algorithm for factoring large integers and Grover's algorithm for searching an unsorted database as well as quantum complexity classes, such as BQP.

3.1 Computability

Computability is a concept in theoretical computer science that refers to the ability to solve a problem algorithmically using a computational machine.

A problem is said to be computable if there exists an algorithm that can solve it in a finite number of steps, given a set of input data.

The concept of **computability** was first formalized by the mathematician Alan Turing in the 1930s. He introduced the notion of a Turing machine (cf. section 2.4), which is a theoretical model of a computing device that can manipulate symbols on an infinitely long tape according to a set of rules.

computability
Computability is the ability of a problem to be solved by an algorithm within a finite amount of time.

Turing showed that any problem that can be solved algorithmically can be solved by a Turing machine, and conversely, any problem that cannot be solved by a Turing machine is not computable.

In addition to Turing machines, there are other formal models of computation, such as lambda calculus and recursive functions. All of these models are equivalent in terms of their computational power, meaning that any problem that can be solved by one model can be solved by another. These two formal models will not be part of that course.

The concept of computability has important implications for the limits of computation. It is known that there exist problems that are not computable, such as the halting problem, which asks whether a given program will halt when run on a specific input. Turing's proof of the undecidability of the halting problem showed that there are problems that are beyond the reach of any algorithmic solution.

Computability theory is a fundamental area of study in computer science, and has applications in many areas, such as algorithm design, artificial intelligence, and cryptography. The concept of computability also plays a central role in the development of programming languages, compilers, and other software tools.

Halting problem and similar problems

The **halting problem** is a famous example of an undecidable problem in computer science. The problem asks whether a given computer program will eventually halt (i.e. stop running) when executed on a particular input. Despite its simple formulation, the halting problem is provably undecidable, meaning that there is no known algorithmic solution that can always answer the question correctly for all possible inputs.

halting problem
The halting problem is the problem of determining whether a given program will terminate or run forever.

The proof of the undecidability of the halting problem was first given by Alan Turing in 1936. He showed that there exists no algorithm that can determine, given a program and its input, whether the program will halt or run indefinitely. This result has important implications for the limits of computation, as it shows that there are problems that cannot be solved algorithmically.

The halting problem is just one example of an undecidable problem in computer science. Other examples include the problem of determining whether a given Diophantine equation (a polynomial equation with integer coefficients) has a solution or the problem of finding a Hamiltonian cycle in a graph (a cycle that visits every vertex exactly once). These problems are undecidable in the sense that there is no algorithm that can always provide a correct solution for all possible inputs.

Despite being unsolvable in the general case, some specific instances of these problems can be solved using heuristics or approximation algorithms.

For example, the Traveling Salesman Problem (TSP), which asks for the shortest possible route that visits a given set of cities exactly once, is NPhard (will be introduced in section 3.3) and therefore is unlikely to have a general algorithmic solution. However, there are many approximation algorithms that can provide near-optimal solutions in practice.

Undecidable problems are a fundamental concept in theoretical computer science, and have important implications for the development of algorithms and software systems. By understanding the limits of computation, researchers can design more efficient algorithms and develop new approaches to solving difficult problems.

The Church-Turing thesis

The Church-Turing thesis is a central concept that relates to the idea of computability. The thesis states that any problem that can be solved algorithmically can be solved by a Turing machine.

The thesis is named after the mathematician Alonzo Church and the computer scientist Alan Turing, both of whom independently developed formal models of computation in the 1930s.

DEFINITION: CHURCH-TURING THESIS

The Church-Turing thesis states that any problem that can be solved algorithmically can be solved by a Turing machine. (Church, 1936)

The Church-Turing thesis is not a formal mathematical theorem, but rather a conjecture based on a set of observations and empirical evidence. The thesis is supported by the fact that all known models of computation, such as Turing machines, lambda calculus, and recursive functions, are equivalent in terms of their computational power. This means that any problem that can be solved by one model can be solved by another.

The Church-Turing thesis has important implications for the limits of computation.

It implies that there are problems that are beyond the reach of any algorithmic solution, such as the halting problem. It also suggests that there are problems that are computable, but not efficiently solvable, such as certain problems in the complexity class NP (cf. section 3.3).

While the Church-Turing thesis has not been proven rigorously, it is widely accepted as a fundamental principle in computer science. It serves as a guiding principle for the development of algorithms and computational models, and has helped shape the field of theoretical computer science.

One example of a computable function is the factorial function, which takes a non-negative integer n as input and returns the product of all positive integers up to and including n. This function can be computed using a simple iterative algorithm, where we multiply each integer from 1 to n together.

Another example of a computable function is the greatest common divisor (GCD) function, which takes two positive integers as input and returns their largest common divisor. This function can be computed using the Euclidean algorithm, which repeatedly subtracts the smaller number from the larger number until one of them becomes zero, at which point the other number is the GCD.

On the other hand, an example of a non-computable function is the halting function, which takes a computer program and its input as input, and returns 1 if the program halts on the input, and 0 otherwise. As mentioned earlier, the halting problem is provably undecidable, meaning that there is no algorithm or Turing machine that can compute this function for all possible inputs.

Another example of a non-computable function is the busy beaver function, which takes a positive integer n as input and returns the maximum number of steps that a Turing machine with n states can run before halting.

The busy beaver function grows faster than any computable function, meaning that there is no algorithm or Turing machine that can compute this function for all possible inputs.

In summary, computable functions are those that can be computed using algorithms or Turing machines, while non-computable functions are those that cannot. While many natural functions are computable, there are also many interesting and important non-computable functions in computer science and mathematics.

3.2 Decidability and Decision Problems

Decidability refers to the ability of a problem or language to be solved or recognized by an algorithm, within a finite amount of time and using a finite amount of resources. A problem or language is said to be decidable if there exists an algorithm that can correctly determine whether any given input e.g. belongs to the language or satisfies the problem.

Decidability is closely related to the concept of computability, which refers to the ability of a function or problem to be solved by an algorithm or computation. In particular, all decidable problems and languages are computable, but not all computable problems and languages are decidable.

An example of a decidable problem that is computable is the problem of determining whether a given natural number is prime. There exists an algorithm, called the Sieve of Eratosthenes, that can determine whether a given natural number is prime or not, and the algorithm terminates in a finite amount of time for any input. An example of a computable problem that is not decidable is the halting problem, which is the problem of determining whether a given program, when executed on a particular input, will eventually terminate or run forever. Although there are algorithms that can partially solve the halting problem for certain cases, there is no algorithm that can solve the halting problem for all possible programs and inputs, making the problem undecidable.

Decidability is important in computer science because it provides a way to determine the limits of what can and cannot be computed. For example, if a problem is undecidable, it means that there is no algorithm that can solve it for all possible inputs, which has significant implications for computer programs and software systems that may encounter such problems in practice.

To be more accurate, the difference between decidability and computability lies in the level of certainty and efficiency of the algorithms used to solve or recognize problems or languages.

Decidability refers to the ability of a problem or language to be solved or recognized with 100% certainty by a specific algorithm within a finite amount of time and resources. This means that for any instance of a decidable problem or language, the algorithm will always give a correct answer, either “yes” or “no”.

Computability, on the other hand, refers to the broader concept of the ability of a function or problem to be solved by an algorithm or computation in general, without necessarily guaranteeing 100% correctness or efficiency.

A computable problem or language can be solved or recognized by an algorithm, but the algorithm may not always give a correct answer or may require an impractically large amount of time and resources.

All decidable problems and languages are computable because they can be solved or recognized by algorithms, but not all computable problems and languages are decidable because some may not have algorithms that can guarantee 100% correctness or efficiency.

Decidability is also relevant to many other fields of study, including mathematics, logic, and philosophy. In mathematics, the concept of decidability is closely related to the idea of a proof, as a problem is decidable if and only if there exists a proof that can be verified in a finite amount of time.

In logic and philosophy, decidability is often used to analyze the limits of knowledge and reasoning, and to investigate the nature of truth and certainty.

Decision problems and their computational complexity

Decision problems are a type of computational problem that can be answered with either a “yes” or “no” answer. In other words, they involve deciding whether a given input satisfies a certain property or meets a certain criterion. Decision problems can be categorized based on their computational complexity, which is a measure of the amount of computational resources needed to solve them.

One way to measure computational complexity is to consider the running time of an algorithm that solves the problem. This is often expressed as a function of the input size, typically denoted by n. For example, an algorithm with running time O(n) means that the time required to solve the problem is proportional to the size of the input.

Decision problems can be classified based on their computational complexity into three categories:

- Decidable Problems: Decidable problems are those for which there exists an algorithm that can correctly decide whether a given input satisfies the given property in a finite amount of time. This means that the running time of the algorithm is bounded by some function of the input size.
- Undecidable Problems: Undecidable problems are those for which no algorithm can correctly decide whether a given input satisfies the given property in a finite amount of time. This means that there is no algorithm that can solve the problem for all possible inputs.
- Semi-decidable Problems: Semi-decidable problems are those for which there exists an algorithm that can correctly decide whether a given input satisfies the given property, but may not halt on all inputs that do not satisfy the property. This means that the running time of the algorithm is not necessarily bounded by a function of the input size, and may not halt for some inputs that do not satisfy the property.

In general, decidable problems are considered to be the most tractable, while undecidable problems are considered to be the most difficult. Semidecidable problems lie somewhere in between. Many important decision problems in computer science and mathematics have been shown to be undecidable, including the halting problem for Turing machines and the problem of determining whether a given Diophantine equation has a solution.

Decidable and undecidable problems for different types of formal languages

The **word problem** is decidable for regular, context-free, and context-sensitive grammars because these grammars have effective algorithms for parsing words and recognizing the language generated by the grammar.

word problem
The word problem is about determining whether a given word is generated by a formal language.

For regular grammars, a finite state automaton can be constructed that recognizes the language generated by the grammar. The word problem for regular grammars can be solved by simulating the automaton on the input and determining whether the automaton accepts the word.

For context-free grammars, a pushdown automaton can be constructed that recognizes the language generated by the grammar. The word problem for context-free grammars can be solved by simulating the pushdown automaton on the input and determining whether the automaton accepts the word (see also CYK algorithm and LR parsing).

For context-sensitive grammars, linear-bounded automata can be constructed that recognize the language generated by the grammar. The word problem for context-sensitive grammars can be solved by simulating the linearbounded automaton on the input and determining whether the automaton accepts the word. A linear-bounded automaton (LBA) is a type of Turing machine that has a tape with a length proportional to the length of the input, making it a more restricted model of computation than a general Turing machine.

In all of these cases, the word problem is decidable because there exists an algorithm that terminates and correctly answers whether a given word belongs to the language generated by the grammar.

emptiness problem
The emptiness problem is about determining whether a formal language contains any words.

The**emptiness problem** is decidable for regular and context-free grammars because effective algorithms exist for both of these grammar types that can recognize whether the language generated by the grammar is empty.

For regular grammars, a finite state automaton can be constructed that recognizes the language generated by the grammar. The emptiness problem for regular grammars can be solved by checking if the automaton accepts any word.

For context-free grammars, a pushdown automaton can be constructed that recognizes the language generated by the grammar. The emptiness problem for context-free grammars can be solved by checking if the automaton accepts the empty word.

However, the emptiness problem is undecidable for context-sensitive and unrestricted grammars. This is because these grammars are more powerful than regular and context-free grammars, and thus, there is no general algorithm that can decide whether their languages are empty or not.

For context-sensitive grammars and unrestricted grammars, the problem reduces to the halting problem, which is known to be undecidable. This means that there is no algorithm that can determine whether a given word is generated by a context-sensitive or unrestricted grammar.

finiteness problem
The finiteness problem is about determining whether a formal language is finite or infinite.

The **finiteness problem** is decidable for regular and context-free grammars because it is possible to construct a finite automaton or pushdown automaton that can recognize a finite language, respectively. However, the finiteness problem is undecidable for context-sensitive and unrestricted grammars because these grammars can generate languages that are recursively enumerable but not decidable, which means that a Turing machine cannot determine whether the language generated by the grammar is finite or infinite.

equivalence problem
The equivalence problem is about determining if two grammars generate the same language.

The **equivalence problem** is decidable for regular grammars using finite automata, but it is undecidable for context-free, context-sensitive, and unrestricted grammars. This is because these grammars have more powerful generating capabilities than regular grammars, and a Turing machine cannot determine whether two such grammars generate the same language.

The table 3.1 summarizes the decidable and undecidable problems for different types of formal languages:

Table 5: Table 3.1: Decision Problems and their Decidability for Different Types of Grammars

Problem	Question	Type of Grammar			
		regular	contextfree	context-sensitive	type 0
Word	$w \in L$?	decidable	decidable	decidable	undecidable
Emptiness	$L = \emptyset$?	decidable	decidable	undecidable	undecidable
Finiteness	$\|L\| < \infty$?	decidable	decidable	undecidable	undecidable
Equivalence	$L_1 = L_2$?	decidable	undecidable	undecidable	undecidable

Applications of decision problems in computer science

Decision problems have many important applications in computer science. They are used in i.a. programming language theory, database systems, artificial intelligence, and computer security. Here are some examples:

- Type Checking: Type checking is the process of verifying that a program satisfies certain type rules. Type checking is an example of a decidable problem, as there are algorithms that can decide whether a given program satisfies certain type rules.
- Database Query Optimization: Database query optimization involves finding the most efficient way to execute a given database query.

  This problem is an example of a semi-decidable problem, as there are algorithms that can find a good query plan, but there may not be an algorithm that can guarantee finding the best query plan.
- Automated Theorem Proving: Automated theorem proving involves finding proofs of mathematical theorems using computer algorithms.

  This problem is an example of a semi-decidable problem, as there are algorithms that can find proofs of some theorems, but there may not be an algorithm that can find proofs of all theorems.

- Compiler Optimization: Compiler optimization involves finding ways to optimize the performance of compiled code. This problem is an example of a semi-decidable problem, as there are algorithms that can find some optimizations, but there may not be an algorithm that can find the best possible optimizations.
- Program Analysis: Program analysis involves analyzing the behavior of computer programs to find errors, security vulnerabilities, or performance bottlenecks. Many program analysis problems are examples of decidable or semi-decidable problems, including data flow analysis, and pointer analysis.

In summary, decision problems have many important applications in computer science, and understanding their computational complexity is essential for developing efficient algorithms and software systems.

3.3 Complexity Theory

In complexity theory, time complexity and space complexity are measures of the amount of resources required by an algorithm to solve a problem.

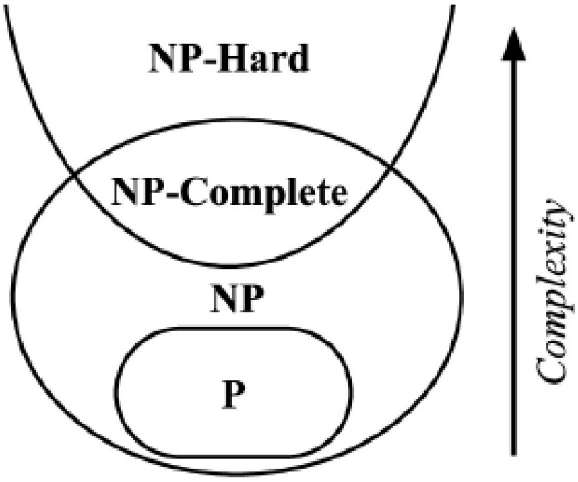
Time complexity measures the number of steps or operations required by an algorithm to solve a problem, as a function of the input size. This function is usually denoted by a big-O notation, where $O(f(n))$ denotes that the number of steps required by the algorithm is bounded by a constant multiple of a function $f(n)$ for sufficiently large input sizes. For example, an algorithm with time complexity $O(n^2)$ would require at most $c \cdot n^2$ steps to solve a problem of size n, where c is a constant (cf. section 1.4).

Space complexity measures the amount of memory or storage required by an algorithm to solve a problem, also as a function of the input size. This function is usually denoted by a big-O notation, where $O(g(n))$ denotes that the amount of memory required by the algorithm is bounded by a constant multiple of a function $g(n)$ for sufficiently large input sizes. For example, an algorithm with space complexity $O(n)$ would require at most $c \cdot n$ units of memory to solve a problem of size n, wher ec is a constant.

Time and space complexity are important factors in algorithm analysis and design, as they help determine the efficiency and scalability of an algorithm.

Therefore, complexity classes are used to classify problems according to the amount of resources (such as time and space) required to solve them (cf. fig. 3.1).

Figure 24: Figure 3.1: Relations between Complexity Classes (cf. (Monteiro, 2019))



complexity classes Complexity classes categorize problems based on their computational difficulty and the resources

There are several different **complexity classes** including:

required to solve them.

1. P (Polynomial time): A problem is in the complexity class P if it can be solved in polynomial time, which means that the time complexity of the algorithm for solving the problem is a polynomial function of the size of the input. In other words, the class of decision problems that can be solved by a deterministic Turing machine in polynomial time. Problems in P are considered to be efficiently solvable. Examples of problems in P include sorting and searching.
2. NP (Non-deterministic polynomial time): A problem is in the complexity class NP if it can be verified in polynomial time, but not necessarily solved in polynomial time. That is, there exists a polynomialtime algorithm that can verify the solution to the problem, but there may not be a polynomial-time algorithm that can find the solution.

   In other words, the class of decision problems that can be solved by a nondeterministic Turing machine in polynomial time. Examples of problems in NP include the satisfiability problem.
3. NP-hard (Non-deterministic polynomial time hard): A problem is NP-hard if it is at least as hard as the hardest problem in NP. In other words, if there exists a polynomial-time algorithm that can solve any NP-hard problem, then it can also solve all problems in NP. An example of an NP-hard problem is the Boolean satisfiability problem and the traveling salesman problem.
4. NP-complete (Non-deterministic polynomial time complete): A problem is NP-complete if it is both NP-hard and in NP. In other words, it is the hardest problem in NP. Examples of NP-complete problems include the Boolean satisfiability problem and the traveling salesman problem.

The relationship between these complexity classes is a central question in complexity theory. It is not currently known whether P equals NP, which would imply that all NP problems are also in P and can be solved in polynomial time. The Clay Mathematics Institute has offered a $1 million prize for the solution to this problem, known as the P versus NP problem.

The relationship between NP classes can be visualized as a Venn diagram, with NP being the set of problems that can be verified in polynomial time, NP-hard being the set of problems that are at least as hard as the hardest problems in NP, and NP-complete being the set of problems that are both in NP and NP-hard. All NP-complete problems are in NP-hard, but not all NP-hard problems are in NP-complete. It is an open question whether NP-complete problems can be solved in polynomial time or not, and the answer to this question is one of the most important open problems in computer science.

Additionally, in complexity theory, PSPACE and EXPSPACE are two classes of decision problems that involve the amount of memory used during computation. PSPACE, which stands for Polynomial Space, is the class of decision problems that can be solved by a deterministic Turing machine using a polynomial amount of memory. On the other hand, EXPSPACE, which stands for Exponential Space, is the class of decision problems that can be solved by a deterministic Turing machine using an exponential amount of memory. This means that the size of the memory required to solve problems in EXPSPACE grows exponentially with the size of the input. In summary, PSPACE includes all problems that can be solved with polynomial memory, while EXPSPACE includes all problems that can be solved with exponential memory. PSPACE is a subset of EXPSPACE, since any problem that can be solved using polynomial memory can also be solved using exponential memory, but the reverse is not true.

For completeness, PTIME (polynomial time) and EXPTIME (exponential time) are complexity classes that describe the time complexity of algorithms.

The class PTIME includes all decision problems that can be solved by a deterministic Turing machine in polynomial time, i.e., in a number of steps that is polynomial in the size of the input. This class includes many practical algorithms for a wide range of problem domains. In contrast, the class EXPTIME includes all decision problems that can be solved by a deterministic Turing machine in exponential time, i.e., in a number of steps that is exponential in the size of the input. This class includes many problems that are intractable in practice, as the running time of algorithms that solve them grows too quickly as the input size increases. In general, PTIME algorithms are considered to be efficient, while EXPTIME algorithms are considered to be inefficient for most practical purposes.

In summary, the relations between the complexity classes can be described in sets (cf. fig. 3.1):

- P is a subset of NP (P $\subset$ NP).
- NP-hard is a superset of NP (NP-hard $\supset$ NP).
- NP-complete is the intersection of NP and NP-hard (NP-complete = NP $\cap$ NP-hard).
- PSPACE is a superset of NP (PSPACE $\supset$ NP).
- EXPSPACE is a superset of PSPACE (EXPSPACE $\supset$ PSPACE).

Complexity theory has a wide range of applications in computer science, including algorithm design and analysis. The insights and techniques provided by complexity theory can help identify and address bottlenecks in algorithms, leading to more efficient and effective solutions. For example, knowing the complexity class of a problem can guide the design of algorithms to solve that problem. If a problem is known to be in P, then it is possible to design an algorithm that solves the problem in polynomial time. On the other hand, if a problem is NP-hard, then it is unlikely that there exists a polynomial-time algorithm for that problem, and so other approaches must be considered, such as heuristics or approximation algorithms.

Complexity theory can also be used to analyze the performance of algorithms.

For example, it can be used to determine the worst-case and average-case time complexity of an algorithm, and to analyze the space complexity of an algorithm. This information can be used to guide algorithm selection and optimization. Overall, complexity theory provides valuable insights and tools for algorithm design and analysis, helping to improve the efficiency and effectiveness of computational systems in a wide range of applications.

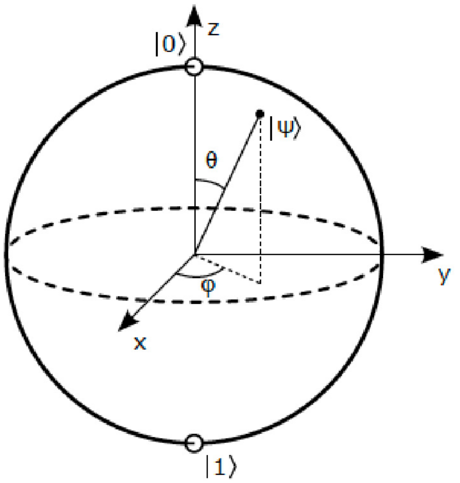
3.4 Quantum Computing

Quantum computing is a field of computer science and physics that explores the use of quantum mechanics to process and store information.

While classical computers use bits that can only be in a state of 0 or 1, quantum computers use quantum bits, or **qubits**, which can exist in a superposition of 0 and 1 states, meaning they can exist in both states simultaneously. Qubits are typically represented using a vector in a twodimensional complex vector space, known as the Bloch sphere. The state of a qubit can be visualized as a point on the surface of the Bloch sphere, with the north and south poles representing the two classical states 0 and 1. This allows quantum computers to perform certain tasks much faster than classical computers, and opens up the possibility of solving problems that are currently intractable.

qubits
A qubit is a unit of quantum information that can exist in a superposition of two states.

Figure 25: Figure 3.2: Bloch Sphere as Geometrical Representation of a Qubit (cf.(Smite-Meister, 2022))



The **superposition** of qubits refers to the fact that a qubit can exist in a linear combination of its possible states, with each state having a complex coefficient (cf. fig. 3.2). Specifically, given a qubit in the quantum state $|\psi\rangle$, it can be represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$, and $|0\rangle$ and $|1\rangle$ represent the two possible standard basis states of the qubit. The coefficients α and β are known as probability amplitudes, and their squared magnitudes give the probabilities of measuring the qubit in the corresponding state. The superposition of qubits allows for quantum parallelism and the ability to perform multiple calculations simultaneously, leading to the potential for exponential speedup in certain quantum algorithms.

superposition
Superposition is the ability of a qubit to exist in multiple states simultaneously.

In other words, when a qubit is measured, it collapses to one of its possible states with a probability determined by the squared magnitude of its probability amplitude. For example, if $\alpha = 0.6$ and $\beta = 0.8$, the probability of measuring the qubit in the state $0\rangle$ is $|\alpha|^2 = 0.36$, and the probability of measuring it in the state $|1\rangle$ is $|\beta|^2 = 0.64$.

Besides qubits, quantum gates are the building blocks of quantum computing.

Quantum gates are the equivalent of classical logic gates in a quantum computer. They are the basic building blocks of quantum circuits, which are analogous to classical circuits. Quantum gates operate on qubits and can perform operations such as superposition, entanglement, and phase shift. Overall, quantum gates and qubits are the fundamental building blocks of quantum computing, allowing for the development of quantum algorithms that can solve certain problems faster than classical algorithms.

Quantum algorithms are a set of instructions designed to be executed on a quantum computer to solve specific problems. Two of the most famous quantum algorithms are Shor's algorithm and Grover's algorithm.

Shor's algorithm
Shor's algorithm is a quantum algorithm for factoring large integers efficiently.

Shor's algorithm is a quantum algorithm that can be used to factorize large integers, which means it can be used to find the prime factors of a composite integer. This is considered to be a difficult problem for classical computers. In more detail, the algorithm consists of two main steps: first, the quantum Fourier transform is applied to the input number, and then a period-finding algorithm is used to determine the period of a modular function. The period can then be used to find the prime factors of the original number. The algorithm is named after its inventor, mathematician Peter Shor, and was first published in 1994. The algorithm relies on the properties of quantum computing to perform a large number of calculations simultaneously, leading to an exponential speedup over classical algorithms.

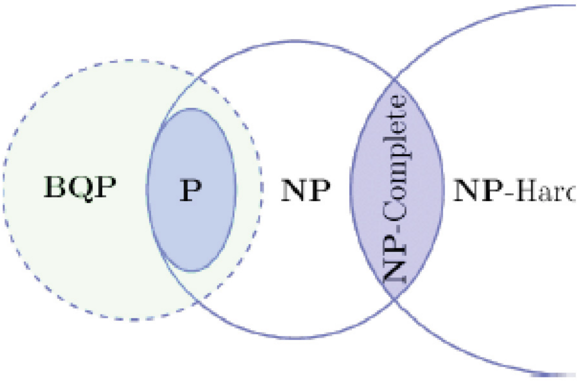
Shor's algorithm is important because it is exponentially faster than the best-known classical algorithms for integer factorization. This has significant implications for cryptography, as many public key encryption schemes rely on the difficulty of factoring large composite numbers. With the development of quantum computers capable of running Shor's algorithm, these encryption schemes would become vulnerable to attacks.

Grover's algorithm
Grover's algorithm is a quantum algorithm that provides quadratic speedup for searching an unsorted database.

Grover's algorithm is a quantum algorithm that can be used to search an unsorted database. The algorithm uses the concept of quantum parallelism to search through all possible solutions simultaneously, leading to a quadratic speedup over classical algorithms. This algorithm has important implications for data search and optimization problems.

Both Shor's and Grover's algorithms illustrate the potential power of quantum computing in solving problems that are difficult or impossible for classical computers to solve efficiently. However, building a practical quantum computer with enough qubits and stability to run these algorithms at scale remains a significant challenge.

Figure 26: Figure 3.3: Complexity Classes including BQP (cf. (Padilha, 2014))



In quantum computing, there are several complexity classes that describe the power of quantum computers compared to classical computers. One of the most important quantum complexity classes is BQP (bounded-error quantum polynomial time). BQP is the set of decision problems that can be solved by a quantum computer in polynomial time with a bounded probability of error. More formally, a decision problem is in BQP if there exists a quantum algorithm that solves the problem with an error probability of at most 1/3 for all instances, and runs in polynomial time with respect to the input size. BQP is an important complexity class because it contains several problems that are believed to be hard for classical computers, such as factoring large integers and finding discrete logarithms. Shor's algorithm is a well-known example of a quantum algorithm that runs in BQP.

However, it is not yet clear whether BQP contains NP, it is contained in NP, or whether both classes have an intersection (cf. fig. 3.3). If BQP is contained in NP, then some problems that can be solved efficiently on a quantum computer could also be solved efficiently on a classical computer.

On the other hand, if BQP contains NP, then some problems that are believed to be intractable on a classical computer could be solved efficiently on a quantum computer. This is still an open question in the field of quantum computing.

Quantum computing has the potential to revolutionize several fields due to its ability to solve problems that classical computers cannot. Some of the promising applications of quantum computing are:

- Cryptography: Quantum computers can break some of the currently used public-key cryptography systems, such as RSA and elliptic curve cryptography. On the other hand, post-quantum cryptography (also called quantum-resistant cryptography) provides a secure way of communication, for example, through quantum key distribution protocols, post-quantum public-key encryption and digital signature algorithms.
- Optimization: Many real-world problems, such as resource allocation, scheduling, and route optimization, can be formulated as optimization problems. Quantum computers can solve some of these optimization problems much faster than classical computers, for example, through the use of quantum annealing algorithms.

- Simulation: Quantum computers can simulate the behavior of quantum systems, which is difficult or impossible to do on classical computers.

  This has applications in areas such as material science, drug discovery, and climate modeling.
- Machine learning: Quantum computers can potentially improve machine learning algorithms by enabling faster training and inference on large datasets.
- Financial modeling: Quantum computers can be used for financial modeling, such as portfolio optimization and risk management, by enabling faster and more accurate simulations of complex financial systems.

Overall, quantum computing has the potential to impact several fields by providing solutions to currently unsolvable problems or enabling faster and more efficient solutions to existing problems.

SUMMARY

Computability, decidability, and decision problems are fundamental concepts in theoretical computer science. These concepts deal with the notion of whether a problem can be solved by an algorithm or not, and whether it can be solved efficiently. The study of complexity theory explores the relationships between different types of computational problems and the resources required to solve them. It also defines complexity classes like P, NP, NP-hard, and NP-complete, which help to categorize problems based on their computational difficulty.

Quantum computing is a rapidly developing field that seeks to exploit quantum mechanics to solve certain computational problems more efficiently than classical computers. Quantum computers use qubits, which can exist in a superposition of states, to perform parallel computations. They also use quantum gates to manipulate qubits, allowing for more complex operations to be performed.

UNIT 4

LOGIC

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the syntax and semantics of propositional and predicate logic
- apply equivalence and normal forms
- apply the translation between natural language and propositional resp. predicate logic
- apply resolution and tableau calculus

4. LOGIC

Introduction/Case study

The topics propositional logic, predicate logic, resolution calculus, and tableau calculus are fundamental topics in mathematical logic and computer science that form the basis of automated reasoning and theorem proving.

Propositional logic (also known as sentential logic) deals with the study of logical propositions and their connectives. It is a branch of symbolic logic that studies logical relationships between propositions, which are statements that can be either true or false.

Predicate logic extends propositional logic (also known as first-order logic) to include quantifiers and predicates. It is an extension of propositional logic that allows for the use of quantifiers and variables to describe more complex logical relationships.

The resolution calculus is a method for automated theorem proving in logic that uses a form of proof by contradiction, where a negation of the statement to be proved is assumed and then used to derive a contradiction.

Tableau calculus, also known as tableaux, is a method for reasoning in logic that involves constructing a tree-like structure of possible interpretations of a logical statement, and then attempting to find a closed branch that satisfies the statement.

Propositional and predicate logic, as well as resolution calculus and tableau calculus, can be applied in hardware design. These fields are used to design and verify digital circuits, including CPUs, memory controllers, and other digital components. They are also used in the verification of software and hardware systems to ensure their correctness and reliability. The tools and techniques used in these areas can help improve the efficiency and effectiveness of hardware design and verification.

Outline
Section 4.1 defines the syntax and semantics, logical equivalence, normal forms and applications for propositional logic. In section 4.2 are syntax and semantics, quantifiers and predicate transformations, logical equivalence, normal forms as well as application of predeicate logic. Section 4.3 describes the resolution calculus for propositional and predicate logic.

Section 4.4 explains the tableau calculus for propositional and predicate logic.

4.1 Propositional Logic

Propositional logic, also known as propositional calculus or sentential logic, is a branch of mathematical logic that studies propositions, which are statements that can be either true or false. Propositional logic is concerned with the manipulation of these propositions using logical operations to determine the truth value of compound propositions. It is widely used in computer science, artificial intelligence, and automated reasoning systems to model and reason about complex systems. In propositional logic, propositions are represented by symbols, and logical operators such as negation, conjunction, disjunction, implication, and equivalence are used to form complex propositions from simpler ones. The validity of a propositional logic formula can be determined using various techniques such as truth tables, logical equivalences, and deductive reasoning.

The syntax of propositional logic consists of three main components: propositional variables, logical connectives, and parentheses. Propositional variables are placeholders for propositions, usually denoted by letters such as $P, \quad Q,$ and $R.$ Logical connectives are symbols that connect propositions together, such as in binding order negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$). Parentheses are also used to group propositions and ensure the correct order of evaluation. Using these components, propositional logic allows us to construct compound propositions from simpler propositions, and to reason about their truth values based on the truth values of their constituent parts.

A technique to determine the validity of propositional logic statements is via truth tables. Before we can deal with combined statements, the basic propositional operators have to be introduced by truth tables with the truth value T (true) and F (false):

BASIC PROPOSITIONAL OPERATORS

Conjunction (AND)

Table 13

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction (OR)

Table 14

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

Negation (NOT)

Table 15

P	$\neg P$
T	F
F	T

Exclusive Or (XOR)

Table 16

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T

P	Q	$P \oplus Q$
F	F	F

Implication (IF...THEN)

Table 17

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Equivalence (IF AND ONLY IF)

Table 18

P	Q	$P \Leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

Here are some examples of combined statements in propositional logic translated from natural language:

- "If it is raining, then I will stay indoors and watch a movie." Symbolic notation: $P \Rightarrow (Q \wedge R)$ Meaning: If P is true (it is raining), then Q and R are true (I will stay indoors and watch a movie).
- "Either I will go for a run, or I will go to the gym." Symbolic notation: $S \vee T$Meaning: Either S is true (I will go for a run), or T is true (I will go to the gym).
- "If the store is open, and I have enough money, then I will buy a new shirt." Symbolic notation: $P \wedge Q \Rightarrow R$ Meaning: If both P and Q are true (the store is open and I have enough money), then R is true (I will buy a new shirt).
- "It is not the case that I will go swimming and stay outside all day." Symbolic notation: $\neg (S \wedge U)$ Meaning: S and U cannot both be true (I will go swimming and stay outside all day).

Let us consider the truth table for the first combined statement using the basic truth tables $P \Rightarrow (Q \wedge R)$:

Table 19

P	Q	R	$Q \wedge R$	$P \Rightarrow (Q \wedge R)$
T	T	T	T	T
T	T	F	F	F
T	F	T	F	F
T	F	F	F	F
F	T	T	T	T
F	T	F	F	T
F	F	T	F	T
F	F	F	F	T

The statement "If it is raining, then I will stay indoors and watch a movie" is an implication. It means that the occurrence of the antecedent (in this case, "it is raining") will lead to the occurrence of the consequent (in this case, "I will stay indoors and watch a movie"). If the antecedent is false (it is not raining), then the implication is automatically true regardless of the truth value of the consequent. The implication is only false if the antecedent is true and the consequent is false (i.e. if it is raining but the person does not stay indoors and watch a movie).

Logical equivalence and inference rules are essential concepts in propositional logic that help to establish the validity of arguments.

Logical equivalence refers to the idea that two statements have the same truth value, meaning they are either both true or both false, under all possible circumstances. In other words, they are equivalent. The symbol used to denote logical equivalence is $\equiv$. For example, the statements ($P \land Q$) $\equiv$ ($Q \land P$) and ($P \lor Q$) $\equiv$ ($Q \lor P$) are logically equivalent since they have the same truth value under all possible combinations of truth values for P and Q. Another example of logical equivalence is the distributive law, which states that ($P \land Q$) $\lor$ R is logically equivalent to ($P \lor R$) $\land$ ($Q \lor R$).

Logical equivalence
Logical equivalence is the relationship between two statements that have the same truth values under all possible interpretations.

Additionally, there are transformation rules that can be used to convert complex logical expressions into simpler forms or to prove the equivalence of different logical expressions. Important transformation rules are:

- Double Negation: $\neg\neg P$ is equivalent to P
- Always True: ($P \lor \neg P$) is a tautology, since it is always true that either P is true or P is false. It can be reduced to “True”.
- Always False: ($P \land\neg P$) is a contradiction, since it is never true that both P is true and P is false. It can be reduced to “False”.
- Identity: An expression that is equivalent to another expression, regardless of the truth values of its variables. For example, ($P \lor Q$) $\land \neg Q$ is equivalent to $P \land \neg Q$. It can be transformed with ($P \lor Q$) $\land \neg Q \equiv$ ($P \land \neg Q$) $\lor$ ($Q \land \neg Q$) $\equiv P \land \neg Q$[6].
- De Morgan’s Laws:
  – not(P and Q) is equivalent to (not P) or (not Q): $\neg$($P \land Q$) $\equiv \neg P \lor \neg Q$
  – not(P or Q) is equivalent to (not P) and (not Q): $\neg$($P \lor Q$) $\equiv \neg P \land \neg Q$
- Implication elimination: This rule states that the expression $P \Rightarrow Q$ is equivalent to $\neg P$ $\lor$ Q. This means that we can replace any implication in a logical expression with an equivalent disjunction.
- Equivalence elimination: This rule states that the equivalence symbol ($\Leftrightarrow$) can be replaced by a combination of implication ($\Rightarrow$) and reverse implication ($\Leftarrow$) symbols. This rule states that the expression $P \Leftrightarrow Q$ is equivalent to $P \Rightarrow Q \land Q \Rightarrow P$. Thus, the expression using the implication elimination is equivalent to ($\neg P \lor Q$) $\land$ ($\neg Q \lor P$).

By applying these rules (and others), we can transform complex logical expressions into simpler forms, such as conjunctive normal form (CNF) or disjunctive normal form (DNF), which can be easier to work with when using decision procedures.

Inference rules, on the other hand, are a set of rules that allow us to derive new valid statements from existing ones. These rules provide a systematic and rigorous way to establish the validity of arguments. There are many inference rules in logic, including modus ponens, modus tollens, hypothetical syllogism, disjunctive syllogism, and constructive dilemma.

INFERENCE RULES

Modus Ponens (MP) says if P implies Q and P is true, then we can infer that Q is true. Symbolically, ($P \land$ ($P \Rightarrow Q$)) $\Rightarrow Q$.

[6] $Q \land \neg Q$ can be transformed to False by rule Always False.

Modus Tollens (MT) says if P implies Q and Q is false, then we can infer that P is false. Symbolically, $(\neg Q \land (P \Rightarrow Q)) \Rightarrow \neg P$.

Hypothetical Syllogism (HS) says if P implies Q and Q implies R, then we can infer that P implies R. Symbolically, $((P \Rightarrow Q) \land (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)$.

Disjunctive Syllogism (DS) says if P or Q is true, but not both, and P is false, then we can infer that Q is true. Symbolically, $((P \lor Q) \land \neg P) \Rightarrow Q$.

Constructive Dilemma (CD) says if P implies Q and R implies S, and P or R is true, then we can infer that Q or S is true. Symbolically, $(((P \Rightarrow Q) \land (R \Rightarrow S)) \land (P \lor R)) \Rightarrow (Q \lor S)$.

These inference rules are used in logical reasoning to establish the validity of arguments and to derive new statements from existing ones. They are a fundamental part of propositional logic and predicate logic. The inference rules are truth-functional **tautologies**. A proof of the inference rules is to show that they are tautologies. This can be shown by truth tables. The following truth table demonstrates the proof of the tautology of Modus Ponens. The other inference rules can be proven analogously.

tautologies
A tautology is a statement that is always true, regardless of the truth values of its propositional variables.

Table 20

P	Q	$P \Rightarrow Q$	$P \land (P \Rightarrow Q)$	$(P \land (P \Rightarrow Q)) \Rightarrow Q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

In general, there are three types of a proposition or statement. Tautology, contradiction, and contingency are terms used to describe the truth value of a proposition or statement.

A tautology is a proposition that is always true, regardless of the truth values of its individual components. For example, the statement “Either it will rain tomorrow or it will not rain tomorrow” is a tautology because it is true no matter what the weather is like.

A contradiction is a proposition that is always false, regardless of the truth values of its individual components. For example, the statement “It is raining and it is not raining” is a contradiction because it cannot be both raining and not raining at the same time in the same place.

A contingency is a proposition that is neither a tautology nor a contradiction.

It is a statement whose truth value depends on the truth values of its individual components. For example, the statement "It will rain tomorrow" is a contingency because it is true or false depending on whether or not it actually rains tomorrow.

Important concepts are normal forms that are standard representations of a formula that makes it easier to analyze and reason about. There are several normal forms, including the conjunctive normal form (CNF) and disjunctive normal form (DNF). In CNF, a formula is expressed as a conjunction (i.e. AND) of **clauses** , where each clause is a disjunction (i.e. OR) of literals (i.e. either a propositional variable or its negation). For example, the CNF of the formula $(P \Rightarrow Q) \wedge \neg R$ would be $(\neg P \vee Q) \wedge \neg$R.

clauses
A clause is a disjunction (logical OR) of literals, where a literal is either a propositional variable or its negation.

Let's consider the propositional statement: "If it is sunny or if it is cloudy, then I will go for a walk if it is not too cold": $(P \vee Q) \Rightarrow (R \wedge \neg S)$. To derive its conjunctive normal form (CNF), we can follow these steps:

$$\begin{aligned}
&(P \vee Q) \Rightarrow (R \wedge \neg S) \\
&\equiv \neg(P \vee Q) \vee (R \wedge \neg S) \quad (Impl. \quad elim.) \\
&\equiv (\neg P \wedge \neg Q) \vee (R \wedge \neg S) \quad (De \quad Morgan) \\
&\equiv ((\neg P \vee R) \wedge (\neg P \vee \neg S)) \wedge ((\neg Q \vee R) \wedge (\neg Q \vee \neg S)) \\
&(Distribution) \\
&\equiv ((R \vee \neg P) \wedge (\neg S \vee \neg P)) \wedge ((R \vee \neg Q) \wedge (\neg S \vee \neg Q)) \\
&(Commutation) \\
&\equiv (R \vee \neg P) \wedge (R \vee \neg Q) \wedge (\neg S \vee \neg P) \wedge (\neg S \vee \neg Q) \quad (Associativity \\
&)
\end{aligned}$$

After the step Distribution, the statement is almost in CNF. The steps Commutation and Associativity are used to explain them and to reach the CNF.

In **Disjunctive Normal Form** , a formula is expressed as a disjunction of conjunctions of literals. For example, the DNF of the formula $(P \Rightarrow Q) \wedge \neg R$ would be $(\neg P \wedge \neg R) \vee (Q \wedge \neg R)$:

Disjunctive Normal Form
Disjunctive Normal Form is a way of writing logical statements as a disjunction (OR) of conjunctions (ANDs) of literals.

$$\begin{aligned}
(P \Rightarrow \ Q) \wedge \neg R &\equiv (\neg P \vee Q) \wedge \neg R \quad (Implication \quad elimination) \\
&\equiv (\neg P \wedge \neg R) \vee (Q \wedge \neg R) \quad (Distribution)
\end{aligned}$$

Converting a formula to CNF or DNF can be useful for simplifying and analyzing it, and also for automated reasoning using decision procedures.

Overall, normal forms and decision procedures are important tools in logic, providing ways to simplify and analyze complex logical statements.

The truth table of a propositional statement can be used to derive its corresponding conjunctive normal form (CNF) or disjunctive normal form (DNF), and vice versa. The CNF and DNF can also be used to determine the truth value of the propositional statement for any assignment of truth values to its constituent variables. The literals are A,B,C and F represents the propositional statement:

Table 21

A	B	C	F
F	F	F	F
F	F	T	F
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	T

To convert the truth table into CNF, we first identify the rows where the output F is False, and then create a clause for each of those rows, where each literal in the clause is negated if its corresponding input is True, and not negated if the input is False.

From this truth table, we see that the output F is False for the first, second and fourth rows. Therefore, we create three clauses for these rows and combine them to get the CNF:

To obtain the DNF, we take the rows where the output F is True, and form a disjunction that correspond to those rows, where each literal in the clause is negated if its corresponding input is False, and not negated if the input is True.

$$\begin{array}{l}(A \vee B \vee C) \wedge \\ (A \vee B \vee \neg C) \wedge \\ (A \vee \neg B \vee \neg C)\end{array}$$

From the truth table, we see that the output F is True for the third, fifth, sixth, seventh and eighth rows. Therefore, we connect them with AND for these rows and combine them to get the DNF:

$$\begin{array}{l}(\neg A \wedge B \wedge \neg C) \vee \\ (A \wedge \neg B \wedge \neg C) \vee \\ (A \wedge \neg B \wedge C) \vee \\ (A \wedge B \wedge \neg C) \vee \\ (A \wedge B \wedge C)\end{array}$$

In propositional logic, the reduction of a DNF or CNF involves the simplification of logical expressions by combining terms that are redundant or logically equivalent, thereby reducing the number of terms. This process is useful for optimizing logical circuits and reducing the complexity of logical expressions. The resulting expression is equivalent to the original expression but has fewer terms, making it easier to analyze and understand. The reduction will not be considered here.

Propositional logic has numerous applications in computer science, including:

- Formal verification: Propositional logic is used to verify the correctness of hardware and software systems. By modeling the system as a set of logical statements and using deduction rules to infer new statements, one can determine if the system satisfies certain properties, such as security and liveness.
- Artificial intelligence: Propositional logic is used as a building block in many artificial intelligence systems. It provides a way to represent knowledge and reasoning in a formal, precise manner.
- Computer programming: Propositional logic is used in the design and analysis of computer algorithms. It can be used to reason about the complexity of an algorithm, to prove that it is correct, and to optimize its performance.
- Database systems: Propositional logic is used to express queries in database systems. By translating a query into a logical statement, one can use logical inference to determine if the query is true or false.
- Automated theorem proving: Propositional logic is used in automated theorem proving systems, which attempt to automatically prove mathematical theorems. By modeling a theorem as a logical statement, an automated system can use logical inference rules to determine if the theorem is true or false.

Overall, propositional logic provides a powerful tool for reasoning about complex systems in computer science, enabling the design, analysis, and verification of a wide range of systems and algorithms.

4.2 Predicate Logic

Predicate logic is an extension of propositional logic that allows for more complex statements to be represented and reasoned about. Predicate logic is required in reference to propositional logic because propositional logic is limited to analyzing simple statements that are either true or false, while predicate logic allows for more complex analysis of statements involving quantifiers and variables, making it more expressive and powerful for reasoning about the real world. In other words, predicate logic provides a richer language and more advanced tools for logical reasoning and inference than propositional logic. In predicate logic, propositions are represented using predicates, which are functions that take one or more arguments and evaluate them to be either true or false.

The syntax of predicate logic includes variables, constants, predicates, logical connectives, and quantifiers. Variables represent elements of a domain of discourse, while constants represent specific elements of that domain.

Predicates take variables or constants as arguments and evaluate to true or false depending on the values of those arguments. Logical connectives, such as conjunction, disjunction, and negation, allow for complex propositions to be formed. Quantifiers, such as universal and existential quantifiers, allow for statements to be made about all or some elements in the domain of discourse.

The semantics of predicate logic involves defining the truth value of statements in terms of the truth value of their atomic parts. The truth value of a statement involving a predicate is determined by evaluating the predicate for the values of its arguments. Universal quantifiers assert that a statement is true for all elements in the domain of discourse, while existential quantifiers assert that there exists at least one element in the domain of discourse for which the statement is true.

Predicate logic is a powerful tool for representing and reasoning about complex statements in many areas of mathematics and computer science, including artificial intelligence, database theory, and automated theorem proving. However, the increased complexity of predicate logic compared to propositional logic makes it more difficult to reason about and more computationally expensive to automate.

In a nutshell, predicate logic syntax and semantics are:

- Syntax
  – Predicate symbols: $P(x), Q(x, y), R(y, z)$
  – Function symbols: $f(x), g(x, y), h(z)$
  – Variables: x, y, z
  – Quantifiers: $\forall$ (for all), $\exists$ (there exists)
  – Logical operators: $\neg$(not), $\wedge$(and), $\vee$(or), $\Rightarrow$ (implies), $\Leftrightarrow$(if and only if)

- Semantics
  – A domain of discourse: a set of objects that the variables can take values from
  – Interpretation functions: a mapping of predicate symbols and function symbols to relations and functions on the domain, respectively
  – Truth values: a statement in predicate logic is true or false depending on the interpretation of the symbols and functions and the values of the variables in the domain.

For example, the statement in natural language “All dogs bark.” is in predicate logic

$$\forall x(Dog(x) \Rightarrow Bark(x))$$

In this example, $Dog(x)$ and $Bark(x)$ are predicates that refer to the properties of an object x. The **universal quantifier** $\forall x$ expresses that the statement holds for all possible values of x in a given domain of discourse, which in this case is the set of all dogs. The arrow $\Rightarrow$ indicates that if an object is a dog, then it must bark. Thus, the predicate logic statement captures the meaning of the natural language sentence.

universal quantifier
The universal quantifier expresses that a statement is true for all elements in the domain of discourse.

In predicate logic, quantifiers and variables play a crucial role in forming propositions that involve quantification over a set of objects or individuals.

The two main quantifiers used in predicate logic are the universal quantifier and the existential quantifier.

The universal quantifier, denoted by $\forall$, is used to express that a statement is true for all elements in a given set. For example in mathematics, the statement $\forall x \in \mathbb{N}, x > 0$ reads as "for all x in the set of natural numbers, x is greater than 0".

On the other hand, the **existential quantifier** , denoted by $\exists$, is used to express that a statement is true for at least one element in a given set. For example, the statement $\exists x \in \mathbb{N}, x^2 = 4$ reads as "there exists an x in the set of natural numbers such that x squared is equal to 4".

existential quantifier The existential quantifier expresses that there exists at least one instance of a variable for which a predicate is true.

Variables are used to denote the elements or individuals being quantified.

They can represent any element or individual from the domain of discourse.

For example, in the statement $\forall x \in \mathbb{N}, x > 0$, the variable x represents any natural number, and in the statement $\exists x \in \mathbb{N}, x^2 = 4$, the variable x represents the specific natural number whose square is equal to 4.

Thus, quantifiers and variables provide a powerful tool for expressing complex propositions in predicate logic. They allow us to make generalizations over sets of objects and reason about the existence of specific elements within those sets.

Logical equivalence, as in propositional logic, is an important concept in predicate logic as it allows us to simplify and manipulate complex logical expressions. Inference rules, such as modus ponens and modus tollens, also apply to predicate logic, allowing us to derive new logical statements from existing ones.

The inference rule Modus ponens allows us to infer the truth of a conclusion from the truth of two premises that have a specific form. The form of the premises is as follows:

- Premise 1: $\forall x(P(x) \Rightarrow Q(x))$
- Premise 2: $P(a)$, where a is a specific object in the domain of discourse

From these premises, we can infer that $Q(a)$ must also be true. This can be written as:

$$Conclusion: \quad Q(a)$$

The key to understanding why this rule is valid lies in the meaning of the universal quantifier. When we say that $\forall x(P(x) \Rightarrow Q(x))$ is true, we mean that every object in the domain of discourse that satisfies $P(x)$ also satisfies $Q(x)$. So when we have the additional premise $P(a)$, we know that a satisfies $P(x)$. And since we know that $\forall x(P(x) \Rightarrow Q(x))$, we can conclude that a must also satisfy $Q(x)$, and therefore $Q(a)$ must be true.

Overall, modus ponens is a powerful tool for reasoning in predicate logic, allowing us to make valid inferences based on the relationship between predicates in our domain of discourse.

In predicate logic, we can express statements such as “all cats are animals” or “some dogs are friendly” using quantifiers. For example, the statement “all cats are animals” can be expressed as $\forall x(Cat(x) \Rightarrow Animal(x))$, where $Cat(x)$ represents the property of x being a cat, and $Animal(x)$ represents the property of x being an animal. On the other hand, the statement “some dogs are friendly” can be expressed as $\exists x(Dog(x) \wedge Friendly(x))$, where $Dog(x)$ represents the property of x being a dog, and $Friendly(x)$ represents the property of x being friendly.

Predicate logic can be used to translate the natural language. Some examples including quantifiers, variables and constants are:

- “Every dog has a tail.”: $\forall x(Dog(x) \Rightarrow HasTail(x))$
- “There exists a student who has passed all exams.”:
  $\exists x(Student(x) \wedge \forall y(Exam(y) \Rightarrow Passed(x, y)))$
- “No person is immortal.”: $\forall x(Person(x) \Rightarrow \neg Immortal(x))$
- “Some apples are green.”: $\exists x(Apple(x) \wedge Green(x))$
- “There is a car that can drive faster than 200 km/h.”:
  $\exists\, \exists x(Car(x) \wedge CanDriveFasterThan(x, 200))$
- “Everyone has a favorite color.”: $\forall x\, \exists y(Person(x) \Rightarrow HasFavCol(x, y))$

Logical validity in predicate logic is defined in terms of truth tables. For example, the statement “For all real numbers x, if $x > 0$, then $x^2 > 0$”:

$$\forall x\big(x > 0 \Rightarrow x^2 > 0\big)$$

can be proven by truth tables. For this statement in predicate logic, we can create a truth table to check its validity as follows: First, we need to determine the domain of discourse, or the range of values that the variable can take. Let’s say that the domain of $x$is $\{1, 2, 3\}$. Next, we need to determine all possible truth values for the variable. In this case, we have:

Table 22

x	$x > 0$	$x^2 > 0$	$x > 0 \Rightarrow x^2 > 0$
1	T	T	T
2	T	T	T
3	T	T	T

In each row of the truth table, we evaluate the truth values of the atomic formulas, and then the truth value of the entire compound formula. We can see that in each row, the compound formula is true, which means that it is a tautology and is valid in the chosen domain of discourse.

Predicate logic formulas can be transformed into different normal forms to facilitate reasoning and computation. The two most commonly used normal forms for predicate logic formulas are the Prenex normal form and the Skolem normal form.

The **Prenex Normal Form** (PNF) is a way of expressing a formula in which all of its quantifiers are moved to the front of the formula. For example, the formula $(\forall x \exists y(P(x, y) \wedge Q(y))) \Rightarrow \exists x \forall y R(x, y)$ can be transformed into the PNF as $\forall x \exists y \exists z \forall w((P(x, y) \wedge Q(y)) \Rightarrow R(z, w))$. PNF is useful for simplifying reasoning by sometimes reducing the number of quantifiers that need to be considered.

Prenex Normal Form
Prenex Normal Form is a way to rewrite a formula by moving all quantifiers to the front of the formula.

The Skolem Normal Form (SNF) is a way of expressing a formula in which all of its existential quantifiers are eliminated by introducing new Skolem functions or Skolem constants. Skolem functions are functions that map an element in the domain of discourse to another element in the domain of discourse. Skolem constants are individual constants that represent some fixed element in the domain of discourse. **Skolem Normal Form** is useful for proving the satisfiability of a formula, as it eliminates existential quantifiers which make the formula hard to evaluate.

Skolem Normal Form
Skolem Normal Form is a transformation of formula to eliminate existential quantifiers.

An example is $\forall x \exists y(P(x, y) \Rightarrow \forall z Q(z, y))$ is equivalently transformed to $\forall x(P(x, f(x)) \Rightarrow \forall z Q(z, f(x)))$ where f is a Skolem function that maps each value of x to a corresponding value of y that satisfies the statement.

There are also decision procedures for predicate logic formulas. A decision procedure is an algorithmic method to determine whether a formula is valid, satisfiable, or unsatisfiable. One example of a decision procedure for predicate logic formulas is the resolution calculus. It is a method of proof used in predicate logic that involves converting a formula into clausal form and using the resolution rule to derive new clauses until a contradiction is found or it is shown that the formula is satisfiable.

Another example of a decision procedure for predicate logic formulas is tableau calculus. It is a systematic procedure for constructing a decision tree that determines the truth value of a formula. It recursively applies inference rules to a formula until a contradiction is found, indicating that the formula is unsatisfiable, or until a complete decision tree is constructed, indicating that the formula is satisfiable.

Predicate logic has a wide range of applications in computer science, including automated theorem proving, natural language processing, database systems, and artificial intelligence. In automated theorem proving, predicate logic is used to prove mathematical theorems automatically by applying logical inference rules. This is particularly useful in formal verification of hardware and software systems. In natural language processing, predicate logic is used to represent the meaning of sentences in a formal and structured way, allowing computers to understand and manipulate natural language text. This can be used for tasks such as text classification, information retrieval, and machine translation. In database systems, predicate logic is used as the basis for query languages such as SQL. This allows users to specify complex queries over large databases in a concise and precise way.

In artificial intelligence, predicate logic is used to represent knowledge and reasoning, allowing AI systems to reason about the world in a structured and logical way. This is particularly useful in domains such as expert systems, automated planning, and robotics. Overall, predicate logic is a powerful and versatile tool for representing knowledge and reasoning in a wide range of computer science applications.

4.3 Resolution Calculus

resolution calculus
The resolution calculus is a proof technique used to show that a statement is satisfiable by assuming its negation and deriving a contradiction.

The **resolution calculus** is a method in mathematical logic and automated theorem proving used for deriving logical consequences from a set of clauses or formulas, using the resolution rule as the main inference rule.

It is a proof calculus that operates on formulas in propositional or firstorder predicate logic, and can be used to prove the validity or satisfiability of a logical formula. The resolution calculus is a class of automated deduction methods that operate by attempting to find a refutation proof, which means that they aim to prove that a statement is false by deriving a contradiction. The resolution calculus is widely used in automated theorem proving, and has been applied to various areas of computer science, including verification, planning, and natural language processing.

The Resolution principle is a rule of inference used in the resolution calculus, which is a proof procedure for propositional and predicate logic. The principle is used to construct a refutation, which is a proof that a given statement is false.

Before using the resolution calculus for propositional logic, the logical statements must be converted to (1) negated CNF (conjunctive normal form), the CNF will be (2) transformed to the clausal form, and (3) the empty clause will be derived by building resolvents in the resolution process. Additionally, the statements should be organized into a set of clauses, each of which is a disjunction of literals. Once we have met these requirements, we can apply the resolution principle to derive new clauses until we either reach a contradiction or can no longer derive any new clauses.

Before applying the resolution calculus to propositional logic, the given formulas must be transformed into Conjunctive Normal Form (CNF) by applying the following steps (1):

1. Negate the entire expression (want to prove with a refutation proof)
2. Eliminate all equivalences and implications in the formula.
3. Move negations inwards using De Morgan’s laws and double negation elimination.
4. Distribute disjunctions over conjunctions using the distributive law.
5. Eliminate any remaining double negations.

For example, we consider the formula:

$A \vee (B \wedge C) \vee (\neg A \wedge \neg B) \vee (\neg A \wedge B \wedge \neg C)$

First, we negate the formula:

$$\neg(A \vee (B \wedge C) \vee (\neg A \wedge \neg B) \vee (\neg A \wedge B \wedge \neg C))$$

Next, we would eliminate the equivalences and implications but there are none. Thus, we move the negations inward using De Morgan’s laws and eliminate double negations:

$$\neg(A \lor (B \land C) \lor (\neg A \land \neg B) \lor (\neg A \land B \land \neg C))$$
$$\equiv \neg A \land (\neg B \lor \neg C) \land (A \lor B) \land (A \lor \neg B \lor C)$$

The next steps (distribute the disjunction over the conjunction, and eliminate any remaining double negations) can be skipped. Thus, we have the formula in CNF, and have to transform it into the clausal form (2). The clausal form is a representation of a logical formula in the form of a set of disjunctive clauses, where each clause is a disjunction of literals, i.e., either a variable or its negation. It is a standard form that is often used in automated theorem proving and logic programming. The clausal form for the example is

$$CF = \{\neg A, (\neg B, \neg C), (A, B), (A, \neg B, C)\}$$

We can now apply the resolution calculus to derive new formulas and check for satisfiability or validity (3). The resolution principle works by taking two clauses (disjunctions of literals) that share a complementary literal, meaning one clause has a literal and its negation appears in the other clause. The complementary literals are then resolved or eliminated by taking the disjunction of the remaining literals in each clause. This results in a new clause that is more general than the original clauses, as it contains all the literals that were not complementary.

This process is repeated until either a contradiction is obtained or no more new clauses can be generated. If a contradiction is obtained, then the negated statement is refuted, since it is impossible for all the premises to be true and the negation of the conclusion to be true at the same time.

Thus, it is proven that the original statement (before the negation of the entire expression) is satisfiable and in some cases also a tautology.

Let’s prove if the original formula is satisfiable. We use the resolution calculus in conjunction with the clausal form to find a contradiction.

Table 6

$\neg A, (\neg B, \neg C), (A, B), (A, \neg B, C)$	(initial: clausal from)
B	(5: clauses 1 and 3)
$(\neg B, C)$	(6: clauses 1 and 4)
$\neg C$	(7: clauses 2 and 5)
C	(8: clauses 5 and 6)
	(empty clause: clauses 7 and 8)

The calculus has the characteristic of maintaining the satisfiability of a statement, which means that the new set of clauses is satisfiable if and only if the original one is satisfiable. Therefore, if the rule is repeatedly applied and produces an empty clause, which represents a contradiction and is unsatisfiable, this is evidence to determine that the original set of clauses is also unsatisfiable.

The concepts of tautology and unsatisfiability are closely interconnected in logic. A formula is said to be a tautology if it is true under all possible truth assignments to its variables. On the other hand, a formula is said to be unsatisfiable if it is false under all possible truth assignments to its variables. In between these two areas lies satisfiability without a formula is being a tautology or unsatisfiability.

It turns out that a formula is a tautology if and only if its negation is unsatisfiable.

This can be seen by considering the truth table of the negation of the formula: if the negation is false under all possible truth assignments, then the original formula must be true under all possible truth assignments, and thus a tautology. Conversely, if the negation is true under no possible truth assignment, then the original formula must be false under all possible truth assignments, and thus a contradiction.

Similarly, in the resolution calculus, if we can derive the empty clause from a set of clauses, then we know that the set of clauses is unsatisfiable.

This is because the empty clause represents a contradiction, and if we can derive a contradiction from a set of clauses, then the set of clauses must be unsatisfiable. Therefore, we can say that the resolution calculus is a proof system for unsatisfiability, and by extension, for tautologies through the connection between tautology and unsatisfiability.

The resolution calculus can also be applied to predicate logic, but the process is more complex than for propositional logic. In predicate logic, we have variables that represent objects and predicates that represent relationships between those objects. To apply the resolution calculus to predicate logic, we first need to convert the formula into prenex normal form, where all the quantifiers (for all and there exists) are moved to the front of the formula. After this conversion, we can apply the same resolution rules as in propositional logic. However, we also need to introduce Skolem functions to eliminate existential quantifiers.

The resolution calculus can be a powerful tool for proving the validity or invalidity of formulas in predicate logic. However, the process of converting the formula into prenex normal form and introducing Skolem functions can be time-consuming and may require some creativity. Additionally, unlike in propositional logic, not all formulas in predicate logic have a finite resolution proof, which means that the process of proving validity or invalidity may not terminate in some cases.

Additionally, the resolution calculus has refutation completeness. It means that the resolution calculus is guaranteed to find a refutation if one exists, and this property makes it a powerful tool for automated theorem proving in artificial intelligence and computer science. However, the resolution calculus is not complete for all forms of logic, such as modal logic or intuitionistic logic, and it can also suffer from the problem of combinatorial explosion when dealing with large sets of clauses.

4.4 Tableau Calculus

Tableau calculus, also known as tableau method or tree method, is a

formal proof system used in logic to determine the validity of a formula or argument. It was introduced by the German philosopher and logician Georg Cantor in the late 19th century, and has since been further developed and refined by other logicians.

Tableau calculus
Tableau calculus is a proof technique in mathematical logic that uses a tree structure to systematically check the validity of a formula or set of formulas.

The tableau calculus involves the construction of a tree-like structure called a tableau or proof tree, where each node represents a set of formulas and each branch represents a step in the proof. The rules of the calculus are applied to each node to determine whether the formulas in the set are valid, contradictory, or undetermined. The calculus uses a systematic and algorithmic approach to exhaustively explore all possible ways to construct the proof tree.

One of the main advantages of the tableau calculus is its ease of use and ability to handle a wide variety of logical systems, including propositional logic, first-order predicate logic, modal logic, and many others. It also provides a clear and intuitive way to understand the structure of a proof, making it useful for both teaching and research.

However, one limitation of the tableau calculus is that it can be computationally expensive for complex formulas or systems, especially when dealing with large sets of formulas or infinite domains. Additionally, the calculus can sometimes generate proof trees that are difficult to interpret or visualize, making it challenging to extract meaningful information from the proof.

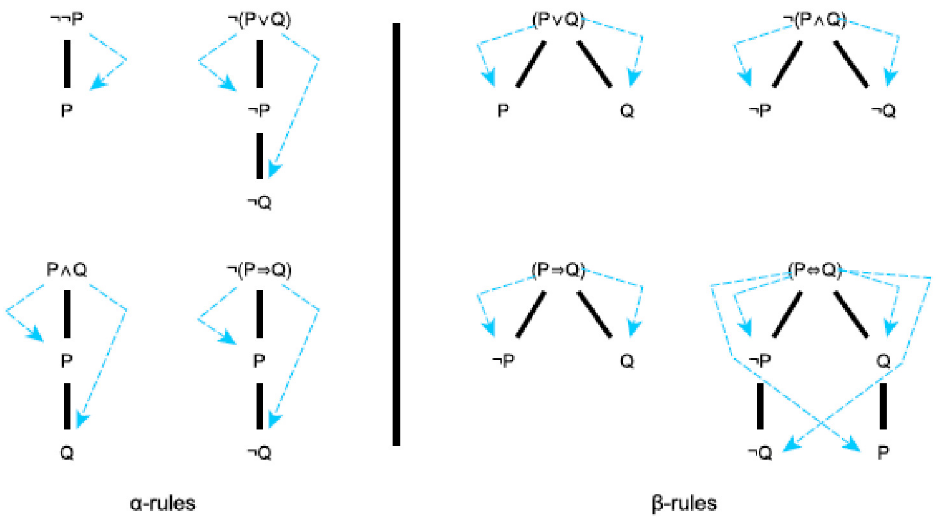
Overall, the tableau calculus is a powerful and versatile tool in logic that offers a systematic and algorithmic approach to determine the validity of a formula or argument. Its simplicity and generality make it a popular choice for both theoretical and practical applications in logic and computer science.

The syntax of the tableau calculus involves the construction of a tree-like structure called a tableau, which consists of a set of nodes connected by edges. Each node represents a formula, and the edges represent the logical relationships between these formulas. The tableau starts with a single node containing the negation of the formula being analyzed, and the goal is to construct a **closed tableau** in which every branch contains a contradiction.

closed tableau
A closed tableau is a completed tableau where all branches are closed, indicating that the formula is unsatisfiable.

Thus, the original formula is a tautology. If at least one branch is open then the original formula is satisfiable and the truth values can be derived from the open branch(es). If all branches are open then the original formula is unsatisfiable and thus a contradiction. The construction of the tableau has to derive every combined statement to its (atomic) literals or negated literals.

Figure 27: Figure 4.1: α and β Expansion Rules of the Tableau Calculus (adapted from (Hoffmann, 2009))



In a tableau, a branch is considered closed if it contains a pair of complementary literals, i.e., a literal and its negation. This means that both the literal and its negation cannot be simultaneously true, making the branch unsatisfiable. On the other hand, if a branch does not contain any complementary literals then it is considered an **open branch**.

open branch
An open branch in tableau calculus is a branch where no contradiction can be derived.

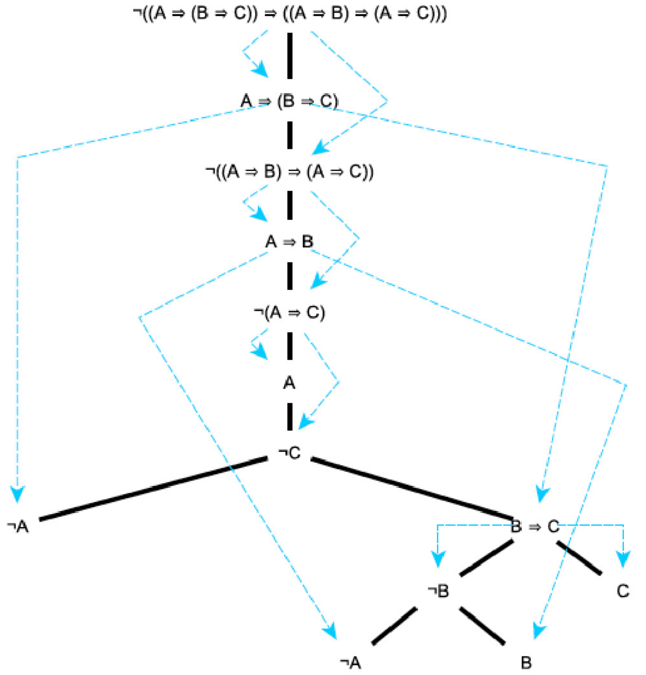
The semantics of the tableau calculus is based on the idea of satisfiability.

A formula is said to be satisfiable if there exists an interpretation of its variables that makes it true. The tableau calculus aims to show that a formula is unsatisfiable by constructing a closed tableau in which every branch contains a contradiction. If such a tableau can be constructed, then the formula is unsatisfiable, and its negation is a valid formula.

In tableau calculus, the alpha (α-rules) and beta (β-rules) expansion rules are used to systematically construct and evaluate a proof tree for a given formula. Apply the alpha rules first to any (sub-)formulas, such as double negation, conjunction, negative disjunction, and negative implication.

Then apply the beta rules such as disjunction, negative conjunction, implication, and equivalence (cf. fig. 4.1). The reason why the alpha rules are applied before the beta rules in the tableau calculus for propositional logic is that it ends in a reduced tableau (less branches) compared to the other way around. By using these alpha and beta rules, the tableau calculus can systematically evaluate the satisfiability of a given formula and construct a proof tree if one exists.

Figure 28: Figure 4.2: Tableau Calculus for $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$ (adapted from (Hoffmann, 2009))



For example, the formula $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$ is analyzed by tableau calculus. Firstly, the formula is negated to $\neg((A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)))$ and the tree-structure of the tableau is created using the α and β expansion rules (cf. fig. 4.2). Let's consider the most left branch from the negated formula (always start node) to $\neg A$.

It is a **closed branch** because it contains a pair of complementary literals A and $\neg A$. We follow this procedure for all other branches from left to right. The next branch contains also a pair of complementary literals A and $\neg A$ and the following contains B and $\neg B$. Finally, the most right branch contains C and $\neg C$. This means that all branches are closed and thus the whole tableau is closed. The result is that the negated formula is a contradiction and the original a tautology.

closed branch
A closed branch is a branch in a tableau where a contradiction is derived.

The method for predicate logic is similar to that of propositional logic, but with some important differences. In predicate logic, instead of propositions, we have predicates, functions, and quantifiers. To accommodate these new features, the tableau calculus includes additional rules for handling existential and universal quantifiers. The main idea is to create a tree-like structure, where each node represents a formula or a subformula. Tableau calculus for predicate logic is not part of this course book.

Soundness and completeness are important properties of any proof system, including tableau calculus. Soundness means that if a formula is provable in the calculus, then it must be valid. Completeness, on the other hand, means that if a formula is valid, then it must be provable in the calculus. In the context of tableau calculus for propositional logic, it can be shown that the calculus is both sound and complete. This means that every provable formula is valid, and every valid formula is provable using the calculus.

One of the main applications of tableau calculus is in automated theorem proving, where it can be used to decide whether a given formula is valid or satisfiable. It is also used in formal verification, where it can be used to verify the correctness of digital circuits, software programs, and other systems.

Tableau calculus is also used in natural language processing, where it can be used to parse and understand complex sentences. Additionally, it has applications in philosophy, where it can be used to analyze the logical structure of arguments and to evaluate the validity of philosophical claims.

SUMMARY

Propositional logic is a branch of mathematical logic that deals with propositions, their logical relationships, and their validity.

Propositional logic involves a set of rules for constructing complex propositions from simpler ones and for determining the truth of those propositions.

Predicate logic extends propositional logic by introducing variables and quantifiers, allowing for the expression of complex relationships between objects and properties.

The resolution calculus is a proof technique that can be used for propositional and predicate logic. It is used to show that a statement is satisfiable by assuming its negation and deriving a contradiction.

It can be used to prove if a statement is a contradiction or tautology.

The tableau calculus is another proof technique that can be used for propositional and predicate logic. The tableau calculus involves constructing a tree-like structure of formulas and applying a set of rules to determine if the formulas are satisfiable or not.

UNIT 5

ALGORITHM AND PROGRAM VERIFICATION

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the importance of program analysis in ensuring program correctness and reliability
- know the fundamental principles of static and dynamic program analysis
- understand the different types of semantics in programming languages, including algebraic, operational, and denotational semantics
- understand the concept of abstract interpretation and its role in program verification

5. ALGORITHM AND PROGRAM VERIFICATION

Introduction/Case study

Algorithm and program verification are crucial areas in computer science that deal with ensuring that software systems are correct and perform as intended. Verification techniques are employed to check the correctness of algorithms and programs, and to ensure that they meet specific requirements and (formal) specifications. There are several approaches to algorithm and program verification, including program analysis, algebraic, operational and denotational semantics, and abstract interpretation. Each of these techniques provides unique ways to analyze and verify programs and algorithms, with varying levels of precision and efficiency. In this context, the learning objectives are to understand the fundamental concepts, principles, and techniques in each of these areas, and to develop skills in applying them to practical problems in algorithm and program verification.

Outline

Algorithm and program verification covers program analysis (section 5.1), algebraic semantics, operational semantics, denotational semantics (section 5.2), and abstract interpretation (section 5.3). Program analysis involves examining the behavior of a program, typically for the purposes of detecting errors, improving performance, or optimizing code. Algebraic semantics is a mathematical framework for analyzing and understanding the behavior of programs, which involves representing programs as mathematical structures and defining their behavior in terms of algebraic properties.

Operational semantics is a framework for describing the behavior of programs in terms of operational steps or transitions, which can be used to prove the correctness of programs or reason about their behavior. Denotational semantics involves describing the meaning of programs in terms of mathematical objects or structures, such as functions or sets, and provides a way to reason about the behavior of programs in a precise and formal way. Finally, abstract interpretation is a technique for analyzing the behavior of programs by approximating their behavior using a simplified or abstract model, which can help to detect errors or improve performance.

Overall, these topics are all related to the goal of making programming more reliable, secure, and efficient by providing formal methods for reasoning about programs. Each topic brings a different set of techniques and tools to the table, and understanding them can help programmers write better code and create more robust software systems.

5.1 Program Analysis

Program analysis is a vital process for ensuring the correctness and reliability of software systems. It involves examining the code of a program to identify potential issues and bugs, as well as to ensure that the program behaves as expected. In the modern world, where software plays an essential role in almost every aspect of our lives, program analysis has become more important than ever before. It helps us detect errors before they cause significant damage, ensures compliance with regulations, and enhances the overall performance of software systems. This article will delve into the various techniques and approaches used in program analysis and their importance in ensuring the quality of software.

There are several types of **program analysis**, which are used to extract information about the behavior of programs.

program analysis
Program analysis is the process of analyzing computer programs to determine their correctness, security, performance, behavior, and other properties.

- Static Analysis: This type of analysis involves examining the source code of a program without actually executing it. It is often used to detect errors, security vulnerabilities, and other types of issues before the program is even compiled or run.
- Dynamic Analysis: This type of analysis involves executing a program with specific inputs and observing its behavior. It is often used to find errors, performance bottlenecks, and other issues that can only be detected at runtime.
- Symbolic Execution: This is a form of static analysis that involves executing a program with symbolic inputs instead of concrete ones.

  It is often used to automatically generate test cases and to find path conditions that may lead to errors.
- Model Checking: This is a type of formal verification that involves checking whether a model of a system meets a set of desired properties.

  It is often used to verify hardware designs and protocols.
- Data Flow Analysis: This is a type of static analysis that involves tracking the flow of data through a program. It is often used to detect security vulnerabilities and to optimize code.

Static analysis involves examining the source code or intermediate representation of a program to detect potential errors or defects. The main goal of static analysis is to identify possible issues that could arise during runtime and to ensure the correctness, reliability, and security of a program.

Static analysis can be performed at different levels of abstraction, including syntax, control flow, data flow, and program semantics. It relies on various techniques such as abstract interpretation, type checking, constraint solving, model checking, and theorem proving. These techniques can be automated to analyze large and complex software systems, and they can also be customized to meet specific requirements. One of the main advantages of static analysis is that it can detect issues early in the development process, before the code is executed, which can save time and resources.

It can also be used to enforce coding standards, improve code quality, and optimize program performance.

The following examples are introducing static analysis on simple programs.

```
int main() {
    int x = 1;
    x = x + 2;
    return x;}
```

Here, the static analysis would perform constant propagation to determine that the value of x at the end of the program is 3, and would also detect that there are no undefined behavior or null pointer dereferences.

```
public class MyClass {
   private int x;

   public void setX(int value) {
      x = value; }

   public int getX() {
      return x; }
}
```

Here, the static analysis would perform data flow analysis to determine that the getX method always returns the value of x, and that the setX method always sets the value of x.

```
def foo(x):
   if x > 0:
      return x + 1
   else:
      return x - 1
```

Here, the static analysis would perform control flow analysis to determine that the function returns either x+1 or x-1 depending on whether x is greater than 0 or not, and that there are no undefined variables or function calls.

On the other hand, dynamic analysis refers to the analysis of a program while it is executing. This method of analysis is used to evaluate the behavior of a program in different situations and identify potential issues.

Dynamic analysis is performed using a variety of techniques, such as code instrumentation, debuggers, and profilers.

One example of dynamic analysis is runtime debugging. This involves running a program in a debugger, which allows the programmer to pause the program's execution at any point, examine the current state of the program, and step through the code one instruction at a time. By observing the program's behavior in real-time, the programmer can identify bugs and issues that may not be apparent from a static analysis of the code. Another example of dynamic analysis is profiling. Profiling involves collecting data on a program's execution, such as how much time is spent on each function call, how often certain code paths are executed, and how much memory is used. This data can be used to identify performance bottlenecks, memory leaks, and other issues that may impact the program's performance. For instance, suppose a developer wants to optimize the performance of a website.

They could use a profiler to collect data on the website's execution and identify which parts of the code are causing slowdowns. By analyzing the profiling data, they could then make changes to the code to improve its performance, such as optimizing frequently executed functions or reducing the amount of memory used by the program.

The **program verification** is the process of formally proving that a program meets its intended behavior and specification. It involves rigorous analysis of the code to ensure that it satisfies certain properties or constraints.

program verification
Program verification is the process of ensuring that a computer program satisfies its specification and behaves correctly for all possible inputs.

The goal of program verification is to increase the reliability and safety of software by detecting and eliminating errors before the program is executed. There are several techniques used in program verification, including formal methods, testing, and model checking. Formal methods use mathematical logic to prove the correctness of a program, while testing involves running the program with different inputs to identify bugs and errors. Model checking involves verifying a finite state machine model of the program to check whether it satisfies certain properties.

Program verification is important in safety-critical applications such as avionics, medical devices, and transportation systems. It is also used in security-critical applications such as cryptographic protocols and secure communication systems. By ensuring that a program is free of bugs and errors, program verification can help to prevent disasters and save lives.

Hoare logic, also known as Floyd-Hoare logic, is a formal system for reasoning about the correctness of computer programs. It was developed by Tony Hoare and Robert Floyd in the late 1960s as a way to formally verify the correctness of computer programs, specifically for imperative programs. The logic is based on a set of axioms and inference rules that allow us to make statements about the behavior of programs. Hoare logic is widely used in program verification and has been instrumental in the development of tools for automatic program verification.

Hoare logic
Hoare logic is a formal system for reasoning about the correctness of computer programs, by specifying preand post-conditions of program fragments and using inference rules to derive the validity of these conditions.

Hoare logic is a formalism based on predicate logic that enables mathematical statements about programs to be formulated and proved. Alternatively interpreted, these rules describe the semantics of the programming language and its constructs. This form of semantics description is referred to as axiomatic semantics. Since this language is Turing-complete, the concepts can at least in principle be transferred to all other programming languages.

The basic building block of Hoare logic is the Hoare triple of the form $\{P\}\mathrm{S}\{Q\}$: If the condition P holds and the program S is executed, then the condition Q holds subsequently. So, P describes a precondition and Q a post-condition of the program S. Logical rules are formulated based on this. A rule $\frac{A}{B}$ means that we can derive that B also holds from A. For the empty program skip, the following rule obviously applies to any condition P:

$$\frac{}{\{P\}\mathrm{skip}\{P\}}$$

The meaning of an assignment $\mathrm{x{:=}E}$ is described by the rule

$$\frac{}{\{P[E/x]\}\mathrm{x{:=}E}\{P\}}$$

where $P[E/x]$ describes the condition P when every free occurrence of x has been replaced by E. For example, if we execute the command $\mathrm{x{:=}y{+}1}$ and want to ensure that x has the value 5 afterwards, we must ensure that $y + 1$ has the value 5 beforehand, i.e., $\{y + 1 = 5\}\mathrm{x{:=}y{+}1}\{x = 5\}$.

As in this example, in practical applications, we usually assume which property should be fulfilled in the end and derive the property required at the beginning from it.

For the composition of instructions, the corresponding rule requires two assumptions, unlike the two previous rules:

$$\frac{\{P\}\mathrm{S1}\{Q\} \quad \{Q\}\mathrm{S2}\{R\}}{\{P\}\mathrm{S1;S2}\{R\}}$$

To be able to apply these and the other rules, occasionally we need to adjust a pre-condition or post-condition as follows:

$$\frac{P_1 \Rightarrow P_2 \quad \{P2\}\mathrm{S}\{Q_1\} \quad Q_1 \Rightarrow Q_2}{\{P_1\}\mathrm{S}\{Q_2\}}$$

The meaning of an IF statement is described by the following rule: {B ∧ P}S{Q} {¬B ∧ P}T{Q} {P}if B then S else T endif{Q}

$$\frac{\{B \wedge P\}S\{Q\} \qquad \{\neg B \wedge P\}T\{Q\}}{\{P\}\text{if B then S else T endif}\{Q\}}$$

For giving only the general idea of Hoare logic, we skip, the remaining rules. It is here more important to know for what it is used and what is the purpose and general procedure.

Finally, there are various tools for program analysis that are used to verify the correctness of software, detect bugs and vulnerabilities, optimize performance, and more. Some common examples of program analysis tools include: • Static analyzers: These are tools that analyze the source code of a program without actually executing it. They can detect potential issues such as buffer overflows, null pointer dereferences, and race conditions by examining the program's control flow and data flow.

- Dynamic analyzers: Unlike static analyzers, these tools execute the program and observe its behavior at runtime. They can detect issues such as memory leaks, resource leaks, and incorrect locking by monitoring the program's memory usage, input/output behavior, and more.
- Profilers: These tools measure the performance of a program by monitoring its execution time, memory usage, and other metrics. They can help identify bottlenecks and hotspots in the code that can be optimized for better performance.
- Model checkers: These tools verify that a program meets a set of formal specifications or requirements. They can detect issues such as deadlocks, livelocks, and violation of safety properties.
- Fuzzers: These tools generate random or mutated inputs to a program to test its robustness and resilience against unexpected inputs.
- Decompilers: These tools can generate high-level code from compiled binaries or machine code, allowing developers to analyze and understand the behavior of third-party libraries or legacy code.

Overall, program analysis tools can be an essential part of the software development process, helping to ensure the correctness, performance, and security of software.

5.2 Algebraic, Operational and Denotational Semantics

Syntax and semantics are two essential building blocks for describing programming languages. Syntax deals with which character strings are valid sentences (programs) of the language. Syntax includes vocabulary (words) and grammar. Semantics describes what the meaning of a valid sentence (program) should be. For programming languages, this means: What is the behavior of the program when executed? Syntax also defines the structure of a sentence, usually a syntax tree, and explains how to get from a character string to the syntax tree. A semantics describes how to give meaning to this syntactic structure, i.e.: What is the meaning of each construct? How to derive the overall meaning from the individual parts? The syntax and semantics of many programming languages are standardized (C, C++, Java, etc.). For defining the syntax, formal techniques are routinely used in practice: context-free grammars. However, most of these standards describe the behavior of language constructs and their interactions only in natural language, mostly in English, often only using concrete examples. For extensive programming languages, it is almost impossible to define all possible combinations unambiguously and still guarantee consistency in this way. Therefore, there are formal, i.e. mathematical, description techniques for semantics, which are the topic here.

In simple words, "the semantics of a programming language describes the relationship between the syntax and the model of computation." The formal semantics of a programming language is the handbook for the engineer who has to understand not only how to use the language but how it works.

It provides a rigorous specification of how the programs are executed, which reveals the ambiguities and subtleties hidden behind the language used in the programming manuals, but which usually appear in practice.

Semantics is concerned with the interpretation or understanding of programs and how to predict the outcome of program execution. The semantics of a programming language describes the relationship between the syntax and the model of computation. Semantics can be thought of as a function that maps syntactical constructs to the computational model (syntax $\mapsto$ computational model).

There are several widely used techniques (algebraic, operational and denotational) for the description of the semantics of programming languages.

The key purpose of algebraic, operational, and denotational semantics is to provide a mathematical framework for understanding the meaning of computer programs. A semantic description in prose, like in most language standards, may be easier, but it cannot prevent ambiguities, misunderstandings, or contradictions. In contrast, there are advantages to mathematical descriptions of semantics. Let's take a look at the different approaches using the example program $\mathrm{z} := \mathrm{x};\ \mathrm{x} := \mathrm{y};\ \mathrm{y} := \mathrm{z}$.

Algebraic semantics
Algebraic semantics is a formal method for defining the meaning of programming language constructs using algebraic equations.

Algebraic semantics is concerned with representing the meaning of a program in terms of algebraic equations. It is a formal method that defines the meaning of programming language constructs using mathematical equations or axioms. It involves the use of algebraic structures such as sets, functions, and equations to describe the behavior of programming language constructs. The idea is to break down the program into smaller components and then define equations that describe how those components relate to one another. For example, the meaning of a program might be defined in terms of the meanings of its individual statements, with equations that describe how those statements combine to produce the overall behavior of the program.

Operational semantics
Operational semantics defines the meaning of a program by specifying how its execution proceeds on a machine.

Operational semantics, on the other hand, is concerned with defining the meaning of a program in terms of its execution on a computer. It involves the use of operational rules or transition rules that describe the behavior of a program step-by-step. The idea is to define a set of rules that describe how the program behaves as it runs. For example, a rule might be defined that describes how a particular statement modifies the state of the program. Operational semantics can be used to specify the dynamic behavior of programming languages and can help in analyzing and verifying the correctness of programs.

Operational semantics describes the semantics of $\mathrm{z} := \mathrm{x};\ \mathrm{x} := \mathrm{y};\ \mathrm{y} := \mathrm{z}$ by defining how the program should be executed: A sequence of two statements separated by ';' executes the individual statements one after the other. An assignment of the form $Variable\!:\, = Expression$ first evaluates the expression to a value and then assigns that value to the variable. For a state $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$, which assigns the values 5, 7, and 0 to variables x, y, and z, the following evaluation sequence results:

$$\begin{array}{rl}
\langle z\colon = x; x\colon = y; y\colon = z, & [x \mapsto 5, y \mapsto 7, z \mapsto 0]\rangle \\
\to \langle x\colon = y; y\colon = z, & [x \mapsto 5, y \mapsto 7, z \mapsto 5]\rangle \\
\to \langle y\colon = z, & [x \mapsto 7, y \mapsto 7, z \mapsto 5]\rangle \\
\xrightarrow{\hspace{6cm}} & [x \mapsto 7, y \mapsto 5, z \mapsto 5]
\end{array}$$

Denotational semantics takes a more abstract approach to defining the meaning of a program. It is a formal method that defines the meaning of programming language constructs in terms of mathematical objects or functions. It involves the use of mathematical functions called denotations that map programs to mathematical objects, such as sets or functions. It is concerned with defining the meaning of a program in terms of its effect on mathematical objects or structures. For example, the meaning of a program might be defined in terms of the function it computes, or in terms of a set of mathematical relations that hold between the program's input and output. Denotational semantics can be used to specify the semantics of programming languages, and can help in analyzing the behavior and correctness of programs.

Denotational semantics
Denotational semantics is a formal approach to defining the meaning of programming language constructs in terms of mathematical objects and functions.

Denotational semantics only concerns itself with the effect of execution, not the individual computational steps. Accordingly, the semantics of such a program is a function that maps an initial state to a final state. For a sequence, the function is obtained by composing (performing in sequence) the functions of the two statements. The meaning of an assignment is the function that changes the passed state so that the variable is assigned the value of the expression. This results for $z \quad := \quad x; \quad x \quad := \quad y; \quad y \quad := \quad z$ in:

$$\begin{array}{l}
\mathcal{D}\backslash\mathrm{lsem}z\colon = x; x\colon = y; y\colon = z\backslash\mathrm{rsem}(\sigma) = (\mathcal{D}\backslash\mathrm{lsem}y\colon = z\backslash\mathrm{rsem} \circ \mathcal{D} \\
\backslash\mathrm{lsem}x\colon = y\backslash\mathrm{rsem} \circ \mathcal{D}\backslash\mathrm{lsem}z\colon = x\backslash\mathrm{rsem})(\sigma) \\
= \mathcal{D}\backslash\mathrm{lsem}y\colon = z\backslash\mathrm{rsem}(\mathcal{D}\backslash\mathrm{lsem}x\colon = y\backslash\mathrm{rsem}(\mathcal{D}\backslash\mathrm{lsem}z\colon = x\backslash\mathrm{rsem}(\sigma))) \\
= \mathcal{D}\backslash\mathrm{lsem}y\colon = z\backslash\mathrm{rsem}(\mathcal{D}\backslash\mathrm{lsem}x\colon = y\backslash\mathrm{rsem}(\sigma[z \mapsto \sigma(x)])) \\
= \mathcal{D}\backslash\mathrm{lsem}y\colon = z\backslash\mathrm{rsem}(\sigma[z \mapsto \sigma(x), x \mapsto \sigma[z \mapsto \sigma(x)](y)) \\
= \mathcal{D}\backslash\mathrm{lsem}y\colon = z\backslash\mathrm{rsem}(\sigma[z \mapsto \sigma(x), x \mapsto \sigma(y)]) \\
= \sigma[z \mapsto \sigma(x), x \mapsto \sigma(y), y \mapsto \sigma[z \mapsto \sigma(x), y \mapsto \sigma(y)](z)] \\
= \sigma[x \mapsto \sigma(y), y \mapsto \sigma(x), z \mapsto \sigma(x) \\
]
\end{array}$$

This yields again for $\sigma = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$:

$$\mathcal{D}\backslash\mathrm{lsem}z\colon = x; x\colon = y; y\colon = z\backslash\mathrm{rsem}(\sigma) = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$$

The main difference between these three approaches is the level of abstraction at which they operate. Algebraic semantics is concerned with the details of the program's structure, while operational semantics focuses on the program's execution. Denotational semantics takes a more abstract view, defining the program's meaning in terms of mathematical objects or structures. Despite these differences, all three approaches share the goal of providing a rigorous mathematical framework for understanding the meaning of computer programs. In summary, the difference between these three approaches lies in the methods used to describe the meaning of programming language constructs. Algebraic semantics uses mathematical equations or axioms, operational semantics uses operational rules, and denotational semantics uses mathematical functions called denotations.

5.3 Abstract Interpretation

Abstract interpretation is a technique for formally verifying and analyzing the behavior of computer programs. Its main goal is to establish the correctness of programs by systematically over-approximating their possible behaviors. It achieves this by modeling program executions as mathematical abstractions that capture the essential features of the program's behavior, while ignoring details that are irrelevant to its correctness.

Abstract interpretation is particularly useful in cases where formal verification of programs directly at the level of source code is infeasible or impractical, due to the complexity of the program, the limited resources available, or the lack of a precise specification. In such cases, it provides a systematic and scalable approach to program analysis that can be used to detect errors, find bugs, and ensure program correctness.

The key advantage of abstract interpretation is that it can analyze programs at a high level of abstraction, without having to deal with the lowlevel details of the program implementation. This makes it possible to analyze large and complex programs more efficiently than other techniques, such as model checking or theorem proving. It also makes it possible to analyze programs that are written in languages that are difficult to formalize or that do not have precise semantics.

In summary, abstract interpretation is a powerful technique for verifying and analyzing the behavior of programs. Its main goals are to ensure program correctness and to provide insights into the program's behavior.

It is particularly useful in cases where formal verification is infeasible or impractical, and provides a systematic and scalable approach to program analysis.

abstract domains
An abstract domain is a mathematical abstraction used in abstract interpretation to represent a set of concrete program states.

One of the key components of abstract interpretation is the use of **abstract domains**. An abstract domain is a set of abstract values that can be used to approximate the possible values that a variable can take during the execution of a program. The choice of an appropriate abstract domain depends on the properties of the program being analyzed and the type of analysis being performed.

One example of an abstract domain is the interval domain, which represents sets of integers as closed intervals, and can be used to analyze properties of integer variables in a program. Here is an example code snippet that could be analyzed using the interval domain:

```
int x = 5;
int y = 7;
if (x < y) {
  x = y + 1;
}
return x;
```

Using the interval domain, we can represent the values of x and y as intervals and track how they change throughout the program. For example, after the first two lines, we might represent the values as $x \in [5,5], y \in [7,7]$.

After the conditional statement, we might represent them as $x \in [8, + \infty], y \in [7,7]$. This tells us that x has been updated to be greater than y by at least one, and y has not changed. Let's take a look at the approach of abstract semantics after choosing the abstract domain of integers. Before, the abstract interpretation framework is introduced.

The abstract interpretation framework provides a systematic way of designing and implementing abstract domains for different program analysis tasks. It consists of four main components:

- Abstract Domain: This is the set of abstract values and operations that can be used to reason about the behavior of a program. An abstract domain is defined by a set of **abstract operators** that are used to perform computations on the abstract values.
- Abstraction Function: This function maps concrete program states to abstract states in the chosen abstract domain. The abstraction function defines the level of abstraction at which the program is being analyzed.
- Galois Connection: This is a mathematical concept that defines the relationship between the concrete domain and the abstract domain.
  It provides a way of comparing the precision of two abstract domains.
- Widening Operator: This operator is used to accelerate the convergence of the abstract interpretation process. It is used to merge the abstract values obtained during different iterations of the analysis to obtain a more precise result.

abstract operators
Abstract operators are mathematical functions that operate on abstract values.

The use of abstract domains and the abstract interpretation framework allows us to reason about complex programs and properties that are difficult to verify using traditional testing methods.

To apply abstract semantics to the above code snippet, we need to first choose an abstract domain to work with. Let's choose the interval domain, where each variable is represented by a range of possible values. Now let's consider an abstract input where x is in the interval $x \in [2,6]$ and y is in the interval $y \in [7,10]$. This means that the concrete values of x and y could be any values in the given ranges.

Next, we apply the abstract interpretation rules to obtain an abstract output. In this case, we can see that x is less than y according to the abstract input, since $[2, \quad 6]$ is less than $[7, \quad 10]$. Therefore, we can abstractly execute the if statement and set x to the interval $x \in [8,11]$ $(\mathrm{y} + 1)$ since the condition is true. If the condition were false, we would set x to the interval $x \in [2,6]$ (its original value).

So the abstract output for this code snippet, given the input described above, is that x is in the interval $x \in [8,11]$.

A more general description of the abstract semantics of the above code snippet: Firstly, we define the abstract domain and abstract operations that approximate the concrete behavior of the code:

- set the intervals of x and y in the abstract domain.
- $\mathrm{x := y + 1}$: set the interval of x to be $\left[l_y + 1, u_y + 1\right]$, where $\left[l_y, u_y\right]$ is the interval of y.
- if $\mathrm{if(x<y)\{s1\}else\{s2\}}$: if the interval of x is less than the interval of y, then compute the abstract semantics of s1 (here: $\mathrm{x=y+1;;}$); otherwise, compute the abstract semantics of s2 (here: skip).
- The abstract output is $x \in \left[l_y + 1, u_y + 1\right]$ when if is true, otherwise $x \in [l_x, u_x]$(its original value).

Using this abstract domain and operations, we can compute the abstract semantics of the code as follows:

- Let $x = 5$ and $y = 7$ be the inputs.
- Set the intervals of x to be $x \in [2, 6]$ and y to be $y \in [7, 10]$.
- Check if $x < y$: the abstract semantics of this comparison are true for all possible inputs.
- After abstractly executing the if statement $\mathrm{(s1)}$, the interval $x \in [8, 11]$ because of $y + 1$.
- The abstract output is $x \in [8, 11]$.

Therefore, the abstract semantics of the code for the given input is $x \in [8, 11]$.

The abstract output obtained from abstract interpretation can be used to reason about the behavior of the function in a more general and efficient way than concrete interpretation. By analyzing the abstract output, we can determine if the function is free from certain types of errors such as buffer overflow, null pointer dereference, etc. It also helps in identifying unreachable code, redundant code, and dead code. Additionally, it can help to optimize the code by eliminating unnecessary computations or data.

Abstract interpretation can also provide valuable insights into the performance characteristics of the function, such as upper bounds on running time and memory usage. Therefore, the abstract output can be used to improve the quality and reliability of the software, reduce the cost of testing and debugging, and enhance the overall performance of the program.

Transferred to the example, the abstract output can be used to reason about the behavior of the code snippet as follows. Since the abstract output tells us that the return value is x (8), and we can conclude that the function always returns a value that is one more than y (7). Thus, we can say that the function computes the y+1.

In abstract interpretation, soundness and completeness are two important properties that determine the effectiveness and correctness of the analysis.

Soundness means that the analysis never produces a false positive, i.e., it never reports a property that does not actually hold in the program. In other words, the analysis conservatively approximates the behavior of the program, ensuring that all reported properties are true. If an analysis is sound, then any error found by the analysis is a true error in the program.

Completeness, on the other hand, means that the analysis never produces a false negative, i.e., it never misses a property that actually holds in the program. In other words, the analysis is able to accurately capture all possible program behaviors, ensuring that all true properties are reported.

If an analysis is complete, then it is guaranteed to find every error in the program.

Ideally, we want an abstract interpretation to be both sound and complete.

However, in practice, achieving both properties at the same time can be difficult, as there is often a trade-off between precision and efficiency. A more precise analysis may be less efficient and may take longer to run, while a less precise analysis may be faster but may produce more false positives or false negatives.

Therefore, in practice, we often need to choose between soundness and completeness, depending on the specific requirements and constraints of the analysis. For example, for safety-critical systems, it is often more important to prioritize soundness and ensure that all reported properties are true, even if this means sacrificing some completeness. On the other hand, for other types of systems, completeness may be more important, and it may be acceptable to have some false positives if it means finding all true properties.

SUMMARY

Program Analysis deals with techniques for analyzing programs to check for correctness, identify potential bugs or vulnerabilities, and optimize performance. The goal is to provide automated tools and methods that can help programmers verify that their code works as intended and meets certain requirements.

Algebraic, Operational and Denotational Semantics are three different approaches to defining the meaning of programming languages.

Algebraic semantics uses mathematical structures such as algebraic equations to define the behavior of a language. Operational semantics focuses on the step-by-step execution of a program and defines its meaning in terms of the states it transitions through.

Denotational semantics defines the meaning of a program in terms of its effect on abstract mathematical objects.

Abstract Interpretation is a formal method for analyzing programs that aims to approximate their behavior without necessarily executing them. The goal is to reason about the behavior of programs in a way that is both sound (i.e., guarantees that the analysis is correct) and efficient (i.e., scales well to large programs).

UNIT 6

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the concepts of machine learning
- understand the difference between supervised and unsupervised learning
- apply linear and non-linear as well as logistic regression
- understand the structure and concepts of artificial neural networks

6. ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Introduction/Case study

Machine Learning (ML) grew out of Artificial Intelligence and accomplishes new capabilities for computers. An important field is using ML in applications that can not be programmed by hand (e.g. autonomous driving, handwriting recognition, natural language processing and computer vision).

Also self-customized programs, e.g. Amazon and Netflix product recommendations, are application scenarios of ML.

ML is used in several fields of daily live. An example is the automatic recognition of identities at airports. The passport is not needed anymore and also the identity is not checked by humans. Face recognition using a camera in conjunction with ML fulfills this task. In other scenarios are used large data from e.g. medical records, engineering and web traffic.

Thus, an important question is "What is Machine Learning?". Tom Mitchell defines ML as:

DEFINITION: MACHINE LEARNING

A computer program is said to learn from experience $\mathcal{E}$ with respect to some task $\mathcal{T}$ and some performance measure $\mathcal{P}$, if its performance on $\mathcal{T}$, as measured by $\mathcal{P}$, improves with experience $\mathcal{E}$. (Mitchell, 1997)

Let's take a look at a concrete scenario from IT Security. Think about your E-Mail program that monitors which mails you mark or not mark as SPAM, and based on that learns to filter SPAM. In this example, the classification of mails (SPAM or not) is task $\mathcal{T}$, the monitoring of labelling mails as SPAM or not is experience $\mathcal{E}$ and the amount of correctly classified mails is performance $\mathcal{P}$.

An additional example in the area of IT Security is the classification of malware or no malware. Malware is spread over computers, network devices and distributed via different types of networks. The detection and prevention of malware is difficult in all mentioned areas. Nowadays, the classification by hand with around 300.000 new variants per day is almost impossible. In conjunction with the steadily increasing number of attacks and the mentioned variants of malware (polymorphism) approaches like signature-based antivirus programs can not find unknown/new variants.

ML provides behavior-based approaches to identify also new variants. An introduction and the development of this example is described in the next section. Before the deep dive into the topic, an overview of the sections is given.

Outline
Section 6.1 gives an overview about the difference between supervised and unsupervised learning. The terms regression and classification are also explained. Section 6.2 introduces linear and non-linear regression with one and multiple variables. Terms and topics like gradient descent, parameters, hypothesis and cost function are part of that section. In section 6.3, the topic classification respectively logistic regression is focused. Additionally, how to deal with two classes versus multiple classes. Unit 6 concludes with addressing neural networks (cf. section 6.4). Non-linear hypothesis are described and solutions using neural networks are explained.

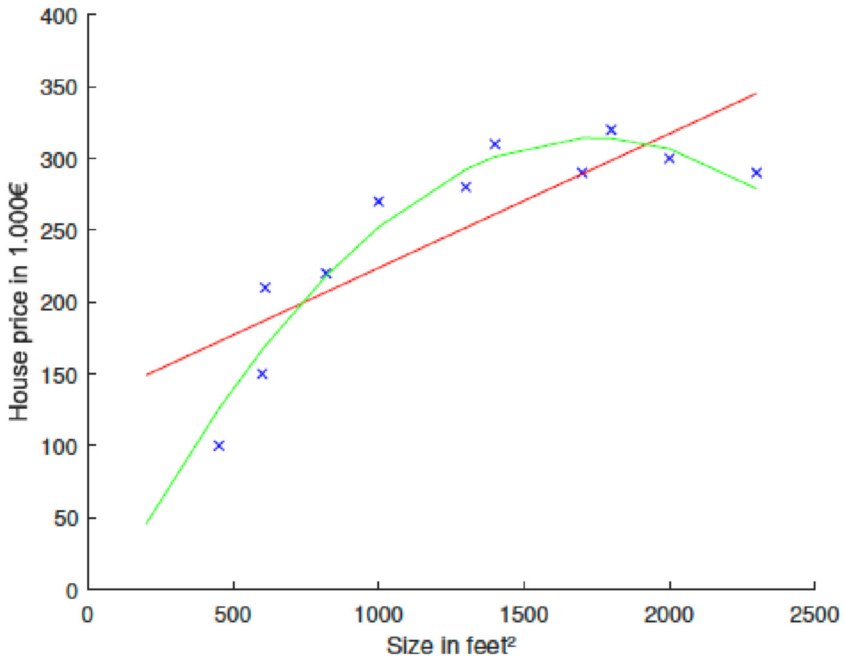
6.1 Supervised vs. Unsupervised Learning

Supervised learning can be summarized as "right answers are given". This means that for given data is already known what the correct output should look like. Thus, a relation exists between the input and the output. Furthermore, supervised learning problems are distinguished between regression and classification. Regression deals with predicting results in a continuous output. Thus, input variables are mapped to a continuous function.

Classification instead deals with predicting results in a discrete output.

Here input variables are mapped to discrete categories.

Figure 29: Figure 6.1: Regression - House Price Prediction ((Lawall, 2021) based on ANDREW)

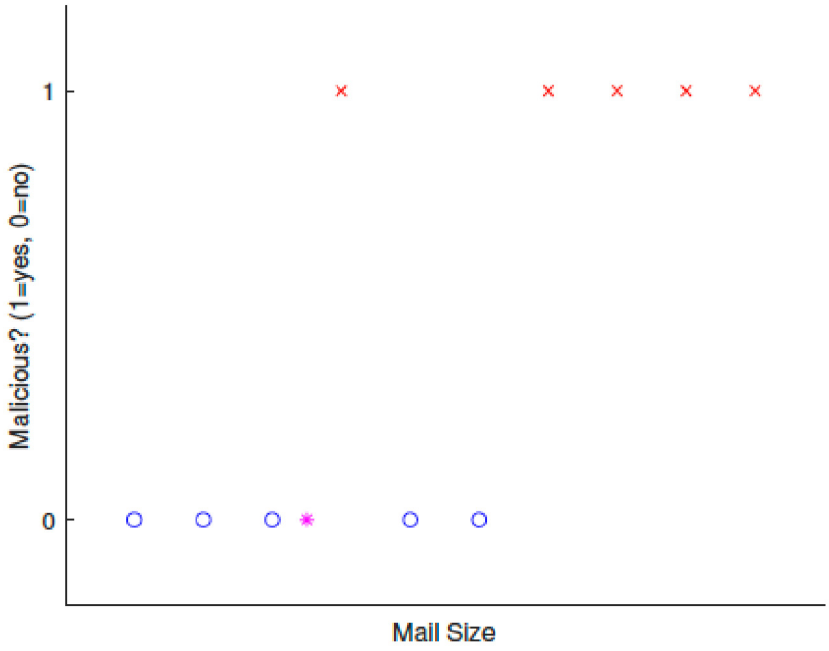


The difference between regression and classification will be illustrated by examples for predicting house prices and malicious mails. A simplified scenario for regression describes the connection between the size of houses and the house prices. Figure 6.1 illustrates collected data for different sizes and resulting prices. The abscissa represents the house sizes in square feet and the ordinate the house prices in thousand euro. Thus, a single data point (x) displays a house price for a given house size. The price as function of size is a continuous output. Thus, the problem belongs to regression.

regression
Regression predicts continuous valued outputs and has the goal to fit data points.

For the prediction of continuous valued outputs (here price), a **regression** Regression function has to be found. This function allows the prediction of prices from arbitrary sizes. The linear regression function is not nicely fitting the data rather the non-linear regression function. Linear means a straight line and non-linear "curvy" lines. In general, the regression function can be calculated by using the method of least squares with given kind of function (linear or non-linear). In this example, for the size 1300 square feet the linear regression function predicts price 252.000 euros and the non-linear regression function 292.000 euros.

Figure 30: Figure 6.2: Classification - Malware in Mail Prediction (One Feature)



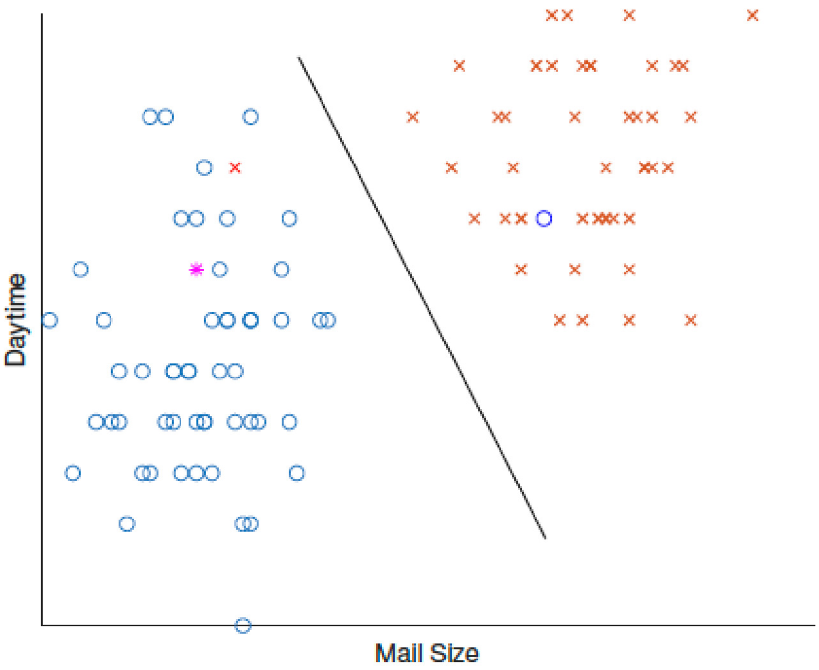
After regression, the **classification** will be introduced. Therefore, a simplified scenario for classifying mails by size in malicious or not malicious is used. Figure 6.2 shows a data set referring to mails that are malicious (x) or not malicious (o). In this example these two classes are used. Generally, classification is not limited to two classes but for simplification two are appropriate for now. In this scenario regression not works because the output values are discrete (0 or 1). Thus, classification is used for predicting the output. For a given mail size (*) the prediction results in malicious or not.

classification
Classification predicts discrete valued outputs and has the goal to separate data points.

In that case the correct prediction is not easily recognizable. Thus, more **features** (e.g. daytime) are used to classify the mail in malicious or not, cf. figure 6.3.

features
Feature is a measurable property or observed characteristic.

Figure 31: Figure 6.3: Classification - Malware in Mail Prediction (Two Feature)



In figure 6.3, the black line separates data of classes not malicious (lefthand side) with data points o from malicious (right-hand side) mails x.

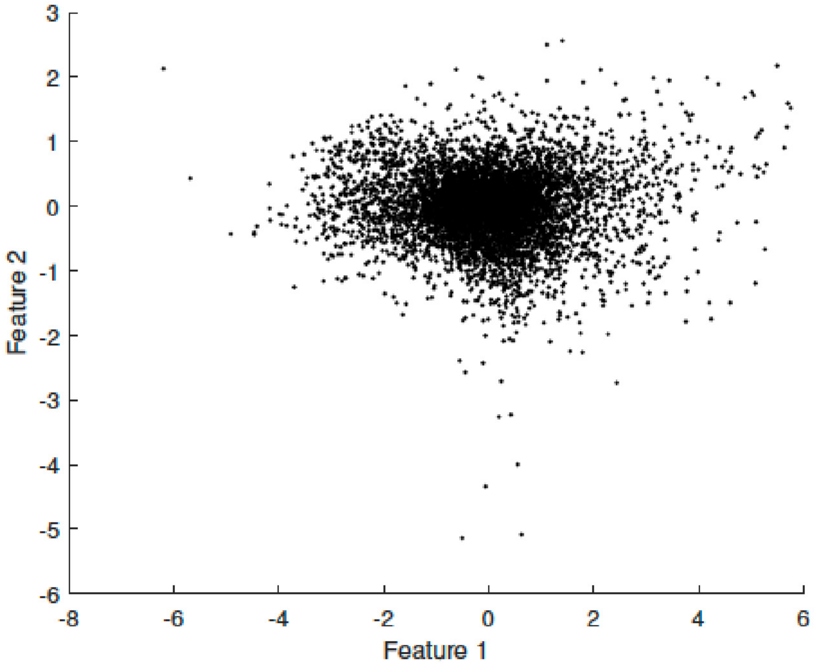
The outliers at both classes are “accepted” in linear classification. Linear means it is separated by a straight line. In the example, a new mail with specific size and daytime (*) is predicted as not malicious because the data point is in this mentioned class. Similar to the regression, in classification are also used non-linear classifiers.

labeled
Labeling of data gives the data informative tags.

In summary, it can be stated that regression predicts continuous valued outputs from one or more features. The goal for regression is to fit the data points. In contrast, classification predicts discrete valued outputs (“classes”) from one or more features and tries to separate the data points, different classes, as much as possible. For both - regression and classification - the prediction function can be linear or non-linear. Additionally, supervised learning deals with **labeled** data. This means that for given data the correct answers are known. For example, a mail is classified by human if it is malicious or not. This information is used for building the prediction function.

In unsupervised learning, the given data is not labeled (“right answers are not given”) and no structure is available. Therefore, methods like cluster analysis are used. In cluster analysis given data will be grouped by commonalities. Thus, the structure can be derived by clustering the data points by their relations. An example is the network traffic monitored at the mirror port of a switch. The grouping into sets of network traffic can be used to find out anomalies e.g. communication with an attackers Command and Control Server, abnormal routes resp. computers or protocols.

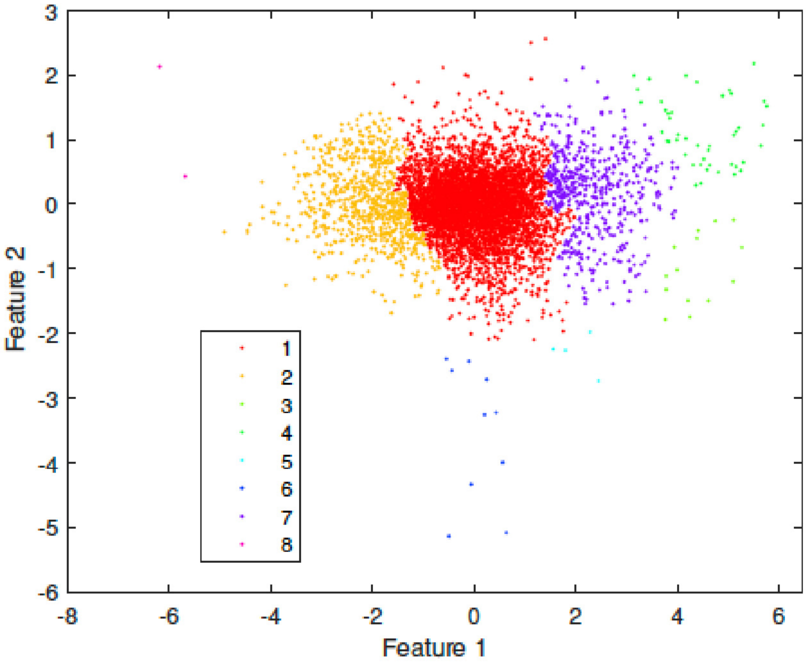
Figure 32: Figure 6.4: Unsupervised Learning - Data Points



Figures 6.4 and 6.5 illustrate the general concept of unsupervised learning.

The collected data points (.) in conjunction with two features are represented in figure 6.4. There are given no labels for the output values and no structure, only the raw data is plotted. After using a cluster algorithm, the data points are structured into clusters. In figure 6.5 are represented eight clusters in different colors (cf. legend 1 to 8). For example, cluster 8 (top left) represents communication to Command and Control Servers, cluster 3 (right side) the usage of abnormal protocols and cluster 5 (light blue) communication to abnormal computers.

Figure 33: Figure 6.5: Unsupervised Learning - Cluster Analysis



In summary, it can be stated that unsupervised learning gives the possibility to derive structure from unlabeled data. The structure can be accomplished e.g. by cluster analysis using relations of features of the data points. The resulting groups can be analyzed to know what is represented by a specific group.

6.2 Linear and non-linear Regression

At the beginning the univariate linear regression/regression with one variable (feature) will be explained. Therefore, the known regression problem house prices is reused. The data points in figure 6.1 can be also represented in table format, cf. table 6.1.

The table is used to establish notation for developing the model representation.

Table 7: Table 6.1: Training Set of House Prices

Size in $\text{feet}^2 (x)$	Price in k euros (y)
450	100
600	150
610	210

820	220
...	...

The variable x denotes the input variable/feature, y is the output variable and m the size of the training set of data. Thus, a tuple $\left(x^{(i)}, y^{(i)}\right)$ is called i-th training sample out of the training data set with m samples.

Hint: (i) is no exponentiation, it is only an index. The X and Y denotes the set of input and output variables with $X = Y = \mathbb{R}$.

The formal description for supervised learning problems is

$$h: X \rightarrow Y$$

with the goal to learn a function $h(x)$ with given training data to predict as good as possible the output value y. The function $h(x)$ is therefore called the **hypothesis**. In linear regression, the hypothesis is a straight line

hypothesis
Hypothesis in general is used for predict values or separate classes.

$$h_\theta(x) = \theta_0 + \theta_1 \cdot x$$

with input variable x and parameters θ_0, θ_1. The parameters determine the line with the gradient θ_1 and the y-axis intercept θ_0.

The question is how to choose these parameters to get the best results.

Therefore, the idea is to choose θ_0, θ_1 such that the hypothesis $h_\theta(x)$ is close to y for all training samples $\left(x^i, y^i\right)$ with $i \quad = \quad 1, \ . \ . \ . \ , m$. The accuracy of the hypothesis can be measured by using the **cost function** $J(\theta_0, \theta_1)$.

cost function
Cost function gives an overall difference value comparing the hypothesis with inputs x and actual outputs y.

This is calculated by using the mean squared deviations. The cost function is defined as

$$J\left(\theta_0, \theta_1\right) = \frac{1}{2m} \sum_{i=1}^{m} \left(x^i - y^i\right)^2$$

with the sum over the squared errors between the predicted value $h_\theta\left(x^{(i)}\right)$ and the actual value $y^{(i)}$.

The overall goal is to minimize the cost function so that the hypothesis is as close as possible to the data points (cf. fig. 6.1, linear regression function). Thus, the goal can be formalized with min

$$\min_{\theta_0, \theta_1} J\left(\theta_0, \theta_1\right)$$

to calculate the minimum of the cost function and so the values for parameters θ0, θ1.

Table 8: Table 6.2: Training Set of House Prices

Size in $feet^2(x)$	Price in k euros (y)
$x^{(1)} = 450$	$y^{(1)} = 100$
$x^{(2)} = 600$	$y^{(2)} = 150$
$x^{(3)} = 610$	$y^{(3)} = 210$
$x^{(4)} = 820$	$y^{(4)} = 220$

Table 6.2 helps to explain the mathematics behind that. For simplification, only four data points of table 6.1 are used and the hypothesis is $h_\theta(x) = \theta_1 \cdot x$. This means that the hypothesis starts in the center point of the coordinate system, here $(0, 0)$. Thus, the axis intercept $\theta_0 = 0$ and the goal is simplified to $\underset{\theta_1}{\min} J\left(\theta_1\right)$

with

$$J\left(\theta_1\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2m}\sum_{i=1}^{m}\left(\theta_1 \cdot x^i - y^i\right)^2$$

The number of training samples $\left(x^{(i)}, y^{(i)}\right)$ equals $m = 4$. To calculate the cost function $J(\theta_1)$, the parameter is fixed to $\theta_1 = 1$ for the first try.

$$J\left(\theta_1\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{8}\left(350^2 + 450^2 + 400^2 + 600^2\right)$$
$$= 105.625$$

In second try, the parameter is fixed to $\theta_1 = 2$.

$$J\left(\theta_1\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(2 \cdot x^{(i)}\right) - y^{(i)}\Big)^2 = \frac{1}{8}\Big(800^2 + 1050^2 + 1010^2 + 1420^2$$
$$\Big) = 4.779.000$$

Thus, the hypothesis with $\theta_1 = 1$ has more accuracy in reference to the training data because the cost value is less than with $\theta_1 = 2$. If the line would fit to all given training data, the value of the cost function is 0, meaning there is no difference between the hypothesis and the actual output values y. That is not every time possible with linear hypothesis.

For finding the best fitting hypothesis, the parameters can be calculated by minimizing the cost function $J(\theta_0, \theta_1)$. Before starting with the minimizing step, the notation so far is summarized in table 6.3.

Table 9: Table 6.3: Summary of Univariate Linear Regression Model

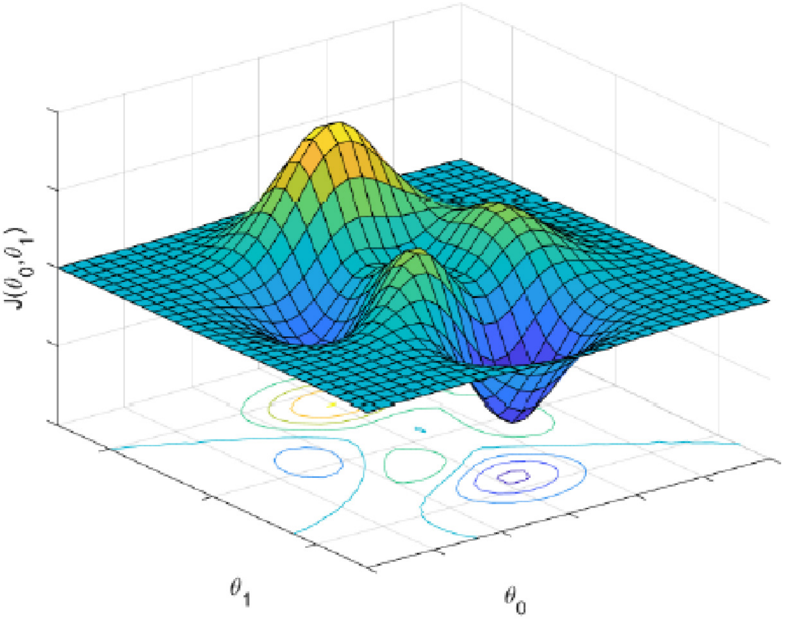
Hypothesis:	$h_\theta(x) = \theta_0 + \theta_1 \cdot x$
Parameters:	θ_0, θ_1
Cost function:	$J\left(\theta_0, \theta_1\right) = \frac{1}{2m}\Sigma_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$
Goal:	$\min_{\theta_0, \theta_1} J\left(\theta_0, \theta_1\right)$

Lets dive into methods for minimizing the cost function $J(\theta_0, \theta_1)$. The general process with **gradient descent** is to start with some random values for θ_0 and θ_1 and keep changing them to reduce the cost function $J(\theta_0, \theta_1)$ til ending up at a minimum. The found minimum can be the global as well as a local minimum.

gradient descent Gradient Descent Algorithm searches a minimum by stepwise descent the curve in derivative direction.

Figure 6.6 shows a cost function $J(\theta_0, \theta_1)$ in conjunction with the contour plot showing different altitude. Depending on the starting assignment of θ_0 and θ_1, the algorithm can end up in different minima. For example, two start configurations are located at the hill (orange area) near to each other. The more left point ends up in the local minimum (blue area at the left) whereas the more right point ends up in the global minimum (dark blue area at the right). Of course, the global minimum with determined θ_0, θ_1 has the lowest cost value and thereby the higher accuracy for the hypothesis.

Figure 34: Figure 6.6: Gradient Descent for Cost Function $J(\theta_0, \theta_1)$



The gradient descent algorithm consists of the partial derivative of the cost function and a learning rate. The algorithm is defined as

$$\text{repeat until convergence } \{\ \theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J\left(\theta_0, \theta_1\right), \text{with } j = 0, 1\ \}$$

with a learning rate α (“step length for descent”). If α is too small, the algorithm can be slow because of only small steps. If α is too large, the algorithm can fail to converge or even diverge (“overshooting”).

Lets come back to the example depicted in table 6.2. The hypothesis is $h_\theta(x) = \theta_0 + \theta_1 \cdot x$

and the cost function $J\left(\theta_0, \theta_1\right) = \frac{1}{2m} \sum_{i=1}^{m} \left(x^{(i)} - y^{(i)}\right)^2$.

The partial derivatives of the cost function are

$$\frac{\partial}{\partial \theta_j} J\left(\theta_0, \quad \theta_1\right) = \frac{\partial}{\partial \theta_j} \frac{1}{2m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 \cdot \left(x^{(i)}\right) - y^{(i)}\right)^2$$

Thus, the partial derivative for θ_0 $(j = 0)$ is

$$\frac{\partial}{\partial \theta_0} J\left(\theta_0, \theta_1\right) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

and for θ_1 $(j = 1)$

$$\frac{\partial}{\partial \theta_1} J\left(\theta_0, \theta_1\right) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x^{(i)}$$

Thus, the algorithm calculates the minimum by simultaneously updating θ_0 and θ_1

repeat until convergence {

$$\begin{aligned} \theta_0 &= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \\ \theta_1 &= \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x^{(i)} \end{aligned}$$

}

The resulting values for parameter θ_0 and θ_1 are used in the hypothesis $h_\theta(x)$to fit the data points of the training data.

Multivariate Linear Regression

Multivariate regression uses multiple features instead of one feature (x).

Multivariate regression
Multivariate Regression is a regression with two or more features.

The mentioned example (house prices) is extended as shown in table 6.4.

More features like number of bedrooms (x_2) and floors (x_3) are included.

Furthermore, the following notation is introduced: the number of features is $n = 3$, the input of the i-th training sample is $x^{(i)}$ and the value of feature j in the i-th training sample is $x_j^{(i)}$. For example, $x^{(2)}$ is the n-dimensional

vector $x^{(2)} = \begin{pmatrix} 600 \\ 4 \\ 1 \end{pmatrix} \text{and} x_2^{(2)} = 4\,.$

Table 23

Size in $\text{feet}^2(x_1)$	Bedrooms (x_2)	Floors (x_3)	Price in k euros (y)
450	2	1	100
600	4	1	150
610	2	2	210
820	3	2	220
...	...	...	...

Source: Table 6.4: Training Set of House Prices - Multiple Features

The hypothesis in this example is that $h_\theta(x) \quad = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_3$.

A trick is used to generally define it: Adding an “not present” feature x_0 with $x_0 = 1$ ($\forall i : x_0^{(i)} = 1$) the hypothesis can be written as

$$h_\theta(x) = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ . \ . \ . \ + \theta_n \cdot x_n = \theta^T \cdot x$$

with $x = \begin{pmatrix} x_0 \\ x_1 \\ . \ . \ . \\ x_n \end{pmatrix} \in \mathbb{R}^{n\,+\,1}$ and $\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ . \ . \ . \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n\,+\,1}$

The additional feature x_0 is the cause for the $+1$ in the dimension of x and θ.

The notation for multivariate linear regression is summarized in table 6.5.

Table 10: Table 6.5: Summary of Multivariate Linear Regression Model and Gradient Descent

Hypothesis:	$h_\theta\left(x\right) = \theta^T \cdot x = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 + \ . \ . \ . + \theta_n \cdot x_n$
Parameters:	$\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ . \ . \ . \\ \theta_n \end{pmatrix}$
Cost function:	$J\left(\theta\right) = J\left(\theta_0, \theta_1, \ . \ . \ . \ , \theta_n\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$
Goal:	$\min_{\theta} J\left(\theta\right)$
Gradient Descent:	$\text{repeat til conv. } \{ \ \theta_j = \theta_j - \alpha\frac{\partial}{\partial\theta_j}J\left(\theta\right), j = 0, \ . \ . \ . \ , n \ \}$

Looking into the details, the cost function except the hypothesis is the same in comparison to linear regression with one feature. The hypothesis includes all features ($x_1, \ . \ . \ . \ , x_n$) and the artificial feature ($x_0 = 1$).

The gradient descent algorithm for multiple features ($n > 1$) is similar. In detail, the algorithm uses the partial derivatives

$$\theta_j = \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_j^{(i)}, \text{with } j = 0, \ . \ . \ . \ , n$$

repeat until convergence {

$$\begin{aligned} &\theta_0 = \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_0^{(i)} \\ &\theta_1 = \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_1^{(i)} \\ &\theta_2 = \theta_2 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_2^{(i)} \\ &. \ . \ . \} \end{aligned}$$

Keep in mind that $x_0^{(i)} = 1$ for all i. Thus, $x_0^{(i)}$ in partial derivative θ_0 is only for generalization reasons.

Non-linear Regression

In the example house prices, figure 6.1 shows a green non-linear line for the hypothesis. Let us assume that the hypothesis is defined as $h_\theta(x) = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot \sqrt{x}$ with x is the size of the house. The hypothesis can be reduced to $h_\theta(x) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2$ where x_1 represents values of x and x_2 values of $\sqrt{x}$. Thus, this problem is reduced to an already known linear regression problem.

This is also possible if values are combined. Imagine, e.g. length (x_1) and width (x_2) are features and they are combined to the house area by $x_1 \cdot x_2$.

Thus, the non-linear hypothesis is e.g. $h_\theta(x) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_1 \cdot x_2$. This can be reduced to $h_\theta(x) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2$ with new x_2 representing values of $x_1 \cdot x_2$. The introduction of a new variable x_3 was intentionally avoided that the algorithms work as introduced before. In general, every non-linear regression problem can be reduced by using the mentioned mechanism.

6.3 Logistic Regression

For introducing classification, the example from figure 6.2 (malware in mail) is used again. The classification example has two classes for malicious mails. The discrete output value y denotes whether a mail is in class “malicious” $y = 1$ or in “not malicious” $y = 0$. The concept of the hypothesis can not be assigned one-to-one to classification. The predicted values of the hypothesis $h_\theta(x)$ in regression are not limited to $0 \leq h_\theta(x) \leq 1$. The idea is to formalize the hypothesis in such a way that

$$\text{if } h_\theta(x) \geq 0.5, \quad \text{predict} \quad \text{“}y = 1\text{”}$$

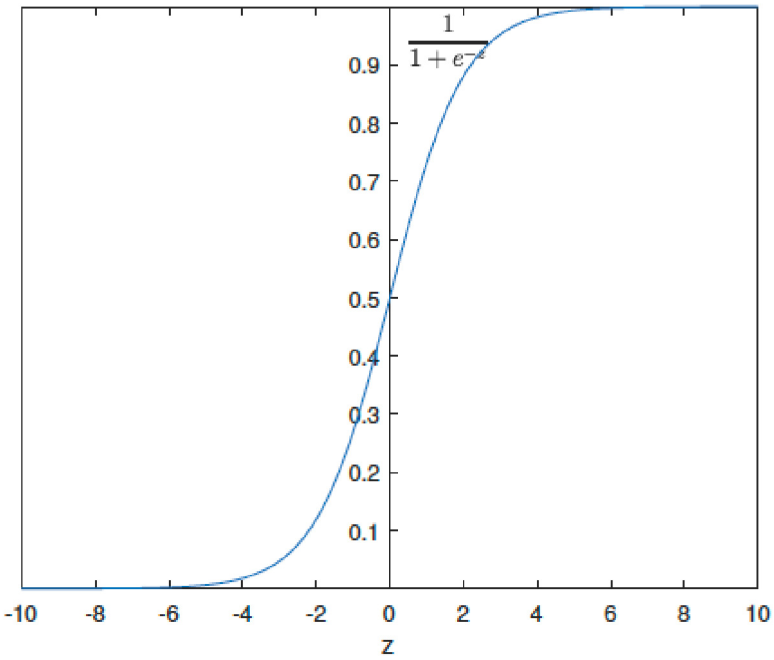
$$\text{if } h_\theta(x) < 0.5, \quad \text{predict} \quad \text{“}y = 0\text{”}$$

Thus, logistic regression for classification is used that $0 \leq h_\theta(x) \leq 1$. For logistic regression, the regression hypothesis $h_\theta\left(x\right) = \theta^T x$ is embedded into the sigmoid function (cf. fig.

6.7) $g\left(z\right) = \frac{1}{1 + e^{-z}}$ resulting in

$$h_\theta\left(x\right) = g\left(\theta^T x\right) = \frac{1}{1 + e^{-\theta^T x}}$$

Figure 35: Figure 6.7: Sigmoid Function g(z) = 1 1+e−z



How to interpret the predicted output value of the $h_\theta(x)$ Lets take a look at an example with a concrete input value x (new mail to be classified).

The input value is $x = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ \text{Mail Size} \end{pmatrix}$ results e.g. in predicted value $h_\theta(x) = 0.8$. This value gives a probability of 80% that this mail is malicious or from the other perspective 20% that the mail is not malicious. Thus, the value represents the “probability for input value x, parameterized by $\theta \mathrm{that} y = 1$”.

decision boundary Decision Boundary separates different classes and is also called hypothesis.

The **decision boundary** $h_\theta(x)$ (hypothesis in classification) is explained using the sigmoid function in figure 6.7. Thus, the resulting class is decided by

$$\begin{aligned} y = 1, \quad \text{if} \quad h_\theta\big(x\big) \geq 0.5 \Leftrightarrow \theta^T x \geq 0 \\ y = 0, \quad \text{if} \quad h_\theta\big(x\big) < 0.5 \Leftrightarrow \theta^T x < 0 \end{aligned}$$

The decision between class $y = 1$ and $y = 0$ is explained by using example from figure 6.3, taking into account the classification of malicious mails with two features. The decision boundary (black line) is already calculated. Lets assume a linear decision boundary $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = -20 + 2x_1 + x_2$.

Thus, predict class $y = 1$ if $-20 + 2x_1 + x_2 \geq 0$ or in other words $2x_1 + x_2 \geq 20$.

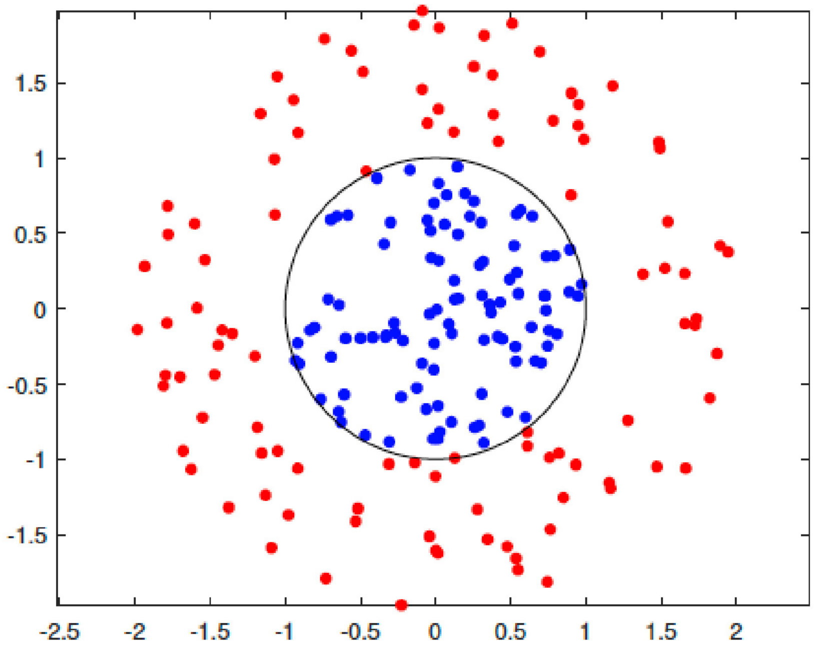
If $2x_1 + x_2 < 20$ the prediction is the other class $y = 0$.

The usage of non-linear hypothesis is also possible. A little change to the mail example, the decision boundary is then a circle $h_\theta(x) = \theta_0 + \theta_1 x_1 + + \theta_2 x_2 + \theta_3 x_1^2 + + \theta_4 x_2^2$ as depicted in figure 6.8. As before, the red data points represent the malicious and the blue ones the not malicious mails.

For simplification, the boundary is $h_\theta(x) = -1 + x_1^2 + x_2^2$ ($\theta_1 = 0, \theta_2 = 0$). As with linear boundary function, the prediction is $y = 1$ if $-1 + x_1^2 + x_2^2 \geq 0$.

Thus, predicted values fulfilling $x_1^2 + x_2^2 \geq 1$ (outside of the circle) mean the input is classified as malicious ($y = 1$).

Figure 36: Figure 6.8: Non-linear Decision Boundary



After the definition of the hypothesis respectively decision boundary the cost function is explained for classification. As in regression, the question is: how to choose parameters θ? In logistic regression, the cost function is $J\left(\theta\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$ whereas the main part of the equation is substituted by $\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \sum_{i=1}^{m} cost\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$. Thus, the logistic regression cost function is defined as

$$cost\left(\left(h_\theta\left(x\right), y\right)\right) = \begin{cases} -log(h_\theta(x)) \text{ if } y = 1 \\ -log(1 - h_\theta(x)) \text{ if } y = 0 \end{cases}.$$

Lets take a look at the first part of the cost function $-log(h_\theta(x))$ if $y = 1$.

During the calculation with the learning algorithm, if $y = 1$, $h_\theta(x) = 1$ the $cost(h_\theta(x), y) = 0$ because the logarithm of 1 is equal to 0. If $h_\theta(x) \to 0$ then $cost(h_\theta(x), y) \to \infty$. Thus, the learning algorithm is penalized by high cost if the prediction $h_\theta(x) = 0$ but $y = 1$. The part $-\log(1 - h_\theta(x))$ if $y = 0$ has the characteristic that if $y = 0, \quad h_\theta(x) = 0$ the $\cos t(h\theta(x), y) = 0$ because the algorithm of 1 is again 0.

The logistic regression cost function defined in sections can be written as

$$cost(h_\theta(x), y) = \ -y \cdot log(h_\theta(x)) - (1 - y) \cdot log(1 - h_\theta(x))$$

If $y = 1$, the cost function cost$cost(h_\theta(x), y) = \ -log(h_\theta(x))$ and if $y = 0$, the cost function $\cos t(h_\theta(x), y) = \ -\log(1 - h_\theta(x))$. By that equation the cost function $J(\theta)$ is defined after back substitution as

$$\begin{aligned} J(\theta) &= \tfrac{1}{m}\textstyle\sum_{i=1}^{m} cost\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \\ &= -\tfrac{1}{m}\left[\textstyle\sum_{i=1}^{m} y^{(i)} \cdot log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \cdot log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] \end{aligned}$$

Same as in regression, to find the best fitting hypothesis, the parameters θ are calculated by minimizing this cost function $J(\theta)$. By gradient descent, $\underset{\theta}{\min} J\left(\theta\right)$ is solved by the algorithm

repeat until convergence {

$$\begin{aligned} \theta_j &= \theta_j - \alpha \frac{\partial}{\partial\theta_j} J\left(\theta\right) \\ &= \theta_j - \alpha \tfrac{1}{m}\textstyle\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_j^{(i)}, \text{with } j = 0, \ . \ . \ ., n \end{aligned}$$

}

The gradient descent algorithm is identical to the algorithm in regression.

The derivation is dispensed due to longer calculation. The difference is the hypothesis with $h_\theta\left(x^{(i)}\right) = \frac{1}{1 + e^{-\theta^T \cdot x}}$. Thus, table 6.6 summarizes the elements of logistic regression.

Table 11: Table 6.6: Summary of Linear Logistic Regression with two Classes

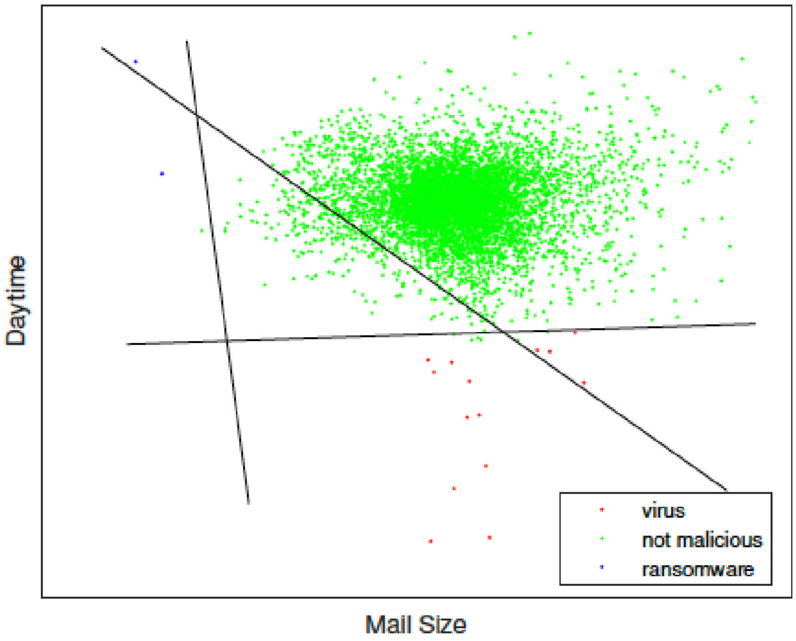
Hypothesis:	$h_\theta\left(x\right) = g\left(\theta^T \cdot x\right) = \frac{1}{1+e^{-\theta^T \cdot x}}, \text{ with } g\left(z\right) = \frac{1}{1+e^{-z}}$
Parameters:	$\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \cdots \\ \theta_n \end{pmatrix}$
Cost function:	$J\left(\theta\right) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \cdot log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \cdot log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right]$
Goal:	$\min_\theta J\left(\theta\right)$
Gradient Descent:	$\text{repeat til conv. } \{ \ \theta_j = \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_j^{(i)}, \text{with } j = 0, \ldots, n\}$

Multi-class Classification

After classification with two classes, the question arises: how to deal with more than two classes? This is called multi-class classification. Therefore, the example with malicious mails is updated. The classes are not only malicious or not but instead the type of malware is introduced. The example is adapted to virus ($y = 1$), ransomware ($y = 2$) and not malicious ($y = 3$).

Figure 6.9 illustrates the mentioned classes and decision boundaries. The colors depict the labeled data points with virus (red), ransomware (blue) and not malicious mails (green). The decision boundaries are calculated by reducing the problem from multi-class back to binary (two classes) classification.

Figure 37: Figure 6.9: Multi-class Classification - Malware in Mail



The method one-vs-all uses the already known algorithm. The reduction to one-vs-all is not dependent on the amount of classes. For every calculation of a decision boundary, one class is chosen and all others are "labeled" as the other class. Thus, the problem is a binary classification.

The goal is to train a logistic regression classifier $h_\theta^{(c)}\left(x\right)$ for each class c to predict the probability that $y = c$.

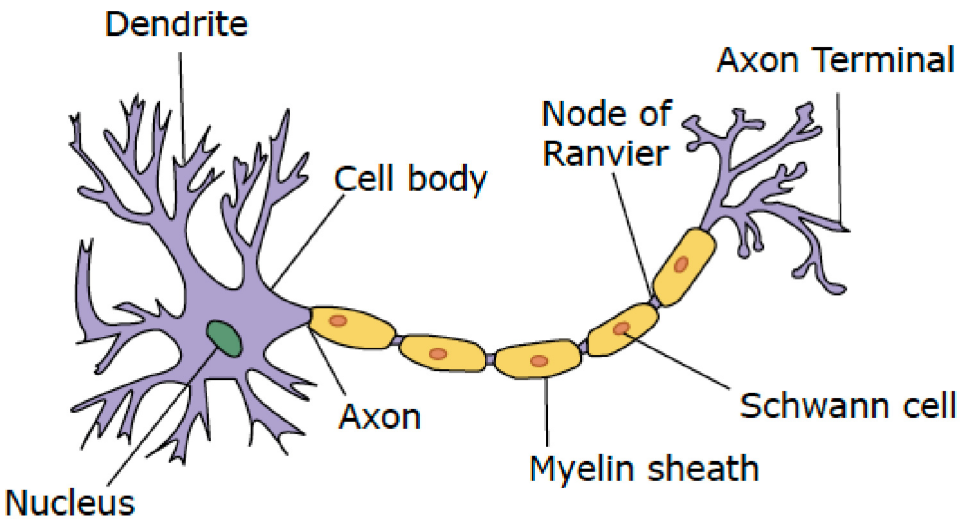
For example, choose the class virus. Thus, the red data points are separated from the "green + blue" data points. Using a linear hypothesis, the result is the almost horizontal straight line. The next step is to choose e.g. the class not malicious. Thus, the green data points are separated from the remaining "red + blue" data points. The result is the straight line from top left to bottom right. In the figure, it looks as the algorithm is finished because of the separation by the two decision boundaries. This is a "false friend". So far, only two decision boundaries are calculated. Hint: The number of decision boundaries are equal to number of classes. Therefore, the third straight line is resulting with choosing class ransomware. Thus, the blue data points are separated from the "green + red" data points. The result is the vertical straight line on the left.

On a new input x, the prediction for the right class is made by selecting the class c that maximizes the hypothesis $h_\theta^{(c)}\left(x\right)$. Thus, the formally written $\max_c h_\theta^{(c)}\left(x\right)$. In the example, the prediction can be written for the classes as vector $\begin{pmatrix} \text{virus} \\ \text{ransomware} \\ \text{not malicious} \end{pmatrix}$ with e.g. predicted values for input x $h_\theta^{(c)}\left(x\right) = \begin{pmatrix} 0.7 \\ 0.1 \\ 0.2 \end{pmatrix}$. Thus, the probability for a virus is 70%, for ransomware is 10% and not malicious is 20%. In total, the sum has to be 100%. The prediction will result to virus because of the maximal value $h_\theta^{(1)}\left(x\right) = 0.7$ with the probability of 70%.

6.4 Artificial Neural Networks

Artificial neural networks try to imitate the human brain. Many interconnected neurons form the neural network. Figure 6.10 illustrates a single neuron of the human brain. Such a neuron consists of input wires (dendrites) and output wires (axons). Thus, a neuron gets electrical impulse over the dendrites, processes it internally and axons can give another impulse to next dendrites of another neuron.

Figure 38: Figure 6.10: Neuron in Human Brain (Dhp1080, 2019)



For artificial neural networks, this concept is used to model artificial neurons.

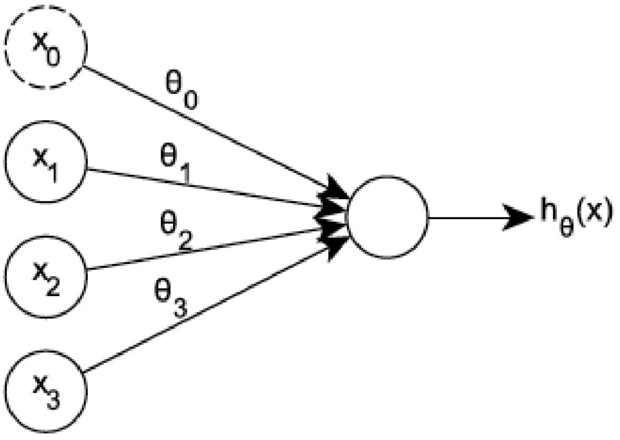
Figure 6.11 represents the model for artificial neurons. An artificial neuron has as electrical impulse the input values x_j with $j = 1, \ldots, n$ (all features) depicted as nodes and the dashed bias node x_0 ($x_0 = 1$). The input wires are represented as weighted edges connected to the “empty” node.

The values of weights are represented by θ_j, also called parameters as before.

activation function Activation function determines if a neuron fires or not.

The output wire is an edge with **activation function** $h_\theta(x)$. The activation function in neurons is the same logistic function (hypothesis) as in classification.

Figure 39: Figure 6.11: Artificial Neuron as Graph

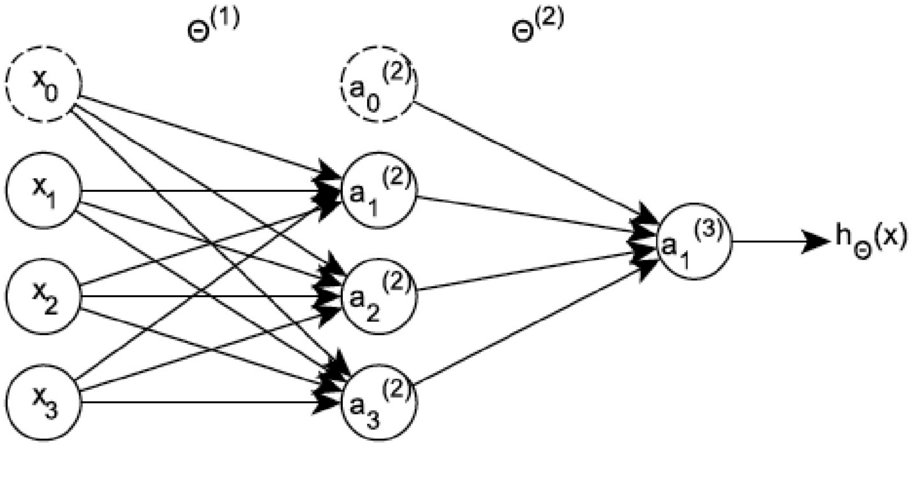


In the example, the artificial neuron consists of

$$\begin{aligned} \text{input values } x &= \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}, \\ \text{weights } \theta &= \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix} \text{ and} \\ \text{activation function } h_\theta(x) &= \frac{1}{1+e^{-\theta^T x}} \end{aligned}$$

Artificial neural networks are built by neurons. They are introduced by abstract example. The first layer is called the input layer, followed by zero to many hidden layer and finally an output layer. In figure 6.12 the layer 1 is the input layer x ($x_0, \ldots, x_3$), layer 2 is a hidden layer ($a_0^{(2)}, \ldots, a_3^{(2)}$) and layer 3 the output layer with one node referring to $h_\Theta(x)$. The weights are omitted in the figure for clarity.

Figure 40: Figure 6.12: Artificial Neural Network



In general, $a_i^{(j)}$ represents the activation function of neuron i at layer j of an artificial neural network. Thus, $a_i^{(j)}$ can be recognized as function $h_\theta(x)$ in each node. The nodes $a_0^{(j)} = 1$ are the added bias nodes at each layer j. The output of the network is depicted by $h_\Theta(x)$. Be aware of the difference between θ (vector) and Θ. $\Theta^{(j)}$ is a **matrix of weights** controlling function mapping from layer j to layer $j+1$. Additionally, the input layer with x_i can be written as $a_i^{(1)}$ of course without an activation function in place.

matrix of weights
Matrix of weights is the adjacency matrix of weighted sub-graph of the network.

Let us take a look in detail at the example depicted in figure 6.12. At training stage, in layer 1 the input values are the training data sets with here $x_1, \ldots, x_3$. In hidden layer (layer 2), the nodes $a_1^{(2)}, \ldots, a_3^{(2)}$ are defined as

$$
\begin{aligned}
a_1^{(2)} &= h_\Theta\Big(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\Big)\\
a_2^{(2)} &= h_\Theta\Big(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\Big)\\
a_3^{(2)} &= h_\Theta\Big(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\Big)
\end{aligned}
$$

Hint: The bias node $a_0^2 = 1$ is not listed.

The output node $a_1^{(3)}$ is defined as

$$
h_\Theta\Big(x\Big) = a_1^{(3)} = h_\Theta\Big(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}\Big)
$$

If the artificial neural network consist of s_j nodes in layer j and s_{j+1} nodes in layer $j+1$ then matrix $\Theta^{(j)}$ is of dimension $s_{j+1} \times s_j + 1$. In the example, $\Theta^{(1)}$ is of dimension 3×4 with three nodes ($x_1, \ldots, x_3$) in layer 1 and three nodes ($a_1^{(2)}, \ldots, a_3^{(2)}$) in layer 2. Additionally, in layer 1 is a bias node. Thus layer 1 has $3 \quad + \quad 1$ nodes. The schema of matrix $\Theta^{(1)}$ can be depicted in definition of a $a_1^{(2)}, \ldots, a_3^{(2)}$.

Reusing the example in figure 6.12, the forward propagation with concrete values is shown. It is explained using one input data set for the input layer.

The data can be summarized by

$$\text{input values } x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 3 \end{pmatrix},$$

$$\text{weight matrices } \Theta^{(1)} = \begin{pmatrix} -10 & 2 & 3 & 4 \\ 2 & 2 & -5 & 4 \\ -1 & -1 & 4 & -2 \end{pmatrix}, \Theta^{(2)} = \begin{pmatrix} 4 & -2 & -3 & 4 \end{pmatrix}$$

The activation function for hidden and output layers is $h_\Theta\left(\Theta \cdot a_i^{(j)}\right) = \frac{1}{1 + e^{-\Theta a_i^{(j)}}}$. The activation function has a matrix-vector multiplication $\Theta \cdot a_i^{(j)}$ as parameter value.

Let us start with the forward propagation. Input layer x respectively $a^{(1)}$ equals the input vector and so needs no calculation. The hidden layer (layer 2) is calculated by using weight matrix $\Theta^{(1)}$ and input values x

$$\begin{aligned} a_1^{(2)} &= h_\Theta\left(-10 \cdot 1 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 3\right) = \frac{1}{1 + e^{-14}} = 0.99 \\ a_2^{(2)} &= h_\Theta\left(2 \cdot 1 + 2 \cdot 3 - 5 \cdot 2 + 4 \cdot 3\right) = \frac{1}{1 + e^{-10}} = 0.99 \\ a_3^{(2)} &= h_\Theta\left(-1 \cdot 1 - 1 \cdot 3 + 4 \cdot 2 - 2 \cdot 3\right) = \frac{1}{1 + e^{2}} = 0.12 \end{aligned}$$

The output node $a_1^{(3)}$ is calculated by using weight matrix $\Theta^{(2)}$ and values $a_i^{(2)}$ with $i = 0, \ldots, 3$

$$h_\Theta\left(x\right) = a_1^{(3)} = h_\Theta\left(4 \cdot 1 - 2 \cdot 0.99 - 3 \cdot 0.99 + 4 \cdot 0.12\right) = \frac{1}{1 + e^{0.47}} = 0.38$$

Further deep dive into the topic is omitted because of required advanced mathematics. Just for imagination and further reading the cost function $J(\Theta)$ for multi-class classification problems is defined

$$J\left(\Theta\right) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log\left(h_\Theta\left(x^{(i)}\right)_k\right) + \left(1-y_k^{(i)}\right) log\left(1-h_\Theta \left(x^{(i)}\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{j,i}^{(l)}\right)^2$$

with number of layers L, number of nodes in layer s_l and number of output nodes (possible classes) K. The regularization parameter λ is used to reduce overfitting of the hypothesis. Hint: regularization parameter can also be used in regression and classification. Additionally, the details of forward and back propagation in neural networks are omitted. **Forward propagation** is described above. From input values x to output neuron(s) with $h_\Theta(x)$, neurons in each layer process and forward regarding their activation function.

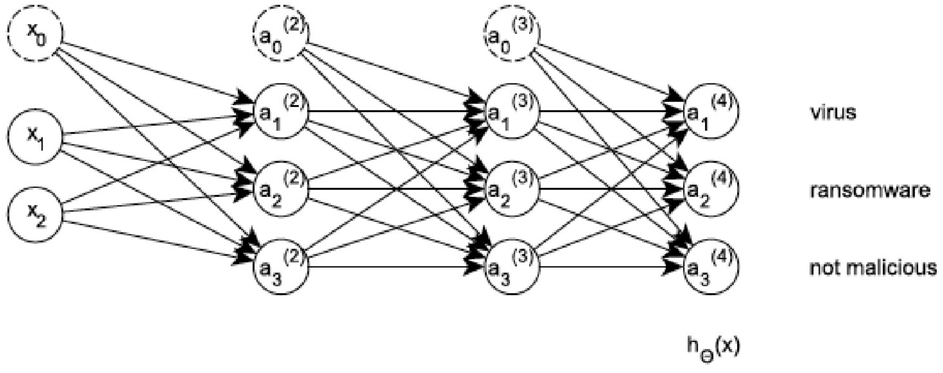
Forward propagation
Forward propagation is the calculation from input layer to output layer.

In back propagation, gradient descent is used for optimizing the weights/ parameters θ_j at the edges in the neural network (or via using the weight matrices $\Theta^{(j)}$). Therefore, the neuron(s) at the output layer are the starting point and processed through all layers towards the input layer. In **back propagation** an excerpt of required elements is the cost function $J(\Theta)$, the weight matrices $\Theta^{(j)}$ at layer j and the activation functions of each neuron $a_i^{(j)}$ of node i at layer j. Thereafter, the neural network is ready to predict outputs y for new entries at the input layer. In reference to sections before, the predicted output can be the prediction for linear or non-linear regression problems or linear or non-linear classification problems (also multi-class).

back propagation
Back propagation is an algorithm to train neural networks using gradient descent and the cost function.

After the introduction of artificial neural networks, the question arises: what are “needful” scenarios for them? Lets take a look at an example from IT security for classifying mails. At section 6.3 in figure 6.9, the multi-class classification problem implies two features - mail size and daytime. The classes for predicted output are virus, ransomware or not malicious. An adapted artificial neural network for this problem is illustrated in figure 6.13.

Figure 41: Figure 6.13: Multi-class Classification with Artificial Neural Network



The exemplary neural network consists of the input layer with two features x_1, x_2, two hidden layers with $a_1^{(2)}, \ldots, a_3^{(2)}$ at layer 2 and $a_1^{(3)}, \ldots, a_3^{(3)}$ at layer 3, and the output layer with $a_1^{(4)}, \ldots, a_3^{(4)}$ representing the classes virus, ransomware and not malicious. The bias nodes are not explicitly mentioned. Vector $h_\Theta\left(x\right) \in \mathbb{R}^3$ with probabilities for the mentioned classes is used to determine which class is predicted. Therefore, the maximal value in $h_\Theta(x)$ gives the class. $h_\Theta\left(x\right) = \begin{pmatrix}0\\1\\0\end{pmatrix}$ would predict with probability of 100% the mail as ransomware. Keep in mind that also intermediate results are possible like $h_\Theta\left(x\right) = \begin{pmatrix}0.3\\0.1\\0.6\end{pmatrix}$ predicting a mail as not malicious with probability 60%.

In real world scenarios, more features could be required for correct predictions.

In the example, features are e.g. mail size, daytime, sender, geographical location, and so on. Lets assume 100 features $(x_1, \ldots, x_{100})$ are involved and non-linear classification is needed. Remember, non-linear classification means a non-linear classifier is used. The used part $\theta^T x$ in $h_\theta\left(x\right) = g\left(\theta^T x\right) = \frac{1}{1+e^{-\theta^T x}}$ could be

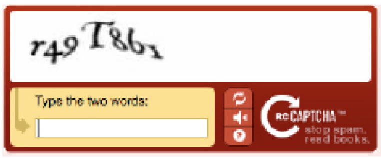
$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1 x_3 + \ldots$$

Thus, in non-linear logistic regression (classification), the features $(n = 100)$ with the combination of each feature with each other results to $\frac{n(n-1)}{2} = 4950$ features. If the combinations of features are extended like $x_1 x_2 x_3$, $x_1^2 x_3$, $x_{14}^3$with “power of 3”, the resulting features are 170.000 features.

So much combinations result in a better fitting classifier (or hypothesis in regression). Overfitting will not be addressed. Thus, in such a scenario a neural network reduces the effort by connecting features as needed in a graph-based neural network.

Another impressive example is image recognition in computer vision. From security point of view, captchas are used to avoid automated brute forcing of passwords at websites. An attacker could use automated tools to crack the password with billions of requests per second. Captchas are used so that users have to provide user, password and solve the captcha to avoid this attack. Captchas contain a random image with e.g. letters and numbers involved. Figure 6.14 depicts the randomly produced string “r49T861”. An attacker wants to automatically recognize captchas so that the brute force attack will work again.

Figure 42: Figure 6.14: Captcha used in Websites at Authentication Step



Therefore, an artificial neural network can be used to recognize the written text. Let us assume that the attacker wants to recognize fictitious numbers and letters in images (captchas). The image e.g. is displayed in low quality with 100×40 pixels. Thus, 4000 features are the input values $x = \begin{pmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \cdots \\ \text{pixel 4.000 intensity} \end{pmatrix}$. If the hypothesis consists of features "power of 2" (e.g. $x_1 x_4, x_3 x_{189}$), the number of features is $n = 7\,.\,998\,.\,000 \approx 8$ million.

A neural network will be trained with different captchas using supervised learning. After finished training step, the neural network can predict the captcha so that the text in the image can be automatically inserted at the website for each single brute force attempt. This example shows that machine learning is not restricted to avoid and find attacks or attempts.

For example, intrusion detection or intrusion prevention systems look at network traffic or behavior of computers to prevent successful attacks. The captcha example shows that also attackers can use these techniques to successfully attack.

SUMMARY

Nowadays, artificial intelligence and machine learning are key factors to detect or prevent cyber attacks. The shortcomings of e.g.

signature-based antivirus are overcame by such approaches. The static comparison of known signatures of malware is reconditioned using regression and classification. Therefore, the determination is not programmed by hand. Machine learning is the learning of a specific task by experience (training data). Thus, arbitrary malware can be found by prediction.

Supervised and unsupervised learning are different basic approaches of machine learning. Supervised learning uses labeled data sets consisting of input features and output values. Regression predicts continuous output values whereas classification predicts discrete output values. For both, linear as well as non-linear hypothesis functions are used. In regression, the hypothesis returns a predicted output value for a given input by using the input as parameter for the function. Classification predicts with given input the appropriate class as output value. The hypothesis is the decision boundary to determine the “right” class. The returned class is predicted with a probability for all possible classes and the most fitting (maximal probability) is chosen. In contrast to supervised learning, unsupervised learning learns with not labeled and unstructured data. Data is grouped/clustered by commonalities.

Linear and non-linear regression or classification (logistic regression), including univariate and multivariate regression, have main constituents the hypothesis with their parameters, the cost function and a goal. In regression, the hypothesis is constructed linear or non-linear and the related parameters are calculated by minimizing the cost function. Thus, the error between the prediction (hypothesis) and the actual training output is minimal. A method for minimizing the cost function and finding the parameters of the hypothesis is gradient descent. In classification, the procedure to find the parameters of the hypothesis is the same. There the hypothesis uses the sigmoid function to predict a probability between 0 and 1. The goal is to find parameters for hypothesis that the classes are separated as distinct as possible. Multi-class

classification deals with more than two classes. The only difference is that the prediction has more classes involved, with each class having their own probability for a specific input.

Artificial neural networks consist of neurons building a directed weighted graph. The network has different layers reaching from one input to arbitrary hidden til one output layer. The nodes in the graph are the neurons with their activation function (hypothesis).

The edges represent interconnections of neurons with parameters for neurons as weights. The neurons at the input layer represent the input values of a network and the “last” neuron(s) the output values.

In forward propagation, the output is calculated following the directed edges. Thus, the prediction is at the output layer differing between continuous output value (regression) and discrete value (probability of class(es) in classification). In back propagation, the parameters (weights) of the activation functions are optimized by minimizing the cost function e.g. with gradient descent. The term “back” indicates the process starts at the output layer and finishes at the input layer.