Syllabus Information Systems

Jan Martijn van der Werf

April 26, 2019

Natur und Kunst

Natur und Kunst, sie scheinen sich zu fliehen,
Und haben sich, eh' man es denkt, gefunden;
Der Widerwille ist auch mir verschwunden,
Und beide scheinen gleich mich anzuziehen.

Es gilt wohl nur ein redliches Bemühen!
Und wenn wir erst in abgemeßnen Stunden;
Mit Geist und Fleiß uns an die Kunst gebunden,
Mag frei Natur im Herzen wieder glühen.

So ist's mit aller Bildung auch beschaffen:
Vergebens werden ungebundne Geister
Nach der Vollendung reiner Höhe streben.

Wer Großes will, muß sich zusammenraffen:
In der Beschränkung zeigt sich erst der Meister,
Und das Gesetz nur kann uns Freiheit geben.

Johann Wolfgang Goethe (1880)

Preface

This syllabus is the outcome of teaching the course Information Systems several times. When students start the course, they often consider mathematics to be difficult and hard. The many different obscure symbols, Greek letters, abstract notations and formulae many mathematicians employ strenghten that idea.

With this syllabus, I want to introduce the reader to the world of behavioral modeling using mathematical notions. Whereas many books on such topics are very theoretical and dive into the details of these notions, this syllabus wants to focus on the practical side of modeling. So, the focus is more on questions like "How can I use it?", "How can I create a model?" and "Are these two models the same?", rather than on presenting the underlying ideas of the mathematical notions.

Many of the more mathematical modeling notations are rarely used in practice. So, why study them? If introduced properly, these notions form a powerful tool in the toolbox of the modeller. Instead of teaching only the standard tools and techniques currently used in industry, focusing on he underlying mathematical notations helps to understand these standards better, including the standards of the future." And, more importantly, experience learns that knowing these mathematical notions, you will better and quicker understand stakeholders and their problems, it shapes your thinking and creativity, and better models!

This syllabus is the result of fruitful and inspiring discussions with many different people across different universities. A special word of thanks to Joost Gabriels, Kees van Hee and Natalia Sidorova from Eindhoven, and Fabiano Dalpiaz, Sietse Overbeek and Sjaak Brinkkemper from Utrecht.

Jan Martijn van der Werf
Utrecht, 2016

Contents

1 Introduction

Organizations nowadays heavily depend on information systems. An information system supports an organization by collecting, storing, and retrieving (business) data, as well as it supports or executes the processes within the organization, or even cross-organizational. With the ever growing dynamism of society, not only should the organization be agile, but also should its information systems be easy adaptable to changing requirements.

1.1 Background: Information Systems

Early information systems were collections of monolithic systems, each dedicated to its own task. Each system was designed to automate a frequently occurring (business) process within an organization, such as updating the general ledgers independently of other processes. Based on the *control flow*, i.e., the order of actions needed to perform the process, a program was constructed that processed an input file, producing other files. These files were totally unrelated, thereby making it hard to maintain consistency. A nice example of such a dedicated monolithic system is the mechanical tabular of Hollerith [6], which is considered to be the first automated information system. The machine worked with punched cards. It was introduced in 1890 [7], when the census of 1890 was estimated not to be completed before 1900. Hollerith, who was a statistician, approached the United States Census Office, and offered to use his mechanical tabular. The use of the mechanical tabular allowed reducing the time needed to publish the first results from eight years to only six weeks. The census was completed within two years [8].

Although early information systems were highly modularized, each module had its own dedicated task, and they were not integrated well. Processed files were unrelated. For example, records in several files represented different aspects of the same objects. In the sixties of the last century, the focus

was more on the data aspect of information systems. In 1968, IBM introduced the Information Management System / Virtual Storage (IMS/VS) [2]. However, it took until the introduction of the relational data model by Codd [4] for specialized database systems to be adopted. These database systems not only focused on data storage and retrieval, but also allowed for more advanced data management, including transaction management and authorization. Instead of monolithic systems each processing their own data file, all data files were integrated into database management systems. In this way, multiple application programs could be developed concurrently as soon as the database was defined.

The introduction of database management systems in information systems improved the integration of independent monolithic systems. However, support for changes in the control flow was minimal, since the control flow was hard-coded in the information systems. The introduction of workflow management systems in the nineties of the last century allowed separating the code into control flow and business logic. In other words, the sequencing of task is separated from the internal logics of each individual task. In turn, this led to the introduction of *Process Aware Information Systems* (PAIS). In a PAIS, the control flow layer is introduced. Hence, a PAIS has three aspects: the *data aspect*, the *control flow aspect* and the *business logic aspect*. All aspects are designed independently, and then integrated into a single system.

1.2 Design Process of Information Systems

The design of an information system and its components comprises many different activities and disciplines. Most design approaches only differ in the grouping and ordering of design tasks. In this syllabus, we consider an abstract model of the design process. In any design process we can distinguish nine important activities, which are executed in some order. Fig. 1.1 depicts an abstract model of the design process. Nodes are objects, either formal or informal, a double arc denotes an activity that creates an object out of the other. Note that the figure does not prescribe any order of the activities, it only depicts the relation between the activities.

Scoping and justification An information system supports an organization in the "real world". This means that terms and activities in the real world need to be translated into formal concepts, as the information system needs to understand these. *Domain experts* need to have a deep understanding of the real world. Together with the *requirements engineer*, the *scope* of a component is fixed, describing the boundaries of the component, the stakeholders involved in the component, and the functionality of the component. This activity is called *scoping*. It results in a requirements document in terms of the client, i.e. the

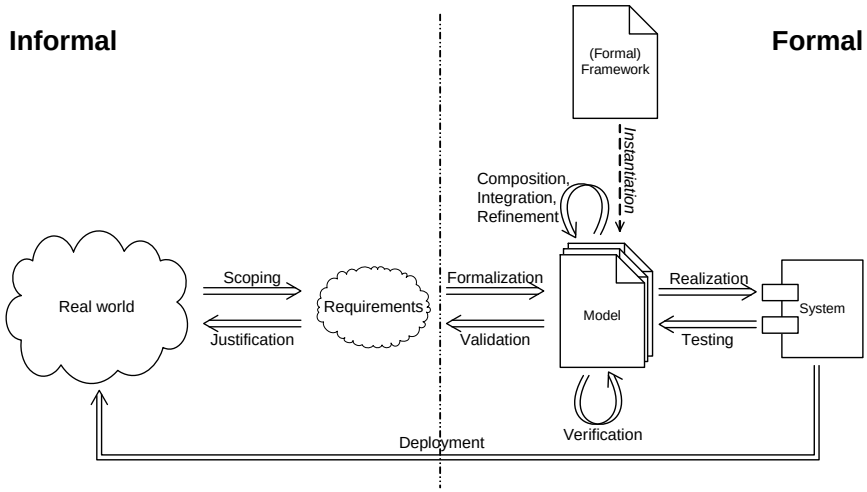


Figure 1.1: Meta-model of design process of an information system

functionality of the component is described in natural domain-specific language and (mostly) informal diagrams.

The rationale of each decision taken in the scoping activity has to be justified. For each decision, there has to be a reason in the real world. The activity of checking the requirements against the real world, is called *justification*.

Formalization and validation The requirements are still informal, while the component to be built is formal. The activity of translating the scoped world from informal requirements to (formal) models is called *formalization*. Models describe the requirements. All models together form the *architecture* of the component. An *architectural framework* defines for an architecture which type of models are needed and how these are related. If an architecture is according to some framework, i.e., it has all the models prescribed by the framework, we say the architecture is an *instance* of the framework.

A model is based on some *modeling language*. A modeling language defines the concepts that can be used, and their semantics. The activity of formalization is error prone, as requirements are expressed in natural language and informal diagrams. Therefore, the requirements are often ambiguous, and the models need to be checked as to whether they describe the requirements as intended. This activity is called *validation*. Validation can be done in many ways. One way is by guiding stakeholders through the model explaining the model. Another often used practice is by creating a prototype from the models, such that the stakeholders can get a look and feel of the system. In general, validation cannot be automated due to the informal nature of

the requirements. However, this task is crucial during development.

Composition, integration, refinement and verification Once the first models are created and validated, these models can be *integrated* into larger models, *decomposed* into smaller models to focus on different aspects, or *refined* into more precise models. Each step should be *verified* to be correct, i.e. the new collection of models should have at least the same properties as the original collection. The main difference between verification and validation is that verification is checking whether the model is correct, whereas validation is checking whether it is the correct model. While in refinement the focus lies on extending a model with more specified functionality, in integration the focus lies on combining different models into new models, in such manner that all properties of the models are preserved, and the composition has some additional properties.

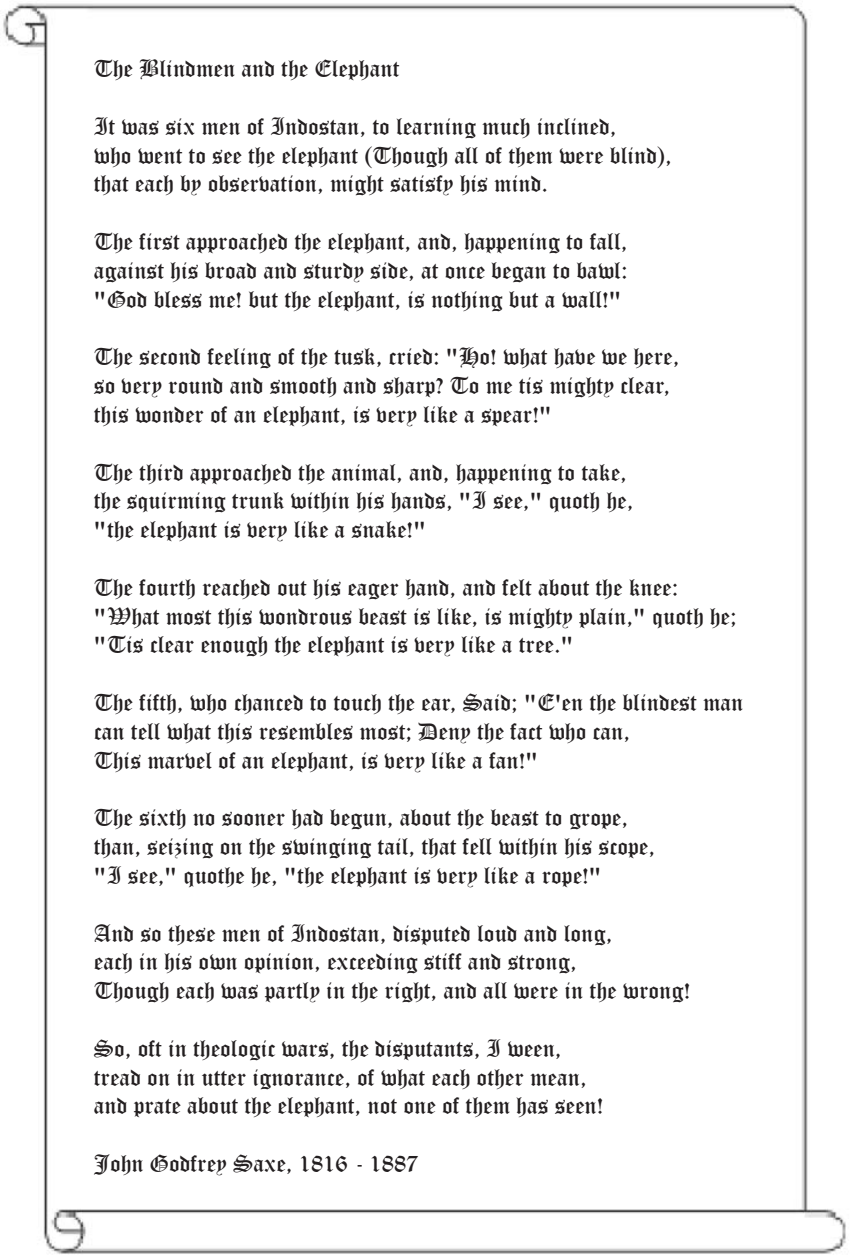
Realization, testing and deployment When the models reach a sufficient degree of precision, the component can be *realized*. In software development, this involves the search for existing subcomponents and their configuration, and the construction of new subcomponents. All subcomponents are integrated into a single component. To check whether the realized component indeed satisfies the design, it needs to be *tested* against the verified models. When the component is realized and thoroughly tested, it is *deployed* in the real world.

1.3 Modeling and Verification

Models play a central role in the design of an information system. A model is an abstract representation of some aspect of a real world system to analyze a set of properties of the system. We assume that properties are chosen such that if a property holds in the model, it also holds in the real world system. The activity of creating these models is called *modeling*. It comprises formalization, integration, composition and refinement.

Many different modeling languages exist, each focusing on different aspects of the system. Some languages focus on modeling the data aspect, like Entity-Relationship Diagrams [3], or on the process aspect, like Petri nets [11] and the Business Process Model and Notation (BPMN) [9].

It is desirable for models to be consistent with each other. In the poem of Saxe (Fig. 1.2), each of the blind men models a single aspect of the elephant. Each of them has a correct model of the elephant, focusing on some aspects. Together, the models represent the elephant as a whole. The design of an information system is similar. Each of the models describes some aspects of the system. Together, the models represent the information system. Therefore, verification in a modeling process does not only require



The Blindmen and the Elephant

It was six men of Indostan, to learning much inclined,
who went to see the elephant (Though all of them were blind),
that each by observation, might satisfy his mind.

The first approached the elephant, and, happening to fall,
against his broad and sturdy side, at once began to bawl:
"God bless me! but the elephant, is nothing but a wall!"

The second feeling of the tusk, cried: "Ho! what have we here,
so very round and smooth and sharp? To me tis mighty clear,
this wonder of an elephant, is very like a spear!"

The third approached the animal, and, happening to take,
the squirming trunk within his hands, "I see," quoth he,
"the elephant is very like a snake!"

The fourth reached out his eager hand, and felt about the knee:
"What most this wondrous beast is like, is mighty plain," quoth he;
"Tis clear enough the elephant is very like a tree."

The fifth, who chanced to touch the ear, Said; "E'en the blindest man
can tell what this resembles most; Deny the fact who can,
This marvel of an elephant, is very like a fan!"

The sixth no sooner had begun, about the beast to grope,
than, seizing on the swinging tail, that fell within his scope,
"I see," quothe he, "the elephant is very like a rope!"

And so these men of Indostan, disputed loud and long,
each in his own opinion, exceeding stiff and strong,
Though each was partly in the right, and all were in the wrong!

So, oft in theologic wars, the disputants, I ween,
tread on in utter ignorance, of what each other mean,
and prate about the elephant, not one of them has seen!

John Godfrey Saxe, 1816 - 1887

Figure 1.2: The Blindmen and the Elephant, John Godfrey Saxe, 1816 - 1887

checking for correctness of each of the models, but also checking for the consistency between models.

1.4 Modeling and Mathematics

Many different notations and formalisms exist for modelling information systems. Industrial standards are typically created and maintained by large consortia. As each participating organisation like their own symbols and notations to be included, such standards can become very large and cumbersome. For example, the Business Process Model and Notation (BPMN) standard [9] has more than 50 symbols that can be used. This makes it hard to grasp all details of such a notation, and is one of the reasons that many people only use a very small subset of the language.

Another aspect of industry standards, is that many are either informal or semi-formal. As an example, consider the BPMN model depicted in Fig. 1.3. Ask several persons the intention of the model, and you most certainly obtain different answers! One possible explanation of the model might be that once activity A is finished, activities B and C are executed in any order, completed by activities D and E. A second person might state that there is a choice between B and C. And a third person, who actually read the complete BPMN standard (which is 508 pages long!), states that activities A, B and C are executed, after which activity D is executed twice. Who has the right interpretation of this model? According to the standard, the third person is right: activity D is executed twice. But, what was the intention of the modeller who created this model? Was it the intention of the first person, the second or the third? Or had the modeller a different intention?

This example shows that it is important that people have a common understanding of the model to actually be able to discuss the intention behind the model. When the same model leaves room for multiple interpretations, how will you be able to transfer your ideas and knowledge to other people?

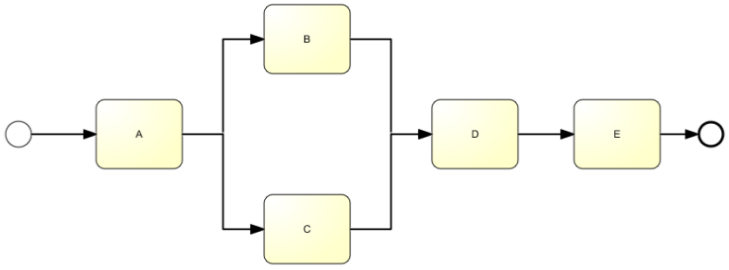


Figure 1.3: BPMN model with different interpretations among modellers

That is the reason why we introduce mathematical notions in modeling. It ensures that everybody has the same interpretation of the model. One might disagree, but at least the modeller is able to transfer their intention.

1.5 Outline

This syllabus is structured as follows. In Chapter 2, we introduce the basic mathematical notions such as sets, relations and functions. This chapter is intended for the readers that are inexperienced with mathematical notations like first order logic and set theory.

Chapter 3 introduces graphs. Many of the modeling notations have a graph-based visualisation, or can even be represented by graphs. In this chapter, we introduce the mathematical notation of graphs, and show how one can systematically traverse and search the nodes within the graph. Last, we explain the shortest path algorithm by E.W. Dijkstra.

A first modeling notation to capture information flows are transition systems, presented in Chapter 4. Transition systems are labelled graphs that represent the states in which a system can be, and how the system can move from one state to another. After introducing the mathematical notions, we show how such systems can be compared and constructed.

The last chapter, Chapter **??** of this syllabus introduces Petri nets. Petri nets form a natural abstraction of transition systems. In this chapter, we introduce the basic notions, and show how these models can be analysed using techniques presented in the previous chapters. Additionally, we present different classes of Petri nets with their properties.

2

Sets, Relations and Functions

2.1 Sets

Within mathematics, a *set* represents a collection of objects. This collection can be of anything, for example, all students registered for the course "information systems" can be represented as a set: each student following this course belongs to this collection.

We use curly brackets to denote a set. Suppose the students John, Claire, Peter, Alice and Bob follow the course "information systems", then the set representing this class is denoted by $C = \{\text{John}, \text{Claire}, \text{Peter}, \text{Alice}, \text{Bob}\}$. If the collection does not contain any elements, we say the set is empty, which we denote with $\emptyset$.

On sets, we want to make statements, like "all students that participate in course X, are students of our university". To formalize this statement, let U be the set of all students of our university. Then the statement can be reformulated to: "all students that are element of the set of students that follow course X are an element of the set of all students of our university". Thus, all elements of C are an element of U as well. We can write this in logic:

$$\forall x \in C : x \in U$$

Read this as "for all x that are element of C, x is an element of U as well. In set theory we say that C is a subset of U, denoted by $C \subseteq U$. Similarly, we could make a statement that no student that is enrolled in the course is a university student. We first rephrase this to: "all students that are enrolled in the course, are not a student of the university. Additionally, we need another operator, the negation, denoted by $\neg$. With this additional operator we can formalize the statement:

$$\forall x \in C : \neg(x \in U)$$

The next definition shows all shorthand notations we will use throughout the course.

Definition 2.1 (Set notation)**.** *A* set *is a collection of* elements*. We denote a set by listing its elements between braces. E.g., a set S with elements a, b and c is denoted as $\{a, b, c\}$. The* empty set*, i.e. the set with no elements, is denoted by $\emptyset$. Let A and B be two sets. we define the following operations.*

- *$|A|$ denotes the number of elements of A.*
- *an element a is* included *in a set A, denoted by $a \in A$. As a shorthand for $\neg(a \in A)$ we write $a \notin A$.*
- *the* intersection *of two sets, denoted by $A \cap B$, is a set containing the elements which are in both sets: $A \cap B = \{x \mid x \in A \ \wedge \ x \in B\}$.*
- *the* union *of two sets, denoted by $A \cup B$, is the set containing all elements of both sets: $A \cup B = \{x \mid x \in A \ \vee \ x \in B\}$.*
- *the* difference *between two sets, denoted by $A \setminus B$, is the set containing all elements of A which are not in B: $A \setminus B = \{x \mid x \in A \ \wedge \ x \notin B\}$.*
- *the set B is a* subset *of A, denoted by $B \subseteq A$ if all elements of B are also in A: $\forall x \in B : x \in A$. The set B is called a* proper subset*, denoted by $B \subset A$, if $B \subseteq A$, but not $A = B$. The* powerset*, denoted by 2^A, is the set of all subsets of A: $2^A = \{A' \mid A' \subseteq A\}$. Note that $A \in 2^A$.*

The sets A and B are disjoint *if $A \cap B = \emptyset$. A* partition *of a set A is a set $P \subseteq 2^A$ such that $A = \bigcup_{A' \in P} A'$ and $\forall A', A'' \in P : A' \cap A'' \neq \emptyset \implies A' = A''$.*

Sets may be infinite, i.e., it is impossible to list all elements of that set. Within mathematics many infinite sets exist. For example, the set of natural numbers, the set of real numbers or the set of prime numbers, to name just a few.

Some sets have a special symbol. Within this syllabus, we denote the set of all natural numbers by $\mathbb{N} = \{0, 1, 2, \ldots\}$. The set of all positive natural numbers, i.e, the set of all natural numbers that are larger than 0, is denoted by $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.

A common way to visualize sets is using a Venn-diagram. An example is depicted in Fig. 2.1. This figure depicts four sets: A, B, C, and D.

2.2 Relations

Elements can be related to other elements. For example, let $S = \{a, b, c\}$ denote a set of students, and let $C = s, t, u, v$ denote a set of courses. To

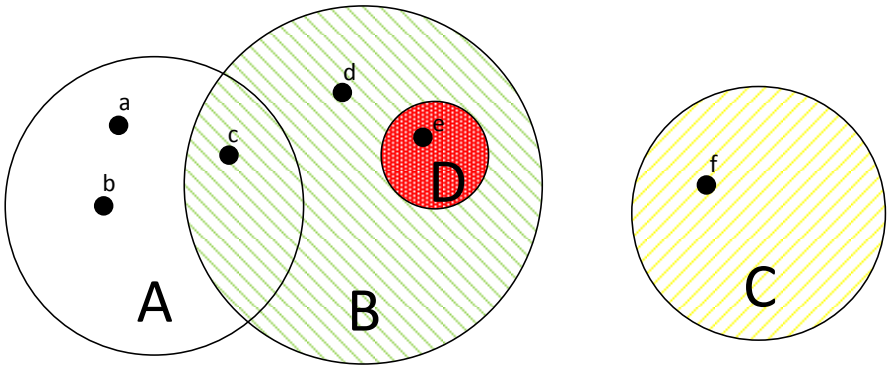


Figure 2.1: Four sets depicted as a Venn-diagram

state that a student follows a course, we need a new set, where each elements states that a student follows a specific course.

For this, we introduce the *Cartesian product*, that relates each element in one set with each element in some other set. In our student example, the Cartesian product of S and U is a set of "pairs": $S \times C = \{(a,s),(a,t),(a,u),(a,v),(b,s),(b,t),(b,u),(b,v),(c,s),(c,t),(c,u),(c,v)\}$. The relation $S \times C$ is visually shown in Fig. 2.2. Thus, the Cartesian product contains all possible combinations!

Definition 2.2 (Cartesian product)**.** *The* Cartesian product *of two sets A and B is defined as the set $A \times B = \{(a,b) \mid a \in A \ \wedge \ b \in B\}$. Set A is called the* source set*, and set B is called the* target set*.*

On the Cartesian product, we define the projection functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ by $\pi_1((a,b)) = a$ and $\pi_2((a,b)) = b$ for all $(a,b) \in A \times B$.

In the Cartesian product any element of the one set is related to any element of the other set. However, often, not all elements need to be related: in our example, some courses might be optional! Therefore, a relation is "only" a subset of the Cartesian product: it relates some elements of one set to some elements of some other set.

In our example, it might be that a followed courses s and t, b only followed course t and c did not yet follow any course. This relation is then defined by $R = \{(a,s),(a,t),(b,t)\}$. This relation R is visualized in Fig. 2.3. Al elements in the relation are part of the Cartesian product. Hence, relation $R \subseteq S \times C$. We call S the *domain* of relation R, and C the *range* of relation R. Notice that not all elements need to be in the domain or the range of the relation. In our example, not all students are present: student c did not yet follow a course! Similarly, not all courses are followed by some student.

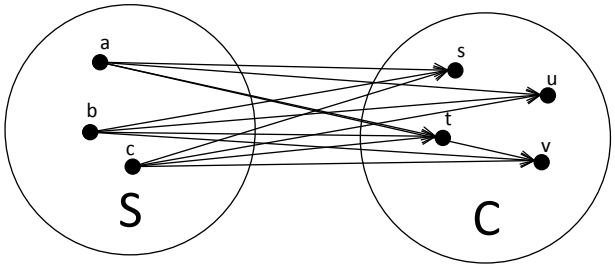


Figure 2.2: Cartesian product of two sets S and T.

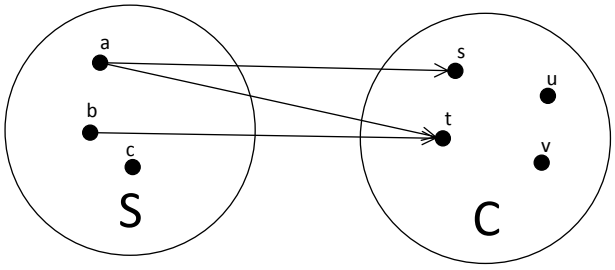


Figure 2.3: Relation R

By inverting the relation, i.e., by switching all pairs in the relation, we get the inverse relation. In our case $R^{-1} = \{(s, a), (t, a), (t, b)\}$.

This gives us the following operations on relations:

Definition 2.3 (Relation, domain, range, inverse)**.** *Let A and B be two sets. A set $R \subseteq A \times B$ is called a* relation from A to B*. The* domain *of the relation, denoted by $dom(R)$, is the set $dom(R) = \{a \in A \mid \exists b \in B : (a, b) \in R\}$. Its* range*, denoted by $rng(R)$, is the set $rng(R) = \{b \in B \mid \exists a \in B : (a, b) \in R\}$. Its* inverse*, denoted by R^{-1}, is a relation $R^{-1} \subseteq B \times A$ such that $R^{-1} = \{(b, a) \in B \times A \mid (a, b) \in R\}$.*

Given some sets and a relation, we want to test statements. An example could be that course t is mandatory for all students. A mandatory course should be followed by all students. Rephrased: all students have followed course s. In formula this becomes:

$$\forall x \in S : (x, t) \in R$$

To check the statement, we need to consider all students, and check whether the specified relation exists, i.e., R should contain elements (a, t), (b, t) and (c, t). As the latter element is missing, this statement is not true!

Similarly, we would like to express that each course has at least one student that participated. For this, we need another quantifier, the "there exists" quantifier, that we denote with $\exists$. With this quantifier, we can rephrase the statement to "for all courses there exists at least one student that participated in that course. In formula:

$$\forall y \in C : \exists x \in S : (x, y) \in R$$

To check the statement, we need to consider all courses, and then check whether there is a student that enrolled: for course s we have student a, for course t we have a as well, but no student enrolled for course u. Hence, the statement is false!

Functions

Another important class of relations is the class of functions. A relation from A to B is a function if each element of A is related to at most one element of B. A function is partial if not all elements of A are mapped onto an element of B. If the inverse of a function is again a function, it is injective; it is surjective if $rng(f) = B$.

Definition 2.4 ((Partial) function, identity, injection, surjection, bijection)**.** *Let A and B be two sets. A relation $f \subseteq A \times B$ is a function from A to B, denoted by $f : A \to B$, if $(a, b_1) \in f$ and $(a, b_2) \in f$ imply $b_1 = b_2$ for all $a \in A$ and $b_1, b_2 \in B$. We write $f(a) = b$ for $(a, b) \in f$.*

A special function is the identity function *$id : A \to A$, which is a function that maps any element to itself: $id(a) = a$ for all $a \in A$.*

A function is called a partial function*, denoted by $f : A \rightharpoonup B$, if $dom(f) \subset A$. When $dom(f) = \emptyset$ the function is called the* empty function*, denoted by $\emptyset$. If $f(a_1) = f(a_2)$ implies $a_1 = a_2$ for any $a_1, a_2 \in dom(f)$, the function f is an* injection*. It is a* surjection *if for any $b \in B$, there exists an $a \in dom(f)$ such that $f(a) = b$. An injective and surjective function is called a* bijection*.*

Properties of Relations

A special class of relations is the class of relations that have the same source set and target set. On these relations we define the notions of reflexivity, symmetry, transitivity and antisymmetry.

Let us consider the set of all integers, denoted by $\mathbb{Z}$. A relation we can define on these numbers is the "equal to" relation, which we denote by $(\mathbb{Z}, =)$, i.e.,

$$\forall x, y \in \mathbb{Z} : (x, y) \in = \Leftrightarrow x = y$$

This relation has a special property: each element is mapped on itself. This is what we call *reflexive*. Another property this relation has, is *symmetry*: if (a, b) is in the relation, then (b, a) is in the relation as well.

Let us consider another relation on the integers, the "smaller than" relation, denoted by $(\mathbb{Z}, <)$. Is this relation reflexive? No, a number cannot be smaller than itself! Thus, it is never the case that $a < a$, for any number $a \in \mathbb{Z}$. If a relation has this property, we say it is *irreflexive*. If we consider three integers, say a, b and c, such that $(a, b) \in <$ and $(b, c) \in <$, then $a < b$ and $b < c$. But also $a < c$. If this holds for any three elements in the domain of the relation, we call the relation *transitive*.

Another well-known relation on the set of integers is the "smaller than, or equal to" relation, denoted by $(\mathbb{Z}, \leq)$. In fact $\leq = < \cup =$. As $=$ is reflexive, the relation $\leq$ is reflexive as well, and similarly, because $<$ is transitive, $\leq$ is transitive as well.

Is the "smaller than, or equal to" relation symmetric? For this, we need to check whether $a \leq b$ implies $b \leq a$ for all integers. It is quite obvious to see that this is not the case. For example, choose $a = 3$ and $b = 4$. Then clearly the statement does not hold. However, there is an interesting observation in this relation: this statement is only true if we compare the numbers with themselves, i.e., $a \leq b$ and $b \leq a$ is only true if $a = b$. This is what we call *antisymmetry*.

Definition 2.5 ((Ir)reflexive, symmetric, transitive, antisymmetric)**.** *Let A be a set and let $R \subseteq A \times A$ be a relation. R is* reflexive *if $(a, a) \in R$ for all $a \in A$, and it is* irreflexive *if $\neg((a, a) \in R)$ for all $a \in A$. If $(a, b) \in R$ implies $(b, a) \in R$ for all $a, b \in A$, the relation is* symmetric*. If $(a, b) \in R$ and $(b, c) \in R$ imply $(a, c) \in R$ for all $a, b, c \in A$, the relation is* transitive*. Relation R is* antisymmetric *if $(a, b) \in R$ and $(b, a) \in R$ imply $a = b$ for all $a, b \in A$.*

Using these definitions, we define orderings and equivalences on sets. Both types of relations are reflexive and transitive. In addition, an ordering relation is also antisymmetric, whereas an equivalence relation is symmetric. This leads to the following definitions.

Definition 2.6 (Partial order, total order, least element, top element, well-ordered)**.** *Let A be a set. A relation $R \subseteq A \times A$ is a* partial order*, denoted by (A, R), if R is reflexive, transitive and antisymmetric. A partial order is called a* total order*, if in addition $(a, b) \in R$ or $(b, a) \in R$ for all $a, b \in A$. An element $a \in A$ is a* least element *of (A, R) if $\neg\exists b \in A : (b, a) \in R$. It is a* top element *of (A, R) if $\neg\exists b \in A : (a, b) \in R$. If (A, R) is a total order, and any non-empty subset $B \subseteq A$ has a least element, (A, R) is* well-ordered*.*

Note that in a total order both the least element and the top element are unique.

Definition 2.7 (Equivalence relation)**.** *Let A be a set. A relation $R \subseteq A \times A$ is an* equivalence relation *if it is reflexive, symmetric and transitive.*

The reflexive closure of a relation R is the smallest relation that is reflexive and contains R. The transitive closure of a relation R is defined as the smallest transitive relation S such that R is contained in it.

Definition 2.8 (Reflexive closure, transitive closure)**.** *Let A be a set and let $R \subseteq A \times A$ be a relation. Its* reflexive closure *$S \subseteq A \times A$ is a relation such that $(a, b) \in S$ if and only if $a = b$ or $(a, b) \in R$ for all $a, b \in A$. Its* transitive closure *$T \subseteq A \times A$ is a relation such that $R \subseteq T$, T is transitive and for all relations $T' \subseteq A \times A$ such that $R \subseteq T'$ and T' is transitive, then $T \subseteq T'$.*

2.3 Bags

A set only indicates whether an element is present or not. As an example, consider the set of all Euro coins: 1ct, 2ct, 5ct, 10ct, 20ct, 50ct, 1E and 2E. The set of all coins can be denoted by $\{1\text{ct}, 2\text{ct}, 5\text{ct}, 10\text{ct}, 20\text{ct}, 50\text{ct}, 1\text{E}, 2\text{E}\}$. However, to denote that your wallet contains 5 euro coints, 2 10ct coins and 3 50ct coins, the set notation does not suffice. For this, we introduce the notion of a *bag*, or *multiset*, in which also the number of occurrences of the elements is considered. In a bag, we write the number of elements in superscript behind the element. In this way, the bag representing my wallet can then be denoted by $W = [10\text{ct}^2, 50\text{ct}^3, 1\text{E}^5]$. If an element only occurs once, we omit the superscript, and if the element does not occur at all, we leave that element out.

Bags can be added, which means that we add the elements of both bags (we call this elementwise). Suppose I receive 10 1ct coins, 5 50ct coins, and 4 2 Euro coins, then my wallet now looks like: $W + [1\text{ct}^{10}, 50\text{ct}^5, 2\text{E}^4] = [1\text{ct}^{10}, 10\text{ct}^2, 50\text{ct}^8, 1\text{E}^5, 2\text{E}^4]$.

Substraction is similar to addition: we just substract elementwise. Thus, if I need to pay 1,50 euro with a single euro coin and a 50ct coin, we have $W - [50\text{ct}, 1\text{E}] = [10\text{ct}^2, 50\text{ct}^2, 1\text{E}^4]$. However, subtraction is slightly more complicated than addition: I cannot substract more than I have. Thus, if we are short in some element, we cannot substract. In our wallet example, if I have to pay with 5 euro coins, i.e., $W - [1\text{E}^5]$, we are 1 euro coin short, thus this subtraction is not possible, we do not borrow any coins!

Definition 2.9 (Bags)**.** *Let S be a set. A* bag *B over S is a function $B : S \to \mathbb{N}$. For $s \in S$, $B(s)$ denotes the number of occurrences of s in the bag B. We write $\mathbb{N}^S$ for the set of all bags over S. The* empty bag*, i.e., the bag for which all elements the multiplicity is 0, is denoted by $\emptyset$. Bags are denoted by listing the occurring elements between square brackets and we use superscripts for the multiplicity of the occurrences. If the multiplicity of an element is 0, we omit the element. A bag m consisting of two occurrences of a, three occurrences of b and a single occurrence of c*

is denoted by $m = [a^2, b^3, c]$. *The* characteristic function $\chi : 2^S \to \mathbb{N}^S$ *is defined as* $\chi(S')(s) = 1$ *if* $s \in S'$ *and* $\chi(S')(s) = 0$ *otherwise for all* $s \in S$ *and* $S' \subseteq S$.

Definition 2.10 (Bag notation)**.** *Let* $X, Y \in \mathbb{N}^S$. *On bags, we define the following operations:*

- $s \in X$ *if and only if* $X(s) > 0$ *for all* $s \in S$;
- $(X + Y)(s) = X(s) + Y(s)$ *for all* $s \in S$;
- $(X - Y)(s) = \max(0, X(s) - Y(s))$ *for all* $s \in S$;
- $X = Y$ *if and only if* $\forall s \in S : X(s) = Y(s)$
- $X \leq Y$ *if and only if* $\forall s \in S : X(s) \leq Y(s)$;
- $X < Y$ *if and only if* $X \leq Y$ *and* $X \neq Y$.

The projection of X *on elements of a set* $U \subset S$ *is denoted by* $X_{|U}$, *and is defined by* $X_{|U}(u) = X(u)$ *for all* $u \in U$ *and* $X_{|U}(s) = 0$ *for all* $s \in S \setminus U$.

2.4 Sequences

Last in this chapter, we introduce the notion of sequences. Bags only count the number of occurrences of elements, a sequence also takes the order of the elements into account.

Consider our alphabet, which contains 26 characters. Thus, we can represent it as a set of 26 symbols. However, we cannot represent a single word with it. Within a language, we create words by putting letters one of the other. The ordering makes that we understand their meaning. Consider the words "saint" and "Stain". Their set representation would be $\{s, a, i, n, t\}$. In fact, we can create many words with these letters. As the number of occurrences is abstracted away, the word "saints" is represented by the same set. With bags we have a similar problem: both words are represented by the same bag $[s, a, i, n, t]$.

Sequences are used to represent the order in which elements occur. For example, the word "saint" is represented by the sequence $\langle s, a, i, n, t\rangle$, and "saints" by $\langle s, a, i, n, t, s\rangle$. A sequence always has a certain length. In our example of "saint", the sequence has length 5. Each position has an element. Thus, asking the element on position 3 results in the element i.

Definition 2.11 (Sequences)**.** *Let* S *be a set. A* sequence σ *over* S *of length* $n \in \mathbb{N}$ *is a function* $\sigma : \{1, \ldots, n\} \to S$. *If* $n > 0$ *and* $\sigma(i) = a_i$ *for* $i \in \{1, \ldots, n\}$, *we write* $\sigma = \langle a_1, \ldots, a_n\rangle$ *and* σ_i *for* $\sigma(i)$. *The length of* σ *is denoted by* $|\sigma|$.

The sequence of length 0 *is called the* empty sequence *and is denoted by* ϵ. *The set of all finite sequences over* S *is denoted by* S^*; *the set* S *is called the* alphabet *of* S^*.

An element $s \in S$ *is included in a sequence* $\sigma \in S^*$, *denoted by* $s \in \sigma$, *if* $\exists 1 \leq i \leq |\sigma| : \sigma(i) = s$.

Let $\mu, \nu \in S^*$. Concatenation, *denoted by* $\sigma = \mu; \nu$, *is defined as* $\sigma : \{1, \ldots |\mu| + |\nu|\}$ *such that for* $1 \leq i \leq |\mu|$: $\sigma(i) = \nu(i)$ *and for* $|\mu| < i \leq |\mu| + |\nu|$: $\sigma(i) = \nu(i - |\mu|)$.

The projection of a sequence $\sigma \in S^*$ *on a set* $U \subseteq S$, *denoted by* $\sigma_{|U}$ *is inductively defined as* $\epsilon_{|U} = \epsilon$, $(\langle t \rangle; \sigma)_{|U} = \langle t \rangle; \sigma_{|U}$ *if* $t \in U$, *and* $(\langle t \rangle; \sigma)_{|U} = \sigma_{|U}$ *if* $t \notin U$. *Let* $U, R \subseteq S$ *and* $\mu, \nu \in S^*$. *Then* $(\mu_{|U})_{|R} = \mu_{|U \cap R}$ *and* $(\mu; \nu)_{|U} = \mu_{|U}; \nu_{|U}$.

We denote a subsequence of $\sigma \in S^*$ *from index* i *to* j *by* $\sigma_{[i..j]}$. *If* $j \leq i$, *then* $\sigma_{[i..j]} = \epsilon$, *otherwise* $\sigma_{[i..j]} = \langle \sigma(i), \ldots, \sigma(k) \rangle$ *where* $k = \min(j, |\sigma|)$.

Further, we define a partial order $\leq$ *on sequences by* $\mu \leq \nu$ *if and only if there exists a sequence* $\rho \in S^*$ *such that* $\nu = \mu; \rho$.

To denote the number of occurrences of elements in a sequence, we introduce the Parikh vector [10], which is a bag representing the number of occurrences of each element in the sequence.

Definition 2.12 (Parikh vector)**.** *Let* S *be a set and let* $\sigma \in S^*$ *be a sequence. The* Parikh vector *of* σ, *denoted by* $\overrightarrow{\sigma}$ *is inductively defined by* $\overrightarrow{\epsilon} = 0$ *and* $\overrightarrow{\langle a \rangle; \sigma} = [a] + \overrightarrow{\sigma}$ *for all* $a \in S$.

3 Graphs

Graphs play an important role in the design and analysis of information systems. In this section, we introduce the basic concepts of graph theory.

One way to visualize a relationship between elements is shown in Fig. ??. The elements are depicted as dots, and the relations between these dots as directed arrows, to indicate the domain and range of the relationship. Graphs generalize this visualization. A graph consists of *nodes*, representing the elements, and *edges*, that represent the relationship. An Example is depicted in Fig. 3.1, which represents a part of the Dutch railway system. The nodes and edges represent the train stations and the tracks that connect these stations, respectively.

Figure 3.1: Part of the Dutch railway network

Another graph is depicted in Fig. 3.2. This graph represents a computer network. The nodes represent the different elements in the network, and the edges depict how these elements are connected.

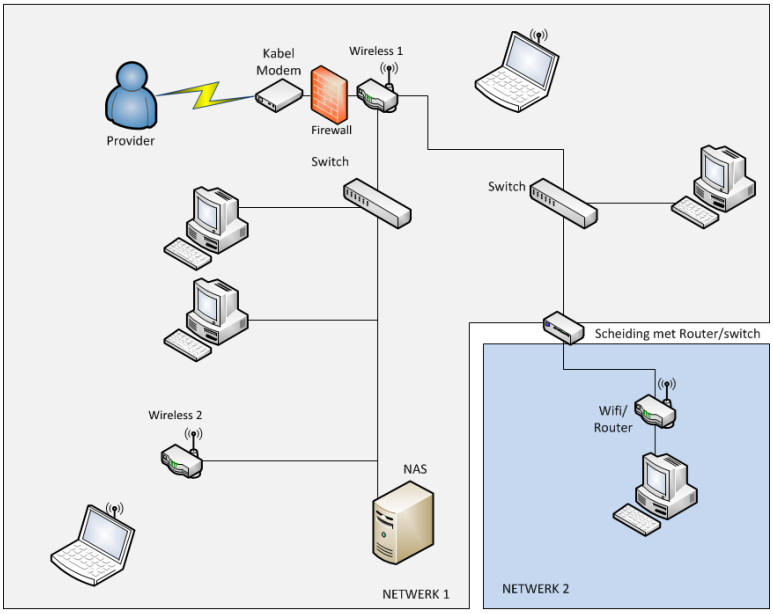


Figure 3.2: A computer network

These graphs are very useful to express properties like "Nijmegen is reachable from Eindhoven" or "to reach the NAS from any external system passes the Firewall". By traversing the edges of the graph one can validate such properties.

In this chapter, we first introduce the mathematical concepts for graphs, and next show how we can systematically traverse a graph to check properties, together with an application: the shortest path between two nodes.

3.1 Mathematical Definition

A graph consists of a set of nodes, and arcs between them. Nodes are sometimes referred to as *vertices*. Arcs have a direction, i.e., an arc has a head and a tail. If the set of arcs is symmetric, i.e. if (u, v) is an arc, (v, u) is also an arc, the graph is called undirected. A special class of graphs is the class of bipartite graphs, in which the vertices are partitioned into two sets, and there are no arcs whose tail and head are in the same set.

Definition 3.1 (Directed graph, undirected graph, bipartite graph)**.** *A graph G is a pair (V, A) with V a set of* vertices *and a relation $A \subseteq V \times V$ called the* arcs*. An arc $(u, v) \in A$ is directed from the* tail *u to the* head *v. If the relation A is symmetric, the graph is* undirected*. The graph is a* bipartite graph *if $\{V_1, V_2\}$ is a partitioning of V and $\forall (u, v) \in A : (u \in V_1 \Leftrightarrow v \in V_2) \wedge (u \in V_2 \Leftrightarrow v \in V_1)$.*

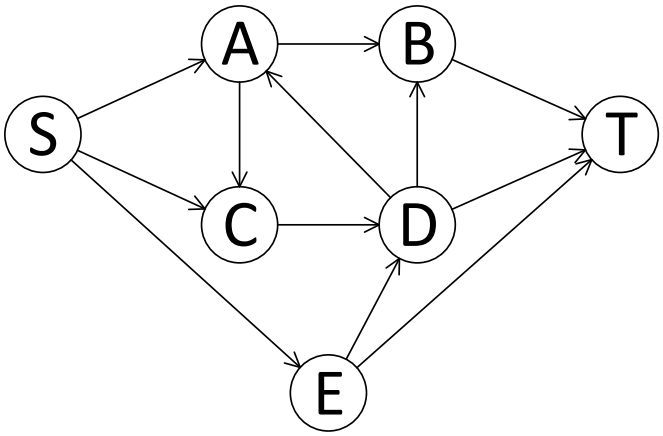


Figure 3.3: An example graph

An example graph is depicted in Fig. 3.3. This graph $G = (V, A)$ has 7 nodes: $V = \{S, A, B, C, D, E, T\}$ and 13 directed edges, i.e. arcs: $A = \{(S, A), (S, C), (S, E), (A, B), (A, C), (B, T), (C, D), (D, A), (D, B), (D, T), (E, D), (E, T)\}$.

Vertices are connected via directed arcs. The direct neighbours of a vertex v are either in the preset, i.e. the set of all vertices for which there is an arc pointing to v, or in the postset, i.e. the set of all vertices for which there is an arc to starting from v.

In the example graph, the preset of node A is ${}^\bullet_G A = \{S, D\}$. Its postset is the set $A^\bullet_G = \{B, C\}$.

Definition 3.2 (Preset, postset)**.** *Let $G = (V, A)$ be a directed graph. Let $u \in V$ be a vertex. The* preset *of a u, denoted by ${}^\bullet_G u$ is the set ${}^\bullet_G u = \{v \in S \mid (v, u) \in A\}$. The* postset *of u, denoted by $u^\bullet_G$ is the set $u^\bullet_G = \{v \in S \mid (u, v) \in A\}$. We lift the preset and postset to sets, i.e. ${}^\bullet_G U = \bigcup_{u \in U} {}^\bullet_G u$ and $U^\bullet_G = \bigcup_{u \in U} u^\bullet_G$ for some $U \subseteq V$. If the context is clear, we omit the subscript.*

Note that in an undirected graph the preset and postset of any vertex are identical. In a graph, we can choose a vertex and from this vertex traverse via the arcs to other vertices, thus creating a path. If we can traverse either way on the path, it is an undirected path. A path is a cycle if the start and end vertex of the path are the same. If the graph does not contain any cycles, it is an acyclic graph. A circuit is a cycle in which all vertices occur only once.

In our example graph, the sequence $\langle S, A, C, D, A, C, D, T\rangle$ is a path of G: all edges are in the graph, and the corresponding nodes are connected

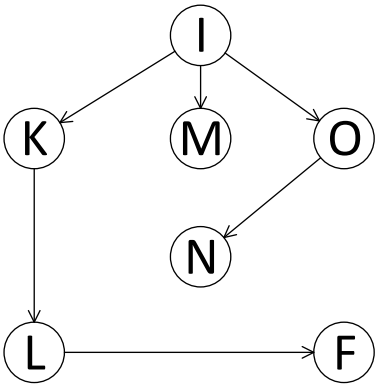


Figure 3.4: An example Tree

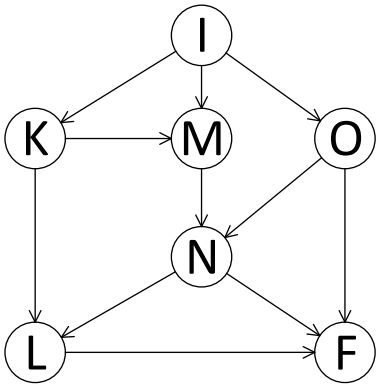


Figure 3.5: An acyclic graph

via an arc. Also $\langle S, A, B, T, E, S\rangle$ is a path of G: we can traverse arcs (S, A), (A, B) and (B, T) forward and arcs (T, E) and (E, S) backward to obtain the path. Hence, it is an undirected path. Another path of the graph is $\langle A, C, D, A\rangle$. As the first and last node are identical, it is a cyclic path. Except for the first node and last node (it is a cycle), all nodes occur only once in the path. Hence, $\langle A, C, D, A\rangle$ is a circuit.

Definition 3.3 ((Un)directed path, cycle, acyclic graph, circuit)**.** *Let $G = (V, A)$ be a graph. A sequence $p \in V^*$ of length $k > 0$ is a* directed path *if $(p_{i-1}, p_i) \in A$ for all $1 < i \leq k$. It is an* undirected path *if either $(p_{i-1}, p_i) \in A$ or $(p_i, p_{i-1}) \in A$ for all $1 < i \leq k$. It is a* cycle *if $p_1 = p_k$. If a graph does not contain any cycles, it is called an* acyclic graph*. A path $p \in V^*$ is called a* trail *if all elements occur only once: $\forall v \in V : \overrightarrow{p}(v) \leq 1$. A directed or undirected trail p is called a* circuit *if $(p_k, p_1) \in A$.*

A graph is connected if it is possible to reach any vertex from any other vertex, without taking the direction of the arcs into account. It is strongly connected if this property holds while respecting the direction of the arcs.

The example graph depicted in Fig. 3.3 is connected: from any node one can reach all other nodes by either traversing forward or backward the edges of the graph. However, it is not strongly-connected: the postset of node T is empty. Consequently, no node can be reached from this node!

Definition 3.4 ((Strongly) connected graph)**.** *Let $G = (V, A)$ be a graph. It is* connected *if for any two vertices $v_1, v_2 \in V$ there exists an undirected path from v_1 to v_2. It is* strongly connected *if for any two vertices $v_1, v_2 \in V$, there exists a directed path from v_1 to v_2.*

An important class of graphs are forests. A forest is a graph not containing any circuit. If the forest is also connected, it is a tree.

Consider the example in Fig. 3.4. It contains no circuits. Hence, it is a tree. The root of the tree is node I: it is the only node with no incoming arcs.

Definition 3.5 (Forest, tree)**.** *A graph $G = (V, A)$ is a* forest *if it does not contain any circuit. It is a* tree *if it is a connected forest.*

To inspect only parts of a graph, we introduce the notion of subgraphs. A subgraph generated by a subset of vertices of a graph is called an induced subgraph. If the subgraph is maximal with respect to the connections, it is a component.

The Graph $G' = (\{A, B, C, D\}, \{(A, B), (A, C), (C, D), (D, B)\})$ is a subgraph of Graph G, as the nodes and edges of G' are subsets of the nodes and edges of G, respectively. However, it is not an induced graph, as the arc (D, A) is not present in G'.

Definition 3.6 (Subgraph, induced subgraph, component)**.** *Let $G = (V, A)$ and $G' = (V', A')$ be two graphs. The graph G' is a* sub graph *of G, denoted by $G' \subseteq G$ if $V' \subseteq V$ and $A' \subseteq A$. The subgraph G' is* induced *if $A' = A \cap (V' \times V')$. A subgraph G' is a* component *if it is a maximal, connected, induced subgraph, i.e. there is no larger subgraph $G'' \subseteq G$ such that $G' \subseteq G''$ and G'' is a connected, induced subgraph.*

A special class of subgraphs are the spanning trees: a subgraph is a spanning tree if it is a tree containing all nodes of the graph. In our example graph of Fig. 3.3, the graph $G'' = (V, A')$ with $A' = \{(S, A), (S, E), (A, C), (A, B), (E, D), (E, T)\}$ is a spanning tree: the subgraph is a tree containing all nodes of G.

Definition 3.7 (Spanning tree)**.** *Let $G = (V, A)$ be a graph. A subgraph $G' = (V, A')$ is a* spanning tree *if G' is a tree.*

In acyclic graphs, it is possible to order the vertices such that for each vertex occurring in the order, its predecessors are smaller with respect to this order. We call this ordering a topological sort.

It is important to realize that a topological sort is only possible if the graph is acyclic.

Consider the acyclic graph depicted in Fig. 3.5. A topological sort of the nodes of the spanning tree in Fig. 3.4 is I, K, M, O, N, L, F.

Definition 3.8 (Topological sort)**.** *Let $G = (V, A)$ be an acyclic graph. A* topological sort *is a partial order of the vertices $\sqsubseteq_G \subseteq V \times V$ such that $\forall (u, v) \in A : u \sqsubseteq_G v$.*

Graphs can be drawn in many different ways. If a bijective function exists that transforms one graph into another graph, the two graphs are isomorphic.

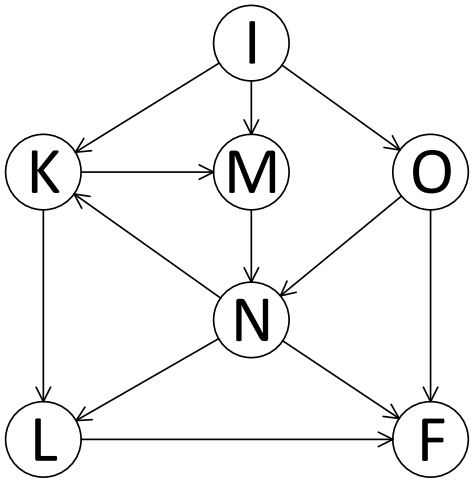


Figure 3.6: Graph isomorphic to graph G (Fig. 3.3)

Consider the graph H depicted in Fig. 3.6. Its layout is completely different from graph G, depicted in Fig. 3.3. To check whether these two graphs are isomorphic, we need to construct a bijective function that maps each node of graph G to a node of graph H. Let us construct such a function. let $f : V_G \to V_H$ be a function such that $f = \{(S, I),\ (A, K),\ (B, L),\ (C, M),\ (D, N),\ (E, O),\ (T, F)\}$. This is easy to check that f is both injective and surjective, i.e., it is bijective. Furthermore, if we map all arcs of G, we obtain $\{(f(S), f(A)),\ (f(S), f(C)),\ (f(S), f(E)),\ (f(A), f(B)),\ (f(A), f(C)),\ (f(B), f(T)),\ (f(C), f(D)),\ (f(C), f(E)),\ (f(D), f(A)),\ (f(D), f(B)),\ (f(D), f(T)),\ (f(E), f(D)),\ (f(E), f(T))\}$, which are exactly the arcs of H. Similarly, we can map all arcs of H to the arcs of G. For this mapping, we should use the inverse of f, as we need to map elements of H to elements of G, whereas f maps it the other way around. Consequently, graphs G and H are isomorphic!

Definition 3.9 (Isomorphic graphs)**.** *Let $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$ be two graphs and let $f : V_1 \to V_2$ be a bijective function. Graphs G_1 and G_2 are* isomorphic *with respect to f if $\forall (u, v) \in A_1 : (f(u), f(v)) \in A_2$ and $\forall (u, v) \in A_2 : (f^{-1}(u), f^{-1}(v)) \in A_1$.*

For many applications of graphs, not only the nodes have some meaning, the arcs have properties as well. We therefore often label the arcs. To do so, we introduce labelled graphs. In a labelled graph, each vertex and each arc has a label. In later chapters we present several applications of labelled graphs.

Definition 3.10 (Labelled (directed) graph)**.** *Let* Σ *be some set. A* labelled (directed) graph G *is a 3-tuple* (V, A, λ) *where* $A \subseteq V \times \Sigma \times V$ *is the set of* labeled arcs*, such that* $(V, \{(v, v') \mid \exists a \in \mathcal{A} : (v, a, v') \in A\})$ *is a (directed) graph, and the partial function* $\lambda : V \to \Sigma$ *is a* vertex labelling function*.*

Last in this section, we define the operations to combine graphs: union, intersection and difference are defined on their substituents.

Definition 3.11 (Union, intersection, difference)**.** *Let* $G_1 = (V_1, A_1)$ *and* $G_2 = (V_2, A_2)$ *be two graphs. The* union *of* G_1 *and* G_2 *is defined as* $G_1 \cup G_2 = (V_1 \cup V_2, A_1 \cup A_2)$*. The* intersection *of* G_1 *and* G_2 *is defined as* $G_1 \cap G_2 = (V_1 \cap V_2, A_1 \cap A_2)$*. The* difference *of* G_1 *with* G_2 *is defined as* $G_1 \setminus G_2 = (V_1 \setminus V_2, A_1 \setminus A_2)$*.*

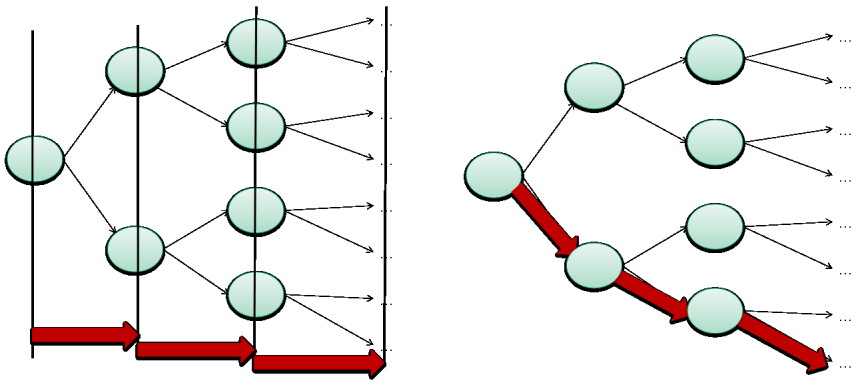
3.2 Traversing Graphs

Consider the railway system depicted in Fig. 3.1. One of the questions one might ask is "what is the shortest path from Nijmegen to Eindhoven?". To solve this question, one first needs to ask: "Can I reach Eindhoven from Nijmegen?". Traversing a graph is often referred to as searching a graph. Searching a graph always results in a spanning tree: all nodes are visited in a certain order.

To search in a graph systematically two basic algorithms exist: breadth-first and depth first. The different approaches are depicted in Fig. 3.7.

Breadth-First Searching

Within Breadth-First Searching (BFS), we consider for each node all its children, before we continue. As we do not want to visit nodes multiple



(a) Breadth-First (b) Depth-First

Figure 3.7: Two approaches to search a graph

times, we introduce the variable *COLOR*, that initially marks all nodes as unvisited. We then start traversing from the given node by visiting all children.

The algorithm is shown in Algorithm 1. First, the algorithm sets the color of all nodes to none (lines 2-3). Next, we start visiting our nodes, beginning from node a. We first color this node gray (line 4), indicating that we start to analyze it. We then add the node to the queue[1] Q (line 5). Now, we can start emptying the queue (line 6): we take the first element, which we name b, of the queue (line 7), and visit all its children c. If this c already has a color, we already added it to the queue, so we do not need to visit it anymore (line 9). If the node has no color, we set the color of c to gray (line 10), indicating that we add it to the queue Q (line 11). Once we have inspected all childrens of b, we are done with b, which is why we color the node black (line 12). We repeat this until the queue is empty: then all nodes have been visited.

Algorithm 1: Breadth-first Search (BFS)

input : $G = (V, E)$, node $a \in V$

```
1  Algorithm BFS (G,a)
2      foreach node x ∈ V do
3          COLOR(x) := none;
4      COLOR(a) := gray;
5      AddToQueue (a, Q);
6      while Q not empty do
7          b := GetAndRemoveFirst (Q);
8          foreach c ∈ b•_G do
9              if COLOR(c) = none then
10                 COLOR(c) := gray;
11                 AddToQueue (c, Q);
12         COLOR(b) :=black;
```

Consider again graph G depicted in Fig. 3.3. Let us start from node S. The steps are shown in Table 3.1. Initially, the queue is empty (step 0). We color node S gray, and add it to the queue (step 1). Next, we consider all children of S, i.e., $S^{\bullet}_{G} = \{A, C, E\}$. Each of these nodes have not been seen yet, so we add them to the queue (step 2). As the first element of the queue is A, we remove this node from the queue. Its children are $A^{\bullet}_{G} = \{B, C\}$. As C is already colored gray, we only add B to the queue. We are done with

[1] A queue can be seen as a waiting line in front of a desk: the first element will be served, whereas new elements have to join at the end of the line. We call this First-in-First-Out (FIFO) behaviour.

Table 3.1: BFS execution on graph G (Fig. 3.3)

Step	Node	Queue
0	S	ϵ
1		$\langle S \rangle$
2	S	$\langle A, C, E \rangle$
3	A	$\langle C, E, B \rangle$
4	C	$\langle E, B, D \rangle$
5	E	$\langle B, D, T \rangle$
6	B	$\langle D, T \rangle$
7	D	$\langle T \rangle$
8	T	ϵ

A, so we color it black. Now we go back to the queue. The first element of the queue is C, and we repeat the process until the queue is empty.

Depth-First Searching

Breadth-First Searching traverses through the graph by first considering all children. Another approach is to first traverse as deep as possible in the graph: diving as far as possible. We call this approach Depth-First Searching (DFS). The algorithm is shown in Algorithm 2. Similar to BFS, we color the nodes, to prevent visiting nodes multiple times. We first initialize the color for all nodes (line 3), after which we select some node a that is not yet colored. Initially all nodes are uncolored, thus any node can be picked (lines 5-6). On this node we apply the algorithm DFS-visit, which does the actual traversing.

The function DFS-visit is what we call a recursive function: within the function, it calls itself (line 11). How does the algorithm work? First, we color the selected node, a, gray (line 8), to indicate that we started with this node. Next, we select an uncolored child of a, which we call b (line 9-10), and apply DFS-visit on b (line 11). Once we have executed DFS-visit on all children of a, we color a black (line 12), indicating that we are finished with that node.

Why is this algorithm called depth first? Suppose we apply **DFS-visit** on some node a. The algorithm states that we pick some unvisited child node b, on which we need to call **DFS-visit** again. Thus, the algorithm first traverses all children of b before it continues with the next child of a. In this way, the algorithm first "dives" to a node that either has no children anymore, or to a node whose children all have been visited previously.

Notice the loop in lines 4-6. If DFS-visit ends, it might be that not yet all nodes have been colored, and thus not yet visited by the algorithm. The loop of lines 4-6 ensures that those nodes are visited as well!

Algorithm 2: Depth-first Search (DFS)

```
   input  : G = (V, E)
 1 Algorithm DFS (G)
 2     foreach a ∈ V do
 3         COLOR(a) := none
 4     foreach a ∈ V do
 5         if COLOR(a) = none then
 6             DFS-visit (G, a);
 7 Procedure DFS-visit (G, a)
 8     COLOR(a) := gray;
 9     foreach b ∈ a•_G do
10         if COLOR(b) = none then
11             DFS-visit (G, b);
12     COLOR(a) := black;
```

To apply the algorithm, we can use a stack [2] to represent the order in which we need to apply DFS-visit. The node remains on the stack until we color it black.

Let us apply the algorithm on the example graph G depicted in Fig. 3.3. The steps are shown in Table 3.2. We start the algorithm from S, which we color gray and place on top of the stack (step 0), as we call DFS-visit on it. Next, we pick some child of S, say A. We call DFS-visit on A, and thus add it to the stack (step 1). Now A is on top of the stack, thus we need to consider a child of A, say B. We apply DFS-visit on it, and thus add B to the stack (step 2). Now, we consider the children of B, which is only T, and add T to the stack (step 3). Node T has no children (step 4). Hence, we can color it black and remove it from the stack (step 5). Now node B is again on top of the stack (step 6). As it has no further children, we color it black, and remove it from the stack (step 7). Last element on the stack is node A. As node A also has node C as its child, and C is uncoloured, we apply DFS-visit on it, by putting C on top of the stack (step 8). Node C has unvisited node D as child, thus we add D to the stack (step 9). As node D has no uncoloured children, we are finished with D (step 10), color D black, and remove it from the stack (step 11).

Now, node C is on top of the stack, but it has no further unvisited children (step 12). Hence, we color it black and remove it from the stack (step 13). Again, node A is on top of the stack. As all children of A are

[2] Different from a queue, a stack is a sequence from which the last element on top of the stack is handled. New elements join at the end of the sequence. We call this Last-in-First-Out (LIFO) behaviour.

Table 3.2: Applying DFS on graph G (Fig. 3.3)

Step	Node	Stack
0		$\langle S \rangle$
1	S	$\langle S, A \rangle$
2	A	$\langle S, A, B \rangle$
3	B	$\langle S, A, B, T \rangle$
4	T	$\langle S, A, B, T \rangle$
5	T	$\langle S, A, B \rangle$
6	B	$\langle S, A, B \rangle$
7	B	$\langle S, A \rangle$
8	A	$\langle S, A, C \rangle$
9	C	$\langle S, A, C, D \rangle$
10	D	$\langle S, A, C, D \rangle$
11	D	$\langle S, A, C \rangle$
12	C	$\langle S, A, C \rangle$
13	C	$\langle S, A \rangle$
14	A	$\langle S, A \rangle$
15	A	$\langle S \rangle$
16	S	$\langle S, E \rangle$
17	E	$\langle S, E \rangle$
18	E	$\langle S \rangle$
19	S	$\langle S \rangle$
20	S	$\langle \rangle$

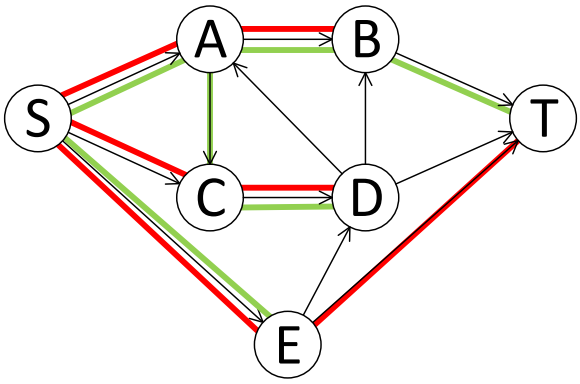


Figure 3.8: Spanning trees generated by BFS (red) and DFS (green)

colored (step 14), we color it black and remove it from the stack (step 15). We are back at S. The only uncolored child of S is node E, hence we add it to the stack (step 16). Visiting E reveals that we cannot visit any uncolored child anymore (step 17). Hence we color it black and remove it from the stack (step 18). We are back at S, which has no uncolored children anymore (step 19). Hence, we color it black, and remove it from the stack (step 20). As the stack is empty, we are back at the DFS-routine (line 6).

As all nodes are now colored, we can exit the foreach-loop (line 4-6), which terminates the DFS algorithm.

Both the BFS and DFS algorithms create a spanning tree of the nodes of a graph. However, they are complete different, as shown in Fig. 3.8 for the example graph.

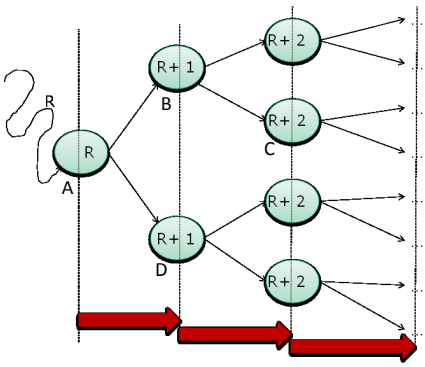


Figure 3.9: BFS indicates steps

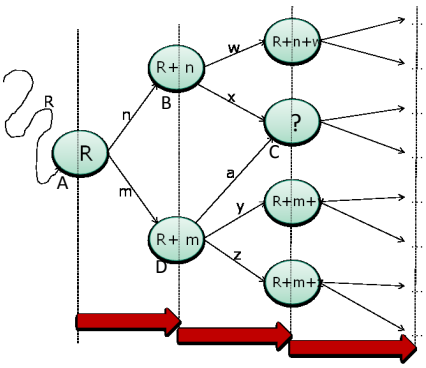


Figure 3.10: BFS to find the shortest path

3.3 Shortest Paths

Let us return to the example of the railways, as depicted in Fig. 3.1. We would like to answer the question "What is the fasted way to go from Nijmegen to Eindhoven?". To answer this question, we need to find the shortest path between two nodes.

How do we know the shortest number of steps from one node to some other node, in a structured way? Consider Fig. 3.9. Within each step of the BFS algorithm, we advance with each step of the algorithm. Thus, if we reach node A with R steps, we know that we can reach node B with $R+1$ steps, and thus node C with $R+2$ steps. As mathematicians like to generalize, we first add labels to the arcs that indicate some costs to take that arc. In other words, we make it a labelled graph, where the labels indicate the costs, as can be seen in Fig. 3.10. Now walking the path $\langle A, B, C\rangle$, the path has costs R (the costs to get at A) $+n$ (the costs to get from A to B) $+x$ (the costs to get from B to C). Now suppose there is an additional path from A to C via D, i.e., the path $\langle A, D, C\rangle$, with costs $R+m+a$. What are the costs of getting from A to C? We thus need to compare $R+n+x$ and $R+n+a$. If the latter is the smallest, the shortest path is $\langle A, B, C\rangle$, otherwise it is the former, $\langle A, D, C\rangle$.

This is exactly how Dijkstra's shortest path algorithm is working. The full algorithm is shown in Algorithm 3. It roughly follows the same structure as the earlier presented BFS algorithm. Initially, we set the distance and parent of each node to infinity (∞), and nothing ($\perp$), respectively (lines 2-4). The algorithm starts from node a. Going from a to a has distance 0, which we set in line 5. Instead of coloring the visited nodes, we keep a list Q that contains all the nodes we remain to visit (line 6). While this list Q is not empty, we need to visit nodes. We always choose the node with smallest distance (line 8). The function FindEltWithSmallestDistance

finds the element with the smallest distance, and returns this element after removing it from the list. We call this element b. Next, the algorithm checks each neighbour c of b. As we have a labelled graph, the arc is labelled with the distance, i.e., (b, d, c) is the arc from b to c with distance d (line 9). If the distance going to c from b is smaller than the distance we already have for c, we update the distance and parent of c (lines 11-12).

Algorithm 3: Dijkstra's shortest path

input : $G = (V, E, \lambda)$, node $a \in V$

```
1  Algorithm Dijkstra (G,a)
2      foreach node x ∈ V do
3          DISTANCE(x) := ∞;
4          PARENT(x) := ⊥;
5      DISTANCE(a) := 0;
6      Q := V;
7      while Q not empty do
8          b = FindEltWithSmallestDistance (Q, DISTANCE);
           /* Return element with smallest distance in Q    */
           /* And removes it                                */
9          foreach (b, d, c) ∈ E do
               /* (b, c) is the arc, with weight d          */
10             if DISTANCE(b) + d < DISTANCE(c) then
11                 DISTANCE(c) = DISTANCE(b) + d;
12                 PARENT(c) = b;
```

Consider the graph depicted in Fig. 3.11. We want to calculate the shortest path from S to T. The steps of the algorithm are shown in Table 3.3. The algorithm initializes all distances to infinity, and sets the distance of S to 0 (step 0). As node S has the smallest distance, the algorithm picks this node (indicated by the bold number in the table). From S, traversing to A has costs 7, traversing to C has costs 1, and traversing to E has costs 3 (step 1). Now, the unvisited node with shortest distance is node C. Hence, we choose this node as our next step. From C we traverse to D with costs $1+3 = 4$, and to E with costs $1+1 = 2$. We update the distances accordingly (step 2). The node with shortest distance is node E (with distance 2). From this node, traversing to D has costs 3, which is smaller than the original costs of D, so we update the value of D (step 3). Similarly, we proceed to node A (step 4), and from node A to node B (step 5). From node B, we reach node T with distance 9 (step 6), after which all nodes have been visited.

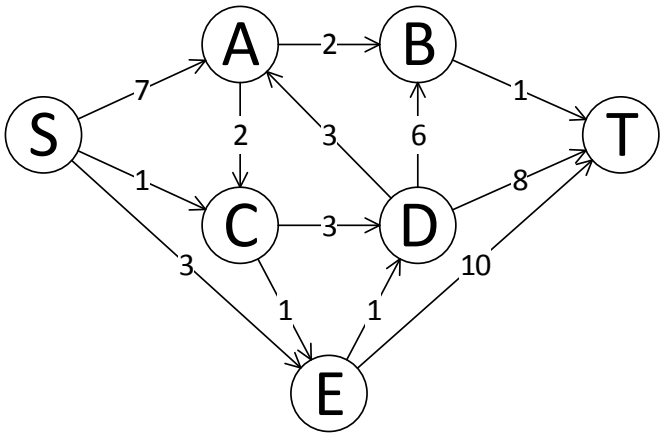


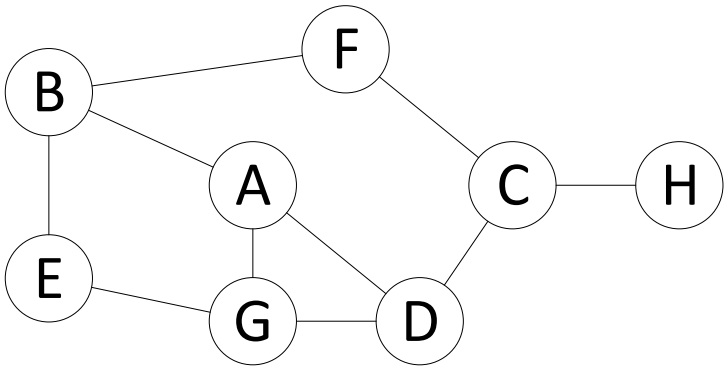
Figure 3.11: Example Weighted Graph

Table 3.3: Dijkstra's algorithm on Graph of Fig. 3.11

Step	**Node**	**Parent**	**S**	**A**	**B**	**C**	**D**	**E**	**T**
0		$\perp$	**0**	∞	∞	∞	∞	∞	∞
1	S	$\perp$		7	∞	**1**	∞	3	∞
				S		S		S	
2	C	S		7	∞		4	**2**	∞
				S			C	C	
3	E	C		7	∞		**3**		12
				S			E		E
4	D	E		**6**	9				11
				D	D				D
5	A	D			**8**				11
					D				D
6	B	A							**9**
									B

3.4 Exercises

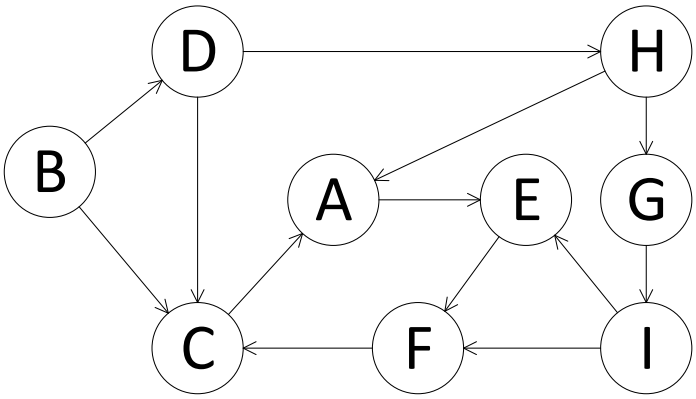
Exercise 3.1. *Formalize this undirected graph in terms of $G = (V, A)$:*



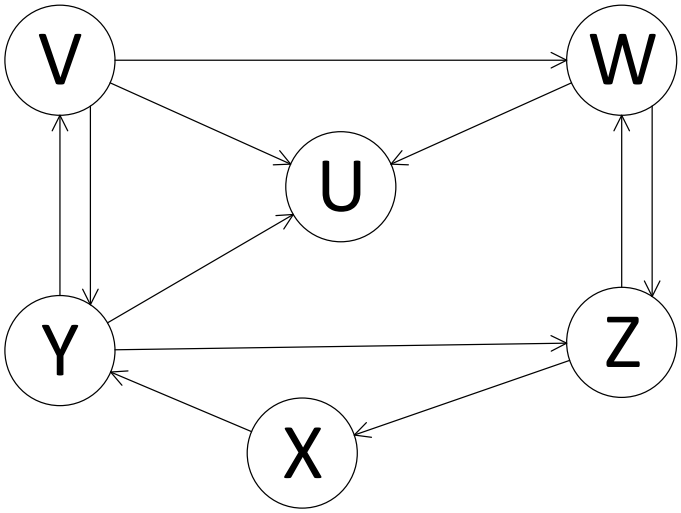
Exercise 3.2. *Draw the following graph:*

$$\begin{aligned} V &= \{ \quad A, B, C, D, E, F \ \} \\ A &= \{ \quad (A,B), (A,F), (B,F), (A,C), (B,D), (C,F), (E,C), \\ &\qquad (C,A), (D,B), (C,D), (D,E) \} \end{aligned}$$

Exercise 3.3. *Formalize this graph in terms of $G = (V, A)$:*



Exercise 3.4. *Given is the following graph:*



Is this graph isomorphic to the graph of Exercise 3.2? If so, provide a bijective function and show that the graphs are isomorphic. If not, provide a reason why not.

Exercise 3.5. *Calculate the BFS for the undirected graph of Exercise 3.1. Start at node A.*

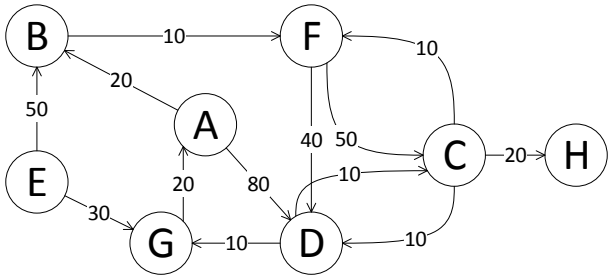
Exercise 3.6. *Calculate the DFS for the undirected graph of Exercise 3.1. Start at node A.*

Exercise 3.7. *Calculate the BFS for the directed graph of Exercise 3.3. Start at node A.*

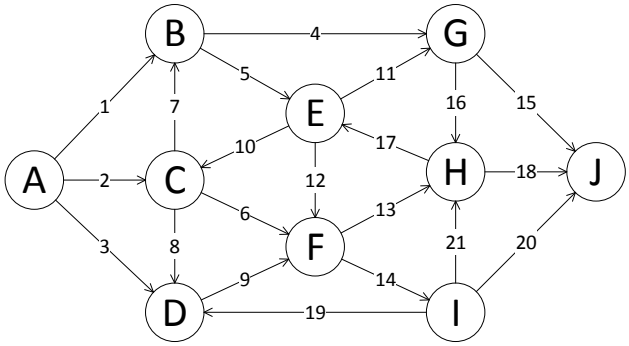
Exercise 3.8. *Calculate the DFS for the directed graph of Exercise 3.3. Start at node A.*

Exercise 3.9. *Explain why a topological sort is only possible on acyclic graphs.*

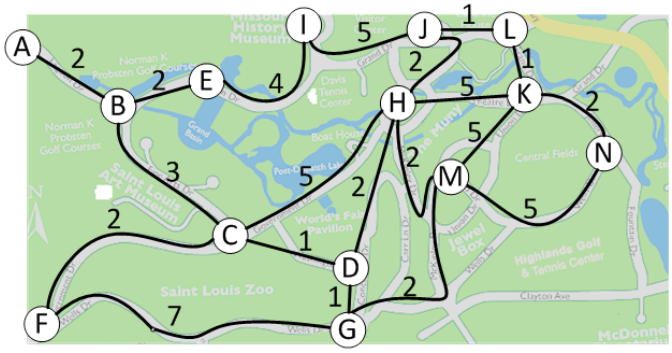
Exercise 3.10. *Calculate using Dijkstra's algorithm the shortest path from A to G for the following graph:*



Exercise 3.11. *Calculate using Dijkstra's algorithm the shortest path from A to J for the following graph:*



Exercise 3.12. *Calculate using Dijkstra's algorithm the shortest path from A to N for the following graph:*



4

Modeling with Transition Systems

Consider the following situation: a pedestrian crossing is located at a busy street. To ensure that pedestrians can cross safely, the local government wants to place traffic lights. You are asked to design a safe system with the following characteristics:

R-1 The traffic light for cars should now and then turn red to ensure cars will not drive too fast;

R-2 To cross, a pedestrian needs to push a button, after which in some time the pedestrian light becomes green;

R-3 Both traffic lights should be red before one of the lights becomes green again.

You are not only asked to design the system, but also to show that the system indeed satisfies the above properties. How will you proceed?

One way to look at a system is from a behavioural perspective: a system is in some state, and from time to time it switches state, either due to some internal action, or an external action. An example is a Dutch traffic light: it is either in a state with a red light, a yellow light, or a green light. If the traffic light is red, it can become green, after some time it turns yellow and back to red. To model this behavior, we can use graphs: the nodes represent the states, and the arcs represent transitions from one state to the other. The Dutch traffic light represented as a graph is shown in Fig. 4.1. This is what we call a transition system. The dangling arrow pointing at the node "Red", indicates that this is the *initial state* of the traffic light: we always start the system in this initial state. This node "Red" also has a double-lined border. This indicates that the state is a *final state*. The final states of a transition system indicate "happy states": in these states we may stop the system.

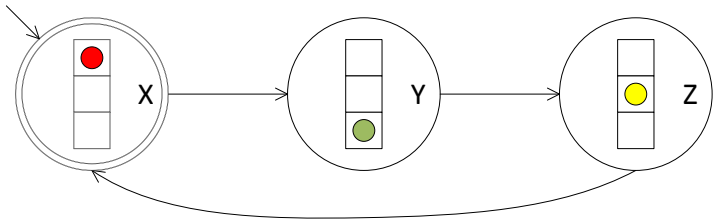


Figure 4.1: Dutch traffic light modelled as transition system T

Modeling the pedestrian situation starts with identifying the states: what should the state represent in order to model the required system? In the case of a simple traffic light, the states represent the different colors. One way to identify the different states, is to list the objects the system should model, and the states these can be in. In our pedestrian situation, we can identify the following objects:

- Traffic light for cars. States: Red, Yellow, Green;
- Traffic light for pedestrians. States: Red, Green;
- Button for pedestrians. States: Pushed, Unpushed;

Every state in our system should state something about these objects. The set of all possible states is thus the Cartesian product of the states of these objects. Naming the set of possible states of the traffic light for cars C, we get $C = \{R, Y, G\}$. Similarly, we obtain $P = \{R, G\}$ for the pedestrian light, and $B = \{U, P\}$ for the button. The possible states of our system thus becomes: $S = C \times P \times B$. In other words, each state is a tuple of 3 elements: the traffic light for cars, the pedestrian light, and the button. As an example, the state (R, G, U) means that the cars have a red light, the pedestrians green, and the button is unpushed.

To model the system, we first need to identify the initial state. A state that satisfies the above conditions is the state in which both traffic lights are red: no car nor pedestrian may pass. For the button, let us assume there are no pedestrians, and hence the button is unpushed. The initial state thus becomes (R, R, U). Now what are the possible actions in this state? The cars can get green, or some pedestrian may "push" the button. Let us first consider the latter. What will happen if the pedestrian pushes the button? The pedestrian needs to cross, and as the traffic light for cars is initially red, pedestrians can directly get green, and there is no need to maintain that the button is pushed, thus leading to the state (R, G, U). In the former case, we go to the state (G, R, U), from which we can go to (Y, R, U) and back to (R, R, U). This is the default loop for the cars. What happens if we push

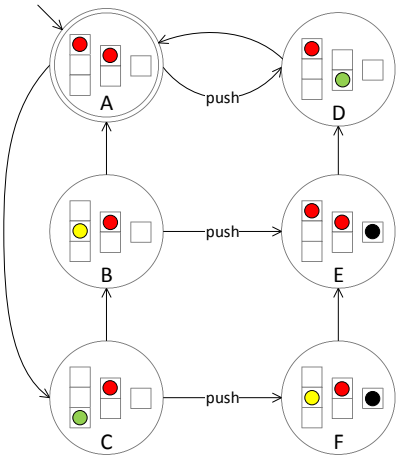


Figure 4.2: The Pedestrian System PS_0

the button in state (G, R, U)? The traffic light should become red, but it first needs to become yellow. Thus, we add a "push" arc from (G, R, U) to the new state (Y, R, P), indicating that the traffic light is yellow, and the button is pushed. From this state, we need to go to the state in which both traffic lights are red, and the button is pushed. This latter is important, as we cannot simply return to the (R, R, U) state: in this state, the cars can get green, whereas we want the pedestrian light to become green! Hence, we add a new state (R, R, P), and an arrow from (Y, R, P) to (R, R, P). In this state, the pedestrian light may turn green: (R, G, U). From this state, we return to the state (R, R, U). Lastly, we need to consider what happens if we push the button when we are in state (Y, R, U). The light needs to become red, so that pedestrians can get green. That is the already existing state (R, R, P). In this way, we obtain the pedestrian system as depicted in Fig. 4.2.

Is this solution correct? For this, we need to check the above conditions. The first condition states that the traffic light for cars should always be able to loop. Is it always possible in any state in which both traffic lights are red that the cars obtain green? This is a reachability problem, that we can solve using the BFS algorithm explained in Sect. 3.2. We execute for each node where both traffic lights are red the algorithm and check whether there is a state reachable in which the traffic light for cars is green. Similarly, we can check the second condition: execute the BFS algorithm for each state in which the button is pushed whether the pedestrians will receive a green light. The third condition is harder to check. It is a path property, stating that from any path in which one of the lights is green to a state in which

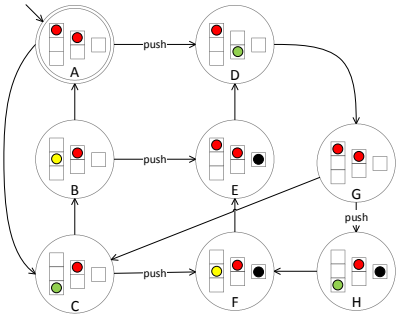


Figure 4.3: Alternative PS_1

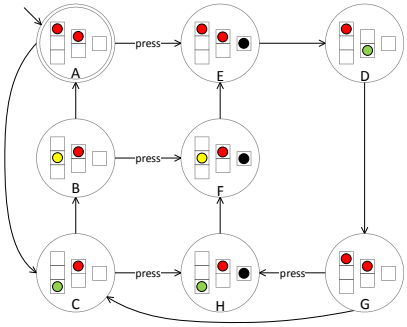


Figure 4.4: Alternative PS_2

the other light is green, a state is passed in which both lights are red. In other words, we cannot have a transition directly leading from $(G, R, \cdot)$ to $(R, G, \cdot)$ [1]. Any path between these states should contain a state $(R, R, \cdot)$. In our system, all conditions are satisfied, so it is a correct solution!

Is this the only possible solution? For this, we need to check whether there are alternatives. For example, we chose that if some pedestrian pushes the button in the state (R, R, U) the system directly can move to the state (R, G, U), i.e., the pedestrian directly gets green. An alternative is in the state where cars have green: (G, R, U). In the first presented solution, the system moves to state (G, R, P), indicating the cars still have green, and the button is pushed. An alternative would be that the system moves to (Y, R, P): the cars directly get yellow. This is modelled in Fig. 4.3. Another alternative would be that the system first goes to a state (R, R, P), and only then moves to (R, G, U). This alternative models that it may take some time for the pedestrian light to turn green. This gives the alternative depicted in Fig. 4.4.

Many models possible! Each of these models explain the same situation, but made different choices. Therefore, we always need to write down our design decisions: what were the reasons to model state transitions? A possible explanation for the first presented situation and the first alternative is that we do not want pedestrians to wait too long, as they will cross anyhow, as they are pedestrians... A possible explanation of the second alternative is that we do not know how much time the system needs to give the pedestrian green, and that the model is completely symmetric.

Important to realize is that all alternatives are correct solutions. There is no single solution. That is the beauty of modelling. The down side is

[1] With the symbol "$\cdot$", we indicate that this element in the tuple may have an arbitrary value

that this also makes it difficult to analyze whether you have constructed a correct model. That is the challenge of modelling. It takes time and practice to learn to evaluate alternatives, and choose a best solution for the circumstances.

To model a complete system, we should for each state ask ourselves the question "what can happen in this state?". This leads either to the creation of new states, or to already existing states.

In the remainder of this chapter, we will provide means to learn to better understand your model, to analyze whether it is a correct solution, and a way to compare different models. In the last section, we return to modelling approaches.

4.1 Mathematical Definition

A labelled transition system (LTS) consists of a set of *states* and *transitions* between states labelled by actions from a set of action labels. The set of states are the vertices of its graph, and the transitions are the arcs of the graph. From the outside, only the action labels $\mathcal{A}$ are visible. A special action is the silent action, denoted by τ. The silent action, also called a τ-step, is not an element of the set of action labels. Different from the action labels in $\mathcal{A}$, the silent action is not visible from the outside.

An example transition system is depicted in Fig. 4.5. This transition system L contains 7 states, 5 visible actions, and 2 silent actions. The transition system has one initial state d and two final states a, f. Formalized, we obtain the transition system $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ with:

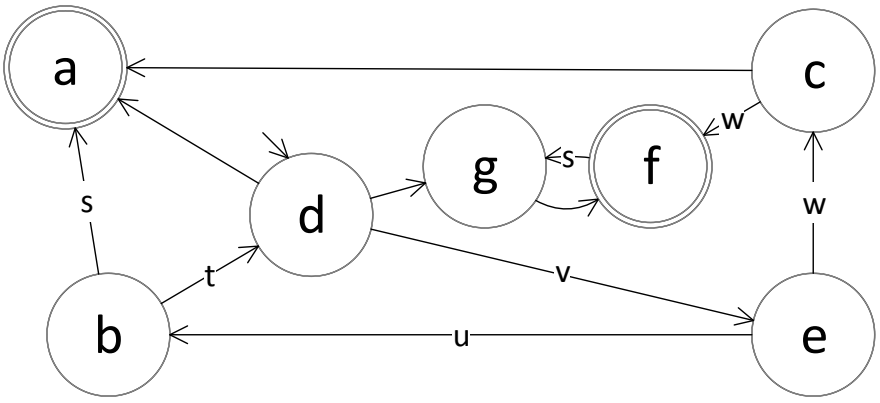


Figure 4.5: Example transition system L

$$
\begin{aligned}
S &= \{ \quad a, b, c, d, e, f, g \quad \} \\
\mathcal{A} &= \{ \quad s, t, u, v, w \quad \} \\
\rightarrow &= \{ \quad (b, s, a), (b, t, d), (d, \tau, a), (d, v, e), (d, \tau, g), (e, u, b), \\
&\qquad (e, w, c), (c, \tau, a), (c, w, f), (g, \tau, f), (f, s, g) \quad \} \\
s_i &= \quad d \\
\Omega &= \{ \quad a, f \quad \}
\end{aligned}
$$

Definition 4.1 (Labelled Transition System)**.** *A* labelled transition system *(LTS) is a 5-tuple* $(S, \mathcal{A}, \rightarrow, s_0, \Omega)$ *where*

- S *is a set of* states*;*
- $\mathcal{A}$ *is a set of* actions*;*
- $\rightarrow \subseteq (S \times (\mathcal{A} \cup \{\tau\}) \times S)$ *is a* transition relation*, where* $\tau \notin \mathcal{A}$ *is the silent action.*
- $(S, \rightarrow, \emptyset)$ *is a labelled graph, called the* reachability graph*;*
- $s_0 \in S$ *is the* initial state*;*
- $\Omega \subseteq S$ *is the set of* final states*.*

A final state is often referred to as an accepting state*.*
Its induced graph *is the labeled graph* $(S, \rightarrow, \emptyset)$*.*

4.2 Behavior of a Transition System

Now that we have defined the syntax of transition systems, we can define the behavior of a system. We do this in two steps: we first discuss single steps in a system, and then continue with sequences of actions.

Actions

If a transition is in some state, we can execute one of its outgoing transitions. If the transition is labeled with some visible action, we say that the system executes that action, leading to a new state. In case the transition is labeled with a silent action, i.e., τ, the system moves invisibly to that new state. If a state has no outgoing transitions, we call the state a deadlock. We formalize this as follows:

Definition 4.2. *Let* $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ *be an LTS, and let* $s, s' \in S$ *be two states of it, and* $a \in \mathcal{A} \cup \{\tau\}$ *some action. We use the following notions:*

- *we write* $(L : s \xrightarrow{a} s')$ *if and only if* $(s, a, s') \in \rightarrow$

- *An action* $a \in \mathcal{A} \cup \{\tau\}$ *is called* enabled *in a state* $s \in S$*, denoted by* $(L : s \xrightarrow{a})$*, if there exists a state* s' *such that* $(L : s \xrightarrow{a} s')$*.*

- *If* $(L : s \xrightarrow{a} s')$*, we say that state* s' *is* reachable *from* s *by an action labelled* a*.*

- *A state* $s \in S$ *is called a* deadlock *if no action* $a \in \mathcal{A} \cup \{\tau\}$ *exists such that* $(L : s \xrightarrow{a})$*.*

- *A set of states* $U \subseteq S$ *is called a* livelock *if from* U *only states within* U *itself can be reached, i.e., for all* $s \in U$*,* $a \in \mathcal{A}$ *and* $s' \in S$ *such that* $(L : s \xrightarrow{a} s')$*, then* $s' \in U$*.*

In our example transition system L, we have one deadlock: state A, as it does not have any outgoing transitions. In state d we have three enabled transitions, of which one is visible, labeled with action v, and the other two are invisible. Thus, we have $(L : d \xrightarrow{v} e)$,$(L : d \rightarrow a)$, and $(L : d \rightarrow g)$. There is one livelock in L: $\{f, g\}$. All transitions from these states remain in this set of states!

In our example, there are several silent steps. For example, from state d, we can follow two silent transitions, followed by the visible action s. As the first two transitions are silent, we cannot *observe* them. We do not know whether we are in state d, a, g or f, as long as we do not observe any visible action. Only after observing the visible action s or v, we know the origin of that action. Thus, the transition system could move internally, without any observation. For this, we introduce the $\Longrightarrow$ relation.

Definition 4.3. *Let* $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ *be an LTS, and let* $s, s' \in S$ *be two states of it. We use the following notions:*

- *We define* $\Longrightarrow$ *as the smallest relation such that* $(L : s \Longrightarrow s')$ *if* $s = s'$ *or* $\exists s'' \in S : (L : s \Longrightarrow s'' \xrightarrow{\tau} s')$*. As a notational convention, we may write* $\xRightarrow{\tau}$ *for* $\Longrightarrow$*.*

- *For* $a \in \mathcal{A}$*, we define* $\xRightarrow{a}$ *as the smallest relation such that* $(L : s \xRightarrow{a} s')$ *if* $\exists s_1, s_2 \in S : (L : s \Longrightarrow s_1 \xrightarrow{a} s_2 \Longrightarrow s')$*.*

In our example transition system L, we have $(L : d \xRightarrow{v} e)$, $(L : d \Longrightarrow d)$, $(L : d \Longrightarrow a)$, and $(L : d \Longrightarrow f)$. In addition, we have $(L : d \xRightarrow{s} g)$.

Sequences

We are not only interested in which actions can be taken from a certain state, but also how we can reach that state from some other state. For that, we might need to take several steps. For this, we use a sequence of steps. In our example transition system L, we might want to execute actions v, u, t, v, and finally action w. This is a sequence of actions! We therefore lift the notation of actions to sequences:

Definition 4.4. *Let $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ be an LTS, and let $s_0, s_n \in S$ be two states of it. We use the following notions:*

- *For the empty sequence ϵ, we have $(L : s \xrightarrow{\epsilon} s')$ if and only if $(L : s \Longrightarrow s')$.*
- *Let $\sigma \in \mathcal{A}^*$ be a sequence of length $n > 0$, and let $s_0, s_n \in S$. Sequence σ is a* firing sequence*, denoted by $(L : s_0 \xrightarrow{\sigma} s_n)$, if there exist states $s_{i-1}, s_i \in S$ such that $(L : s_{i-1} \overset{\sigma(i)}{\Longrightarrow} s_i)$ for all $1 \leq i \leq n$.*
- *We write $(L : s_0 \xrightarrow{*} s_n)$ if there exists a sequence $\sigma \in \mathcal{A}^*$ such that $(L : s_0 \xrightarrow{\sigma} s_n)$*
- *If $(L : s_0 \xrightarrow{*} s_n)$, we say s_n is* reachable *from s_0*
- *A firing sequence $\sigma \in \mathcal{A}^*$ from state $s \in S$ is an* accepting sequence *for s if there exists a final state $s_f \in \Omega$ such that $(L : s \xrightarrow{\sigma} s_f)$.*

In our example transition system, we have $(L : d \xrightarrow{\langle v,u,t,v,w \rangle} c)$. Notice that this firing sequence is accepting as well. Since $(L : c \Longrightarrow a)$, we have $(L : d \xrightarrow{\langle v,u,t,v,w \rangle} a)$ as well, and a is a final state!

Definition 4.5 (Reachable states, language of an LTS)**.** *Let $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ be an LTS. The reachable states from a state $s \in S$ is the set $\mathcal{R}(L, s) = \{s' \mid (L : s \xrightarrow{*} s')\}$. The set of all firing sequences from state s_i is denoted by $\mathcal{T}(L) = \{\sigma \mid \exists s \in S : (L : s_i \xrightarrow{\sigma} s)\}$. The language of the LTS L, denoted by $\mathcal{L}(L) \subseteq \mathcal{A}^*$, is the set of accepting sequences, i.e. $\mathcal{L}(L) = \{\sigma \in \mathcal{A}^* \mid \exists s_f \in \Omega : (L : s_i \xrightarrow{\sigma} s_f)\}$.*

In our example transition system L, we have $\langle v, u, t, v, w \rangle \in \mathcal{L}(L)$. Notice that L has an infinite language, as its induced graph is cyclic. We have for example $\mathcal{R}(L, d) = \{a, b, c, e, f, g\}$, $\mathcal{R}(L, a) = \emptyset$ and $\mathcal{R}(L, g) = \{f, g\}$.

4.3 Comparing Systems

As we have seen in the introduction, a single problem may have many different solutions. However, we would like to compare these solutions to analyze whether they behave equivalent, or where they deviate.

Consider the pedestrian system introduced in this chapter. Condition R-1 states that the system for cars should behave the same as the traffic light depicted in Fig. 4.1. However, the pedestrian system has an action *push*, whereas the traffic light of Fig. 4.1 does not have such action. Or it might be that this action is called *push* in one system, and *press* in some other system. For this purpose, we introduce two operators: renaming and hiding.

Definition 4.6 (Renaming, Hiding)**.** *Let $\mathcal{A}$ and $\mathcal{A}'$ be two sets. Let $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ be an LTS. Let $r : \mathcal{A} \rightarrow (\mathcal{A}' \cup \{\tau\})$ be a function. We define the operation ρ_r on an LTS by $\rho_r(L) = (S, \mathcal{A}', \rightarrow', s_i, \Omega)$, where for $s, s' \in S$ and $a \in \mathcal{A}$ we have $(s, R(a), s') \in \rightarrow'$ if and only if $(s, a, s') \in \rightarrow$. A special case of renaming is* hiding*, in which actions are either renamed to τ, or remain identical. We define the hide operation on a subset $H \subseteq \mathcal{A}$, denoted by $\tau_H(L)$, by $\tau_H(L) = \rho_h(L)$, where $h(a) = \tau$ for all $a \in H$ and $h(a) = a$ otherwise.*

Thus, to be able to compare the first alternative with the second, we need to rename action *push* into action *press*. For this, we create the function $r = \{(\text{push}, \text{press})\}$. To compare each of the solutions with the traffic light, we need to hide the actions *push* and *press*, thus, we compare the traffic light with $\tau_{\{\text{push,press}\}}(PS)$, where PS stands for the pedestrian system under investigation.

Language Equivalence

A first notion of equivalent behaviour is *language equivalence*. Two LTSs are *language equivalent* if the languages are identical.

Definition 4.7 (Language equivalence)**.** *Let L and L' be two LTSs. They are* language equivalent *if $\mathcal{L}(L) = \mathcal{L}(L')$.*

In our example, the three alternative pedestrian systems all have the same language: $\mathcal{L}(PS_0) = \mathcal{L}(PS_1) = \{\text{push}\}^*$ and $\mathcal{L}(PS_2) = \{\text{press}\}^*$. Using the above renaming function r, we obtain:

$$\mathcal{L}(PS_0) = \mathcal{L}(PS_1) = \mathcal{L}(\rho_r(PS_2))$$

Simulation equivalence

Language equivalence only states that two LTSs should accept the same traces, it has no requirements on the intermediate states. For this we introduce several notions of equivalence.

Strong Bisimulation

The strongest notion of equivalence of two LTSs is to check for isomorphism on their reachability graphs. However, this notion is often too strong.

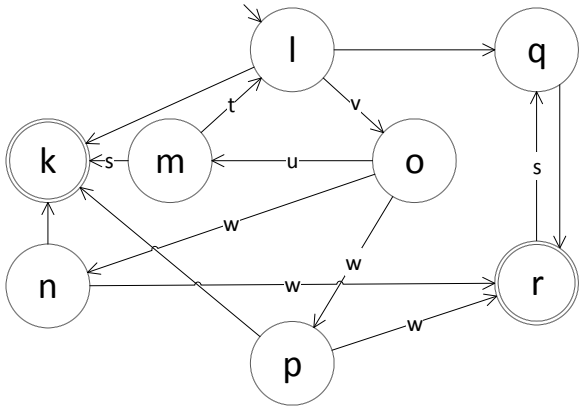


Figure 4.6: Transition system L' strongly bisimilar to Fig. 4.5

Therefore, we introduce the notion of *strong simulation*. An LTS L' strongly simulates an LTS L if in any two related states, any action L can do, LTS L' can perform as well. If both L' strongly simulates L and L strongly simulates L', we say L and L' are strongly bisimilar.

Definition 4.8 (Strong (bi)simulation)**.** *Let $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$ and $L' = (S', \mathcal{A}', \rightarrow', s_i', \Omega')$ be two LTSs. The relation $Q \subseteq S \times S'$ is a* strong simulation*, denoted by $L \ {}_s\preceq_Q \ L'$, if:*

1. *the initial states are related:*
   $(s_i, s_i') \in Q$*;*
2. *the final states are related:*
   $\forall \bar{s} \in S', s_f \in \Omega : (s_f, \bar{s}) \in Q \implies \bar{s} \in \Omega'$*; and*
3. *related states reach related states:*
   $\forall s, s' \in S, a \in \mathcal{A} \cup \{\tau\}, \bar{s} \in S' : ((L : s \xrightarrow{a} s') \wedge (s, \bar{s}) \in Q) \implies (\exists \bar{s}' \in S' : (L' : \bar{s} \xrightarrow{a} \bar{s}') \wedge (s', \bar{s}') \in Q)$.

If both Q and Q^{-1} are strong simulations, Q is a strong bisimulation*, denoted by $L \ {}_s\simeq_Q \ L'$.*

Clearly, the different pedestrian systems are not strongly similar. For example, state D in the first alternative, PS_1, reaches state G, whereas in PS_0, state D reaches state A.

Consider the example transition system L' depicted in Fig. 4.6. Simulates this transition system our example transition system L? To check this, we first need to construct a relation between the different states. Let us consider the following relation: $Q = \{(a, k), (b, m), (c, p), (d, l), (e, n), (e, o), (f, r), (g, q)\}$. To check for simulation, we need to check all conditions

for strong simulation. First, are the inital states related? We have for the initial state d that $(d, l) \in Q$, and l is the initial state of L'. Check!

Now, the second criterion, is each final state related? We have k and r as the only states of L' that are related to the final states of L, and these are the final states of L', so check!

Lastly, consider the third criterion. For each state, we need to check for each state of L that the related states in L' have (a) the same actions and (b) these actions lead to a related state. Thus, for state d, we have two silent actions leading to a and g, and one visible action v leading to state e. State a relates to state k, and $(L' : l \to k)$. Similarly, $(L' : l \to q)$ and $(g, q) \in Q$, and for the visible action we have $(L' : l \xrightarrow{v} o)$ and $(e, o) \in Q$. We repeat this for all states, leading to the conclusion that L' simulates L. Similarly, we can check that L simulates L', and thus that the two are strongly bisimilar.

Important to see is that we can typically construct a simulation relation by starting at the initial state, and then work out the relation for each next reachable state. This requires creativity from the analysist, and thus practice!

For a more elaborate overview of simulation relations, we refer the reader to [5].

4.4 Combining Systems

We return to our example of the pedestrian system. We started by observing that each state in the system consists of the state of three separate objects: the traffic light for cars with states C, the traffic light for the pedestrians with states P, and the button with states B. We then considered for each possible state the possible actions, which resulted in the three alternatives depicted in Fig. 4.2, Fig. 4.3 and Fig. 4.4.

As part of the specification, we could have drawn separate transition systems for each of the elements, as shown in Fig. 4.7. We label all actions uniquely.

For each of the different objects, we can ask ourselves, "what are the possible actions we allow of the other objects?". Let us do so. In state X, the pedestrian light can turn green, and the button can be pushed. We thus add a self-loop to that state for d, e, f and g. In state Y and Z, we do not want the traffic light to turn, only the button can be pressed. We thus add a self-loop transition labeled with f. For the pedestrian light, if the light is red, the traffic light may loop, and the button may be pushed. Once the traffic light is green, only the button can be "unpushed", i.e., we add a self-loop with label g. For the button, if the button is unpushed, the traffic light for cars may loop. Only if the button is pushed (state U), we allow the pedestrian light to turn green. Additionally, if the button is pushed, we do

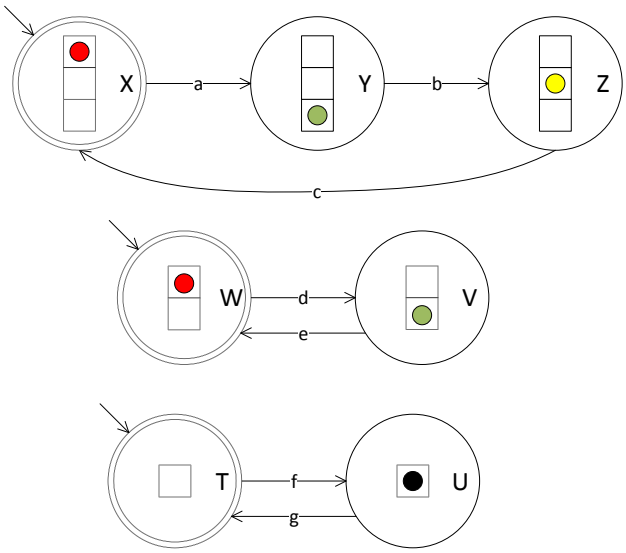


Figure 4.7: The separate objects of the Pedestrian System: (from top to bottom) car Light, Pedestrian light and Button object

not want that the traffic light could loop, so we do not add a to the set of possible actions in U. An important observation is that if the light is green, we do not want the button to be pushed, only once the pedestrian light becomes red again. Thus, the transitions e and g are actually the same. So, we label them the same (e). This results in the systems depicted in Fig. 4.8. Notice that the self-loops in fact represent multiple arcs (one arc for each action).

In the remainder of this section, we show how we can combine these indivual systems into a large system that results in a correct solution.

For this, we introduce the *synchronous product*. The synchronous product is again a transition system. Its set of states is the cartesian product of the constituent transition systems. Similarly, the initial state and final states are created by taking the cartesian product of the respective states of the consistuents. An action in the synchronous product is only allowed if that action can be executed by all its consituents. This results in the following definition:

Definition 4.9 (Synchronous product)**.** *Consider a set of N LTSs, say $L_1 = (S_1, \mathcal{A}_1, \rightarrow_1, s_1^i, \Omega_1)$, ..., $L_N = (S_N, \mathcal{A}_N, \rightarrow_N, s_N^i, \Omega_N)$. Their* synchronous

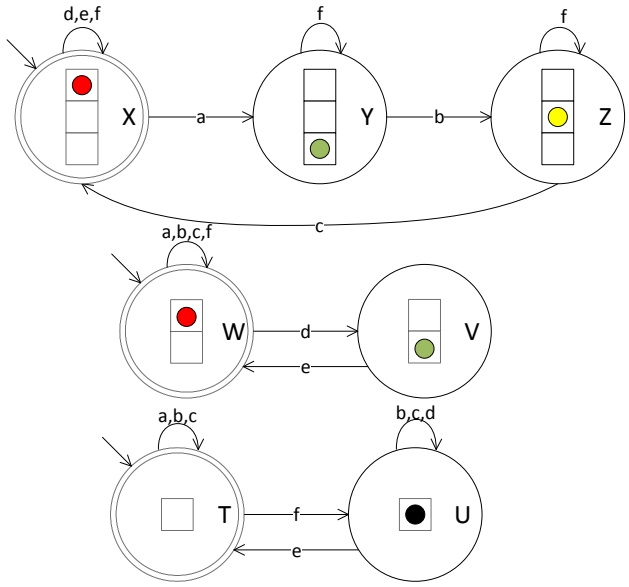


Figure 4.8: The separate objects of the Pedestrian System: (from top to bottom) car Light, Pedestrian light and Button object

product, *denoted by* $L_1 \times \ldots \times L_N$, *is the LTS* $(S, \mathcal{A}, \rightarrow s^i, \Omega)$ *with:*

$$\begin{array}{rcl} S & = & S_1 \times \ldots \times S_n \\ \mathcal{A} & = & \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n \\ \rightarrow & = & \{(s, a, s') \mid (L_i : \pi_i(s) \xrightarrow{a} \pi_i(s')) \textit{ for all } 1 \leq i \leq N\} \\ s^i & = & \{s_1^i\} \times \ldots \times \{s_N^i\} \\ \Omega & = & \Omega_1 \times \ldots \times \Omega_N \end{array}$$

Let us apply this on our pedestrian example depicted in Fig. 4.8. We start in the initial state, (X, W, T). In each of the constituents (being X, W and T), actions a and f are in common. Hence, these are the only two actions that leave state X. Firing action a results in the state (Y, W, T); firing f results in (X, W, U). Similarly, we go through each of the states, resulting in the transition system depicted in Fig. 4.8. We leave it to the reader to show that this system is delay bisimilar to system PS_2.

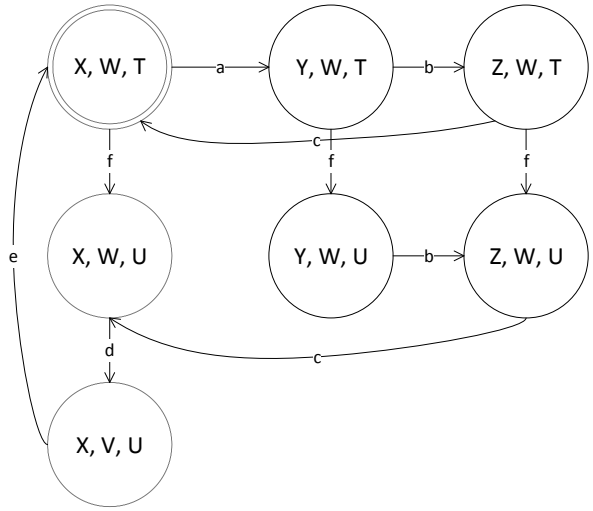


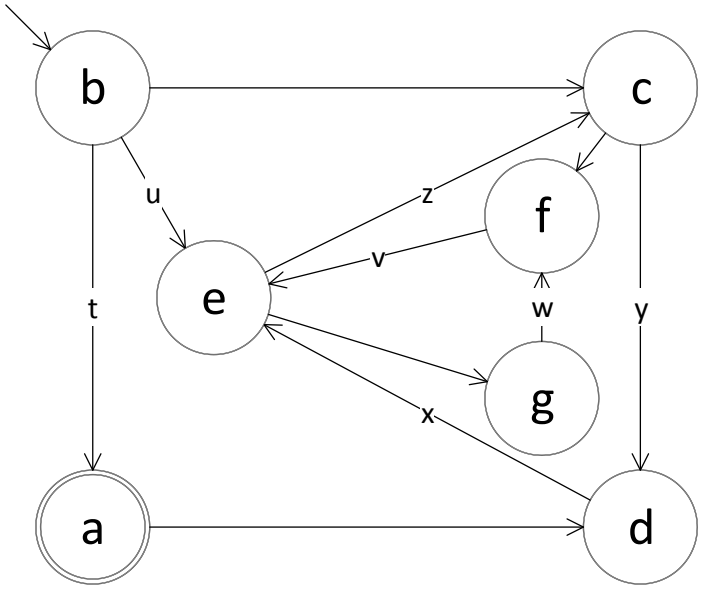
Figure 4.9: The synchronous product of the separate objects of the Pedestrian system: PS_3

4.5 Exercises

Exercise 4.1. *Explain why the $\Longrightarrow$ relation is reflexive.*

Exercise 4.2. *Given is some transition system $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$. Explain why $\rightarrow \subseteq \Longrightarrow$.*

Exercise 4.3. *Given is the following transition system:*



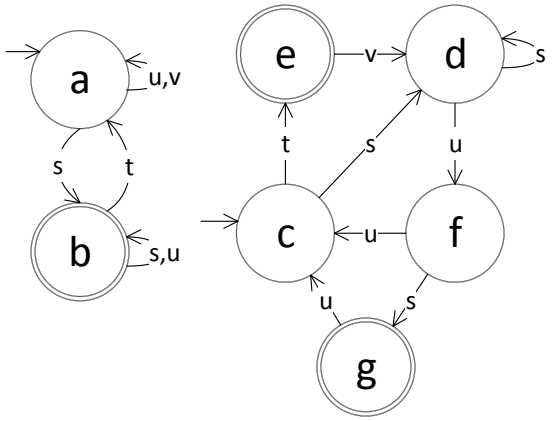
a. *Formalize this transition system in $L = (S, \mathcal{A}, \rightarrow, s_i, \Omega)$.*

b. *What is the language of this transition system?*

c. *Does this transition system have deadlocks? If so, which?*

d. *Does this transition system contain a livelock? If so, give the livelock, and show that it is a livelock. If not, explain why it does not contain any livelock.*

Exercise 4.4. *Create the synchronous product of the following two transition systems:*



Exercise 4.5. *Given is the following transition system T:*

$$
\begin{aligned}
S &= \{\ K, L, M, N, O, P, Q\ \} \\
\mathcal{A} &= \{\ t, u, v, w, x, y, z\ \} \\
\rightarrow &= \{\ (K, \tau, N), (N, \tau, O), (O, x, P), (L, t, K), (L, u, P), \\
&\quad\ (Q, v, P), (P, z, M), (L, \tau, M), (M, y, O), (P, w, Q)\ \} \\
s_i &= L \\
\Omega &= \{K\}
\end{aligned}
$$

Transition System L is the transition system of Exercise 4.3.

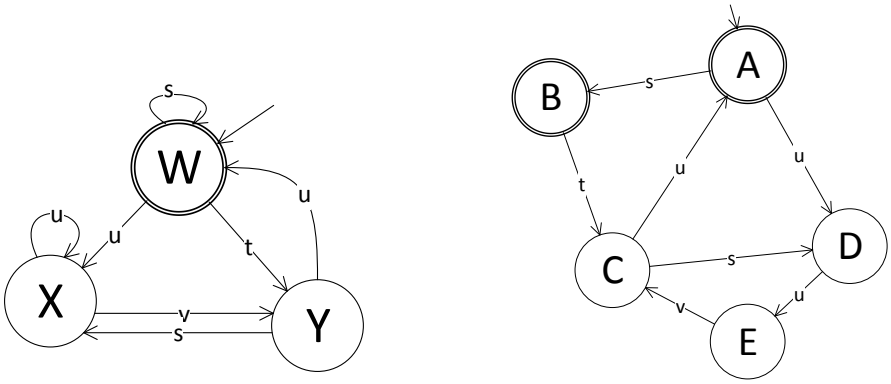
a *Draw transition system T as a labeled graph.*

b *Does transition system T strongly simulate transition system L? If so, give the relation, and show that it is a strong simulation. If not, explain why.*

c *Does transition system L strongly simulate transition system T? If so, give the relation, and show that it is a strong simulation. If not, explain why.*

d *Are transition system T and L strongly bisimilar? If so, give the relation, and show that it is a strong bisimulation. If not, explain why.*

Exercise 4.6. *In Excercise 1.17 of [1], you created two transition systems: one for the waiter, and one for the customer. Create their synchronous product that represents the system of these two objects.*



(a) Transition system L_1 (b) Transition system L_2

Figure 4.10: Two transition systems

Exercise 4.7. *Given are the two transition systems shown in Fig. 4.10:*

a Does transition system L_1 strongly simulate transition system L_2? If so, give the relation, and show that it is a strong simulation. If not, explain why.

b Does transition system L_2 strongly simulate transition system L_1? If so, give the relation, and show that it is a strong simulation. If not, explain why.

Exercise 4.8. *Explain the following statement: "two transition systems that are isomorphic, are strongly bisimilar as well".*

A

List of Symbols

A.1 Logic, Sets and Relations

Let A and B be logical statements, and let $P(\cdot)$ be a predicate. Let S and T be possibly infinite sets.

$A \wedge B$	A and B should both hold
$A \vee B$	A or B should hold, or both
$\neg A$	A does not hold
$A \implies B$	If A holds, then B holds as well
$\emptyset$	The empty set, containing no elements
$\{a, b, c\}$	A set containing the elements a, b and c
$a \in S$	A is an element in set A
$S \cup T$	The union of the sets A and B Elements in this set occur either in A or in B
$S \cap T$	The intersection of sets A and B Elements in this set occur in both A and B
$S \setminus T$	The difference of S with T All elements that are in S but not in T
$\forall a \in S : P(a)$	For all elements a in the set A predicate $P(a)$ should hold Remember, if $A = \emptyset$, then the statement is always true!
$\exists a \in S : P(a)$	There exists an element a in the set A such that predicate $P(a)$ holds
$S \times T$	Cartesian product. $S \times T = \{(a, b) \mid a \in S, b \in T\}$
$\pi_i((a, b))$	Projection to the i-th element of the pair (a, b)
$R \subseteq S \times T$	Relation R from S to T
$dom(R)$	Domain of R. If $R \subseteq S \times T$, $dom(R) \subseteq S$
$rng(R)$	Range of R. If $R \subseteq S \times T$, $rng(R) \subseteq T$
R^{-1}	Inverse of R. If $R \subseteq S \times T$, then $R^{-1} \subseteq T \times S$ such that $R^{-1} = \{(b, a) \mid (a, b) \in R\}$.

A.2 Bags

Let $A, B \in \mathbb{N}^S$ be two bags over some posssibly infinite set S.

$\emptyset$	The empty bag containing no elements
$\mathbb{N}^S$	The set of all possible bags over S A bag A is a function $A : S \to \mathbb{N}$, assigning a natural number to each element in S
$[a^2, b, c^3]$	The bag or multiset containing two times elements a one b, and three times element c
$A + B$	The sum of bags A and B $\forall s \in S : (A + B)(s) = A(s) + B(s)$
$A - B$	The difference of bags A and B. Exists only if $B \leq A$ $\forall s \in S : (A - B)(s) = A(s) - B(s)$
$A_{\|T}$	Projection of bag A on set T $\forall s \in S : (A_{\|T})(s) = \begin{cases} A(s) & \text{if } s \in T \\ 0 & \text{otherwise} \end{cases}$

A.3 Sequences

Let $\sigma, \tau \in S^*$ be two sequences over a possibly infinite set S.

S^*	The set of all possible finite sequences over S A sequence σ of length N assigns to each number an element, i.e., $\sigma : \{1, \ldots, N\} \to S$
ϵ	The empty sequence, containing no elements
$\langle a, b, c \rangle$	The sequence of a followed by b followed by c
$\|\sigma\|$	How many elements the sequence contains
$\sigma; \tau$	Concatenation of two sequences
$\sigma_{\|T}$	Projection of T on σ, defined inductively: $\epsilon_{\|T} = \epsilon$, $(\langle a \rangle; \sigma)_{\|T} = \langle a \rangle; \sigma_{\|T}$ if $a \in T$ and $(\langle a \rangle; \sigma)_{\|T} = \sigma_{\|T}$ if $\neg(a \in S)$
$\mathfrak{A}(\sigma)$	Alphabet of σ, i.e., the set of all elements that occur in σ
$\overrightarrow{\sigma}$	Parikh vector of σ, i.e., how often each element occurs in σ Defined inductively: $\overrightarrow{\epsilon} = \emptyset$, $\overrightarrow{(\langle a \rangle; \sigma)} = [a] + \overrightarrow{\sigma}$

A.4 Graphs

Let $G = (V, A)$ be a directed graph.

${}^{\bullet}_{G}s$	All nodes in the graph that point to node s ${}^{\bullet}_{G}s = \{t \mid (t, s) \in A\}$
$s^{\bullet}_{G}$	All nodes in the graph to which node s points $s^{\bullet}_{G} = \{t \mid (s, t) \in A\}$

A.5 Transition Systems

Let $L_1 = (S_1, A_1, \rightarrow_1, s_1, \Omega_1)$ and $L_2 = (S_2, A_2, \rightarrow_2, s_2, \Omega_2)$ be two transition systems. Let $a \in A_1$ and $\sigma \in A^*$. Let $Q \subseteq S_1 \times S_2$ be some relation.

τ	The silent step, $\tau \notin A_1$, $\tau \notin A_2$
$(L : s \xrightarrow{a} s')$	In L_1, action a leads from state s to state s'
$(L : s_0 \xrightarrow{\sigma} s_n)$	Firing sequence in L_1 from s_0 to s_n, defined inductively: $(L_1 : s \xrightarrow{\epsilon} s')$ if $s = s'$, and $(L_1 : s \xrightarrow{\langle a \rangle ; \sigma} s')$ if $\exists s'' \in S_1 : (L_1 : s \xrightarrow{a} s'')$ and $(L_1 : s'' \xrightarrow{\sigma} s')$
$(L_1 : s \Longrightarrow s')$	Silent step in L from s to s' $(L_1 : s \Longrightarrow s')$ iff $s = s' \vee \exists s'' \in S_1 : ((L : s \Longrightarrow s'') \wedge (L : s'' \rightarrow s'))$
$(L_1 : s \overset{a}{\Longrightarrow} s')$	Action a can occur in L_1 from s to s' with the help of τ-steps $(L_1 : s \overset{a}{\Longrightarrow} s')$ iff $\exists s'', s''' \in S_1 : (L_1 : s \Longrightarrow s'' \xrightarrow{a} s''' \Longrightarrow s')$
$\mathcal{L}(L_1)$	Language of L_1: all firing sequences from the initial state to some final state: $\mathcal{L}(L_1) = \{\sigma \mid \exists s_f \in S_1 : (L : s_1 \overset{\sigma}{\Longrightarrow} s_f)\}$
$L_1 \ {}_s\preceq_Q \ L_2$	L_1 is strongly simulated by L_2, or L_2 strongly simulates L_1. See Def. 4.8
$L_1 \ {}_s\simeq_Q \ L_2$	L_1 is strongly bisimilar to L_2 $L_1 \ {}_s\simeq_Q \ L_2$ iff $L_1 \ {}_s\preceq_Q \ L_2$ and $L_2 \ {}_s\preceq_{Q^{-1}} \ L_1$. See Def. 4.8
$L \ \preceq_Q \ L'$	L_1 is delay simulated by L_2, or L_2 delay simulates L_1. See Def. **??**
$L \ \simeq_Q \ L'$	L_1 is delay bisimilar to L_2 $L_1 \ \simeq_Q \ L_2$ iff $L_1 \ \preceq_Q \ L_2$ and $L_2 \ \preceq_{Q^{-1}} \ L_1$. See Def. **??**
$L_1 \times L_2$	The synchronous product of L_1 and L_2. See Def. 4.9

B

Solutions to Exercises

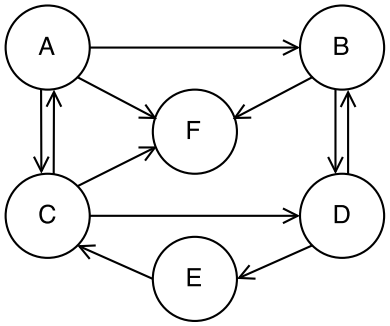
B.1 Solutions of Chapter 3

Exercise 3.1.

$$
\begin{aligned}
V &= \{ \quad A, B, C, D, E, F, G, H \ \} \\
A &= \{ \quad (A,B),(B,A),(A,D),(D,A),(A,G),(G,A),(B,E),(E,B), \\
&\qquad (G,E),(E,G),(D,G),(G,D),(B,F),(F,B),(D,C),(C,D), \\
&\qquad (F,C),(C,F),(C,H),(H,C) \ \}
\end{aligned}
$$

Tip: Count the number of elements in sets V and A. These should be equal to the number of vertices and arcs in the figure (i.e., $|V| = 8$, $|A| = 20$).

Exercise 3.2. Graphically representing the graph results in the following figure:



Tip: There are various notations for *bidirectional arcs* ($\leftrightarrow$, $\rightleftarrows$, $-$). If you have directed arcs in the graph, we suggest using an arrow notation ($\leftrightarrow$, $\rightleftarrows$) for bidirectional arcs to avoid ambiguity.

Exercise 3.3.

$$\begin{aligned} V &= \{ \quad A, B, C, D, E, F, G, H, I \ \} \\ A &= \{ \quad (B, D), (B, C), (D, C), (D, H), (C, A), (H, A), (H, G), (A, E), \\ &\qquad (E, F), (F, C), (G, I), (I, E), (I, F) \ \} \end{aligned}$$

Observe that the order of the elements in sets V and A does not matter.

Exercise 3.4. This graph H is isomorphic to the graph of Exercise 3.2. As a first indicator, we observe that $|V_G| = |V_H| = 6$ and $|A_G| = |A_H| = 11$. In other words, the number of nodes and arcs in both graphs are equal. If this was not the case, it would be impossible for the graphs to be isomorphic.

We map the vertices in G to the vertices in H in a function $f : V_G \to V_H$ such that

$$f = \{(A, V), (B, W), (C, Y), (D, Z), (E, X), (F, U)\}$$

Subsequently, we can map all arcs of G to H using f:

$$\begin{aligned} \{ \quad & (f(A), f(B)), (f(A), f(F)), (f(B), f(F)), (f(A), f(C)), (f(B), f(D)), \\ & (f(C), f(F)), (f(E), f(C)), (f(C), f(A)), (f(D), f(B)), (f(C), f(D)), \\ & (f(D), f(E)) \ \} = \\ \{ \quad & (V, W), (V, U), (W, U), (V, Y), (W, Z), (Y, U), (X, Y), (Y, V), (Z, W), \\ & (Y, Z), (Z, X) \ \} \end{aligned}$$

Conversely, we could map H to G in a function $f' : V_H \to V_G$.

We observe that graphs G and H are isomorphic. All vertices in G can be mapped to a vertex in H, and applying this function to all arcs in G (A_G) results in a formalization of A_H.

Exercise 3.5. Method 1: Alphabetic

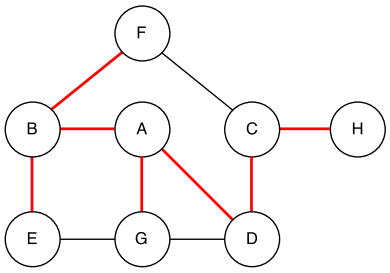
Step	Node	Queue
0	A	ϵ
1		$\langle A \rangle$
2	A	$\langle B, D, G \rangle$
3	B	$\langle D, G, E, F \rangle$
4	D	$\langle G, E, F, C \rangle$
5	G	$\langle E, F, C \rangle$
6	E	$\langle F, C \rangle$
7	F	$\langle C \rangle$
8	C	$\langle H \rangle$
9	H	ϵ

Method 2: Visual left-to-right top-to-bottom $\langle F, B, A, C, H, E, G, D \rangle$

Step	Node	Queue
0	A	ϵ
1		$\langle A \rangle$
2	A	$\langle B, G, D \rangle$
3	B	$\langle G, D, F, E \rangle$
4	G	$\langle D, F, E \rangle$
5	D	$\langle F, E, C \rangle$
6	F	$\langle E, C \rangle$
7	E	$\langle C \rangle$
8	C	$\langle H \rangle$
9	H	ϵ

Other methods: If you are certain you found an alternative correct solution, you can check your solution with your teaching assistant.

Both methods result in the same spanning tree

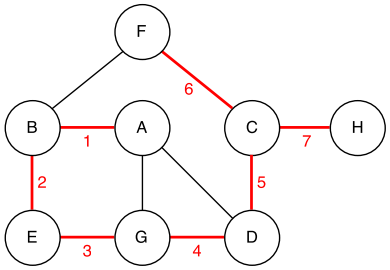


Exercise 3.6. Method: Alphabetical

Step	**Node**	**Stack**
0		$\langle A \rangle$
1	A	$\langle A, B \rangle$
2	B	$\langle A, B, E \rangle$
3	E	$\langle A, B, E, G \rangle$
4	G	$\langle A, B, E, G, D \rangle$
5	D	$\langle A, B, E, G, D, C \rangle$
6	C	$\langle A, B, E, G, D, C, F \rangle$
7	F	$\langle A, B, E, G, D, C, F \rangle$
8	F	$\langle A, B, E, G, D, C \rangle$
9	C	$\langle A, B, E, G, D, C, H \rangle$
10	H	$\langle A, B, E, G, D, C, H \rangle$
11	H	$\langle A, B, E, G, D, C \rangle$
12	C	$\langle A, B, E, G, D, C \rangle$
13	C	$\langle A, B, E, G, D \rangle$
14	D	$\langle A, B, E, G, D \rangle$
15	D	$\langle A, B, E, G \rangle$
16	G	$\langle A, B, E, G \rangle$
17	G	$\langle A, B, E \rangle$
18	E	$\langle A, B, E \rangle$
19	E	$\langle A, B \rangle$
20	B	$\langle A, B \rangle$
21	B	$\langle A \rangle$
22	A	$\langle A \rangle$
23	A	ϵ

For other methods, the same applies as in Exercise 3.5.

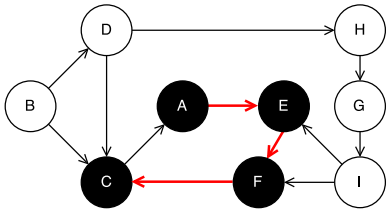
Spanning tree corresponding to Exercise 3.6 (the red numbers indicate the order in which the arcs were traversed):



Exercise 3.7. The BFS for Exercise 3.3:

Step	Node	Queue
0	A	ϵ
1		$\langle A \rangle$
2	A	$\langle E \rangle$
3	E	$\langle F \rangle$
4	F	$\langle C \rangle$
5	F	ϵ

Tree from Node A



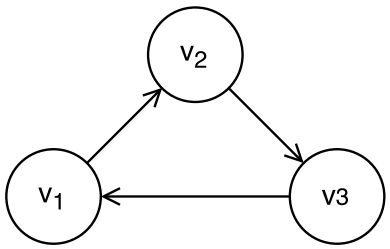
Observe that we do not denote this tree as a *spanning tree*, because it does not contain all vertices.

Exercise 3.8. The DFS for Exercise 3.3:

Step	Node	Stack
0		$\langle A \rangle$
1	A	$\langle A, E \rangle$
2	E	$\langle A, E, F \rangle$
3	F	$\langle A, E, F, C \rangle$
4	C	$\langle A, E, F, C \rangle$
5	C	$\langle A, E, F \rangle$
6	F	$\langle A, E, F \rangle$
7	F	$\langle A, E \rangle$
8	E	$\langle A, E \rangle$
9	E	$\langle A \rangle$
10	A	$\langle A \rangle$
11	A	ϵ

The tree from A is equal to the one presented in Ex 3.7.

Exercise 3.9. A topological sort on a graph with $V = \{v_1, v_2, \ldots, v_n\}$ requires that for every arc (v_i, v_j) v_i must precede v_j $(i ¡ j)$. Consider a cyclic graph $G' = \{V', A'\}$ where $V' = v_1, v_2, v_3$ and $A' = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$ as depicted in the Figure below.



Here, a topological sort would result in an impossible scenario. For vertex v_1 you need v_3 and v_2 as a prerequisite, for v_3 the prerequisites are v_2 and v_1. This implies that v_1 precedes v_3, which in turn precedes v_2 and v_1. Using a transitive relation we see that v_1 precedes itself, i.e. $1 < 1$. This scenario is impossible, as a number cannot be smaller than itself!

Exercise 3.10. The shortest path resulting from Dijkstra's is $\langle A, B, F, D, G\rangle$ with weight 80:

Step	**Node**	**Parent**	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
0		$\perp$	**0**	∞	∞	∞	∞	∞	∞	∞
1	A	$\perp$		$\mathbf{20_A}$	∞	80_A	∞	∞	∞	∞
2	B	A			∞	80_A	∞	$\mathbf{30_B}$	∞	∞
3	F	B			80_F	$\mathbf{70_F}$	∞		∞	∞
4	D	F			$\mathbf{80_F}$		∞		80_D	∞
5	C	F					∞		$\mathbf{80_D}$	100_C
6	C	F					∞			100_C

Note that we choose C over G in step 4. There are only three arcs on the shortest path from A to C $\langle A, B, F, C\rangle$, but there are four on the shortest path from A to G $\langle A, B, F, D, G\rangle$. Moreover, we do not replace 80_F with 80_D for C in step 4 as these two are equal (Dijkstra's looks at new value smaller than the old value and therefore disregards equal values).

Exercise 3.11. The shortest path according to Dijkstra's is $\langle A, B, G, J\rangle$ with a total weight of 20.

Step	**Node**	**Parent**	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
0		$\perp$	**0**	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	A	$\perp$		$\mathbf{1_A}$	2_A	3_A	∞	∞	∞	∞	∞	∞
2	B	A			$\mathbf{2_A}$	3_A	6_B	∞	5_B	∞	∞	∞
3	C	A				$\mathbf{3_A}$	6_B	8_C	5_B	∞	∞	∞
4	D	A					6_B	8_C	$\mathbf{5_B}$	∞	∞	∞
5	G	B					$\mathbf{6_B}$	8_C		21_G	∞	20_G
6	E	B						$\mathbf{8_C}$		21_G	∞	20_G
7	F	C								21_G	22_F	$\mathbf{20_G}$
8	J	G								21_G	22_F	

Exercise 3.12. The shortest path according to Dijkstra's is $\langle A, B, C, D, G, M, N\rangle$, with a total weight of 14.

Step	Node	Parent	A	B	C	D	E	F	G	H	I	J	K	L	M	N
0		$\perp$	**0**	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	A	$\perp$		$\mathbf{2_A}$	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	B	A			5_B	∞	$\mathbf{4_B}$	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	E	B			$\mathbf{5_B}$	∞		∞	∞	∞	8_E	∞	∞	∞	∞	∞
4	C	B				$\mathbf{6_C}$		7_C	∞	10_C	8_E	∞	∞	∞	∞	∞
5	D	C						$\mathbf{7_C}$	7_D	8_D	8_E	∞	∞	∞	∞	∞
6	F	C							$\mathbf{7_D}$	8_D	8_E	∞	∞	∞	∞	∞
7	G	D								$\mathbf{8_D}$	8_E	∞	∞	∞	9_G	∞
8	H	D									$\mathbf{8_E}$	10_H	13_H	∞	9_G	∞
9	I	E										10_H	13_H	∞	$\mathbf{9_G}$	∞
10	M	G										$\mathbf{10_H}$	13_H	∞		14_M
11	J	H											13_H	$\mathbf{11_J}$		14_M
12	L	J											$\mathbf{13_H}$			14_M
13	K	H														$\mathbf{14_M}$

Note that there is another shortest path of length 14: $\langle A, B, C, D, H, J, L, K, N\rangle$. However, as the first time the algorithm hits node N, the parent M is stored. As the value never gets smaller, as required by Dijkstra's algorithm (see line 10), the parent is never updated!

B.2 Solutions of Chapter 4

Exercise 4.1. Let $L = (S, A, \rightarrow, s_0, \Omega)$ be an LTS. The definition of the silent step is $(L : s \Longrightarrow s')$ iff $s = s'$ or $\exists s'' \in S : (L : s \Longrightarrow s'') \wedge (L : s'' \rightarrow s')$. Thus, for any state s, we have $(L : s \Longrightarrow s)$, since $s = s$. Hence $\Longrightarrow$ is reflexive.

Exercise 4.2. Let $L = (S, A, \rightarrow, s_0, \Omega)$ be an LTS. The definition of firing $a \in \mathcal{A} \cup \{\tau\}$ is $(L : s \stackrel{a}{\Longrightarrow} s')$ iff $\exists s'' \in S : (L : s \Longrightarrow s'') \wedge (L : s'' \stackrel{a}{\rightarrow} s')$. Suppose LTS L can fire a possibly silent action $a \in A \cup \{\tau\}$ from s to s', i.e., $(L : s \stackrel{a}{\rightarrow} s'')$. Since $\Longrightarrow$ is reflexive, $(L : s \Longrightarrow s)$. Hence, choose $s'' = s$. Then $(L : s \rightarrow s'')$ and $(L : s'' \stackrel{s}{\rightarrow} a)$, which exactly matches the definition of firing. Hence $smallarrow \subseteq \Longrightarrow$.

Exercise 4.3.

a. The LTS $L = (S, A, \rightarrow, s_0, \Omega)$ is defined by:

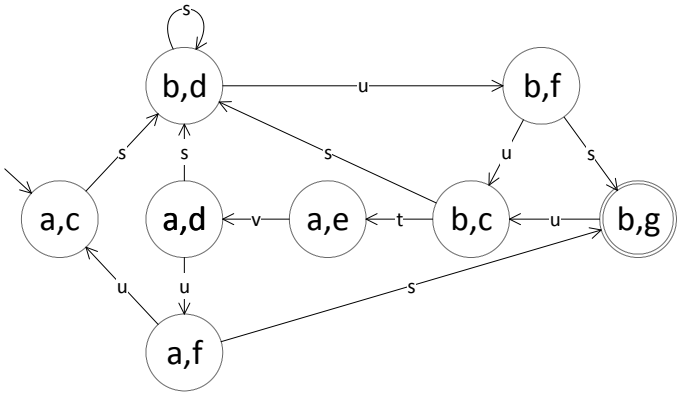
$$\begin{aligned} S &= \{ \quad a, b, c, d, e, f, g \quad \} \\ A &= \{ \quad t, u, v, w, x, y, z \quad \} \\ \rightarrow &= \{ \quad (b, t, a), (b, \tau, c), (b, u, e), (a, \tau, d), (c, \tau, f), (c, y, d), \\ &\qquad (d, x, e), (f, v, e), (e, \tau, g), (e, z, c)(g, w, f) \quad \} \\ s_0 &= \quad b \\ \Omega &= \{ \quad a \quad \} \end{aligned}$$

b. The language of this transition system $\mathcal{L}(L) = \{\langle t \rangle\}$, as this is the only sequence from the initial state to a final state.

c. As all states have at least one outgoing arc, there is no deadlock in L.

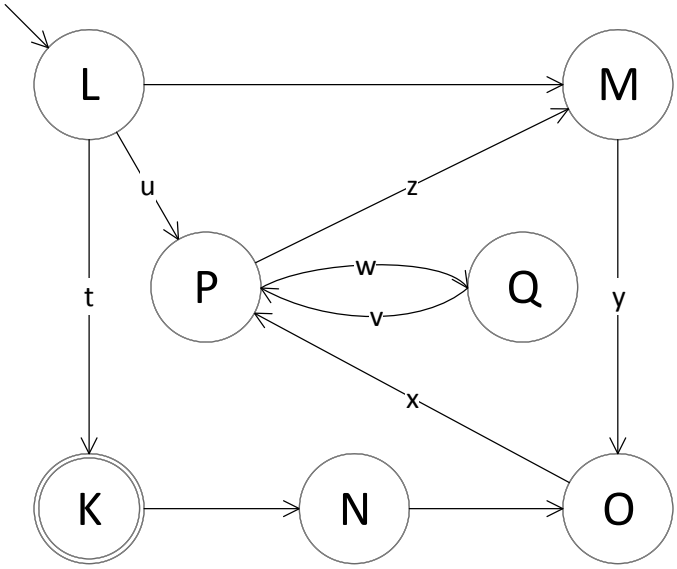
d. The system has one minimal livelock: $\{e, c, d, f, g\}$. Any arc leaving from these states returns in this set. Hence, it is a livelock. Notice that the set of all nodes is a livelock as well, but this one is trivial.

Exercise 4.4.



Exercise 4.5.

a. Transition system T is graphically depicted as:



b. Does T strongly simulate L? Thus, we need to show that $L \ {}_s\preceq_Q T$. Such a relation Q does not exist, since relating state d in L is not possible: if we relate it to N in T, then T cannot simulate action x in L. Additionally, because of the τ-steps, we cannot relate state a of L to state O in T. Hence, there is no strong simulation.

c. To answer whether L strongly simulate T, we need to show that $T \ {}_s\preceq_R T$. Such a relation R does not exist: the only candidates to relate state N

to are states a or d. It cannot be related to a. Because a and K are the only final states, we have $(K, a) \in R$. If $(N, a) \in R$, then $(T : K \to N)$ cannot be simulated. Similarly, if $(N, d) \in R$, then $(N : N \to O)$ cannot be simulated, as the τ-step should be explicit in a strong simulation.

d. Transition systems T and L are not strongly bisimilar, since for both direction no strong simulation relation can be found, as shown in (b) and (c).

Exercise 4.7.

a. To verify whether L_1 simulates L_2, we need to find a relation Q such that $L_2 \ {}_s\preceq_Q \ L_1$. Let $Q = (A, W), (B, W), (C, Y), (D, X), (E, X)$. Then, (1) the inital states are related, and (2) each final state of L_2 is related to a final state of L_1. Next, we need to validate all steps:

- $(L_2 : A \xrightarrow{s} B)$, and $(A, W) \in Q$. Then $(L_1 : W \xrightarrow{s} W)$, and $(B, W) \in Q$.
- $(L_2 : A \xrightarrow{u} D)$, and $(A, W) \in Q$. Then $(L_1 : A \xrightarrow{u} X)$, and $(D, X) \in Q$.
- $(L_2 : B \xrightarrow{t} C)$, and $(B, W) \in Q$. Then $(L_1 : W \xrightarrow{t} Y)$, and $(C, Y) \in Q$.
- $(L_2 : C \xrightarrow{u} A)$, and $(C, Y) \in Q$. Then $(L_1 : Y \xrightarrow{u} W)$, and $(A, W) \in Q$.
- $(L_2 : C \xrightarrow{s} D)$, and $(C, Y) \in Q$. Then $(L_1 : Y \xrightarrow{s} X)$, and $(D, X) \in Q$.
- $(L_2 : D \xrightarrow{u} E)$, and $(D, X) \in Q$. Then $(L_1 : X \xrightarrow{u} X)$, and $(E, X) \in Q$.
- $(L_2 : E \xrightarrow{v} C)$, and $(E, X) \in Q$. Then $(L_1 : X \xrightarrow{v} Y)$, and $(C, Y) \in Q$.

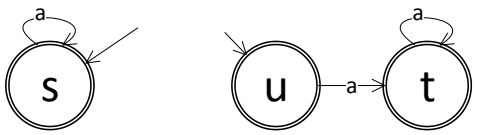
Hence, all steps fulfill the step-criterion of simulation. Hence $L_2 \ {}_s\preceq_Q \ L_1$.

b. To verify whether L_2 simulates L_1, we need to find a relation R such that $L_1 \ {}_s\preceq_Q \ L_2$. Suppose $(W, A) \in R$. Since, $(L_1 : W \xrightarrow{s} W)$ and $(L_2 : A \xrightarrow{s} B)$, also $(W, B) \in R$, since otherwise R would not be a simulation relation. But then, $(L_1 : W \xrightarrow{u} X)$, but there is no state $Z \in S_2$ such that $(L_2 : B \xrightarrow{u} Z)$. Hence, such relation R with $L_1 \ {}_s\preceq_Q \ L_2$ does not exist.

Exercise 4.8. Under the assumption that the isomorphism relation relates the initial state and final states, we only need to consider the third option. Let L_1 and L_2 be two transition systems that are isomorphic with relation f (i.e., $f : S_1 \to S_2$). By definition of isomorphism, we have $(A, t, B) \in \to_1$

if and only if $(f(A), t, f(B)) \in \rightarrow_2$. Hence, if $(L_1 : A \xrightarrow{t} B)$, then also $(L_2 : f(A) \xrightarrow{t} f(B))$, which directly satisfies the third criterion of strong simulation.

Notice that the reverse is not true: if two transition systems are bisimilar, then they need not be isomorphic. As an example, consider the transition systems $L_1 = (s, a, (s, a, s), a, a)$ and $L_2 = (t, u, a, (u, a, t), (t, a, t), u, t, u)$, as depicted below. Let $Q = (s, t), (s, u)$. Then it is easy to see that $L_1 \ {}_s\preceq_Q \ L_2$ and $L_2 \ {}_s\preceq_{Q^{-1}} \ L_1$, i.e., L_1 and L_2 are strongly bisimilar. However, their graphs are not isomorphic.



(a) L_1 (b) L_2

Bibliography

[1] W.M.P. van der Aalst and C. Stahl. *Modeling Business Processes: a Petri Net-Oriented Approach*. MIT Press, 2011.

[2] K.R. Blackman. IMS celebrates thirty years as an IBM meeting. *IBM Systems Journal*, 37(4), 1998.

[3] P.P. Chen. The Entity-Relationship Model: Towards a unified view of Data. *ACM Transactions on Database Systems*, 1:9–36, January 1976.

[4] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[5] R.J. van Glabbeek. The Linear Time - Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves. In *Proceedings of CONCUR 1993*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, Berlin, 1993.

[6] H. Hollerith. An Electric Tabulating System. *The Quarterly*, X(16):238–255, April 1889.

[7] H. Hollerith. *In connection with the electric tabulation system which has been adopted by U.S. government for the work of the census bureau.* PhD thesis, Columbia University School of Mines, 1890.

[8] H. Hollerith. The Electrical Tabulating Machine. *Journal of the Royal Statistical Society*, 57(4):678–689, December 1894.

[9] Object Management Group. Business Process Modeling Notation, V1.1. http://www.omg.org/spec/BPMN/1.1/PDF/, 2008.

[10] R.J. Parikh. On Context-Free Languages. *Journal of the ACM*, 13(4):570 – 581, October 1966.

[11] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer-Verlag, Berlin, 1985.