Information Containers

Data structures and graphs

LEO LIBERTI

April 24th, 2011

Contents

Contents iii

1 Introduction **3**

1.1 A motivation for data structures . . . 3
1.2 Motivations for graphs . . . 7
1.3 Exercises . . . 14

2 Mathematical structures **19**

2.1 The formal language . . . 19
2.2 Sets . . . 20
2.3 Functions . . . 21
2.4 Sequences . . . 22
2.5 Relations . . . 23
2.6 Groups . . . 25
2.7 Fields . . . 28
2.8 Vector spaces . . . 28
2.9 Exercises . . . 29

3 Graphs **31**

3.1 Graphs and digraphs . . . 32
3.2 Subgraphs . . . 34
3.3 Walks, paths and cycles . . . 34
3.4 Trees . . . 35
3.5 Stables and cliques . . . 35
3.6 Operations on graphs . . . 35
3.7 Exercises . . . 35

4 Data structures **37**

4.1 Types . . . 37
4.2 The main definition . . . 38
4.3 Arrays . . . 39
4.4 Lists . . . 39
4.5 Queues . . . 39

4.6 Hash maps 39
4.7 Trees 39

Bibliography **41**

Index **43**

Preface

Although this looks like a book, it is not a book. Perhaps one day it will become a book, but for the moment it is just a set of notes designed to help me think about how to teach a fundamental computer science course for students at Ecole Polytechnique. It may serve as a reference, and hopefully it will even clarify things. But students using these notes should also rely on other books. I would advise the "polycopié" for INF421 written by Philippe Baptiste and Luc Maranget, as well as the recent book by Kurt Mehlhorn and Peter Sanders [**?**].

This material was written with teaching in mind, by someone who studied mathematics (rather than computer science) in college. For a mathematician, teaching computer science is devious. For purposes of clarity, mathematicians never hesitate in giving different technical views of the same fundamental concept. But "the computer" is actually a real object with a set of corresponding physical properties. Bending facts — whilst keeping the functional properties valid — for didactical purposes amounts to lie so that the readers can better understand a concept. In this material I only refer to *conceptual models* of a computer, not to the actual physical object. Thus, if I believe I can be clearer, I will not refrain from distorting some physical fact whilst keeping the functional description valid.

Let me dispel a myth about learning computer science. Students often believe that a computer science course will teach them how to use and program computers. This is less than half true. By analogy, would you consider yourself a pianist after attending a musical theory course? Of course not: you have to

actually put your hands on the keyboard and and practice for ten years or so; naturally, a good supporting musical theory course can speed things up whilst you teach your brain and hands to adapt to the new expressive medium. Programming computers is as much a practice as it is a science. A computer science course can help you steer towards a good direction, but it is no substitute for practice. Quite the reverse is true, in fact: there are some brilliant coders which learned the trade all on their own, without ever following a course. Although they are now becoming a minority, learning to program computers has always been an affair between the coder and the machine (no teacher involved) until relatively recently, when universities started opening computer science departments. Compare with mathematics: budding mathematicians have followed mathematics courses ever since mathematics existed, and “learning mathematics”, “teaching mathematics” and “creating mathematics” were always considered to be necessary activities for any mathematician. Computer science is different, and requires a lot of solitary work between coder and machine. So you should not expect to succeed in this course without the proverbial blood, sweat and tears. Get programming.

CHAPTER

1 Introduction

This introductory chapter is a collection of motivating examples treated informally. No formal definitions will be introduced here. The primary purpose of the chapter is to invite the reader to further the study of data structures and graphs. Another important purpose is to establish certain key ideas which will be discussed in depth later on.

1.1 A motivation for data structures

A data structure is an organized arrangement of information in the computer memory. The main message in this section is:

The way information is arranged in a computer memory may impact algorithmic efficiency — it is therefore important to employ the best structure.

A scientist gathers data samples $a = (a_1, \dots, a_n) \in \mathbb{R}^n$. The experimental protocol requires the application of the function $f : \mathbb{R}^n \to \mathbb{R}$, given by:

CPU time is measured in terms of number of elementary operations (taking a negligible time) performed by a program.

$$f(x) = \sum_{i=1}^{n} i x_i$$

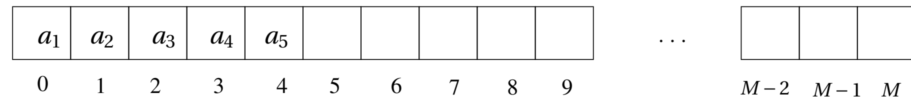
to the samples. The scientist writes the computer program given in Alg. 1, and

Algorithm 1 weightedSum

Input: an integer n, an array of floating point numbers $a \in \mathbb{R}^n$
Output: a floating point number s containing the result
1: $s \leftarrow 0$
2: **for** $i \in \{1, \ldots, n\}$ **do**
3: $\quad s \leftarrow s + i\,a_i$
4: **end for**

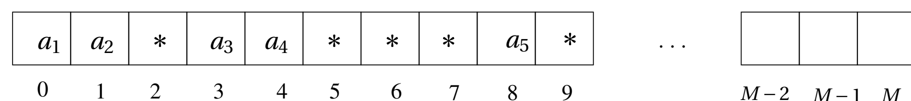
then runs the program on a collection of 1000 samples of size $n = 100$. How long will the program take to complete?

The answer to the above question mainly depends on how we store and manipulate information within the computer memory. We can safely assume that our model for the computer memory is a finite, linearly arranged array of "boxes", indexed from 0 to M, each of which can contain a piece of data. We might then imagine that a sequence of 5 floating point numbers $a_1, \ldots, a_5$ is stored in memory as follows.



With this memory model, reading the value a_i at the i-th iteration of the loop at Line 2 would require a constant CPU time (say, for simplicity, one unit of CPU time), as the index of the box containing a_i is simply $(i-1)$. Since there are n iterations in the loop, Alg. 1 would take n CPU time units to complete.

This, however, is a very coarse model of what *really* happens. A more convincing model would take into account the fact that most operating systems nowadays are time-sharing, i.e. they share the CPU time among an unspecified number of applications. This gives the user the appearance that each application is run by a dedicated CPU. Specifically, we are going to pretend that Alg. 1 program receives just enough CPU time to write at most two floating point numbers in memory during its allocated slot. A more accurate memory representation would then be:



where $*$ denotes characters written to memory by other programs in execution. The effect on the memory is that pieces of data which should logically be arranged in contiguous boxes end up being fragmented. As Alg. 1 has no way of knowing the index of the box holding each piece of data, this situation is troublesome.

Data are *fragmented* whenever logically related data elements are non-contiguous in memory.

A more suitable memory structure for holding the data would employ two boxes for each piece of data: how for holding the actual data, and another to store the index of the next relevant box pair.

a_1	0x2	a_2	0x5	$*$	a_3	0x7	a_4	0xC	$*$	$*$	$*$	a_5	-0x1
0	1	2	3	4	5	6	7	8	9	A	B	C	D

In the above representation, we used hexadecimal notation for denoting box indices, and we used the negative index $-0x1$, which would never be used to index an actual box, to mark the end of the array. A clearer graphical representation of the list just discussed is shown in Fig. 1.1 In this linked list, each

Hexadecimal notation: integers are expressed in base 16. Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. We prefix hex numbers with 0x. E.g. 0x1C is 28 in base 10.

a_1 → a_2 → a_3 → a_4 → a_5

Figure 1.1: A graphical representation of a linked list.

piece of data takes two adjacent boxes: the left box stores the actual data, and the right one stores the index of the following box. This wastes data but allows a more flexible use of memory (there are other advantages associated to this representation, which will be discussed later on).

Linked lists are amongst the most fundamental data structures. They can also be seen as a special type of graph, namely paths (linked sequences of elements).

A → B

A=*parent/head*
B=*child/subnode/tail*

A B

A=*root* B=*leaf*

If we look at Alg. 1 in more detail, however, we also discover a disadvantage: Line 3 requires reading from memory the value of a_i given the value for i. In the very simplest analysis, accessing the i-th element of data a_i requires starting from the beginning and following the links in the list i times, thereby yielding a CPU time proportional to i. Line 3 makes it clear that Alg. 1 takes a CPU time proportional to i to update s at the i-th iteration, since a_i needs to be read. Summing up, the CPU time required by Alg. 1 is proportional to

$$1+2+\ldots+n=\frac{n(n+1)}{2},$$

which is one order of magnitude larger than we would have obtained by using a linear array whose boxes contain data elements with contiguous indices.

A *tree* is also a special type of graph, notably a connected graph without cycles (closed paths).

We now propose a third memory structure which improves on this situation, whilst still allowing for fragmented data storage: the binary tree shown in Fig. 1.2. Each tree element v is called a *node*. In the present case, each

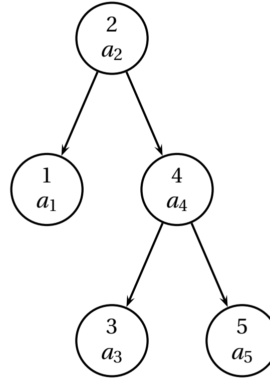


Figure 1.2: A tree structure.

node consists of three contiguous memory boxes: the middle box stores a data element, the left box stores the index of the middle box of the left subnode v^- and the right box stores the index of the middle box of the right subnode v^+. A procedure for finding the element a_i in the tree is given in Alg. 2. This procedure is recursive (this feature will be discussed in much more detail later). If we denote the root node of the tree by r, then $\text{treeFind}(r,i)$ will correctly return a_i. For example, if $r = 2$ and we call $\text{treeFind}(2,3)$, Alg. 2 establishes that $v < i$

A procedure is *recursive* when one of the steps is a call to the procedure itself, with different arguments.

Algorithm 2 $\text{treeFind}(v,i)$

Input: a box index v, an integer i with $i \leq n$
Output: the data element a_i
1: **if** $v = i$ **then**
2: **return** a_v
3: **else if** $v > i$ **then**
4: **return** $\text{treeFind}(v^-, i)$
5: **else if** $v < i$ **then**
6: **return** $\text{treeFind}(v^+, i)$
7: **end if**

and hence calls itself at Line 6 as $\text{treeFind}(4,3)$, then it establishes that $v > i$ and hence calls itself at Line 4 as $\text{treeFind}(3,3)$ and finally verifies that $v = i$ and returns a_3. This all works because $i \in \{1,\ldots,n\}$ and because the tree contains all n values of the sequence a arranged in a special way. Namely, for each node v the left subtree contains data values a_i with $v > i$ and the right subtree contains data values a_i with $v < i$, with node v containing the data value a_v.

A *subtree* is a tree which is also part of another tree.

The crucial observation for this memory structure is that in order to retrieve a_i, for any given $i \in \{1,\dots,n\}$, we always start from the root node and, at worst, we only need to access as many nodes as the path from the root to the node containing a_i: in the worst case, this may be a leaf node. The length of the path from the root to the deepest leaf node in a tree is known as the *height* of the tree. If the binary tree is balanced then the height of the tree is approximately $\log_2 n$. Thus, the CPU time taken by Alg. 1 is proportional to $n \log_2 n$, which is less than $n(n+1)/2$, as was the case for the linked list structure. This shows that a balanced binary tree is a good compromise between fragmentation and efficiency.

In a *balanced tree*, for each node v of the tree the subtree rooted on v^- contains approximately as many nodes as the subtree rooted on v^+.

In the following, we shall refer to the model consisting of a finite linearly arranged array of boxes as *memory*, and to box indices as *memory addresses*.

1.2 Motivations for graphs

Graphs are used in mathematics, science and engineering to represent relations on elements of a set. The main message in this section is:

Data elements are not the only essential piece of information in data; the relations between the elements are also vitally important.

1.2.1 Data and graphs

Different pieces of information relating to similar occurrences are often structured. Think of a spreadsheet: different rows refer to different items with a common set of attributes, organized by columns. The same holds in most databases, where each table (equivalent to a sheet of a spreadsheet) holds a set of records (equivalent to rows) with a common set of properties (equivalent to columns). Searching, sorting and querying data organized this way yields a relation on the data. For example, a sorting operation on the sequence $(a_4, a_3, a_5, a_1, a_2)$ according to the indices results in the ranking $(a_1,\dots,a_5)$. This can be modelled by the relation consisting of the following set of ordered pairs: $(a_1,a_2),(a_2,a_3),(a_3,a_4),(a_4,a_5)$. We can represent this as the graph shown in Fig. 1.3. The similarity with the linked list representation of Fig. 1.1 is striking.

A *relation* on a set is a set of pairs of elements of the set.

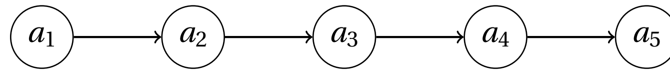


Figure 1.3: Graph of the order relation $(a_1, \ldots, a_5)$.

Different relations on the same records yield different graphs: for example, $(a_2, a_1), (a_2, a_4), (a_4, a_3), (a_4, a_5)$ corresponds to the tree of Fig. 1.2. Thus, data structures can be modelled by graphs.

The usefulness of representing data structures by means of the "graph" abstraction is that the whole body of theoretical and algorithmic results on graphs can be applied to the data structure in question.

1.2.2 The web graph

Information need not be as structured as spreadsheets or databases. Each web page, for example, corresponds to a file, usually written in Hyper-Text Markup Language (HTML), which is a sequence of words of a formal language (the HTML tags) interspersed with words of a natural language (English, French and so on). A specific HTML tag, *name*, permits the creation of the hyperlink *name* pointing to the information stored in the Uniform Resource Locator (URL) *url*. This yields a relation consisting of ordered pairs of web pages whenever the first contains a hyperlink to the second. The graph corresponding to this relation is huge and constantly evolving. A 2009 version of the web graph[1] counts 4,780,950,903 URLs and 7,944,351,835 hyperlinks.

A *language* over the alphabet A is a subset of A^*, the set of all finite sequences of characters in A.

In a formal language each sentence has a precisely defined meaning. This is not the case for natural languages.

Graphs that change over time are called *dynamic*.

Fig. 1.4 and 1.5 show small web subgraphs around two organizational websites: the Institute for Electrical and Electronics Engineers (IEEE) and the French car manufacturer PSA corporate website. As the web graph is a dynamic, the two subgraphs in the figures correspond to snapshots taken in 2008. The PSA web subgraph shows a more tree-like structure than the IEEE subgraph. This is likely to be an effect of the PSA corporate structure, organized more hierarchically than an academic society.

A subgraph is a graph which is also part of another graph.

[1] http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Web+Graph

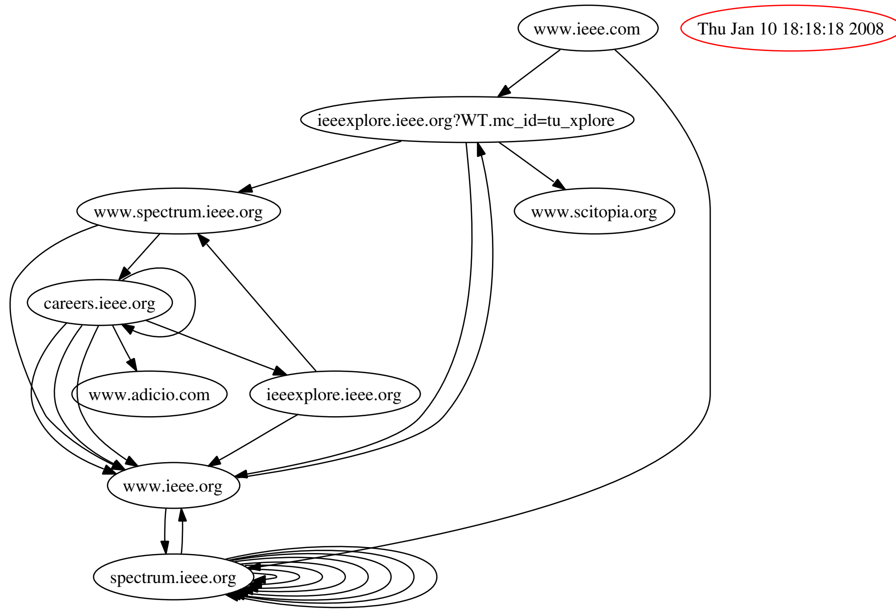


Figure 1.4: A web subgraph around www.ieee.org.

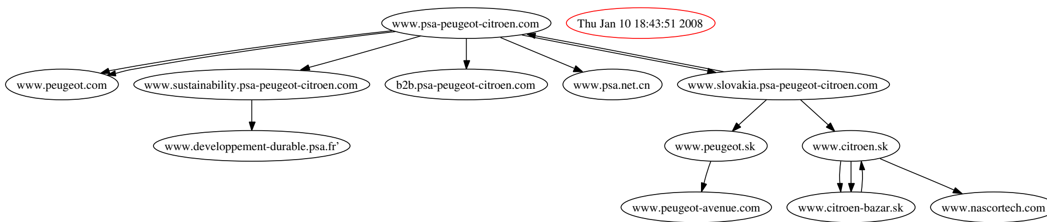


Figure 1.5: A web subgraph around www.psa-peugeot-citroen.com.

1.2.3 The internet graph

The set of all internet routers also yields a graph whose relations consists of pairs of connected routers. Unlike previous examples, this relation is symmetric: if router A is connected to router B, then router B is connected to router A. Fig. 1.6 shows[2] a picture of autonomous systems of Internet Protocol (IPv4) numbers dated 2005 — each autonomous system corresponds more or less to an Internet Service Provider (ISP).

In a symmetric relation $\sim$, if $a \sim b$ then $b \sim a$.

[2] http://www.caida.org/research/topology/as_core_network/2005/

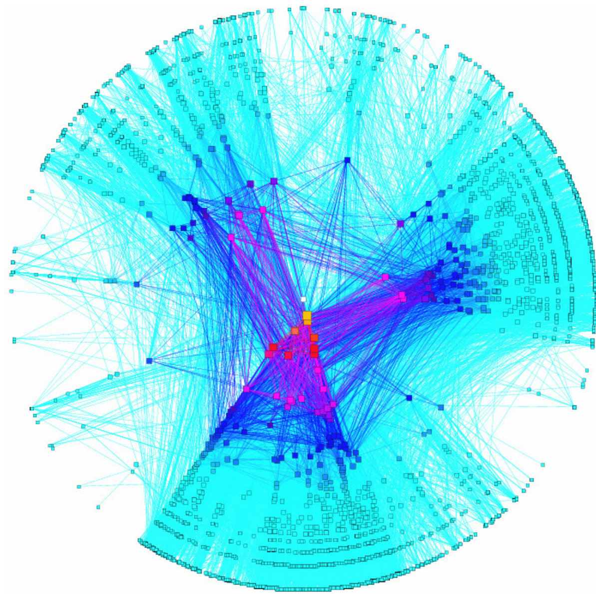
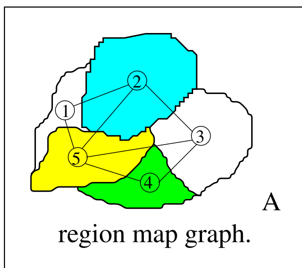


Figure 1.6: The ISP graph in 2005.

1.2.4 Maps and graphs



Geographical maps are usually modelled as graphs exploiting two separate features: borders between regions and roads between places. In the first instance, the map is seen as a set of disjoint regions delimited by borders. The relation between regions is given by adjacency: two regions are adjacent if they share part of their borders. In the second instance, the map is seen as a set of different places, which are pairwise related if there is a road connecting them. In the first case the relation is symmetric, whereas in the second place it may not be so (think of one-way roads).

1.2.4.1 Region maps

Graphs associated to region maps are famous in mathematics mostly because of the the four-colour theorem, which states that for any such graph, four colours are sufficient to colour the regions in such a way that no two adjacent regions are coloured the same way. The four-colour theorem was first stated by Francis Guthrie, the brother of Frederick, a student of Augustus De Morgan, professor of mathematics at University College, Dublin. Professor De Morgan could not find a proof of this seemingly simple statement, and wrote to Sir William Rowan

Hamilton on 23rd Oct. 1852:

> *A student of mine asked me today to give him a reason for a fact which I did not know was a fact — and do not yet. He says that if a figure be anyhow divided and the compartments differently coloured so that figures with any portion of common boundary line are differently coloured — four colours may be wanted, but not more [. . .] The more I think of it, the more evident it seems. If you retort with some very simple case which makes me out a stupid animal, I think I must do as the Sphynx did [. . .].*

Sir Hamilton answered on 26th Oct. that he was not likely to attempt to solve the problem soon. Several mathematicians got interested in this problem, until a solution involving the use of computers was announced in 1976 [1]. Because large parts of this proof involved computer software, which is bug-prone, it was mistrusted by mathematicians for a while. In 2005, B. Werner and G. Gonthier encoded this proof inside the Coq proof assistant [6], reducing the need for trusting software simply to the Coq kernel.

Graphs were first conceived in order to represent a region map. Leonhard Euler asked himself whether it was possible to walk over the seven bridges of the city of Königsberg (see Fig. 1.7) exactly once and end up at the starting place. The Königsberg graph is a region map graph where the regions are

Walks traversing all relations of a graph exactly once and ending up at the starting element are called *Eulerian*.

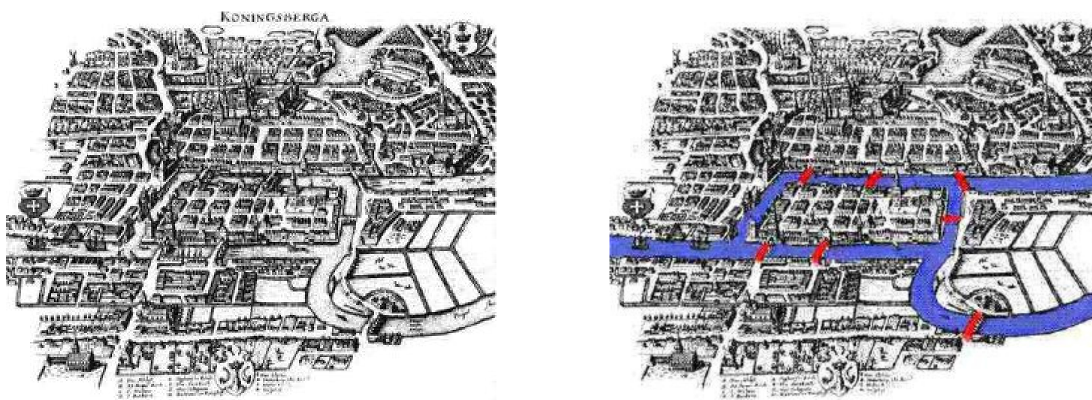
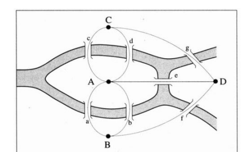


Figure 1.7: The map of the city of Koenigsberg and the seven bridges.

delimited by the shores of the river and the relation is given by the bridges connecting the shores (see side picture); this relation is symmetric. By observing that in Eulerian graphs all elements appear in an even number of relation pairs,



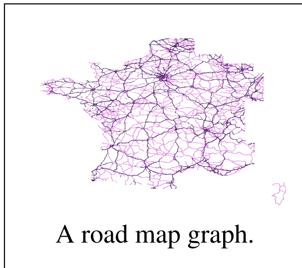
The Königsberg graph.

Euler was able to show in [5] that no walk in the Königsberg graph traversed all bridges exactly once whilst ending at the starting point.

We remark that the Königsberg graph has a distinguishing feature: some relation pairs appear *twice* (e.g. $\{A, C\}$ is an unordered pair appearing twice in the relation because of two bridges connecting shore A with shore C); such graphs are called *multigraphs*. The need for a multigraph arises in this case because the problem requires determining whether it is possible to walk over all bridges: in other words, the application requires information about the *relation* (i.e. the bridges) rather than the elements (i.e. the shores). This is not usually the case: in most of the graph applications shown above[3] the information was associated to the elements rather than the relation itself.

A graph with an irreflexive relation (i.e. $a \not\sim a$ for all elements a) and such that no relation pair appears multiple times is called *simple*.

1.2.4.2 Road maps



A road map graph.

Global Positioning Systems (GPS) exploit the graph representation of a road map in order to compute shortest or fastest paths from any starting place to any destination (see Fig. 1.8). This is a very active research field with applications

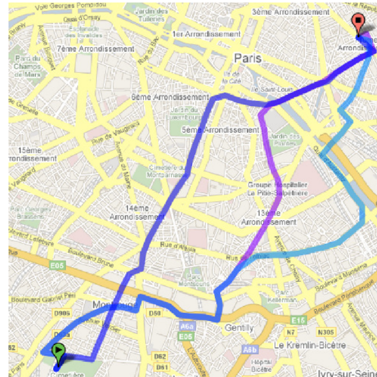


Figure 1.8: Three paths within the road map graph of Paris.

to transportation and logistics [13, 12].

[3] With the notable exception of the web subgraphs in Fig. 1.4-1.5, where a web page can contain several hyperlinks to another page, thus yielding a multigraph.

1.2.5 Molecules and graphs

The word "graph" really comes from the interplay between mathematics and chemistry: it was first introduced in [16], at a time when chemical formulæ were being associated to chemical diagrams expressing the valence of atoms in a molecule. In chemical graphs, the relation between atoms is given by the atomic bonds: this is a symmetric relation.

The *valence* of an atom is the number of bonds the atom is involved in.

Water (H_2O) is usually shown as the graph in Fig. 1.9, left. Methane, CH_4, is shown in Fig. 1.9, right. It appears clear that hydrogen has valence 1, oxygen has valence 2, and carbon has valence 4. Although the chemical graphs shown

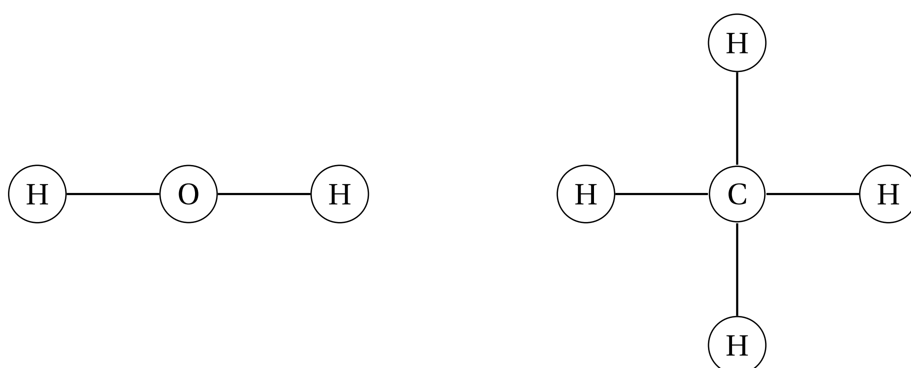


Figure 1.9: Chemical graphs for water and methane.

in Fig. 1.9 are trees, there also exist chemical graphs involving cycles, such as benzene (C_6H_6), shown in Fig. 1.10 (left). More complex molecules based on

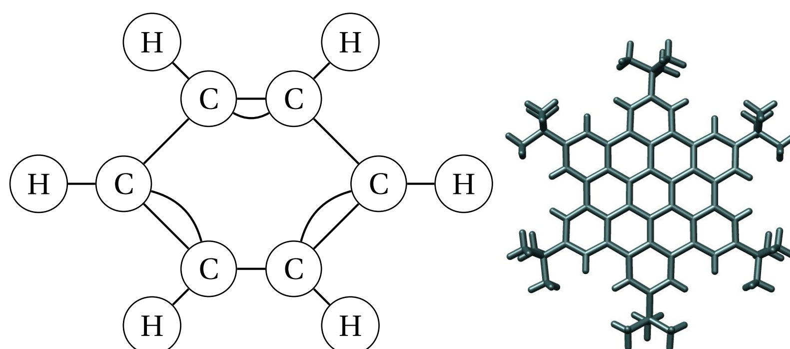


Figure 1.10: Chemical graphs for benzene and a picture of the hexa-peri-hexabenzocoronene molecule.

the hexagonal shape shown in Fig. 1.10 also exist (see[4] Fig. 1.10, right). The

The set of cycles of a graph forms a vector space over the field $\{0,1\}$.

classification of such molecules requires an analysis of the cycle space of the associated graph.

1.2.5.1 Proteins

An *Angstrom* (Å) is a unit of measure corresponding to 10^{-10} meters.

When each pair in a graph relation has an associated numerical value, we say the graph is *weighted*.

Proteins are special types of molecules consisting of a *backbone* to which several *side chains* are attached. The functionality of each protein strongly depends on the shape the protein takes in the three-dimensional space [14]. Let V be the set of atoms of the protein. Finding this shape involves finding a function $x : V \to \mathbb{R}^3$ satisfying a certain number of constraints on the available data. For example, Nuclear Magnetic Resonance (NMR) allows the determination of certain inter-atomic distances within around 5Å. Supposing these data consist in a set of real values d_{uv} for some (unordered) pairs of atoms $\{u, v\}$ in a set E, we can form the *protein graph* consisting of the (symmetric) relation E on the set V. A possible protein shape will then be given by an embedding x satisfying the distance constraints

$$\forall \{u, v\} \in E \quad \|x_u - x_v\| = d_{uv}. \tag{1.1}$$

Naturally, since experimental data can never be precisely measured, and because of certain inherent limitations of the NMR apparatus, Eq. 1.1 have to be replaced by inequalities. Several approaches to solving this problem exist in the literature [11].

Finding the shape of a graph in a Euclidean space is a important task in wireless networks (the localization of wireless sensors can be estimated by means of their mutual distances, obtained by means of the power each sensor uses to communicate with other sensors) and in graph drawing[5]. As concerns the latter, we remark that a graph and its graphical representation on the page are two very different entities, which are unfortunately often confused.

1.3 Exercises

1. Propose a change to Alg. 1 so that it only takes CPU time proportional to n to compute $f(a)$.

[4] http://en.wikipedia.org/wiki/File:Hexa-peri-hexabenzocoronene_ChemEurJ_2000_1834_commons.jpg
[5] http://www.graphdrawing.org

2. Consider Alg. 2: what happens with the call treeFind(2,0)? What happens with the call treeFind(2,5) if node 1 contains a_5 and node 5 contains a_1?

3. Among all types of trees for storing $a_1,\ldots,a_5$, balanced trees yield the best CPU time for reading the value stored in a node. What sort of trees yield the *worst* CPU time?

4. Fig. 1.3 describes the relation on data elements yielded by a sorting operation. What relation best describes the effect of querying a set V of data elements for a specific element v? What if v is not found in V?

5. Do you think the web graph has root nodes? And leaf nodes? Can you *prove* it does?

6. Prove that there exists at least a time instant when the web graph is not a tree.

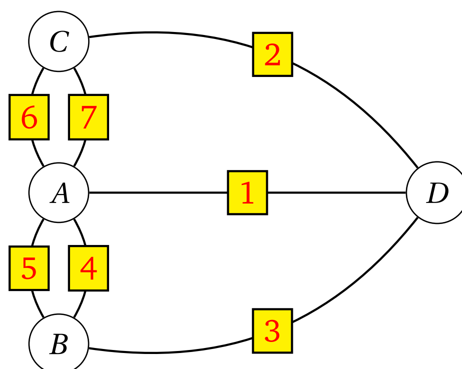
7. The four-colour theorem proves that four colours suffice to colour any map in $\mathbb{R}^2$ in such a way that no two adjacent regions receive the same colour. Prove that two colours suffice to do the same for maps in $\mathbb{R}^1$.

8. Draw the graph on the elements A,B,C,D with relation given by

$$\{A,B\},\{A,C\},\{A,D\},\{B,D\},\{C,D\}$$

with the weights $d_{AB} = d_{AC} = d_{AD} = d_{BD} = d_{CD} = 1$ such that Eq. 1.1 is satisfied.

9. Find a Eulerian walk starting and ending in A and traversing all relation pairs in the graph below.



10. With reference to Exercise 9, add a multiple relation pair $\{A,D\}$ and a pair $\{B,C\}$; then find Eulerian walks starting and ending in A,B,C,D.

11. Consider the graph in Fig. 1.11. and list *all* cycles. You should find ten of them (count the empty cycle, but keep in mind that all cycles must be Eulerian subgraphs).

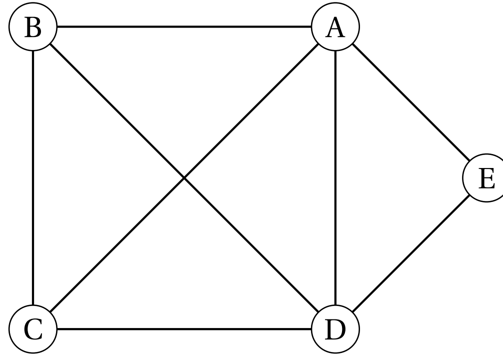
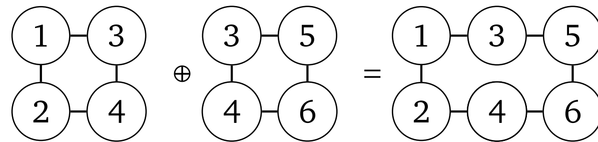


Figure 1.11: Find the ten cycles.

12. Consider the "cycle sum" shown below:



Now consider the set of cycles of the graph in Fig. 1.11 and show that it is a vector space over the field $\{0, 1\}$ under the $\oplus$ cycle operation. [*Hint*: you will need a formal definition of "cycle" which suits your needs and is consistent with the idea of cycle proposed in this chapter.]

13. Take a dozen seconds to look at the graph relations in Fig. 1.12, then answer the following questions: (a) which one has the highest number of relation pairs? (b) which one looks most symmetric? (c) which one looks more complex? Now verify whether your answers were correct.

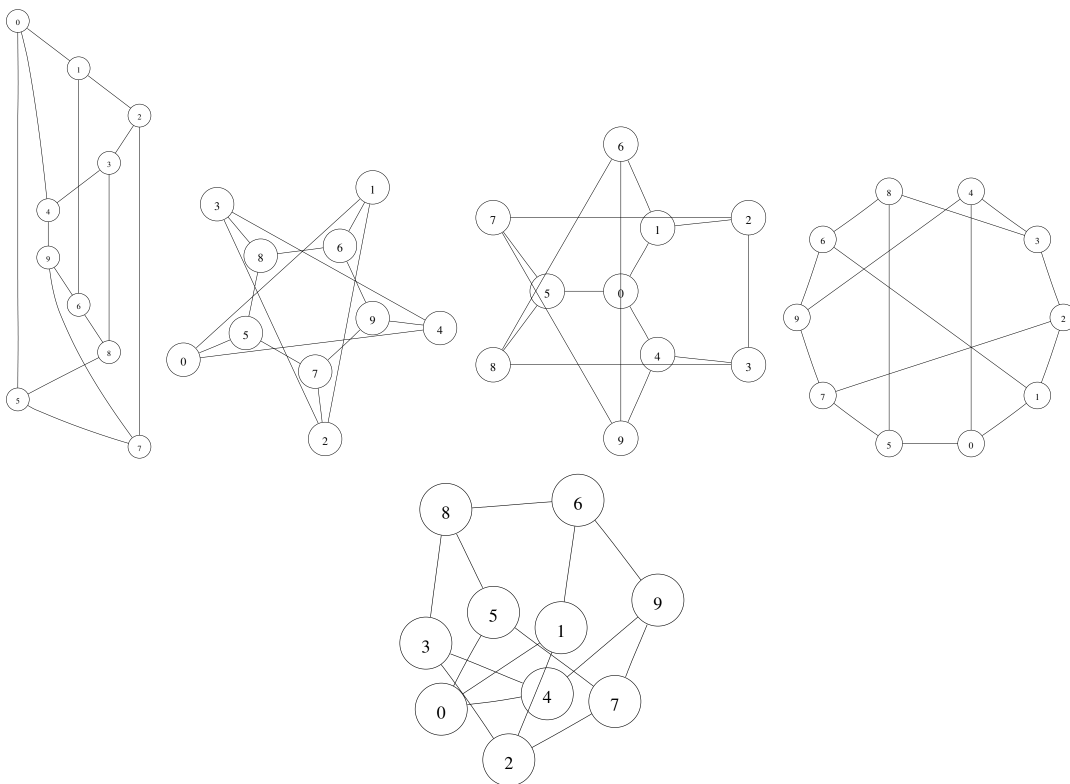


Figure 1.12: Compare these graphs.

CHAPTER

2 Mathematical structures

In this chapter we shall lay out the mathematical foundations for discussing data structures. An important purpose of this chapter is also to establish a formal language which will be used throughout the rest of the book. We recap some well-known mathematical structures which we use repeatedly in this book: sets, functions, sequences, relations, groups, fields, vector spaces. The treatment of these concepts is not completely formalized down to the last detail: the interest is to try and provide a sufficiently solid mathematical foundation to concepts which should already be (at least) intuitively known. The interested reader can consult books in logic [8], set theory [10, 3] and algebra [4].

2.1 The formal language

We write mathematical formulæ as sentences of a formal language over an alphabet A consisting of the following elements.

The mathematical structures discussed in the book can be described with smaller alphabets than A.

- Countably infinitely many variable symbols (e.g. $x, V, v_1, y_3^4, Z, \alpha, \bar{\omega}$ and so on. We never use words, such as var, to denote a mathematical variable, because the word var is written the same as the product of the three symbols v, a, r.
- The relation $\in$.

- Brackets, which are used to emphasize the correct reading order of the sentences.
- The logical connectives $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies) $\neg$ (not).
- The existential quantifier $\exists$ (there exists) and the universal quantifier $\forall$ (for all).

Valid sentences are all and only those that are constructed recursively as follows:

1. variable symbols are valid sentences;
2. if x, y are variable symbols, $x \in y$ is a valid sentence;
3. if P is a valid sentence, (P) is also a valid sentence;
4. if P, Q are valid sentences, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$ and $\neg P$ are also valid sentences;
5. if P is a valid sentence and x is a variable symbol, $\forall x(P)$ and $\exists x(P)$ are also valid sentences.

All other symbols we use are shorthand for valid sentences constructed recursively as above. For example, $P \leftrightarrow Q$ means $P \rightarrow Q \wedge Q \rightarrow P$; $x = y$ means $\forall z(z \in x \leftrightarrow z \in y)$. Other important shorthand symbols are $\cap$ (set intersection), $\cup$ (set union), $\smallsetminus$ (set difference).

2.2 Sets

We take the formal approach to sets proposed by the Zermelo-Fraenkel list of axioms with the Axiom of Choice: in short, the ZFC theory. In particular, in this theory the "universe" of sets is given by the well-founded sets [10]. Limiting the attention to $\mathbb{WF}$ allows us disregard two "nasty" questions: (a) is there anything which is not a set? (b) is there a set x containing itself as an element? Since we only consider sets in $\mathbb{WF}$, and since $\mathbb{WF}$ only contain sets by construction, the answer to (a) is no: everything in our universe is a set. As for question (b), since every set in $\mathbb{WF}$ is obtained recursively as a power set operation on

The class $\mathbb{WF}$ of *well-founded* sets constructed by starting with the empty set $\varnothing$ and recursively applying the "power set" operation $\mathscr{P}$: $\mathscr{P}(x)$ is the set consisting of all subsets of x.

some existing set, and since the recursion starts with the empty set, no set can contain itself.

The set $\mathbb{N}$ of natural numbers is constructed in $\mathbb{WF}$ as follows:

$$\begin{array}{rll} \varnothing & \text{is called} & 0 \\ \{0\} & \text{is called} & 1 \\ \{0,1\} & \text{is called} & 2 \\ \{0,1,2\} & \text{is called} & 3, \end{array}$$

and so on. If written out explicitly, 3 means $\{\varnothing,\{\varnothing\},\{\varnothing,\{\varnothing\}\}\}$. Although this notation is very cumbersome, the definition above is consistent with the intuitive interpretation of the natural numbers. The natural number b is a *successor* of the natural number a if $b = \{0,1,\ldots,a\}$. The class $\mathbb{N}$ is also a well-founded set, namely $\mathbb{N} = \{0,1,2,\ldots\}$, which is also denoted by ω.

The set $\varnothing$ is defined as the only set in $x \in \mathbb{WF}$ satisfying $\forall y\,(y \not\in x)$.

If b is the successor of a then a is the *predecessor* of b and a, b are *consecutive*.

2.3 Functions

Given sets $x, y \in \mathbb{WF}$, the *pair* set $\{x, y\}$ is also in $\mathbb{WF}$ (by the Pairing Axiom of ZFC [10]). The set $\{x,\{x, y\}\}$ is called an *ordered pair* and denoted by (x, y). A *function* f from a set X to a set Y, denoted $f : X \to Y$, is a set of ordered pairs (x, y) where $x \in X$ and $y \in Y$, and such that for any $x \in X$ there is at most one $y \in Y$ such that $(x, y) \in f$. Thus, we can denote a pair $(x, y) \in f$ by

$$f(x) = y.$$

We denote the subset $X' \subseteq X$ such that for each $x' \in X'$ there exists a $y \in Y$ with $f(x') = y$ the *domain* of f, denoted by $\operatorname{dom} f$. We denote the subset $Y' \subseteq Y$ such that for each $y' \in Y'$ there exists a $x \in X$ with $f(x) = y'$ the *range* of f, denoted by $\operatorname{ran} f$.

A function $f : X \to Y$ is *injective* if

$$\forall u, v \in X \quad (u \neq v \to f(u) \neq f(v))$$

Injective (resp. surjective) functions are also called *one-to-one*, or *1-to-1* (resp. *onto*).

and *surjective* if

$$\forall y \in Y \exists x \in X \quad (f(x) = y).$$

A function is a *bijection* if it is both injective and surjective.

If X, Y, Z are three sets, and $f : X \to Y$ and $g : Y \to Z$ are two functions, the function $g \circ f : X \to Z$ given by

$$\forall x \in X \quad (g \circ f)(x) = g(f(x))$$

is defined whenever $\mathrm{dom}(g) \supseteq \mathrm{ran}(f)$, and is called the *composition* of g and f. The *identity function* is a bijection $\mathbf{1} : X \to X$ such that $\forall x \in X\ \mathbf{1}(x) = x$. If $f : X \to Y$ and $g : Y \to X$ are bijections and $g \circ f = \mathbf{1}$, then g is the *inverse* of f. Every bijection $f : X \to Y$ has a unique inverse g (denoted by f^{-1}), mapping $Y \to X$ and defined by setting $g(f(x)) = x$.

If $f^{-1} = g^{-1}$ for any two bijections $f, g : X \to Y$, then $f = g$.

Informally speaking, the cardinality of a set is the number of its elements. The formal definition involves establishing a bijection between the set whose cardinality must be established and a set whose cardinality is already known [10]: the two sets are then defined to have the same cardinality. Since in this book we mostly deal with finite sets, it suffices to find a bijection between a given set S and sets $n \in \mathbb{N}$: the cardinality of S is then defined to be n (this is denoted by $|S| = n$).

2.4 Sequences

A *sequence* a on a set S is an injective function $a : \mathbb{N} \to S$. A sequence a is *finite* if $\exists \ell \in \mathbb{N}$ such that $|\mathrm{dom}\, a| = \ell$. As a rule, $\mathrm{dom}\, a$ is a set of consecutive natural numbers, starting with either 0 or 1. The *length* of a sequence a is the cardinality of its domain, denoted by $|a|$. For any ordered pair $(i, s) \in a$ (where $i \in \mathrm{dom}\, a$ and $s \in \mathrm{ran}\, a$), instead of using the function notation $a(i) = s$ we emphasize the sequence definition by writing $a_i = s$, and denote the sequence a, indexed from 1 on consecutive natural numbers and of length ℓ, as:

$$a = (a_1, a_2, \ldots, a_\ell).$$

Thus, a_i denotes the i-th element of the sequence a, and i is the *index* of the element a_i.

As remarked in Sect. 1.1, the computer memory can be represented by a finite sequence $\mathbb{M} = (a_0, a_1, \ldots, a_M)$ of length $M + 1$, where $\mathscr{A}$ is an alphabet. As such, sequences are the most fundamental data structures. The index of a character in memory is also called a *pointer*. Although pointers are absent in Java, they are one of the main strengths of C, C++ and several other computer languages, as they allow direct access to the content of the computer memory.

An *alphabet* is a non-empty finite set. The name comes from the context: alphabet elements are called *characters*; sequences of characters are called *words*, sequences of words are called *sentences*.

2.4.1 Cartesian products

Consider a set $\mathscr{S}$ of sequences of length $k \in \mathbb{N}$ on S, all indexed on the set $K = \{1,\ldots,k\}$. For all $i \in K$, we define $\pi_i(\mathscr{S}) = \{a_i \mid a \in \mathscr{S}\}$. We also denote $\mathscr{S}$ as:

$$\pi_1(\mathscr{S}) \times \cdots \times \pi_k(\mathscr{S}),$$

and call it the *Cartesian product* of $\pi_1(\mathscr{S}),\ldots,\pi_k(\mathscr{S})$. For each $i \in K$, the set $\pi_i(\mathscr{S})$ is called the *projection* of $\mathscr{S}$ on the i-th coordinate.

The Euclidean space $\mathbb{R}^3$ is the Cartesian product $\mathbb{R}\times\mathbb{R}\times\mathbb{R}$.

Debuggers are computer programs that can monitor the execution of another computer program. This is useful to find bugs that arise at execution time. A debugger can be instructed to watch the value of the memory at a certain address (say i) and stop the execution if the value stored at that address belongs to a certain pre-specified range (e.g. stop if memory byte i contains an ASCII character between a and z). This stopping condition can be written by means of a projection as $\pi_i(\mathbb{M}) \cap \{\text{a},\ldots,\text{z}\} \neq \varnothing$.

A *byte* stores a binary number between 0 and 11111111, i.e. 0 and 255.

ASCII stands for American Standard Code for Information Interchange, and is a function mapping the set $\{0,\ldots,255\}$ to an alphabet.

2.5 Relations

A k-ary relation on a set S is a set R consisting of sequences of S having length k. We shall mostly deal with binary relations, i.e. sets of ordered pairs of elements of S. We denote relation pairs $(a,b) \in R$ by aRb. A relation $\sim$ on S is *reflexive* if $a \sim a$ for all $a \in S$, and *irreflexive* if $a \not\sim a$ for all $a \in S$. A relation $\sim$ is *symmetric* if $a \sim b$ implies $b \sim a$, and *antisymmetric* if $a \sim b$ implies $b \not\sim a$. A relation $\sim$ is *transitive* if $a \sim b$ and $b \sim c$ imply $a \sim c$. A reflexive, symmetric and transitive relation is an *equivalence* relation.

Warning: a binary relation is *not* a function $S \to S$. A relation might contain two pairs (s,t) and (s,u) with $t \neq u$, whereas in a function f, for each s there can be at most one t with $f(s) = t$. On the other hand, a function $S \to S$ *is* a binary relation on S.

For example, the relation "a is a predecessor of b" for $a,b \in \mathbb{N}$ contains the pairs $(0,1),(1,2),(2,3)\ldots$, is irreflexive, antisymmetric and not transitive. The relation "b is a successor of a" contains the pairs $(1,0),(2,1),(3,2)\ldots$, is irreflexive, antisymmetric and not transitive. The union of these two relations is also a relation which contains the pairs $(0,1),(1,0),(1,2),(2,1),(2,3),(3,2)$ and so on. These relation is irreflexive, symmetric and not transitive, and corresponds to the concept that "a,b are consecutive".

Consider the set $S = \{1,2,3,4,5\}$ under the predecessor relation

$$P = \{(1,2),(2,3),(3,4),(4,5)\}$$

(a graphical representation of this relation is given in Fig. 1.3). This relation is not transitive: for example $(1,2),(2,3) \in P$ but $(1,3) \not\in P$. Since intransitivity is due to missing pairs, we might consider enriching the relation with more pairs until it becomes transitive. The resulting relation is called the *transitive closure* and is transitive by definition. In this example, the missing pairs are $P' = \{(1,3),(1,4),(1,5),(2,4),(2,5),(3,5)\}$. The transitive closure of P, denoted by

Transitive closures can also be defined for graphs; we shall see that it amounts to essentially the same things as for relations.

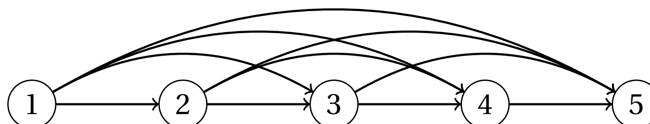


Figure 2.1: A graph representing the < relation on $\{1,2,3,4,5\}$.

$\text{trcl}(P)$, is $P \cup P'$, shown in Fig. 2.1. We remark that $\text{trcl}(P)$ is $<$, the ordinary "less than" relation on natural numbers.

2.5.1 Equivalence relations and set partitions

Given an equivalence relation $\sim$ on a finite set S and an element $x \in S$ we denote by $\text{eqcl}^{\sim}(x)$ the *equivalence class* of x with respect to $\sim$. This is the set of all $y \in S$ such that $y \sim x$. We define:

We call $S/\sim$ a *quotient set* (quotient modulo $n \in \mathbb{N}$ also yields a set of equivalence classes: $\{0, n, 2n, \ldots\}$, $\{1, n+1, 2n+1, \ldots\}$ and so on).

$$S/\sim = \{\text{eqcl}^{\sim}(x) \mid x \in S\}.$$

We prove that $S/\sim$ forms a partition of S. Let $x \neq y \in S$ and suppose the intersection $\text{eqcl}^{\sim}(x) \cap \text{eqcl}^{\sim}(y)$ is non-empty and contains the element z. Then z is $\sim$-equivalent to all the elements of the equivalence class of x and to all those of the equivalence class of y. By transitivity, $\forall t \in \text{eqcl}^{\sim}(y)$ $(t \in \text{eqcl}^{\sim}(x))$ and $\forall t \in \text{eqcl}^{\sim}(x)$ $(t \in \text{eqcl}^{\sim}(y))$, thus establishing that $\text{eqcl}^{\sim}(x) = \text{eqcl}^{\sim}(y)$. Therefore, if two equivalence classes are distinct, they must have empty intersection.

$\mathscr{S} = \{A_1, \ldots, A_k\}$ is a *partition* of S if: (a) $\forall i \le k$ $(A_i \subseteq S)$; (b) $S = \bigcup_{i=1}^{K} A_i$; (c) $A_i \cap A_j = \varnothing$ for all $i \neq j$.

Conversely, each partition $\mathscr{S} = \{A_1, \ldots, A_k\}$ of a set S induces a relation $\sim$, defined so that:

$$\forall i \in \{1, \ldots, k\}\ \forall a, b \in A_i \quad (a \sim b). \tag{2.1}$$

We prove that $\sim$ is an equivalence relation: setting $b = a$ in Eq. 2.1 yields reflexivity and interchanging a, b yields symmetry. As for transitivity, suppose $a \sim b$ and $b \sim c$, and assume $a \in A_i$ for some $i \le k$. Since $a \sim b$ we have $b \in A_i$ and since $b \sim c$ we have $c \in A_i$ too, whence, setting $b = c$ in Eq. 2.1, we have $a \sim c$.

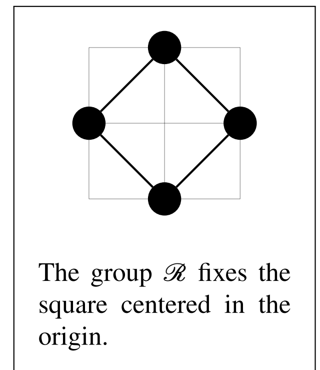
2.6 Groups

A *group* is a well-founded set G together with a function $\otimes : G^2 \to G$ called *product*. For $g, h \in G$ we denote $\otimes(g, h)$ simply by gh. The group product satisfies the following conditions:

- $\forall a, b, c \in G\ (ab)c = a(bc)$ [ASSOCIATIVITY]
- there is a unique element $e \in G$ such that for all $g \in G$ we have $eg = ge = g$ [IDENTITY]
- for each $g \in G$ there is a unique element $h \in G$ (denoted by g^{-1}) such that $gg^{-1} = g^{-1}g = e$ [INVERSE].

In general, gh might be different from hg. If $gh = hg$ for any $g, h \in G$, the group is called *abelian*.

For example, the set $\mathscr{R}$ of vector rotations around the origin by the angles 0, $\pi/2$, π, $3\pi/2$ forms a group under composition, with identity 0, where $(\pi/2)^{-1} = 3\pi/2$ and $\pi^{-1} = \pi$. The set $\mathbb{F}_n = \{m \pmod n \mid m \in \mathbb{Z}\}$ of integers modulo a positive integer n is a group under addition $\pmod n$ with identity 0, as for every $m \in \mathbb{Z}$, $m \pmod n + (-m) \pmod n = 0$. The set $\mathbb{F}_p^* = \{m \pmod p \mid m \in \mathbb{Z} \wedge m \pmod p \neq 0\}$ of *nonzero* integers modulo a prime number p: this set is a group with identity 1 under multiplication $\pmod p$. For $p = 5$ we obtain the following *multiplication table* (both rows and columns indexed by group elements g, h identifying a table entry containing the product gh):



The group $\mathscr{R}$ fixes the square centered in the origin.

	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

All these groups are abelian.

2.6.1 Permutations

A *permutation* is a bijection from a set V to itself. We shall limit our interest to *finite* permutations, i.e. such that $|V|$ is finite, and usually $V = \{1,\ldots,n\}$.

We denote a permutation π on the set $[n] = \{1,\ldots,n\}$ by listing the action of the permutation on each element on $[n]$, for example:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

sends $1 \to 2$, $2 \to 3$, $3 \to 4$ and $4 \to 1$. The product of π by the permutation $\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix}$, defined by applying σ first and π later, and denoted as $\pi\sigma$, has the following effect:

$$\begin{array}{ccc} 1 \xrightarrow{\sigma} & 4 & \xrightarrow{\pi} 1 \\ 2 \xrightarrow{\sigma} & 3 & \xrightarrow{\pi} 4 \\ 3 \xrightarrow{\sigma} & 2 & \xrightarrow{\pi} 3 \\ 4 \xrightarrow{\sigma} & 1 & \xrightarrow{\pi} 2, \end{array}$$

i.e. it is the permutation:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 2 \end{pmatrix}.$$

The product of permutations maps an ordered pair of permutations to a permutation. Whenever an operation mapping from a set product $V \times V$ to a set U is such that $U \subseteq V$, we say that the operation is *closed*.

We remark that the product of permutations is a composition of bijections. Since the composition of two bijections on the same set is another bijection on that set, the product of two permutations is still a permutation. Proving that the product of permutations is associative is easy but long tedious. The identity is the permutation $e = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix}$, and the inverse of each permutation is obtained by simply "reversing the arrows": if a permutation π sends i to j, then π^{-1} sends j to i. In other words, this means that π^{-1} sends $\pi(i)$ to i, and therefore that $\pi^{-1}(\pi(i)) = i$ for all $i \in [n]$, which implies that $(\pi^{-1}\pi)(i) = i$, i.e. that $\pi^{-1}\pi = e$. The group of all permutations on $[n]$ is denoted S_n. The group of all permutations on a finite set V is denoted by $\mathrm{Sym}(V)$. We remark that

$$|\mathrm{Sym}(V)| = |V|!.$$

2.6.2 Cycle permutations

The integer ℓ is the *length* of the cycle.

A *cycle permutation* (or simply a *cycle*) is a permutation $\pi \in \mathrm{Sym}(V)$ with a sequence $(v_1,\ldots,v_\ell)$ of elements of V such that $\pi(v_i) = v_{i+1}$ for all $i < ell$ and $\pi(v_\ell) = v_1$, and $\pi(v) = v$ for all other elements $v \in V \smallsetminus \{v_1,\ldots,v_\ell\}$. Informally, the action of π on V is described graphically in Fig. 2.2 for a case where $\ell = 6$.

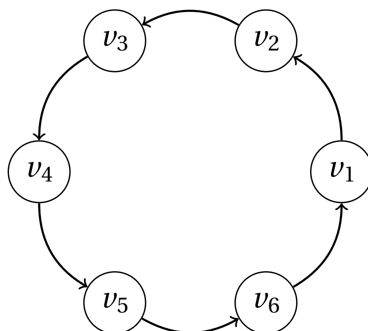


Figure 2.2: The action of a cycle permutation.

Cycles allow a more compact way of writing permutations. The permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix},$$

for example, only swaps 1 and 2 but still takes 9 pairs of integers to write down: this is wasteful. But we can easily recognize that π is the cycle of length 2 sending $1 \to 2$ and $2 \to 1$ and fixing all the other elements of $[n]$. We therefore write π as $(1,2)$. In general, a cycle permutation $\pi \in \mathrm{Sym}(V)$ sending $\pi(v_i)$ to v_{i+1} for all $i < \ell$ and $\pi(v_\ell)$ to v_1 is denoted by its defining sequence $(v_1, \ldots, v_\ell)$.

Let $\pi = (v_1, \ldots, v_h)$ and $\sigma = (u_1, \ldots, u_k)$ be two cycles in $\mathrm{Sym}(V)$. If these two cycles list no common elements, then $\pi\sigma$ simply sends $v_i \to v_{i+1}$ for $i < h$, $u_i \to u_{i+1}$ for $i < k$, $v_h \to v_1$ and $u_k \to u_1$. In other words, the actions of π and σ are *disjoint*. As a consequence $\pi\sigma = \sigma\pi$. We write the product of two disjoint cycles by simply juxtaposing the two cycles, namely:

Disjoint cycles commute. This is false for permutations in general.

$$(v_1, \ldots, v_h)(u_1 \ldots, u_k).$$

If π, σ have some common elements, this analysis no longer holds. For example, if $\pi = (1,2,3)$ and $\sigma = (1,2)$, $\pi\sigma$ has the following effect (we apply σ first and π later): $1 \to 2 \to 3$, $2 \to 1 \to 2$, $3 \to 3 \to 1$, which we can write as $(1,3)$. What is true, however, is that any product of non-disjoint cycles can be written as a product of (possibly different) disjoint cycles, and moreover that any permutation can be written as a product of disjoint permutations in a unique way apart from the order of the factors (see [4], p. 59).

2.7 Fields

A *field* is a set F together with two functions: one, $\oplus : F^2 \to F$, called *sum*, and another, $\otimes : F^2 \to F$, called *product*. For $a, b \in F$ we denote $\oplus(a, b)$ by $a + b$ and $\otimes(a, b)$ by ab. The field operations satisfy the following conditions:

- $(F, \oplus)$ is an abelian group with identity symbol 0
- $(F \smallsetminus \{0\}, \otimes)$ is a group with identity symbol 1
- the product distributes over the sum:

$$\forall a, b, c \in F \quad (a + b)c = ac + bc \wedge a(b + c) = ab + ac.$$

Examples of infinite fields are the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$ and the complex numbers $\mathbb{C}$. For every positive prime integer p, the set $\mathbb{F}_p = \{0, 1, \ldots, p-1\}$ is a finite field with respect to addition and multiplication $\pmod p$: we already discussed the additive and multiplicative groups $\mathbb{F}_p, \mathbb{F}_p^*$ in Sect. 2.6, and it is easy to show distributivity.

The finite field $\mathbb{F}_2$ has a special importance in computer science, as it allows operations over the two values of “true” and “false” (often interpreted as “presence” or “absence”).

2.8 Vector spaces

A *vector space* over a field F is an additive group V together with a field F with an operation $\odot : F \times V \to V$, called *scalar multiplication*, where $a \odot v$ is commonly denoted av for all $a \in F, v \in V$, which satisfies the following conditions:

- $\forall a, b \in F$ and $x \in V$, $a(bx) = (ab)x$;
- $\forall a, b \in F$ and $x \in V$, $(a + b)x = ax + bx$;
- $\forall a \in F$ and $x, y \in V$, $a(x + y) = ax + ay$;
- $\forall x \in V$, $1x = x$.

For any given field F, the set of all sequences of the same length $n \in \mathbb{N}$ with elements in F forms a vector space over F, under the vector addition $(x_1,\ldots,x_n)+(y_1,\ldots,y_n) = (x_1+y_1,\ldots,x_n+y_n)$ for all $x = (x_1,\ldots,x_n)$ and $y = (y_1,\ldots,y_n)$ in F^n, and the scalar multiplication $a(x_1,\ldots,x_n) = (ax_1,\ldots,ax_n)$ for all $a \in F$ and $x = (x_1,\ldots,x_n) \in F^n$. This vector space is simply denoted by F^n (with the same name as the underlying set) and its sequences are called *vectors*.

A *bit* (which stands for BInary digiT) is a memory box that can store either a 0 or a 1. A *byte* is a sequence of 8 bits. Bytes form a vector space over $\mathbb{F}_2$.

2.9 Exercises

1. Prove that if f, g are two bijections $V \to V$, then $f \circ g$ is also a bijection $V \to V$.

2. Prove that if $f^{-1} = g^{-1}$ for any two bijections $f, g : X \to Y$, then $f = g$.

3. Prove that the product of permutations is associative.

4. Prove that for any permutation π of $[n]$, $e\pi = \pi e = \pi$, that π^{-1} is a permutation, and that $\pi\pi^{-1} = e$.

5. Prove that any permutation on V is a sequence of elements in V, and show that not every sequence of elements in V is a permutation.

6. Show that not all permutations commute.

CHAPTER

3 Graphs

Graphs are a useful way to represent binary relations. They provide a visual way to picture a relation; but this feature may somtimes be a limitation, as Exercise 13 in Sect. 1.3 shows. There are infinitely many different ways to draw the same graph on paper; different drawings suggest different graph properties, but this is simply false. If we consider two different drawings of the same graph to be equivalent, then graphs might be interpreted as equivalence classes of all their possible drawings.

See Sect. 1.2.5.1.

In the framework of data structures in computer science, graphs are used to represent several relations, the main of which being the *pointer* relation: if a program variable v holds the memory address of the memory box storing the value currently held by another program variable u, then v is a *pointer* for u. Pointers define a relation on the set of all program variables; this relation is generally irreflexive, unsymmetric and intransitive.

See Sect. 2.4.

Graphs come with their own terminology. The main aim of this chapter is to get the reader acquainted with that terminology. More complete treatments of the topics below can be found in several textbooks, e.g. [2, 7, 9, 15].

3.1 Graphs and digraphs

Since a binary relation is a set of ordered pairs (see Sect. 2.5), we should indicate an edge between the vertices $u, v \in V$ by $\{(u,v),(v,u)\}$, but we employ the more convenient notation $\{u,v\}$.

A *graph* is defined as a pair $G = (V, E)$ where V is any set, called the *vertex* set, and E is a symmetric binary relation on V called the *edge* set. Examples of graphs are given in Fig. 3.1. If the graph is simply given as G, then we denote by $V(G)$ its vertex set and by $E(G)$ its edge set.

Figure 3.1: Examples of graphs.

Directed graphs are also called *digraphs*.

We denote an arc on two nodes $u, v \in V$ by (u, v).

A *directed graph* is formally defined as a pair $G = (V, A)$ where V is any set, called the *node* set, and A is a binary relation on V, called the *arc* set. Examples of digraphs are given in Fig. 3.2.

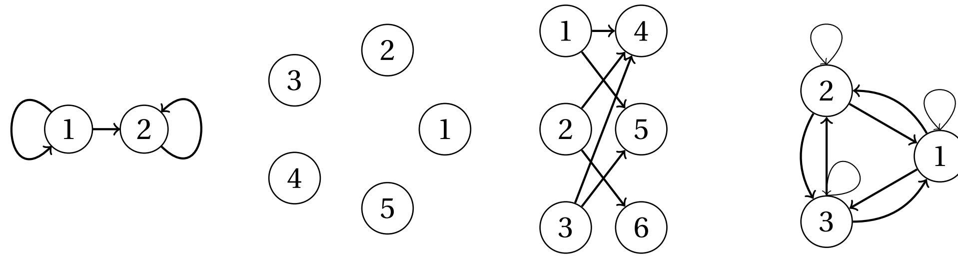


Figure 3.2: Examples of digraphs.

Digraphs with all possible arcs (u, v) aside from loops are also called complete.

The first digraph from the left exhibits *loops* on the nodes, i.e. arcs of the type (v, v) (where v is a node). Graphs and digraphs without loops are called *loopless*, and correspond to irreflexive relations. The second graph is a *stable*, i.e. the arc set defined on the nodes is empty. The third is *bipartite*, i.e. the node set can be partitioned in two sets A, B such that $A \cup B = V$, $A \cap B = \varnothing$ and both A and B are stables (i.e. no arc exists within nodes in A, nor within nodes in B). The fourth digraph is *complete*, i.e. its arc set includes all possible arcs.

The outgoing star is also called the *outstar*, denoted by $N^+(v)$; the incoming star is also called the *instar*, denoted by $N^-(v)$.

Given a digraph $G = (V, A)$ and a node $u \in V$, the node set $\{v \in V \mid (u, v) \in A\}$ is called the *outgoing star* of u. The node set $\{v \in V \mid (v, u) \in A\}$ is called the *incoming star* of u, and is denoted by $N^-(v)$. See Fig. 3.3. The *outdegree* of a node $v \in V$ is the cardinality of its outstar, and similarly, the *indegree* of $v \in V$ is the cardinality of its instar. In Fig 3.3, both the indegree and the outdegree of node 7 are equal to 3. For $u \in V$, the arc set $\{(u, v) \mid (u, v) \in A\}$ is denoted by $\delta^+(u)$, and the arc set $\{(v, u) \mid (v, u) \in A\}$ is denoted by $\delta^-(u)$. Given a digraph,

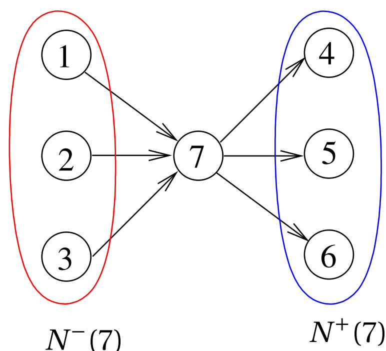


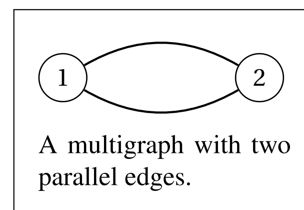
Figure 3.3: Instar and outstar of node 7: $N^-(7) = \{1,2,3\}$ and $N^+(7) = \{4,5,6\}$.

its *underlying graph* replaces every arc (u, v) with the corresponding edge $\{u, v\}$ (see Fig. 3.4).

Figure 3.4: A digraph and its underlying graph.

If (u, v) is an arc, then v is *adjacent* to u, and both u, v are *incident* to the arc (u, v). Moreover, u is the *head* of the arc and v its *tail*. Both u, v are *endpoints* of the arc. If $\{u, v\}$ is an edge, then u, v are adjacent to each other and incident to the edge; both are endpoints of the edge. Arcs/edges are incident to the nodes/vertices that define their endpoints.

Informally, a *multigraph* is like a graph (or a digraph) which has several edges (or arcs) between the same pair of vertices (nodes). Such edges/arcs are called *parallel*. Formally, we define an arc of a multigraph as a triplet (u, v, k) where u, v are the nodes incident to the arc, and $k \in \mathbb{N}$. No two parallel arcs have the same value of k. Graphs/digraphs without loops and parallel edges/arcs are called *simple*.



A multigraph with two parallel edges.

In the following, definitions given for graphs often apply to digraphs and multigraphs with trivial adaptations: we shall specify when this fails to be the case. As a rule of thumb, in theoretical computer science and combinatorics graphs are very common, digraphs slightly less, and multigraphs occur rarely.

In problems arising from practical applications, however, you may have to deal with loops and parallel edges.

Most graphs/digraphs are simple.

3.2 Subgraphs

Very often, problems related to graphs involve finding a subgraph of a certain type in a given graph. This is the case, for example, whenever finding a shortest path or a spanning tree of a graph (see below). Given a graph $G = (V, E)$, a *subgraph* $H = (U, F)$ of G is a graph H such that $U \subseteq V$ and $F \subseteq E$. Notice that once the edge set F is given, the set U can be retrieved by simply taking the set of all vertices appearing in edges of F. Thus, subgraphs are often taken to be sets of edges of the original graph. Some examples of interesting subgraphs are shown in Fig. 3.5.

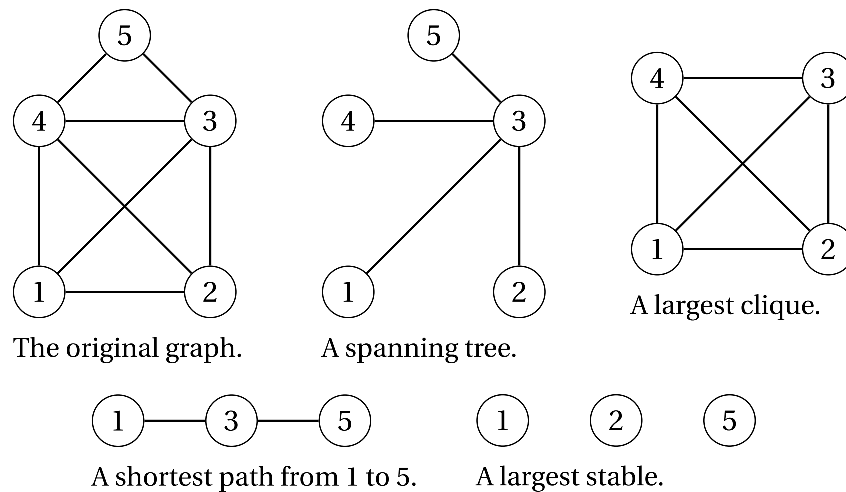


Figure 3.5: Examples of subgraphs.

A subgraph H of G is *spanning* whenever $V(H) = V(G)$ (see the spanning tree example in Fig. 3.5).

3.3 Walks, paths and cycles

A *directed walk* in a digraph $G = (V, A)$ is a sequence $p = (v_1, \ldots, v_k)$ of nodes in V such that $(v_i, v_{i+1}) \in A$ for all $i \in [k-1]$.

A directed walk is also called *diwalk*.

Recall $[k-1] = \{1, \ldots, k-1\}$.

3.4 Trees

3.5 Stables and cliques

3.6 Operations on graphs

3.6.1 Graph complement

3.6.2 Line graph

3.6.3 Induced subgraph

3.6.4 Subgraph contraction

3.7 Exercises

1. Can you imagine a useful situation for a reflexive pointer relation between program variables? What about symmetric? What about transitive?

2. Give a formal definition of parallel edges (we only defined parallel arcs in the text).

CHAPTER

4 Data structures

Data structures are abstract entities conceived to store, relate and manipulate data. In this section we present a formal view of data structures. In short, a data structure is a set of memory cells, with a function mapping each cell to the datum it stores, and with a pointer relation on the cells.

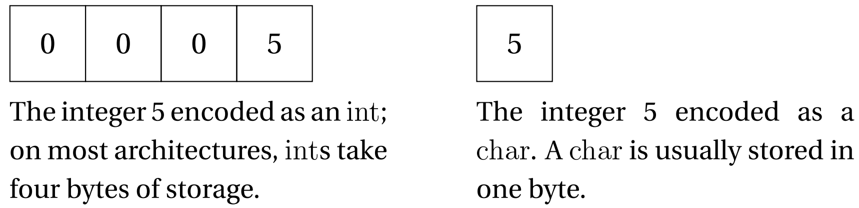
A *memory unit* or *cell* represent a single unit of storage capacity in the computer.

4.1 Types

In most programming languages, data are *typed*: for example, the data item 5 could be assigned an int type (which explicitly states that the symbol 5 is to be considered an integer) or a char type (which states that the symbol 5 is to be interpreted as the fifth character in the ASCII table in the present context), see Fig. 4.1. Types provide the most basic kind of semantic information about the data processed by the computer. Among other things, they are used by the operating system in order to decide how much memory to allocate to data storage, and how to carry out certain operations on these data.

Given a set $\mathbb{D}$ of data items and a set $\mathbb{T}$ of type names, a *data type* is a function $\tau : \mathbb{D} \to \mathbb{T}$.

Most imperative languages have the following elementary types: integer, usually denoted by int or long depending on the size of the integer being stored; floating point, usually denoted by float or double depending on the the size; and character, denoted by char. Several modern languages also include elementary types for boolean values, denoted by boolean, accented characters, and others.



The integer 5 encoded as an int; on most architectures, ints take four bytes of storage.

The integer 5 encoded as a char. A char is usually stored in one byte.

Figure 4.1: Different types yield different encodings. The cases represent memory units.

In C/C++, the unknown data type is denoted by void, whereas in Java it is denoted by Object. Their precise semantics is different.

Several languages include a catch-all type used to specify an "unknown type": when type decisions are taken at run-time, it might happen that the type of a datum is unknown until further analysis has taken place.

4.2 The main definition

We assume the set of data items to be processed by the computer to be $\mathbb{D}$, with type set $\mathbb{T}$ and type function τ. We also assume $\mathbb{D}$ contains the basic data items $\varnothing$ (the empty set), and the elements of the boolean set $\mathbb{B} = \{\text{true}, \text{false}\}$.

A *data structure* is a quintuplet (G, D, O_G, O_D, O_R), where:

- G is a digraph $G = (V, A)$: its nodes model the memory cells, and its arcs the pointer relations between them;
- the function $D : V \to \mathbb{D}$, called the *storage function*, associates graph nodes to data elements;
- the set O_G of *graph operations* is a finite set of functions which map the pair G to another digraph G';
- the set O_D of *data operations* is a set of functions which map D to another storage function D' on the same set V;
- the set O_R of *read operations* is a finite set of functions which map (G, D) to an element in $V \cup A \cup \operatorname{ran} D$.

For example, the array $(1,3,5)$ can be stored by the data structure (P, D, O_G, O_D, O_R) such that:

- P is the directed path $P = (V, A)$ where $V = \{1, 2, 3\}$ and $A = \{(1, 2), (2, 3)\}$;
- D is the function $1 \to 1 \wedge 2 \to 3 \wedge 3 \to 5$;
- O_G only contains the function mapping G to the empty graph $\varnothing$ (this corresponds to deleting the data structure from memory);
- O_D contains all mappings of D to any function $D' : V \to \mathbb{D}$, e.g. writing the integer 2 in node 1 corresponds to mapping D to the function $1 \to 2 \wedge 2 \to 3 \wedge 3 \to 5$;
- O_R contains functions $\mathrm{get}_v : \mathscr{D}_V \to \mathbb{D}$ for each $v \in V$ given by $\mathrm{get}_v(D) = D(v)$ (this corresponds to reading the data element stored in v.

For our definition to make sense, we also need to remark that graph operations changing V must necessarily be paired with a data operation which changes $D : V \to \mathbb{D}$ accordingly.

Although it is always useful to formalize concepts so as attempt to eliminate all ambiguities, in the rest of the book we shall revert to using graphical representations and descriptive names in order to describe graph, data and read operations on data structures.

4.3 Arrays

4.4 Lists

4.5 Queues

4.6 Hash maps

4.7 Trees

Bibliography

[1] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5):711–712, 1976.

[2] C. Berge. *Graphes et hypergraphes.* Dunod, Paris.

[3] K. Ciesielski. *Set Theory for the Working Mathematician.* Cambridge University Press, Cambridge, 1997.

[4] A. Clark. *Elements of Abstract Algebra.* Dover, New York, 1984.

[5] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiæ Scientiarum Imperialis Petropolitanæ*, 8:128–140, 1736.

[6] G. Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[7] F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA, second edition, 1971.

[8] P. Johnstone. *Notes on logic and set theory.* Cambridge University Press, Cambridge, 1987.

[9] B. Korte and J. Vygen. *Combinatorial Optimization, Theory and Algorithms.* Springer, Berlin, 2000.

[10] K. Kunen. *Set Theory. An Introduction to Independence Proofs.* North Holland, Amsterdam, 1980.

[11] L. Liberti, C. Lavor, A. Mucherino, and N. Maculan. Molecular distance geometry methods: from continuous to discrete. *International Transactions in Operational Research*, 18:33–51, 2010.

[12] G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional a^* search on time-dependent road networks. *Networks*, accepted.

[13] G. Nannicini and L. Liberti. Shortest paths in dynamic graphs. *International Transactions in Operations Research*, 15:551–563, 2008.

[14] T. Schlick. *Molecular modelling and simulation: an interdisciplinary guide*. Springer, New York, 2002.

[15] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, Berlin, 2003.

[16] J.J. Sylvester. Chemistry and algebra. *Nature*, 17:284, 1877.

Index

F_2, 26
$[n]$, 24
$\mathbb{N}$, 19
ω, 19

A. De Morgan, 8
action, 24
address, 5
alphabet, 6, 17, 20
array, 2, 3
ASCII, 21
atom, 11
    bond, 11
attribute, 5

balanced, 5
benzene, 11
binary tree, 4, 5
bug, 9
byte, 21

cardinality, 20
Cartesian product, 21
character, 6
characters, 20
chemistry, 11
connective, 18
consecutive, 19, 21
contiguous, 3, 4
CPU time, 2, 3
cycle, 11, 13
    disjoint, 25

database, 5
debugger, 21
distance constraint, 12
distributivity, 26
domain, 19

efficiency, 5
equivalence class, 22
Eulerian walk, 13

field, 14, 26, 27
    finite, 26
    infinite, 26
four-colour theorem, 8, 13
fragmentation, 5
fragmented, 3, 4
function, 19
    bijection, 19
    composition, 20
    identity, 20
    injective, 19
    inverse, 20
    surjective, 19

GPS, 10
graph, 5, 11
    chemical, 11

Eulerian, 9
Königsberg, 9
region, 8
road, 8, 10
simple, 10
web, 13
weighted, 12
drawing, 12
group, 23
abelian, 23, 26
additive, 23, 26
associativity, 23
closure, 24
identity, 23
inverse, 23
multiplicative, 23

height, 5
hexadecimal, 3
HTML, 6
tag, 6
hyperlink, 6

IEEE, 6
index, 20
internet
graph, 7
IPv4, 7
iteration, 2

L. Euler, 9
language
formal, 6
natural, 6
leaf, 5, 13
linked list, 3, 5
list, 3
linked, 3, 5
logistics, 10
loop, 2

map
region, 8
road, 8
memory, 2–5
address, 5
modulo, 23
molecule, 11
multiplication
scalar, 26

NMR, 12
node, 4
leaf, 5
root, 4, 5
sub-, 4
number
natural, 19

operating system, 2

pair, 19
ordered, 5, 6, 19
partition, 22
path, 5
fastest, 10
shortest, 10
permutation
action, 24
cycle, 24
inverse, 24
product, 24
pointer, 20
power set, 18
predecessor, 19, 21
prime, 23, 26
procedure, 4
product
associative, 24
projection, 21
property, 5
protein, 12
backbone, 12
graph, 12
side chain, 12

quantifier, 18
query, 13

range, 19

record, 5
recursive, 4
relation, 5, 17, 22
  antisymmetric, 21
  equivalence, 21, 22
  irreflexive, 10, 21
  reflexive, 21
  symmetric, 7–9, 11, 12, 21
  transitive, 21, 22
  union, 21
root, 4, 5, 13
rotation, 23
router, 7

scalar, 26
  multiplication, 27
sentence
  valid, 18
sentences, 20
sequence, 6, 27
set
  cardinality, 20
  difference, 18
  empty, 18
  finite, 20
  intersection, 18
  partition, 22
  power, 18
  union, 18
  well-founded, 18
sort, 13
spreadsheet, 5
subgraph, 6, 13
subnode, 4
subtree, 4, 5
successor, 19, 21
symbol, 18
  shorthand, 18
  variable, 17

table, 5
theorem
  four-colour, 8, 13
time-sharing, 2
transitive closure, 22
transportation, 10
tree, 4, 6, 11
  balanced, 5, 13
  binary, 4, 5
  height, 5
  sub-, 4, 5

URL, 6

valence, 11
vector, 27
  addition, 27
vector space, 14, 26

W.R. Hamilton, 9
water, 11
web
  graph, 6
  page, 6
website, 6
word, 6, 17
words, 20

ZFC, 18