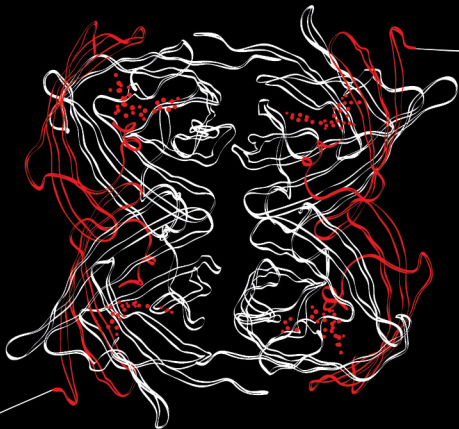
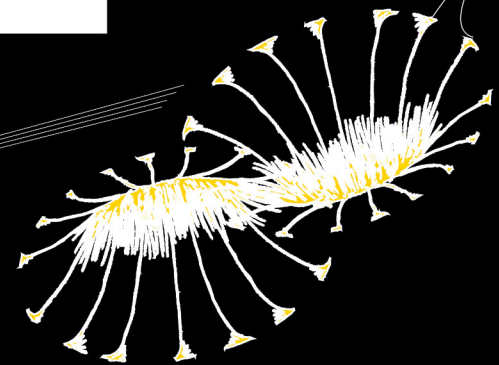




PROPAGATION OF ANNOTATIONS IN A STREAMING CONTEXT

Master Thesis T.E. Klifman



Propagation of Annotations in a Streaming Context

Author:

T.E. Klifman

Graduation Committee:

dr. A. Wombacher

dr.ir. M. van Keulen

J. Amiguët MSc

University of Twente

*Masters Thesis for the study Computer Science,
with a specialization in Information Systems Engineering*

November 30th, 2011

Corresponding e-mail address:

tim@timklifman.nl

Abstract

Sensor networks produce streams of data. These streams may be transformed by processing elements. The data in the streams may be organized in data structures. We define a framework for describing data structures. The processing elements may transform the data structures into new data structures. We define a framework for describing the transformation of a data structure. The data structures may be annotated. We provide a framework for propagating these annotations when the data structure is transformed. We introduce the concept of locality, which is a similarity function that quantifies the closeness of elements in a data structure. The locality concept is used in the annotation transformation model. Finally, 3 use cases from scientific application are used to evaluate the annotation transformation model.

Acknowledgements

This thesis is the result of a series of assignments. Before my graduation project I already did a few projects on this subject, all with my supervisor Andreas Wombacher, to whom I'm much obliged. Over the course of a few years we have investigated the subject of transformations in a streaming context. Although the idea is that you gain scientific insights throughout your studies, I realize I've mostly learned working in a scientific manner, once I started working on these projects with Mr. Wombacher. He taught me to look deeper at concepts, to observe not only what is going on at the surface, but what's really going on. When he asked me a question, such as 'how does this work', or 'why is this such and such', it was not a test, but a sincere attempt to make me get a better understanding of things. Most of the time it was not even a question to which either of us knew the answer, but rather an invitation to embark on a journey of exploration. I thank Andreas Wombacher for everything he taught me over the years.

I would like to thank my two other supervisors, Maurice van Keulen and Juan Amiguët, for their efforts to read my material and provide me with very detailed and useful feedback. I'm also thankful to Roel van der Hoorn who reviewed early drafts of my work and gave me tips about the style of writing.

Finally I would like to thank my family and friends, who have supported me over the years. First of all, my parents, to whom I'm much indebted. They supported me over the years, always encouraging me to take my own decisions. Even when I worked on extra-curricular activities, that caused delays in my studies, they did not complain. I'm also thankful to all my good friends, who over and over had to hear my stories about transformations, but never complained. Not even when I was speaking for too long...

*Tim Klifman
Enschede, 2011*

Table of contents

Abstract	iii
Acknowledgements	v
Table of contents	vii
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement and Research Questions	3
1.3 Approach	4
1.4 Structure of the Report	5
2 Related Work	7
2.1 Provenance in Static Databases	7
2.2 Provenance in Streaming Environments	8
2.3 Data Quality	12
2.4 Conclusion	13
3 Classification of Data Structures	15
3.1 Definition of a Data Structure	15
3.2 Elementary data structures	18
3.2.1 Set	19
3.2.2 Vector	19
3.2.3 Matrix	21
3.2.4 Array	22
3.2.5 Graph	23
3.2.6 Tree	25
3.3 Nested data structures	26
3.3.1 Table	27
3.3.2 Image	27
3.4 Properties of data structures	29
3.4.1 Ordering	29
3.4.2 Dimension	30
3.4.3 Relation based	30

3.4.4	Hierarchical relations	31
3.5	Properties of instances: Size	31
3.6	Overview	32
3.7	Conclusion	33
4	Classification of Transformations	35
4.1	Transformation model	35
4.2	Transformation Diagram	38
4.3	Elementary Operations	40
4.3.1	Set	40
4.3.2	Vector, Matrix and Array	41
4.3.3	Graph	42
4.3.4	Tree	44
4.4	Transformation Properties	45
4.4.1	Transformation Effect Dependency	45
4.4.2	Contributing Element Cardinality	48
4.4.3	Contributing Relation Member Cardinality	51
4.4.4	Reversibility	51
4.4.5	Invariance	54
4.4.6	Data/Structure contribution coherence	55
4.5	Structure by structure transformation	57
4.5.1	One-to-one	57
4.5.2	Many-to-one	60
4.5.3	One-to-many	61
4.5.4	Many-to-many	61
4.6	Multiple Input Sources	64
4.6.1	One-to-one	66
4.6.2	Many-to-one	66
4.6.3	One-to-many	67
4.6.4	Many-to-many	67
4.7	Summary	68
5	Locality	69
5.1	Distance	70
5.1.1	Euclidean distance	70
5.1.2	Manhattan distance	72
5.1.3	Chebyshev distance	73
5.1.4	Path distance	74
5.2	Locality	76
5.2.1	Locality in a set	77
5.2.2	Locality in a vector	77

5.2.3	Locality in a matrix	77
5.2.4	Locality in an array	78
5.2.5	Locality in a graph	78
5.2.6	Locality in a tree	79
5.3	Locality and transformations	80
5.3.1	Distance and Diameter Unchanged	80
5.3.2	Distance Unchanged, Diameter Changed	81
5.3.3	Distance Changed, Diameter Unchanged.	81
5.3.4	Distance Changed, Diameter Changed	82
5.4	Conclusion	83
6	Propagation of Annotations	85
6.1	Annotated data structures	85
6.1.1	Set	87
6.1.2	Vector	87
6.1.3	Matrix	87
6.1.4	Array	88
6.1.5	Graph	88
6.1.6	Tree	88
6.2	Annotation transformation model	89
6.3	Transformation of Annotations	91
6.3.1	One input element / weight 1	91
6.3.2	Multiple input elements / weight 1	93
6.3.3	Multiple input elements / weight $\in \{1,2\}$	94
6.3.4	Multiple input elements / weight depends on distance	96
6.3.5	Multiple input elements / weight depends on locality	98
6.4	Conclusion	99
7	Use Cases	101
7.1	Earth Remote Sensing	101
7.1.1	Description	101
7.1.2	Data structures	105
7.1.3	Transformations	106
7.1.4	Annotations	110
7.1.5	Conclusion	112
7.2	Temperature sensor network	113
7.2.1	Description	113
7.2.2	Data structures	117
7.2.3	Transformations	118
7.2.4	Annotations	122
7.2.5	Conclusion	125
7.3	Bio-informatics: Cell-Graph Analysis	126
7.3.1	Description	126

7.3.2	Data Structures	130
7.3.3	Transformations	131
7.3.4	Annotations	134
7.3.5	Conclusion	137
7.4	Conclusion	138
8	Conclusion	139
8.1	Reflection on research questions	139
8.2	Future research	142
References		145
Appendix A	Symbol Chart	149
Appendix B	Use case results	151
B.1	Transformation & Annotation Listing of Use Case 1	151
B.1.1	Transformation 1	151
B.1.2	Transformation 2	153
B.1.3	Transformation 3	156
B.1.4	Transformation 4	159
B.2	Transformation & Annotation Listing of Use Case 2	160
B.2.1	Transformation 1	160
B.2.2	Transformation 2	161
B.2.3	Transformation 3	164

List of Figures

Figure 1.1: A workflow consisting of three processing elements and two streams	2
Figure 2.2.1: Provenance Taxonomy, adapted from [SPG05]	9
FIGURE 3.1a: DS-diagram	18
Figure 3.2a: DS-Diagram	21
Figure 3.3: An undirected graph	24
Figure 3.4: A directed graph	24
Figure 3.5: A tree	25
Figure 4.1: A Transformation Diagram	38
Figure 4.2: A transformation diagram for Example 4.2	39
Figure 4.3: Diagram of a one-to-one transformation	49
Figure 4.4: Diagram of a one-to-many transformation	49
Figure 4.5: A diagram of a many-to-one transformation	50
Figure 4.6: A diagram of a many-to-many transformation	50
Figure 4.7: A data-only transformation	54
Figure 4.8: A structure-only transformation	55
Figure 4.9: A data/structure incoherent transformation	56
Figure 6.1: three levels of annotations in a graph	88
Figure 6.2: three levels of annotations in a tree	88
Figure 7.1: The orbit of Landsat 7	102
Figure 7.2: A satellite creating a swath by scanning the earth	102
Figure 7.3: Swath track	103
Figure 7.4: Swath Pattern	103
Figure 7.5: An earth remote sensing workflow	104
Figure 7.6: The transformation of the data produced by the satellite	104
Figure 7.7: Workflow of the temperature sensor network	113
Figure 7.8: A map of the area	114
Figure 7.9: The grid	114
Figure 7.10: Dividing lines	114
Figure 7.11: Points moved onto grid	114
Figure 7.12: The workflow of a cell-graph analysis setup	126
Figure 7.13: A cell tissue image	127
Figure 7.14: A cell tissue image in black-and-white	127
Figure 7.15: Application of a raster	128
Figure 7.16: Counting pixels	128
Figure 7.17: Annotation in a cell-tissue image	134

List of Tables

<i>Table 3.1: Overview of elementary data structures and their properties</i>	32
<i>Table 4.1: Dependency of the output data on parts of the input</i>	46
<i>Table 4.2: Dependency of the output structure on parts of the input</i>	46
<i>Table 4.3: Dependency of the output elements on part of the input</i>	47

Chapter 1

Introduction

This chapter gives an introduction to the thesis. We start with a motivation for our research. The problem statement and research questions are presented. Finally we discuss our approach and the structure of this report.

1.1 Motivation

Contemporary sciences, such as environmental research, may involve the use of sensor networks. An example of such research is a sensor web deployed in the Antarctic [DCJ+03]. In this sensor web multiple sensors are deployed to measure temperature and humidity to get a better insight in the atmospherical conditions of Antarctica.

Sensors usually produce data at certain time intervals. For example, each second a temperature value may be generated by the sensor. Instead of storing the data generated by the sensors, they may be processed by other elements in the network. These elements are called processing elements. In a temperature sensor network, processing elements may gather temperature values each second and output the average temperature per day.

The data arrives at the processing elements in streams. This means that the data is generated, transported and processed continuously. We say that a data stream consists of data elements and these elements are ordered by the time of arrival at the processing element. Different processing elements may form a chain where the processing elements are connected by streams. A processing element takes a stream as its input, performs an operation on the data elements in the stream, and outputs a new stream. A chain of processing elements and streams may be described by a workflow chart. Figure 1.1 shows an example workflow.

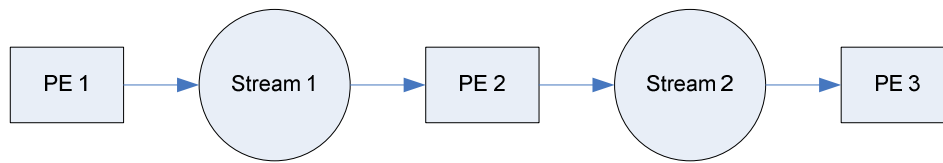


Figure 1.1: A workflow consisting of three processing elements and two streams

In this workflow there are three processing elements and two streams. The three processing elements are all of a different type. We distinguish processing element types according to their input/output behavior [AWK10]:

- PE1 is of the *source* type. Sources form the begin points of a workflow. They produce data streams but consume no existing data streams.
- PE3 is of the *sink* type. Sinks form the endpoints of a workflow. Sinks consume data streams but do not produce data streams of their own.
- PE2 is a *transformation* element. Processing elements that are in the middle of a workflow are called transformations. They produce output streams in a predetermined way based on input data streams.

Transformation elements may perform all kinds of operations: operations that change the contents of the elements in the stream, or operations that leave the contents unchanged, but change the stream itself, for example by making a selection on the elements. It is also possible that multiple streams are consumed

Sometimes it is necessary to indicate that something is wrong with the data. This can be done using annotations: parts of the data are given a specific mark. For example, a temperature sensor may be snowed in causing it to observe the temperature of the snow, instead of the air. These measurements may not need to look wrong: a 26 degree Celsius reading, instead of 24, may be plausible, though in subsequent calculations it may cause serious errors. In this case it would be nice if could annotate the data by saying for a particular element in the stream that the sensor was snowed in when the value was generated.

But what needs to be done with the annotation when the data element it belongs to arrives at a transformation element? We could build the transformation elements in such a way that it discards all data that is marked as snowed in. We could also ignore the annotation and process the data element as if it wasn't annotated. A third possibility is to have the transformation element transform the annotations as well. But to transform the annotations we need to know what kind of annotations we can expect in order to know how to handle them.

It may be very well the case that the person responsible for the development of a transformation element may have no control over the development of other processing elements. In fact a whole workflow may be constructed of processing elements produced by different developers, where the person that constructs the workflow may

have no control over the development of any of the processing elements. When data streams contain annotations that processing elements do not expect they cannot be transformed. Some of these processing elements may not be able to process annotations at all.

In this thesis we present a method where annotations can be transformed in a parallel system. We propose definitions that can be used to describe the concept of a transformation which then can be used to transform annotations accordingly. We focus only on transformations where one element from an input stream is used to produce one element in the output stream. These elements may consist themselves of other elements. We call those elements data structures. Part of these data structures may be annotated instead of the whole structure. We are concerned here with how one data structure is transformed into another and how the transformation of the corresponding annotations can be done.

1.2 Problem Statement and Research Questions

This thesis investigates answers to the following problem:

PROBLEM STATEMENT How can annotations in a stream processing system be propagated?

To contribute an answer to this problem, we need a way of defining the transformation of annotations. To describe the transformation of annotations we need to be able to describe how the transformation of a data structure looks like. To describe the transformation of a data structures, we need to be able to describe how a data structure looks like. This leads us to the following research questions:

RESEARCH QUESTION 1 How can the description of data structures be formalized?

This research question will be answered by investigating how a data structure can be formally defined; what elementary data structures are; how nested data structures can be defined and what the properties are of elementary data structures.

RESEARCH QUESTION 2 What type of transformations of data structures exist and what are their properties?

This research question will be answered by investigating how the transformation of a data structure can be defined; how the algorithm of a transformation can be described; what the properties of a transformation are and how its effect can be described.

With the first group of research questions we investigate what the relation between elements in a data structure is. With the second group of research questions we investigate the relation between elements in the input and elements in the output of a

transformation. With the third group of research questions we investigate what the effect from transformations investigated in the second group is on the relations investigated in the first group.

RESEARCH QUESTION 3 How can locality be used to describe the effect of a transformation on the relation between elements in a data structure?

This research question will be answered by investigating how locality can be defined; how locality can be applied to different data structures; what the possible effects are of a transformation on locality.

The answers to the first three groups of research questions will be used to investigate how the fourth group of research questions can be answered:

RESEARCH QUESTION 4 How can annotations in a data structure be propagated?

This research question will be answered by investigating how data in the elementary data structures can be annotated; by defining an annotation transformation model and applying this model to different scenarios.

1.3 Approach

In this part of the chapter we discuss the approach we have taken for our research. We have started with collecting examples of real life stream processing setups. Three of these examples are used as use cases in this work. We used these examples to find out what data structures are used to store the data products in order to construct a list of elementary data structures. We have developed a framework of formal definitions that can be used to describe data structures.

We used the elementary data structures to gain insight in possible transformations of these data structures. We have investigated how these transformations can be described by an algorithm of elementary operations. To describe such an algorithm one needs still quite a lot of information. We have proceeded to find methods that are less cumbersome. This lead us to analyze properties of the transformations that we could use to characterize the transformations and make a distinction between very complex and lesser complex transformations. For a specific class of transformations we were able to describe the effect by transforming the position of the input elements, which made it unnecessary to describe the full algorithm.

We used the description of the effect of the transformations to develop a model that can be used to decide whether annotations in a data structure should be propagated or not. This model uses a weighting method, based on the elements that contribute to the

output. We've found that we could find these contributing elements using the position transformation function.

In some of these scenario's it was possible to propagate annotations only on the basis of the relation between output elements and contributing input elements. In some other scenario's more information was needed. We found that the change in the relation between elements in a data structure before and after a transformation is in some scenario's the extra information we need to propagate annotations. This lead us to investigate and give a formal definition of the concept of locality. Locality is based on distance between elements, so we investigated different methods of measuring distance in a data structure.

Lastly we wanted to verify whether our model of data structures, transformations, the concept of locality and the methods of propagating annotations would really work when applied to real world, non-trivial cases. For this part we have selected examples from the work flows that we analyzed in the very beginning and worked them out into proper use cases.

1.4 Structure of the Report

In this chapter we introduce the results of our work and in this part we will describe an outline of the rest of the thesis. The structure of this report is mainly related to the research questions, as each group of research questions is addressed in the same chapter.

- In CHAPTER 2 we present an overview of related work. We describe related work in the fields of provenance, data quality and stream processing.
- CHAPTER 3 is the first chapter that addresses research questions. These are RESEARCH QUESTION 1 and its sub questions. We describe what a data structure is and what the elementary data structures are. We introduce formal definitions that can be used to describe specific instances of elementary data structures and which can be extended to describe more specialized data structures. Finally, a classification model is presented that distinguishes between the different data structures. This classification is based on properties that make one type of elementary data structure different from another.
- We describe how data structures can be transformed in CHAPTER 4. We start with a transformation model that we use to investigate five properties of transformations. We then show how any transformation can be constructed using elementary operations. Finally we show how in some scenarios the relation between output elements and their contributing input elements can be

described using a formula based on the position of the elements. This chapter addresses RESEARCH QUESTION 2 and its sub questions.

- In CHAPTER 5 we address RESEARCH QUESTION 3 and its sub questions. We investigate how the relation between elements in a data structure changes over a transformation. Especially we investigate how a property called locality changes. We introduce a definition of locality, which is based on the distance between elements. We show how different distance measurements can be applied to different data structures. Finally we present four scenarios in which the locality between elements may be affected by a transformation.
- The final chapter that addresses research questions is CHAPTER 6 and addresses RESEARCH QUESTION 4 and its sub questions. First, we show for each elementary data structure how annotations at different levels can be made. Second, we present a model for propagating annotations and finally we show five scenarios where we apply the model with different parameters.
- The chapters that address the research questions provide many definitions that can be used to describe data structures and transformations and which can be used to propagate annotations in a data stream system. In each corresponding chapter we have given trivial examples to show how a definition can be applied. In CHAPTER 7 we test our definitions on three non-trivial uses cases as a verification of our work.
- Finally we will present our conclusions, discuss them and describe what future work could be done in CHAPTER 8.

Chapter 2

Related Work

In this chapter we discuss the literature that is related to our work. Most of the papers deal with provenance. Provenance describes the origin of data and includes descriptions of the transformations in a work flow. In this thesis we focus on the propagation of annotations. Because we use descriptions of transformations for the propagation of annotations, we investigate how data provenance approaches deal with this topic. In subchapter 2.1 we review provenance in static databases and in subchapter 2.2 we review provenance approaches in streaming environments. Annotations can be used to assess the quality of data. In subchapter 2.3 we review papers about data quality in streams. Finally we present our conclusions about in what extent the related work fits our own research.

2.1 Provenance in Static Databases

Buneman et al. were one of the first to use the notion of provenance in an information processing context. In their paper they introduce two notions of data provenance: *where*- and *why*-provenance [BKT01].

When performing a query, a set of input tuples is used to generate a set of output tuples. In order to reproduce the output set, one needs the query and the input set. The set of input tuples is referred to as the *why*-provenance. When creating a view on a database, a new set of tuples is generated. When one wants a change in the view to also update the original data, one needs to know where this element in the original data is located. This is what Buneman et al. refer to as *where*-provenance.

The term data lineage shares a great deal of overlap with that of data provenance. The TRIO project [Wid08] is developing a DBMS that focuses on two topics: *uncertainty* and *lineage* of data. The lineage is essentially what Buneman et al. call *why*-provenance. The

TRIO-DBMS is able to generate the set of input tuples that contributed to the output set of a given query. This is also called query-inversion. However, it needs the original input data for this, which makes the method inapplicable for streaming contexts, where data might not be persistent.

2.2 Provenance in Streaming Environments

Both Buneman et al. and the Trio-project focus on the origins of data in static databases. Later the focus shifted towards scientific workflows, some of them using web-services as components of the workflow.

[SM03] propose a method for recording and reasoning over data provenance in web and grid services. They focus on a workflow enactment using web services. In these setups, the provenance information describes which services have been used, what kind of data has been passed between those services and what results have been generated by the services. They created a service-oriented architecture, where they use a specific web service for the recording and querying data provenance.

Any time a web service in the workflow is invoked it also contacts the provenance service, and submits a provenance record. This way it builds a trace of the services that are invoked. Using the provenance records, a user can verify whether a certain web service will produce the same output given a specific input. It cannot perform a complete workflow re-enactment however, as the provenance service does not have an understanding of the workflow script.

Zhao et al. explore a more abstract notion of provenance information for workflows [ZWF06]. In their view provenance incorporates two parts: “all the aspects of the procedure or workflow used to create a data object (prospective provenance, or ‘recipe’) as well as information about the runtime environment in which a procedure was executed and the resources used in its invocation (retrospective provenance).”

The prospective provenance is all that is needed to produce or reproduce a data object and can be used to track its derivation. A workflow consists of a set of calls to procedures. A workflow together with the input data can be used to (re)produce output data. A call specifies the values of the arguments that are passed to a procedure.

Zhao et al. distinguish between a call and an invocation, in that an invocation is a call at a particular moment in time. Any invocation of a call to a procedure on a given data set would result in the same output set. Therefore the provenance on the invocation is retrospective. It isn’t needed to be able to reproduce the output set, but it does describe the circumstances under which the output set was created.

During the years different architectures and systems have been proposed for recording and reasoning over data provenance. Simmhan et al. give an overview of data

provenance techniques that were used in e-science projects around 2005 [SPG05]. They define a taxonomy, which can be used for surveying different provenance models (Figure 2.2.1).

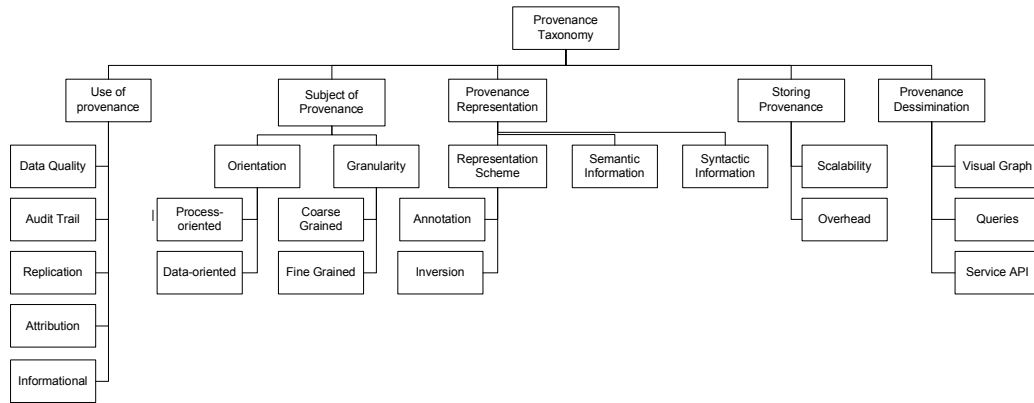


Figure 2.2.1: Provenance Taxonomy, adapted from [SPG05]

Ledlie et al. discuss the problem of storing provenance-aware sensor data [LN+05]. Within the taxonomy model of Figure 2.2.1, they focus on provenance at the data-level. They identify two levels of indexing the data in a streaming context: for each tuple or for sets of tuples. This can be fit to the two granularity levels, coarse and fine grained, of the taxonomy model. Where the taxonomy model defines two aspects of provenance storage, Ledlie et al. define six criteria to evaluate provenance index and query architectures. Two of them are similar to two concepts defined in the taxonomy of [SPG05]. Both define the concept of *scalability*. Ledlie et al. define *resource consumption* which is similar to *overhead* in the taxonomy. In addition four more criteria are provided: *reliability*, *query result quality*, *usability* and *speed*.

The paper of Ledlie et al. concludes by proposing different models for storage. They distinguish between centralized and decentralized systems. Because query processing is becoming increasingly more distributed, a centralized option is not always possible. However, decentralized solutions may give problems when the hosts aren't stable [LN+05].

Within scientific workflows Simmhan et al. identify a specific class of workflows called data-driven workflows [SPG08]. In data driven workflows, data products are first-class parameters to services that consume and transform the input to generate derived products. They propose a framework called Karma2 for recording not only data provenance on the processes but also on the data products themselves. In their approach they use an abstract notion of a workflow, as to be independent from models used.

Blount et al. argue that the annotation model, where each data item is annotated with corresponding data provenance, is unsuitable for streaming environments, because “it fails to exploit the dependency characteristics of data items in streams and causes unnecessary overhead” [BD+07]. They discuss the distinctive features of the provenance problem for stream-oriented middleware systems and introduce their own TVC-model. The proposed TVC model addresses three characteristics:

- The high data rates of streams results in high data volumes if collected. Annotating each data element will increase this burden, and is not efficient from a storage perspective.
- Processing the streams for provenance must be computationally efficient as not to slow down the system’s throughput.
- Most transformations by processing elements output data based on input values accumulated in a fixed time window. They are therefore to be considered stateful as the output depends on a fixed set of past input elements.

Lim et al. use streaming data based on provenance to assess the trust scores of stream sources [LMB09]. Two types of provenance are introduced: physical and logical provenance. Physical provenance is used to show where the item was produced and how it was delivered to the server. This kind of provenance is used to compute trust scores. Logical provenance represents the semantic meaning in the context of a given application.

They address different challenges for using and delivering data provenance. The relevant challenges are summarized in the following the questions:

- How can the data provenance of more complex applications be represented?
- How can provenance data from different sources be combined?
- How can variable and fixed data provenance be combined?
- How can data provenance be used to improve the data quality?
- How can the size of provenance information be minimized, to improve efficient delivery?
- How can it be ensured that provenance information is safe from malicious attacks?

For a few of these questions they give hints for solutions, but essentially they’re left as open questions.

The Tupelo semantic context management system is a concrete attempt to build a server that can be used to register provenance information using semantic web technologies [FG+09]. The server has an API that can be used by processing elements in a workflow to register provenance information. Using ontologies the semantics of provenance information can be described. It supports different protocols so the Tupelo system can be embedded in existing software packages.

With different research groups investigating different aspects of data provenance, the need for standardization has risen. This has been discussed during sessions at conferences on data provenance and ultimately led to the Open Provenance Model [MF+08]. To discuss the potential model, a series of workshops have been organized. For each of these workshops a Provenance Challenge was defined. At the time of writing there have been four Provenance Challenges [Pcw11]. Each challenge consisted of a case that could be used by participants to prepare for the workshop, where each other's results would be discussed and compared.

The first provenance challenge was a case that was used to investigate and discuss the capabilities of different provenance systems and the expressiveness of their provenance representations. It focused on the following three details:

- The representations that systems use to document details of processes that have occurred.
- The capabilities of each system in answering provenance-related queries.
- What each system considers to be within scope of the topic of provenance (regardless of whether the system can yet achieve all problems in that scope).

The second challenge was a case that has been used to investigate and discuss the way how provenance information could be exchanged by different systems. The same case as in the first challenge was used. This time participants were to query over data provenance generated by other teams, as if it were generated by their own system. In particular, two questions were tried to be answered:

- Understand where data in one model is translatable to or has no parallel in another model.
- Understand how the provenance of data can be traced across multiple systems, so adding value to all those systems.

These first two provenance challenges led to the specification of the Open Provenance Model. Version 1.0 was proposed and after a workshop where it was discussed, a revised version 1.01 was published. The third provenance challenge was mainly to identify weaknesses and strengths of the specification, to determine how well it can represent a variety of technologies (not just scientific workflows, but also databases, etc). Many proposals were made that ultimately led to version 1.1 of the open provenance model [Opm09]. Finally the activities of the planned fourth challenge were merged with activities of the World Wide Web Consortium Provenance Incubator Group[Pig11] culminating in the W3C Provenance Working Group [Pwg11]

2.3 Data Quality

When it comes to the streaming environment, most research focuses on one particular application of data provenance: assessing the quality of the data. Bizdikian et al. propose a framework for assessing the Quality of Information in sensor networks. They define Quality of Information as:

The collective effect of information characteristics (or attributes) that determine the degree by which the information is (or perceived to be) fit-to-use for a purpose.
[BD+09]

The latter part ‘fit-to-use for a purpose’ is different from the concept of data provenance. In addition to data provenance this definition of Quality of Information takes into account what an end-user has meant to do with the information. Put differently, Quality of Information represents characteristics of both the producer as well as the consumer. A model with six dimensions, called the 5WH framework is used as metric to describe the quality.

These dimensions are why, when, where, what, who and how. The why-dimension is specific for the application domain, as it represents the purpose. When and where represent temporal and spatial dimensions, and are specific to both the consumer and the producer. Finally, who and how are specific for the provider domain, as they represent the sensors and other providers of data. The different dimensions can be used to perform match-making between sensors and applications.

Klein et al. [KL09] describe in their paper how data quality of streams can be managed. They use five dimensions to define the quality of the data: accuracy, confidence, completeness, data volume and timeliness. Besides giving a definition of quality they also give a formal description of the impact of certain operators on the quality of data, such as joins, selections and aggregations. These concepts of data quality can be used when generating provenance information.

2.4 Conclusion

Most of the literature that we have reviewed focuses on data provenance. Provenance is mainly concerned with the origins of the data. To know where the data came from concepts are described that can be used to model the workflow that produced the data. Some of these research papers presented models to describe how a workflow works on a global level. These approaches do not provide the option of specifying a transformation in detail. For example it is possible to describe that a processing element has two input streams, joins them and produces one output stream. In these models it is not possible however, to describe exactly how the data from these streams are joined.

Other research papers presented a more detailed approach, where a web service could be invoked by processing elements. We are looking for a detailed approach where transformations can be described in detail without having access to the development of the processing elements. This rules out the option of invoking a web service by the processing element.

Provenance is sometimes used to assess the quality of the data in the streams. In this thesis we focus on annotations which also can be used to measure quality of the data. Although data quality is a very generic concept, some of the papers reviewed in this chapters had very specific applications. They are more specific than the approach of our research allows. Where they mainly deal with what to do with the annotations we focus on how to propagate the annotations, whatever annotations they may be.

Classification of Data Structures

In this chapter we present a formal model that can be used to describe data structures. First we give a definition of a generic data structure. We present a list of elementary data structures and show how they can be constructed using the definition of a generic data structure. These data structures all have properties that make them different from other data structures. We address these properties and show how they are specific to the corresponding data structures. We use these properties in a classification model which we present at the end of the chapter. The definitions from this chapter are used in the next chapter to model the transformation of data structures.

The research question that we answer in this chapter is:

RESEARCH QUESTION 1

How can the description of data structures be formalized?

3.1 Definition of a Data Structure

A data structure is an organization of information. The information is divided in data items, and the method of organizing these data items makes up the structure. We call these data items *elements*. A group of elements is called a *collection*. A data structure then consists of a collection of elements and the relation between those elements. First we introduce a formalization to represent a generic data structure. In Chapter 3.2 we show how the formal definition of a generic data structure can be used to represent six different types of elementary data structures.

DEFINITION 3.1 A unique element in a data structure is denoted e_i . Here e denotes an element of a generic data structure and i is a unique natural number. When referred to an element of a specific type of data structure, a different alphabetic character than e may be used.

Within each data structure these elements are grouped into a collection. We call the underlying collection of a data structure its base set. The base set has the same form for each type of data structure.

DEFINITION 3.2 The underlying collection of a data structure is called the base set and is represented by S^{base} .

The base set forms the collection of the elements in the data structure. A data structure may include a relation to organize the elements. Such a relation may be specific to each type of data structure. The relation must have the base set as its domain, co-domain or both. When the relation has the base set as both its domain and co-domain it is called a binary relation on S^{base} .

In Chapter 3.2 we will consider six elementary data structures in detail, but we present here already an example of a specific instance to illustrate the application of the definition of a generic data structure. One of these elementary data structures is the graph.

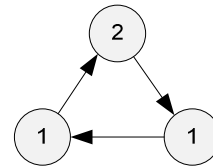
EXAMPLE 3.1 shows such a relation. In the other scenario elements from the base set are related to elements from another set or vice versa. Such a relation occurs in a vector, as we will see in Chapter 3.2.2.

DEFINITION 3.3 A data structure may include a relation. The set of relations among the elements make up the structure. This set of relations is represented by R .

In Chapter 3.2 we will consider six elementary data structures in detail, but we present here already an example of a specific instance to illustrate the application of the definition of a generic data structure. One of these elementary data structures is the graph.

EXAMPLE 3.1

We consider the following graph:



This graph consists of three nodes and three edges. The nodes form the elements, and we give each node a unique number. So here $S^{base} = \{e_1, e_2, e_3\}$. The relation of this data structure is formed by the edges between the nodes: $R = \{(e_1, e_2), (e_2, e_3), (e_3, e_1)\}$.

Any element can have different properties. The most common property of an element is its value, and these values are the actual data. Multiple elements may have the same value. In those cases it is not possible to distinguish between two elements only on the basis of their values. For this reason we have introduced Definition 3.1 to identify a unique element. We define a function that maps each element to a value. The elements are members of the base set and the values are members of the value set. The value function, which is denoted by the λ symbol, maps members of the base set to members of the value set.

DEFINITION 3.4 The *value function* maps each element to a value and is denoted $\lambda: S^{base} \rightarrow S^{value}$, where S^{value} is the value set.




EXAMPLE 3.2 We use the graph from EXAMPLE 3.2. The value set for this graph is given by $S^{value} = \{1, 2\}$ and the value function is given by $\lambda = \{(e_1, 1), (e_2, 2), (e_3, 1)\}$.

We combine these definitions to give a formal definition of a complete data structure.

DEFINITION 3.5 A generic data structure is represented by $\mathbb{S} = (S^{base}, R, \lambda)$ where the tuple (S^{base}, R) forms the structure part and the function λ the data part. To denote a specific type of data structure a non-italic upper case alphabetic character is used instead of \mathbb{S} .

The definitions from this part of the chapter can be used to describe a complete instance of a data structure. Instead of using mathematical symbols it is also possible to describe this visually using what we call a data structure diagram, or DS-diagram.

DEFINITION 3.6 A data structure diagram, or DS-diagram, is a visual description of a data structure. It consists of elements, values and arrows that relate elements to each other and elements to values.

- An element is represented by a circle: 
- A value is represented by a square: 
- An element that is related to another element or to a value is represented by an arrow: 

EXAMPLE 3.3 The graph from EXAMPLE 3.1 can be visually described using a data structure diagram. Figure 3.1a shows the DS-diagram for this graph. Figure 3.1b shows how the base set consists of elements and the value set of values. The relation that relates elements to elements and the value function that relates elements to values are depicted in Figure 3.1c.

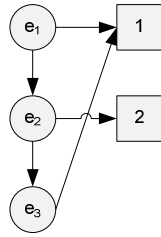


FIGURE 3.1a:
DS-diagram

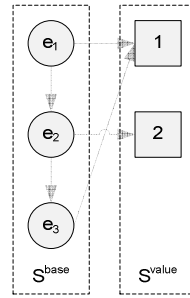


FIGURE 3.1b: base set
and value set

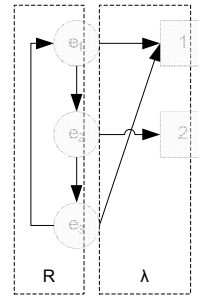


FIGURE 3.1c: relation
and value function

3.2 Elementary data structures

In this part of the chapter we present a list of six elementary data structures. Although there are many different types of data structures, we need a limited list of data structures that are very different from each other. The list is based on examples of how data is organized in programming languages and scientific applications. Java is an object-oriented programming language, and provides many classes to organize data. Although many of these classes have very special characteristics, most are based on arrays, lists and trees [Jav11]. Many scientific workflows involve the use of other data structures than available in programming languages. Other data structures that are used are sets, vectors, matrices, tables, images and graphs [Sci11, DCJ+03, BDN+07].

Some of these data structures are quite similar, and in some cases they are even the same, just bearing different names. For example a vector and a list are essentially the same. In other cases a data structure can be formed using a nesting of other data structures. Finally we have chosen a list of six elementary data structures. We aim to cover a wide range of very different data structures. The six elementary data structures are: set, vector, matrix, array, graph and tree.

Although images and tables are often used, we do not consider them elementary, as we will show in Chapter 3.3 that they can be represented using nestings of the data structures that are on the list of elementary data structures.

We use DEFINITION 3.1 - DEFINITION 3.6 to construct new definitions for each type of elementary data structure. The purpose of these definitions is to create building blocks out of which other data structure can be constructed. This can be done by nesting two or more elementary data structures or by extending the definition of an elementary data structure.

3.2.1 Set

Giving an exact definition of a set is rather difficult, but in a more intuitive way we say that a set is a well-defined collection of objects where the objects are called elements and are said to be members of the set [Gri98]. Well-defined in this case means that for all elements with certainty it can be said whether or not the element is a member of the set. Sets are basically collections without a structure, this means that the only operation on an element with regard to a set is to test whether it is a member of that set or not.

An example of a set using mathematical notation is $\{1,2,3\}$. In a set an element must be unique and can therefore occur only once, meaning that $\{1,2,3,3\} = \{1,2,3\}$. In these cases there is no difference between an element and the value. The elements are natural numbers and these natural numbers are both used as a means of identifying an element as well as given value to that element, i.e. an element *is* its value. But with DEFINITION 3.4 we have separated the identification of an element and its value with a value function. This also applies to sets when considered as a data structure.

The general definition of a data structure includes a relation. Since a set has no real structure we define the relation as an empty set. Elements of a set have no specific characteristics, so we use DEFINITION 3.1 to denote elements of a set.

DEFINITION 3.7 A set data structure S is defined as

$$S = (S^{base}, R, \lambda) = (\{e_1, \dots, e_n\}, \emptyset, \lambda) \text{ where } n = |S^{base}|.$$

3.2.2 Vector

Lay defines a vector as a list of numbers [LAY03]. The notation used is

$\mathbf{u} = [u_1 \ u_2 \ \dots \ u_n]$ where n is the size of the vector. Here u_i is the element u at position i and $1 \leq i \leq n$. An example of a vector using this notation is $\mathbf{v} = [9 \ 4 \ 9]$ where $v_1 = 9$; $v_2 = 4$; $v_3 = 9$. Another way of modeling a vector is by defining a position set function that maps a value to a set of positions [JGT01]: $F: X \rightarrow P(\mathbb{N})$. Here X is the set of values and $P(\mathbb{N})$ is the powerset of the natural numbers. The position set function maps one value to a set of positions, because the same value can occur at multiple positions. For the vector \mathbf{v} the position function has the values $F(9) = \{1,3\}$ and $F(4) = \{2\}$.

Both methods have their limitations. The first method defines the position of an element in an implicit way: the position is determined by the order of the elements in the vector and denoted using a subscript. The second method provides a more explicit way to denote the position, but there's no possibility to distinguish between elements having the same value. We need a definition for a vector data structure that is consistent with the definition of a generic data structure. We need a way of distinguishing between elements with the same value and an explicit way of denoting the position of an element.

We define an element of a vector data structure using DEFINITION 3.1:

DEFINITION 3.8 v_i represents a unique element of a vector data structure of size m where $1 \leq i \leq m$

The subscript i only refers to unique elements and not the position. We need a definition that specifies the position in an explicit way. DEFINITION 3.3 specifies that elements in a data structure may be organized by a relation. We define a function to represent the relation based on the idea of the position set function. We call this the element function of a vector, or vector function, and it can be used to retrieve an element at a given position. The element function relates an index set, which we denote S^{index} in general, to the base set.

DEFINITION 3.9-I The element function of a vector, or vector function, is a function that relates an index set to the base set. The vector function is denoted F_V and is defined as the bijective function $F_V: \mathbb{N}_m \rightarrow S^{base}$ where \mathbb{N}_m is the index set and $\mathbb{N}_m = \{1, 2, \dots, m\}$.

The function F_V can be used to find out which element is located at a certain position in the vector. In some cases we might want to know the reverse: at what position is a certain element located in the vector? Because F_V is bijective, we can create its inverse function, which we call the inverse vector function or the position function of a vector:

DEFINITION 3.9-II The vector function F_V can be inverted and is defined as $F_V^{-1}: S^{base} \rightarrow \mathbb{N}_m$ where $\mathbb{N}_m = \{1, 2, \dots, m\}$.

We use DEFINITION 3.8 and DEFINITION 3.9 to give a definition of a vector. The definition of the value function is the same as for the generic data structure.

DEFINITION 3.10 A vector V of size n is defined as $V = (S^{base}, R, \lambda) = (\{v_1, \dots, v_n\}, F_V, \lambda)$

In the beginning of this chapter we have used to model a data structure as a DS-Diagram. The relation in a vector relates index numbers to elements. We extend DEFINITION 3.6 here to include a symbol for an index.

DEFINITION 3.11 An index in a DS-diagram is represented by a hexagon:



EXAMPLE 3.4

We consider the vector $[1 \ 2 \ 1]$. Here $S^{base} = \{v_1, v_2, v_3\}$, $R = \{(1, v_1), (2, v_2), (3, v_3)\}$ and $\lambda = \{(v_1, 1), (v_2, 2), (v_3, 1)\}$. We use DEFINITION 3.6 and DEFINITION 3.11 to create the DS-diagram for the. Figure 3.2a displays the DS-diagram. Figure 3.2b displays the index set, base set and value set. Figure 3.2c displays the relation and value function.

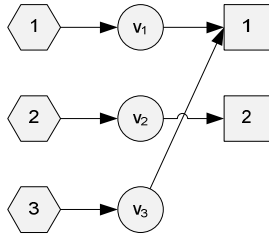


Figure 3.2a: DS-Diagram

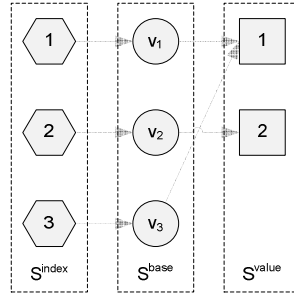


Figure 3.2b: index set, base set and value set

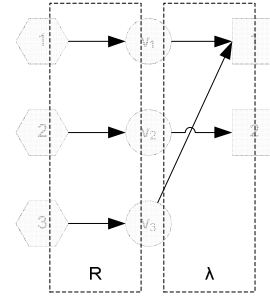


Figure 3.2c: relation and value function

3.2.3 Matrix

In linear algebra, an $m \times n$ matrix is a rectangular array of mn numbers arranged in m rows and n columns [Gri98, p.A-11]. Such a matrix is denoted by $A = (a_{ij})_{m \times n}$ where $1 \leq i \leq m$ and $1 \leq j \leq n$. The number a_{ij} is called the (i, j) -entry. The matrix A can be visually represented as follows [Lay03, p.107]:

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,j} & \cdots & a_{1,n} \\ \vdots & & \vdots & & \vdots \\ a_{i,1} & & a_{i,j} & & a_{i,n} \\ \vdots & & \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,j} & \cdots & a_{m,n} \end{bmatrix}$$

For example:

$$B = (b_{i,j})_{3 \times 2} = \begin{bmatrix} 1 & 2 \\ 1 & 6 \\ 3 & 1 \end{bmatrix}$$

Here B is a 3×2 matrix where $a_{1,2} = 2$ and $a_{2,2} = 6$. When we consider a matrix as a data structure, these definitions are too limited. The mathematical definition limits entries to numbers and the position is explicated using a subscript. When considering a matrix as a data structure, we want to separate the value from the element and model the position using a function. Therefore we redefine an element of a matrix to be no more than a unique element.

DEFINITION 3.12 m_i represents a unique element of an $m \times n$ matrix where $1 \leq i \leq m \cdot n$

Analogously to the vector we can define a matrix function that relates a tuple of two natural numbers to an element. This function can then be used to find out which element is located at position (i, j) .

DEFINITION 3.13-I The matrix function is denoted F_M and is defined as the bijective function $F_M: \mathbb{N}_m \times \mathbb{N}_n \rightarrow S^{base}$ where $\mathbb{N}_m = \{1, 2, \dots, m\}$ and $\mathbb{N}_n = \{1, 2, \dots, n\}$.

The matrix function is bijective and it be inverted to retrieve the position at which a given element is located.

DEFINITION 3.13-II The matrix function F_M can be inverted and is defined as

$$F_M^{-1}: S^{base} \rightarrow \mathbb{N}_m \times \mathbb{N}_n \text{ where } \mathbb{N}_m = \{1, 2, \dots, m\} \text{ and } \mathbb{N}_n = \{1, 2, \dots, n\}.$$

We can now give a definition of a matrix that complies with the definition of the abstract data structure.

DEFINITION 3.14 A matrix M of size $m \times n$ is defined as

$$M = (S^{base}, R, \lambda) = (\{e_1, \dots, e_{m \cdot n}\}, F_M, \lambda).$$

EXAMPLE 3.5

We consider the matrix $\begin{Bmatrix} 4 & 3 \\ 1 & 2 \end{Bmatrix}$ which is represented by $M = (\{m_1, m_2, m_3, m_4\}, \{(1, 1, m_1), (1, 2, m_2), (2, 1, m_3), (2, 2, m_4)\}, \{(m_1, 4), (m_2, 3), (m_3, 1), (m_4, 2)\})$

3.2.4 Array

An array is a higher dimensional generalization of the vector and matrix concepts. A vector is a one-dimensional array and a matrix is a two-dimensional array. Although vectors and matrixes are arrays as well, in this case we emphasize with the term “array” the higher dimensionality. Specific arrays have a certain dimension and each dimension has a boundary. The size of an array is the product of the boundaries of its dimensions.

DEFINITION 3.15 The size of an array is the product of the boundaries of its dimensions. The number of dimensions of an array is $d \in \mathbb{N}$ where $d > 2$. The size of an array is then given by

$$s = \prod_{i=1}^d m_i$$

where m_i is the boundary of the i^{th} dimension.

Using the definition of the dimension and the size of an array we can specify how unique elements of an array can be defined.

DEFINITION 3.16 a_i represents a unique element of an array of size s where $1 \leq i \leq s$.

We can generalize the vector and matrix function and its inverse and use that to give a definition of an array.

DEFINITION 3.17-I The array function F_A is defined as

$$F_A: \left(\prod_{i=1}^d \mathbb{N}_{m_i} \right) \rightarrow S^{base}$$

where $\mathbb{N}_{m_i} = \{1, 2, \dots, m_i\}$ and m_i is the boundary of the i^{th} dimension. **DEFINITION 3.17-II**

The array function F_A can be inverted and is defined as

$$F_A^{-1}: S^{base} \rightarrow \left(\prod_{i=1}^d \mathbb{N}_{m_i} \right)$$

where $\mathbb{N}_{m_i} = \{1, 2, \dots, m_i\}$ and m_i is the boundary of the i^{th} dimension.

DEFINITION 3.18 An array of size s is defined by

$$A = (S^{base}, R) = (\{a_1, \dots, a_s\}, F_A, \lambda)$$

EXAMPLE 3.6

We consider an $3 \times 4 \times 2 \times 1$ array. This array has 4 dimensions and the boundary of the 3rd dimension is 2. The size is $s = 3 \cdot 4 \cdot 2 \cdot 1 = 24$. The array function is given by: $\mathbb{N}_3 \times \mathbb{N}_4 \times \mathbb{N}_2 \times \mathbb{N}_1 \rightarrow S^{base}$ which is equal to $\{1, 2, 3\} \times \{1, 2, 3, 4\} \times \{1, 2\} \times \{1\} \rightarrow S^{base}$.

3.2.5 Graph

In the beginning of this chapter we have used the graph as an example to illustrate the use of the definitions of a generic data structure. There are different types of graphs and in this part of the chapter we address the graph as a data structure in detail. The mathematical definition of a graph is given as follows:

Let V be a finite nonempty set, and let $E \subseteq V \times V$. The pair (V, E) is then called a directed graph [...] where V is the set of *vertices*, or *nodes*, and E is its set of (directed) *edges* or *arcs*. We write $G = (V, E)$ to denote such a graph. When there is no concern about the direction of any edge [...], E is a set of unordered pairs of elements taken from V , and G is now called an *undirected graph*. [Gri98, p. 478]

The difference between directed and undirected graphs are the form of the edges. For example: $\{a, b\}$ is unordered pair and $\{a, b\} = \{b, a\}$; (a, b) is an ordered pair and $(a, b) \neq (b, a)$. Figure 3.3 shows an example of an undirected graph and Figure 3.4 an example of an directed graph.

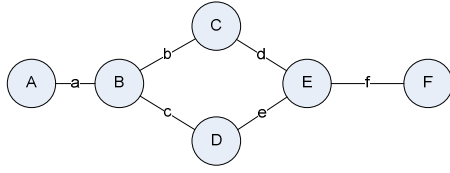


Figure 3.3: An undirected graph with
 $V = \{A, B, C, D, E, F\}$ and E
 $= \{\{A, B\}, \{B, C\}, \{B, D\}, \{C, D\}, \{D, E\}, \{E, F\}\}$

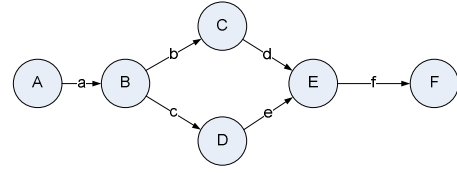


Figure 3.4: A directed graph with
 $V = \{A, B, C, D, E, F\}$ and E
 $= \{(A, B), (B, C), (B, D), (C, D), (D, E), (E, F)\}$

The mathematical definition of a graph consists of a set of nodes and a set of edges. This fits nicely to the definition of a generic data structure. We use this as the basis to define a graph data structure.

DEFINITION 3.19 An element of a graph is called a vertex and a unique vertex is represented by v_i and $v_i \in V$ where $S^{base} = V$

Now that we have defined the elements as the nodes of a graph, we need to define the relation as the set of edges. We give two definitions, one for directed graphs and another for undirected graphs.

DEFINITION 3.20-I In a directed graph the relation R is represented by the edge set E where $E \subseteq V \times V$. Elements of E have the form (v_i, v_j) where $v_i, v_j \in V$. The relation R is asymmetric.

DEFINITION 3.20-II In an undirected graph the relation R is represented by the edge set E where elements of E have the form $\{v_i, v_j\}$ where $v_i, v_j \in V$. The relation R is symmetric.

The only part of the definition of a generic data structure that is not mentioned in the mathematical definition of a graph is the value function. We define this the same way as for any other data structure. Now we can give the definition of a graph data structure:

DEFINITION 3.21 A graph G is represented by $G = (S^{base}, R, \lambda) = (V, E, \lambda)$

3.2.6 Tree

An undirected graph is called a tree if it is connected and does not contain any cycles [Gri98]. Connected means that there must be a path between all pairs of nodes. A cycle is a path from a node that returns back to the node itself without repeating edges. In figure 1, B-C-E-D-B forms a cycle and therefore that graph is not a tree. Although a tree is a graph, we consider it elementary because in practice it is used in a very different way, where it is used to represent a hierarchy. In this sense we speak of parents and children, roots and leafs.

A node can have a child node, which makes this node a parent of the child node. Every node can have only one parent, but a node may have more than one child. A node that has no parent is called a root node and a node that has no children is called a leaf. Figure 3.5 shows a tree and the different types of nodes in a tree.

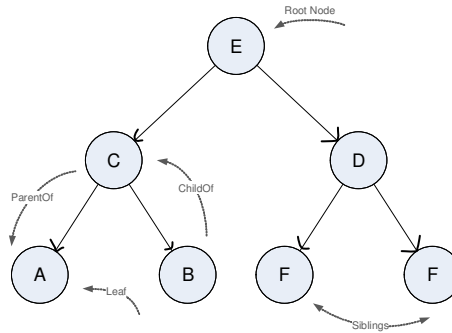


Figure 3.5: A tree

We consider the tree data structure to be a hierarchical tree and this resembles the definition of a directed tree that has a root:

If G is a directed graph, then G is called a directed tree if the undirected graph associated with G is a tree. When G is a directed tree, G is called a rooted tree if there is a unique vertex r , called the root, in G with the in degree of $r = id(r) = 0$, and for all other vertices v , the in degree of $v = id(v) = 1$ [Gri98].

The in-degree $id(v)$ defines how many incoming arcs (edges directed towards the node v). Saying that a root node r has an in degree of 0 means that a root has no parent. Saying that the other nodes v have an in degree of 1 means that they must have exactly one parent. The following two definitions are the conditions for a graph to be tree:

DEFINITION 3.22 The root of a tree is defined by $r = \{v \in V | id(v) = 0\}$

DEFINITION 3.23 A tree T is defined by $T = (S^{base}, R, \lambda) = (V, E_T, \lambda)$. The tree must be constructed such that: it contains a root, is connected, and contains no loops.

3.3 Nested data structures

In chapter 3.1 we mention that we tables and images are frequently used data structures, but that we do not consider them elementary as we can construct these data structures by nesting other elementary data structures. In this part of the chapter we show how this is done. First, we give a definition of a nested data structure and use it to describe a trivial example of a nested data structure. Second, we show how table and image data structures can be described.

DEFINITION 3.24 A nested data structure consists of an outer data structure of which the elements do not have regular values, but consist of inner data structures. A nested data structure may also serve as an inner structure, producing multiple nestings. Elements from an inner outer structure are given a superscript number to distinguish them from elements from the outer structure and elements from other inner structures.

EXAMPLE 3.7

In Chapter 0 we discussed how a set is represented as a data structure. It is possible to create a set of sets. We consider the set $S = \{A, B\}$. The powerset of S is then a set of sets: $P(S) = \{\{A\}, \{B\}, \{A, B\}\}$. The set S can be modeled as a data structure according to DEFINITION 3.7: $S = (\{e_1, e_2\}, \emptyset, \{(e_1, A), (e_2, B)\})$. The powerset of set S can then be modeled as a nested data structure:

$$PS = \left(\begin{array}{l} \{e_1, e_2, e_3\}, \\ \emptyset, \\ \left\{ \begin{array}{l} (e_1, (\{e_1^1\}, \emptyset, \{(e_1^1, A)\})), \\ (e_2, (\{e_1^2\}, \emptyset, \{(e_1^2, B)\})), \\ (e_3, (\{e_1^3, e_2^3\}, \emptyset, \{(e_1^3, A), (e_2^3, B)\})) \end{array} \right\} \end{array} \right)$$

This nested data structure consists of one outer structure and three inner structures. The superscripts of the elements serve to distinguish them from elements from other inner structures.

In the next two subchapters we show how tables and images can be constructed.

3.3.1 Table

We consider tables as they are used in the context of relational databases. Such a table is also called a recordset. A recordset is a set of records, where records are tuples. We represent a table as a data structure by using a set (DEFINITION 3.7) as the outer structure and vectors (DEFINITION 3.10) as the inner structures. The set consists of elements to identify records and the values of these elements are the vectors. The vectors contain the values of the fields in the records.

DEFINITION 3.25 A table T of n records and k columns is defined as a set according to DEFINITION 3.7 with n elements, where each element represents a record and the value of each element is a vector of size k according to DEFINITION 3.10.

EXAMPLE 3.8

We consider the following table of ages and names:

Name	Age
John	52
Frank	29

We represent this table as a data structure:

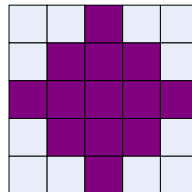
$$T = \left(\begin{array}{l} \{e_1, e_2\}, \\ \emptyset, \\ \left\{ \left(e_1, (\{v_1^1, v_2^1\}, \{(1, v_1^1), (2, v_2^1)\}, \{(v_1^1, John), (v_2^1, 52)\}) \right), \right. \\ \left. \left(e_2, (\{v_1^2, v_2^2\}, \{(1, v_1^2), (2, v_2^2)\}, \{(v_1^2, Pete), (v_2^2, 29)\}) \right) \right\} \end{array} \right)$$

3.3.2 Image

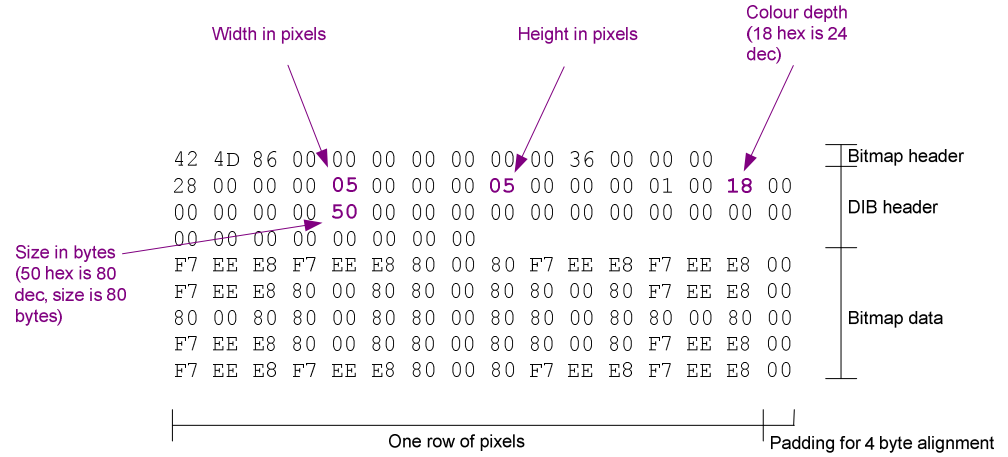
An image is a 2d plane of pixels. In this subchapter we show how an image can be represented by a nested data structure. We start with an example to give an idea of how images are used in practice.

EXAMPLE 3.9

We consider a blown up version of a 5x5 pixels image:



An image is usually stored in a binary format, consisting of a header and pixel data. When this image is stored in the BMP format, the file read out in hexadecimals is as follows:



The header contains information on how to render the image, such as the height and width of the image. The bitmap data consists of RGB values for each pixel, where RGB stands for red, green and blue. An RGB value is a tuple of three numbers, where the numbers represent the shades of the three color bands, giving unique colors. Because this is a 24-bit image, each pixel is represented by three bytes where each byte is used for a band. This gives $2^8 = 256$ possible values for a band and $256^3 = 16,777,216$ unique different colors. In hexadecimals this reads F7 EE E8 for the first pixel. When converted to decimals, we get the three numbers 232, 238 and 247.

We represent an image as data structure by using a nested data structure. The outer structure is a matrix (DEFINITION 3.14) where the elements represent pixels. The inner structures represent the pixels and are formed by vectors (DEFINITION 3.10) of size 3.

DEFINITION 3.26 An image I of n rows and m columns is defined as an $n \times m$ -matrix according to DEFINITION 3.14, with $n \cdot m$ elements, where each element represents a pixel and the value of each element is a vector of size 3 according to DEFINITION 3.10.

EXAMPLE 3.10

We use the three left-most pixels from the top row of the image from EXAMPLE 3.9. A matrix of tuples is given by :
 $[(232, 238, 247) \quad (232, 238, 247) \quad (128, 0, 128)].$

An image consisting of these three pixels can be represented as data structure:

$$I = \left(\begin{array}{l} \{m_1, m_2, m_3\}, \\ \{(1, 1, m_1), (1, 2, m_2), (1, 3, m_3)\}, \\ \left\{ \left((v_1^1, v_2^1, v_3^1), \{(1, 1, v_1^1), (1, 2, v_2^1), (1, 3, v_3^1)\} \right), \{(v_1^1, 232), (v_2^1, 238), (v_3^1, 247)\} \right\}, \\ \left\{ (v_1^2, v_2^2, v_3^2), \{(1, 1, v_1^2), (1, 2, v_2^2), (1, 3, v_3^2)\} \right\}, \{(v_1^2, 232), (v_2^2, 238), (v_3^2, 247)\} \right\}, \\ \left\{ (v_1^3, v_2^3, v_3^3), \{(1, 1, v_1^3), (1, 2, v_2^3), (1, 3, v_3^3)\} \right\}, \{(v_1^3, 128), (v_2^3, 0), (v_3^3, 128)\} \right\} \end{array} \right)$$

3.4 Properties of data structures

As we have seen the different elementary structures are constructed in different ways. It can be recalled from DEFINITION 3.5 that a data structure consists of a set of elements and a set of relations. It is the set of relations that makes elementary structures different from each other. We will now show how these differences can be distilled into a set of properties that can be used to classify the elementary data structures.

Some properties are consistent for the whole class. These are the properties that distinguish one class from another.

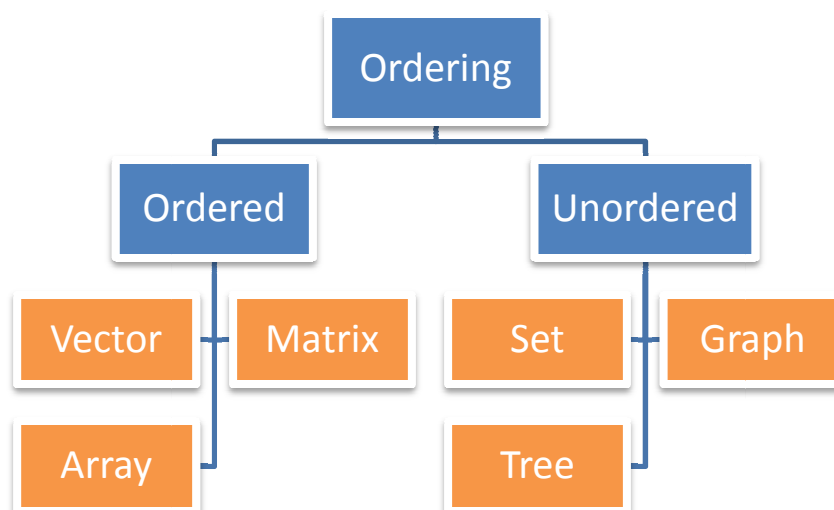
3.4.1 Ordering

As we have seen for vectors, matrixes and arrays the relationship between elements can be modeled as a function that maps element to a natural numbers. Sets have no relations at all and graphs and trees have relations between two elements.

The vector, matrix and array functions are not just for looking up elements. They provide a means of arranging the elements. Because one number is larger than the other, we can say that one element in a vector precedes another.

PROPERTY 1 A data structure **S** is called ordered if R can be modeled as a function F_V , F_M or F_A .

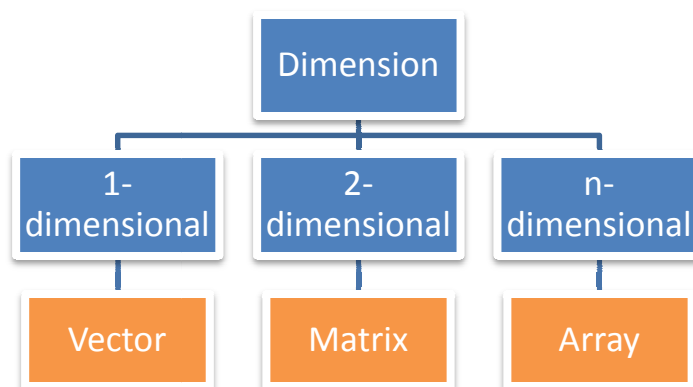
The following tree shows which data structures are ordered and which are not.



3.4.2 Dimension

The three ordered structures differ in one way and that is the amount of natural numbers that is needed to identify an element. This is the dimension of a data structure, as specified in DEFINITION 3.15.

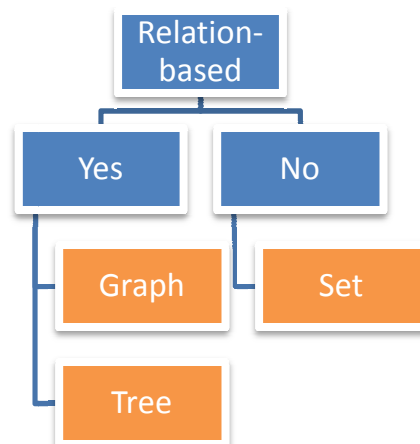
PROPERTY 2 The dimension of a data structure is given by the amount of natural numbers that is needed to identify an element



3.4.3 Relation based

The main difference between graphs and sets is that graphs contain a relation between the elements. Sets are completely free of relations.

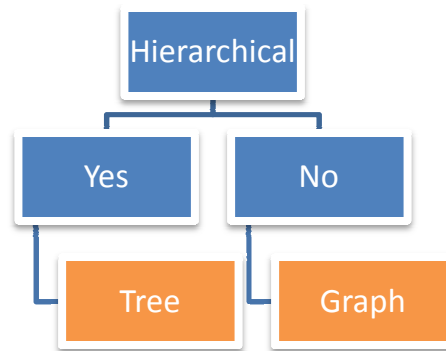
PROPERTY 3 A data structure **S** is relation based if $R \neq \emptyset$



3.4.4 Hierarchical relations

The difference between a graph and a tree is that a tree is hierarchical and a graph is not.

PROPERTY 4 A data structure has the hierarchical property if it is directed and contains no loops.



3.5 Properties of instances: Size

Some properties such as size are particular for the instance of the data structure. These are properties that can be used to distinguish instance within a specific class. Other examples are color depth/palette for an image.

PROPERTY 5 The size of a data structure is given by the amount of elements:

$$size(\mathbf{S}) = |S^{base}|$$

- The size of an n –vector is given by a single integer n that is equal to the amount of elements: $n = |S^{base}|$.
- The size of an $m \times n$ –matrix is given by two integers whose product is equal to the amount of elements: $m \times n = |S^{base}|$
- The size of an array depends on the amount of dimensions d . The size is then given by d integers whose product is equal to the amount of elements:

$$\prod_{i=1}^d n_i = |S^{base}|$$

- The size of a set is equal to the amount of elements which is $|S^{base}|$
- The size of a graph is denoted by the amount of nodes which is equal to the amount of elements: $|V| = |S^{base}|$
- The size of a tree is denoted by the amount of nodes which is equal to the amount of elements: $|V| = |S^{base}|$

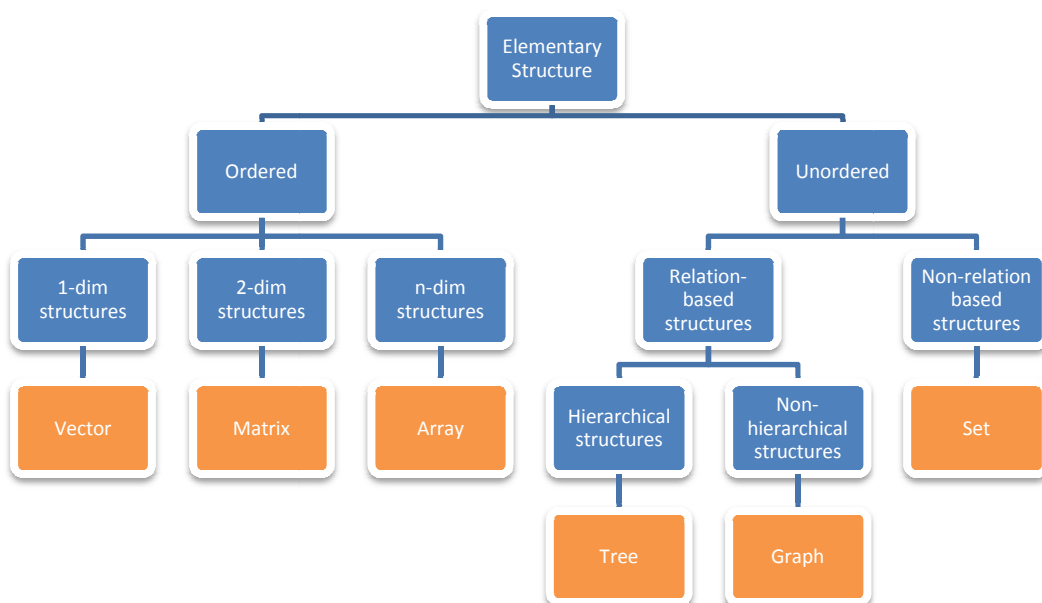
3.6 Overview

The following table shows an overview of all the data structures and their properties:

Table 3.1: Overview of elementary data structures and their properties

Structure	Ordering	Dimension	Relation-based	Hierarchical	Size
Vector	Ordered	1			n
Matrix	Ordered	2			$m \times n$
Array	Ordered	n			$\prod_{i=1}^d n_i$
Set	Unordered		No		$ S^{base} $
Graph	Unordered		Yes	No	$ V $
Tree	Unordered		Yes	Yes	$ V $

Using the properties specified in subchapter 3.4 the elementary data structures are classified by property:



3.7 Conclusion

In this final part of the chapter we will reflect on the results that try to answer the research questions. In this chapter we have addressed the following research question:

RESEARCH QUESTION 1 How can the description of data structures be formalized?

First we have given a formal definition of a generic data structure. The key result of this formalization is the separation of the elements, their values and the structure. The advantage is the bridge that is created between mathematical abstraction and practical use in computer science applications. By separating elements from their values we have provided a method to distinguish between elements that have the same value. The limitation of our model is that we have provided place for only one relation on the elements. It may be argued that there are data structures where more than one relation on the elements, perhaps even of different type, may be needed.

We have proceeded by defining a list of elementary data structures: sets, vectors, matrixes, arrays, graphs and trees. For each of these data structures we have shown how the definitions that we formulated for the generic data structure can be applied. The list of elementary data structures is based on practical examples. It could be argued that the list should be different. The list could be shorter, for example by arguing that vectors and matrixes are also arrays. This way the list could be limited to sets, arrays and graphs, with possible even modeling a set as a graph without edges. The result would be such a list that is too short to be practicable. It may also be argued that other data structures should also be considered elementary. We have chosen not to do so as we found that it some data structures were specializations of data structures from our list, while others could be modeled as a nestings of our elementary data structures. We have shown that it was possible to model tables and images as nestings of sets of vectors and matrixes of vectors, respectively.

Finally we have described four properties to classify the elementary data structures. When it is argued that more elementary data structures are needed, possible more properties are needed. It may also be possible to use other properties than the current ones for the classification. The current four properties are all specific to the relation on the set of elements for each data structure and proved sufficient to make a classification.

In the next chapter we will use the results from this chapter to describe how one data structure can be transformed into another data structure.

Classification of Transformations

In Chapter 3 we have defined a list of elementary data structures and classified them by different properties. In this chapter we will show how these properties can be used to describe possible transformations of these structures. We start with a transformation model for unary transformations. Next, we show how a transformation can be modeled as a series of elementary operations. We describe different properties to characterize transformations to define a class of transformations, for which an important part of the transformation can be described by the transformation of the position of contributing elements. Finally, the framework is extended to n -ary transformations.

The research question that we will answer in this chapter is:

RESEARCH QUESTION 2

What type of transformations of data structures exist and what are their properties?

4.1 Transformation model

In the introduction of this thesis we have seen that processing elements of a workflow convert input data streams to output data streams. A data stream is a sequence of tuples ordered by a timestamp. Each tuple may be a data structure as defined in Chapter 3. A processing element consumes tuples to produce new tuples. In the introduction we describe that elements in a data structure may be annotated. In order to understand how the output data structure can be annotated we investigate in this part of the chapter how an input data structure is transformed into an output data structure. We focus first on unary transformations, where only one input data structure is transformed into an output data structure. At the end of the chapter we will address the n -ary transformations.

We consider first three examples of different types of transformations.

EXAMPLE 4.1

The vector $V = [1,2,3]$ is transformed into $V' = [3,2,1]$ by mirroring the elements. Here the input vector is defined according to definition X as:

$$V = (\{v_1, v_2, v_3\}, \{(1, v_1), (2, v_2), (3, v_3)\}, \{(v_1, 1), (v_2, 2), (v_3, 3)\})$$

The output vector is defined as:

$$V' = (\{v_1, v_2, v_3\}, \{(1, v_3), (2, v_2), (3, v_1)\}, \{(v_1, 1), (v_2, 2), (v_3, 3)\})$$

EXAMPLE 4.2

The matrix $M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ is transformed into the vector $V = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$ by summing the values in each row. The matrix is defined according to definition X as

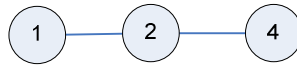
$$M = \left(\begin{array}{l} \{m_1, m_2, m_3, m_4\}, \\ \{(1,1, m_1), (1,2, m_2), (2,1, m_3), (2,2, m_4)\}, \\ \{(m_1, 1), (m_2, 2), (m_3, 3), (m_4, 4)\} \end{array} \right)$$

The vector is defined according to definition X as:

$$V = (\{v_1, v_2\}, \{(1, v_1), (2, v_2)\}, \{(v_1, 3), (v_2, 4)\})$$

EXAMPLE 4.3

The matrix $M = \begin{bmatrix} 1 & 2 \\ 0 & 4 \end{bmatrix}$ is transformed into a graph where a node is created if an element of the matrix has a value >0 and an edge between two nodes is created when the corresponding elements in the matrix are neighbors. Two elements are considered a neighbor when they are next to each other or above/below each other. This results in the graph G:



The matrix M is defined as:

$$M = \left(\begin{array}{l} \{m_1, m_2, m_3, m_4\}, \\ \{(1,1, m_1), (1,2, m_2), (2,1, m_3), (2,2, m_4)\}, \\ \{(m_1, 1), (m_2, 1), (m_3, 0), (m_4, 1)\} \end{array} \right)$$

And the graph G is defined as:

$$G = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_3)\}, \{(v_1, 1), (v_2, 2), (v_3, 4)\})$$

The above examples show three quite different transformations. We can observe a few differences. In EXAMPLE 4.1 the elements are only moved to a new position. The values do not change, and no new elements are created. In the other two examples new elements are created in the process. In EXAMPLE 4.2 the size of the output vector is determined only by the amount of rows in the input matrix. This means that the output structure depends only on the input structure. In EXAMPLE 4.3 the size of the output graph depends on more than the input structure, as it depends on the amount of elements with a value >0 . This means that a 2×2 matrix can result in a graph of size 0, 1, 2, 3 or 4. Another difference is that the values of the vector are calculated by summing values from the matrix at particular positions. This means that in order to produce the values output, both the structure and the values from the input must be taken into account.

In the rest of this chapter we investigate these differences more deeply. We present new definitions and properties to characterize transformations. First we start with a general definition of a transformation of a data structure. In chapter 3 we gave a definition for a general data structure. DEFINITION 3.5 reads:

A data structure S is represented by $\mathbf{S} = (S^{base}, R, \lambda)$

If one data structure is transformed into another, clearly the three components of a data structure must be transformed into three new components. We define a transformation as the creation of three output components based on three input components.

DEFINITION 4.1 A transformation of an input data structure into an output data structure is defined as $T: (S^{base}, R, \lambda) \rightarrow (S^{base'}, R', \lambda')$.

The statement in DEFINITION 4.1 only specifies that one complete data structure is transformed into another complete data structure. It does not specify which parts of the data structure are transformed and how they are transformed. For example, only the value function may be transformed, while leaving the base-set and the relation unchanged. The following definitions can be used to provide more detail on what actually changes in a transformation.

DEFINITION 4.2 When a transformation T changes the base-set S^{base} this effect is denoted as T_E .

DEFINITION 4.3 When a transformation T changes the relation R this effect is denoted as T_S .

DEFINITION 4.4 When a transformation T changes the value function λ this effect is denoted as T_D .

The transformation T is then a combination of the three effects T_D, T_E, T_S .

DEFINITION 4.5 A transformation T is a combination of three effects $T \subseteq \{T_D, T_E, T_S\}$. Using Definition 4.2 - Definition 4.4 we state that the following conditions must always be true:

$$T_D \in T \Leftrightarrow \lambda' \neq \lambda \quad (1)$$

$$T_E \in T \Leftrightarrow S^{base'} \neq S^{base} \quad (2)$$

$$T_S \in T \Leftrightarrow R' \neq R \quad (3)$$

EXAMPLE 4.4

For each of the three examples from the beginning of this subchapter we can formulate the effects of the transformation:

- EXAMPLE 4.1: $T = \{T_S\}$
- EXAMPLE 4.2: $T = \{T_D, T_E, T_S\}$
- EXAMPLE 4.3: $T = \{T_D, T_E, T_S\}$

4.2 Transformation Diagram

In Chapter 3 we have shown how an instance of a data structure can be represented by a data structure diagram, or DS-diagram (DEFINITION 3.6 and DEFINITION 3.11). In this part of the chapter we extend the notation of the DS-diagram to visualize instances of particular transformations.

DEFINITION 4.6 A transformation diagram is an extension of the data structure diagram (DEFINITION 3.6 and DEFINITION 3.11) and is used to visually model a specific instance of a transformation of a data structure. Figure 4.1 shows a transformation diagram. On the left hand side the input is shown and on the right hand side the output. The arrows from left-to-right represent the transformation. When an element, index or value is unchanged after the transformation a solid arrow is drawn. A dashed arrow represents a change: an element, index or value may be directly transformed or contribute in another way to the output.

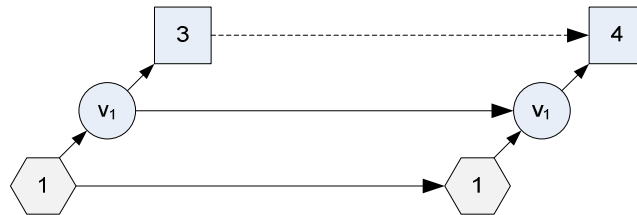


Figure 4.1: A Transformation Diagram

In some cases it is possible to describe the complete transformation using the above notation. In other cases it may be useful to describe only part of the transformation. Figure 4.2 shows the transformation diagram for EXAMPLE 4.2.

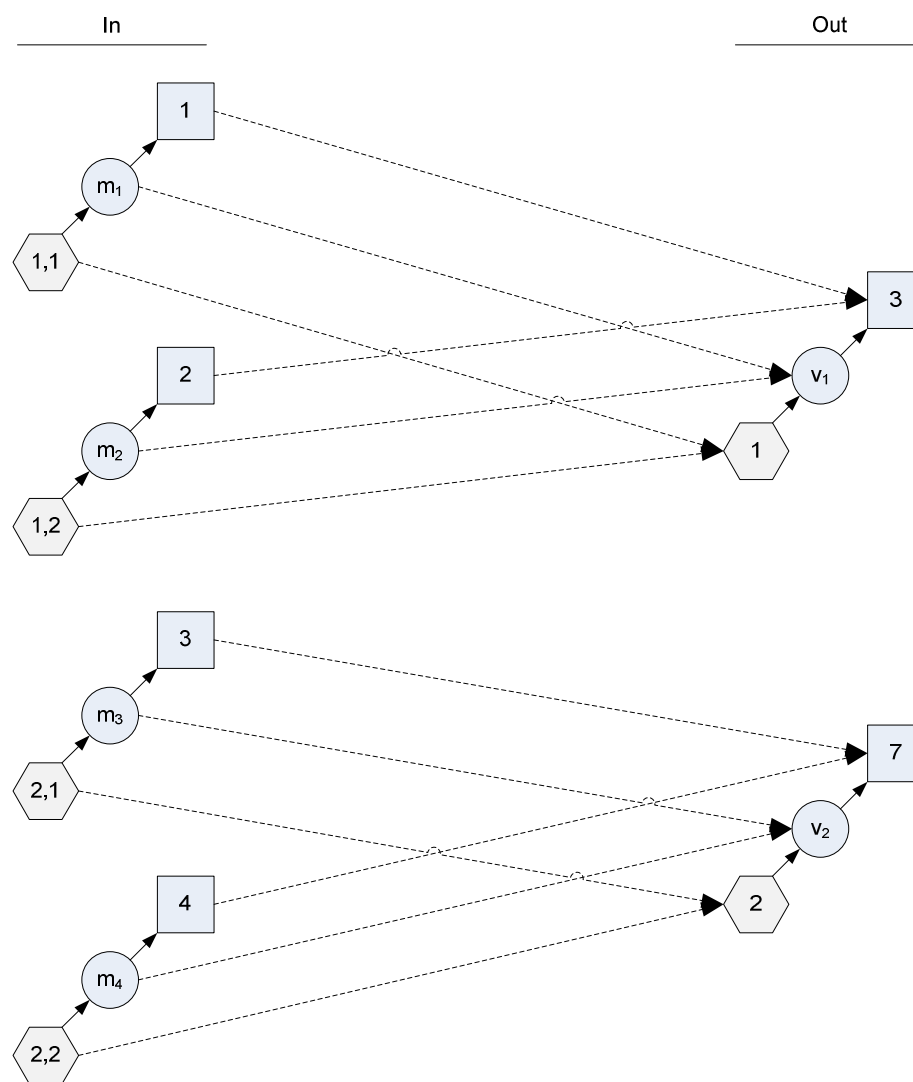


Figure 4.2: A transformation diagram for Example 4.2

4.3 Elementary Operations

The transformation model from DEFINITION 4.5 only describes what parts of the data structure are affected by the transformation, but it does not describe how these parts are affected by the transformation. In this part of the chapter we describe how a transformation can be described by algorithms that involve the execution of elementary operations. We define elementary operations as very small transformations of a data structure. It changes only one element, index or value. A complete instance of a transformation can then be modeled as a series of elementary operations. A transformation may either be a series of elementary operations that operate on the input data structure, or the transformation may read the input using an algorithm and then invoke elementary operations to fill a new data structure.

A data structure consists of three parts (DEFINITION 3.5): elements, relations and values. This means that the set of elementary operations must include at least operations that cause the creation and deletion of relations, elements, and values. For each type of data structure these elementary operations are different. We define for each elementary data structure one set of elementary operations. There may be other variations possible, but that does not matter as it is our aim to show how a complete set of operations can be constructed.

4.3.1 Set

According to DEFINITION 3.7 a set data structure is defined as $S = (S^{base}, R, \lambda) = (\{e_1, \dots, e_n\}, \emptyset, \lambda)$ where $n = |S^{base}|$. This means that a set is a data structure without a relation. The elementary operations only need to cover operations on the elements and the values. An element must have a value, so we add an element and a value with one operation. When an element is deleted, its value must also be automatically deleted.

1. $Insert(element, value) = \begin{cases} T_D: \lambda' := \lambda \cup \{(element, value)\} \\ T_E: S^{base'} := S^{base} \cup \{element\} \end{cases}$
2. $Delete(element) = \begin{cases} T_D: \lambda' := \lambda \setminus \{(element, \lambda(element))\} \\ T_E: S^{base'} := S^{base} \setminus \{element\} \end{cases}$

EXAMPLE 4.5

The vector $[3 \ 2 \ 1]$ is transformed into the set $\{1,2,3\}$. In this case the transformation reads the input and constructs a new data structure using the elementary operations.

```
// Before: S = ( $\phi, \phi, \phi$ )
V
= ( $\{e_1, e_2, e_3\}, \{(1, e_1), (2, e_2), (3, e_3)\}, \{(e_1, 3), (e_2, 2), (e_3, 1)\}$ )
for i = 1..3 do
    Insert( $F_V(i), \lambda(F_V(i))$ )

//After: S = ( $\{e_1, e_2, e_3\}, \emptyset, \{(e_1, 3), (e_2, 2), (e_3, 1)\}$ )
```


4.3.2 Vector, Matrix and Array

The difference between a vector and a set is that the elements are indexed by a unique integer. For each element there is exactly one index number and each index number must refer to exactly one element (DEFINITION 3.9-I). This means that we cannot create or delete an element without performing the same operation on the corresponding index. Another property of the vector is that the list of index numbers must be consecutive. This means that if we want to insert an element at a certain position, some other elements must move to new positions. Instead of defining such an operation, we define an operation that inserts an element at the end and an operation that swaps two elements. These two operations can then be combined to insert an element at a specified position. In the same way we define an operation that deletes an element at the end. This operation can then be combined with swaps to delete an element at a certain position.

$$\begin{aligned}
 1. \text{ Insert}(\text{element}, \text{value}) &= \begin{cases} T_D: \lambda' := \lambda \cup \{(\text{element}, \text{value})\} \\ T_E: S^{base'} := S^{base} \cup \{\text{element}\} \\ T_S: R' := R \cup \{(|S^{base}| + 1, \text{element})\} \end{cases} \\
 2. \text{ Delete} &= \begin{cases} T_D: \lambda' := \lambda \setminus \{(F_V(|S^{base}|), \lambda(F_V(|S^{base}|)))\} \\ T_E: S^{base'} := S^{base} \setminus \{F_V(|S^{base}|)\} \\ T_S: R' := R \setminus \{(|S^{base}|, F_V(|S^{base}|))\} \end{cases} \\
 3. \text{ Swap}(i, j) &= \begin{cases} T_S: R' := R \setminus \{(i, F_V(i)), (j, F_V(j))\} \\ \quad \cup \{(i, F_V(j)), (j, F_V(i))\} \end{cases}
 \end{aligned}$$

The elementary operations for matrix and array data structures work in the same way as the elementary operations for the vector data structure. In these operations, indexes are replaced by respectively two-dimensional and higher dimensional indexes and the function F_V is replaced by F_M and F_A respectively.

EXAMPLE 4.6

Let the element v_4 with the value 2 be added, and elements at position 2 and 3 be swapped in the vector

$$V = (\{v_1, v_2, v_3\}, \{(1, v_1), (2, v_2), (3, v_3)\}, \{(v_1, 4), (v_2, 2), (v_3, 6)\}):$$

$$\text{Insert}(v_4, 2) = \begin{cases} T_E: S^{base'} := \{v_1, v_2, v_3\} \cup \{v_4\} = \{v_1, v_2, v_3, v_4\}; \\ T_S: R' := \{(1, v_1), (2, v_2), (3, v_3)\} \cup \{(3 + 1, v_4)\} \\ \quad = \{(1, v_1), (2, v_2), (3, v_3), (4, v_4)\} \\ T_D: \lambda' := \{(v_1, 4), (v_2, 2), (v_3, 6)\} \cup \{(v_4, 2)\} \\ \quad = \{(v_1, 4), (v_2, 2), (v_3, 6), (v_4, 2)\} \end{cases}$$

and

$$\text{Swap}(2, 3) = \begin{cases} T_S: R' := R \setminus \{(2, F_V(2)), (3, F_V(3))\} \cup \{(2, F_V(3)), (3, F_V(2))\} \\ \quad = \{(1, v_1), (2, v_3), (3, v_2)\} \end{cases}$$

4.3.3 Graph

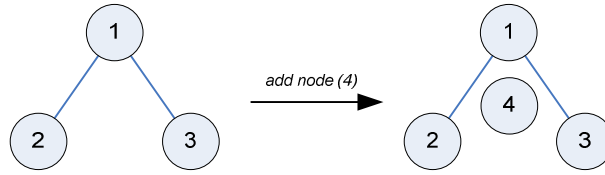
In a graph data structure the elements and the relations are called nodes and edges. In a graph, nodes may exist without edges pointing to them. This means that nodes and edges can be created and deleted separately from each other.

Inserting a node is exactly the same as adding an element to a set.

$$\text{InsertNode}(\text{node}, \text{value}) = \begin{cases} T_D: \lambda' := \lambda \cup \{(\text{node}, \text{value})\} \\ T_E: S^{\text{base}'} := S^{\text{base}} \cup \{\text{node}\} \end{cases}$$

EXAMPLE 4.7

Let a node with value 4 be added to the following graph:



$$\begin{aligned} &\text{InsertNode}(n_4, 4) \\ &= \begin{cases} T_S: S^{\text{base}'} := S^{\text{base}} \cup \{n_4\} = \{n_1, n_2, n_3, n_4\} \\ T_D: \lambda' := \lambda \cup \{(n_4, 4)\} = \{(n_1, 1), (n_2, 2), (n_3, 3), (n_4, 4)\} \end{cases} \end{aligned}$$

Deleting a node is a bit more difficult than inserting it. A node may exist without relations, but a relation cannot exist without nodes. This gives us two options: deleting a node together with all its relations or prohibiting the deletion of a node that has relations. We opt for the second, because it is easier to define:

$$\begin{aligned} &\text{DeleteNode}(\text{node}) \\ &= \begin{cases} T_D: \lambda' := \lambda \setminus \{(\text{node}, \lambda(\text{node}))\} \\ T_E: S^{\text{base}'} := S^{\text{base}} \setminus \{\text{node} \mid \exists \text{node2} \cdot (\text{node}, \text{node2}) \in R \vee (\text{node2}, \text{node}) \in R\} \end{cases} \end{aligned}$$

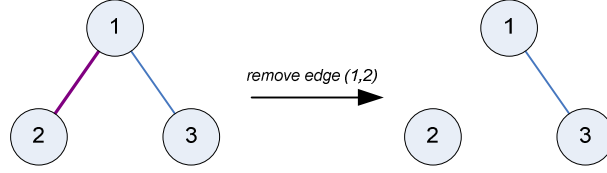
If we want to delete the node with value 2 from the graph in EXAMPLE 4.7 we have a problem. There is an edge connected to this node, and this edge must be removed first. This is easier than deleting nodes because no checks need to be made. An edge can be deleted even if it is connected to a node, because nodes are allowed to exist without edges.

$$\text{DeleteEdge}(\text{node1}, \text{node2}) = \begin{cases} T_S: R' := R \setminus \{(\text{node1}, \text{node2})\} \end{cases}$$

Now it is possible to delete the edges between 1 and 2, and then delete the node with value 2 as is shown in EXAMPLE 4.8.

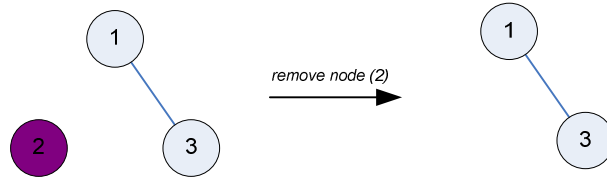
EXAMPLE 4.8

Let the edge between nodes 1 and 2, and consequently the node with value 2 be deleted from the following graph:



$$\text{DeleteEdge}(n_1, n_2) = \left\{ T_S: R' := R \setminus \{(n_1, n_2)\} \right\}$$

Now that the edge is deleted, it is also possible to delete the node with value 2:



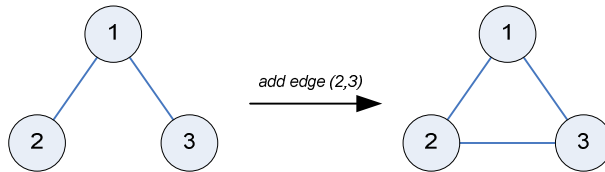
$$\text{DeleteNode}(n_2) = \left\{ \begin{array}{l} T_D: \lambda' := \lambda \setminus \{(n_2, \lambda(n_2))\} \\ T_E: S^{base'} := S^{base} \setminus \{n_2\} \end{array} \right\}$$

The last operation we need to make the set of elementary operations of a graph complete is the addition of an edge. Because an edge cannot be connected to non-existing nodes, a check must be made first if both nodes of the edge that needs to be added really exist.

$$\text{InsertEdge}(node1, node2) = \left\{ T_S: R' := R \cup \{(node1, node2) | node1, node2 \in S^{base}\} \right\}$$

EXAMPLE 4.9

Let a new edge between nodes 2 and 3 be added to the following graph:



$$\text{InsertEdge}(n_2, n_3) = \left\{ \begin{array}{l} T_S: R' := R \cup \{(n_2, n_3) | n_1, n_3 \in S^{base}\} \\ \quad = \{(n_1, n_2), (n_1, n_3), (n_2, n_3)\} \end{array} \right\}$$

4.3.4 Tree

In Chapter 3 we have defined a tree as a directed connected graph with a root and without cycles. Because of the differences with a graph the elementary operations are also different. Each node except the root that is inserted must have a parent. Therefore edges and nodes must be created at the same time:

$$Insert(parent, node, value) = \left\{ \begin{array}{l} T_D: \lambda' := \lambda \cup \{(node, value)\} \\ T_E: S^{base'} := S^{base} \cup \{node\} \\ T_S: R' := R \cup \{(parent, node)\} \end{array} \right\} \left| \begin{array}{l} \\ \\ \\ \end{array} \right. parent \in S^{base} \right\}$$

When a node with children would be deleted the output tree would be unconnected, which is not allowed. In order to keep things simple, we only allow leafs (nodes without children) to be deleted. When a leaf is deleted its relation with its parent is also deleted:

$$Delete(node) = \left\{ \begin{array}{l} T_D: \lambda' := \lambda \setminus \{(node, \lambda(node))\} \\ T_E: S^{base'} := S^{base} \setminus \{node\} \\ T_S: R' := R \setminus \{parent, node \mid \exists parent \cdot (parent, node) \in R\} \end{array} \right\} \left| \begin{array}{l} \\ \\ \neg node2 \cdot (node, node2) \in R \end{array} \right\}$$

A tree contains a root which is a node without a parent. We cannot use the insert operation to create root because that operation needs a parent as a parameter. Deleting a root is not possible, because it has children. The delete operation will rightly fail on an attempt to delete the root, just as any other node with children. To create a root we define a specific operation, which can only be used when the tree is empty:

$$CreateRoot(node, value) = (S^{base} = \emptyset) \Rightarrow \left\{ \begin{array}{l} T_S: S^{base'} := \{node\} \\ R' := \emptyset \\ T_D: \lambda' := \{(node, value)\} \end{array} \right\}$$

It is also possible to make another node the root. This is effectively done by reversing the direction of the arrow between the root and one of its children:

$$SwapRoot(oldRoot, newRoot) = \left\{ \begin{array}{l} T_S: R' := R \setminus \{(oldRoot, newRoot)\} \cup \{(newRoot, oldRoot)\} \end{array} \right\} \left| \begin{array}{l} (oldRoot, newRoot) \in R \\ \neg \exists node \cdot (node, oldRoot) \in R \end{array} \right\}$$

4.4 Transformation Properties

In the previous subchapter we have seen that a transformation of a data structure can be modeled as a sequence of elementary operations. For any specific instance of a transformation this sequence may be different. It is possible to specify an algorithm that produces this sequence based on the input data structure, but specifying this algorithm is still quite a lot of work, and it comes close to defining the actual algorithm that underlies the transformation. We are looking for a method to describe how (a part of) the transformation is performed that is less cumbersome. As we have seen from the examples in subchapter 4.1 not all transformations are as complex. In this part of the chapter we investigate properties of transformations that can be used to characterize transformations. We will use these properties to distinguish between transformations for which we can describe how input elements contribute to output elements, and transformations for which we cannot.

4.4.1 Transformation Effect Dependency

In subchapter 4.1 we state that a transformation is a combination of three effects (DEFINITION 4.5). These three effects are denoted T_D , T_E and T_S and mean that the transformation affects the data, elements and/or structure of the output. Each of these effects may depend on different parts of the input structure. For example, the output structure in the matrix-to-graph transformation from EXAMPLE 4.3 depends on both the input structure and the input data, while the output structure in the matrix-to-vector transformation of EXAMPLE 4.2 depends only on the input structure. In this subchapter we investigate what the possible dependencies are of the output parts on the input parts.

There are three parts in a data structure: the data part D , the element part E and the structure part S . Each output part may depend on zero, one or more of these parts, which gives $\binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 8$ theoretically possible combinations.

DEFINITION 4.7 We extend DEFINITION 4.2 - DEFINITION 4.4 to denote the dependency of an output part on the input. The input is denoted as a combination of the characters D , E and S and is used as a superscript in the transformation effect. For example, T_S^{DS} means that the output structure depends on the input data and structure.

With 8 possible dependencies on the input for each effect, and three effects, we have $8 \times 8 \times 8 = 512$ theoretically possible types of transformations. In practice however, not all of these variants occur very frequently. Some of them may occur very rarely or even not at all. For each effect we list the most common dependency variants and the examples in which they occur. In practice the output only depends on combinations of the data and structure part. Very rarely are the elements directly involved in a transformation. Either they are selected using the structure, for example by their position using the inverse matrix function, or by their value.

Table 4.1: Dependency of the output data on parts of the input

Depends on	Symbol	Description
\emptyset	T_D^\emptyset	Data ex-nihilo: The output data is not influenced in anyway by the input. The transformation generates the data without any input. For example, a transformation may produce values using a random-number generator. This variant occurs in processing elements that act as sources (See chapter 1 Introduction for the definition of a source).
λ	T_D^D	Data-by-data: In this case the values of the output data only depend on the values of the input data. See EXAMPLE 4.10 .
R	T_D^S	Data-by-structure: In this case the input data is not used to produce the output data, but only the input structure is used. For example, the edges of a graph may be transformed into values of an adjacency matrix. See EXAMPLE 4.11 .
R, λ	$T_D^{D,S}$	Data-by-data&structure: This is a very common variant, where the values in the output data depend on both the input structure and the input data. See EXAMPLE 4.2 . Here all values in a row are summed. To produce an output value, elements are retrieved using the inverse matrix function and then the value function is used to find the value of the elements.

[EXAMPLE 4.10](#)

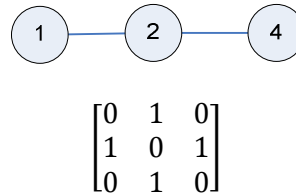
The matrix $M = \begin{bmatrix} 6 & 1 \\ 4 & 3 \end{bmatrix}$ is transformed into a new matrix by giving all the elements a new value using the formula $x \bmod 4$. The new matrix then is: $M = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$. This transformation does not affect the structure or the elements, so $T = T_D^D$.

Table 4.2: Dependency of the output structure on parts of the input

Depends on	Symbol	Description
\emptyset	T_S^\emptyset	Structure ex-nihilo: The output structure does not depend on any part of the input. This is the case for example when the output of a transformation has a fixed structure. Processing elements that act as sources in a workflow create structure ex-nihilo.
λ	T_S^D	Structure-by-data: The output structure depends only on the input data. This happens when a graph is created from an adjacency matrix. See also the inverse of this transformation in the variant T_D^S and EXAMPLE 4.11 .
R	T_S^S	Structure-by-structure: This is a very common variant when converting vectors, matrixes and arrays into vectors, matrixes and arrays. Only the position is used to retrieve elements from the input. See EXAMPLE 4.2 .
R, λ	$T_S^{D,S}$	Structure-by-data&structure: This variant occurs when both the position and the value of an element are used to determine the output structure. In EXAMPLE 4.3 edges are created based on the value of an element and its position.

EXAMPLE 4.11

A graph may be transformed into an adjacency matrix. Each row and column corresponds to a node. When two nodes are connected a 1 is marked at the crossing of the corresponding row and column. An adjacency matrix can be transformed into a graph again. The following graph and adjacency matrix are the same.

**Table 4.3: Dependency of the output elements on part of the input**

Depends on	Symbol	Description
\emptyset	T_E^\emptyset	Elements ex-nihilo: elements are created without using any parts of the input. This variant occurs in processing elements that act as sources.
λ	T_E^D	Element-by-data: A transformation may create new elements or delete elements from the base-set. In this case the creation or deletion of elements depends only on the value of elements. The matrix-to-graph transformation of EXAMPLE 4.3 creates nodes only on the basis of the values of the elements in the matrix.
R	T_E^S	Element-by-structure: In this case the creation or deletion of elements depends only on the structure. In the matrix-to-vector transformation of EXAMPLE 4.2 new elements are created only on the basis of the input structure. It does not matter what the values of the elements are, it only matters how many rows there are in the matrix.
R, λ	T_E^{DS}	Element-by-data&structure: In this case the creation or deletion of elements depend both on the value and position of elements from the input.

The three transformation effects may share the same input part dependencies. This is the case in [EXAMPLE 4.2](#). The transformation produces a vector and creates new elements for that vector. The values of the new elements depend on their position in the vector, because the value is based on the sum of the values of the elements in the corresponding row in the matrix. The transformation must create new elements, new values and a new structure all at once. The following example shows that this does not always have to be the case.

EXAMPLE 4.12

Let T be a transformation that causes a 2×2 matrix to be horizontally flipped and all values negated. $T: \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$. Here all values in the output depend only on the values from the input. It doesn't matter where an element is located; its value simply is negated. At the same time it doesn't matter what the value of an element is, it simply is moved to a new position.

When the transformation effects share similar input part dependencies, the transformation cannot be modeled as serializable sub transformations. The transformation effects are called co-dependent. In the case of EXAMPLE 4.12 the transformation effects are independent: it does not matter whether we first negate the values or flip the structure.

In most cases when new elements are created, the value function or the relation must also change, because they have the base set as their domain or co-domain. This is not always true. For example, new nodes without a value and edges may be added to a graph. When vectors, matrixes and arrays are involved, at least the relation must change, because the function that maps an index number to an element is bijective and must therefore be specified for every element.

4.4.2 Contributing Element Cardinality

In subchapter 4.1 we describe three examples of transformations. In EXAMPLE 4.2 we see that all the elements in a row from the input matrix contribute to one element in the output vector. In EXAMPLE 4.3 nodes in the output graph are based on only one contributing element from the input matrix.

DEFINITION 4.8 In a transformation elements from the input may contribute to the elements of the output. The contributing element cardinality specifies how many elements from the input contribute to elements from the output.

When describing the relations with output elements for an input element, it may be the case that an input element contributes to only one output element. When all input elements contribute only to one output element, the transformation is called *one-to-one*. It is also possible that an input element contributes to more than one output element. In this case the transformation is called *one-to-many*.

DEFINITION 4.9 A transformation is *one-to-one* if each input element contributes to only one output element.

EXAMPLE 4.13

Let the transformation T add the values of each element in the vector $V = [3 \ 6 \ 9]$ to its right neighbor (the value of the last element is added to the first): $V' = [12 \ 9 \ 15]$. Each element from the input is related to only one element from the output thus T is called one-to-one.



Figure 4.3: Diagram of a one-to-one transformation

DEFINITION 4.10 A transformation is *one-to-many* if some or all input elements contribute each to more than an output element.

EXAMPLE 4.14

Let the transformation T multiply the input vector $V = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ with the vector $\begin{bmatrix} 1 & 0 \end{bmatrix}$. Then we get $M = V \times \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 \times 1 & 2 \times 0 \\ 4 \times 1 & 4 \times 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix}$. Here each element of V is used twice and we say that T is one-to-many.

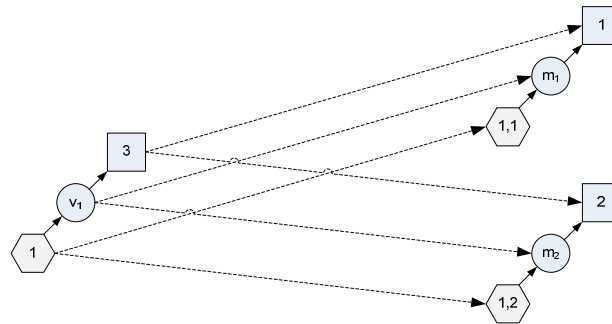


Figure 4.4: Diagram of a one-to-many transformation

Instead of describing to how many output elements an input element contributes it is also possible to describe the reverse. In that case we describe how many input elements contribute to a certain output element. When each output element is related to only one input element we have the same scenario as in DEFINITION 4.9 and the transformation is called *one-to-one*. When an output element is related to more than one input element we say that the transformation is *many-to-one*.

DEFINITION 4.11 A transformation is *many-to-one* if more than one input element contributes to some or all of the output elements.

EXAMPLE 4.15

Let the matrix $\begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$ be transformed into the vector $\begin{bmatrix} 3 \\ 7 \end{bmatrix}$ by horizontally summing all the elements in a row. Here each time two input elements are used to produce one output element and the transformation is many-to-one.

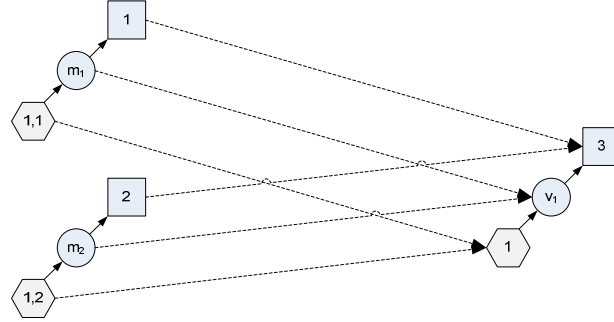


Figure 4.5: A diagram of a many-to-one transformation

It is also possible that a transformation is a combination of the scenario's described in DEFINITION 4.10 and DEFINITION 4.11. We call these transformations *many-to-many*, because an input element may be related to more than one output element and at the same time such an output element may be related to more than one input element.

DEFINITION 4.12 A transformation is *many-to-many* if some or all input elements contribute to more than one output element and at the same time more than one input element contributes to some or all of the output elements.

EXAMPLE 4.16

Let the transformation T produce the matrix $M' = M^2 = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a \times a + b \times c & a \times b + b \times d \\ c \times a + d \times c & c \times b + d \times d \end{bmatrix}$. Here the element in the left-top corner of the matrix M' is produced using the input elements a, b and c . At the same time element a is used to produce three elements in the output matrix. Therefore transformation T is many-to-many.

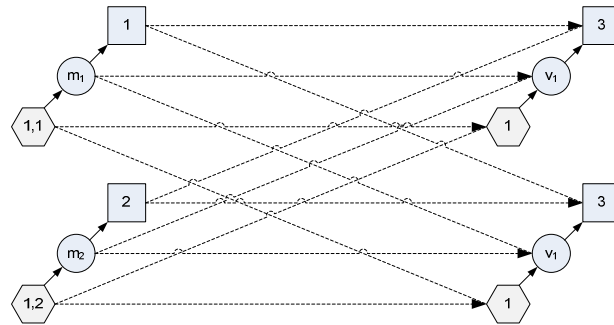


Figure 4.6: A diagram of a many-to-many transformation

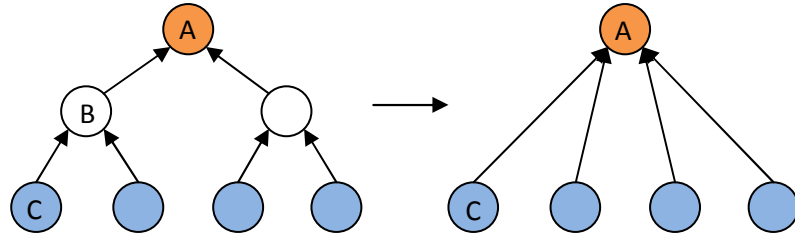
4.4.3 Contributing Relation Member Cardinality

For vectors, matrixes and arrays the relation is a bijective function, so there is exactly one index number for each element. When a matrix is transformed and we know that the one input element contributes to only one output element, we say the transformation is one-to-one. Here the index numbers are also transformed each into one new index number.

In a graph a node may be connected to multiple other nodes. A transformation may use multiple edges from the input to produce one new edge in the output. For graphs and trees it may be the case that one or more members of the relation from the input are used to produce members of the relation in the output.

EXAMPLE 4.17

Let the following tree be transformed into a new tree:



All the white nodes are deleted and the incoming edges and outgoing edges are replaced by one edge: $(C, B) + (B, A) \rightarrow (C, A)$.

In EXAMPLE 4.17 we see that two edges are needed to produce a new edge. Just as with the transformation of elements it is possible that more than one member of the relation is needed to produce a new relation.

DEFINITION 4.13 A transformation may use one or more members of the input relation to produce members of the output relation. This called the contributing relation member cardinality and can be *one-to-one*, *many-to-one*, *one-to-many* and *many-to-many*.

4.4.4 Reversibility

We call a transformation reversible if the original input can be recreated using the output. The transformation that performs this operation is called the inverse transformation and is denoted by T^{-1} .

DEFINITION 4.14 A transformation T is reversible if there exists a transformation T^{-1} that creates the original input from the output. The inverse transformation is defined by $T^{-1}: (S^{base'}, R', \lambda') \rightarrow (S^{base}, R, \lambda)$

Intuitively a transformation is reversible when there is no loss of information. But exactly what information is lost?

EXAMPLE 4.18

Let the matrix $\begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$ be transformed into the vector $\begin{bmatrix} 3 \\ 7 \end{bmatrix}$ by horizontally summing all the elements in a row. When we would like to produce the matrix again using the vector as the input, we do not know anymore how to split the value into two separate values. This information is lost.

EXAMPLE 4.18 shows that due to loss of information it may become impossible to regenerate the original input from the output. It is difficult to formulate exactly what information needs to be preserved for a transformation to be reversible. Starting from simple transformations moving to more complex transformations we show under what conditions certain transformations may be reversible.

To describe reversible transformations we look to the concept of invertible functions from the field of mathematics. A function $f: A \rightarrow B$ is invertible if and only if it is one-to-one and onto [Gri98, p.255]. One-to-one means that each element of B may be mapped to at most one element from A [Gri98, p.225]. Onto means that all the elements of B must be mapped to at least one element of A [Gri98, p.230].

A mathematical function only maps elements to elements. From DEFINITION 4.7 we know that the transformation of data structures involves more than mapping input elements to output elements. Members of the relations and values may all be involved in the transformation and therefore it is not enough to consider the contributing elements to describe a reversible transformation.

Although it is difficult to come with a definition of reversible transformation that covers all possible transformations, it is possible to describe this for specific types of transformations. It is easier to define reversibility for a transformation that does not create new elements, than for a transformation that does.

When a transformation does not create new elements, we only need to look at T_S and T_D . When these two effects are independent, the transformation T is reversible if T_D and T_S are reversible. When a transformation has a data-by-data and structure-by-structure effect, we only need to show that we can create the original structure from the output structure and the original data from the output data. This is the case when contributing element cardinality and the contributing relation member cardinality are both one-to-one. EXAMPLE 4.19 shows a reversible transformation that only has a structure-by-structure effect and EXAMPLE 4.20 shows a reversible transformation that only has a data-by-data effect.

EXAMPLE 4.19

The elements in the vector V are all shifted one place to the right, where the last item is moved to the front: $V = [1 \ 2 \ 3] \rightarrow V' = [3 \ 1 \ 2]$. The only thing that changes here is the position of the elements. The elements are not changed and neither are their values. Because the contributing relation member cardinality is one-to-one, the transformation is reversible.

EXAMPLE 4.20

The values in the vector V are all incremented by one: $V = [1 \ 2 \ 3] \rightarrow V' = [2 \ 3 \ 4]$. The original values can be obtained by decreasing all the values by one.

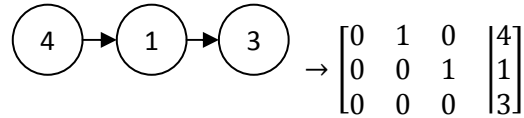
Even when T_D and T_S are not independent it may be possible that the whole transformation T is reversible. EXAMPLE 4.21 shows a transformation that has co-dependent effects. Still this transformation is reversible. The data part from the input is still encoded as data in the output. The structure of the input is not transformed into new structure, but is encoded as values in the output. All the necessary information is preserved and the transformation is reversible.

EXAMPLE 4.21

Let G be a graph where

$$G = (\{a, b, c\}, \{(a, b), (b, c)\}, \{(a, 4), (b, 1), (c, 3)\})$$

This graph can be transformed into an adjacency matrix augmented with a list of nodes:



The output matrix is defined as follows:

$$M = \left(\begin{array}{l} \{m_1, m_2, m_3, a\}, \\ \{m_4, m_5, m_6, b\}, \\ \{m_7, m_8, m_9, c\} \\ \left\{ \begin{array}{l} (1,1, m_1), (1,2, m_2), (1,3, m_3), \\ (2,1, m_4), (2,2, m_5), (2,3, m_6), \\ (3,1, m_{11}), (3,2, m_{12}), (3,3, m_{13}), \\ (1,4, a), (2,4, b), (3,4, c) \end{array} \right\}, \\ \left\{ \begin{array}{l} (m_1, 0), (m_2, 1), (m_3, 0), \\ (m_4, 0), (m_5, 0), (m_6, 1), \\ (m_7, 0), (m_8, 0), (m_9, 0), \\ (a, 4), (b, 1), (c, 3) \end{array} \right\} \end{array} \right)$$

We can see clearly that all the elements and values from the graph are stored in the same way in the matrix. The difference is how the relations are stored. Still there is a one-to-one mapping: for each edge in the graph there is a 1 in the matrix.

In this case it also becomes clear why reversible transformations are different from invertible functions. The matrix contains much more elements and relations than the graph. There is no mapping from these extra elements to the original input.

4.4.5 Invariance

A data structure has three parts and a transformation can affect all three parts (DEFINITION 4.5). A transformation does not always affect all three parts. It can be the case that only the structure is changed but the data remains intact. The opposite is also possible. When either the data or the structure remains unchanged after a transformation we call it invariant over the transformation. This can only happen when the structure and the data sub transformations are independent and no elements are created or deleted.

DEFINITION 4.15 A transformation where the structure part is invariant over a transformation T is called a data-only transformation and is denoted by $T = T_D^D$. Only the value function is changed by the transformation.

EXAMPLE 4.22

Let the following vector be transformed by multiplying the values of all the elements by 2: $[2 \ 3] \rightarrow [3 \ 4]$. Here the structure part is invariant and T_D^D performs the complete transformation, so $T = T_D^D$.

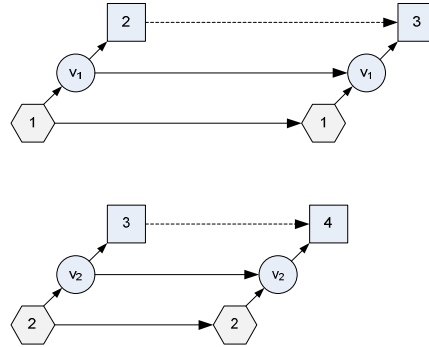


Figure 4.7: A data-only transformation

DEFINITION 4.16 A transformation where the data part is invariant over a transformation T is called a structure-only transformation and is denoted by $T = T_S^S$. Only the relation is changed by the transformation.

EXAMPLE 4.23

Let the elements of a vector be mirrored: $[1 \ 2] \rightarrow [2 \ 1]$. The elements and the values do not change. Only the relation changes, so it is a structure-only transformation.

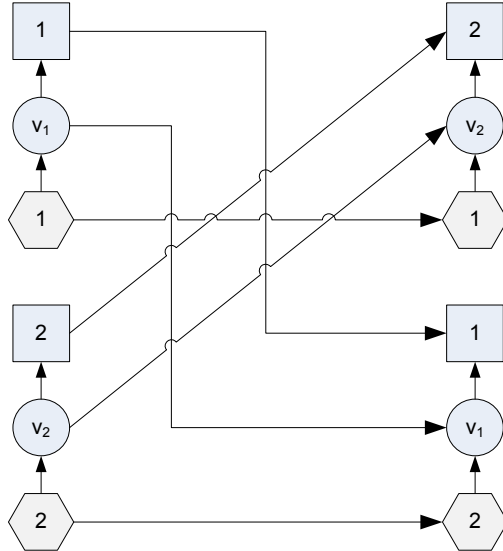


Figure 4.8: A structure-only transformation

4.4.6 Data/Structure contribution coherence

In the matrix-to-vector transformation of EXAMPLE 4.2 the output elements and their index numbers are created using contributing elements from the input on the basis of their position: for each row of elements in the matrix, an element in the vector is created. The values of the output elements are based on the same elements that were used to create the structure. This is not always the case as is shown in

EXAMPLE 4.24

The value of the middle element in the vector $V = [1 \ _ \ 3]$ is found by interpolating the two outer values to produce the output vector $V = [1 \ 2 \ 3]$. Here each input element contributes to the structure of one element in the output, but the two outer elements contribute to the value of the middle element in the output.

DEFINITION 4.17

When other elements from the input are used for the value of an output element than for the structure of that element, the *data/structure contribution is incoherent*.

Figure 4.9 shows the transformation diagram of EXAMPLE 4.24. From the structure point of view, v_2 is the contributing element of v'_2 , but from the data point of view, v_1 and v_2 are the contributing elements.

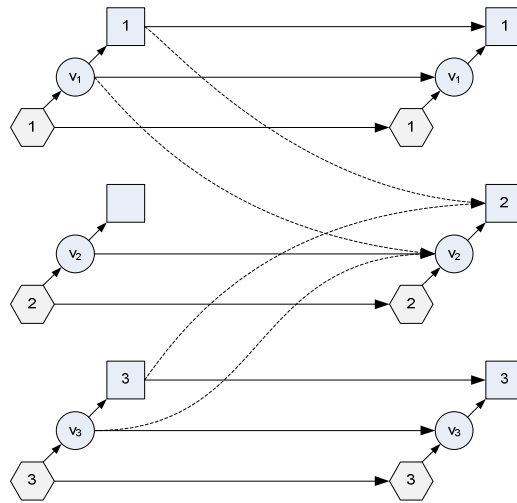


Figure 4.9: A data/structure incoherent transformation

4.5 Structure by structure transformation

In this part of the chapter we describe how for certain types of transformations the input elements that contribute to an output element can be found using a function that describes the transformation of the position function. We describe this for a special class of transformations called structure-by-structure transformations.

DEFINITION 4.18 A *structure-by-structure transformation* is a transformation where the output structure depends only on the input structure (DEFINITION 4.7). It does not matter how the output data is created. Thus a transformation T is a structure-by-structure transformation if $T_S^S \in T$.

In subchapter 4.4.2 we specified that the cardinality of contributing elements could be one-to-one, one-to-many, many-to-one and many-to-many (DEFINITION 4.8). This means that each input element is related to one or more output elements and vice versa.

DEFINITION 4.19 The set of input elements that contribute to an output element is $S^{in}[e^{out}]$

DEFINITION 4.20 The set of output elements to which an input element contributes is $S^{out}[e^{in}]$

For vectors, matrixes and arrays it was possible to model the relations between the elements as a function (DEFINITIONS 3.9-I, 3.13-I and 3.17-I). This means that if we can show how the position of input and output elements relate to each other, we can show which input elements correspond to which output elements.

The method of describing the correspondence between input and output elements depends on the contributing element cardinality. For each of these four cases we will give a description of how the position formula can be used to describe this correspondence. Since vectors and matrixes form a sub class of arrays we use arrays as an example to describe a generic method.

4.5.1 One-to-one

When the correspondence between input elements and output elements is one-to-one we can define the position of an output element as function of the position of the contributing input element. The position of an element is given by an index according to DEFINITION 3.17-I. The position transformation function is a bijective function that maps each index number of the input onto an index number of the output.

DEFINITION 4.21 The position of an output element can be expressed as a transformation of the position of an input element:

$$T_P: \prod_{i=1}^d \mathbb{N}_{m_i} \rightarrow \prod_{j=1}^{d'} \mathbb{N}_{n_j}$$

where d is the number of dimensions of the input data structure; m_i is the boundary of the i^{th} dimension; $\mathbb{N}_{m_i} = \{1, 2, \dots, m_i\}$; d' is the dimension of the output data structure; n_j is the boundary of the j^{th} dimension and $\mathbb{N}_{n_j} = \{1, 2, \dots, n_j\}$. Because this is a one-to-one transformation the following equation must be true:

$$\prod_{i=1}^d m_i = \prod_{j=1}^{d'} n_j$$

The transformation of the position can be used to identify the element that is located at that position. This way the position-transformation can be used to find the output element to which the input element contributes.

DEFINITION 4.22 The set of output elements to which an input element contributes in a one-to-one transformation can be found using the position-transformation function:

$$S^{out}[e^{in}] = \{e^{out} \in S^{base'} \mid e^{out} = F'_A(T_P(F_A^{-1}(e^{in})))\}$$

where F'_A is the function that retrieves an element from the output at a given position, T_P is the position-transformation function and F_A^{-1} retrieves the position of a given element from the input.

This formula may seem rather complex, but step by step it performs the following operations:

- $F_A^{-1}(e^{in})$ - retrieve the position of the input element
- $T_P(F_A^{-1}(e^{in}))$ - use that position to find the position of the corresponding output element
- $F'_A(T_P(F_A^{-1}(e^{in})))$ - retrieve the element from the output that is located at the position calculated in the previous step.

EXAMPLE 4.25

Let the elements of a vector of size 3 be shifted one place to the right (with the last element moving to the front):

$$V \rightarrow V' = [1 \ 2 \ 3] \rightarrow [3 \ 1 \ 2]$$

The transformation of the structural part can be described by:

$$T_S^S: (\{v_1, v_2, v_3\}, \{(1, v_1), (2, v_2), (3, v_3)\}) \rightarrow (\{v_1, v_2, v_3\}, \{(1, v_3), (2, v_1), (3, v_2)\}).$$

Both the input and the output are of dimension 1 and size 3, so

$T_P: \mathbb{N}_3 \rightarrow \mathbb{N}_3$. Specifically for this transformation T_P is specified as

$T_P(i) = i + 1 \bmod 3$. The output element to which v_3 contributes is

$$S^{out}[v_3] = \left\{ F_V' \left(T_P(F_V^{-1}(v_3)) \right) \right\} = \{v_3\} \text{ because } F_V^{-1}(v_3) = 3 \text{ and}$$

$T_P(3) = 3 + 1 \bmod 3 = 1$ and $F_V'(1) = v_3$. This is right, because the elements are only rearranged so the input element is always the same as the output element that it corresponds with.

In some cases we do not want to know what output elements correspond to an input element, but we would like to know the reverse. Because the transformation is one-to-one T_P can be inverted.

DEFINITION 4.23

The position of an input element can be expressed as a transformation of the position of an output element which is the inverse of transformation T_P :

$$T_P^{-1}: \prod_{j=1}^{d'} \mathbb{N}_{n_j} \rightarrow \prod_{i=1}^d \mathbb{N}_{m_i}$$

where d is the number of dimensions of the input data structure; m_i is the boundary of the i^{th} dimension; $\mathbb{N}_{m_i} = \{1, 2, \dots, m_i\}$; d' is the dimension of the output data structure; n_j is the boundary of the j^{th} dimension and $\mathbb{N}_{n_j} = \{1, 2, \dots, n_j\}$.

In the same manner as DEFINITION 4.22 we can now give a formula to find the input elements that correspond to a certain output element:

DEFINITION 4.24

The set of input elements that contribute to an output element in a one-to-one transformation can be found using the inverse position transformation function:

$$S^{in}[e^{out}] = \left\{ e^{in} \in S^{base} \mid e^{in} = F_A \left(T_P^{-1} \left(F_A^{-1'}(e^{out}) \right) \right) \right\}$$

where F_A is the function that retrieves an element from the input at a given position, T_P^{-1} is the inverse position-transformation function and $F_A^{-1'}$ retrieves the position of a given element from the output.

EXAMPLE 4.26

Using the same case as from EXAMPLE 4.25 we now would like to know what input elements correspond to the element v_2 from the output:

$$S^{in}[v_2] = \left\{ F_V \left(T_P^{-1} \left(F_A^{-1'}(v_2) \right) \right) \right\} = \{v_2\} \text{ because } T_P^{-1}(i) = i - 1 \bmod 3; F_A^{-1'}(v_2) = 3; T_P^{-1}(3) = 3 - 1 \bmod 3 = 2; F_A(2) = v_2.$$

4.5.2 Many-to-one

In the many-to-one scenario, an output element has multiple contributing input elements. Still, each input element contributes to exactly one output element. The size of the input data structure is larger than the size of the output data structure, so the following equation must be true:

$$\prod_{i=1}^d m_i > \prod_{j=1}^{d'} n_j$$

where d is the number of dimensions of the input data structure; m_i is the boundary of the i^{th} dimension; d' is the dimension of the output data structure and n_j is the boundary of the j^{th} dimension.

The transformation function T_P for a many-to-one transformation is not bijective because more than one input element is mapped onto the same output element. We can still find the output element to which an input element contributes using DEFINITION 4.22. But we cannot find $S^{in}[e^{out}]$ using T_P^{-1} because T_P is not invertible. DEFINITION 4.25 shows that it is possible to find $S^{in}[e^{out}]$ using T_P instead. This method can also be used for one-to-one scenarios when T_P^{-1} is not specified.

DEFINITION 4.25 For a many-to-one transformation the input elements that correspond to an output element can be found using the following formula:

$$S^{in}[e^{out}] = \left\{ e^{in} \in S^{base} \mid T_P \left(F_A^{-1}(e^{in}) \right) = F_A^{-1'}(e^{out}) \right\}$$

In the one-to-one scenario the set of input elements that corresponds to an output element always contains only one member. In the following example we see that this now contains more than one member:

EXAMPLE 4.27

Let $M = \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix} = \left(\begin{array}{c} \{m_1, m_2, m_3, m_4\}, \\ \{(1, m_1), (2, m_2), (3, m_3), (4, m_4)\}, \\ \{(m_1, 2), (m_2, 4), (m_3, 4), (m_4, 2)\} \end{array} \right)$ be transformed into the vector

$V = \begin{bmatrix} 6 \\ 6 \end{bmatrix} = (\{v_1, v_2\}, \{(1, v_1), (2, v_2)\}, \{(v_1, 6), (v_2, 6)\})$ by summing all the elements in a row. For this transformation the input is of dimension 2 and the output of dimension 1, so $T_P: \mathbb{N}_2 \times \mathbb{N}_2 \rightarrow \mathbb{N}_2$.

Specifically for this transformation we have $T_p(i, j) = i$.

Now $S^{out}[m_2] = \{e^{out} \in S^{base} \mid e^{out} = F'_A(T_p(F_A^{-1}(m_2)))\} = \{v_1\}$

and $S^{in}[v_2] = \{e^{in} \in S^{base} \mid T_p(F_A^{-1}(e^{in})) = F_A^{-1}(v_2)\}$.

$F_A^{-1}(v_2) = 2$ but now we need to check for all the input elements

what $T_p(F_A^{-1}(e^{in}))$ is: $F_A^{-1}(m_1) = (1,1)$ and $T_p(1,1) = 1$;

$F_A^{-1}(m_2) = (1,2)$ and $T_p(1,2) = 1$; $F_A^{-1}(m_3) = (2,1)$ and

$T_p(2,1) = 2$; $F_A^{-1}(m_4) = (2,2)$ and $T_p(2,2) = 2$. From this we conclude that $S^{in}[v_2] = \{m_3, m_4\}$.

4.5.3 One-to-many

The one-to-many scenario is the reverse case of the many-to-one scenario. The size of the input data structure is smaller than the size of the output data structure, so the following equation must be true:

$$\prod_{i=1}^d m_i < \prod_{j=1}^{d'} n_j$$

where d is the number of dimensions of the input data structure; m_i is the boundary of the i^{th} dimension; d' is the dimension of the output data structure and n_j is the boundary of the j^{th} dimension.

In this case it is only possible to define T_p^{-1} and not T_p . Both $S^{in}[e^{out}]$ and $S^{out}[e^{in}]$ must in this scenario be based on T_p^{-1} . To find $S^{in}[e^{out}]$ we use DEFINITION 4.24, but to find $S^{out}[e^{in}]$, we define a new function, similar to DEFINITION 4.25. This method can also be used in a one-to-one scenario where T_p cannot be formulated.

DEFINITION 4.26 For a one-to-many transformation the set of output elements to which an input element contributes can be found using the following formula:

$$S^{out}[e^{in}] = \{e^{out} \in S^{base'} \mid F_A^{-1}(e^{in}) = T_p^{-1}(F_A^{-1}(e^{out}))\}$$

4.5.4 Many-to-many

In the many-to-many scenario neither T_p nor T_p^{-1} can be defined and our previous solutions for finding $S^{in}[e^{out}]$ and $S^{out}[e^{in}]$ do not work. With one-to-many and many-to-one there was at least one direction (from input to output or vice versa) in which there was one element mapped to another and we could specify T_p or T_p^{-1} as a function of one position to another. EXAMPLE 4.28 shows that this is not possible for a many-to-many transformation.

EXAMPLE 4.28

Let $V = [1,3,5,3,1]$ be a vector of size 5 and the transformation T perform a blur on the vector. The blur is performed by having a window of size 3 slide over the vector. The position of the window is measured from its middle. So if we put the window in position three over the vector V , the elements in the window are $[3,5,3]$. If the window is at position 1, the window consists only of $[1,3]$. The window moves from position 1 to 5 and the output in the vector at that position is created by taking the average of the window: $[1|3|5|3|1] \rightarrow [2|3|3\frac{2}{3}|3|2]$. In this case the 3 at position 2 is used to generate the numbers at position 1,2 and 3. The number at position 2 in the output in return is again also linked to multiple elements from the input: 1,3 and 5 at respectively position 1,2 and 3. This make T a many-to-many transformation.

In a many-to-many scenario, a function that transforms a position must have multiple positions as its output. This can be done by creating a function that transforms one position into a set of positions.

DEFINITION 4.27

The function that transforms the position of an input element into a set of positions of the output elements to which it contributes is:

$$T_{P++}: \prod_{i=1}^d \mathbb{N}_{m_i} \rightarrow P\left(\prod_{j=1}^{d'} \mathbb{N}_{n_j}\right)$$

where d is the number of dimensions of the input data structure; m_i is the boundary of the i^{th} dimension; $\mathbb{N}_{m_i} = \{1,2, \dots, m_i\}$; d' is the dimension of the output data structure; n_j is the boundary of the j^{th} dimension and $\mathbb{N}_{n_j} = \{1,2, \dots, n_j\}$. We use ++ to indicate that the result is a set of positions.

EXAMPLE 4.29

The position for EXAMPLE 4.28 can be specified using DEFINITION 4.27: $T_{P++}: \mathbb{N}_5 \rightarrow P(\mathbb{N}_5)$ where $T_{P++}(i) = \{\max(1, i-1), i, \min(5, i+1)\}$. In the case of position 3 this results in:
 $T_{P++}(3) = \{\max(1,2), 3, \min(5,4)\} = \{2,3,4\}$. For position 5:
 $T_{P++}(5) = \{\max(1,4), 5, \min(5,6)\} = \{4,5\}$.

Now that we have a function that gives a set of the positions of all the output elements to which an input element contributes we specify a new method of finding the set of output elements for a given input element.

DEFINITION 4.28 The set of output elements to which a given input element contributes in a many-to-many transformation is given by:

$$S^{out}[e^{in}] = \{e^{out} \in S^{base'} \mid F_A^{-1'}(e^{out}) \in T_{P++}(F_A^{-1}(e^{in}))\}$$

where $F_A^{-1}(e^{in})$ is the position of the input element and $T_{P++}(\dots)$ the set of positions that correspond to the position of the input element. $F_A^{-1'}(e^{out})$ is the position of the output element, and for all output elements it is verified if its position is part of the set of positions that correspond to the position of the input element.

EXAMPLE 4.30

Using DEFINITION 4.28 we can now find the set of output elements that correspond to the element v_4 :

$$S^{out}[v_4] = \{e^{out} \in S^{base'} \mid F_A^{-1'}(e^{out}) \in T_{P++}(F_A^{-1}(v_4))\}$$

First we need to find $F_A^{-1}(v_4) = 4$. Now $T_{P++}(4) = \{3, 4, 5\}$. Now we need to compare the positions of all the input elements:

$$F_A^{-1'}(v_1) = 1; F_A^{-1'}(v_2) = 2; F_A^{-1'}(v_3) = 3; F_A^{-1'}(v_4) = 4;$$

$F_A^{-1'}(v_5) = 5$. Only the output positions of v_3 , v_4 and v_5 correspond to the input position of v_4 so $S^{out}[v_4] = \{v_3, v_4, v_5\}$.

To find out what input elements are related to an output element we need a transformation function similar to T_{P++} . We call this function T_{P++}^{-1} where we use $^{-1}$ to indicate that this function gives the input positions for an output position even though it is not a real inverse of the function T_{P++} .

DEFINITION 4.29 The function that transforms the position of an output element into a set of positions of the contributing input elements is:

$$T_{P++}^{-1} : \prod_{j=1}^{d'} \mathbb{N}_{n_j} \rightarrow P\left(\prod_{i=1}^d \mathbb{N}_{m_i}\right)$$

where d is the number of dimensions of the input data structure; m_i is the boundary of the i^{th} dimension; $\mathbb{N}_{m_i} = \{1, 2, \dots, m_i\}$; d' is the dimension of the output data structure; n_j is the boundary of the j^{th} dimension and $\mathbb{N}_{n_j} = \{1, 2, \dots, n_j\}$. We use $++$ to indicate that the result is a set of positions.

The input elements that correspond to a specific output element can now be found using the following formula:

DEFINITION 4.30 The set of input elements that contribute to a given output element in a many-to-many transformation is given by:

$$S^{in}[e^{out}] = \{e^{in} \in S^{base} \mid F_A^{-1}(e^{in}) \in T_{P++}^{-1}(F_A^{-1'}(e^{out}))\}$$

When in a one-to-one scenario either T_P or T_P^{-1} cannot be formulated, the formulas of the one-to-many and many-to-one scenarios can be used instead. In a many-to-many scenario it may also be possible that only T_{P++} or T_{P++}^{-1} can be formulated. In these cases $S^{in}[e^{out}]$ can also be found using T_{P++} and $S^{out}[e^{in}]$ using T_{P++}^{-1} .

DEFINITION 4.31 The set of output elements to which a given input element contributes in a many-to-many transformation can be found using T_{P++}^{-1} :

$$S^{out}[e^{in}] = \{e^{out} \in S^{base'} \mid F_A^{-1}(e^{in}) \in T_{P++}^{-1}(F_A^{-1'}(e^{out}))\}$$

DEFINITION 4.32 The set of input elements that contribute to a given output element in a many-to-many transformation can be found using T_{P++} :

$$S^{in}[e^{out}] = \{e^{in} \in S^{base} \mid F_A^{-1'}(e^{out}) \in T_{P++}(F_A^{-1}(e^{in}))\}$$

4.6 Multiple Input Sources

Until now we have considered only unary transformations: one data structure is transformed into another data structure. It is very well possible that more than one input data structure used in a transformation; these are called binary transformations if there are two input data structures, or n -ary transformations if there are more.

In this subchapter we show how the definitions from subchapter 0 can be modified to be used in an n -ary transformation. First we introduce a new definition to identify an input data structure using an index, which we will use in an example of an n -ary transformation.

DEFINITION 4.33 An n -ary transformation transforms n input data structures into an output data structure. Each of these input data structures is numbered using the index $[k]$, where $1 \leq k \leq n$. An element of a numbered input data structure is marked with k as a superscript: e_i^k where i is the number that identifies a unique element (DEFINITION 3.1) within the input data structure with number k . Specific parts of an input data structure can be referred to using the index k . $S^{base}[k]$ is the base set of the input data structure with number k .

EXAMPLE 4.31

We have three vectors:

$$V[1] = [v_1^1 \quad v_2^1 \quad v_3^1]; V[2] = [v_1^2 \quad v_2^2 \quad v_3^2]; V[3] \\ = [v_1^3 \quad v_2^3 \quad v_3^3]$$

We can create a matrix using the three vectors as an input:

$$M = \begin{bmatrix} v_1^1 & v_1^2 & v_1^3 \\ v_2^1 & v_2^2 & v_2^3 \\ v_3^1 & v_3^2 & v_3^3 \end{bmatrix}$$

Here $S^{base}[1] = \{v_1^1, v_2^1, v_3^1\}$ and $F_V[2] = \{(1, v_1^2), (2, v_2^2), (3, v_3^2)\}$. The position transformation function is: $T_P[k](i) = (k, i)$ where i is the position of an element in one of the input vectors. There is only one output data structure, so we don't need k in the inverse position transformation function: $T_P^{-1}(i, j) = j$.

Just as with unary transformations, we consider different scenarios accord to the contributing element cardinality: one-to-one, many-to-one, one-to-many and many-to-many. It must be stressed that we do not refer here to the cardinality of the input sources. The transformation in EXAMPLE 4.31 is one-to-one because each element of the different input data structures contributes only to one element in the output and for each output element there is only one contributing element from any of the input data structures.

In the one-to-one and one-to-many scenarios the index number of the input data structure that contains the contributing element may be directly obtained from the output element, because there is only one contributing input element, and hence there is only one contributing input source.

DEFINITION 4.34 In one-to-one and one-to-many scenarios the index number of the input data structure that contains the contributing input element can be specified as a function of the output element to which that input element contributes. This function is denoted as:

$$in_{\#}(e^{out})$$

4.6.1 One-to-one

The most common one-to-one transformation is a join of multiple data structures to create a new data structure. There are no elements created or deleted, because the elements are only arranged in a new data structure. The elements also do not get a new value, and therefore a join is a structure-only transformation. In EXAMPLE 4.31 a join transformation is shown. To find $s^{in}[e^{out}]$ and $S^{out}[e^{out}]$ we need to extend DEFINITION 4.22 and DEFINITION 4.24.

DEFINITION 4.35 The set of output elements to which an input element contributes in a n -ary one-to-one transformation is given by:

$$\begin{aligned} S^{out}[e^{in}] &= \{e^{out} \in S^{base'} \mid (\exists k \cdot e^{in} \in S^{base}[k]) \wedge e^{out} = F'_A \left(T_P[k] \left(F_A[k](e^{in}) \right) \right) \} \end{aligned}$$

DEFINITION 4.36 The set of input elements that contribute to an output element in an n -ary one-to-one transformation is given by:

$$\begin{aligned} S^{in}[e^{out}] &= \{e^{in} \in S^{base}[k] \mid k = in_{\#}(e^{out}) \wedge e^{in} = F_A[k] \left(T_P^{-1} \left(F_A^{-1'}(e^{out}) \right) \right) \} \end{aligned}$$

4.6.2 Many-to-one

Similar to the unary transformation, we can only define T_P and not T_P^{-1} . To find $S^{out}[e^{in}]$ in an n -ary many-to-one transformation we use DEFINITION 4.35 and to find $S^{in}[e^{out}]$ we extend DEFINITION 4.25.

DEFINITION 4.37 The set of input elements that contribute to an output element in an n -ary many-to-one transformation is given by:

$$\begin{aligned} S^{in}[e^{out}] &= \{e^{in} \in S^{base}[k] \mid 1 \leq k \leq n \wedge T_P[k] \left(F_A^{-1}[k](e^{in}) \right) = F_A^{-1'}(e^{out}) \} \end{aligned}$$

EXAMPLE 4.32

A binary transformation combines two 2x2 matrixes by adding the elements at the same position to produce another 2x2 matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

The three data structures are described by:

$$\begin{aligned} M[1] &= \left(\{m_1^1, m_2^1, m_3^1, m_4^1\}, \right. \\ &\quad \{(1,1, m_1^1), (1,2, m_2^1), (2,1, m_3^1), (2,2, m_4^1)\}, \\ &\quad \{(m_1^1, 1), (m_2^1, 2), (m_3^1, 3), (m_4^1, 4)\} \left. \right) \\ M[2] &= \left(\{m_1^2, m_2^2, m_3^2, m_4^2\}, \right. \\ &\quad \{(1,1, m_1^2), (1,2, m_2^2), (2,1, m_3^2), (2,2, m_4^2)\}, \\ &\quad \{(m_1^2, 5), (m_2^2, 6), (m_3^2, 7), (m_4^2, 8)\} \left. \right) \end{aligned}$$

$$M' = \left(\begin{array}{l} \{m'_1, m'_2, m'_3, m'_4\}, \\ \{(1,1, m'_1), (1,2, m'_2), (2,1, m'_3), (2,2, m'_4)\}, \\ \{(m'_1, 6), (m'_2, 8), (m'_3, 10), (m'_4, 12)\} \end{array} \right)$$

We can find the contributing input elements using:

$$T_P^{-1}(i, j) = (i, j)$$

So the contributing input elements of m'_2 are:

$$S^{in}[e^{out}] = \{m_2^1, m_2^2\}$$

4.6.3 One-to-many

Similar to the unary transformation, for a one-to-many n -ary transformation we can only define T_P^{-1} and not T_P . To find $S^{in}[e^{out}]$ in an n -ary many-to-one transformation we use DEFINITION 4.36 and to find $S^{out}[e^{in}]$ we extend DEFINITION 4.26.

DEFINITION 4.38 The set of output elements to which an input element contributes in an n -ary one-to-many transformation is given by:

$$\begin{aligned} S^{out}[e^{in}] \\ = \{e^{out} \in S^{base'} \mid (\exists k \cdot e^{in} \in S^{base}[k]) \wedge F_A^{-1}[k](e^{in}) = T_P^{-1}(F_A^{-1'}(e^{out}))\} \end{aligned}$$

4.6.4 Many-to-many

It is not possible to define either T_P or T_P^{-1} for an n -ary many-to-many transformation, similarly to the unary many-to-many transformation. For these types of transformations $T_{P++}[k]$ and T_{P++}^{-1} need to be defined according to DEFINITION 4.27 and DEFINITION 4.29. We extend DEFINITION 4.28, DEFINITION 4.30, DEFINITION 4.31 and DEFINITION 4.32 to find $S^{out}[e^{in}]$ and $S^{in}[e^{out}]$ for a many-to-many n -ary transformation.

DEFINITION 4.39 The set of output elements to which an input element contributes in an n -ary many-to-many transformation is given by:

$$\begin{aligned} S^{out}[e^{in}] \\ = \{e^{out} \in S^{base'} \mid 1 \leq k \leq n \wedge F_A^{-1'}(e^{out}) \in T_{P++}[k](F_A^{-1}[k](e^{in}))\} \end{aligned}$$

DEFINITION 4.40 The set of input elements that contribute to an output element in an n -ary many-to-many transformation is given by:

$$\begin{aligned} S^{in}[e^{out}] \\ = \{e^{in} \in S^{base}[k] \mid 1 \leq k \leq n \wedge F_A^{-1}[k](e^{in}) \in T_{P++}^{-1}(F_A^{-1'}(e^{out}))\} \end{aligned}$$

DEFINITION 4.41 The set of output elements to which a given input element contributes in an n -ary many-to-many transformation, when $T_{P++}[k]$ cannot be defined, is given by:

$$S^{out}[e^{in}] = \{e^{out} \in S^{base'} \mid 1 \leq k \leq n \wedge F_A^{-1}[k](e^{in}) \in T_{P++}^{-1}(F_A^{-1'}(e^{out}))\}$$

DEFINITION 4.42 The set of input elements that contribute to a given output element in a n -ary many-to-many transformation, when T_{P++}^{-1} cannot be defined, is given by:

$$S^{in}[e^{out}] = \{e^{in} \in S^{base}[k] \mid 1 \leq k \leq n \wedge F_A^{-1'}(e^{out}) \in T_{P++}[k](F_A^{-1}[k](e^{in}))\}$$

4.7 Summary

In this chapter we showed how the transformation of one data structure into another can be represented by sub transformations that produce the new data part and that produce the new structure part.

We have formulated five properties of which the first describes how the sub transformations can take only the data part, the structure part, both or neither as their input. The second property describes how over a transformation one or more elements from the input may be related to elements from the output: they may be one-to-one, one-to-many, many-to-one and many-to-many. The third property describes this idea for relations, which also may be one-to-one, etc. Reversibility is the fourth property and it describes under which circumstances a transformation may be reversed. A transformation may be reversed if the original input can be recreated using the output. The last property is invariance. In some transformations, the data part may not change while in other the structure part may not change. The part that does not change is said to be invariant.

Elementary operations are the smallest possible transformations of a data structure. It may be possible to define different variants of elementary operations, but the set of elementary operations must be chosen in such a way that any transformation of a data structure can be modeled as a series of these elementary operations. A transformation that is build using these elementary operations is called a elementary transformation.

When only the sub transformation that creates the structure part of the output only depends on the structure part of the input it is called a structure-by-structure transformation. For these types of transformations, the transformation of the structure part can be described without having knowledge of the transformation of the data part. For structure-to-structure transformations that only use vectors, matrixes and arrays as input and output it is possible to describe the relation between input and output elements using a mathematical formula. For each of the four cases - one-to-one, many-to-one, one-to-many and many-to-many – a different method is described to identify the input elements that correspond to an output element and vice versa.

Chapter 5

Locality

In this chapter we discuss the concept of locality. Locality is property that can be used when the effects of a transformation cannot fully be described using a position transformation function as described in chapter 4. We define locality as a property of the combination of two elements in a data structure. Locality is a concept based on the distance between two elements. Before giving a formal definition we will look at how distance between elements of a data structure can be measured. Using the definition of distance and the various methods of measuring the distance between two elements, we will give a definition of locality. We then show for each of the elementary data structures how locality can be applied.

Transformation of a data structure may also have an effect on the locality of the elements. In the last part of this chapter we discuss the effect of the transformation in four scenarios.

The research question that we answer in this chapter is:

RESEARCH QUESTION 5

What is locality and how is it affected by a transformation?

5.1 Distance

Locality is a concept based on distance, so before giving a formal definition of locality we need to give a definition of distance. From the Encyclopedia of Distances [DD09, p.03-04] we get the following definition:

A function $d: X \times X \rightarrow \mathbb{R}$ is called a distance function if the following properties hold for all $x, y \in X$:

1. $d(x, y) \geq 0$ (*non-negativity*).
2. $d(x, y) = d(y, x)$ (*symmetry*).
3. $d(x, y) = 0$ if and only if $x = y$ (*identity of indiscernibles*).
4. $d(x, y) \leq d(x, z) + d(z, y)$ (*triangle inequality*).

There are many different possible functions on a set X that have the above four properties. For our definition of locality it does not matter what specific function is used. In the next part of this subchapter we will describe four different distance function. The Euclidean distance is the most common distance. We describe the Manhattan and Chebyshev distance because these are frequently used in digital applications. These three distance functions have no application in graphs and trees. We describe the path distance to show how distances can be measured in these types of data structures.

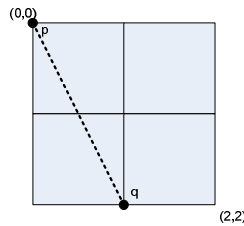
5.1.1 Euclidean distance

The Euclidean distance measures the distances between two points over a straight line. These points are Cartesian coordinates: the position of point p can be written as (p_1, \dots, p_n) where n is the number of dimensions. We use the following definition of Euclidean distance [DD09, p.94]:

$$d_E(x, y) = \sqrt{\prod_{i=1}^n (x_i - y_i)^2}$$

EXAMPLE 5.1

We have a grid with points p and q at coordinates $(0,0)$ and $(1,2)$ respectively:



The distance between p and q is given by:

$$d_E(p, q) = \sqrt{(0 - 1)^2 + (0 - 2)^2} = \sqrt{5}$$

When we are concerned with data structures we are not concerned with distance between Cartesian coordinates, but with distance between elements. Matrixes have a position function that maps index numbers to elements. Matrixes look like two-dimensional planes but the index numbers are constrained to whole numbers. Using the inverse position function of a matrix (DEFINITION 3.13-II) we can find the index number of an element. We use the general definition to specify a specific Euclidean distance for matrixes:

DEFINITION 5.1 The Euclidean distance between elements of a matrix is defined as

$$d_E^M(e_1, e_2) = d_E(F_M^{-1}(e_1), F_M^{-1}(e_2))$$

$$= \sqrt{(F_M^{-1}(e_1)[1] - F_M^{-1}(e_2)[1])^2 + (F_M^{-1}(e_1)[2] - F_M^{-1}(e_2)[2])^2}$$

where $F_M^{-1}(e)[i]$ denotes the i^{th} coordinate.

EXAMPLE 5.2

In the matrix below the distance for the middle element to all the other elements is calculated using the Euclidean distance:

4,2	3,6	3,2	3	3,2	3,6	4,2
3,6	2,8	2,2	2	2,2	2,8	3,6
3,2	2,2	1,4	1	1,4	2,2	3,2
3	2	1	0	1	2	3
3,2	2,2	1,4	1	1,4	2,2	3,2
3,6	2,8	2,2	2	2,2	2,8	3,6
4,2	3,6	3,2	3	3,2	3,6	4,2

Just as for the matrix we can define Euclidean distance for vectors and arrays:

DEFINITION 5.2 The Euclidean distance between elements of a vector is defined as:

$$d_E^V(e_1, e_2) = d_E(F_V^{-1}(e_1), F_V^{-1}(e_2)) = \sqrt{(F_V^{-1}(e_1) - F_V^{-1}(e_2))^2}$$

$$= |F_V^{-1}(e_1) - F_V^{-1}(e_2)|$$

Since the position for a vector returns only one index number we don't need to consider individual coordinates as with the matrix.

DEFINITION 5.3 The Euclidean distance between elements of an n -dimensional array is defined as:

$$d_E^A(e_1, e_2) = d_E(F_A^{-1}(e_1), F_A^{-1}(e_2))$$

$$= \sqrt{\sum_{i=1}^n (F_M^{-1}(e_1)[i] - F_M^{-1}(e_2)[i])^2}$$

where $F_M^{-1}(e)[i]$ denotes the i^{th} coordinate.

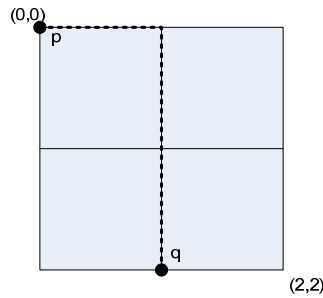
5.1.2 Manhattan distance

The Euclidean distance function is the most natural distance on a real plane \mathbb{R}^2 [DD09, p.323]. There are many more distance functions possible on the real plane, some of which may be restricted to digital spaces (\mathbb{Z}^n) [D09, p.332], which are called digital distance functions. Vector, matrix and array data structures are essentially digital spaces, so we can use the digital distance functions. One example is the Manhattan distance, also called the Taxi-cab distance. It has that name because the streets of the city of Manhattan form a grid. The taxi-cab problem deals with the question what the distance is to take a taxi from one point on that grid to another. The Manhattan distance as a digital distance function is defined as [DD09, p.333]:

$$d_M(x, y) = \sum_{i=1}^n |x_i - y_i|$$

EXAMPLE 5.3

We return to EXAMPLE 5.1 where we have the following grid:



The Euclidean distance was $\sqrt{5}$. The Manhattan distance between point p and q is:

$$d_M(p, q) = |0 - 1| + |0 - 2| = 3$$

We use the general definition of the Manhattan distance to definitions for the Manhattan distance in a vector, matrix and array:

DEFINITION 5.4 The Manhattan distance between elements of a vector is equal to the Euclidean distance and is defined by:

$$d_M^V(e_1, e_2) = d_E^V(e_1, e_2) = |F_V^{-1}(e_1) - F_V^{-1}(e_2)|$$

DEFINITION 5.5 The Manhattan distance between elements of a matrix is defined by:

$$d_M^M(e_1, e_2) = |F_M^{-1}(e_1)[1] - F_M^{-1}(e_2)[1]| + |F_M^{-1}(e_1)[2] - F_M^{-1}(e_2)[2]|$$

where $F_M^{-1}(e)[i]$ denotes the i^{th} coordinate.

DEFINITION 5.6 The Manhattan distance between elements of an array of dimension n is defined by:

$$d_M^A(e_1, e_2) = \sum_{i=1}^n |F_M^{-1}(e_1)[i] - F_M^{-1}(e_2)[i]|$$

where $F_M^{-1}(e)[i]$ denotes the i^{th} coordinate.

EXAMPLE 5.4

In the matrix below the distance for the middle element to all the other elements is calculated using the Manhattan distance:

6	5	4	3	4	5	6
5	4	3	2	3	4	5
4	3	2	1	2	3	4
3	2	1	0	1	2	3
4	3	2	1	2	3	4
5	4	3	2	3	4	5
6	5	4	3	4	5	6

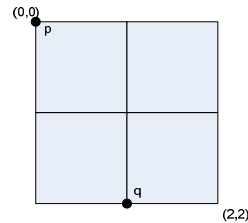
5.1.3 Chebyshev distance

In this part we give a second example of a digital distance function: the Chebyshev distance. The Chebyshev distance as a digital distance function is defined as [DD09, p.333]:

$$d_C(x, y) = \max \left(\bigcup_{i=1}^n \{|x_i - y_i|\} \right)$$

EXAMPLE 5.5

We return to EXAMPLE 5.1 where we have the following grid:



The Euclidean distance was $\sqrt{5}$. The Chebyshev distance between point p and q is:

$$d_C(p, q) = \max \{|0 - 1|, |0 - 2|\} = 2$$

We use the general definition of the Chebyshev distance to give definitions for the Chebyshev distance in a vector, matrix and array:

DEFINITION 5.7 The Chebyshev distance between two elements in a vector is equal to the Euclidean distance and is defined by:

$$d_C^V(e_1, e_2) = d_E^V(e_1, e_2) = \max(\{|F_V^{-1}(e_1) - F_V^{-1}(e_2)|\}) \\ = |F_V^{-1}(e_1) - F_V^{-1}(e_2)|$$

DEFINITION 5.8 The Chebyshev distance between two elements in a matrix is defined by:

$$d_C^M(e_1, e_2) \\ = \max(\{|F_V^{-1}(e_1)[1] - F_V^{-1}(e_2)[1]|, |F_V^{-1}(e_1)[2] - F_V^{-1}(e_2)[2]|\})$$

where $F_M^{-1}(e)[i]$ denotes the i^{th} coordinate.

DEFINITION 5.9 The Chebyshev distance between two elements in an array of dimension n is defined by:

$$d_C^A(e_1, e_2) = \max\left(\bigcup_{i=1}^n \{|F_V^{-1}(e_1)[i] - F_V^{-1}(e_2)[i]|\}\right)$$

where $F_M^{-1}(e)[i]$ denotes the i^{th} coordinate.

EXAMPLE 5.6

In the matrix below the distance for the middle element to all the other elements is calculated using the Chebyshev distance:

3	3	3	3	3	3	3
3	2	2	2	2	2	3
3	2	1	1	1	2	3
3	2	1	0	1	2	3
3	2	1	1	1	2	3
3	2	2	2	2	2	3
3	3	3	3	3	3	3

5.1.4 Path distance

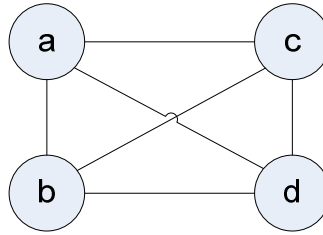
The three distance functions presented above can only be applied to the vector, matrix and array data structures. In graphs and trees distance is measured in a different way. According to Definition 3.21 in a graph data structure the elements are called nodes and the relation on the nodes consists of edges between two nodes. The distance between two nodes is measured as the length of the shortest path between two nodes and is denoted d_{path} . A path between u and v is a sequence of edges

$w_0w_1, w_1w_2, \dots, w_{n-1}w_n$ with $u = w_0$ and $v = w_n$ such that $w_i \neq w_j$ for $i \neq j$, $i, j \in \{0, 1, \dots, n\}$. [DD09, p.257]

From this definition we know that a series of nodes is only a path, if no node occurs more than once.

EXAMPLE 5.7

In the following graph ad, db, ba, ac is not a path, but ad, dc is.

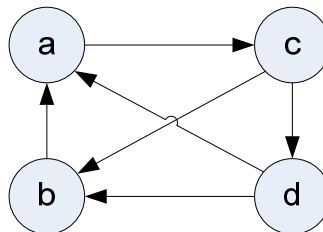


There are five possible paths, from a to d : ac, cd with length 2; ac, cb, bd with length 3; ab, bd with length 2; ab, bc, cd with length 3 and finally ad with length 1. The shortest of these five paths is ad , so $d_{path}(a, d) = 1$.

The graph in EXAMPLE 5.7 is undirected, so a path from u to v is the same as the path from v to u , hence $d_{path}(a, d) = d_{path}(d, a)$. In a directed graph this is not the case and the symmetry rule (rule #2) of the definition of a distance is violated. When used for a directed graph, d_{path} does not represent a distance function according to the official definition.

EXAMPLE 5.8

The following graph is a directed version of the graph in EXAMPLE 5.7. Some paths from that example are not a path in this digraph: ad ; ac, cb, bd ; ab, bd and ab, bc, cd . The only possible path now is ac, cd and so $d_{path}(a, d) = 2$.



5.2 Locality

Now that we have shown how different functions can be used to measure distance between elements in a data structure, we can use those concepts to define locality.

DEFINITION 5.10 Locality is a normalized degree that specifies how close one element of a data structure is to another. Locality is denoted as $l(m_1, m_2)$ where l can take any real value in the range of $[0, .1]$. Locality is defined as a ratio between the distance of the first element to the second and the diameter of the data structure. The distance is denoted by $d(e_1, e_2)$ and the diameter is denoted by $\varnothing(\mathbb{S})$. The locality is then defined by

$$l(e_1, e_2) = \frac{\varnothing(\mathbb{S}) - d(e_1, e_2)}{\varnothing(\mathbb{S})}$$

The definition of locality hinges on two concepts: distance and diameter. We have a definition of distance, but we still need to define what the diameter of data structure is.

DEFINITION 5.11 The diameter of a data structure is an upper boundary on the distances between all pairs of elements. The diameter of a specific instance of a data structure is denoted $\varnothing(\mathbb{S})$. By definition $\varnothing(\mathbb{S}) \geq d(e_1, e_2)$ for all $e_1, e_2 \in S^{base}$.

The function that we have defined to measure locality is what is called a similarity. A function $s: X \times X \rightarrow \mathbb{R}$ is called a similarity if it is non-negative (1), symmetric (2) and if $s(x, y) \leq s(x, x)$ holds for $x, y \in X$, with equality if and only if $x = y$ (3) [DD09, p.04]. These three properties hold for the definition of locality:

1. From DEFINITION 5.11 we know that $\varnothing(\mathbb{S}) \geq d(e_1, e_2)$ for all $e_1, e_2 \in S^{base}$. Therefore $\varnothing(\mathbb{S}) - d(e_1, e_2) \geq 0$ and $\frac{\varnothing(\mathbb{S}) - d(e_1, e_2)}{\varnothing(\mathbb{S})} \geq 0$. Hence $l(e_1, e_2)$ is non-negative for all $e_1, e_2 \in S^{base}$.
2. Let's assume that $\frac{\varnothing(\mathbb{S}) - d(e_1, e_2)}{\varnothing(\mathbb{S})} = \frac{\varnothing(\mathbb{S}) - d(e_2, e_1)}{\varnothing(\mathbb{S})}$, then $\varnothing(\mathbb{S}) - d(e_1, e_2) = \varnothing(\mathbb{S}) - d(e_2, e_1)$ and $d(e_1, e_2) = d(e_2, e_1)$ which is true according to the symmetry property of the definition of a distance. Hence $l(e_1, e_2)$ is symmetric.
3. This property holds if $\frac{\varnothing(\mathbb{S}) - d(e_1, e_2)}{\varnothing(\mathbb{S})} = \frac{\varnothing(\mathbb{S}) - d(e_1, e_1)}{\varnothing(\mathbb{S})}$ when $e_1 = e_2$ and $\frac{\varnothing(\mathbb{S}) - d(e_1, e_2)}{\varnothing(\mathbb{S})} < \frac{\varnothing(\mathbb{S}) - d(e_1, e_1)}{\varnothing(\mathbb{S})}$ when $e_1 \neq e_2$ for all $e_1, e_2 \in S^{base}$. The diameter is a constant in these equations and can be removed from both sides. We then get the equation $d(e_1, e_2) = d(e_1, e_1)$ which is true according to property 3 of the definition of a distance, and the inequality $d(e_1, e_2) > d(e_1, e_1)$ which is true, because $d(e_1, e_1) = 0$ (property 3) and $d(e_1, e_2) > 0$ (property 1 and 3). Hence, the third property of a similarity holds for $l(e_1, e_2)$.

To apply the definition of locality on a data structure we need to specify which distance function is used and what the diameter is. In the rest of this subchapter we give a definition of locality for each elementary data structure.

5.2.1 Locality in a set

From Definition 3.7 we know that a set data structure does not contain a relation. The absence of the relation gives us no means of measuring distance between two elements. To apply locality to a set we need a definition of distance in a set.

DEFINITION 5.12 Distance in a set is specified as $d^S(e_1, e_2) = 0$ if and only if $e_1 = e_2$. For all $e_1 \neq e_2$ we have $d^S = |S^{base}|$. The diameter is given by $\varnothing = |S^{base}|$. The locality in a set is denoted as $l^S(e_1, e_2)$.

EXAMPLE 5.9 We have the set $S = (\{e_1, e_2\}, \emptyset, \lambda)$. Here $l(e_1, e_1) = \frac{2-0}{2} = 1$ and $l(e_1, e_2) = \frac{2-2}{2} = 0$.

5.2.2 Locality in a vector

For a vector the Euclidean, Manhattan and Chebyshev distance all measure the same distance between two elements so we do not need to make a choice between those three when defining locality.

DEFINITION 5.13 The diameter of a vector is given by its size $\varnothing(V) = |S^{base}|$ and the locality of two elements in a set is defined by:

$$l^V(e_1, e_2) = \frac{\varnothing(S) - d(e_1, e_2)}{\varnothing(S)} = \frac{|S^{base}| - |F_V^{-1}(e_1) - F_V^{-1}(e_2)|}{|S^{base}|}$$

EXAMPLE 5.10 We have the vector $V = [v_1 \ v_2 \ v_3]$. The distance between v_1 and v_2 is $d(v_1, v_2) = |F_V^{-1}(v_1) - F_V^{-1}(v_2)| = 1$. The diameter is $\varnothing(V) = |S^{base}| = 3$. The locality of v_1 and v_2 then is: $l^V(e_1, e_2) = \frac{3-1}{3} = \frac{2}{3}$.

5.2.3 Locality in a matrix

Locality in a matrix is similar as in a vector. Only here it makes a difference what distance metric is chosen. The diameter of a matrix also depends on this choice. We define the diameter of an $n \times m$ -matrix as the distance from the origin to the point $n \times m$: $\varnothing(M) = d((0,0), (n, m))$.

DEFINITION 5.14 Locality in a matrix must be specified for a specific distance metric. We use the Euclidean metric as example. Locality is then defined as:

$$l_E^M(e_1, e_2) = \frac{\varnothing(S) - d(e_1, e_2)}{\varnothing(S)} = \frac{d_E^M((0,0), (n, m)) - d_E(e_1, e_2)}{d((0,0), (n, m))}$$

EXAMPLE 5.11

We have the 2×2 -matrix $M = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix}$.

When we use the Euclidean metric we have: the diameter is

$$\varnothing(M) = d_E^M((0,0), (2,2)) = \sqrt{8}; d(m_1, m_2) = \sqrt{2}; \text{ so the locality of } m_1 \text{ and } m_2 \text{ is } l_E^M(m_1, m_2) = \frac{\sqrt{8}-\sqrt{2}}{\sqrt{8}} = \frac{1}{2}.$$

When we use the Manhattan metric we have: the diameter is

$$\varnothing(M) = d_M^M((0,0), (2,2)) = 4; d_M^M(m_1, m_2) = 2; \text{ so the locality of } m_1 \text{ and } m_2 \text{ is } l_M^M(m_1, m_2) = \frac{4-2}{4} = \frac{1}{2}.$$

When we use the Chebyshov metric we have: the diameter is

$$\varnothing(M) = d_C^M((0,0), (2,2)) = 2; d_C^M(m_1, m_2) = 1; \text{ so the locality of } m_1 \text{ and } m_2 \text{ is } l_C^M(m_1, m_2) = \frac{2-1}{2} = \frac{1}{2}.$$

5.2.4 Locality in an array

Locality in an array works the same as in a matrix. The size of an array of dimension d is given by $n_1 \times \dots \times n_d$. The diameter is given by $\varnothing(A) = d((0_1, \dots, 0_d), (n_1, \dots, n_d))$.

DEFINITION 5.15

The locality of two elements in an array is defined (in this case using the Euclidean metric) as

$$\begin{aligned} l_E^A(e_1, e_2) &= \frac{\varnothing(S) - d(e_1, e_2)}{\varnothing(S)} \\ &= \frac{d_E^A((0_1, \dots, 0_d), (n_1, \dots, n_d)) - d_E^A(e_1, e_2)}{d_E^A((0_1, \dots, 0_d), (n_1, \dots, n_d))} \end{aligned}$$

5.2.5 Locality in a graph

The distance between two nodes in a graph is given by the path metric d_{path} . The diameter is given by the maximal length of the shortest (u, v) -path in a graph:

$$\varnothing(A) = \max(\{d_{path}(u, v) | u, v \in V\}).$$

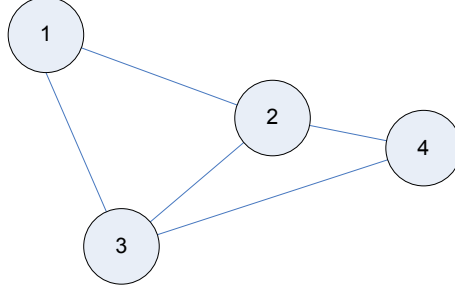
DEFINITION 5.16

The locality of two nodes in a graph is defined as:

$$\begin{aligned} l^G(e_1, e_2) &= \frac{\varnothing(S) - d(e_1, e_2)}{\varnothing(S)} \\ &= \frac{\max(\{path(u, v) | u, v \in V\}) - d_{path}(e_1, e_2)}{\max(\{path(u, v) | u, v \in V\})} \end{aligned}$$

EXAMPLE 5.12

We have the following graph:



In this case the diameter is 2. The locality of nodes 1 and 3 is:

$$l^G(e_1, e_3) = \frac{2 - 1}{2} = \frac{1}{2}$$

5.2.6 Locality in a tree

When we treat a tree as a graph we can use the path distance to measure distance between two nodes. This is not practical when we consider trees as data structures. We define the distance between two nodes as the longest distance from either of the two nodes to their lowest common ancestor.

DEFINITION 5.17 The distance between two nodes is defined as:

$$d_{tree}(e_1, e_2) = \max(d_{path}(lca(e_1, e_2), e_1), d_{path}(lca(e_1, e_2), e_2))$$

where $lca(e_1, e_2)$ is the lowest common ancestor of the nodes e_1 and e_2 .

DEFINITION 5.18 The diameter of a tree is the longest path from any node to the root:

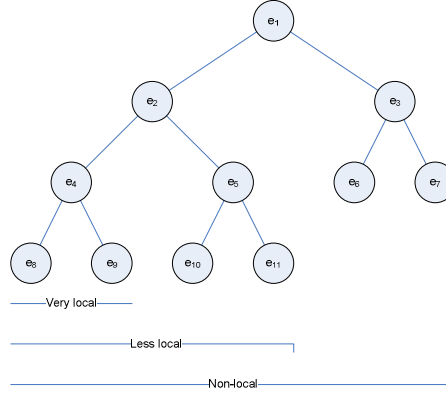
$$\mathcal{O}(T) = \max(\{d(r, e) \mid r, e \in S^{base}\})$$

where r is the root node. The locality in a tree is then defined as:

$$l^T(e_1, e_2) = \frac{\mathcal{O}(T) - d_{tree}(e_1, e_2)}{\mathcal{O}(T)}$$

EXAMPLE 5.13

We have the following tree:



$$\text{Here } \mathcal{O}(T) = 3; lca(e_8, e_9) = e_4; lca(e_5, e_8) = e_2; l(e_8, e_9) = \frac{3 - \max(\{d(e_8, e_4), d(e_9, e_4)\})}{3} = \frac{3-1}{3} = \frac{2}{3}; l(e_5, e_8) = \frac{3 - \max(\{1, 2\})}{3} = \frac{1}{3}.$$

5.3 Locality and transformations

Locality consists of two components (Definition 5.10): a distance measurement and a diameter. The transformation of a data structure may affect the locality in four ways:

- The distance between all the elements stays the same and the diameter stays the same: locality is not affected.
- The distance between all the elements stays the same, but the diameter changes: locality is affected.
- The distance between the elements changes, but the diameter stays the same: locality is affected.
- The distance between the elements changes and the diameter changes: locality changes unless the distance between the elements and the diameter are scaled with the same ratio.

In this part of the chapter we show for each of these four scenarios how the locality in the data structure is affected. We limit ourselves to vectors, matrixes and arrays as they share the same distance metrics.

5.3.1 Distance and Diameter Unchanged

In this scenario both the distance function and diameter remain unchanged after the transformation. A distance is measured between two elements. When the base set of elements changes, the distance function must change. Therefore the distance function can only be the same if the base set stays the same. Also for each pair of two elements the distance must be equal to the distance between those two elements after the transformation. EXAMPLE 5.14 shows that this is the case when a vector is mirrored.

EXAMPLE 5.14

The vector V is mirrored by a transformation: $[v_1 \ v_2] \rightarrow [v_2 \ v_1]$. Before the transformation we have:

$$\begin{aligned} F_V &= \{(1, v_1), (2, v_2)\} \quad d = \{(v_1, v_2, |1 - 2|), (v_2, v_1, |2 - 1|)\} = \\ &= \{(v_1, v_2, 1), (v_2, v_1, 1)\} \\ \emptyset(V) &= |S^{base}| = 2 \end{aligned}$$

After the transformation we have:

$$\begin{aligned} F'_V &= \{(1, v_2), (2, v_1)\} \quad d' = \{(v_1, v_2, |2 - 1|), (v_2, v_1, |1 - 2|)\} = \\ &= \{(v_1, v_2, 1), (v_2, v_1, 1)\} \\ \emptyset(V') &= |S^{base'}| = 2 \end{aligned}$$

Now we see that $d = d'$ and $\emptyset(V) = \emptyset(V')$ so the locality is preserved.

5.3.2 Distance Unchanged, Diameter Changed

In this part we look at transformations where the distance between elements that exist before and after the transformation have not changed, but where the transformation causes the diameter to change. This has the effect that the locality between two elements that exist before and after the transformation, changes. This is the case when elements are added or deleted.

EXAMPLE 5.15

We have the matrix $M = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix}$ here $d(m_1, m_2) = 1$ and $\max(d) = 3$ (when using Manhattan distance) so $l(m_1, m_2) = \frac{1}{3}$. We now add m_5 to position (1,3) and m_6 to position (2,3). This gives us the matrix $M' = \begin{bmatrix} m_1 & m_2 & m_5 \\ m_3 & m_4 & m_6 \end{bmatrix}$. Still $d(m_1, m_2) = 1$, but now $\max(d') = 4$ so $l(m_1, m_2) = \frac{1}{4}$.

EXAMPLE 5.16

We have the matrix M from EXAMPLE 5.15. We now delete m_3 and m_4 . Now $\max(d') = 2$ so $l(m_1, m_2) = \frac{1}{2}$.

5.3.3 Distance Changed, Diameter Unchanged.

In this part we look at transformations that cause the distance between elements to change while the maximum distance stays the same. This is the case for rearrangement transformations. When elements in a matrix, vector, or array are swapped, the maximum distance doesn't change, but the distance between individual elements changes.

EXAMPLE 5.17

We have the matrix $M = \begin{bmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \end{bmatrix}$ and we swap the elements m_1 and m_3 . Before the swap we had $d(m_1, m_3) = 2$ and after the swap we have $d(m_1, m_3) = 1$.

5.3.4 Distance Changed, Diameter Changed

It is also possible that both the distance between elements and the diameter changes. This happens when one type of data structure is transformed into another by creating completely new elements. Only in some of these cases the maximum distance changes with the same degree as the distance between all the elements. This is the case for example when an image is stretched. When the distance between the elements increases with the same ratio as the diameter, locality is preserved.

EXAMPLE 5.18

In the following transformation a 2x2 matrix is resized to a 4x4 matrix:

$$\begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \rightarrow \begin{bmatrix} m'_1 & m'_2 & m'_3 & m'_4 \\ m'_5 & m'_6 & m'_7 & m'_8 \\ m'_9 & m'_{10} & m'_{11} & m'_{12} \\ m'_{13} & m'_{14} & m'_{15} & m'_{16} \end{bmatrix}$$

$$\text{Here } T(m_1) = \begin{bmatrix} m'_1 & m'_2 \\ m'_5 & m'_6 \end{bmatrix} \text{ and } T(m_2) = \begin{bmatrix} m'_3 & m'_4 \\ m'_7 & m'_8 \end{bmatrix}.$$

The distance between the elements m_1 and m_2 from the input is using the Euclidean method is: $d_E(m_1, m_2) = 1$. If we take the distance between each corresponding element from the output we see that it has increased twofold: $d'_E(m'_1, m'_3) = d'_E(m'_2, m'_4) = d'_E(m'_5, m'_7) = d'_E(m'_6, m'_8) = 2$.

At the same time the diameter also increases twofold:

$$\frac{\varnothing(M')}{\varnothing(M)} = \frac{\sqrt{|4^2 + 4^2|}}{\sqrt{|2^2 + 2^2|}} = \frac{\sqrt{32}}{\sqrt{8}} = 2$$

The locality of elements m_1 and m_2 is also the same as the locality of their corresponding output elements:

$$l_E^M(m_1, m_2) = \frac{\varnothing(M) - d_E^M(m_1, m_2)}{\varnothing(M)} = \frac{\sqrt{8} - 1}{\sqrt{8}} \approx 0,65 \text{ and } l_E^{M'}(m'_1, m'_3) = \frac{\sqrt{32} - 2}{\sqrt{32}} \approx 0,65.$$

EXAMPLE 5.19

When we use the case from EXAMPLE 5.18 but now use the Manhattan distance, we get the same results:

$$\frac{\varnothing(M')}{\varnothing(M)} = \frac{|4 + 4|}{|2 + 2|} = \frac{8}{4} = 2$$

$$\frac{d_M^{M'}(m'_1, m'_3)}{d_M^M(m_1, m_2)} = \frac{|1 - 1| + |3 - 1|}{|1 - 1| + |2 - 1|} = \frac{2}{1} = 2$$

$$l_M^M(m_1, m_2) = \frac{4 - 1}{4} = l_M^{M'}(m'_1, m'_3) = \frac{8 - 2}{8} = \frac{3}{4}$$

EXAMPLE 5.20

When we use the case from EXAMPLE 5.18 but now use the Chebyshov distance, we get the same results:

$$\frac{\varnothing(M')}{\varnothing(M)} = \frac{\max(\{4, 4\})}{\max(\{2, 2\})} = \frac{4}{2} = 2$$

$$\frac{d_C^{M'}(m'_1, m'_3)}{d_C^M(m_1, m_2)} = \frac{\max(\{|1 - 1|, |3 - 1|\})}{\max(\{|1 - 1|, |2 - 1|\})} = \frac{2}{1} = 2$$

$$l_C^M(m_1, m_2) = \frac{4 - 1}{4} = l_C^{M'}(m'_1, m'_3) = \frac{8 - 2}{8} = \frac{3}{4}$$

5.4 Conclusion

In this final part of the chapter we will reflect on the results that try to answer the research questions. In this chapter we have addressed the following research question:

RESEARCH QUESTION 3 How can locality be used to describe the effect of a transformation on the relation between elements in a data structure?

In this chapter we have introduced the definition of locality. Locality is a normalized function based on the distance between two elements and the diameter of a data structure. First we have presented a formal definition of distance. We have given examples of four different ways of measuring distance in different data structures. We have used the definition of distance to introduce a new definition: locality. Locality is based on the distance between two elements and the diameter of the data structure that contains the elements. Finally we have shown what the possible effects are of a transformation on the locality of two elements.

The results of this chapter are used in chapter 6 where locality is used in one of the scenario's to propagate annotations.

Propagation of Annotations

In the introduction we have described a workflow that consists of processing elements that transform streams into other streams. The data in these streams may be organized as data structures. In Chapter 3 we define how these data structures can be described and in Chapter 4 we define how the transformation of these data structures can be described. The data structures in the stream may be annotated. In this model we describe how annotations in a data structure can be transformed. First, we describe how data structures can be annotated. Next, we present a model for deciding on the propagation of annotations. Finally, we show how this model is applied in five different scenarios.

In this chapter we answer the following research question:

RESEARCH QUESTION 4

How can annotations in a data structure be propagated?

6.1 Annotated data structures

Data structures can be annotated to mark parts of the data structure with extra information. Mostly annotations are used in streaming setups to indicate that there are quality issues with part of the data. In Chapter 3 we define what a data structure is and how it is denoted (DEFINITION 3.6). We extend this definition to create a new definition for an annotated data structure:

DEFINITION 6.1 An annotated data structure is a data structure that contains annotations and is denoted by $S^* = (S^{base}, R, \lambda, A)$ where $(S^{base}, R, \lambda) = S$ according to DEFINITION 3.5 and A is a set of annotations.

Since the data is transformed in a streaming setup, annotations also may need to be transformed. This means that in addition to T_D and T_S we are now also concerned with T_A which described the transformation of the annotations.

DEFINITION 6.2 A transformation T^* is a transformation of an annotated data structure where $T^*: \mathbb{S}^* \rightarrow \mathbb{S}^{*'}$ and $T^* = T \cup \{T_A\}$.

According to DEFINITION 3.5 a data structure consists of a set of elements, a relation and a value-function. Annotations can occur in all three of them, depending on how the system is set up. In each of these cases we denote the set of annotations by A_E for annotated elements, A_R for annotated elements of the relation and A_λ for annotated values. For example, we might want to annotate an element so that it remains annotated even when it gets a new value. An element of the relation can be annotated when for example a complete row in a matrix must remain annotated, even after extra elements are added to that row. Finally, sometimes it may be necessary to annotate the value of an element, so it only remains annotated as long as the value is not changed. We consider annotating subsets of elements the same as annotating individual elements.

When a data structure is annotated extra information is added. Annotations may contain all kinds of information, but it is not the aim of our research to investigate the semantics of actual annotations. We consider binary annotations for simplicity: something is annotated or not; we do not specify any value for the annotation.

This means that in the case of annotated elements the set of annotations is subset of the set of elements: $A_E \subseteq S^{base}$. An element e is annotated if $e \in A_E \wedge e \in S^{base}$. Elements are given a value by the λ function. The set of all the values in the data structure is called S^{value} . The set of annotated values is then a subset of the value set: $A_\lambda \subseteq S^{value}$.

The relation may be of a very different type according to the particular type of data structure. For vectors, matrixes and arrays, the relation can be modeled as a function where the domain consists of a numbered index and the co-domain of elements. For graphs and trees the relation is binary relation on the set of nodes. This has an impact on how to specify annotations on the relation. In some cases whole elements of the relation are annotated and $A_R \subseteq R$. In other cases the basis of A_R will be formed by parts of the elements of the relation. We discuss these implications for each type of data structure.

It is also possible that an annotation cannot be related to a specific element, relation or value. In this case we can only say that the data structure as a whole is annotated. We indicate this by keeping \mathbb{S}^* but with $A = \{complete\}$.

We will now look at how annotations can be made on each type of elementary data structure.

6.1.1 Set

From DEFINITION 3.7 we know that a set data structure does not contain a relation. This means that annotations on a set data structure can only be made on elements, their values, and the set as a whole.

Suppose we have a set of temperature values

$S = \{22.0, 26.5, 22.0\} = (\{e_1, e_2, e_3\}, \emptyset, \{(e_1, 22.0), (e_2, 26.5), (e_3, 22.0)\})$ and we know that any value below 25°C is an outlier. We can annotate the corresponding elements: $A_E = \{e_1, e_3\}$ or annotate the values: $A_\lambda = \{22.0\}$. Here it becomes clear why it makes a difference in annotating elements or values. When elements have the same value and only the value is annotated it is not possible to distinguish between which element is annotated.

When we cannot annotate specific values or elements and still want to indicate that the set in the example is annotated we denote it by:

$$S^* = (\{e_1, e_2, e_3\}, \emptyset, \{(e_1, 22.0), (e_2, 26.5), (e_3, 22.0)\}, \{complete\})$$

6.1.2 Vector

Annotating a vector can be done as for any data structure by annotation elements, values, elements of the relation and the vector as a whole. What makes annotating vectors different from annotating other structures such as sets and graphs, is how annotations are made at the relation level. Relations in a vector are formed by a function that maps an index number to an element (DEFINITION 3.9-I). So this relation consists of an index number and an element. In this case annotating the whole relation means effectively annotating an element. It is more useful to only make an annotation for the position. We have the annotated vector $V^* = [22 \ 15 \ 21^*]$. The annotation could be made in three different ways: $A_E = \{v_3\}$; $A_\lambda = \{21\}$; $A_R = \{3\}$.

The difference becomes clear when we reverse the vector: $V^{*'} = [21^* \ 15 \ 22]$. When the annotation is made for the element or the value, the annotation stays the same, but when the annotation is made at the relation level, the annotation must be transformed: $A'_R = \{1\}$.

6.1.3 Matrix

Annotating a matrix is similar to annotating a vector. Annotations on elements, values and the matrix as a whole are the same as for other data structures. For a vector it is possible to annotate positions. This is also the case for a matrix. A position in a matrix is denoted by two natural numbers: (i, j) and the annotations in a matrix could look like $A_R = \{(2,2), (3,1)\}$. It may be useful to annotate a whole row or column. Such annotations are indicated by a * for one of the index numbers: $(1,*)$ means that the whole first row is annotated and $(*,2)$ means that the whole second column is annotated.

6.1.4 Array

Annotating arrays is equal to annotating matrixes and vectors. In a matrix whole rows and columns could be annotated. This is also possible in an array. Any number in the index can be replaced by a * indicating that the whole range for that index number is annotated, for example: $(1,*,6,*)$.

6.1.5 Graph

A graph data structure consists of nodes and edges and is denoted by $G = (S^{base}, R, \lambda) = (V, E, \lambda)$ (DEFINITION 3.21). The nodes are the elements and the edges are the relations of a graph data structure. Annotations in a graph can be made at all the places as specified for a general data structure: node, value, edge and the graph as a whole. This is expressed graphically in Figure 6.1.

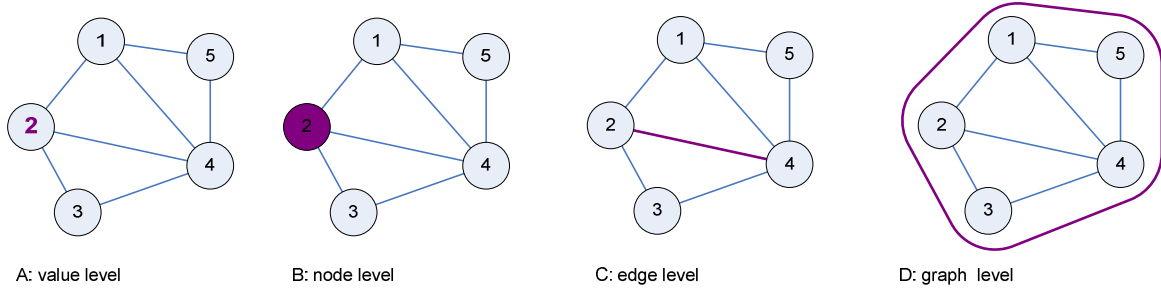


Figure 6.1: three levels of annotations in a graph

At the value level annotations are given by $A_\lambda \subseteq \lambda(V)$ and at the node level annotations are given by $A_E \subseteq V$. An edge is a combination of two nodes. Annotating edges means that the whole relation is part of the annotation: $A_R \subseteq E$. Lastly, when the graph as a whole is annotated it is specified by $A = \emptyset$.

6.1.6 Tree

Annotating a tree works exactly as annotating graphs.

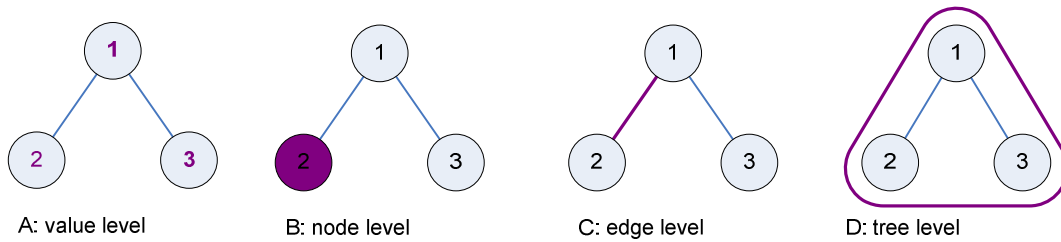


Figure 6.2: three levels of annotations in a tree

6.2 Annotation transformation model

In this subchapter we introduce the annotation transformation model which can be used to decide on the propagation of annotations on elements. DEFINITION 6.2 states that the transformation of an annotated data structure also includes a transformation of the annotations called T_A . How the transformation of the annotations looks like depends on the transformation of the data and the structure and choices made by the user.

These choices are about making a decision under which conditions an annotation should be propagated or not. We propose a model for propagating annotations for elements, where the decision is based on the information related to the input elements. Although we use only structural information in this model as a basis for deciding whether an annotation should be propagated, it is also possible to extend the model to include the value of an element.

There are two methods of describing the transformation of annotations. The first method looks at all the annotated elements in the input and describes where the annotation should be placed in the output. The second method looks at all the output elements and decides on the basis of the contributing input elements whether this particular output element should be annotated or not. Our model is based on the second method because in this way it is possible to compare the different input elements that contribute to the output element.

In our model the transformation of annotations consists of going over all the output elements and deciding for each element whether it should be annotated on the basis of the contributing input elements.

DEFINITION 6.3 An element is annotated if it is a member of the annotation set:
 $e \in A$

DEFINITION 6.4 The transformation of the annotations consists of calculating for all the output elements an annotation weight. When the annotation weight exceeds a certain threshold, the output element is annotated:

$$T_A: A' = \{e^{out} \in S^{base'} \mid w_a[e^{out}] \geq threshold\}$$

DEFINITION 6.5 To decide whether an output element should be annotated, an annotation weight is calculated:

$$w_a[e^{out}] = \left(\frac{\sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}] * a[e^{in}]}{\sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}]} \right)$$

Where according to DEFINITION 4.19: $S^{in}[e^{out}]$ is the set of contributing input elements of the output element e^{out} ;

$$a[e^{in}] = \begin{cases} 1, & e^{in}.annotated \\ 0, & !e^{in}.annotated \end{cases}$$

and $w[e^{in}]$ is the weight that the input element carries in producing the output element.

What this formula does is finding all the contributing input elements for the given output element. If an element is annotated $a[e^{in}]$ has value 1. This value is then multiplied by the weight $w[e^{in}]$ of the input element in the transformation. This weight is specific to the transformation and can take any numeral value. The sum of all the weights multiplied by the annotation value is then compared to a threshold. If it is higher than the threshold, the output element is annotated. We divide this sum by the sum of all the weights to normalize the value. This way the value can be compared to a threshold $\in [0,00..1,00]$. We call the denominator of the formula the normalization value:

$$\sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}]$$

There are two parts in this formula that are particular to the type of transformation. The first is assigning a weight to an element. This can be done on the basis of the value of the element and its relations. We do not take the value of an element into account. The second part is finding the input elements that correspond to the output element. The annotation transformation therefore needs the following information:

- Description of the elements and structure of the input
- Description of the elements and structure of the output
- A position transformation function
- A formula that assigns weights to contributing elements

In the next subchapter we apply the model to structure-by-structure transformations as described in Chapter 4.

6.3 Transformation of Annotations

The annotation transformation model depends mainly on the cardinality of the contributing input elements and the weight assigned to each contributing element. Different scenarios can be distinguished according to the cardinality of the transformation and the weight chosen. We denote the amount of contributing input elements of an output element by n . On the basis of n and the weights that are assigned to input elements we describe five scenarios. This is not a complete list, but functions as a list of examples to show how the generic formula can be applied.

- $n = 1$; $w[e^{in}] = 1$ for all e^{in} . In this scenario there is exactly one input element for each output element. Also we assume that each element has the same influence on the output element and therefore we assign the weight 1.
- $n \geq 1$; $w[e^{in}] = 1$ for all e^{in} . In this case we assume there may be more than one input element, but each input element has the same contribution.
- $n \geq 1$; $w[e^{in}] = \{1,2\}$. In this case we show how in a matrix multiplication some elements may have more influence on the output than others.
- $n \geq 1$; $w[e^{in}]$ *depends on distance*. In this case we show a more complex scenario where the distance between a contributing input element and the output element is used to determine the weight.
- $n \geq 1$; $w[e^{in}]$ *depends on locality*. In this case we show a scenario where the locality of a contributing input element and the output element is used to determine the weight.

The rest of this subchapter describes how the annotation transformation model is applied to each of the five scenarios.

6.3.1 One input element / weight 1

In this part of the chapter we consider one-to-one transformations where exactly one input element contributes to an output element (DEFINITION 4.9). This element can be found using DEFINITION 4.24. There is only one input element and we base the decision to annotate the output element only on the fact whether that input element is annotated we use a weight of 1.

Because there is only one input element and the weight is 1, there's no need to normalize. This gives us the following formula:

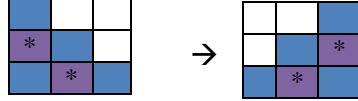
$$w_a[e^{out}] = \sum_{e^{in} \in S^{in}[e^{out}]} 1 * a[e^{in}]$$

Because the transformation is one-to-one, the set $S^{in}[e^{out}]$ contains exactly one element. This means that we can replace e^{in} in the formula by the part from DEFINITION 4.24 that retrieves the input element:

$$w_a[e^{out}] = a \left[F_A \left(T_P^{-1} \left(F_A^{-1'}(e^{out}) \right) \right) \right]$$

EXAMPLE 6.1

Let the transformation T flip an image.



The stars indicate annotated cells. To describe the propagation of the annotations we need the following information:

- The elements and structure of the input: $(S^{base}, R) = \left(\begin{matrix} m_1, m_2, m_3, \\ m_4, m_5, m_6, \\ m_7, m_8, m_9, \end{matrix} \left\{ \begin{matrix} (1,1, m_1) (1,2, m_2) (1,3, m_3) \\ (2,1, m_4) (2,2, m_5) (2,3, m_6) \\ (3,1, m_7) (3,2, m_8) (3,3, m_9) \end{matrix} \right\} \right)$
- The annotation information from the input: $A = \{m_4, m_8\}$
- The elements and structure from the output: $(S^{base'}, R') = \left(\begin{matrix} m'_1, m'_2, m'_3, \\ m'_4, m'_5, m'_6, \\ m'_7, m'_8, m'_9, \end{matrix} \left\{ \begin{matrix} (1,1, m'_1) (1,2, m'_2) (1,3, m'_3) \\ (2,1, m'_4) (2,2, m'_5) (2,3, m'_6) \\ (3,1, m'_7) (3,2, m'_8) (3,3, m'_9) \end{matrix} \right\} \right)$
- The inverse position transformation function:
$$T_P^{-1}(i, j) = (i, 3 - j + 1)$$

Using this information we can now calculate for each output element whether it should be annotated or not. For example:

$$\begin{aligned} w_a[m'_9] &= a \left[F_A \left(T_P^{-1} \left(F_A^{-1'}(e^{out}) \right) \right) \right] = a \left[F_A \left(T_P^{-1}(3,3) \right) \right] \\ &= a[F_A(3,1)] = a[m_7] = 0 \end{aligned}$$

And

$$\begin{aligned} w_a[m'_6] &= a \left[F_A \left(T_P^{-1} \left(F_A^{-1'}(e^{out}) \right) \right) \right] = a \left[F_A \left(T_P^{-1}(2,3) \right) \right] \\ &= a[F_A(2,1)] = a[m_4] = 1 \end{aligned}$$

We use *threshold* = 1, so $m'_9 \notin A$ and $m'_6 \in A$.

6.3.2 Multiple input elements / weight 1

In the *many-to-one* (DEFINITION 4.11) and *many-to-many* (DEFINITION 4.12) scenarios as described in Chapter 4, multiple elements contribute to one output element. Some of these contributing input elements may be annotated while others are not. In this case we choose to annotate an element in the output only if more than a specific share of the contributing input elements is annotated. We do this by modifying the generic annotation model to use a weight of 1 for all the elements (which effectively means $w[e^{in}]$ can be left out of the formula). This way the sum of all the weights is equal to the amount of elements and we can normalize the result by dividing by n , where $n = |S^{in}[e^{out}]|$:

$$w_a[e^{out}] = \frac{1}{n} \sum_{e^{in} \in S^{in}[e^{out}]} a[e^{in}]$$

This formula first identifies the contributing input elements, sums 1 if they are annotated – otherwise 0, and divides the result by the amount of contributing input elements.

EXAMPLE 6.2

Let the matrix $M = \begin{bmatrix} 2^* & 4 & 6 \\ 3 & 5^* & 7^* \end{bmatrix}$ be transformed into the vector $V = \begin{bmatrix} 12 \\ 15 \end{bmatrix}$ by summing all the values in a row. The elements with a * are annotated. We annotate an element in the output if more than 50% of its input is annotated. To describe T_A we need to know that:

$$(S^{base}, R) = \left(\{m_1, m_2, m_3\}, \{(1,1, m_1), (1,2, m_2), (1,3, m_3), (2,1, m_4), (2,2, m_5), (2,3, m_6)\} \right)$$

$$(S^{base'}, R') = (\{v_1, v_2\}, \{(1, v_1), (2, v_2)\})$$

$$T_p(i, j) = i$$

$$A = \{m_1, m_5, m_6\}$$

Now we can calculate what elements in the output should be annotated:

$$\begin{aligned} w_a[v_1] &= \frac{1}{n} \sum_{e^{in} \in S^{in}[v_1]} a[e^{in}] = \frac{1}{3} \sum_{e^{in} \in \{m_1, m_2, m_3\}} a[e^{in}] \\ &= \frac{1}{3} * (1 + 0 + 0) = \frac{1}{3} \end{aligned}$$

And:

$$\begin{aligned} w_a[v_2] &= \frac{1}{n} \sum_{e^{in} \in S^{in}[v_2]} a[e^{in}] = \frac{1}{3} \sum_{e^{in} \in \{m_4, m_5, m_6\}} a[e^{in}] \\ &= \frac{1}{3} * (0 + 1 + 1) = \frac{2}{3} \end{aligned}$$

When we use an arbitrary *threshold* $= \frac{1}{2}$ we have $v_1 \notin A$ and $v_2 \in A$.

6.3.3 Multiple input elements / weight $\in \{1,2\}$

We consider a special case where a square matrix is multiplied with itself. Again multiple elements are used to produce an output element. We use the self-multiplication of a matrix from EXAMPLE 4.16:

$$M' = M^2 = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a \times a + b \times c & a \times b + b \times d \\ c \times a + d \times c & c \times b + d \times d \end{bmatrix}$$

One of the input elements is used twice, this occurs when the position of the output element is equal to the position of the input element. We assign a weight of 1 to elements that contribute once to the same output element and a weight of 2 to elements that contribute twice to the same output element:

$$w[e^{in}] = \begin{cases} 1 & F_M^{-1}(e^{in}) \neq F_M^{-1'}(e^{out}) \\ 2 & F_M^{-1}(e^{in}) = F_M^{-1'}(e^{out}) \end{cases}$$

Of all the contributing input elements there is only one with weight 2, the rest has weight 1. Therefore:

$$\sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}] = n + 1 \text{ where } n = |S^{in}[e^{out}]|$$

This gives the following annotation formula:

$$w_a[e^{out}] = \frac{1}{n+1} \sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}] * a[e^{in}]$$

Because matrix multiplication is a many-to-many transformation we need to define T_{P++}^{-1} (DEFINITION 4.29). We can then use DEFINITION 4.30 to find the contributing input elements. For the multiplication of a square $m \times m$ -matrix the position transformation function is given by:

$$T_{P++}^{-1}(i, j) = \{(i, 1) \dots (i, m)\} \cup \{(1, j) \dots (m, j)\}$$

EXAMPLE 6.3

We consider the multiplication of a 2x2 matrix with a threshold of 1:

$$M' = M^2 = \begin{bmatrix} 2^* & 0^* \\ 0^* & 2 \end{bmatrix}^2 = \begin{bmatrix} 2 \times 2 + 0 \times 0 & 2 \times 0 + 0 \times 2 \\ 0 \times 2 + 2 \times 0 & 0 \times 0 + 2 \times 2 \end{bmatrix} = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$$

We need the following information:

$$(S^{base}, R) = \left(\{m_1, m_2\}, \{(1,1, m_1), (1,2, m_2)\}, \{m_3, m_4\}, \{(2,1, m_3), (2,2, m_4)\} \right)$$

$$(S^{base'}, R') = \left(\{m'_1, m'_2\}, \{(1,1, m'_1), (1,2, m'_2)\}, \{m'_3, m'_4\}, \{(2,1, m'_3), (2,2, m'_4)\} \right)$$

$$T_{P++}^{-1}(i, j) = \{(i, 1), (i, 2)\} \cup \{(1, j), (2, j)\}$$

$$A = \{m_1, m_2, m_3\}$$

Now:

$$\begin{aligned} w_a[m'_1] &= \frac{1}{3+1} \sum_{e^{in} \in \{m_1, m_2, m_3\}} w[e^{in}] * a[e^{in}] \\ &= \frac{1}{4} (2 * 1 + 1 * 1 + 1 * 1) = 1 \end{aligned}$$

And:

$$\begin{aligned} w_a[m'_2] &= \frac{1}{3+1} \sum_{e^{in} \in \{m_1, m_2, m_4\}} w[e^{in}] * a[e^{in}] \\ &= \frac{1}{4} (1 * 1 + 2 * 1 + 1 * 0) = \frac{3}{4} \end{aligned}$$

And:

$$\begin{aligned} w_a[m'_3] &= \frac{1}{3+1} \sum_{e^{in} \in \{m_1, m_3, m_4\}} w[e^{in}] * a[e^{in}] \\ &= \frac{1}{4} (1 * 1 + 2 * 1 + 1 * 0) = \frac{3}{4} \end{aligned}$$

And:

$$\begin{aligned} w_a[m'_4] &= \frac{1}{3+1} \sum_{e^{in} \in \{m_2, m_3, m_4\}} w[e^{in}] * a[e^{in}] \\ &= \frac{1}{4} (1 * 1 + 1 * 1 + 2 * 0) = \frac{2}{4} \end{aligned}$$

With a *threshold* = 1 we get $A = \{m'_1\}$ and $M'^* = \begin{bmatrix} 4^* & 0 \\ 0 & 4 \end{bmatrix}$. With a *threshold* = $\frac{3}{4}$ we get $A = \{m'_1, m'_2, m'_3\}$ and $M'^* = \begin{bmatrix} 4^* & 0^* \\ 0^* & 4 \end{bmatrix}$.

6.3.4 Multiple input elements / weight depends on distance

In this part we consider a more complex transformation. We extend EXAMPLE 4.28 to create a transformation that blurs values in a vector using a window of size 5.

EXAMPLE 6.4

Let the input vector be $V = [5 \ 4 \ 6 \ 3 \ 8 \ 7 \ 5 \ 6 \ 4]$. Now, the window at position i is given by $window[i] = [i - 2 \ \dots \ i + 2]$, so $window[4] = [4 \ 6 \ 3 \ 8 \ 7]$. The values at position $i - 2$ and $i + 2$ are taken once, the values at positions $i - 1$ and $i + 1$ are taken two times and the value at position i is taken three times, and the weighted average is calculated to produce a value in the output at position i . So,

$$output[4] = \frac{1*4+2*6+3*3+2*8+1*7}{1+2+3+2+1} = \frac{48}{9} = \frac{16}{3} \approx 5,3.$$

When the window falls partly out of the range of the input vector, the positions outside the vector are omitted. The complete output is: $V' = [4,8 \ 4,6 \ 5 \ 5,3 \ 6,1 \ 6,2 \ 5,8 \ 5,4 \ 4,8]$.

The formula that produces an output value using a window of size 5 is given by:

$$\lambda'(e^{out}) = \frac{\sum_{i=\max(1,p)}^{\min(n,p)} (3 - d(p,i)) \times \lambda(e^{in})}{\sum_{i=\max(1,p)}^{\min(n,p)} (3 - d(p,i))}$$

$$\text{with } p = F_V^{-1'}(e^{out}); e^{in} = F_V(i); n = |S^{base}|; d(p,i) = |p - i|$$

From this formula we can distill the inverse position transformation function to find the contributing input elements for an output element (DEFINITION 4.29).

$$T_{p++}^{-1}(i) = \{\max(1, i - 2), \dots, \min(n, i + 2)\}$$

We can then use the generic annotation transformation model to decide which output elements should be annotated (DEFINITION 6.5):

$$w_a[e^{out}] = \frac{\sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}] * a[e^{in}]}{\sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}]}$$

We base the contribution weights on the distance between input and output elements:

$$w[e^{in}] = 3 - d(F_V^{-1}(e^{in}), F_V^{-1'}(e^{out}))$$

EXAMPLE 6.5

We consider the case from EXAMPLE 6.4 and assume the following:

$$\begin{aligned}
 &(S^{base}, R) \\
 &= (\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}, \\
 &\{(1, v_1), (2, v_2), (3, v_3), (4, v_4), (5, v_5), (6, v_6), (7, v_7), (8, v_8), (9, v_9)\}) \\
 &(S^{base'}, R') \\
 &= (\{v'_1, v'_2, v'_3, v'_4, v'_5, v'_6, v'_7, v'_8, v'_9\}, \\
 &\{(1, v'_1), (2, v'_2), (3, v'_3), (4, v'_4), (5, v'_5), (6, v'_6), (7, v'_7), (8, v'_8), (9, v'_9)\})
 \end{aligned}$$

$$A = \{v_4, v_6\}$$

We can now calculate A' . First we calculate $S^{in}[v'_2]$ using DEFINITION 4.30:

$$S^{in}[v'_2] = \{e^{in} \in S^{base} \mid F_V^{-1}(e^{in}) \in T_{P++}^{-1}(F_V^{-1'}(v'_2))\}$$

, where

$$\begin{aligned}
 T_{P++}(F_A^{-1'}(v'_2)) &= T_{P++}(2) = \{\max(1, i-2), \dots, \min(n, i+2)\} = \\
 &= \{\max(1, 2-2), \dots, \min(9, 2+2)\} = \{1, 2, 3, 4\}
 \end{aligned}$$

From here we know that $S^{in}[v'_2] = \{v_1, v_2, v_3, v_4\}$.

$$\begin{aligned}
 &w_a[v'_2] \\
 &= \frac{(w[v_1] * a[v_1]) + (w[v_2] * a[v_2]) + (w[v_3] * a[v_3]) + (w[v_4] * a[v_4])}{w[v_1] + w[v_2] + w[v_3] + w[v_4]} \\
 &= \frac{(2 * 0) + (3 * 0) + (2 * 0) + (1 * 1)}{2 + 3 + 2 + 1} = \frac{1}{8}
 \end{aligned}$$

We can calculate the annotation weights for all elements:

$$\begin{bmatrix} 0 & \frac{1}{8} & \frac{2}{9} & \frac{4}{9} & \frac{4}{9} & \frac{4}{9} & \frac{2}{9} & \frac{1}{8} & 0 \end{bmatrix}$$

on the basis of a threshold three elements in the output get annotated:

$$A' = \{v'_4, v'_5, v'_6\}$$

6.3.5 Multiple input elements / weight depends on locality

In the previous cases the weight was used to compare contributing input elements. When we use weights based on the distance, the weights have no absolute meaning. They cannot be compared to weights in other cases. Because the weights are not normalized, the total result needs to be normalized. Locality is a means of normalizing distance, as the distance is measured against the maximum distance possible in that data structure. Here we show how locality can be used as a weight. We use the case from EXAMPLE 6.4 only this time we base the weight on the locality instead of the distance.

The new annotation formula is given by:

$$w_a[e^{out}] = \frac{1}{n} \sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}] * a[e^{in}]$$

The weight is now based on the locality of the input and output elements:

$$w[e^{in}] = l_E^V(e^{in}, e^{out}) = \frac{\mathcal{O}(V) - d_E^V(e^{in}, e^{out})}{\mathcal{O}(V)}$$

$$\text{where } \mathcal{O}(V) = |S^{base}|; d_E^V(e^{in}, e^{out}) = |F_V^{-1}(e^{in}) - F_V^{-1}(e^{out})|$$

Since the weights are now all in the range of [0..1] we can simply normalize the result by dividing by n .

EXAMPLE 6.6

Using the above case we can now calculate the annotation values for each output element. We describe the full calculation for one output element: $Annotated(v'_5)$. The diameter is given by the size of the vector: $\mathcal{O}(V) = |S^{base}| = 9$.

$$w_a[v'_5] = \frac{1}{5} * \left(\frac{7}{9} * 0 + \frac{8}{9} * 1 + \frac{9}{9} * 0 + \frac{8}{9} * 1 + \frac{7}{9} * 0 \right) = \frac{1}{5} * \frac{16}{9} = \frac{16}{45}$$

For all the output elements the annotation levels are given by:

$$\left[0 \quad \frac{7}{36} \quad \frac{7}{45} \quad \frac{16}{45} \quad \frac{16}{45} \quad \frac{16}{45} \quad \frac{7}{45} \quad \frac{7}{36} \quad 0 \right]$$

The annotated output is given by $A' = \{v'_4, v'_5, v'_6\}$ which in this case is the same as when we used the distance as the weight.

6.4 Conclusion

In this final part of the chapter we will reflect on the results that try to answer the research questions. In this chapter we have addressed the following research question:

RESEARCH QUESTION 4 How can annotations in a data structure be propagated?

In order to understand how annotations could be propagated, it necessary to know annotations can be made in a data structure. We have shown for each type of elementary data structure how annotations can be made: at the level of elements, relations, values and the data structure as a whole.

In the second part of this chapter we have presented a model for transforming annotations at the element-level. When the transformation is of the structure-from-structure type, the annotations can be propagated using the position transformation function. We have shown for five different scenarios how the results from Chapter 4 can be used in the model presented in this chapter. In the first scenario there is only one contributing input element and the output element is simply annotated when the contributing input element is annotated. In the second scenario there is more than one contributing input element and the output is annotated when more than a certain share of the contributing input elements is annotated. In the third case the contributing input elements are given a weight of 1 or 2. In the fourth scenario the weight depends on the distance between contributing input element and output element. The last scenario shows how in addition to the results from Chapter 4, the results from Chapter 5 can be used in the annotation transformation model presented in this chapter. In this scenario the weight depends on the locality of the contributing input element and output element.

In the next chapter we will validate the annotation model presented in this chapter by applying the results to three non-trivial real-life examples.

Chapter 7

Use Cases

In the chapters 3 – 6 we have provided definitions that can be used as building blocks to describe how annotations can be propagated in a work flow. We have given many trivial examples to show how these building blocks can be used. In this chapter we present three non-trivial cases distilled from scientific applications to show how the building blocks can be used in practice.

7.1 Earth Remote Sensing

Earth remote sensing is about gathering data using sensors positioned in the air. This can be done using planes, balloons and satellites. In this subchapter we focus on creating photographic maps using satellites. We present a use case from scientific literature that describes a possible work flow for the processing of the data produced by the satellites. This case is extended to create a practical example with complete specifications to which the annotation transformation model can be applied.

We start by giving an overall description of the use case. We proceed to give an overview of the data structures used in the workflow, based on Chapter 3. Next we describe the transformations in the workflow using the definitions from Chapter 4. Finally we use these descriptions of the data structures and the transformations to describe the propagation of the annotations using the material from Chapter 6.

7.1.1 Description

SciDB is a data management system that works with very large scale array data [Sci11] and is still under development. In the early phase of the development process a few use cases were defined. One of these deals with earth remote sensing and imaging satellites in particular [Fre08]. The Landsat program has orbited multiple satellites of the type described in the SciDB use case, of which Landsat 7 is the most recent. The Landsat 7 Science Data Users Handbook provides many details about this satellite and the data it

generates [Nasa11]. The satellite has a near polar orbit which ensures, in combination with the rotation of the earth, that the whole earth between 81 degrees north and south is covered. Figure 7.1 shows the orbit of the satellite.

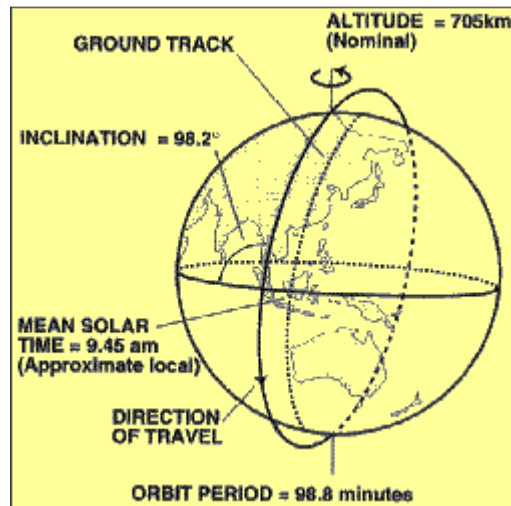


Figure 7.1: The orbit of Landsat 7

The satellite scans the earth in lines from west to east while it moves along the track from south to north (and vice versa). These are called scan lines and are perpendicular to the ground track. A series of scan lines is called a swath [HDF08]. Figure 7.2 shows a satellite moving along its track.

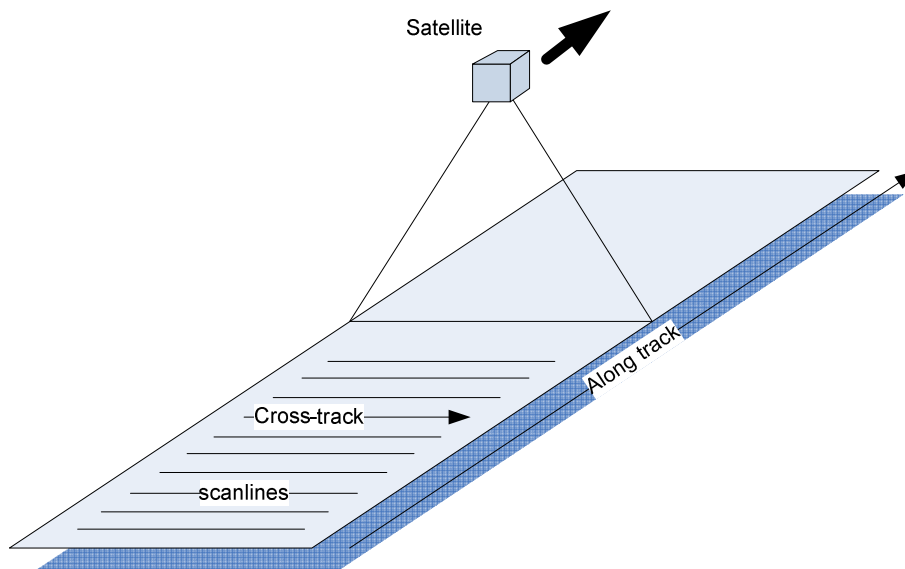


Figure 7.2: A satellite creating a swath by scanning the earth [HDF08]

Figure 7.3 shows the track of a swath. The ground track is approximately 185 km wide and is framed each 170 km. The exact size of the covered area depends on the resolution, which is 30 meter, so one pixel represents an area of 30x30 meter. The resulting image swath consists of 6000 scan lines of 6600 pixels each. The total area covered by an image swath therefore has a width of 198m and a length of 190m (6000x30m) [Nasa11, p.49]. The satellite orbits the earth just over 14 times a day. Each orbit it has moved 2752km east with respect to the previous orbit, which creates a large gap between two orbits. Each day its position with respect to the ground moves the width of one swath and in 16 days it covers the whole area between the tracks of two consecutive orbits as is shown in Figure 7.4.

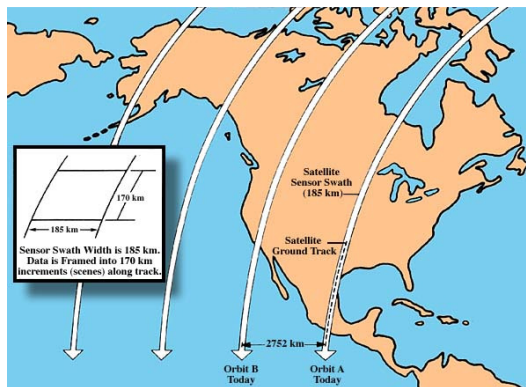


Figure 7.3: Swath track [Nasa11, p.38]

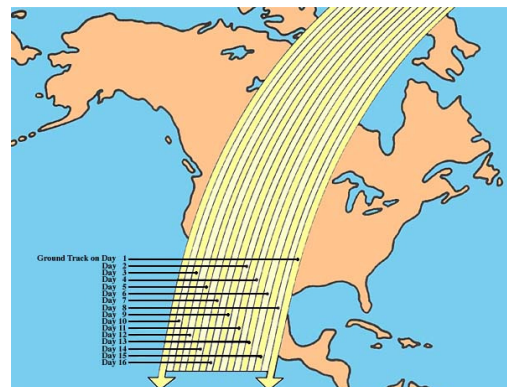


Figure 7.4: Swath Pattern [Nasa11, p.38]

The earth rotates while the satellite moves from pole to pole. This causes the satellite to move from east to west with respect to the ground. The produced swaths are therefore skewed. Also the swaths have overlap (ranging from 7% on the equator and 80% near the poles) because the earth is a globe. To produce proper image maps, these swaths need to be projected [Nasa11, p.102].

The projection algorithm is quite complex and the used data structures are quite large, therefore we simplify the workflow that generates image from satellite data. Figure 7.5 shows the workflow. A box represents a processing element (source, transformation and sink). A circle represents a data structure.

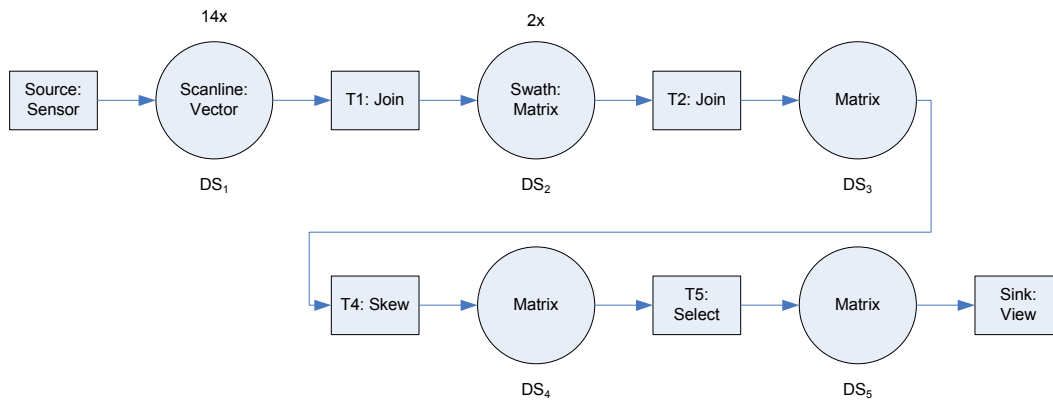


Figure 7.5: An earth remote sensing workflow

Although a real scan line is 6600 pixels wide and a real image swath 6000 scan lines, we consider scan lines of 4 pixels and swaths of 7 scan lines. Our workflow produces a projected image from two swaths, so 14 scan lines are needed. The first processing element is the sensor, which is of the source type, which generates 14 scan lines. Each consecutive 7 scan lines are framed into a swath. This transformation is performed by the processing element T1. The processing element T2 combines two swaths to create an image which is still skewed. The skewed image is unskewed by T3 and finally T4 selects the right portion of the unskewed image to produce a usable image which is consumed by a sink. We ignore here that real swaths need projection and contain overlap. The only image processing we consider is the unskewing. Figure 7.6 shows how the workflow looks when the data is viewed as colored pixels.

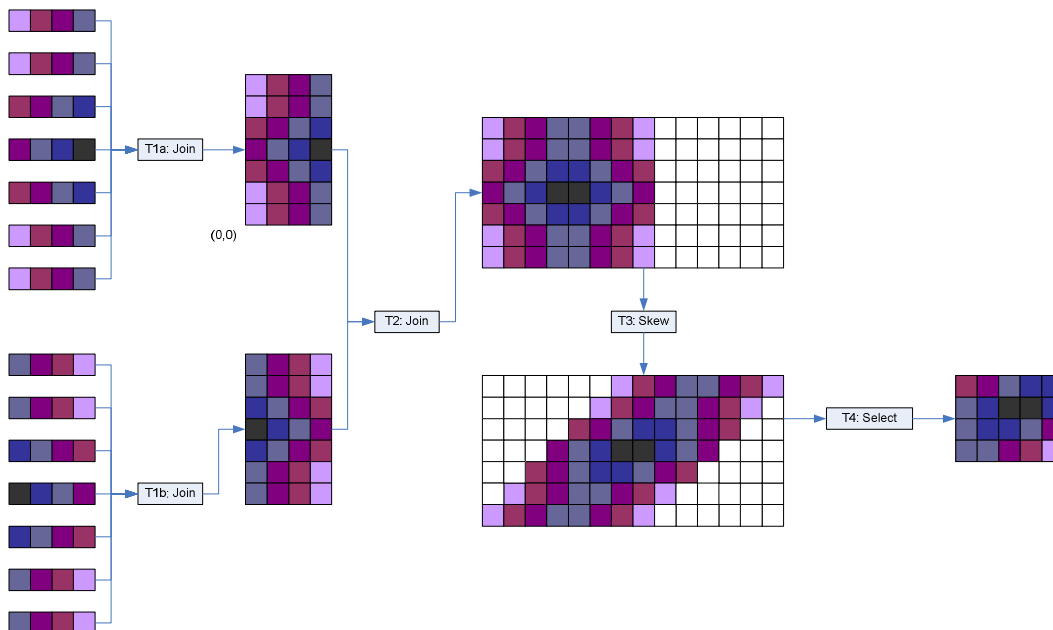


Figure 7.6: The transformation of the data produced by the satellite

7.1.2 Data structures

All the data structures used in the workflow are image data. Essentially they are treated as one and two-dimensional arrays, so we choose to model them as vectors and matrixes of pixels instead. We use the definitions from Chapter 3 to give a formal description for each of the five data structures.

- **Data Structure 1:** The satellite captures 14 scan lines of 7 pixels each. We number each of the scan lines $1 \leq i \leq 14$ and each pixel within a scan line $1 \leq j \leq 4$. Pixel number j in scan line number i is then defined as (using the definition of a vector, DEFINITION 3.10):

$$V_{pixel}[i][j] = (\{r, g, b\}, \{(1, r), (2, g), (3, b)\}, \lambda) \text{ with } \lambda: \{r, g, b\} \rightarrow \{0 \dots 255\}$$

We model a scan line as a vector of pixels (DEFINITION 3.10). The number i denotes the number of the scan line, which we will consider to be the number of the input source (DEFINITION 4.33).

$$\begin{aligned} V_{scanline}[i] \\ = (\{v_1^i, \dots, v_4^i\}, \{(1, v_1^i), \dots, (4, v_4^i)\}, \{(v_1^i, V_{pixel}[i][1]), \dots, (v_4^i, V_{pixel}[i][4])\}) \end{aligned}$$

- **Data Structure 2:** The 14 scan lines are combined into swaths of 7 scan lines each. The result is a 7×4 -matrix. We number the swaths: $1 \leq k \leq 2$. The matrix that represents swath number k is given by:

$$M_{swath}[k] = \left(\begin{array}{c} \{m_{4 \times (i-1)+j}^k | 1 \leq i \leq 7, 1 \leq j \leq 4\}, \\ \{(i, j, m_{4 \times (i-1)+j}^k) | 1 \leq i \leq 7, 1 \leq j \leq 4\}, \\ \lambda \end{array} \right)$$

- **Data Structure 3:** The two swaths are joined to create an image. Empty cells are added which are needed in the next processing step. The result is a 7×14 -matrix:

$$M_{image} = \left(\begin{array}{c} \{m_{14 \times (i-1)+j} | 1 \leq i \leq 7, 1 \leq j \leq 14\}, \\ \{(i, j, m_{14 \times (i-1)+j}) | 1 \leq i \leq 7, 1 \leq j \leq 14\}, \\ \lambda \end{array} \right)$$

- **Data Structure 4:** The previous data structure is unskewed producing a new image:

$$M_{image} = \left(\begin{array}{c} \{m_{14 \times (i-1)+j} | 1 \leq i \leq 7, 1 \leq j \leq 14\}, \\ \{(i, j, m_{14 \times (i-1)+j}) | 1 \leq i \leq 7, 1 \leq j \leq 14\}, \\ \lambda \end{array} \right)$$

- **Data Structure 5:** The unskewed image still contains a lot empty pixels. The final data structure is produced by selecting a rectangular range of pixels:

$$M_{image} = \left(\begin{array}{c} \{m_{6 \times (i-1)+j} | 1 \leq i \leq 4, 1 \leq j \leq 6\}, \\ \{(i, j, m_{6 \times (i-1)+j}) | 1 \leq i \leq 4, 1 \leq j \leq 6\}, \\ \lambda \end{array} \right)$$

The following table gives an overview of the properties of the five data structures:

	DS ₁	DS ₂	DS ₃	DS ₄	DS ₅
Type	Vector	matrix	matrix	matrix	Matrix
Amount	14	2	1	1	1
Size	7	7x4	7x8	7x14	4x6
Ordering	Ordered	Ordered	Ordered	Ordered	Ordered
Dimension	1	2	2	2	2
Relation based	n/a	n/a	n/a	n/a	n/a
Hierarchical	n/a	n/a	n/a	n/a	n/a

7.1.3 Transformations

In this part we describe the processing elements that perform a transformation on the data structures. There are four processing elements of the transformation type. We use the definitions of Chapter 4 to describe the relation between input and output elements. These descriptions will in the next part of the chapter be used to show how annotations can be propagated.

- **Transformation 1:** The first transformation is of the join type as described in the introduction of this chapter. T1 consists of two separate sub transformations, T1[1] and T1[2], where each transformation takes 7 scan lines and converts it into a swath. The output swaths are numbered, which are the numbers in brackets. For each output swath k , the inverse position transformation function (DEFINITION 4.23) is given by:

$$T_p^{-1}[k](i, j) = j$$

For each swath seven different input data structures are used. This means that the contributing input elements for two different output elements may come from different input data structures. This is a one-to-one n -ary transformation, of the join type as described in Chapter 4. For these type of transformations, the input number can directly be calculated on the basis of the output element (DEFINITION 4.34). For swath k this input number function is given by:

$$in_{\#}[k](e^{out}) = 7 \times (k - 1) + i \text{ with } (i, j) = F_M^{-1}[k](e^{out})$$

We use the input number function to find the input elements that contribute to an output element of the swath $M_{swath}[k]$ in this one-to-one n -ary transformation:

$$S^{in}[k][e^{out}] = \left\{ e^{in} \in S^{base}[h] \mid h = in_{\#}[k](e^{out}) \wedge e^{in} = F_V[h] \left(T_P^{-1}[k] \left(F_M^{-1'}[k](e^{out}) \right) \right) \right\}$$

We can now find for each output element its contributing input element. For example $S^{in}[1][m_6^1]$. First we need to know the position of the output element: $F_M^{-1'}[1](m_6^1) = (2,2)$. Using the inverse position transformation function we can find the position of the input element: $T_P^{-1}[1](2,2) = 2$. The input number of the contributing data structure is given by: $in_{\#}[1](m_6^1) = 2$. The contributing input element can be found using: $F_V2 = v_2^2$.

The same calculations can be done for each output element. The following table shows four examples and the calculations for all output elements are given in Appendix B.1.

k	e^{out}	$F_M^{-1'}[k](e^{out}) = (i, j)$	$T_P^{-1}[k](i, j) = p$	$in_{\#}[k](e^{out}) = n$	$F_V[n](p)$
1	$m_6^{1'}$	$F_M^{-1'}[1](m_6^1) = (2,2)$	$T_P^{-1}[1](2,2) = 2$	$in_{\#}[1](m_6^1) = 2$	$F_V2 = v_2^2$
1	$m_{24}^{1'}$	$F_M^{-1'}[1](m_{24}^1) = (6,4)$	$T_P^{-1}[1](6,4) = 4$	$in_{\#}[1](m_{24}^1) = 6$	$F_V[6](4) = v_4^6$
2	$m_8^{2'}$	$F_M^{-1'}[2](m_8^2) = (2,4)$	$T_P^{-1}[2](2,4) = 4$	$in_{\#}[2](m_8^2) = 9$	$F_V[9](4) = v_4^9$
2	$m_{14}^{2'}$	$F_M^{-1'}[2](m_{14}^2) = (4,2)$	$T_P^{-1}[2](4,2) = 2$	$in_{\#}[2](m_{14}^2) = 11$	$F_V[11](2) = v_2^{11}$

- **Transformation 2:** The second transformation is also a join. The two data structures that form the output of T1 are joined to produce one matrix. Extra columns are added which are needed by T3. The inverse position transformation function for T2 is given by (DEFINITION 4.23):

$$T_P^{-1}(i, j) = (i, 1 + (j - 1 \bmod 4)), \text{ for } j \leq 8$$

From this function it follows that the row of the output element is the same as the row of its contributing input element. In the output matrix the two input matrixes are joined from left to right, such that the first four elements in a row in the output matrix correspond to the four elements in a row of the first input matrix. The second four elements in a row of the output matrix correspond to the four elements in a row of the second input matrix. For example the fifth element of the fourth row of the output matrix corresponds to the first element of the fourth row of the second input matrix. The elements in column 9-14 are added extra, so there are no corresponding input elements. For this reason the

function is only defined for $j \leq 8$. To find the contributing input data structure for a given output element we define the input number function (DEFINITION 4.34):

$$in_{\#}(e^{out}) = \left\lfloor \frac{j}{4} \right\rfloor \text{ where } (i, j) = F_M^{-1}(e^{out})$$

We use the input number function to find the input elements that contribute to an output element of the matrix M_{image} in this one-to-one n -ary transformation:

$$S^{in}[e^{out}] = \left\{ e^{in} \in S^{base}[k] \mid k = in_{\#}(e^{out}) \wedge e^{in} = F_M[k] \left(T_P^{-1} \left(F_M^{-1'}(e^{out}) \right) \right) \right\}$$

For example, we can find the contributing input element of m'_{16} . First we need to find its position: $F_M^{-1'}(m'_{16}) = (2, 2)$. The position of the contributing input element is: $T_P^{-1}(2, 2) = (2, 2)$ and the number of the input number is: $in_{\#}(m'_{16}) = 1$. The contributing output element is now given by $F_M[1](2, 2) = m_6^1$. The following table shows four examples (full calculations are given in Appendix B.1):

e^{out}	$F_M^{-1'}(e^{out}) = (i, j)$	$T_P^{-1}(i, j) = (p, q)$	$in_{\#}(e^{out}) = n$	$F_M[n](p, q)$
m'_{16}	$F_M^{-1'}(m'_{16}) = (2, 2)$	$T_P^{-1}(2, 2) = (2, 2)$	$in_{\#}(m'_{16}) = 1$	$F_M[1](2, 2) = m_6^1$
m'_{25}	$F_M^{-1'}(m'_{25}) = (2, 11)$	$T_P^{-1}(2, 11)$ <i>not defined</i>	n/a	n/a
m'_{74}	$F_M^{-1'}(m'_{74}) = (6, 4)$	$T_P^{-1}(6, 4) = (6, 4)$	$in_{\#}(m'_{74}) = 1$	$F_M[1](6, 4) = m_{24}^1$
m'_{22}	$F_M^{-1'}(m'_{22}) = (2, 8)$	$T_P^{-1}(2, 8) = (2, 4)$	$in_{\#}(m'_{22}) = 2$	$F_M[2](2, 4) = m_8^2$
m'_{48}	$F_M^{-1'}(m'_{48}) = (4, 6)$	$T_P^{-1}(4, 6) = (4, 2)$	$in_{\#}(m'_{48}) = 2$	$F_M[2](4, 2) = m_{14}^2$

- **Transformation 3:** The third transformation skews the image. This means that almost all elements are moved. This transformation takes only one input and is reversible, so we can defined both the position transformation function (DEFINITION 4.21) and its inverse (DEFINITION 4.23):

$$T_P(i, j) = \left(i, 1 + ((6 - i + j) \bmod 14) \right)$$

$$T_P^{-1}(i, j) = \left(i, ((j - 8 + i) \bmod 14) + 1 \right)$$

Starting from the bottom each row is moved one position more to the right. Elements that ‘fall off the map’ are added to beginning of the row. We find the contributing input elements using DEFINITION 4.24. For example the contributing input element for output element m'_{48} is given by:

$$S^{in}[m'_{48}] = \{e^{in} \in S^{base} \mid e^{in} = F_M(T_P^{-1}(F_M^{-1'}(m'_{48})))\}$$

First we calculate the position: $F_M^{-1'}(m'_{48}) = (4,9)$. The position of the contributing input element is then given by: $T_P^{-1}(4,9) = (4, ((9 - 6 + 4) \bmod 13) - 1) = (4,6)$. $F_M(4,6) = m_{45}$, so the contributing input element then is: $S^{in}[m'_{48}] = \{m_{45}\}$. The following table shows four examples (full calculations are given in Appendix B.1):

e^{out}	$F_M^{-1'}(e^{out}) = (i, j)$	$T_P^{-1}(i, j) = (p, q)$	$F_M(p, q)$
m'_{21}	$F_M^{-1'}(m_{21}) = (2, 7)$	$T_P^{-1}(2, 7) = (2, 2)$	$F_M(2, 2) = m_{16}$
m'_{17}	$F_M^{-1'}(m_{17}) = (2, 3)$	$T_P^{-1}(2, 3) = (2, 11)$	$F_M(2, 11) = m_{25}$
m'_{75}	$F_M^{-1'}(m_{75}) = (6, 5)$	$T_P^{-1}(6, 5) = (6, 4)$	$F_M(6, 4) = m_{74}$
m'_{27}	$F_M^{-1'}(m_{27}) = (2, 13)$	$T_P^{-1}(2, 13) = (2, 8)$	$F_M(2, 8) = m_{22}$
m'_{51}	$F_M^{-1'}(m_{51}) = (4, 9)$	$T_P^{-1}(4, 9) = (4, 6)$	$F_M(4, 6) = m_{48}$

- **Transformation 4:** The last transformation in the workflow makes a rectangular selection of the pixels in the centre. This is a one-to-one non-reversible transformation. The inverse position transformation function is given by:

$$T_P^{-1}(i, j) = (i + 2, j + 4), \text{ for } 1 \leq i \leq 4, 1 \leq j \leq 6$$

We find the contributing input elements using DEFINITION 4.24. For example the contributing input element for output element m'_{11} is given by:

$$S^{in}[m'_{11}] = \{e^{in} \in S^{base} \mid e^{in} = F_M(T_P^{-1}(F_M^{-1'}(m'_{11})))\}$$

We start by calculation the position in the output matrix: $F_M^{-1'}(m'_{11}) = (2, 5)$. The position in the input matrix is then: $T_P^{-1}(2, 5) = (2 + 2, 5 + 4) = (4, 9)$. Finally, $F_M(4, 9) = m_{52}$ so the contributing input element is: $S^{in}[m'_{11}] = \{m_{52}\}$. The following table shows two examples (the complete calculation is shown in Appendix B.1):

e^{out}	$F_M^{-1'}(e^{out}) = (i, j)$	$T_P^{-1}(i, j) = (p, q)$	$F_M(p, q)$
m'_{11}	$F_M^{-1'}(m_{11}) = (2, 5)$	$T_P^{-1}(2, 5) = (4, 9)$	$F_M(4, 9) = m_{52}$
m'_{19}	$F_M^{-1'}(m_{19}) = (4, 1)$	$T_P^{-1}(4, 1) = (6, 5)$	$F_M(6, 5) = m_{75}$

In Chapter 4 we list five properties of data transformations. The following table lists these properties for each transformation. From this table it becomes clear that most transformations in the workflow have the same properties.

	T1	T2	T3	T4
<i>Structure dependency</i>	<i>S</i>	<i>S</i>	<i>S</i>	<i>S</i>
<i>Data dependency</i>	-	-	-	-
<i>Element dependency</i>	-	-	-	<i>S</i>
<i>Element cardinality</i>	1: 1	1: 1	1: 1	1: 1
<i>Relation cardinality</i>	1: 1	1: 1	1: 1	1: 1
<i>Reversible</i>	Yes	Yes	Yes	No
<i>Invariance</i>	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>
<i>Data/structure coherence</i>				
<i>Transformation formula</i>	$T = \{T_S^S\}$	$T = \{T_S^S\}$	$T = \{T_S^S\}$	$T = \{T_S^S, T_E^S\}$

7.1.4 Annotations

We assume that there are some annotations in the data structures that are generated by the sources. The data structures that are generated by the source are the 14 vectors $V[1] \dots V[14]$. Each vector data structure has its own annotation set: $A[1] \dots A[14]$. We assume that the annotation sets have the following values: $A[1] = \emptyset, A[2] = \{v_2^2\}, A[3] \dots A[10] = \emptyset, A[11] = \{v_2^{11}\}, A[12] \dots A[14] = \emptyset$. We now show for each transformation how the annotations can be propagated.

- **Transformation 1:** This transformation is one-to-one so we can use the annotation formula from Chapter 6.3.1:

$$w_a[e^{out}] = a \left[F_V[in_{\#}(e^{out})] \left(T_P^{-1} \left(F_M^{-1'}(e^{out}) \right) \right) \right]$$

$$\text{where } a[e^{in}] = \begin{cases} 1, & e^{in} \in A[in_{\#}(e^{out})] \\ 0, & e^{in} \notin A[in_{\#}(e^{out})] \end{cases}$$

To find out if an output element must be annotated, we need to find out whether its contributing input element is annotated or not. In the previous part of this chapter we calculated for each output element the contributing input element. We now need to know if the contributing input element is annotated. We do this by retrieving the annotation set of the contributing input data structure. For the output element $m_6^{1'}$ the contributing input element is v_2^2 . The annotation set of the contributing data structure is given by $A[in_{\#}(e^{out})] = A[2] = \{v_2^2\}$. Since the contributing input element is part of the annotation set, the output element must be annotated as well. The following table shows this calculation for four examples.

k	e^{out}	e^{in}	$in_{\#}(e^{out})$	$A[in_{\#}(e^{out})]$	$a[e^{in}]$	$e^{out} \in A$
1	$m_6^{1'}$	v_2^2	2	$A[2] = \{v_2^2\}$	1	<i>true</i>
1	$m_{24}^{1'}$	v_4^6	6	$A[6] = \emptyset$	0	<i>false</i>
2	$m_8^{2'}$	v_4^9	9	$A[9] = \emptyset$	0	<i>false</i>
2	$m_{14}^{2'}$	v_2^{11}	11	$A[11] = \{v_2^{11}\}$	1	<i>true</i>

Appendix B.1 shows the calculation for each output element. From this table we find that the two annotation sets in the output are formed by:

$$A'[1] = \{m_6^{1'}\}; A'[2] = \{m_{14}^{2'}\}$$

- **Transformation 2:** This transformation is a join just as T1. The annotations in the output are also calculated using the same formula. For four output elements this gives the following result:

e^{out}	$in_{\#}(e^{out}) = n$	e^{in}	$A[in_{\#}(e^{out})]$	$a[e^{in}]$	$e^{out} \in A$
m'_{16}	1	m_6^1	$A[1] = \{m_1^6\}$	1	<i>true</i>
m'_{25}	n/a	n/a	n/a	0	<i>false</i>
m'_{74}	1	m_{24}^1	$A[1] = \{m_1^6\}$	0	<i>false</i>
m'_{22}	2	m_8^2	$A[2] = \{m_{14}^2\}$	0	<i>false</i>
m'_{48}	2	m_{14}^1	$A[2] = \{m_{14}^2\}$	1	<i>true</i>

From the complete listing in Appendix 3.1 we retrieve the output annotation set:

$$A' = \{m'_{16}, m'_{48}\}$$

- **Transformation 3:** Since there is only one input data structure, there is no need to find the contributing annotation set. Output elements are simply annotated when the contributing input element is annotated.

e^{out}	e^{in}	A	$a[e^{in}]$	$e^{out} \in A$
m'_{21}	m_{16}	$A = \{m_{16}, m_{48}\}$	1	<i>true</i>
m'_{17}	m_{25}	$A = \{m_{16}, m_{48}\}$	0	<i>false</i>
m'_{75}	m_{74}	$A = \{m_{16}, m_{48}\}$	0	<i>false</i>
m'_{27}	m_{22}	$A = \{m_{16}, m_{48}\}$	0	<i>false</i>
m'_{52}	m_{48}	$A = \{m_{16}, m_{48}\}$	1	<i>true</i>

From the complete listing in Appendix 3.1 we retrieve the output annotation set:

$$A' = \{m'_{21}, m'_{51}\}$$

- **Transformation 4:** Just as with T3, output elements are annotated when their contributing input element is annotated.

e^{out}	e^{in}	A	$a[e^{in}]$	$e^{out} \in A$
m'_{11}	m_{51}	$A = \{m_{21}, m_{51}\}$	1	<i>true</i>
m'_{19}	m_{75}	$A = \{m_{21}, m_{51}\}$	0	<i>false</i>

From the complete listing in Appendix 3.1 we retrieve the output annotation set:

$$A' = \{m'_{11}\}$$

7.1.5 Conclusion

For this use case it was possible to apply the annotation without difficulties. This is mainly due to the data structures and transformations used. All data structures are of a similar type: either vector or matrix. The transformations are all one-to-one so we had to find only one contributing input element for each output element. The output element would then be annotated simply when its contributing input element was annotated. Therefore picking the right threshold was not an issue here. There was no transformation of the values, so it was no problem that the annotation model does not take the values into account. Finally we conclude that the annotation model is fully applicable to this use case.

7.2 Temperature sensor network

Environmental sensor networks are used to monitor the environment. These sensor networks may be deployed on a large scale to cover a large geographic area, for example weather stations. They may also be deployed on a local scale, where they normally measure relatively straightforward generic properties such as temperature and humidity [HM06]. An example of such a local sensor network is deployed in the Antarctic to monitor microclimates on earth, to mimic environmental circumstances on Mars [DCJ+03]. This sensor network consists of 14 nodes that each measure internal temperature, external temperature and humidity. The nodes are connected to each other through a wireless network. One node functions as the mother node, which is attached to a laptop, and continuously downloads data from the other nodes.

We use this application to derive a practical use case. Besides collecting the data we add an extra step that creates a color map using the temperature readings by interpolation of the data. We proceed with describing the workflow in detail. Next we show how the data structures, transformations, and the propagation of annotations can be modeled using the definitions from Chapters 3–6.

7.2.1 Description

The case here presented is a simplified version of real life applications. The sensor network in the case consists of three sensors that each produces a temperature measurement. The locations of the sensors are converted into points on a map. A grid is placed over this map to move the points to the nearest point on the grid. Now the positions of the points on the grid can be used to put the temperature values in a matrix. The empty cells in the matrix are completed using interpolation on the three temperature measurements. Finally the values are converted into color codes to produce a color map. Figure 7.7 outlines the workflow of the case.

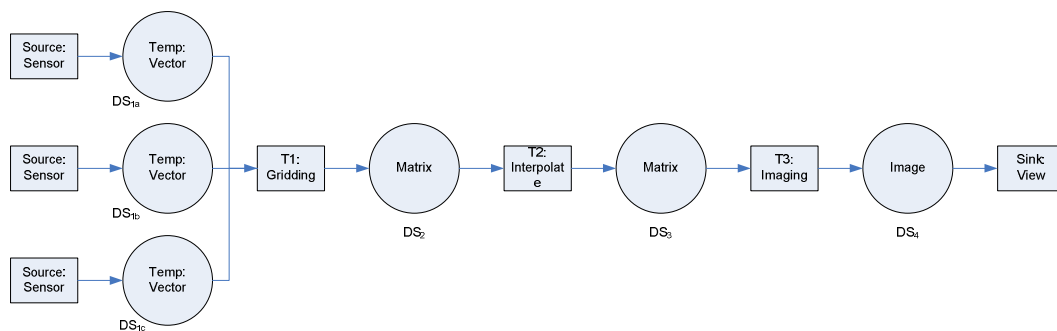


Figure 7.7: Workflow of the temperature sensor network

The three processing steps from the workflow:

- The first processing is done by the three sources, which are the sensors. Each of them have a fixed location and generate streams of temperature readings. A temperature reading consists of a real number value. The following table shows the number of the sensor, its position, and the temperature value that is produced:

Sensor	Location	Temp
1	52.427, 5.581	28,0
2	52.429, 5.616	27,0
3	52.394, 5.592	25,0

- The sensors are placed at specific coordinates. We want to put the data in a matrix in such a way that the position in the matrix resembles the real position. This means that we need to discretize the positions. This is done using a grid. Figures 7.8 shows the original map and Figure 7.9 show the grid that we want the points to be placed on. The dividing lines in figure 7.10 are used to move the points onto the grid, which is shown in figure 7.11.

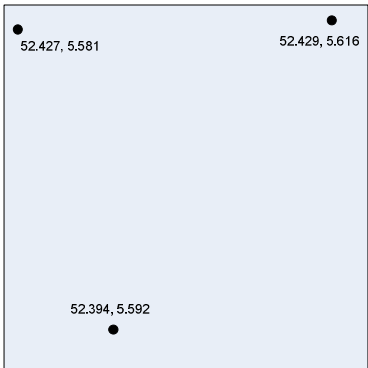


Figure 7.8: A map of the area

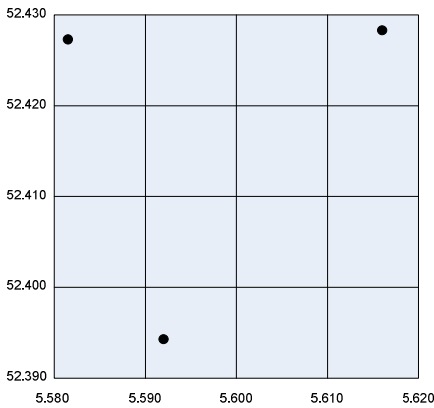


Figure 7.9: The grid

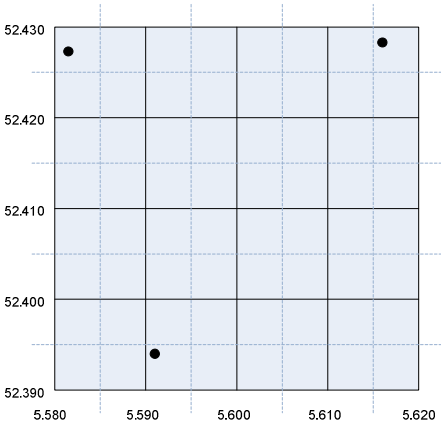


Figure 7.10: Dividing lines

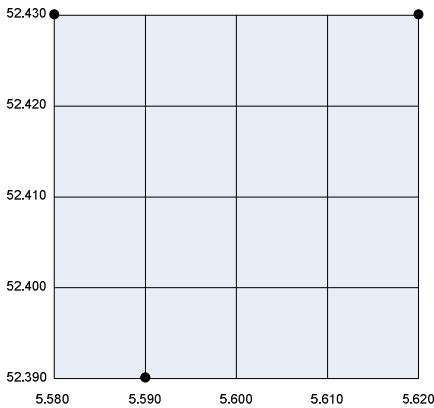


Figure 7.11: Points moved onto grid

The grid in figure 7.11 consists of 5×5 points. We let each point correspond to a cell in a 5×5 matrix. Data from sensor number 1 is then placed in cell (1,1) which is the top left cell. Data from sensor number 2 is then placed in cell (1,5) and data from sensor number 3 is placed in cell (5,2). This gridding may be part of the transformation. In that case the location of a sensor must also form part of the input. We choose to keep the position of the sensors in this case fixed, so the data always is placed in cells (1,1), (1,5) and (5,2). The output of this processing element is:

$$\begin{bmatrix} 28,0 & \square & \square & \square & 27,0 \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & 25,0 & \square & \square & \square \end{bmatrix}$$

- The matrix has now three cells that contain a value, while the other cells are still empty. The values of the empty cells are calculated using interpolation. The interpolation method that we use is called Inverse Distance Weighting (IDW) [She68]. The method uses a set of given values to interpolate a new value. It differs per implementation which and how many given values are used to calculate an interpolated value. In our case there are three sensors and hence three given values. For each of the empty cells in the matrix, we will use all three given values as input for the interpolation algorithm. An interpolated value can be calculated using:

$$u(\mathbf{x}) = \frac{\sum_{i=1}^n w_i(\mathbf{x}) * u_i}{\sum_{i=1}^n w_i(\mathbf{x})}$$

In this formula, $u(\mathbf{x})$ is the interpolated value at point \mathbf{x} and u_i are the input values. In this case $n = 3$ because we have three input values. The influence of an input value on the interpolated value is inversely related to the distance from the input point to the interpolated point. This weight is $w_i(\mathbf{x})$ is defined as:

$$w_i(\mathbf{x}) = \frac{1}{d(\mathbf{x}, \mathbf{x}_i)^p}$$

Here $d(\mathbf{x}, \mathbf{x}_i)$ is the distance from the known input point \mathbf{x}_i to the unknown point \mathbf{x} . Specifically $x_1 = (1,1)$; $x_2 = (1,5)$; $x_3 = (5,2)$. The choice for p determines the influence of far nodes on the interpolation. We use $p = 2$ because that presents the easiest calculation of the distance, while it also gives satisfactory empirical results [She68]. For $p = 2$ we get the following distance based weight (based on Euclidean distance, DEFINITION 5.1):

$$d(\mathbf{x}, \mathbf{y})^2 = |(x_1 - y_1)^2 + (x_2 - y_2)^2|$$

The weights 1-3 for point x at position (2,2) to the three input cells are:

$$w_1(x) = \frac{1}{|(2-1)^2+(2-1)^2|} = \frac{1}{2}; w_2(x) = \frac{1}{|(2-1)^2+(2-5)^2|} = \frac{1}{10}; w_3(x) = \frac{1}{|(2-2)^2+(2-5)^2|} = \frac{1}{9}.$$

The interpolated value is then:

$$u(x) = \frac{\frac{1}{2} * 28,0 + \frac{1}{10} * 27,0 + \frac{1}{9} * 25,0}{\frac{1}{2} + \frac{1}{10} + \frac{1}{9}} \approx 27,4$$

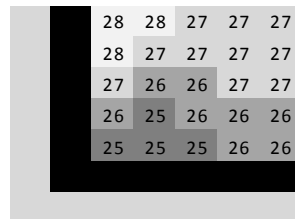
After the interpolation is performed for all unknown values we get:

$$\begin{bmatrix} 28,0 & 27,7 & 27,2 & 27,0 & 27,0 \\ 27,7 & 27,4 & 27,0 & 26,9 & 27,0 \\ 26,7 & 26,4 & 26,4 & 26,6 & 26,7 \\ 25,6 & 25,4 & 25,6 & 26,0 & 26,4 \\ 25,2 & 25,0 & 25,2 & 25,7 & 26,1 \end{bmatrix}$$

- The last processing step involves transforming the matrix into a picture. Each value is rounded to the next integer:

$$\begin{bmatrix} 28 & 27 & 27 & 27 & 27 \\ 27 & 27 & 27 & 26 & 27 \\ 26 & 26 & 26 & 26 & 26 \\ 25 & 25 & 25 & 26 & 26 \\ 25 & 25 & 25 & 25 & 26 \end{bmatrix}$$

All values are in the range [25-28]. We assign to each value from the range a different color. There are four values in the range, so we need four colors. Each cell is transformed into a pixel of the corresponding color. A legend is added which of a border and numbers. The matrix has small size because of the practical nature of this case. The resulting image is therefore much smaller in reality. This means that we cannot add a real legend, so we represent it with single black and gray pixels. The (enlarged version) of the resulting image is:



In the rest of this subchapter we will show how the data products, transformations and propagation of the annotations can be modeled using the definitions from Chapters 3 – 6. The locality concept from Chapter 5 is specifically involved in this case.

7.2.2 Data structures

In this part of the chapter we give a description of all the data structures used in the workflow. We use the definitions of Chapter 3.

- **Data Structure 1:** The data structures used in the first processing step are created by the sources (the temperature sensors). The three sources all produce single values, which we model as vectors of size 1. The number of the vector is the number of the input source.

$$V[1] = [28,0] = (\{v_1^1\}, \{(1, v_1^1)\}, \{v_1^1, 28,0\})$$

$$V[2] = [27,0] = (\{v_1^2\}, \{(1, v_1^2)\}, \{v_1^2, 27,0\})$$

$$V[3] = [25,0] = (\{v_1^3\}, \{(1, v_1^3)\}, \{v_1^3, 25,0\})$$

- **Data Structure 2:** The gridding and joining of the three input vectors results in a matrix with empty cells:

$$M = \begin{bmatrix} 28,0 & \square & \square & \square & 27,0 \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & 25,0 & \square & \square & \square \end{bmatrix}$$

$$= (\{m_{5 \times (i-1)+j} \mid 1 \leq i, j \leq 5\}, \{(i, j, m_{5 \times (i-1)+j}) \mid 1 \leq i, j \leq 5\}, \{(m_1, 28,0), (m_5, 27,0), (m_{22}, 25,0)\})$$

- **Data Structure 3:** The completed matrix has the same structure as the input matrix:

$$M = \begin{bmatrix} 28,0 & 27,3 & 27,0 & 26,9 & 27,0 \\ 27,2 & 27,0 & 26,8 & 26,8 & 26,8 \\ 26,7 & 26,5 & 26,5 & 26,6 & 26,7 \\ 26,1 & 25,9 & 26,1 & 26,3 & 26,5 \\ 25,8 & 25,0 & 25,8 & 26,2 & 26,4 \end{bmatrix}$$

$$= (\{m_{5 \times (i-1)+j} \mid 1 \leq i, j \leq 5\}, \{(i, j, m_{5 \times (i-1)+j}) \mid 1 \leq i, j \leq 5\}, \lambda)$$

- **Data structure 4:** The final output data structure is a matrix that can be used to display a black-and-white image with a border and a legend on the screen. We represent the colors as integer values in the range [0..255]. The result is the matrix:

$$M = \begin{bmatrix} 180 & 0 & 242 & 242 & 216 & 216 & 216 \\ 180 & 0 & 242 & 216 & 216 & 216 & 216 \\ 180 & 0 & 216 & 165 & 165 & 216 & 216 \\ 180 & 0 & 165 & 127 & 165 & 165 & 165 \\ 180 & 0 & 127 & 127 & 127 & 165 & 165 \\ 180 & 0 & 0 & 0 & 0 & 0 & 0 \\ 180 & 180 & 180 & 180 & 180 & 180 & 180 \end{bmatrix}$$

$$= (\{m_{7 \times (i-1)+j} \mid 1 \leq i, j \leq 7\}, \{(i, j, m_{7 \times (i-1)+j}) \mid 1 \leq i, j \leq 7\}, \lambda)$$

7.2.3 Transformations

In this part we describe the three transformations that are performed in the temperature sensor workflow.

- **Transformation 1:** The first transformation involves placing the values of three size 1 input vectors at the appropriate position in a matrix. This transformation is a join as presented in the introduction of this chapter. There are three input vectors. Each vector contains only one element. Each element is placed at a fixed position in the output matrix. This position depends on the number of the input vector k . The position transformation function (DEFINITION 4.21) is given by:

$$T_P[k] = \begin{cases} (1,1), & k = 1 \\ (5,2), & k = 2 \\ (1,5), & k = 3 \end{cases}$$

Most of the elements in the output matrix are empty. For three elements there is a contributing input element, each from a different input source. The inverse position transformation function (DEFINITION 4.23) is given by:

$$T_P^{-1}(i,j) = \begin{cases} 1, & (i,j) = (1,1) \text{ or } (i,j) = (5,2) \text{ or } (i,j) = (1,5) \\ \text{undefined}, & \text{else} \end{cases}$$

The input source number depends on the position of the output element:

$$in_{\#}(e^{out}) = \begin{cases} 1, & F_M^{-1}(e^{out}) = (1,1) \\ 2, & F_M^{-1}(e^{out}) = (5,2) \\ 3, & F_M^{-1}(e^{out}) = (1,5) \\ \text{undefined}, & \text{else} \end{cases}$$

We use the input number function to find the input elements that contribute to an output element in a one-to-one n -ary transformation (DEFINITION 4.36):

$$S^{in}[e^{out}] = \{e^{in} \in S^{base}[k] \mid k = in_{\#}(e^{out}) \wedge e^{in} = F_V[k](T_P^{-1}(F_M^{-1'}(e^{out})))\}$$

We can now find for each output element its contributing input element. For example $S^{in}[m'_1]$. First we need to know the position of the output element: $F_M^{-1'}(m'_1) = (1,1)$. Using the inverse position transformation function we can find the position of the input element: $T_P^{-1}(1,1) = 1$. The input number of the contributing data structure is given by: $in_{\#}(m'_1) = 1$. The contributing input element can be found using: $F_V1 = v_1^1$.

The same calculations can be done for each output element. The following table shows four examples and the calculations for all output elements are given in Appendix B.2.

e^{out}	$F_M^{-1'}(e^{out}) = (i, j)$	$T_P^{-1}(i, j) = p$	$in_{\#}(e^{out}) = n$	$F_V[n](p)$
m'_1	$F_M^{-1'}(m_1) = (1, 1)$	$T_P^{-1}(1, 1) = 1$	$in_{\#}(m_1) = 1$	$F_V1 = v_1^1$
m'_5	$F_M^{-1'}(m_5) = (1, 5)$	$T_P^{-1}(1, 5) = 1$	$in_{\#}(m_5) = 2$	$F_V[2](1) = v_1^2$
m'_{22}	$F_M^{-1'}(m_{22}) = (5, 2)$	$T_P^{-1}(5, 2) = 1$	$in_{\#}(m_{22}) = 3$	$F_V[3](1) = v_3^1$
m'_{25}	$F_M^{-1'}(m_{25}) = (5, 5)$	$T_P^{-1}(5, 5) \text{ n/a}$	$in_{\#}(m_{25}) \text{ n/a}$	n/a

- **Transformation 2:** The second transformation involves interpolating the unknown values on the basis of the three known values. For the known values the contributing input element is the element at the same position in the input matrix. For each unknown value the set of contributing input elements is the same. This is reflected in the inverse position transformation function (DEFINITION 4.29):

$$T_{P++}^{-1}(i, j) = \begin{cases} \{(i, j)\}, & (i, j) = (1, 1) \text{ or } (i, j) = (5, 2) \text{ or } (i, j) = (1, 5) \\ \{(1, 1), (1, 5), (5, 2)\}, & \text{else} \end{cases}$$

We use DEFINITION 4.30 to find the contributing input elements:

$$S^{in}[e^{out}] = \{e^{in} \in S^{base} \mid F_M^{-1}(e^{in}) \in T_{P++}^{-1}(F_M^{-1'}(e^{out}))\}$$

The following table shows four examples (the complete listing can be found in Appendix B.2):

e^{out}	$F_M^{-1'}(e^{out}) = (i, j)$	$T_{P++}^{-1}(i, j)$	$S^{in}[e^{out}]$
m'_1	1,1	$\{(1,1)\}$	$\{m_1\}$
m'_4	1,4	$\{(1,1), (1,5), (5,2)\}$	$\{m_1, m_5, m_{22}\}$
m'_{22}	5,2	$\{(5,2)\}$	$\{m_{22}\}$
m'_{25}	5,5	$\{(1,1), (1,5), (5,2)\}$	$\{m_1, m_5, m_{22}\}$

- **Transformation 3:** The final transformation is for a large part data-only. The other part consists of moving all elements from the input to the right, and adding new elements to the left. This is reflected in the position transformation function and its inverse:

$$T_P(i, j) = (i, j + 2)$$

$$T_P^{-1}(i, j) = \begin{cases} (i, j - 2), & 1 \leq i \leq 5, 3 \leq j \leq 7 \\ \text{undefined}, & \text{else} \end{cases}$$

The inverse function is only defined for those elements that come directly from the input. Using DEFINITION 4.24 we can find the contributing input elements:

$$S^{in}[e^{out}] = \{e^{in} \in S^{base} \mid e^{in} = F_M \left(T_P^{-1} \left(F_M^{-1'}(e^{out}) \right) \right)\}$$

The following table shows for four elements what the contributing input elements are:

e^{out}	$F_M^{-1'}(e^{out}) = (i, j)$	$T_P^{-1}(i, j) = (p, q)$	e^{in}
m'_1	1,1	n/a	n/a
m'_5	1,5	1,3	m_3
m'_{17}	3,3	3,1	m_{11}
m'_{38}	6,3	n/a	n/a

In Chapter 4 we have listed five properties of data transformations. The following table lists these properties for each transformation. From this table it becomes clear that most transformations in the workflow have the same properties.

	T1	T2	T3
<i>Structure dependency</i>	S	S	S
<i>Data dependency</i>	-	D, S	D
<i>Element dependency</i>	\emptyset	-	S
<i>Element cardinality</i>	1: 1	*: 1	1: 1
<i>Relation cardinality</i>	1: 1	1: 1	1: 1
<i>Reversible</i>	Yes	No	Yes
<i>Invariance</i>	D	S	<i>none</i>
<i>Data/structure coherence</i>	Yes	No	Yes
<i>Transformation formula</i>	$T = \{T_S^S, T_E^\emptyset\}$	$T = \{T_S^S, T_D^{DS}\}$	$T = \{T_S^S, T_D^D, T_E^S\}$

7.2.4 Annotations

Here we assume that the temperature measurement produced by one of the sensors is annotated. For each sensor the annotation set is represented by $A[n]$. In this case we have $A[1] = \{v_1^1\}$, $A[2] = \{v_1^2\}$ and $A[3] = \emptyset$. We now show how after each transformation the annotations are propagated:

- **Transformation 1:** This transformation is one-to-one so we can use the annotation formula from Chapter 6.3.1:

$$w_a[e^{out}] = a \left[F_V[in_{\#}(e^{out})] \left(T_P^{-1} \left(F_M^{-1'}(e^{out}) \right) \right) \right]$$

$$\text{where } a[e^{in}] = \begin{cases} 1, & e^{in} \in A[in_{\#}(e^{out})] \\ 0, & e^{in} \notin A[in_{\#}(e^{out})] \end{cases}$$

We now need to find for each output the contributing input element and the annotation set of the contributing input data structure. The following table lists four examples:

e^{out}	e^{out}	n	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
m'_1	v_1^1	1	$A[1] = \{v_1^1\}$	1	true
m'_5	v_1^2	2	$A[2] = \{v_1^2\}$	1	true
m'_{22}	v_3^1	3	$A[3] = \emptyset$	0	false
m'_{25}	n/a	n/a	n/a	0	false

From the listing in Appendix B.2 we retrieve the output annotation set:

$$A' = \{m'_1, m'_{22}\}$$

- **Transformation 2:** This is the most complex transformation of the workflow. We want to propagate the annotations without having to implement the complete details of the transformation. Because we know that the influence of the contributing input elements is inversely proportional to the distance, we make use of the locality property. We use the annotation formula from Chapter 6.3.6 adapted for a matrix:

$$w_a[e^{out}] = \frac{1}{n} \sum_{e^{in} \in S^{in}[e^{out}]} w[e^{in}] * a[e^{in}]$$

$$\text{where } w[e^{in}] = l_E^M(e^{in}, e^{out}) = \frac{\emptyset(M) - d_E^M(e^{in}, e^{out})}{\emptyset(M)}$$

$$\text{and } \emptyset(M) = d_E((0,0), (5,5)) = 5\sqrt{2}; \quad d_E^M(e^{in}, e^{out}) = |F_M^{-1}(e^{in}) - F_M^{-1'}(e^{out})|$$

For the known values, the contributing element is only one element, which is the one at the same position in the input matrix. Above formula will then be basically the same as the one in Chapter 6.3.1:

$$w_a[e^{out}] = a \left[F_A \left(T_P^{-1} \left(F_A^{-1'}(e^{out}) \right) \right) \right]$$

In this case the output element is simply annotated if the contributing input element is annotated. For the unknown values the annotation formula depends on a fixed set of three contributing input elements: $\{m_1, m_5, m_{22}\}$. Using this set and $n = 3$ we get the following formula to calculate the annotation weight:

$$w_a[e^{out}] = \frac{1}{3} * (l(e^{out}, m_1) * a[m_1] + l(e^{out}, m_5) * a[m_5] + l(e^{out}, m_{22}) * a[m_{22}])$$

In this case we have $a[m_1] = 1$; $a[m_5] = 0$; $a[m_{22}] = 1$ so we get:

$$w_a[e^{out}] = \frac{1}{3} * (l(e^{out}, m_1) + l(e^{out}, m_{22}))$$

The following table lists four examples (the full listing can be found in Appendix B.2). We use here *threshold* = 0,40:

e^{out}	$S^{in}[e^{out}]$	$l(e^{out}, m_1)$	$l(e^{out}, m_5)$	$l(e^{out}, m_{22})$	$w_a[e^{out}]$	$e^{out} \in A$
m'_1	$\{m_1\}$	1,00	n/a	n/a	1,00	<i>true</i>
m'_3	$\{m_1, m_5, m_{22}\}$	0,72	0,72	0,42	0,38	<i>false</i>
m'_5	$\{m_5\}$	n/a	1,00	n/a	0,00	<i>false</i>
m'_{12}	$\{m_1, m_5, m_{22}\}$	0,68	0,49	0,72	0,47	<i>true</i>

From the listing in Appendix B.2 we retrieve the output annotation set:

$$A' = \{m'_1, m'_6, m'_7, m'_8, m'_{11}, m'_{12}, m'_{13}, m'_{16}, m'_{17}, m'_{18}, m'_{21}, m'_{22}, m'_{23}\}$$

If we put these results in a matrix the cells with a 1 are annotated and those with 0 are not:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Transformation 3: In the last transformation the input elements are shifted 2 places to the right. The values of these elements are transformed one-to-one. Finally there are extra elements added. These elements are added without any dependency on the input, so they will never be annotated. The other elements are simply annotated if their contributing input element is annotated. The following table lists four examples:

e^{out}	e^{in}	$a[e^{in}]$	$e^{out} \in A$
m'_1	n/a	0	<i>false</i>
m'_5	m_3	0	<i>false</i>
m'_{17}	m_{11}	1	<i>true</i>
m'_{38}	n/a	0	<i>false</i>

From the complete listing in Appendix B.2 we retrieve the output annotation set:

$$A' = \{m'_3, m'_4, m'_{10}, m'_{11}, m'_{12}, m'_{17}, m'_{18}, m'_{19}, m'_{24}, m'_{25}, m'_{26}, m'_{31}, m'_{32}, m'_{33}\}$$

If we put these results in a matrix the cells with a 1 are annotated and those with 0 are not:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

7.2.5 Conclusion

This use case proved to be more complex than the first use case. In this case the data structures are also of a similar type: vector and matrix. However, the transformations are more complex. The first transformation puts the data from the sensors in a matrix. In reality a gridding process is used to transform a geographic location into a position in the matrix. We have assumed that the sensors were fixed and that the data would always be put in the same cells, thereby discarding the gridding formula. Without this assumption, for example when the data included not only the temperature measurement but also the position of the sensor, the position in the matrix would not only be based on the input number, but also on the value of one of the elements, which is of the form $T_S^{D,S}$. For this type of transformations it is not possible to use the position transformation function to determine what the contributing input elements are and our annotation model would fail. A solution would be extending the model by taking the value into account.

The second transformation is also complex. Here we have assumed that the set of contributing input elements is fixed and equal for all output elements. In reality they might not be fixed as transformation 1 may cause the data from the sensors to be placed in different cells. In this case the contributing input elements can only be found by looking at the value of the cells (only non-empty cells contribute to the interpolated values), which results in the same problem type as transformation 1. Also it may be very well the case that the set of contributing input elements is not equal for all output elements; especially when the matrix is large and there are quite a few known values. When there are a lot of known values a selection must be made. Such a selection can be made by taking only those values within a certain distance. To measure a distance only the position of elements is needed and our annotation model still works.

The third transformation is one-to-one and the annotations can be propagated without difficulty. This transformation and the first one are one-to-one so it is not an issue what the threshold should be. The second transformation could produce annotation values in the whole range which made specifying what threshold to use more important. What a right threshold is, is not the focus of this thesis however.

With the assumptions we have made in this case we have shown that the annotation model is fully applicably. However, to make a more realistic case it would be better to take values into account in the annotation model, by considering the location of a sensor as a value of an element.

7.3 Bio-informatics: Cell-Graph Analysis

This use case mainly serves to show an example of a complex transformation which cannot be completely described using the definitions put forward in this thesis. Classification systems are used to determine automatically if pictures of breast tissue contain cancer cells [BDN+]. They process image data from tissue samples to create graphs of cells. Metrics are extracted from these graphs to determine if there are cancer cells present. We have adapted the methodology to create a use case. In some cases we have simplified the process, for example images of a small resolution, and in other case we have added details to complete the workflow.

7.3.1 Description

The Cell-Graph Analysis workflow consists of one source, five transformations and one sink. In between five data structures are transmitted. The workflow is visualized as follows:

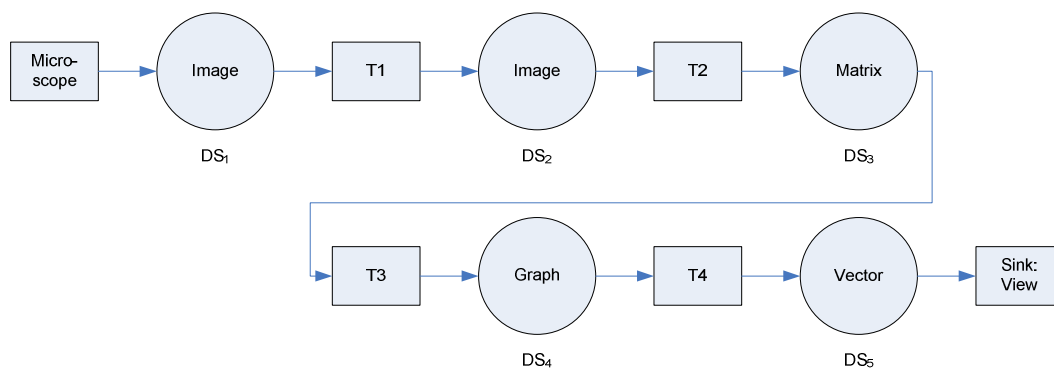


Figure 7.12: The workflow of a cell-graph analysis setup

The microscope produces an image. This image is converted to black-and-white by T1. A grid is used to group the pixels in the black-and-white image. T2 outputs a matrix where a value 1 is registered when the majority of the pixels in that grid cell is black. T3 transforms the matrix in a graph by clustering values in the matrix. Finally the graph is analyzed and T4 outputs a vector containing two metrics. We will describe each step in detail:

- The images generated by the microscope are 50x50 pixels. In reality these images might be much larger, but we choose a low resolution to keep things practical. We following images show the original and a blown-up version of the example that we use in this case:

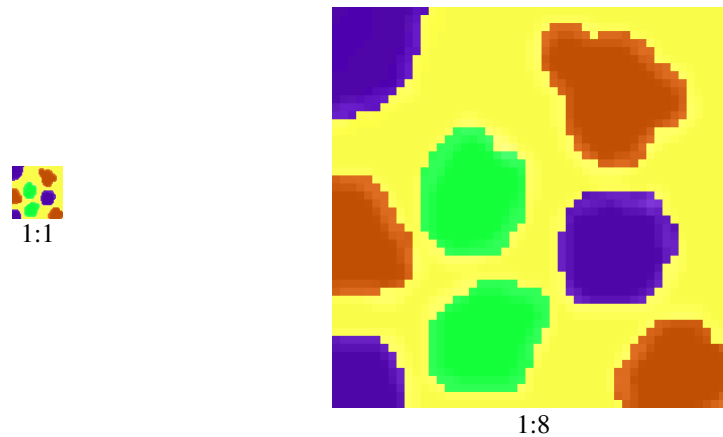


Figure 7.13: A cell tissue image

The purple, brown and green pixels represent cell-tissue material while the sandy pixels represent non-cell-tissue material.

- The color image is converted into a binary image consisting of only black and white pixels. An output pixel is colored black when the input pixel has a color that belongs to cell-tissue material. The following two images show the original size and a blown-up version of the output:



Figure 7.14: A cell tissue image in black-and-white

- A grid is used to group pixels in squares. The elements in the output matrix all correspond to a cell in the grid. The following figure shows the grid layout:

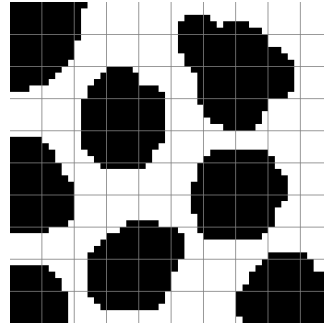


Figure 7.15: Application of a raster

When the majority of the pixels is colored black, the corresponding element in the matrix is assigned the value 1, otherwise the value 0. The following figure shows in red which grid cells contain a majority of black pixels:

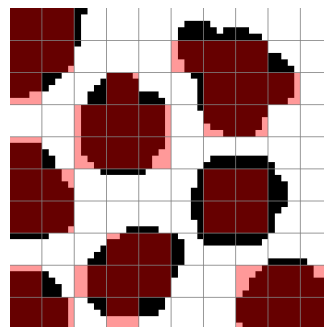


Figure 7.16: Counting pixels

The output matrix that is generated from the image is:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

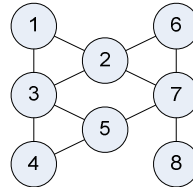
- The matrix is transformed into a graph. The nodes in a graph are formed by clustering elements from the matrix with value 1 that have Manhattan distance $d_M(e_1, e_2) = 1$ (DEFINITION 5.5). The following matrix shows which elements are grouped together:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 6 & 6 & 6 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 & 6 & 6 & 6 & 0 \\ 0 & 0 & 2 & 2 & 2 & 0 & 6 & 6 & 0 & 0 \\ 3 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 & 0 & 0 & 7 & 7 & 0 & 0 \\ 3 & 3 & 0 & 0 & 0 & 0 & 7 & 7 & 0 & 0 \\ 0 & 0 & 0 & 5 & 5 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 5 & 5 & 0 & 0 & 8 & 8 & 8 \\ 4 & 4 & 0 & 5 & 0 & 0 & 0 & 8 & 8 & 8 \end{bmatrix}$$

An edge is created between two nodes if one of the contributing input elements of the first node is within Chebyshev distance $d_C(e_1, e_2) = 2$ (Definition 5.8) from one of the contributing input elements of the second node. This is done by taking all the contributing input elements from one node, expanding it with all elements within Chebyshev distance 2. If the expanded the collection forms an overlap with contributing input elements of the second node, an edge is created. The *'s in the matrix below are within distance 2 of the contributing input elements of node 1. There is overlap with contributing input elements of node 2 and 3, so edges are created between node 1 and 2, and between node 1 and node 3.

$$\begin{bmatrix} 1 & 1 & * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & * & * & 0 & 6 & 6 & 6 & 0 & 0 \\ 1 & * & * & * & 0 & 0 & 6 & 6 & 6 & 0 \\ * & * & * & * & 2 & 0 & 6 & 6 & 0 & 0 \\ * & * & * & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 & 0 & 0 & 7 & 7 & 0 & 0 \\ 3 & 3 & 0 & 0 & 0 & 0 & 7 & 7 & 0 & 0 \\ 0 & 0 & 0 & 5 & 5 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 5 & 5 & 0 & 0 & 8 & 8 & 8 \\ 4 & 4 & 0 & 5 & 0 & 0 & 0 & 8 & 8 & 8 \end{bmatrix}$$

The final graph looks as follows:



- Finally two metrics are extracted from the graph. The number of nodes is calculated and the average degree of the nodes. In this case there are 8 nodes, and the average degree is: $\frac{1}{8} * (2 + 4 + 4 + 2 + 3 + 2 + 4 + 1) = 2\frac{3}{4}$. The final output is: $\begin{bmatrix} 8 & 2\frac{3}{4} \end{bmatrix}$

7.3.2 Data Structures

There are five data structures being used in the workflow. The first two data structures are images of 50x50 pixels. We represent them as matrixes here. The third is a 10x10 matrix, the fourth a graph, and the final output is a vector of size 2. We use the definitions from Chapter 3 to give a formal description for each of the five data structures.

- **Data structure 1:** The workflow starts with images generated by a digital microscope. In reality these images have a higher resolution, but to keep things practical for this use case we assume the images are 50 by 50 pixels.

$$M = (\{m_{50*(i-1)+j} | 1 \leq i, j \leq 50\}, \{(i, j, m_{50*(i-1)+j}) | 1 \leq i, j \leq 50\}, \lambda)$$

- **Data structure 2:** The images produced by the digital microscope are colored. The first transformation converts the color image into a binary image, consisting only of purely black and white pixels. The output data structure is of the exact same size as the input.

$$M = (\{m_{50*(i-1)+j} | 1 \leq i, j \leq 50\}, \{(i, j, m_{50*(i-1)+j}) | 1 \leq i, j \leq 50\}, \lambda)$$

- **Data structure 3:** A grid is used to compartmentalize the binary image in 10x10 squares of 5x5 pixels each. When the majority of the pixels in a square is black, a 1 is assign to the corresponding cell in the output matrix, otherwise a 0. The output matrix corresponds to the squares in the grid and is a 10x10 matrix.

$$M = (\{m_{10*(i-1)+j} | 1 \leq i, j \leq 10\}, \{(i, j, m_{10*(i-1)+j}) | 1 \leq i, j \leq 10\}, \lambda)$$

- **Data structure 4:** A graph is constructed on the basis of the 0-1 matrix. The output graph contains in this particular case eight nodes and eleven edges. The size of the output structure depends on the values in the input structure, and may therefore vary. The nodes have no value, that's why the value function is replaced by the empty set. The graph for this particular transformation is denoted by:

$$\begin{aligned} G &= (\{v_i | 1 \leq i \leq 8\}, \\ &\{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_6\}, \{v_2, v_7\}, \{v_6, v_7\}, \{v_3, v_5\}, \{v_3, v_4\}, \\ &\{v_4, v_5\}, \{v_5, v_7\}, \{v_7, v_8\}\}, \emptyset) \end{aligned}$$

- **Data structure 5:** In the final transformation metrics are extracted from the graph. In this case we consider only two metrics: the number of nodes and the average degree of a node. The metrics are presented in a 2-size vector. The particular graph in this case contains 8 nodes and the average node degree is $2\frac{3}{4}$.

$$V = \left(\{v_1, v_2\}, \{(1, v_1), (2, v_2)\}, \left\{ (v_1, 8), \left(v_2, 2\frac{3}{4} \right) \right\} \right)$$

The following table gives an overview of the properties of the five data structures:

	DS₁	DS₂	DS₃	DS₄	DS₅
<i>Type</i>	matrix	matrix	matrix	Graph	vector
<i>Size</i>	50 × 50	50 × 50	10 × 10	<i>variable</i>	2
<i>Ordering</i>	ordered	ordered	ordered	unordered	ordered
<i>Dimension</i>	2	2	2	n/a	1
<i>Relation based</i>	n/a	n/a	n/a	yes	n/a
<i>Hierarchical</i>	n/a	n/a	n/a	no	n/a
<i>Element-relation card.</i>	1: 1	1: 1	1: 1	1:*	1: 1

7.3.3 Transformations

In this part we describe the processing elements that perform a transformation on the data structures. There are four processing elements of the transformation type. We use the definitions of Chapter 4 to describe the relation between input and output elements. These descriptions will in the next part of the chapter be used to show how annotations can be propagated.

- **Transformation 1:** The first transformation is data-only. It assigns 1's to pixels with a color that belong to cell tissue material. To non-cell tissue material a 0 is assigned. Essentially the color image is transformed into a black-and-white image of the same size. This has the effect that the position of output elements is exactly the same as their contributing input element:

$$T_P^{-1}(i, j) = (i, j)$$

We can now use DEFINITION 4.24 to find the contributing input elements:

$$S^{in}[e^{out}] = \{e^{in} \in S^{base} \mid e^{in} = F_M \left(T_P^{-1} \left(F_M^{-1'}(e^{out}) \right) \right)\}$$

For example $S^{in}[m'_{35}]: F_M^{-1'}(m'_{35}) = (1, 3); T_P^{-1}(1, 3) = (1, 3); F_M(1, 3) = m_{35}$.

- **Transformation 2:** The second transformation uses a grid to compartmentalize the image in 10x10 squares of 5x5 pixels. Each square relates to one element in the output matrix. If the majority of the pixels in the square is black, a 1 is assigned to the value of the element, otherwise a 0 is assigned. This is a many-to-one transformation and the inverse position transformation function is (DEFINITION 4.29):

$$T_{p++}^{-1}(i, j) = \{(5 * (i - 1) + p, 5 * (j - 1) + q) | 1 \leq p, q \leq 5\}$$

We use DEFINITION 4.30 to find the contributing input elements:

$$S^{in}[e^{out}] = \{e^{in} \in S^{base} \mid F_M^{-1}(e^{in}) \in T_{p++}^{-1}(F_M^{-1'}(e^{out}))\}$$

The contributing input elements for the grid cell (4, 2) is given by $S^{in}[m'_{32}]$. First we need to find the inverse positions:

$$T_{p++}^{-1}(4, 2) = \left\{ \begin{array}{l} (16, 6), (16, 7), (16, 8), (16, 9), (16, 10), \\ (17, 6), (17, 7), (17, 8), (17, 9), (17, 10), \\ (18, 6), (18, 7), (18, 8), (18, 9), (18, 10), \\ (19, 6), (19, 7), (19, 8), (19, 9), (19, 10), \\ (20, 6), (20, 7), (20, 8), (20, 9), (20, 10) \end{array} \right\}$$

Using the position function we can now find the contributing input elements:

$$S^{in}[m'_{32}] = \left\{ \begin{array}{l} m_{756}, m_{757}, m_{758}, m_{759}, m_{760} \\ m_{806}, m_{807}, m_{808}, m_{809}, m_{810} \\ m_{856}, m_{857}, m_{858}, m_{859}, m_{860} \\ m_{906}, m_{907}, m_{908}, m_{909}, m_{910} \\ m_{956}, m_{957}, m_{958}, m_{959}, m_{960} \end{array} \right\}$$

- **Transformation 3:** The third transformation creates a graph. Neighboring 1's in the matrix are clustered, and for each cluster a node is added to the graph. When 1's in a cluster are within a distance of 2 from 1's in the other cluster an edge is added between the two corresponding nodes. The transformation is many-to-one as neighboring elements from the input matrix with value 1 are clustered to create one node in the graph. This means that the creation of the output structure depends on both the input structure and data. For this type of transformation it is not possible to define a position transformation function that can be used to find the contributing input elements. For example, the set of contributing input elements of node v_1 is: $S^{in}[v'_1] = \{m_1, m_2, m_{11}, m_{12}\}$.

The following table lists all the contributing input elements for the elements of the graph in this particular case:

e^{out}	$S^{in}[e^{out}]$
v'_1	$\{m_1, m_2, m_{11}, m_{12}\}$
v'_2	$\{m_{24}, m_{33}, m_{34}, m_{35}, m_{43}, m_{44}, m_{45}\}$
v'_3	$\{m_{41}, m_{51}, m_{52}, m_{61}, m_{62}\}$
v'_4	$\{m_{81}, m_{91}, m_{92}\}$
v'_5	$\{m_{74}, m_{75}, m_{83}, m_{84}, m_{85}, m_{94}\}$
v'_6	$\{m_{16}, m_{17}, m_{18}, m_{27}, m_{28}, m_{29}, m_{37}, m_{38}\}$
v'_7	$\{m_{57}, m_{58}, m_{67}, m_{68}\}$
v'_8	$\{m_{88}, m_{89}, m_{90}, m_{98}, m_{99}, m_{100}\}$

- **Transformation 4:** The final transformation outputs a vector that is always of size 2. The output structure does not depend on the input and is embedded in the transformation. The output data is only depended on the input structure. The transformation is many-to-many where all input elements contribute to each of the output elements.

$$S^{in}[e^{out}] = S^{base}$$

In Chapter 4 we list five properties of data transformations. The following table lists these properties for each transformation.

	T1	T2	T3	T4
Structure dependency	S	S	D, S	\emptyset
Data dependency	D	D, S	D, S	D
Element dependency	-	S	D, S	\emptyset
Element cardinality	1:1	1:*	1:*	1:*
Relation cardinality	1:1	1:*	1:*	1:*
Reversible	No	No	No	No
Invariance	S	none	D	D
Transformation formula	$T = \{T_D^D, T_S^S\}$	$T = \{T_D^{DS}, T_E^S, T_S^S\}$	$T = \{T_D^{DS}, T_E^{DS}, T_S^{DS}\}$	$T = \{T_D^S, T_E^\emptyset, T_S^\emptyset\}$

7.3.4 Annotations

Here we assume that part of the original image created by the microscope is annotated. The whole region from pixel (18, 7) to (29, 20) is annotated.

$$A = \{m_{50*(i-1)+j} | 18 \leq i \leq 32, 7 \leq j \leq 20\}$$

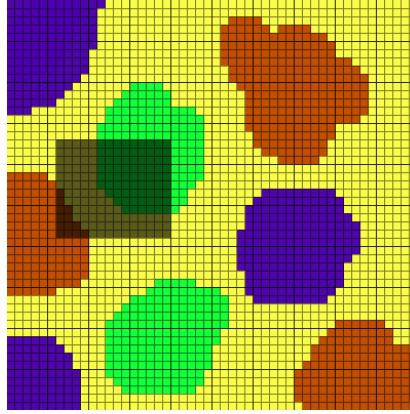


Figure 7.17: Annotation in a cell-tissue image

- **Transformation 1:** The first transformation is data only. For one-to-one transformations we can use the annotation formula from Chapter 6.3.2:

$$w_a[e^{out}] = a \left[F_M \left(T_P^{-1} \left(F_M^{-1'}(e^{out}) \right) \right) \right]$$

For each of the elements in the output we need to check if its contributing input element was annotated. There are 2500 elements in the data structure, so we give one example: m'_{35} .

$F_M^{-1'}(m'_{35}) = (1,35); T_P^{-1}(1,5) = (1,5); F_M(1,5) = m_{35}; a[m_{35}] = 0$; so $m'_{35} \notin A$. The complete annotation set for the output is given by:

$$A' = \{m'_{50*(i-1)+j} | 18 \leq i \leq 32, 7 \leq j \leq 20\}$$

- **Transformation 2:** The second transformation is many-to-one. In this case we annotate an element if the majority of the contributing input elements are annotated. This is done using the formula from Chapter 6.3.3:

$$w_a[e^{out}] = \frac{1}{n} \sum_{e^{in} \in S^{in}[e^{out}]} a[e^{in}]$$

There are 100 elements in the output. We calculate the annotation value for a few elements as an example. In the previous part of this chapter we have calculated what the set of contributing input elements for m'_{32} was:

$$S^{in}[m'_{32}] = \begin{Bmatrix} v_{756}, v_{757}, v_{758}, v_{759}, v_{760}, \\ v_{806}, v_{807}, v_{808}, v_{809}, v_{810}, \\ v_{856}, v_{857}, v_{858}, v_{859}, v_{860}, \\ v_{906}, v_{907}, v_{908}, v_{909}, v_{910}, \\ v_{956}, v_{957}, v_{958}, v_{959}, v_{960} \end{Bmatrix}$$

Twelve of these elements are annotated:

$$v_{857}, v_{858}, v_{859}, v_{860}, v_{907}, v_{908}, v_{909}, v_{910}, v_{957}, v_{958}, v_{959}, v_{960} \in A$$

And thirteen are not:

$$v_{756}, v_{757}, v_{758}, v_{759}, v_{760}, v_{806}, v_{807}, v_{808}, v_{809}, v_{810}, v_{906}, v_{956} \notin A$$

Therefore we have:

$$w_a[m'_{32}] = \frac{1}{n} (12 * 1 + 13 * 0) = \frac{12}{25}$$

We set the threshold of $threshold = \frac{1}{2}$ and m'_{32} is not annotated. There are six grid cells, that contain annotated pixels. For each of the corresponding output elements we count the number of annotated pixels. When thirteen or more pixels are annotated the output element is also annotated.

e^{out}	Annotated pixels	$e^{out} \in A$
m'_{33}	15	true
m'_{34}	15	true
m'_{42}	20	true
m'_{43}	25	true
m'_{44}	25	true
m'_{52}	20	true
m'_{53}	25	true
m'_{54}	25	true
m'_{62}	8	false
m'_{63}	10	false
m'_{64}	10	false

It is too exhaustive to give a calculation for all the output elements here, but neither of the output elements not listed above is annotated. The output annotation set is:

$$A' = \{m'_{33}, m'_{34}, m'_{42}, m'_{43}, m'_{44}, m'_{52}, m'_{53}, m'_{54}\}$$

- **Transformation 3:** In the previous part of this chapter we have seen that it is not possible to find the set of contributing input elements using a mathematical formula. This means that the option we have is annotating the whole data structure. We basically then have a many-to-one transformation where all input elements contribute to one single output. Also in such a case a decision needs to be made? Do we want to annotate the whole output structure even if only a few of the input elements are annotated? In this case we annotate the whole output

structure if more than 5% of its input was annotated. We calculate this as follows:

$$\left(\frac{|A|}{|S^{base}|} = \frac{8}{100} = 0,07 \right) \geq 0,05$$

When we would have a method for finding the contributing input elements, it would be possible to calculate the annotation values for the output. In the previous part we have listed the contributing input elements for the graph in the particular instance of the case. We use this to calculate the annotation values:

e^{out}	$S^{in}[e^{out}]$	$ S^{in}[e^{out}] \cap A $	$ S^{in}[e^{out}] $	$e^{out} \in A$
v'_1	$\{m_1, m_2, m_{11}, m_{12}\}$	0	4	$\left(\frac{0}{4} \geq 0,5\right) = false$
v'_2	$\{m_{24}, m_{33}, m_{34}, m_{35}, m_{44}, m_{45}\}$	4	7	$\left(\frac{4}{7} \geq 0,5\right) = true$
v'_3	$\{m_{41}, m_{51}, m_{52}, m_{61}, m_{62}\}$	1	5	$\left(\frac{1}{5} \geq 0,5\right) = false$
v'_4	$\{m_{81}, m_{91}, m_{92}\}$	0	4	$\left(\frac{0}{4} \geq 0,5\right) = false$
v'_5	$\{m_{74}, m_{75}, m_{83}, m_{84}, m_{94}\}$	0	6	$\left(\frac{0}{6} \geq 0,5\right) = false$
v'_6	$\{m_{16}, m_{17}, m_{18}, m_{27}, m_{29}, m_{37}, m_{38}\}$	0	8	$\left(\frac{0}{8} \geq 0,5\right) = false$
v'_7	$\{m_{57}, m_{58}, m_{67}, m_{68}\}$	0	4	$\left(\frac{0}{4} \geq 0,5\right) = false$
v'_8	$\{m_{88}, m_{89}, m_{90}, m_{98}, m_{100}\}$	0	6	$\left(\frac{0}{6} \geq 0,5\right) = false$

The output annotation set now would be $A = \{v'_2\}$.

- **Transformation 4:** When in the previous transformation the data structure as a whole would be annotated, we can now make no other choice than to annotate the whole data structure again. Even though for this transformation we can find the contributing input elements using a formula, the information of annotations on the element level is lost.

When we consider the annotation set $A = \{v'_2\}$ that we would have if in the previous transformation when we would be able to identify the contributing input elements, we can also calculate the annotations for the output elements. Both output elements depend on the all of the input elements. We annotate the output elements when more than 10% of the input elements is annotated:

$$w_a[e^{out}] = \frac{|A|}{|S^{base}|}$$

$$w_a[v'_1] = w_a[v'_2] = \frac{1}{8} \geq 0,10$$

This means that $A' = \{v'_1, v'_2\}$

7.3.5 Conclusion

The final case is the most complex of the three. Where in the first two use cases we had similar data types, we have now very different types: vector, matrix, image and graph, of which the last is most different from the others. Also we had transformations of very different type. The first was a data-only, which poses no problem for our annotation model. The second was many-to-one and quite a large amount of elements were transformed into a single new element. The annotation model worked fine for this transformation. The fourth transformation also didn't cause problems when applying the annotation model.

It was the third transformation that caused the most problems. Here we did not have a function to find the contributing input elements, and the only option we had was propagating the annotations to the whole data structure. When using this option we needed to define a specific threshold. Again, this was not the aim of this thesis, but we must point out that picking the right threshold is of importance to the propagation of the annotations in this case.

Only when we made a very strong assumption, namely that we did know what the contributing elements were, it was possible to propagate annotations to elements. In practice however, images will be very different from another, resulting in very different graphs. Therefore we can only conclude that the annotation model is not sufficient to have useful propagations in the third transformations.

7.4 Conclusion

In this chapter we have applied the definitions from this thesis to three use cases. For each of the cases we have described the successful applications and the limitations of the annotation model. We will briefly summarize the most important conclusions.

The annotation model consists basically of three parts: finding the right contributing input elements, assigning a weight to them, and comparing the normalized result to a threshold.

Finding the right contributing input elements worked fine for most transformations, but proved difficult for three of them. In these cases the value was necessary to find the contributing input elements. Therefore the annotation model failed, unless we made strong assumptions. Extending the annotation model to take also the value into consideration would be a solution.

Assigning a weight was in none of the transformations a problem. For most of the transformations it was a simple matter of counting what contributing input elements were annotated by assigning each a weight of one. In the transformation that interpolated values in a matrix, assigning a weight was more difficult, but applying the locality property proved successful.

Most of the transformations were either one-to-one or many-to-one. For a one-to-one transformation it was not necessary to pick a specific threshold and an output element was simply annotated when its contributing input element was annotated. For most many-to-one transformations it was only a matter of counting how many contributing input elements were annotated. A threshold of 50% was chosen which is an arbitrary choice. For some other many-to-one transformations and the many-to-many transformations, calculating the annotation value was more complex and picking the right threshold was even more important. As this was not the aim of this thesis we have chosen arbitrary values, but it needs to be stressed that in practice picking the right threshold is important.

Chapter 8

Conclusion

In this chapter we will reflect on the research questions presented in the introduction. At the end of this chapter we will discuss potential future work.

8.1 Reflection on research questions

In the introduction we have formulated four research questions. We will now reflect on each of these research questions separately. After reflecting on the four research questions we will reflect on the overall problem statement.

RESEARCH QUESTION 1 How can the description of data structures be formalized?

This research question is addressed in Chapter 3. We have presented a formal description of a data structure and we have shown how this could be applied to six elementary data structures. The key result of this formalization is the separation of the elements, their values and the structure. This way a bridge is created between mathematical abstractions and practical use in computer science applications. We have chosen six elementary data structures: sets, vectors, matrixes, arrays, graphs and trees. We have shown that these elementary data structures can be used to construct two other types of data structures: tables and images. Finally we have described four properties to classify the elementary data structures. These four properties are all specific to the relation on the set of elements for each data structure and proved sufficient to make a classification.

RESEARCH QUESTION 2 How can the description of a transformation of data structure and the effects of that transformation be formalized?

In Chapter 4 we have addressed this research question. We have used the definitions from Chapter 3 to investigate how a formal definition of a transformation of a data structure can be given. First we have defined two aspects of a transformation. A data structure consists of a data part and a structure part. We have defined the two aspects as one that produces the data part and the other that produces the structure part.

We have then proceeded by showing that the production of any data structure can be described by a sequence of elementary operations. Giving such a description is too cumbersome so we have investigated other ways to describe aspects of a transformation. We have formulated properties that describe various aspects of a transformation on a higher level than elementary operations. We have used these properties to distinguish a special class for which we could describe the effect of the transformation by describing the transformation of the position of the elements. To describe a transformation using this method the transformation must be such that, to produce the output structure, only the input structure is needed. These transformations are called structure-from-structure transformations. Also the transformation must be uniform: when an element at a certain position is used to produce an element at another position, also the value of the output element must be produced using only the value of the contributing input element.

For these types of transformations we have shown that it is possible to link contributing input elements to an output element by a transformation of the position. We have shown that it is sufficient to specify only the structure of the input and output data structures, and a position transformation function. This is much less information than describing the full elementary operations algorithm. The results are used to address RESEARCH QUESTION 4, where we will see that this is sufficient information to propagate annotations in certain scenarios.

RESEARCH QUESTION 3 How can locality be used to describe the effect of a transformation on the relation between elements in a data structure?

In some cases it may be enough to have a formal description of the transformation of the structure part. In other cases more information may be needed. Locality is a distance-based property that provides such information and is introduced in Chapter 5. First we have presented a formal definition of distance. We have given examples of four different ways of measuring distance between elements in different data structures. We have used the definition of distance to introduce a formal definition of locality. Locality is based on the distance between two elements and the diameter of the data structure that contains the elements. Finally we have shown what the possible effects are of a transformation on the locality of two elements.

Using the results from RESEARCH QUESTION 1,2 and 3 we can now address RESEARCH QUESTION 4.

RESEARCH QUESTION 4 How can annotations in a data structure be propagated?

This RESEARCH QUESTION is addressed in Chapter 4. In order to understand how annotations could be propagated, it necessary to know annotations can be made in a data structure. We have shown for each type of elementary data structure how annotations can be made: at the level of elements, relations, values and the data structure as a whole.

In the second part of this chapter we have presented a model for transforming annotations at the element-level. When the transformation is of the structure-from-structure type, the annotations can be propagated using the position transformation function. We have shown for five different scenarios how the results from Chapter 4 can be used in the model presented in this chapter. In the first scenario there is only one contributing input element and the output element is simply annotated when the contributing input element is annotated. In the second scenario there is more than one contributing input element and the output is annotated when more than a certain share of the contributing input elements is annotated. In the third case the contributing input elements are given a weight of 1 or 2. In the fourth scenario the weight depends on the distance between contributing input element and output element. The last scenario shows how in addition to the results from Chapter 4, the results from Chapter 5 can be used in the annotation transformation model presented in this chapter. In this scenario the weight depends on the locality of the contributing input element and output element.

The results of each of these research questions can now be used to address the overall problem statement.

PROBLEM STATEMENT How can annotations in a stream processing system be propagated?

The results of the four research questions have been validated by applying them to three use cases in Chapter 7. The results of this chapter form an answer to the general problem statement. We repeat here the most important conclusions. The annotation model consists basically of three parts: finding the right contributing input elements, assigning a weight to them, and comparing the normalized result to a threshold. Picking the right threshold was not the aim of this thesis. So the results of this thesis hinge on the other two aspects. Assigning weights was fully possible for all transformations. In a more complex transformation the locality property was successfully applied as a weight. Finding the contributing input elements worked fine in most cases. Only in a few cases it was only possible when assumptions were made. Without these assumptions the value

of an input element needs to be taken into account, which is not possible in our model. For the first use case we could apply the annotation model without any problems. For the second use case we could apply the annotation model when certain assumptions were made. However, these assumptions still provided a valid use case. Only in the third use case the assumptions that needed to be made were so strong that the use case would be not valid anymore.

8.2 Future research

Our thesis introduces many new concepts, but with each answer comes new questions. We present here a few topics based on this research that could be further investigated.

We have presented a formal definition of a data structure. One part of this definition is a relation on the elements. For each type of elementary data structure we have provided a very specific relation. For vectors, matrixes and arrays this relation could be modeled as a function that maps a natural number to an element. In graphs and trees the relation consists of edges. These are very different concepts, but they all provide a certain way of organizing the elements. It would be interesting to investigate this part more to generalize the concept of the relation. The field of topology organizes elements using metrics. We have used this concept to define our locality property. Topology might serve as a good candidate to provide a means for generalizing the concept of the relation in a data structure.

In our annotation model we have discarded the value of an element. Although it worked fine for many transformations, it would be interesting to investigate whether there's a method for extending the annotation model to values. This can be done at two points. The first point is extending the function that finds the contributing input elements by taking the value into account. This could be done using a predicate logic. The second point is to consider the weights. The weights are now only based on the position. Certain values may be more important than others and this may influence the decision of propagating annotations.

We have discussed only simple joins where data structures were glued together. In practice there are many more n-ary transformations thinkable and it is worth while to investigate these. This can be done in two ways: data structures within one stream may be accumulated and transformed into one new data structure, or data structure from different streams may be combined.

As discussed before we have chosen sometimes arbitrary thresholds. However it may be very important to pick a right threshold. This may be done by self-learning algorithms, adjusting the threshold over time, or empirical research could be done to investigate what the right thresholds are in certain scenarios.

The annotation model calculations an annotation value for each output element and then decides what whether or not that output element should be annotated.

Accumulative effects are ignored however. A next processing element simply sees that element annotated in its input. In the current model it is not possible to say that an element is for x% annotated.

We have only considered binary annotations. An element was simply annotated or it was not. In practice, the annotations usually have certain values, for example an element may be marked as “weak”. Different values may have different meanings. The semantics of such annotations is now completely ignored. Investigating what the effect would be of combining annotations of different semantics may worth investigating.

References

- [AWK10] Annotations: Dynamic Semantics in Stream Processing. 2010. J.Amiguet, A.Wombacher, T.E.Klifman. Presented at PIKM'10.
- [BD+07] Marion Blount, John Davis, Archan Misra, Daby Sow and Min Wang. 2007. "A Time-and-Value Centric Provenance Model and Architecture for Medical Event Streams". In *Proceedings of the 1st ACM SIGMOBILE International Workshop on Systems and Networking Support for Healthcare and Assisted Living Environments*, HealthNet 2007, pp. 95-100.
- [BDN+07] Cell-Graph Mining for Breast Tissue Modeling and Classification. 2007. Cagatay Bilgin, Cigdem Demir, Chandandeep Nagi, Bulent Yener. Proceedings of the 29th Annual International Conference of the IEEE EMBS. Cité Internationale, Lyon, France. August 23-26, 2007.
- [BKT01] Peter Buneman, Sanjeev Khanna and Wang-Chiew Tan. 2001. "Why and Where: A Characterization of Data Provenance". In *Lecture Notes in Computer Science*, volume 1973, International Conference on Database Theory, ICDT 2001, pp. 316-330.
- [DCJ+03] Sensor Web in Antarctica: Developing an intelligent, autonomous platform for locating biological flourishes in cryogenic environments. 2003. K.A. Delin, R.P. Harvey, N.A. Chabot, S.P. Jackson, Mike Adams, D.W. Johnson, and J.T. Britton. Presented at the 34th Lunar and Planetary Science Conference, 17-21 March 2003, Houston, TX.
- [FG+09] Joe Futrelle, Jeff Gaynor, Joel Plutchak, James D. Myers, Robert E. McGrath, Peter Bajcsy, Jason Kastner, Kailash Kotwani, Jong Sung Lee, Luigi Marini, Rob Kooper, Terry McLaren and Yong Liu. 2009. "Semantic Middleware for E-science Knowledge Spaces". In *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC 2009, pp. 1-6.
- [Fre08] Earth remote sensing use cases for SciDB. 2008. James Frew.
<http://trac.scidb.org/raw-attachment/wiki/UseCases/Earth%20remote%20sensing%20use%20cases%20for%20SciDB.pdf>
- [HDF08] HDF-EOS Interface Based on HDF5, Volume 1: Overview and Examples.
<http://edhs1.gsfc.nasa.gov/waisdata/rel7/pdf/175emd001r5.pdf>

- [HM06] Environmental Sensor Networks: A revolution in the earth system science? 2006. Jane K. Hart, Kirk Martinez. *Earth-Science Reviews* 78 (2006) 177 – 191, Elsevier.
- [Jav11] Oracle. Java documentation. Accessed on October 18th, 2011.
<http://download.oracle.com/javase/1.4.2/docs/guide/collections/overview.html>
- [LMB09] Hyo-Sang Lim, Yang-Sae Moon and Elisa Bertino. 2009. "Research Issues in Data Provenance for Streaming Environments". In *Proceedings of the 2nd SIGSPATIAL ACM GIS 2009 International Workshop on Security and Privacy in GIS and LBS*, SPRINGL '09, pp. 58-62.
- [LN+05] Jonathan Ledlie, Chaki Ng, David A. Holland, Kiran-Kumar Muniswamy-Reddy, Uri Braun, Margo Seltzer. 2005. "Provenance-Aware Sensor Data Storage". In *21st International Conference on Data Engineering Workshops*, ICDEW 2005, pp.1189.
- [MF+08] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. McGrath, Jim Myers and Patrick Paulson. 2008. The Open Provenance Model: An Overview. In *Lecture Notes in Computer Science*, volume 5272, Second International Provenance and Annotation Workshop, IPAW 2008, pp. 323–326.
- [Nasa11] Landsat 7 Science Data Users Handbook. 2011. Nasa.
http://landsathandbook.gsfc.nasa.gov/pdfs/Landsat7_Handbook.pdf
- [Opm09] Open Provenance Model, specification 1.1. 2009. Accessed on July 13th, 2010.
<http://openprovenance.org>
- [Pcw11] Provenance Challenge Wiki. Accessed on July 13th, 2010.
<http://twiki.ipaw.info/bin/view/Challenge/WebHome>
- [Pig11] W3C Provenance Incubator Group Wiki. Accessed on October 18th, 2011.
http://www.w3.org/2005/Incubator/prov/wiki/W3C_Provenance_Incubator_Group_Wiki
- [Pwg11] Provenance Working Group Website. Accessed on October 18th, 2011.
http://www.w3.org/2011/prov/wiki/Main_Page
- [Sci11] Overview of SciDB. 2010. The SciDB Development Team. SIGMOD '10 June 6-11, Indianapolis, Indiana, USA.
- [She68] A two-dimensional interpolation function for irregularly-spaced data. 1968. Donald Sheppard. *Proceedings – 1968 ACM National Conference*.

- [SM03] Martin Szomszor and Luc Moreau. 2003. "Recording and Reasoning over Data Provenance in Web and Grid Services." In *Lecture Notes in Computer Science*, volume 2888, On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, pp. 603–620.
- [SPG05] Yogesh L. Simmhan, Beth Plale and Dennis Gannon. 2005. "A Survey of Data Provenance in e-Science". In *SIGMOD Record*, volume 34, number 3, Sept. 2005, pp. 31-36.
- [SPG08] Yogesh L. Simmhan, Beth Plale and Dennis Gannon. 2008. "Karma2: Provenance Management for Data Driven Workflows". In *International Journal of Web Services Research*, volume 5, number 2, pp. 1-22.
- [Wid08] Jenifer Widom. 2008. "Trio: A System for Data, Uncertainty, and Lineage". In *Managing and Mining Uncertain Data*. ISBN: 978-0-387-09689-6. Chapter 5, pp. 113-148.
- [ZWF06] Yong Zhao, Michael Wilde, and Ian Foster. 2006. "Applying the Virtual Data Provenance Model". In *Lecture Notes in Computer Science*, volume 4145, International Provenance and Annotation Workshop, IPAW 2006, pp. 148 – 161.

Appendix A Symbol Chart

Symbol	Definition	Description
e_i	3.1	A unique element
\mathcal{S}^{base}	3.2	The base set; collection of elements
R	3.3	Relation
\mathcal{S}^{value}	3.4	The value set
λ	3.4	The value function
\mathbb{S}	3.5	A data structure in general
S	3.7	A set data structure
v_i	3.8	A unique element of a vector
F_V	3.9-I	The vector function; returns the element at a position
F_V^{-1}	3.9-II	The inverse vector function: returns the position of an element
V	3.10	A vector data structure
m_i	3.12	A unique element of a matrix
F_M	3.13-I	The matrix function
F_M^{-1}	3.13-II	The inverse matrix function
M	3.14	A matrix data structure
d	3.15	Dimension
a_i	3.16	A unique element of an Array
F_A	3.17-I	The array function
F_A^{-1}	3.17-II	The inverse array function
v_i	3.19	A node of a graph
V	3.19	The set of nodes of a graph
E	3.20	The edge set of a graph
G	3.21	A graph data structure
r	3.22	The root of a tree
T	3.23	A tree data structure
T	4.1	A transformation
T_E	4.2	The effect of a transformation on the elements
T_S	4.3	The effect of a transformation on the structure
T_D	4.4	The effect of a transformation on the data
$\mathcal{S}^{in}[e^{out}]$	4.16	The set of input elements that contribute to an output element
$\mathcal{S}^{out}[e^{in}]$	4.17	The set of output elements to which an input element contributes
T_P	4.18	The position transformation function
T_P^{-1}	4.20	The inverse position transformation function
T_{P++}	4.24	The position transformation function for many-to-many transformations
T_{P++}^{-1}	4.26	The inverse position transformation function for many-to-many transformations
$in_{\#}[e^{out}]$	4.31	The input source number of the data structure that contributed to the output element
d_E^M	5.1	Euclidean distance in a Matrix
d_E^V	5.2	Euclidean distance in a Vector
d_E^A	5.3	Euclidean distance in an Array
d_M^V	5.4	Manhattan distance in a Vector
d_M^M	5.5	Manhattan distance in a Matrix
d_M^A	5.6	Manhattan distance in an Array

Symbol	Definition	Description
d_C^V	5.7	Chebyshov distance in a Vector
d_C^M	5.8	Chebyshov distance in a Matrix
d_C^A	5.9	Chebyshov distance in an Array
$l(e_1, e_2)$	5.10	The locality of two elements in a data structure
$\mathcal{O}(\mathcal{S})$	5.11	The diameter of a data structure
\mathcal{S}^*	6.1	An annotated data structure
T^*	6.2	The transformation of an annotated data structure
$e \in A$	6.3	The element e is annotated
T_A	6.4	The transformation of the annotations
$w_a[e^{out}]$	6.5	The annotation weight, if it exceeds a threshold, e^{out} is annotated
$a[e^{in}]$	6.5	1 if the input element is annotated, otherwise 0
$w[e^{in}]$	6.5	The weight of the contributing input element in the transformation

Appendix B Use case results

B.1 Transformation & Annotation Listing of Use Case 1

These are the complete listings of the description of the transformation of elements and the propagation of annotations in use case 1: Earth Remote Sensing.

B.1.1 Transformation 1

k	e^{out}	$i : j$	p	n	e^{in}	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
1	$m_1^{1'}$	1 : 1	1	1	v_1^1	$A[1] = \emptyset$	0	FALSE
1	$m_2^{1'}$	1 : 2	2	1	v_2^1	$A[1] = \emptyset$	0	FALSE
1	$m_3^{1'}$	1 : 3	3	1	v_3^1	$A[1] = \emptyset$	0	FALSE
1	$m_4^{1'}$	1 : 4	4	1	v_4^1	$A[1] = \emptyset$	0	FALSE
1	$m_5^{1'}$	2 : 1	1	2	v_1^2	$A[2] = \{v_2^2\}$	0	FALSE
1	$m_6^{1'}$	2 : 2	2	2	v_2^2	$A[2] = \{v_2^2\}$	1	TRUE
1	$m_7^{1'}$	2 : 3	3	2	v_3^2	$A[2] = \{v_2^2\}$	0	FALSE
1	$m_8^{1'}$	2 : 4	4	2	v_4^2	$A[2] = \{v_2^2\}$	0	FALSE
1	$m_9^{1'}$	3 : 1	1	3	v_1^3	$A[3] = \emptyset$	0	FALSE
1	$m_{10}^{1'}$	3 : 2	2	3	v_2^3	$A[3] = \emptyset$	0	FALSE
1	$m_{11}^{1'}$	3 : 3	3	3	v_3^3	$A[3] = \emptyset$	0	FALSE
1	$m_{12}^{1'}$	3 : 4	4	3	v_4^3	$A[3] = \emptyset$	0	FALSE
1	$m_{13}^{1'}$	4 : 1	1	4	v_1^4	$A[4] = \emptyset$	0	FALSE
1	$m_{14}^{1'}$	4 : 2	2	4	v_2^4	$A[4] = \emptyset$	0	FALSE
1	$m_{15}^{1'}$	4 : 3	3	4	v_3^4	$A[4] = \emptyset$	0	FALSE
1	$m_{16}^{1'}$	4 : 4	4	4	v_4^4	$A[4] = \emptyset$	0	FALSE
1	$m_{17}^{1'}$	5 : 1	1	5	v_1^5	$A[5] = \emptyset$	0	FALSE
1	$m_{18}^{1'}$	5 : 2	2	5	v_2^5	$A[5] = \emptyset$	0	FALSE
1	$m_{19}^{1'}$	5 : 3	3	5	v_3^5	$A[5] = \emptyset$	0	FALSE
1	$m_{20}^{1'}$	5 : 4	4	5	v_4^5	$A[5] = \emptyset$	0	FALSE
1	$m_{21}^{1'}$	6 : 1	1	6	v_1^6	$A[6] = \emptyset$	0	FALSE
1	$m_{22}^{1'}$	6 : 2	2	6	v_2^6	$A[6] = \emptyset$	0	FALSE
1	$m_{23}^{1'}$	6 : 3	3	6	v_3^6	$A[6] = \emptyset$	0	FALSE
1	$m_{24}^{1'}$	6 : 4	4	6	v_4^6	$A[6] = \emptyset$	0	FALSE
1	$m_{25}^{1'}$	7 : 1	1	7	v_1^7	$A[7] = \emptyset$	0	FALSE
1	$m_{26}^{1'}$	7 : 2	2	7	v_2^7	$A[7] = \emptyset$	0	FALSE
1	$m_{27}^{1'}$	7 : 3	3	7	v_3^7	$A[7] = \emptyset$	0	FALSE
1	$m_{28}^{1'}$	7 : 4	4	7	v_4^7	$A[7] = \emptyset$	0	FALSE

k	e^{out}	i	j	p	n	e^{in}	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
2	$m_1^{2'}$	1	1	1	8	v_1^8	$A[8] = \emptyset$	0	FALSE
2	$m_2^{2'}$	1	2	2	8	v_2^8	$A[8] = \emptyset$	0	FALSE
2	$m_3^{2'}$	1	3	3	8	v_3^8	$A[8] = \emptyset$	0	FALSE
2	$m_4^{2'}$	1	4	4	8	v_4^8	$A[8] = \emptyset$	0	FALSE
2	$m_5^{2'}$	2	1	1	9	v_1^9	$A[9] = \emptyset$	0	FALSE
2	$m_6^{2'}$	2	2	2	9	v_2^9	$A[9] = \emptyset$	0	FALSE
2	$m_7^{2'}$	2	3	3	9	v_3^9	$A[9] = \emptyset$	0	FALSE
2	$m_8^{2'}$	2	4	4	9	v_4^9	$A[9] = \emptyset$	0	FALSE
2	$m_9^{2'}$	3	1	1	10	v_1^{10}	$A[10] = \emptyset$	0	FALSE
2	$m_{10}^{2'}$	3	2	2	10	v_2^{10}	$A[10] = \emptyset$	0	FALSE
2	$m_{11}^{2'}$	3	3	3	10	v_3^{10}	$A[10] = \emptyset$	0	FALSE
2	$m_{12}^{2'}$	3	4	4	10	v_4^{10}	$A[10] = \emptyset$	0	FALSE
2	$m_{13}^{2'}$	4	1	1	11	v_1^{11}	$A[11] = \{v_2^{11}\}$	0	FALSE
2	$m_{14}^{2'}$	4	2	2	11	v_2^{11}	$A[11] = \{v_2^{11}\}$	1	TRUE
2	$m_{15}^{2'}$	4	3	3	11	v_3^{11}	$A[11] = \{v_2^{11}\}$	0	FALSE
2	$m_{16}^{2'}$	4	4	4	11	v_4^{11}	$A[11] = \{v_2^{11}\}$	0	FALSE
2	$m_{17}^{2'}$	5	1	1	12	v_1^{12}	$A[12] = \emptyset$	0	FALSE
2	$m_{18}^{2'}$	5	2	2	12	v_2^{12}	$A[12] = \emptyset$	0	FALSE
2	$m_{19}^{2'}$	5	3	3	12	v_3^{12}	$A[12] = \emptyset$	0	FALSE
2	$m_{20}^{2'}$	5	4	4	12	v_4^{12}	$A[12] = \emptyset$	0	FALSE
2	$m_{21}^{2'}$	6	1	1	13	v_1^{13}	$A[13] = \emptyset$	0	FALSE
2	$m_{22}^{2'}$	6	2	2	13	v_2^{13}	$A[13] = \emptyset$	0	FALSE
2	$m_{23}^{2'}$	6	3	3	13	v_3^{13}	$A[13] = \emptyset$	0	FALSE
2	$m_{24}^{2'}$	6	4	4	13	v_4^{13}	$A[13] = \emptyset$	0	FALSE
2	$m_{25}^{2'}$	7	1	1	14	v_1^{14}	$A[14] = \emptyset$	0	FALSE
2	$m_{26}^{2'}$	7	2	2	14	v_2^{14}	$A[14] = \emptyset$	0	FALSE
2	$m_{27}^{2'}$	7	3	3	14	v_3^{14}	$A[14] = \emptyset$	0	FALSE
2	$m_{28}^{2'}$	7	4	4	14	v_4^{14}	$A[14] = \emptyset$	0	FALSE

B.1.2 Transformation 2

e^{out}	i	j	p	q	n	e^{in}	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
m'_1	1	1	1	1	1	m_1^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_2	1	2	1	2	1	m_2^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_3	1	3	1	3	1	m_3^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_4	1	4	1	4	1	m_4^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_5	1	5	1	1	2	m_1^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_6	1	6	1	2	2	m_2^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_7	1	7	1	3	2	m_3^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_8	1	8	1	4	2	m_4^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_9	1	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{10}	1	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{11}	1	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{12}	1	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{13}	1	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{14}	1	14	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{15}	2	1	2	1	1	m_5^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{16}	2	2	2	2	1	m_6^1	$A[1] = \{m_6^1\}$	1	TRUE
m'_{17}	2	3	2	3	1	m_7^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{18}	2	4	2	4	1	m_8^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{19}	2	5	2	1	2	m_5^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{20}	2	6	2	2	2	m_6^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{21}	2	7	2	3	2	m_7^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{22}	2	8	2	4	2	m_8^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{23}	2	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{24}	2	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{25}	2	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{26}	2	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{27}	2	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{28}	2	14	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{29}	3	1	3	1	1	m_9^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_3	3	2	3	2	1	m_{10}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{31}	3	3	3	3	1	m_{11}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{32}	3	4	3	4	1	m_{12}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{33}	3	5	3	1	2	m_9^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{34}	3	6	3	2	2	m_{10}^2	$A[2] = \{m_{14}^2\}$	0	FALSE

e^{out}	i	j	p	q	n	e^{in}	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
m'_{35}	3	7	3	3	2	m_{11}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{36}	3	8	3	4	2	m_{12}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{37}	3	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{38}	3	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{39}	3	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{40}	3	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{41}	3	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{42}	3	14	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{43}	4	1	4	1	1	m_{13}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{44}	4	2	4	2	1	m_{14}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{45}	4	3	4	3	1	m_{15}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{46}	4	4	4	4	1	m_{16}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{47}	4	5	4	1	2	m_{13}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{48}	4	6	4	2	2	m_{14}^2	$A[2] = \{m_{14}^2\}$	1	TRUE
m'_{49}	4	7	4	3	2	m_{15}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{50}	4	8	4	4	2	m_{16}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{51}	4	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{52}	4	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{53}	4	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{54}	4	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{55}	4	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{56}	4	14	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{57}	5	1	5	1	1	m_{17}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{58}	5	2	5	2	1	m_{18}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{59}	5	3	5	3	1	m_{19}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{60}	5	4	5	4	1	m_{20}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{61}	5	5	5	1	2	m_{17}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{62}	5	6	5	2	2	m_{18}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{63}	5	7	5	3	2	m_{19}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{64}	5	8	5	4	2	m_{20}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{65}	5	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{66}	5	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{67}	5	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{68}	5	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{69}	5	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{70}	5	14	n/a	n/a	n/a	n/a	n/a	0	FALSE

e^{out}	i	j	p	q	n	e^{in}	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
m'_{71}	6	1	6	1	1	m_{21}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{72}	6	2	6	2	1	m_{22}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{73}	6	3	6	3	1	m_{23}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{74}	6	4	6	4	1	m_{24}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{75}	6	5	6	1	2	m_{21}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{76}	6	6	6	2	2	m_{22}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{77}	6	7	6	3	2	m_{23}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{78}	6	8	6	4	2	m_{24}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{79}	6	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{80}	6	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{81}	6	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{82}	6	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{83}	6	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{84}	6	14	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{85}	7	1	7	1	1	m_{25}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{86}	7	2	7	2	1	m_{26}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{87}	7	3	7	3	1	m_{27}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{88}	7	4	7	4	1	m_{28}^1	$A[1] = \{m_6^1\}$	0	FALSE
m'_{89}	7	5	7	1	2	m_{25}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{90}	7	6	7	2	2	m_{26}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{91}	7	7	7	3	2	m_{27}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{92}	7	8	7	4	2	m_{28}^2	$A[2] = \{m_{14}^2\}$	0	FALSE
m'_{93}	7	9	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{94}	7	10	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{95}	7	11	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{96}	7	12	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{97}	7	13	n/a	n/a	n/a	n/a	n/a	0	FALSE
m'_{98}	7	14	n/a	n/a	n/a	n/a	n/a	0	FALSE

B.1.3 Transformation 3

e^{out}	i	j	p	q	e^{in}	A	$a[e^{in}]$	$e^{out} \in A$
m'_1	1	1	1	9	m_9	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_2	1	2	1	10	m_{10}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_3	1	3	1	11	m_{11}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_4	1	4	1	12	m_{12}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_5	1	5	1	13	m_{13}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_6	1	6	1	14	m_{14}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_7	1	7	1	1	m_1	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_8	1	8	1	2	m_2	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_9	1	9	1	3	m_3	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{10}	1	10	1	4	m_4	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{11}	1	11	1	5	m_5	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{12}	1	12	1	6	m_6	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{13}	1	13	1	7	m_7	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{14}	1	14	1	8	m_8	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{15}	2	1	2	10	m_{24}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{16}	2	2	2	11	m_{25}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{17}	2	3	2	12	m_{26}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{18}	2	4	2	13	m_{27}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{19}	2	5	2	14	m_{28}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{20}	2	6	2	1	m_{15}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{21}	2	7	2	2	m_{16}	$A = \{m_{16}, m_{48}\}$	1	TRUE
m'_{22}	2	8	2	3	m_{17}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{23}	2	9	2	4	m_{18}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{24}	2	10	2	5	m_{19}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{25}	2	11	2	6	m_{20}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{26}	2	12	2	7	m_{21}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{27}	2	13	2	8	m_{22}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{28}	2	14	2	9	m_{23}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{29}	3	1	3	11	m_{39}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_3	3	2	3	12	m_{40}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{31}	3	3	3	13	m_{41}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{32}	3	4	3	14	m_{42}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{33}	3	5	3	1	m_{29}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{34}	3	6	3	2	m_3	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{35}	3	7	3	3	m_{31}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{36}	3	8	3	4	m_{32}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{37}	3	9	3	5	m_{33}	$A = \{m_{16}, m_{48}\}$	0	FALSE

e^{out}	i	j	p	q	e^{in}	A	$a[e^{in}]$	$e^{out} \in A$
m'_{38}	3	10	3	6	m_{34}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{39}	3	11	3	7	m_{35}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{40}	3	12	3	8	m_{36}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{41}	3	13	3	9	m_{37}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{42}	3	14	3	10	m_{38}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{43}	4	1	4	12	m_{54}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{44}	4	2	4	13	m_{55}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{45}	4	3	4	14	m_{56}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{46}	4	4	4	1	m_{43}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{47}	4	5	4	2	m_{44}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{48}	4	6	4	3	m_{45}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{49}	4	7	4	4	m_{46}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{50}	4	8	4	5	m_{47}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{51}	4	9	4	6	m_{48}	$A = \{m_{16}, m_{48}\}$	1	TRUE
m'_{52}	4	10	4	7	m_{49}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{53}	4	11	4	8	m_{50}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{54}	4	12	4	9	m_{51}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{55}	4	13	4	10	m_{52}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{56}	4	14	4	11	m_{53}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{57}	5	1	5	13	m_{69}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{58}	5	2	5	14	m_{70}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{59}	5	3	5	1	m_{57}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{60}	5	4	5	2	m_{58}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{61}	5	5	5	3	m_{59}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{62}	5	6	5	4	m_{60}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{63}	5	7	5	5	m_{61}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{64}	5	8	5	6	m_{62}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{65}	5	9	5	7	m_{63}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{66}	5	10	5	8	m_{64}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{67}	5	11	5	9	m_{65}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{68}	5	12	5	10	m_{66}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{69}	5	13	5	11	m_{67}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{70}	5	14	5	12	m_{68}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{71}	6	1	6	14	m_{84}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{72}	6	2	6	1	m_{71}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{73}	6	3	6	2	m_{72}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{74}	6	4	6	3	m_{73}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{75}	6	5	6	4	m_{74}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{76}	6	6	6	5	m_{75}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{77}	6	7	6	6	m_{76}	$A = \{m_{16}, m_{48}\}$	0	FALSE

e^{out}	i	j	p	q	e^{in}	A	$a[e^{in}]$	$e^{out} \in A$
m'_{78}	6	8	6	7	m_{77}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{79}	6	9	6	8	m_{78}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{80}	6	10	6	9	m_{79}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{81}	6	11	6	10	m_{80}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{82}	6	12	6	11	m_{81}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{83}	6	13	6	12	m_{82}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{84}	6	14	6	13	m_{83}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{85}	7	1	7	1	m_{85}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{86}	7	2	7	2	m_{86}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{87}	7	3	7	3	m_{87}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{88}	7	4	7	4	m_{88}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{89}	7	5	7	5	m_{89}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{90}	7	6	7	6	m_{90}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{91}	7	7	7	7	m_{91}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{92}	7	8	7	8	m_{92}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{93}	7	9	7	9	m_{93}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{94}	7	10	7	10	m_{94}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{95}	7	11	7	11	m_{95}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{96}	7	12	7	12	m_{96}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{97}	7	13	7	13	m_{97}	$A = \{m_{16}, m_{48}\}$	0	FALSE
m'_{98}	7	14	7	14	m_{98}	$A = \{m_{16}, m_{48}\}$	0	FALSE

B.1.4 Transformation 4

e^{out}	i	j	p	q	e^{in}	A	$a[e^{in}]$	$e^{out} \in A$
m'_1	1	1	3	5	m_{33}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_2	1	2	3	6	m_{34}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_3	1	3	3	7	m_{35}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_4	1	4	3	8	m_{36}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_5	1	5	3	9	m_{37}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_6	1	6	3	10	m_{38}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_7	2	1	4	5	m_{47}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_8	2	2	4	6	m_{48}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_9	2	3	4	7	m_{49}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{10}	2	4	4	8	m_{50}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{11}	2	5	4	9	m_{51}	$A = \{m_{21}, m_{51}\}$	1	TRUE
m'_{12}	2	6	4	10	m_{52}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{13}	3	1	5	5	m_{61}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{14}	3	2	5	6	m_{62}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{15}	3	3	5	7	m_{63}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{16}	3	4	5	8	m_{64}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{17}	3	5	5	9	m_{65}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{18}	3	6	5	10	m_{66}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{19}	4	1	6	5	m_{75}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{20}	4	2	6	6	m_{76}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{21}	4	3	6	7	m_{77}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{22}	4	4	6	8	m_{78}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{23}	4	5	6	9	m_{79}	$A = \{m_{21}, m_{51}\}$	0	FALSE
m'_{24}	4	6	6	10	m_{80}	$A = \{m_{21}, m_{51}\}$	0	FALSE

B.2 Transformation & Annotation Listing of Use Case 2

These are the complete listings of the description of the transformation of elements and the propagation of annotations in use case 2: Temperature Sensor Network.

B.2.1 Transformation 1

e^{out}	i	j	p	n	e^{in}	$A[n]$	$a[e^{in}]$	$e^{out} \in A$
m'_1	1	1	1	1	v_1^1	$A[1] = \{v_1^1\}$	1	TRUE
m'_2	1	2	n/a	n/a	n/a	n/a	0	FALSE
m'_3	1	3	n/a	n/a	n/a	n/a	0	FALSE
m'_4	1	4	n/a	n/a	n/a	n/a	0	FALSE
m'_5	1	5	1	2	v_1^2	$A[2] = \emptyset$	0	FALSE
m'_6	2	1	n/a	n/a	n/a	n/a	0	FALSE
m'_7	2	2	n/a	n/a	n/a	n/a	0	FALSE
m'_8	2	3	n/a	n/a	n/a	n/a	0	FALSE
m'_9	2	4	n/a	n/a	n/a	n/a	0	FALSE
m'_{10}	2	5	n/a	n/a	n/a	n/a	0	FALSE
m'_{11}	3	1	n/a	n/a	n/a	n/a	0	FALSE
m'_{12}	3	2	n/a	n/a	n/a	n/a	0	FALSE
m'_{13}	3	3	n/a	n/a	n/a	n/a	0	FALSE
m'_{14}	3	4	n/a	n/a	n/a	n/a	0	FALSE
m'_{15}	3	5	n/a	n/a	n/a	n/a	0	FALSE
m'_{16}	4	1	n/a	n/a	n/a	n/a	0	FALSE
m'_{17}	4	2	n/a	n/a	n/a	n/a	0	FALSE
m'_{18}	4	3	n/a	n/a	n/a	n/a	0	FALSE
m'_{19}	4	4	n/a	n/a	n/a	n/a	0	FALSE
m'_{20}	4	5	n/a	n/a	n/a	n/a	0	FALSE
m'_{21}	5	1	n/a	n/a	n/a	n/a	0	FALSE
m'_{22}	5	2	1	3	v_1^3	$A[3] = \{v_1^3\}$	1	TRUE
m'_{23}	5	3	n/a	n/a	n/a	n/a	0	FALSE
m'_{24}	5	4	n/a	n/a	n/a	n/a	0	FALSE
m'_{25}	5	5	n/a	n/a	n/a	n/a	0	FALSE

B.2.2 Transformation 2

e^{out}	i	j	p	q	$S^{in}[e^{out}]$	$l(e^{out}, m_1)$	$l(e^{out}, m_5)$	$l(e^{out}, m_{22})$	$a[e^{in}]$	$e^{out} \in A$
m'_1	1	1	1	1	$\{m_1\}$	1,00	n/a	n/a	1,00	TRUE
m'_2	1	2	1	2	$\{m_1, m_5, m_{22}\}$	0,86	0,58	0,43	0,43	TRUE
m'_3	1	3	1	3	$\{m_1, m_5, m_{22}\}$	0,72	0,72	0,42	0,38	FALSE
m'_4	1	4	1	4	$\{m_1, m_5, m_{22}\}$	0,58	0,86	0,37	0,31	FALSE
m'_5	1	5	1	5	$\{m_5\}$	n/a	1,00	n/a	0,00	FALSE
m'_6	2	1	2	1	$\{m_1, m_5, m_{22}\}$	0,86	0,42	0,55	0,47	TRUE
m'_7	2	2	2	2	$\{m_1, m_5, m_{22}\}$	0,80	0,55	0,58	0,46	TRUE
m'_8	2	3	2	3	$\{m_1, m_5, m_{22}\}$	0,68	0,68	0,55	0,41	TRUE
m'_9	2	4	2	4	$\{m_1, m_5, m_{22}\}$	0,55	0,80	0,49	0,35	FALSE
m'_{10}	2	5	2	5	$\{m_1, m_5, m_{22}\}$	0,42	0,86	0,40	0,27	FALSE
m'_{11}	3	1	3	1	$\{m_1, m_5, m_{22}\}$	0,72	0,37	0,68	0,47	TRUE
m'_{12}	3	2	3	2	$\{m_1, m_5, m_{22}\}$	0,68	0,49	0,72	0,47	TRUE
m'_{13}	3	3	3	3	$\{m_1, m_5, m_{22}\}$	0,60	0,60	0,68	0,43	TRUE
m'_{14}	3	4	3	4	$\{m_1, m_5, m_{22}\}$	0,49	0,68	0,60	0,36	FALSE
m'_{15}	3	5	3	5	$\{m_1, m_5, m_{22}\}$	0,37	0,72	0,49	0,29	FALSE
m'_{16}	4	1	4	1	$\{m_1, m_5, m_{22}\}$	0,58	0,29	0,80	0,46	TRUE
m'_{17}	4	2	4	2	$\{m_1, m_5, m_{22}\}$	0,55	0,40	0,86	0,47	TRUE
m'_{18}	4	3	4	3	$\{m_1, m_5, m_{22}\}$	0,49	0,49	0,80	0,43	TRUE
m'_{19}	4	4	4	4	$\{m_1, m_5, m_{22}\}$	0,40	0,55	0,68	0,36	FALSE
m'_{20}	4	5	4	5	$\{m_1, m_5, m_{22}\}$	0,29	0,58	0,55	0,28	FALSE
m'_{21}	5	1	5	1	$\{m_1, m_5, m_{22}\}$	0,43	0,20	0,86	0,43	TRUE
m'_{22}	5	2	5	2	$\{m_1, m_5, m_{22}\}$	n/a	n/a	1,00	1,00	TRUE
m'_{23}	5	3	5	3	$\{m_1, m_5, m_{22}\}$	0,37	0,37	0,86	0,41	TRUE
m'_{24}	5	4	5	4	$\{m_1, m_5, m_{22}\}$	0,29	0,42	0,72	0,34	FALSE
m'_{25}	5	5	5	5	$\{m_1, m_5, m_{22}\}$	0,20	0,43	0,58	0,26	FALSE

B.2.3 Transformation 3

e^{out}	i	j	p	q	e^{in}	$a[e^{in}]$	$e^{out} \in A$
m'_1	1	1	1	1	m_1	1	TRUE
m'_2	1	2	1	2	m_2	1	TRUE
m'_3	1	3	1	3	m_3	0	FALSE
m'_4	1	4	1	4	m_4	0	FALSE
m'_5	1	5	1	5	m_5	0	FALSE
m'_6	1	6	n/a	n/a	n/a	0	FALSE
m'_7	1	7	n/a	n/a	n/a	0	FALSE
m'_8	2	1	2	1	m_6	1	TRUE
m'_9	2	2	2	2	m_7	1	TRUE
m'_{10}	2	3	2	3	m_8	1	TRUE
m'_{11}	2	4	2	4	m_9	0	FALSE
m'_{12}	2	5	2	5	m_{10}	0	FALSE
m'_{13}	2	6	n/a	n/a	n/a	0	FALSE
m'_{14}	2	7	n/a	n/a	n/a	0	FALSE
m'_{15}	3	1	3	1	m_{11}	1	TRUE
m'_{16}	3	2	3	2	m_{12}	1	TRUE
m'_{17}	3	3	3	3	m_{13}	1	TRUE
m'_{18}	3	4	3	4	m_{14}	0	FALSE
m'_{19}	3	5	3	5	m_{15}	0	FALSE
m'_{20}	3	6	n/a	n/a	n/a		FALSE
m'_{21}	3	7	n/a	n/a	n/a		FALSE
m'_{22}	4	1	4	1	m_{16}	1	TRUE
m'_{23}	4	2	4	2	m_{17}	1	TRUE
m'_{24}	4	3	4	3	m_{18}	1	TRUE
m'_{25}	4	4	4	4	m_{19}	0	FALSE
m'_{26}	4	5	5	5	m_{20}	0	FALSE
m'_{27}	4	6	n/a	n/a	n/a	0	FALSE
m'_{28}	4	7	n/a	n/a	n/a	0	FALSE
m'_{29}	5	1	5	1	m_{21}	1	TRUE
m'_{30}	5	2	5	2	m_{22}	1	TRUE
m'_{31}	5	3	5	3	m_{23}	1	TRUE
m'_{32}	5	4	5	4	m_{24}	0	FALSE
m'_{33}	5	5	5	5	m_{25}	0	FALSE
m'_{34}	5	6	n/a	n/a	n/a	0	FALSE
m'_{35}	5	7	n/a	n/a	n/a	0	FALSE
m'_{36}	6	1	n/a	n/a	n/a	0	FALSE
m'_{37}	6	2	n/a	n/a	n/a	0	FALSE

e^{out}	i	j	p	q	e^{in}	$a[e^{in}]$	$e^{out} \in A$
m'_{38}	6	3	n/a	n/a	n/a	0	FALSE
m'_{39}	6	4	n/a	n/a	n/a	0	FALSE
m'_{40}	6	5	n/a	n/a	n/a	0	FALSE
m'_{41}	6	6	n/a	n/a	n/a	0	FALSE
m'_{42}	6	7	n/a	n/a	n/a	0	FALSE
m'_{43}	7	1	n/a	n/a	n/a	0	FALSE
m'_{44}	7	2	n/a	n/a	n/a	0	FALSE
m'_{45}	7	3	n/a	n/a	n/a	0	FALSE
m'_{46}	7	4	n/a	n/a	n/a	0	FALSE
m'_{47}	7	5	n/a	n/a	n/a	0	FALSE
m'_{48}	7	6	n/a	n/a	n/a	0	FALSE
m'_{49}	7	7	n/a	n/a	n/a	0	FALSE