LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

Efficient and Practical Access to Any Graph Data: Data Structures for Continuous Image Graphs and Beyond

Simon Brodt

Diplomarbeit

Beginn der Arbeit:	01. Mai 2009
Abgabe der Arbeit:	8. November 2010
Betreuer:	Prof. Dr. François Bry
	Dr. Tim Furche

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 8. November 2010 .....

Simon Brodt

Zusammenfassung

Wir definieren eine Schnittstelle für den einheitlichen und effizienten Zugriff auf Graphdaten, die in unterschiedlichen Datenstrukturen gespeichert sind. Wir zeigen, dass eine einheitliche Schnittstelle objektive Vergleiche verschiedener Speicherverfahren erleichtert. Zudem ermöglicht sie die Implementierung effizienter und von der Repräsentationsform der Daten unabhängiger Auswertungsalgorithmen für Anfragen auf Graphen. Desweitern wird ein System abstrakter Klassen eingeführt mit dessen Hilfe sich konkrete Datenstrukturen einfach und bequem implementieren lassen, ohne dass dabei die Effizienz beeinträchtigt wird.

Wir stellen sorgfältige Implementierungen der Adjazenz-Matrix-, der Adjazenz-Listen- und einer weiteren Datenstruktur zur Verfügung, welche auf der Pre-Post-Kodierung basiert. Außerdem wird Single-Interval-Compression (SIC), ein Intervall basiertes Speicherverfahren für Graphen, zum ersten Mal vollständig implementiert. Darüber hinaus wird eine neue Erweiterung von SIC auf allgemeine Graphen eingeführt und implementiert, die eine höchstens 50% vom Optimum abweichende die Intervalldarstellung des Graphen liefert. Zwei weitere neue Speicherverfahren nutzen die lokale Struktur des Graphen. Das erste bedient sich baum-ähnlicher Strukturen im Graphen um diesen in einzelne Komponenten zu zerlegen, welche dann unabhängig voneinander behandelt werden können. Das zweite verschmilzt die Knoten starker Zusammenhangskomponenten des Graphen, um seine Größe zu verringern und speichert die Descendant-Relation des entstehenden DAGs. Beide Speicherverfahren profitieren von dem einheitlichen Interface.

Abschließend zeigen wir einige interessante Eigenschaften von Continuous-Image-Graphs (CIGs) einer speziellen Graphklasse, die von der SIC Datenstruktur verwendet wird. Insbesondere zeigen wir, dass Homomorphismen CIGs erhalten, und dass das Verschmelzen der Knoten starker Zusammenhangskomponenten eines Graphen mit der CIG Darstellung seines transitiven Abschlusses verträglich ist.

Abstract

We present an interface providing uniform and efficient access to graph data stored in different data structures. We show that an uniform interface eases objective comparisons between different storage schemes and enables the implementation of efficient evaluation algorithms for queries on graphs which do not depend on the representation of the data. A framework of abstract classes is introduced which leads to simple and comfortable implementations of concrete data structures without affecting efficiency.

We carefully implement the adjacency matrix, the adjacency array and the Pre-Post-Encoding (PPE) data structure. Furthermore a first reasonable implementation of Single-Interval-Compression (SIC), a interval based storage scheme for graphs, is given. We newly introduce and implement Multi-Interval-Compression (MIC), an extension of SIC to arbitrary graphs, providing an interval representation of the graph which is at most 50% from optimal. In addition two storage schemes are defined which exploit the local structure of the graph. The first one uses tree-like structures in the graph to decompose it into components which then can be treated independently. The second one collapses strongly connected components to reduce the size of the graph and stores the descendant relation of the resulting DAG. Both data structures profit from the uniform interface.

Finally we prove some interesting properties of Continous-Image-Graphs (CIGs) a special graph class used by the SIC datastructure. Particularly, we show that homomorphisms preserve CIGs and that collapsing the strongly connected components of a graph is compatible with the CIG representation of its transitive closure or descendant relation.

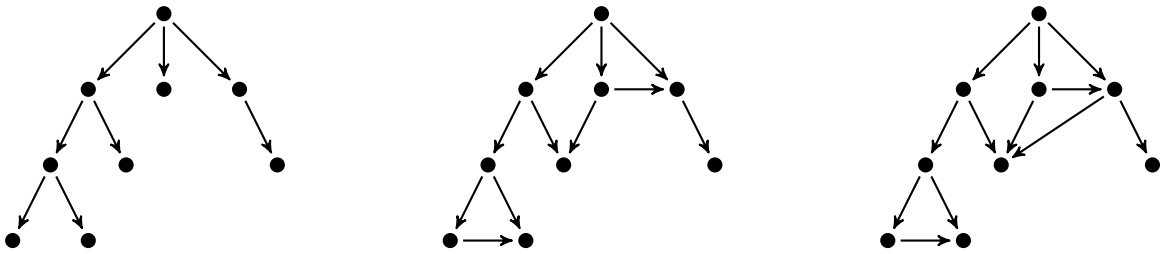
Inhaltsverzeichnis

1. Introduction

Graph-shaped data plays an important role in many fields of computing science. It is used to represent technical as well as social networks, DNA-Sequences, molecules, electronic circuits and so on. Furthermore languages like XML and RDF that enable the description of (labeled) graphs have spread in the web. Therefore the efficient representation and querying of graph data has become a widely discussed problem.

This work concentrates on static approaches of which many have been suggested and discussed, among others graph representation functions [**?**], BDDs [**?**] and hashing schemes [**?**, **?**]. A particularly interesting class of approaches are so called labeling schemes. Their basic idea is to assign labels to the vertices in such a way that one can determine the adjacency and/or reachability of two nodes by analysing their labels only. Such labeling schemes exist for trees [**?**, **?**], CIGs [**?**], interval graphs [**?**, **?**, **?**], planar graphs [**?**] and other graph classes [**?**]. Some of these labeling schemes have been designed specifically for the representation of graphs in relational databases. To that end they provide transformations for queries on graphs into queries on relations that can be answered efficiently by a relational database. The Pre-Post-Encoding (PPE) for instance is such a labeling scheme for trees that is used in many mappings from XML to relational databases [**?**, **?**].

A desire for uniformity. As most of the approaches mentioned above have been developed with different aims and under different assumptions, objective comparisons between them are difficult. Moreover the inhomogeneity makes it hard to develop an evaluation algorithm that works on several, i.e. independently from the currently used data structure. As a result current evaluation algorithms have been mostly tailored to one specific data structure. However an algorithm which is able to choose the most suitable data structure for each graph or even to use different data structures in different parts of the graph, is highly desirable. Consider the following example.



The first of the preceding graphs is a tree and a tree storage scheme like the mentioned Pre-Post-Encoding (PPE) is a suitable data structure. The second is not a tree but belongs to a larger class of graphs so-called Continuous-Image-Graphs (CIGs) which have a special interval representation. This specific property can be used to obtain a very efficient data structure for CIGs (Sec. 3.4). The third is neither a tree nor a CIG but stands for some arbitrary, possibly cyclic graph. Section 3.5 presents a data structure extending the one for CIGs to arbitrary graphs, which would be a good choice for it. The point we want to make here is that for the given graphs different data structures are most desirable. So what do we need? We need an *efficient interface* providing *uniform access* to those different data structures.

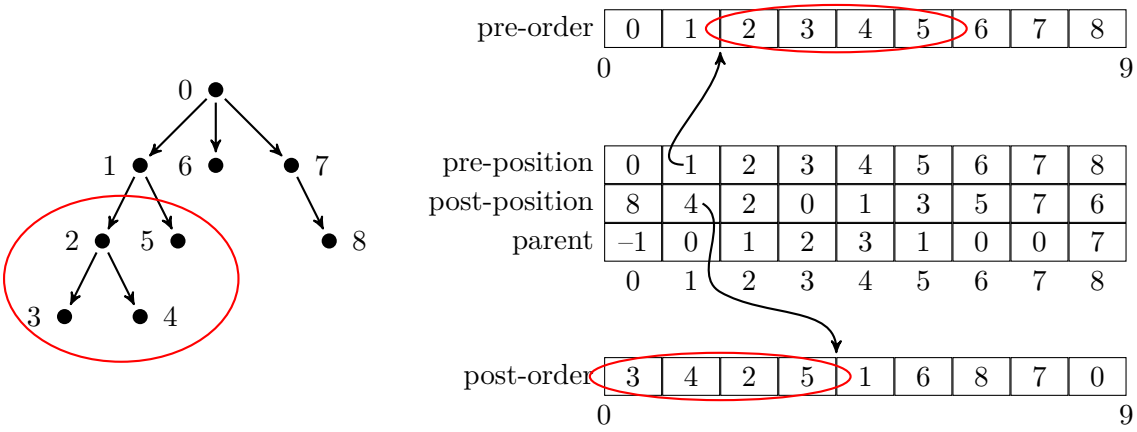
What does "efficient" mean? When talking about efficiency we always focus on a certain set of operations that an interface or data structure should support efficiently. This work actually focuses on two operations that are essential for accessing graph data:

- Testing the adjacency of two vertices , i.e. whether there is an edge from the first to the second vertex or not
- Iterating over all vertices adjacent to some vertex

These two operations suffice to cover adjacency, reachability and any other binary relation over the vertices of a graph. The reason is quite simple. A graph can be identified with its adjacency relation and vice versa. Therefore one may interpret a graph as a binary relation over its vertices. Conversely every binary relation over a set of vertices is the adjacency relation of some graph and can be identified with this graph. So seen from the formal point of view testing reachability on a graph is just the same as testing adjacency on the graph corresponding to its reachability relation. This means that data structures for the reachability relation (see Sections 3.3, 3.4, 3.7) may use the same interface and operation as data structures for the adjacency relation.

Back to efficiency. When is an operation said to be supported efficiently? In the context of a concrete data structure an operation is supported efficiently if it has good complexity. In the context of an interface this does not make any sense. Here efficiency means that the use of the interface neither increases the complexity of a data structure nor affects its practical runtime significantly. For the latter the interface has to be sufficiently low-level.

Trees, CIGs and Beyond. As already mentioned there is a very efficient data structure for representing trees and forests: Pre-Post-Encoding (PPE). PPE defines two orderings on the vertices of the tree. To do so PPE performs a depth first traversal on the tree and orders the vertices first by the point of time they are reached/entered, and second by the point of time they are left. The first ordering is called pre-order, the second post-order. These orderings suffice to determine reachability (and even further relations) on the tree. Combined with only one further number assigned to each vertex, the depth of the vertex [**?**, **?**] or a reference to the parent of a vertex [**?**], the adjacency relation is covered, too.

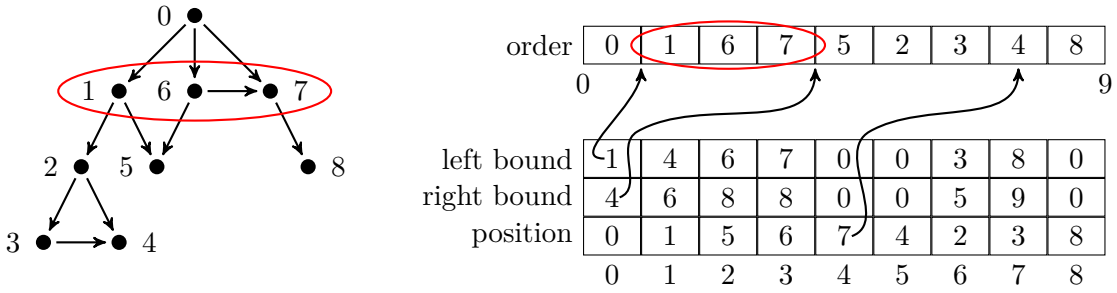


The above figure illustrates how to determine the set of vertices (marked red) reachable from vertex 1. The vertices of the tree are labeled by (arbitrary) identifiers for the vertices.

The two orderings are both represented by an array, namely *pre-order* and *post-order*. The arrays contain the vertices (in fact their identifiers) in their corresponding order. *pre-position* assigns to each vertex its pre-order position (thus its position in the *pre-order* array). *post-position* does the same for post-order. The vertices 2, 3, 4 and 5 which form the set of vertices reachable from vertex 1 are characterized by the following: Firstly the value of pre-position for these vertices is greater than the value of pre-position for vertex 1 and secondly the value of post-position is smaller than the value of post-position for vertex 1. The vertices adjacent to vertex 1, i.e. 2 and 5 are just those vertices which have 1 as parent. For details on the implementation see Section 3.3.

PPE is a well studied data structure, both in theory and practice. Unfortunately most graphs aren't trees. There exist attempts [**?**] to apply Pre-Post-Encoding to more graphs but they are mainly based on the idea of extracting some tree-shaped portion of the graph and applying Pre-Post-Encoding to this portion while storing the rest of the graph elsewhere in a almost uncompressed way. On a non-sparse graph this rest consists of most of the graph and its uncompressed representation uses up to quadratic space.

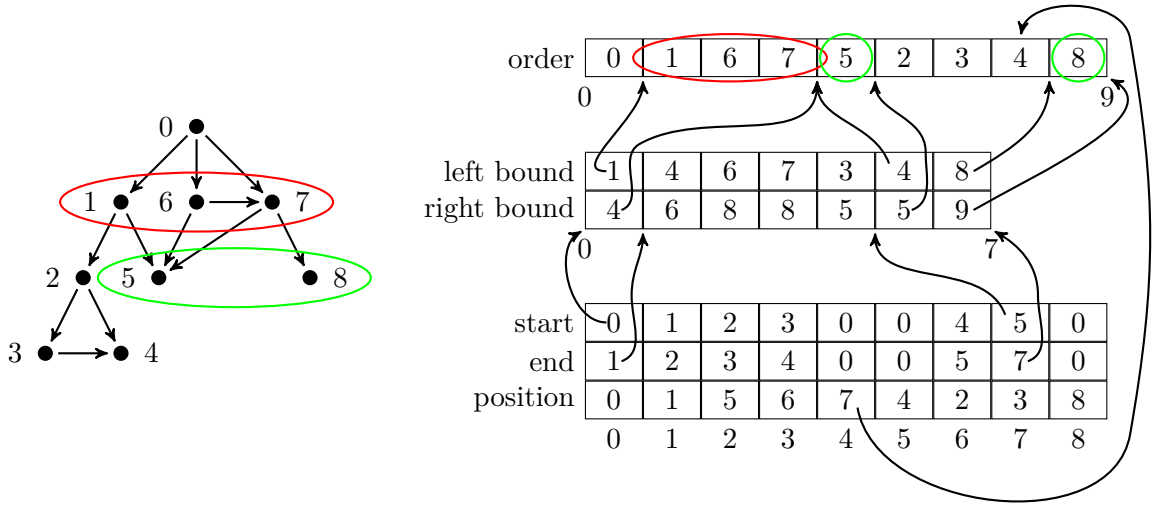
Despite of that CIGs are a real superclass of trees or forests. A graph is a CIG if there exists an ordering of the vertices of the graph in which the set of adjacent vertices may be represented by a single interval for each vertex. The search of such an ordering, if one exists, takes only linear time in the number of edges of the graph [**?**]. Single-Interval-Compression (SIC) makes uses of this. It assigns to each vertex its position in the ordering and the bounds of the interval corresponding to its adjacent vertices (see also Section 3.4). Compared with PPE which uses two orderings, SIC seems to be more efficient as SIC needs only a single ordering. The advantage of PPE is, that one may represent several relations using the same two orderings which is not possible for SIC in general.



In the preceding figure the vertices of the graph are labeled by (arbitrary) identifiers. Like the orderings in PPE, the computed ordering is represented by an array *order*, which contains the vertices (in fact their identifiers) in the computed order. *order* assigns the corresponding vertex to each position in the computed ordering. *position* does the converse and assigns to each vertex its position in the ordering. Finally *left bound* and *right bound* provide the interval bounds for each vertex. For each vertex the set of adjacent vertices may be determined very easily as illustrated by the example of vertex 0. Its adjacent vertices 1, 6, and 7 which are marked red in the figure are just the subsequence of *order* starting with the left and ending with the right interval bound assigned to 0.

The SIC data structure, named sequence map there, is proposed in [**?**] where it is also shown to be particularly suitable for efficient evaluation of complex queries.

Another advantage of the SIC data structure is that it is extensible to arbitrary graphs in a natural way. SIC represents the set of adjacent vertices by a single interval for each vertex. Its extensions loosens this restriction and allows multiple intervals for each node. It is consequently called Multi-Interval-Compression (MIC). The efficiency of MIC is determined by the number of intervals used for the whole graph and so MIC tries to minimize this number. Unfortunately the problem of finding an ordering that minimizes the number of intervals is proved to be NP-complete [**?**]. But there exist good approximations actually there is a 2- and even a 1.5-approximation [**?**]. Using these approximations MIC is able to provide an interval representation of the graph that is only 50% from optimal (see also Section 3.5).

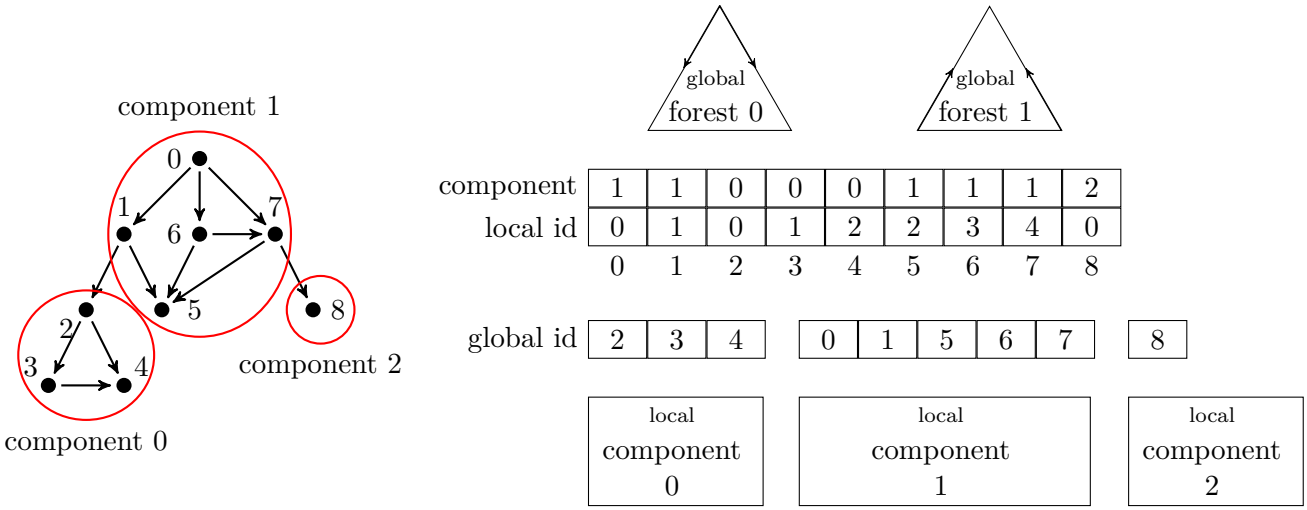


Again the numbers beside the vertices of the graph are identifiers for the vertices. *order* assigns the corresponding vertex to each position in the computed ordering. *position* does the converse and assigns to each vertex its position in the ordering. So far it is just the same for MIC as for SIC. Now there is a difference. *left bound* and *right bound* still provide the interval bounds. But in MIC each vertex may have multiple intervals. The intervals assigned to each vertex are given by the subsequence of *left bound* and *right bound* beginning at the value of *start* and ending at the valueend for that vertex.

The set of vertices adjacent vertices of some vertex are the vertices contained in one of the intervals assigned to the vertex. For the adjacent vertices of vertex 0, marked red in the figure things stayed quite unchanged compared to the SIC example as they are still represented in a single interval. There was only introduced some indirection. But vertex 7 has to use two intervals for its adjacent vertices 5 and 8 which are marked green.

Adapting to local Structure. MIC is an efficient data structure for arbitrary graphs. Nevertheless specialized data structures as Pre-Post-Encoding or SIC are still more efficient on their specific graph classes (in particular w.r.t. space use). So if we were able to decompose graphs in such a way that all or at least many of the resulting components lie in one of those specific graph classes, this may lead to an very efficient data structure. Note that due to the uniform interface it is not necessary that all components fall in the same specific graph class. But how to find such a decomposition.

One possibility which applies well to graphs with a significant amount of tree structure is presented here. The idea is to remove bridges from the graph. Put more precisely we consider the graph undirected and search those edges which are the only connection between two parts of the graph. When we have found those undirected edges we remove all corresponding directed edges, which may be one or two, from the original graph. It is possible to cover the removed edges by two directed forests. After the removal of the bridges the graph decomposes into several components. Each of the components may then be stored by the data structure which is most suitable for that component. The following figure illustrates this.



Just like in the preceeding figures the numbers beside the vertices of the graph are identifiers for the vertices. The bridges of the graph, i.e. the edges from 1 to 2 and from 7 to 8 are stored in *forest 0* and *forest 1*. After removing the briges the graph decomposes into three components. *component* assigns to each vertex the component it belongs to and *local id* defines the local identifier of the vertex in the component. This renaming is due to a special but important property of the used interface which is discussed in Section 2.2. *global id* translates back form the local identifier to the global. The adjacency of two vertices v_1, v_2 is determined as follows: The graph contains an edge from v_1 to v_2 if *forest 0* contains an edge from v_1 to v_2, *forest 1* contains an edge from v_2 to v_1 or v_1 and v_2 fall in the same local component and the edge from v_1 to v_2 is contained in that component. For further details see Section 3.6.

In this work. In Section 2 we present an interface providing uniform and efficient access to graph data stored in different data structures. Furthermore we introduce the abstract framework that has been developed to simplify the implementation of those data structures under the interface. Section 3 informs about the data structures implemented during this diploma thesis. This covers careful implementations of basic data structures like the adjacency matrix (Section 3.1 and the adjacency list data structure (Section 3.2) as well as implementations of PPE (Section 3.3), SIC (Section 3.4), MIC (Section 3.5) and of two data structures using the idea of localization (Sections 3.6 and 3.7). For SIC this is the first reasonable implementation. MIC and the datastructures based on localization are newly de-

veloped in this work. In Section 4 we approache to CIGs in a theoretical way and provides some interesting properties of this graph class. Particularly we show that homomorphisms preserve CIGs and that the transitive closure of a graph remains a CIG if the strongly connected components of the graph are collapsed. Finally Section 5 points to questions that remained open, further ideas and related work that may be affected by the results of this diploma thesis.

2. A Uniform Interface for Graph Data

In this Section we present an interface for accessing graph data. We want the interface to meet a number of requirements.

1. **Uniformity.** Why do we need Uniformity? One reason is given in the introduction. There we focus on one relation on different graphs and emphasize the advantage of an evaluation algorithm that can access any graph in the same manner, regardless of how that graph is stored. This allows us to choose the most fitting data structure for each graph without adapting the evaluation algorithm.

   There is also a second reason. When working on graph data you are frequently interested in a couple of relations not only in a single one. XPath for instance uses 13 different axes or relations [**?**]. Another example are graphs with labeled edges like RDF graphs. One possibility to represent a graph with labeled edges is to partition the edges by label, i.e., to treat all edges with the same label as one relation. Consider an evaluation algorithm for a query language such as XPath using several relations. Though the relations are different they are probably interchangeable meaning that when a relation in some query is replaced by another relation the result is still a query. In XPath, e.g., we can replace each of the structural axes like child or ancestor by another structural axes. This implies that a general evaluation algorithm performs equal or at least very similar operations on all relations. So the use of an uniform interface for all relations should be possible. Such an interface, using a small number of general instead of many specific operations, unburdens the evaluation algorithm of much relation specific code, makes it reuseable and simplifies the integration of new relations.
2. **Efficiency.** The meaning of efficiency in the context of an interface has already been discussed in the introduction. The use of the uniform interface should neither increase the complexity of a data structure nor significantly affect its practical runtime.
3. **Flexibility.** As already mentioned in (1.) we are frequently interested in a couple of relations when working on graph data not only in a single one. The set of used relations differs from application to application. It may even change within one application as shown by the example of graphs with labeled edges. So the interface has to find a flexible way to handle different sets of relations.

How do we meet these requirements? First of all we do not define an interface for graphs but for relations. By this we are able to handle all relations on the graph in a uniform way. The graph is not modeled explicitly but is represented by its adjacency relation or by the set of relations on its vertices, in which one is currently interested. The alternative is to model each relation on the vertices of a graph as a graph. Then the graph is modeled explicitly but working on a set of relations on the vertices of the graph results in a set of graphs. However this is only a question of perspective. The first option seems us to be more natural. Particularly it opens the possibility to cover also general (binary) relations which are not defined over a quadratic domain. Many data structures are extensible to general relations without any cost. There is another advantage, too. Up to this point we have referred to *binary* relations when talking about relations on the vertices of a graph.

But *unary* relations on the vertices which correspond to selections on the set of vertices for example by label, should also be represented. From the relational point of view the difference between unary and binary relations is small. Therefore finding a common generalization is simple.

As implementation language we choose C#. C# is a platform independent object-oriented language which nevertheless is designed to enable highly performance and provides direct access to memory.

To achieve a maximum of uniformity we start with a generic and therefore very general interface (Section 2.1). Second we give a specialized interface which is restricted in such a way that it enables a more efficient and low level access to the data (Section 2.2). Finally we provide a flexible possibility to define sets of relations where different relations in one set may even share data (Section 2.3). The implementation of the Pre-Post-Encoding (Section 3.3) uses this feature.

On interfaces and abstract classes C# offers an "interface" programming construct. Though this construct is used in the implementation it does not really correspond to the term "interface" as used in this work. As we have used the term ïnterfaceßo far, it is not meant as a formal programming construct but in the sense of a docking point for algorithms with the data structures. The C# interfaces might be used as such docking points but unfortunately accessing interface functions is quite ineffective compared with accessing class functions, a general problem of programming languages offering interfaces. For this reason the intended docking points in our implementation are abstract classes implemented directly under the interfaces. Their advantage is first the higher efficiency and second the possibility to predefine some additional operations derived from the basic ones which give a comfortable framework for later implementations. So why do we keep the interfaces at all? The interfaces are a proper formal specification. The abstract classes already contain implementation specific details. So the use of interfaces *and* abstract classes helps to integrate proper modeling with high efficiency and good comfort in practice.

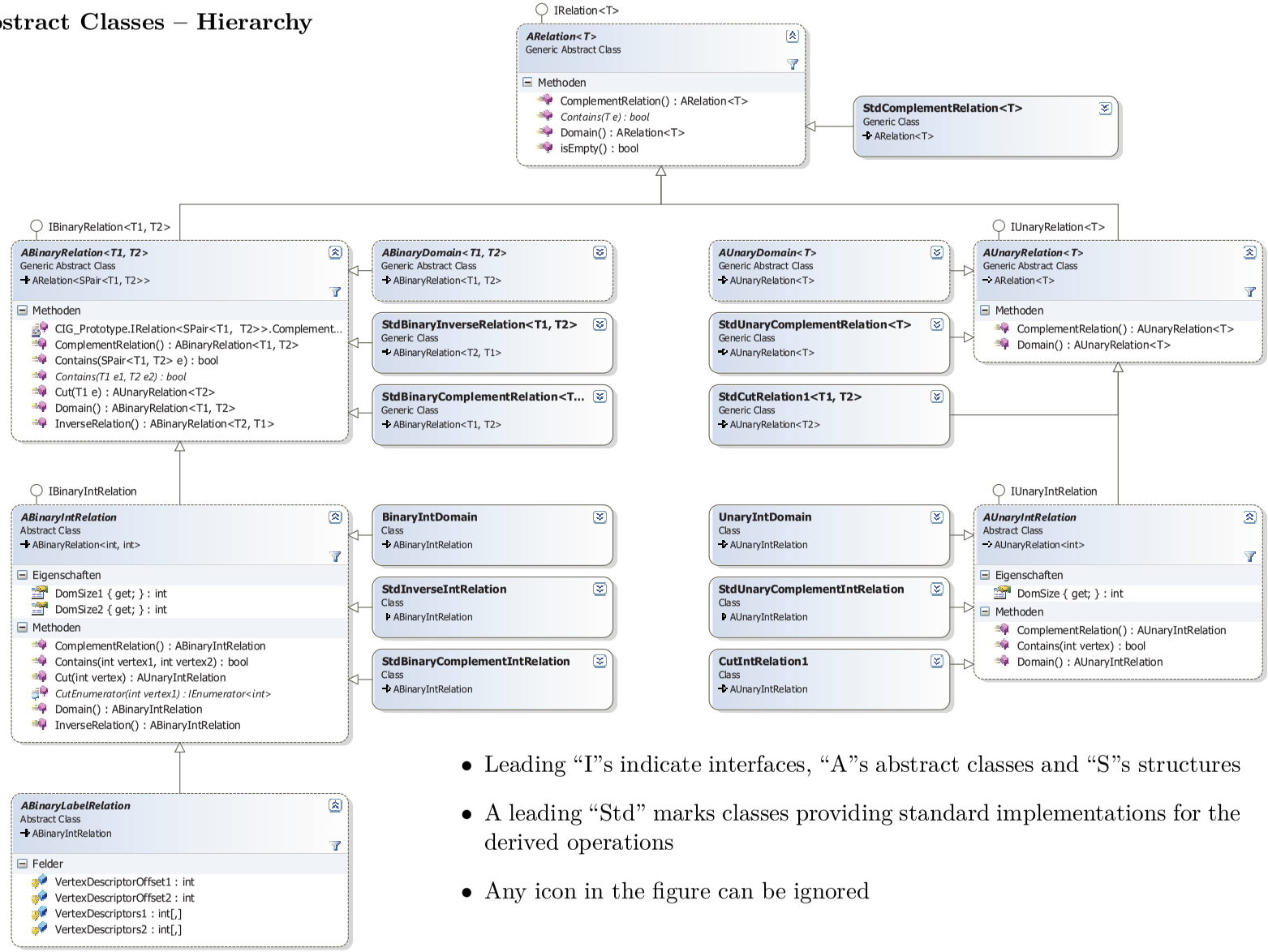
2.1. A Generic Interface for Relations (on Graphs)

Unary and binary relations have different arity or dimension. However both are relations and therefore share some common properties. First, a relation is defined over some *domain*. Second, a relation is a *set of elements* out of the domain, i.e. a subset of the domain.

We model general, computational decidable relations in C# by the generic *IRelation* interface and the underlying *ARelation* abstract class. Generic means that the concrete type of the elements of the relation is not specified explicitly but is left to a type parameter. The interface matches only computational decidable relations as it offers an operation *Contain* which determines whether some element of the specified type is in the relation or not. Furthermore *IRelation* extends the C# *IEnumerable* interface what implies that one is able to iterate over the elements of the relation. The domain of a relation is modeled as a property of the relation and is relation which has itself as domain.

The following figure gives an overview over all abstract classes.

Abstract Classes – Hierarchy



- Leading “I”s indicate interfaces, “A”s abstract classes and “S”s structures
- A leading “Std” marks classes providing standard implementations for the derived operations
- Any icon in the figure can be ignored

ComplementRelation and *isEmpty* are derived operations and *ARelation* and all of its abstract subclasses provide standard implementations of these operations based on the basic ones. So a non-abstract subclass for some concrete data structure has to do only a minimal amount of work to obtain a first fully workable implementation. Specific more efficient implementations of the derived operations may be postponed.

Terminology Sometimes it is useful to address the elements of the (one-dimensional) domain of an unary relation and the elements of one of the dimensions of the two-dimensional domain of a binary relation in the same way. On Graphs this can be done easily using the terms "vertex" or "vertices". On relations there is no equivalent notion. So we will use the terms "vertex" and "vertices" also in the case of non-graph relations. On non-graph binary relations we have to distinguish between vertices in the first and the second dimension of the domain. Therefore, if we talk about vertices in non-graph binary relations, we always give the dimension explicitly. If we do not care which kind of vertices of a relation (unary/-binary relation, first/second dimension) we are talking about then the term "vertices of the relation" is used.

Unary Relations are modeled by the *IUnaryRelation* interface and the *AUnaryRelation* abstract class. When comparing *IUnaryRelation* to *IRelation* one will notice that there is little difference. There are three reasons why *IUnaryRelation* exits though: Firstly unary relations are modeled explicitly, secondly it results in a symmetric modeling of unary and binary relations, and thirdly the difference between unary and binary relations is contrasted because if *IRelation* modeled also unary relations then binary relations would be a specialization of unary relations which is quite unintuitive.

The two basic operations of the *IUnaryRelation* interface are *Contains* which test whether some vertex is in the relation and *GetEnumerator* which is defined by the *IEnumerable* interface and is used by the C# *foreach* statement to provide iteration over all vertices of the relation. The code below demonstrates how to use these operations in practice.

Contains

```
AUnaryRelation<T> relation = //Construct relation
T vertex = //choose vertex

if (relation.Contains(vertex))
{
    ...
}
```

Iteration

```
AUnaryRelation<T> relation = //Construct relation

foreach (T vertex in relation)
{
    ...
}
```

Binary Relations are modeled by the *IBinaryRelation* interface and the *ABinaryRelation* abstract class. Compared to the *IRelation* interface, *IBinaryRelation* has a more specific *Contains* operation. The inherited *Contains* operation from the *IRelation* interface has only one argument and tests whether the *pair* of a vertex in the first and a vertex in the second dimension is in the relation. So a call of this operation looks like this:

Contains (one argument)

```
AbinaryRelation<T1,T2> relation = //Construct relation
T1 vertex1 = //choose vertex in first dimension
T2 vertex2 = //choose vertex in second dimension

if (relation.Contains(new SPair<T1,T2>(vertex1, vertex2)))
{
    ...
}
```

The specialized variant takes two arguments, the vertex in the first and the vertex in the second dimension, and is called in a more comfortable way[1]:

Contains (two arguments)

```
ABinaryRelation<T1,T2> relation = //Construct relation
T1 vertex1 = //choose vertex in first dimension
T2 vertex2 = //choose vertex in second dimension

if (relation.Contains(vertex1, vertex2))
{
    ...
}
```

Furthermore *IBinaryRelation* introduces the *Cut* operation which returns for each element/-vertex in the first dimension all vertices in the second dimension for which the pair of the two vertices is in the relation. The name comes from the following image: The domain of a binary relation can be thought of as two-dimensional plane. The elements, i.e. pairs of vertices, of the relations are points in this plane then. The "cut" at some vertex in the first dimension can be visualized as one-dimensional line at the level of this vertex parallel to the second dimension. All points (pairs of vertices) hit when "cutting" the plane on this line, or rather their second components belong to the "cut".

Consider some relations on the vertices of a graph, the adjacency or child, the reachability or descendant-or-self, the parent and the ancestor relation. If *vertex1* is a vertex of the graph then the "cut at vertex1 through the child relation" consists of all children of vertex1, the "cut through the descendant relation" consists of all descendants of vertex1, the "cut through the parent relation" consists of all parents and so on. So the *Cut* operation is a suitable abstraction for the corresponding specific operations of these relations.

[1] *SPair* is a C# structure, so **new** `SPair<T1,T2>` does not create an object and the original operation is almost as efficient as the new one.

Please note that a "cut" through a binary relation is a unary relation itself and is consequently implemented as subclass of *AUnaryRelation*. So iterating over the cut at some element of the first dimension works just the same way as for all unary relations:

Cut

```
ABinaryRelation<T1,T2> relation = //Construct relation
T1 vertex1 = //choose vertex in first dimension

foreach (T2 vertex2 in relation.Cut(vertex1))
{
    ...
}
```

2.2. Relations over Compact Integer Domains

The interfaces and abstract classes in the preceding section allow to define unary and binary relations over almost every kind of domain. By this relations over very different domains can be treated in the same uniform way. However this generality introduces two problems.

First the type of the element in the domain can become very complex and may carry significant additional information that does neither affect the representation of the relations nor a run of the evaluation on these relation. If the elements of the domain are used directly in this case the additional information significantly increases the representation of the relations and can even slow down the evaluation algorithm.

Second many storage schemes need to assign some data to every vertex or use the vertex as index to arrays or matrices. Both is difficult for arbitrary types.

The solution for both problems is to map the elements of the domain to simple identifiers. The representation of the relations can then be done on the identifiers instead of directly on the elements of the domain.

A good choice for those identifiers are integers. Every unary relation over some finite domain can be mapped to an unary relation over the domain $\{0, \ldots, d-1\}$, where d denotes the size of the domain, and every binary relation over some finite domain can be mapped to an binary relation over the domain $\{0, \ldots, d_1 - 1\} \times \{0, \ldots, d_2 - 1\}$, where d_1 denotes the size of the first and d_2 the size of the second dimension of the domain. We call that kind of domains *compact integer domains* as the use integer range has no gap.

As every unary or binary relation over a finite domain can be mapped to a relation over a compact integer domain, regardless of the type of the vertices, it is possible to split up the implementation of a data structure for some specific type into two parts: The implementation of the mapping and the implementation of efficient data structures for relations over compact integer domains. This diploma thesis concentrates on the second part. The interfaces and abstract classes representing relations over compact integer domains are *IUnaryIntRelation*, *IBinaryIntRelation*, *AUnaryIntRelation*, and *ABinaryRelation*.

Advantages of compact integer domains.

1. Integer as vertex type is efficiently supported by C# and other programming languages.
2. Integer vertices can be used directly as indices for an array or matrix.
3. More efficient iteration over all vertices of the domain or of one of its dimensions using a simple for loop instead of an iterator.
4. Due to (2.) assigning data to the integer vertices is very simple and efficient, too.

The first and the second point are quite obvious. The third is illustrated below. Both code fragments show iterations over all vertices in the first dimension of the domain of some relation. In the first fragment the first dimension is explicitly represented by an unary relation object and the iteration is done using the iterator provided by this object.

```
ABinaryIntRelation relation = //Construct relation

foreach (int v1 in relation.Domain1())
{
    ...
}
```

The second fragment uses the specific form of compact integer domains to do the same iteration with a simple for-loop. This is more efficient as neither objects nor function calls are involved in the iteration. Furthermore simple for-loops are often better supported for compiler as well as run-time (e.g. pipelining) optimizations.

```
ABinaryIntRelation relation = //Construct relation

for (int v1 = 0; v1 < relation.DomSize1; ++v1)
{
    ...
}
```

The last point is discussed in the following paragraphs.

Assigning data to vertices – labels. Many data structures need to assign some data to the vertices of a relation. This is particularly true for labeling schemes which assign some label to each vertex. Frequently the label or generally the assigned data consist of several parts. The SIC data structure for example uses a label consisting of three integers. We assume that most operation will need the whole label or data at the same time and this assumption is at least true for PPE, SIC, MIC and some other data structures. So for each vertex all parts of the label should be stored in a continous memory segment because the time for loading the complete label is minimized in this way.

Of course every storage scheme could care for the locality of its data or label on its own. However we do not only want that the data assigned to a vertex by one storage scheme is stored in a continous memory segment, but want to ensure that continuity also when

several storage schemes are used at the same time: We also wish to continously store all data assigned to a vertex by all the storage schemes. Sometimes this memory arrangement is not optimal and therefore the choice whether this feature is used or not is left to the user. The support of continously storage for severeral relations is related to the third requirement for a uniform interface (see the beginning of this section) and is further detailed in Section 2.3.

For combining the data of different storage schemes automatically we need to know how this data is stored. The *ABinaryLabelRelation* class is introduced to solve this problem[2]. The class assumes that the data or the label assigned to a vertex consists of (or at least may be encoded as[3]). For each dimension of the domain, one two-dimensional array, called *vertex descriptor*, is used to store the data assigned to the vertices in that dimension. The first index of the array corresponds to the vertex and the second index to an entry of the assigned data. The way two-dimensional arrays are mapped to memory guarantees that the data for each vertex is stored continuously.

The array may contain data from other then the current storage schemes. In this way we are able to continuously store the data assigned to a vertex by *different* storage scheme or different instantiations of the same storage scheme. An offset, called *vertex descriptor offset*, determines the start of the data for the current storage scheme in the current *vertex descriptor*.

In case of a graph using these offsets has also another advantage. When a graph is represented as relation then its vertices occur twice in this representaion: Once in the first and once in the second dimension. Some labeling scheme assign data to the vertices in the first as well as to the vertices in the second dimension of the domain of a relation. SIC for instance computes a special ordering of the vertices in the second dimension and assigns an interval in this ordering to each vertex in the first dimension and its corresponding position in the ordering to each vertex in the second dimension. But for a graph the vertices of both dimensions are just the same and so the data assigned to a vertex in the first dimension and the data assigned to it in the second dimension should be stored together. This can be achieve easily by using the same array with different offsets for both dimensions.

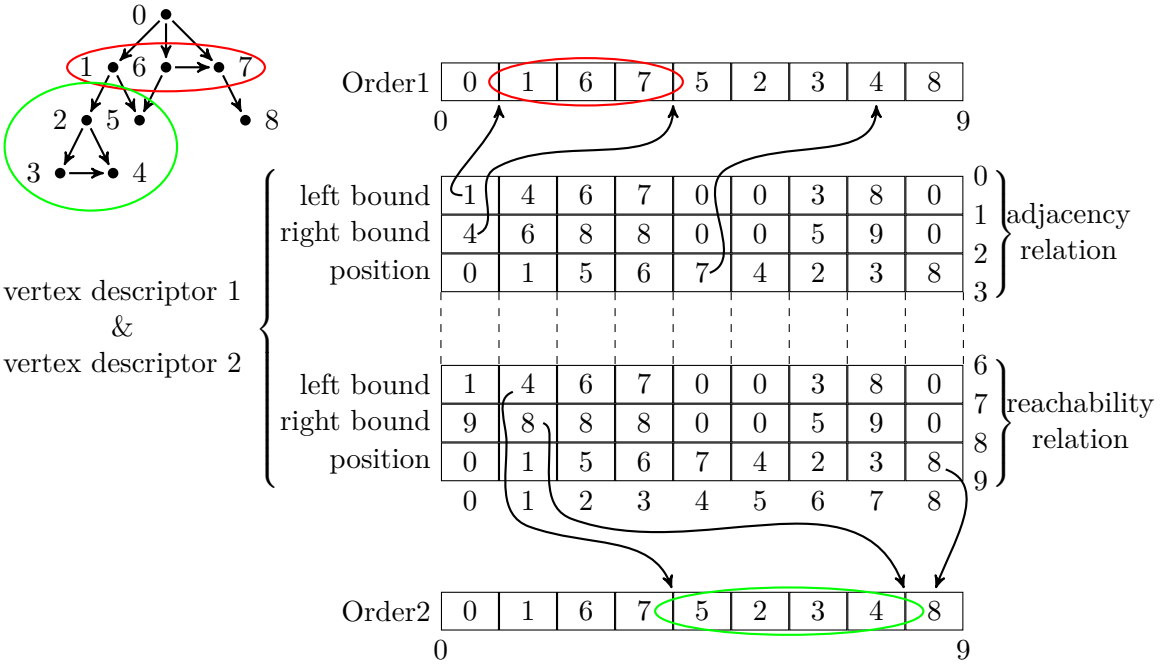
To illustratte this, the following figure shows the adjacency and the reachability relation of a graph each represented in one SIC storage scheme. The graph and the the representation of its adjacency relation is already known from the introduction and the representation of the reachability relation is discussed in Section 3.4. Here we do not focus on the storage schemes but on the way their data is stored. SIC assigns two integer values to each vertex in the first dimension of the domain, namely the left and right bound of the corresponing interval, and one integer value, the position in the order, to each vertex in the second dimension.

In case of a graph, as we find it here, the vertices of the first and second dimension are the same and all three values should be stored together, i.e., the same vertex descriptor array is used for both dimensions. The relation still holds two references, *vertex descriptor 1* for the first dimension and *vertex descriptor 2* for the second, which both point to the same array, though. This is achieved by the right use of the vertex descriptor offsets. Here the vertex descriptor offset of the first dimension of the adjacency relation is 0, the one for the second

[2]Currently the implementation does not contain an equivalent class for unary relations though this might be a good idea.

[3]The so-called "unsafe" features of C# may help to find an efficient encoding.

dimension is 2. For the reachability relation 6 is the offset of the first and 8 the offset of the second dimension. The dashed lines indicate that there is some furter relation which is stored between the adjacency and the decendant relation.



The most important thing to be shown by the figure is, that (1) the data or the label assigned to a vertex by *each* storage scheme is placed continously and that (2) the data or the label assigned to a vertex by *several* storage scheme is placed continously, too.

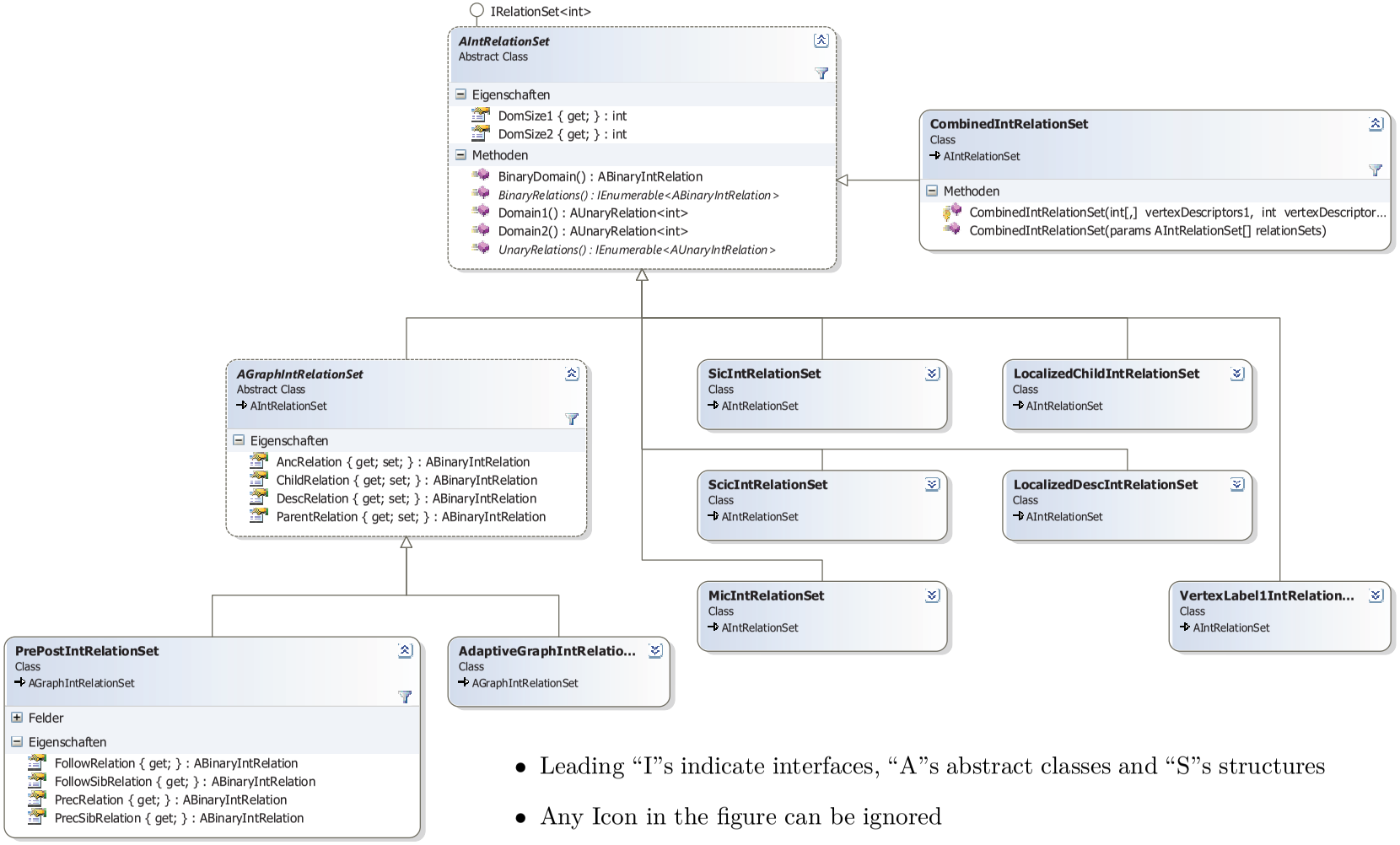
2.3. Sets of Relations

As already stated for several times one is frequently interessted in a certain set of relations and not only in a single one. The preceding paragraphs, which introduced the *ABinaryLabelRelation* class described how memory is organized to enable the combination of data from different storage schemes. The subclasses of *ABinaryLabelRelation* know where their data is located and how it is structured internally. However they do not know about each other, meaning they do not know which other relations exit and where their data is located. They just get told the place where to store their data during initialization.

The generic *IRelationSet* interface models sets of relations. The abstract *AIntRelationSet* class implements this interface for sets of relations over compact integer domains.[4] A relation set knows which relations are available and where they are stored. It does the administrative work during initialisation, i.e., tells each relation where to store its data. The following figure gives an overview over the relation set classes.

[4] The main purpose of the relation sets is to allow data sharing for different relations of the same storage scheme and a continous arrangement of data from several storage schemes. This can hardly be done without knowing about the structure of the data. Therefore we did not implement a more general generic abstract class under the interface here.

Relation Sets – Overview



- Leading “I”s indicate interfaces, “A”s abstract classes and “S”s structures
- Any Icon in the figure can be ignored

This is done in two steps. First there is one non-abstract subclass of *AIntRelationSet* for each implemented storage scheme. This class knows the size of the data or label which is assigned to each vertex by the storage scheme. Furthermore it gives access to the relations provided by the storage scheme. The *PrePostIntRelationSet* for instance provides 8 relations.

This example also shows a the second task of relation sets: They enable data sharing between different relations of one storage scheme. So the three integers assigned to each vertex by PPE are stored only once and are commonly used by all its relations.

The second step is the combination of different basic relation sets. The abstract *AGraphIntRelationSet* class, for example, guarantees access to four important relations on a graph. An implementing subclass, *AdaptiveIntGraphRelationSet*, fullfills this requirement by searching a suitable representation for each relation and finally combining their data. The combination of the data is mainly a copying task on the vertex descriptor arrays.

2.4. Constructor Conventions

Our framework provides a uniform access to the graph data stored in differen storage schemes. We want to keep uniformity for the initializiation of the storage schemes, too. This significantly simplifies the usage of different data structures as the user needs only a minimum of specific knowledge on the implementing class of some storage scheme. Therefore we define some constructor signatures that a subclass of *ABinaryIntRelation* and *ABinaryLabelRelation* should support by convention[5].

Subclasses of *ABinaryIntRelation*. should support a constructor which creates a new instance of the class based on any object that is an instance of *ABinaryIntRelation*. The represented relation is not necessarily the same but of course there has to be a direct connection between both relations. The provided object could, for example represent the adjacency relation of some graph whereas the constructed instance of the subclass of ABinaryIntRelation represents the reachability relation on the same graph.

The initial relation. Obviously a constructor creating new instances of *ABinaryIntRelation* from old ones is not very helpful for the construction of some initial relation. There are some classes intended as entry point under the *ABinaryIntRelation* interface namely the *AdjacencyMatrix* (Sec. 3.1) class and all classes implementing adjacency lists (or arrays, Sec. 3.2). The *UnorderedAddableAdjacencyArray* class is particularly interesting when the number of vertices is unknown in the beginning as it is usually when parsing some XML, RDF or other document. The *UnorderedAddableAdjacencyArray* allows to add vertices to both dimensions of the domain (in case of a graph a vertex is added to both dimensions at the same time) and to add pairs of vertices (edges) to the relation. The following code fragment illustrates the usage of the *UnorderedAddableAdjacencyArray* class and of the constructor recommended for subclasses of *ABinaryIntRelation*.

```
UnorderedAddableAdjacencyArray rel =
                          new UnorderedAddableAdjacencyArray();

rel.AddVertex();            \\add first vertex
rel.AddVertex();            \\add second vertex
rel.AddEdge(0, 1);          \\add edge from the first
                            \\to the second vertex

Sic sicRel = new Sic(rel); \\creating an instance of Sic that
                            \\represents the same relation
```

Subclasses of *ABinaryLabelRelation*. are recommended to support a further constructor. This constructor takes an instance of *ABinaryIntRelation*, two vertex descriptor (an two-dimensional array, one per dimension) and the intended offsets (two integers) of the current relation in the vertex descriptors. This constructor is meant for the implementation of relation sets.

2.5. The GRAPH class

The static *GRAPH* class provides some useful operations on graphs. The *DescRelation* operation for instance computes a representation of the descendant (not self) relation in time $O(m \cdot n + n)$ and Space $O(n^2)$, where n is the number of vertices in the graph and m the number of edges. However the more important operation is *Encode*. Provided an instance of *AIntRelation* which represents a graph, *Encode* looks for efficient representations of the child, the descendant, the parent and the ancestor relation of the graph and finally returns an instance of *AGraphIntRelationSet* giving access to them.[6]

[6] The set of child, descendant, parent and ancestor relation seemed reasonable to us and searching for an representation of several relations at the same time can have a better ,so more compact, result than searching for representations of the single relations independently because the different relations may share data. Sometimes operations similar to *Encode* for the single relations can useful and they should not be hard to implement but we have not done so yet.

3. Data Structures

In this section, we present the implemented data structures. We discuss their main properties, provide a brief introduction into their usage and address some details mostly concerning optimizations. The table on Page 26 gives an overview on the time and space requirements of the data structures and the figure on Page 26 shows the implementing classes in the context of their type hierarchy.

Ordered Iteration.[7] As the vertices are integers they have a natural ordering. When some iteration returns its vertices in ascending order with respect to this ordering, the iteration is said to be *in order*. When it does so in descending order it is said to be in *reverse order*.

3.1. Adjacency Matrix

Every relation $R \subseteq V \times W$ may be represented by a matrix $A = (a_{v,w})_{v \in V, w \in W} \in \{0,1\}^{|V| \times |W|}$, telling for each pair $(v, w) \in V \times W$ whether $(v, w) \in R$ or not: $a_{v,w} = 1 \Leftrightarrow (v, w) \in R$.

Obviously such a $0, 1$-matrix can be implemented using exactly $|V| \cdot |W|$ bits, that is $\lceil (|V| \cdot |W|) / 32 \rceil \cdot 32$ bits in practice, when we assume a word size of 32 bits.

In contrast a naive C# implementation as boolean matrix or array will use $|V| \cdot |W|$ bytes, i.e., eight times as much memory.

The implementation given by the *SBitMatrix* structure does some alignment to improve sequential read- and write-operations on row and columns. Therefore it uses $|V| \cdot (\lceil |W| / 32 \rceil \cdot 32)$ bits, so a little bit more than absolutely necessary.

The concrete memory organization is as follows: Rows are prioritized over columns and for that the rows are placed one after the other each written sequentially. Furthermore the beginning of each row is aligned to a word boundary. This allows faster writing of rows and improved reading and writing of columns.

The implementation is intended to read and write block-, i.e., word-wise where ever possible. In many cases this is already implemented.

The *AdjacencyMatrix* class finally wraps the *SBitMatrix* structure to the *ABinaryIntRelation* interface.

[7] Some query languages like XPath care about the order in which vertices are returned. When a node in some document is represented by the integer vertex which corresponds to the position of the node in document order, then ordered iteration in the sense above is iteration in document order.

Data Structures – Overview Complexity

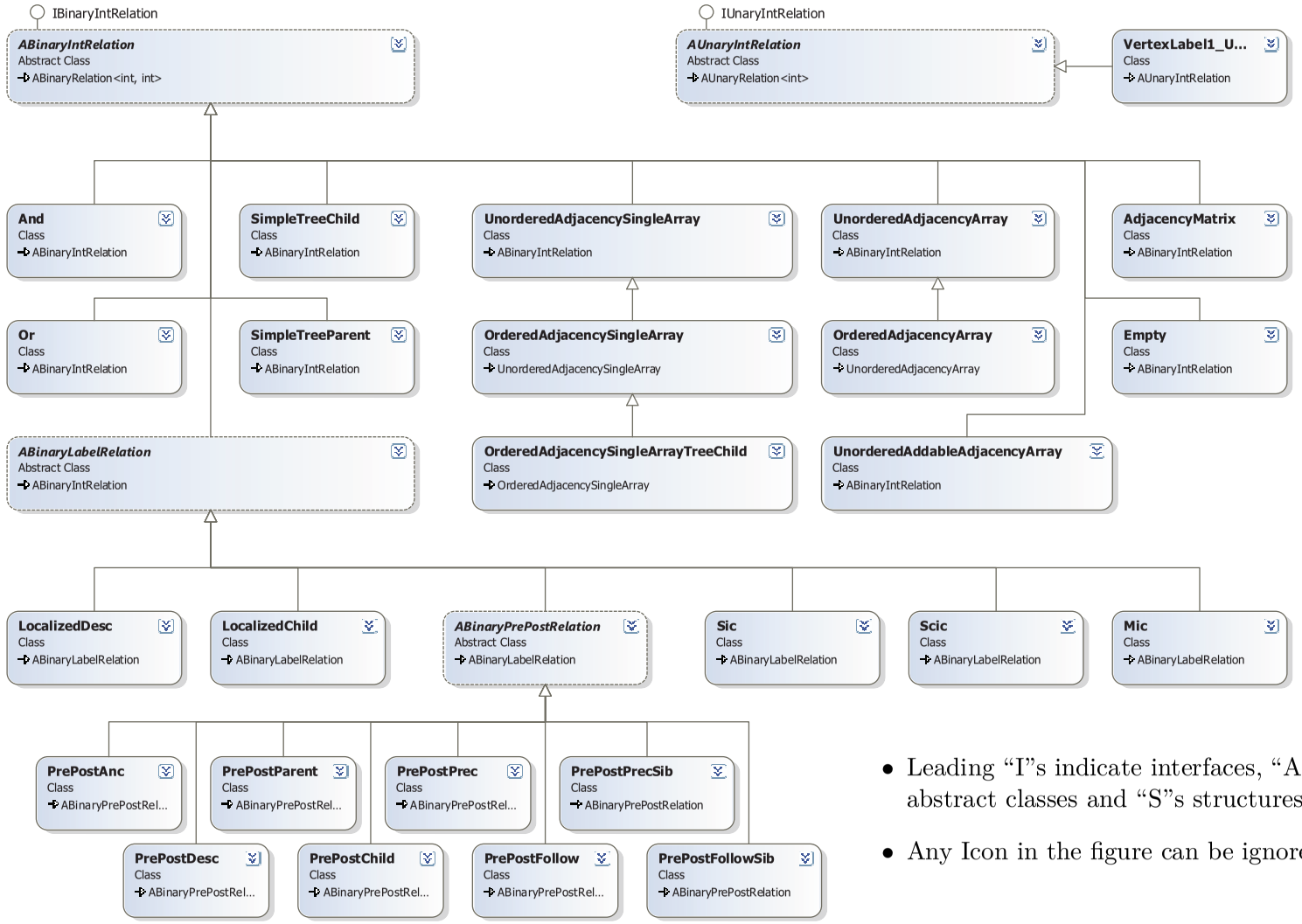
Implementing Class	Rel.	Test	Cut Iteration	Order	Space	Index	Rel. Class
Matrix	R	$O(1)$	$O(n)$	$\nearrow$	$O(n^2)$	$O(m)$	All
UnorderedAdjacencyArray	R	$O(n)$	$O(\|R[v]\|)$	-	$O(n+m)$	$O(n+m)$	All
UnorderedAdjacencySingleArray	R	$O(n)$	$O(\|R[v]\|)$	-	$O(n+m)$	$O(n+m)$	All
UnorderedAddableAdjacencyArray	R	$O(n)$	$O(\|R[v]\|)$	-	$O(n+m)$	$O(n+m)$	All
OrderedAdjacencyArray	R	$O(log(n))$	$O(\|R[v]\|)$	$\nearrow$	$O(n+m)$	$O(n+m)$	All
OrderedAdjacencySingleArray	R	$O(log(n))$	$O(\|R[v]\|)$	$\nearrow$	$O(n+m)$	$O(n+m)$	All
Sic	R	$O(1)$	$O(\|R[v]\|)$	-	$O(n)$	$O(n+m)$	CIR
Scic	R	$O(1)$	$O(\|R[v]\|)$	-	$O(n)$	$O(n+m)$	CCIR
Mic	R	$O(log(n))$	$O(\|R[v]\|)$	-	$O(n+m)$	$O(n^3)$	All
SimpleTreeChild$^\Diamond$	G	$O(1)$	$O(n)$	$\nearrow$	$O(n)$	$O(n)$	Forest
SimpleTreeParent$^\Diamond$	G^{-1}	$O(1)$	$O(1)$	$\nearrow$	$O(n)$	$O(n)$	Forest
OrderdAdjacencySingleArrayTreeChild$^\Diamond$	G	$O(1)$	$O(\|G[v]\|)$	$\nearrow$	$O(n)$	$O(n)$	Forest
PrePostAnc	$(G^{-1})^*$	$O(1)$	$O(\|(G^{-1})^*[v]\|)$	$\searrow$ $^\circ$	$O(n)$	$O(n)$	Forest
PrePostDesc	G^*	$O(1)$	$O(\|G^*[v]\|)$	$\nearrow$ $^\circ$	$O(n)$	$O(n)$	Forest
PrePostParent	G^{-1}	$O(1)$	$O(1)$	$\nearrow$ $^\circ$	$O(n)$	$O(n)$	Forest
PrePostChild	G	$O(1)$	$O(\|G^*[v]\|)$	$\nearrow$ $^\circ$	$O(n)$	$O(n)$	Forest
LocalizedChild	G	$O(1+log(n_v))$	$O(\|G[v]\|)$	-	$O(n+(n+m))$	$O(n)+LC$	Graph
LocalizedDesc	G^*	$O(1+log(\tilde{n}))$	$O(\|G^*[v]\|)$	-	$O(n+(\tilde{n}+\tilde{m}^*))$	$O(n)+GC$	Graph

$(G^{-1})^*$ = ancestor relation
G^* = descendant relation
G^{-1} = parent relation

n_v = number of nodes in the local component of v
$\tilde{n}$ = number of strong components
$\tilde{m}^*$ = number of edges between strong components

$^\circ$ under special conditions, see Sec. 3.3
$^\Diamond$ basic data structure for trees which is not discussed further

Data Structures – Hierarchical Overview



- Leading “I”s indicate interfaces, “A”s abstract classes and “S”s structures
- Any Icon in the figure can be ignored

3.2. Adjacency Array

There are five classes that implement the adjacency array (or adjacency list) datastructure. They differ in three respects: First whether the vertices are stored in order or not, second whether a single array or nested arrays are used for storage and third whether vertices and edges may be added dynamically. The following table gives an overview.

Class	ordered	array	dynamic
UnorderedAdjacencyArray	-	nested	-
UnorderedAdjacencySingleArray	-	single	-
UnorderedAddableAdjacencyArray	-	nested	+
OrderedAdjacencyArray	+	nested	-
OrderedAdjacencySingleArray	+	single	-

Ordered vs. unordered. If the vertices are stored in order then (1) testing whether some pair of vertices is in the relation or not can be done by binary search instead of linear search (and thus in logarithmic instead of linear time) and (2) iterations over cuts are in order. As expected, the initialisation of the ordered variants takes a bit longer than for the unordered ones. But surprisingly the complexity for the initialisation from any instance of *ABinaryIntRelation* is the same for the unordered and ordered variants. If the original relation object iterates over cuts in time linear to the size of the cut, then the time for initialisation is $O(n + m)$. The reason for this is, that the vertices of the first dimension are already implicitly ordered. When constructing the inverse relation then the vertices in the second dimension, i.e., the vertices in the first dimension of the inverse relation, become ordered. The vertices in the first dimention, i.e., the vertices in the second dimension of the inverse relation, remain ordered. So the inverse relation is ordered in both dimensions. Another inversion returns the ordered original relation.

Nested array vs. single array. There are two possibilities how the lists of adjacent vertices for each node can be stored in an array structure. The first is to assign each vertex an array containing all its adjacent vertices. The assignment is done by an array again and so we get an array of arrays (this means we use nested arrays). The second possibility is to store all lists in a single array placing them one after the other. Then for each vertex one has to memorize the start and end index of its list. As the end index of a vertex is the start index of the following one, storing the indices for all nodes can be done in one integer array with one entry per node.

This variant is more effient, because C# does not access array directly but over some array descriptor. The single array variant needs only two array descriptors compared with one global array descriptor and one array descriptor per vertex in the nested array variant. So the single array variant should save some memory because it has to store only one additional integer per vertex and an array descriptor probably needs more space than a single integer. Furthermore the workload for the C# object management is reduced. Finally the single array variant profits more form sequencial iterations over the vertices.

3.3. Pre-Post-Encoding (PPE)

PPE is able to represent several relations on trees. It does so by defining two orderings on the vertices of the tree, called pre- and post order, and assigning to the vertices their position in the orderings. Furthermore the parent of each vertex is stored. This is the PPE variant described in [**?**]. PPE uses the three integer values per vertex, called label, to represent the child (adjacency), the descendant, the parent, and the ancestor relation. Each of these relations corresponds to one class which knows how to determine the relation out of the labels for the vertices. For an illustration see Page 8 in the introduction.

The PPE classes supports the recommended constructors. However when more than one of the relations is needed using these constructors is discouraged because each of the relations will then construct and hold its own copy of the encoding. To obtain instances of the relations that share a single copy of the encoding the constructors of the *PrePostIntRelationSetClass* have to be used.

Ordered iteration. over cuts is provided under special conditions:

1. for all vertices v, $v < \mathit{descendants}(v)$
2. for all vertices v_1, v_2 with $v_1 < v_2$, $v_2 \in \mathit{descendants}(v_1)$ or $\mathit{descendants}(v_1) < v_2$

When each node of an XML document is represented by a vertex (an integer) that corresponds to its position in document order and the represented tree is given by the document structure then these conditions are fulfilled naturally.

In this case Pre-Post-Encoding may also represent the preceding, the following, the preceding sibling and the following sibling relation. The corresponding classes are implemented, too.

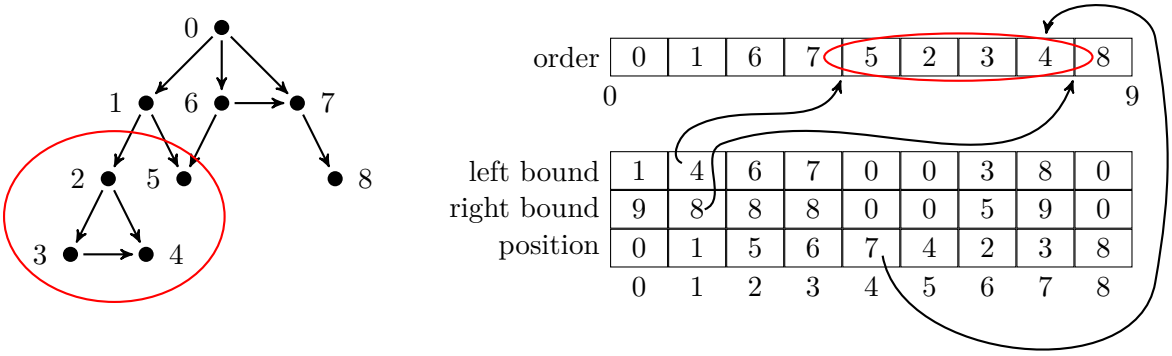
3.4. Single Interval Compression (SIC)

The SIC datastructure uses a special property of so-called Continuous-Image-Graphs (CIGs) for efficient storage of those graphs. CIGs are a real superclass of trees and forest. Their special property is the existence of an ordering in which for each vertex all adjacent vertices fall in one continuous interval. This property can be generalized to non-graph relations. In this case we need an ordering of the vetices of the second dimension in which for each vertex of the first dimension all adjacent vertices fall in one continuous interval. A relation that satisfies this property is called Continuous-Image-Relation (CIR). Obviously every CIG is a CIR. Formal definitions of CIRs and CIGs can be found in Section 4.

Constructing a suitable ordering can be done in time linear to the number of edges in the graph or the number of elements (pairs of vertices) in the relation [**?**]. However the implementation of the PC-Tree data structure which is proposed there is left open by this diploma thesis. The static CCIR class provides an currently unimplemented function *Compute* which is intended to do the computation of the ordering and is meant as entry point to a future implementation of the PC-Tree.

On Page 9 of the introduction is an illustration for the behavior of SIC on the adjacency or child relation of some graph. Here we look on the descendant relation of the same graph and find that (happily) the relations is a CIR, too. Please note that when the child relation

of a graph is representable by SIC this does not have to be true for its descendant relation, too. What is shown by the example is that SIC does not necessarily suffer from more edges. In particular the size of the representation remains the same.



Besides the fact that the adjacency relation as well as the descendant relation of this graph are CIRs the example is special in another way, too. The representations of both relations are based on the same ordering of the vertices. It is unclear how frequently a common ordering for more than one relation can be found. We know that finding a common ordering for the child and the descendant relation is always possible on trees and forest and at least on some other graphs as shown above. However a common ordering for a number of relations has two advantages: First, sharing the order reduces the needed space. Second, when different relations are combined in the evaluation of some query there is a higher chance to profit from sequencial memory access. Therefore this point should be examined in future.

3.5. Multi Interval Compression (MIC)

The idea of MIC is to extend SIC to abitrary graphs by allowing multiple intervals per vertex. The second figure on Page 10 of the introduction illustrates the MIC datastructure. The main task of MIC is to find an ordering that needs a minimum total number of intervals for the representation of the graph. Solving this problem, exactly is very expensive in time as it is NP-complete [**?**]. However there are good approximations [**?**]. These approximations are based on an approximation-preserving transformation to the traveling salesman problem (TS). The TS is 1.5-approximable by the Christofides-Algorithm [**?**].

The Christofides-Algorithm needs to solve the minimum cost perfect matching problem. The efficient implementation of the minimum cost perfect matching problem has been discussed in noumerous articles most mentioned in this very recent paper [**?**]. [**?**] presents an highly optimized algorithm for the minimum cost perfect matching algorithms called Blossom V. Blossom V is the latest version of a familly of algorithms that has been improved over decades. The implementation is freely available and so we recommend to use it instead of developing a new one. The integration of the BlossomV code is not done in this diploma thesis. However the Christofides Algorithm is implemented up to minimum cost perfect matching, leaving only one function in the static *MATCHING* class unimplememted.

A simplification of the Christofides-Algorithm which works without solving minimum matching has implemented, though [**?**]. This implementations provides a 2-approximation for

TS. Therefore our implementation is currently able to find an interval representation for a graph, which needs at most twice as many intervals as needed minimally.

MIC determines wheter a pair of two vertices is in the relation or not by searching an interval of the first vertex which contains the second vertex or rather its position in the ordering. The search can be done in logarithmic time as the intervals are stored in order. MIC has the same worst-case performance in time and space as the ordered variants of adjacency array. However we expect a much better behavior in practice. Particularly on dense graphs MIC should be superior.

3.6. Localized Child

The Localized Child data structure tries to adapt to the local structure of the graph. The idea is to decompose the graph into several components and to leave the storage of each component to the most suitable scheme. The basic behavior of the Localized Child data-structure is already described on Page 11. So we concentrate on some specific details of the implementation here.

Decomposing the Graph. In an undirected graph bridges are those edges of the graph which are the only connection between two parts of the graph. When removed they will cause the graph to decompose in several components. So the decomposition performed by the Localized Child data structure is based on the idea to consider the originally directed graph undirected, to identify all bridges of the undirected graph and to remove all directed edges from the original graph which correspond to one of the bridges.

Finding the bridges of an undirected graph can be done in linear time. However we choose another approach here. Instead of searching the bridges, we compute a spanning tree/forest for the undirected graph[8]. The spanning tree necessarily contains all bridges. We remove all directed edges from the original graph which correspond to an edge in the spanning tree[8]. The graph decomposes into at least as many components as it did if only the bridges were removed. But we have a real chance to achieve a better decomposition.

Now we drop all edges of the spanning tree/forest between vertices in the same component and reinsert the corresponding directed edges in the graph. The remaining edges of the spanning tree/forest still form an forest. Furthermore the reinsertion does not change the components of the decomposed graph. After that each of these component is stored independently in the datastructure which is considered most suitable.

How the removed directed edges of the original graph are stored? All these directed edges run along edges in the constructed forest. Each component of the forest has a root[8]. We split the directed edges into edges that run towards the roots and edges running towards

[8] In fact we work on the symmetric graph of the original graph. For each edge in the original graph the symmetric graph contains the edge and the inverse edge. In other words the symmetric graphs contains edges in both directions for each edge of the undirected graph. We compute the directed spanning tree/-forest of the symmetric graph, which is a spanning tree/forest of the undirected graph when considered undirected. To each (directed) edge of the spanning tree/forest correspond all (at most two) edges in the original graph between the same two vertices. As the spanning tree/forest is directed, each of its components has a root and the same holds when some edges are removed from it.

the leaves. Both sets of edges form a directed forest, one with edges pointing towards the roots of the forest and one with edges pointing to the leaves. Both directed forests can be stored efficiently.

Page 32 gives a pseudo-code illustration of the initialization. The pseudo-code below shows how to determine whether some edge is in the graph or not.

How to apply. By removing bridges the Localized Child exploits some (undirected) tree structure of the graph. By the use of the spanning tree we furthermore have a chance to isolate parts of the graph which are only weakly connected to the exterior but lie on an undirected circle. However the decomposition will work best on quite sparse or clustered graphs. Applying the Localized Child data structure to the descendant or reachability relation of a graph does not make much sense, though. As the reachability relation is transitive it contains almost no tree structure and is neither sparse and nor clustered.

Pseudo-code for Contains.

```
DT = downward tree
UT = upward tree

Contains(v1, v2)
{
   if (v1, v2) is in DT or (v2, v1) is in UT return true
   if v1 and v2 are in the same component
   {
      if the component contains (v1, v2) return true
   }
   return false
}
```

Pseudo-code for Initialisation.

```
G = the original Graph
SG = new empty graph with same vertices as G
// SG for symmetric graph

for each edge (v1, v2) in G
{ //construct symmetric graph
   add (v1, v2) to UG
   add (v2, v1) to UG
}

T = spanning tree of UG

DG = copy of G. //DG for decomposed graph
```

```
for each edge (v1, v2) in T
{ //remove vertices that correspond to an edge in T
   remove (v1, v2) from DG
   remove (v2, v1) from DG
}

search components of DG

for each edge (v1, v2) in T
{ //reinserting edges between vertices in the same component
   if(v1 and v2 are in the same component)
   {
      remove (v1, v2) from T
      if (v1, v2) is an edge of G
      {
         add (v1, v2) to DG
      }
      if (v2, v1) is an edge of G
      {
         add (v2, v1) to DG
      }
   }
}

DT = empty tree // DT for downward tree
UT = empty tree // UT for upward tree

for each edge (v1, v2) in T
{ //partition the removed edges
   if (v1, v2) is an edge of G
   {
      add (v1, v2) to DT
   }
   if (v2, v1) is an edge of G
   {
      add (v1, v2) to UT
   }
}

store DT and UT

for each component of DG
{
   choose any suitable data structure for the component
   store component
}
```

3.7. Localized Descendant Relation

The Localized Descendant data structure is designed to efficiently represent the descendant relation of a graph. This is done in two steps:

First the strongly connected components[9] of the graph are collapsed to a single representing vertex. The idea is that all vertices of a strongly connected component have the same set of descendants and the same set of ancestors and therefore can be merged for the descendant relation. The resulting graph is a directed acyclic graph (DAG). The corresponding strongly connected component (or rather its representing vertex) is assigned to each vertex of the original graph. Furthermore the set of contained nodes is stored for each component. Both takes linear space in the size of the original graph. This approach is commonly used by datastructures for the representation of the descendant or reachability relation [**?**, **?**, **?**].

Second the descendant relation on the DAG has to be represented. In contrast to previous approaches [**?**, **?**, **?**] which always use some specific representation for the DAG we are able to choose the representation we consider most suitable. This is due to our uniform interface. Currently we have three data structures implemented which could be used, namely SIC, MIC and the adjacency matrix structure. The adjacency matrix is only a good choice if the number of strongly connected components is very small. The following pseudo-code illustrates the complete procedure.

```
G = the original Graph
DAG = new empty graph

compute strongly connected components of G

for each component of G
{
   add component identifier as vertex to DAG
}

for each edge (v1, v2) in G
{
   c1 = identifier of component of v1
   c2 = identifier of component of v2
   add (c1, c2) to DAG
}

choose any suitable data structure for DAG
store DAG
```

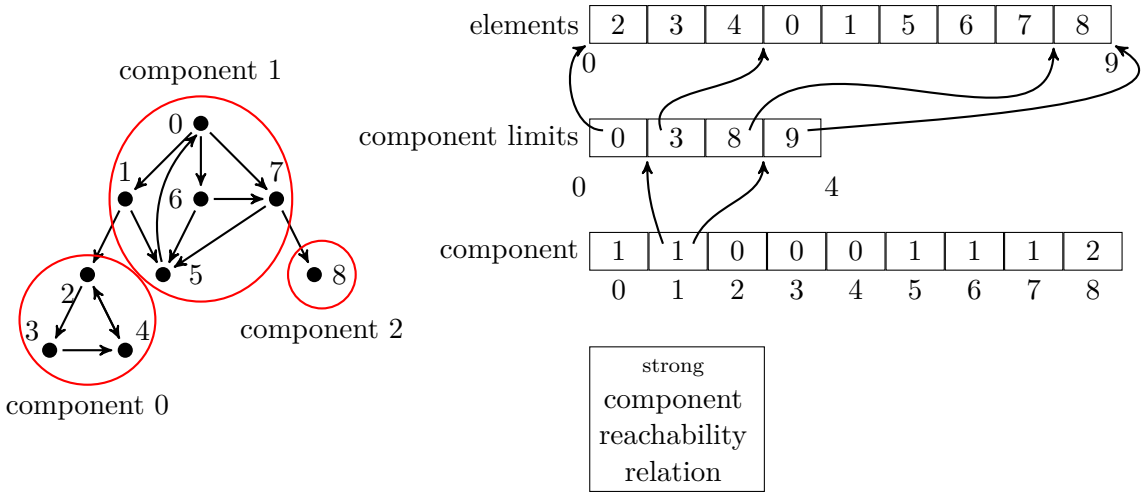
SIC is the most attractive variant as it only takes linear space in the number of strongly connected components. SIC may only be applied when the descendant relation on the DAG is a CIR. We find an interessting property about this: If the descendant relation on the DAG is a CIR then the descendant relation of the original graph is a CIR, too. The converse is

[9] Tarjan's algorithm is used to find the strongly connected components in linear time.

also true. We show this in Theorem 25 on Page 25. When the number of strongly connected components is significantly smaller than the number of vertices then using SIC after collapsing the components will result in a further compression. In the other case using SIC direcly will be more efficient. However the computation of the needed ordering for SIC may always be done after collapsing the components because an ordering on the components can easily be transformed into an ordering on the original graph.

MIC is an interessting data structure for the decendant relation, too. This is as MIC may profit from a large number of edges. [**?**] also proposed an storage scheme using several intervals per node on one ordering. The computation of the intervals is inspired by Pre-Post-Encoding there. The resulting representation looks quite similar, though. It seems as if MIC would solve exactly those problems left open by [**?**] and that it may be viewed as an improvement of the data structure which was proposed there. However there is one big difference between both storage schemes: MIC is applicable to arbitrary relations, the data structure proposed in [**?**] is restriced to the descendant relation. Nevertheless a closer comparison of both structures should be done in future.

The following figure illustrates the Localized Descendant storage scheme. The *component* array assigns to each vertex its corresponding strongly connected component or rather the identifier for the component. The *element* array contains the vertices grouped by their strong components and the *component limits* array assigns to each component identifier the corresponding interval in the *element* array. We use the fact there that the end of the interval for one component is the beginning of the interval for the following component. So we need only one entry per component.



Consider we want to find all descendants of 6 (which are just all vertices of the graph). 6 is contained in the strongly connected component with identifier 1 denoted component 1 here. The decendants of 6 are contained in component 1 and in all components that are decendants of component 1 i.e. 0 and 2. Testing whether some vertex is decendant of 6 is done as follows: If the vertex is in the same component as 6 or its component is a decendant of component 1, the component of 6, then it is a descendant of 6.

4. Theory

In this Section we provide formal definitions of CIGs and CIRs. We examine their behavior when vertices are removed or duplicated. This leads to the very general result that homomorphisms preserve CIGs. We use it to show that collapsing the strongly connected components of a graph is compatible with the CIR representation of its transitive closure which is important for the Localized Descendant storage scheme in Section 3.7.

Definition 1
For $s, e \in \mathbb{N}$, we define the half open interval

$$[s, e[:= \begin{cases} \{n \in \mathbb{N} \mid s \leq n < e\} & \text{if } s \leq e \\ \mathbb{N} \cup \{-1\} \setminus [e, s[\ = \ \{n \in \mathbb{N} \mid s \leq n\} \cup \{n \in \mathbb{N} \cup \{-1\} \mid n < e\} & \text{if } s > e \end{cases}$$

Notation 2
Let $R \subseteq V \times W$ be a relation, then

- $R^{-1} := \{(w, v) \mid (v, w) \in R\}$ is the *inverse relation* of R.
- $R[V'] := \bigcup_{v \in V'} \{w \mid (v, w) \in R\}$ is the *image* of $V' \subseteq V$ under R.
- $R^{-1}[W'] := \bigcup_{w \in W'} \{v \mid (v, w) \in R\}$ is the *inverse image* of $W' \subseteq W$ under R.

Definition 3 (Cut)
Let $R \subseteq V \times W$ be a relation and $v \in V$. Then $R|_v := \{w \in W \mid (v, w) \in R\} = R[\{v\}]$ defines the *cut at v through R*.

Definition 4 (Continuous Image Relation (CIR))
A relation $R \subseteq V \times W$ is called *Continuous Image Relation* (CIR) if there exist an injective function $o : W \to \mathbb{N}$ and functions $s, e : V \to \mathbb{N}$, $s(v) \leq e(v)$ for each $v \in V$ with

$$(v, w) \in R \Leftrightarrow o(w) \in [s(v), e(v)[$$

o, s and e are called a *CIR-embedding* of R.

Definition 5 (Circular Continuous Image Relation (CCIR))
A relation $R \subseteq V \times W$ is called *Circular Continuous Image Relation* (CCIR) if there exist an inj. function $o : W \to \mathbb{N}$ and functions $s : V \to \mathbb{N}$, $e : V \to \mathbb{N} \cup \{-1\}$, for each $v \in V$ with

$$(v, w) \in R \Leftrightarrow o(w) \in [s(v), e(v)[$$

o, s and e are called a *CCIR-embedding* of R.

Notation 6
We will often write s_v instead of $s(v)$ and e_v instead of $e(v)$.

Theorem 7 (Restriction)
Let $R \subseteq V \times W$ be a CIR (or CCIR) and $V' \subseteq V$, $W' \subseteq W$.
Then $R' := R|_{V' \times W'}$ is also a CIR (or CCIR).

Proof
Let o, s and e be a CIR-embedding (CCIR-embedding) of R.
Define

$o' := o|_{W'}$

$s'_v := s_v$, $e'_v := e_v$ for $v \in V'$

Then for all $v \in V'$, $w \in W'$:

$(v,w) \in R' \Leftrightarrow (v,w) \in R \Leftrightarrow o(w) \in [s_v, e_v[\Leftrightarrow o(w) \in [s'_v, e'_v[$

□

Definition 8 (Normalized Intervals)
The CIR-embedding (CCIR-embedding) o , s_v, e_v of an Relation $R \subseteq V \times W$ is called *interval normalized* if

1. $o^{-1}\left[[s_v, e_v[\right] = \emptyset \;\Rightarrow\; s_v = e_v = 0$
2. $o^{-1}\left[[s_v, e_v[\right] \neq \emptyset,\; s_v \leq e_v \;\Rightarrow\; s_v \in o[W], (e_v - 1) \in o[W]$
3. $o^{-1}\left[[s_v, e_v[\right] = W,\; s_v > e_v \;\Rightarrow\; s_v = 0, e_v = -1$
4. $o^{-1}\left[[s_v, e_v[\right] \neq \emptyset,\; o^{-1}\left[[s_v, e_v[\right] \neq W,\; s_v > e_v \;\Rightarrow\; (s_v - 1) \in o[W], e_v \in o[W]$

Theorem 9 (Normalized Intervals)
Each CIR (CCIR) R has an *interval normalized* CIR-embedding (CCIR-embedding).

Proof
Let o, s and e be a CIR-embedding (CCIR-embedding) of R.
Define

$o' := o$

and

if $o^{-1}\left[[s_v, e_v[\right] = \emptyset$:

$s'_v := 0$, $e'_v := 0$

if $o^{-1}\left[[s_v, e_v[\right] \neq \emptyset,\; s_v \leq e_v$:

$s'_v := min(\{o(w) \mid w \in W, o(w) \geq s_v\})$
$e'_v := max(\{o(w) \mid w \in W, o(w) < e_v\}) + 1$

if $o^{-1}\left[[s_v, e_v[\right] = W,\; s_v > e_v$:

$s'_v := 0$, $e'_v := -1$

if $o^{-1}\left[[s_v, e_v[\right] \neq \emptyset,\; o^{-1}\left[[s_v, e_v[\right] \neq W,\; s_v > e_v$:

$s'_v := max(\{o(w) \mid w \in W, o(w) < s_v\}) + 1$
$e'_v := min(\{o(w) \mid w \in W, o(w) \geq e_v\})$

Show for all $v \in V$, $w \in W$ that $o(w) \in [s_v, e_v[\Leftrightarrow o'(w) \in [s'_v, e'_v[$:

Case $o^{-1}\left[\, [s_v, e_v[\,\right] = \emptyset$: clear

Case $o^{-1}\left[\, [s_v, e_v[\,\right] \neq \emptyset,\ s_v \leq e_v$:

"$\Leftarrow$"

clear because $[s'_v, e'_v[\subseteq [s_v, e_v[$ and $o = o'$

"$\Rightarrow$"

If $o(w) \in [s_v, e_v[$ then $o(w) \geq s_v$ and so according to the definition of s'_v, $o(w) \geq s'_v$.
Furthermore $o(w) < e_v$ and so according to the definition of e'_v, $o(w) < e'_v$.
Therefore $o(w) \in [s'_v, e'_v[$

Case $o^{-1}\left[\, [s_v, e_v[\,\right] = W,\ s_v > e_v$: clear

Case $o^{-1}\left[\, [s_v, e_v[\,\right] \neq \emptyset,\ o^{-1}\left[\, [s_v, e_v[\,\right] \neq W,\ s_v > e_v$:

"$\Rightarrow$"

clear because $[s_v, e_v[\subseteq [s'_v, e'_v[$ and $o = o'$ (remember Def 1 for intervals with $s_v > e_v$)

"$\Leftarrow$"

If $o(w) \notin [s_v, e_v[$ then $o(w) < s_v$ and so according to the definition of s'_v, $o(w) < s'_v$.
Furthermore $o(w) >= e_v$ and so according to the definition of e'_v, $o(w) \geq e'_v$.
Therefore $o(w) \notin [s'_v, e'_v[$

□

Definition 10 (Normalized Order)
The CIR-embedding (CCIR-embedding) o , s_v, e_v of an Relation $R \subseteq V \times W$ is called *order normalized* if

$$o[W] = \begin{cases} \{0, \ldots, |W| - 1\} & \text{if } |W| < \infty \\ \mathbb{N} & \text{if } |W| = \infty \end{cases}$$

Theorem 11 (Normalized Order)
Each CIR (CCIR) R has an *order normalized* CIR-embedding (CCIR-embedding).

Proof
Let o, s and e be a CIR-embedding (CCIR-embedding) of R.
W.l.g. o, s and e are interval normalized.
Define

$o'(w) := |\{p \mid p \in W, o(p) < o(w)\}|$

and

if $o^{-1}\left[\, [s_v, e_v[\,\right] = \emptyset$:

$s'_v := 0$, $e'_v := 0$

if $o^{-1}\left[\, [s_v, e_v[\,\right] \neq \emptyset,\ s_v \leq e_v$:

$s'_v := o'\left(o^{-1}(s_v)\right)$
$e'_v := o'\left(o^{-1}(e_v - 1)\right) + 1$

if $o^{-1}\left[\,[s_v, e_v[\,\right] = W,\ s_v > e_v$:

$s_v := 0,\ e_v := -1$

if $o^{-1}\left[\,[s_v, e_v[\,\right] \neq \emptyset,\ o^{-1}\left[\,[s_v, e_v[\,\right] \neq W,\ s_v > e_v$:

$s'_v := o'\left(o^{-1}\left(s_v - 1\right)\right) + 1$

$e'_v := o'\left(o^{-1}\left(e_v\right)\right)$

The above is well-defined as o, s and e are interval normalized.

Then $o'\left(w_1\right) \leq o'\left(w_2\right) \Leftrightarrow o\left(w_1\right) \leq o\left(w_2\right)$ for all $w_1, w_2 \in W$.

Show for all $v \in V$, $w \in W$ that $o\left(w\right) \in [s_v, e_v[\Leftrightarrow o'\left(w\right) \in [s'_v, e'_v[$:

Case $o^{-1}\left[\,[s_v, e_v[\,\right] = \emptyset$: clear

Case $o^{-1}\left[\,[s_v, e_v[\,\right] \neq \emptyset,\ s_v \leq e_v$:

$$
\begin{aligned}
o'^{-1}\left[\,[s'_v, e'_v[\,\right] &= \left\{w \in W \mid s'_v \leq o'\left(w\right) < e'_v\right\} \\
&= \left\{w \in W \mid o'\left(o^{-1}\left(s_v\right)\right) \leq o'\left(w\right) \leq o'\left(o^{-1}\left(e_v - 1\right)\right)\right\} \\
&= \left\{w \in W \mid o\left(o^{-1}\left(s_v\right)\right) \leq o\left(w\right) \leq o\left(o^{-1}\left(e_v - 1\right)\right)\right\} \\
&= \left\{w \in W \mid s_v \leq o\left(w\right) < e_v\right\} \\
&= o^{-1}\left[\,[s_v, e_v[\,\right]
\end{aligned}
$$

Case $o^{-1}\left[\,[s_v, e_v[\,\right] = W,\ s_v > e_v$: clear

Case $o^{-1}\left[\,[s_v, e_v[\,\right] \neq \emptyset,\ o^{-1}\left[\,[s_v, e_v[\,\right] \neq W,\ s_v > e_v$:

$$
\left(o'^{-1}\left[\,[s'_v, e'_v[\,\right]\right)^c = o'^{-1}\left[\,[e'_v, v'_v[\,\right] \overset{\text{see above}}{=} o^{-1}\left[\,[e_v, v_v[\,\right] = \left(o^{-1}\left[\,[s_v, e_v[\,\right]\right)^c
$$

$(S)^c$ denotes the complement of set S.

□

Corollary 12
There is an *interval and order normalized* CIR-embedding (CCIR-embedding) for each CIR (CCIR) R.

Proof
The CIR-embedding (CCIR-embedding) constructed in the proof of Thm. 11 is *interval normalized*.

□

Theorem 13 (Left-side Cloning)
Let $R \subseteq V \times W$ be a CIR (CCIR), $R' \subseteq V' \times W$ a relation with $V' = \bigcup_{v \in V} C_v$, where C_v is a set of clones of v, i.e. $(v', w) \in R' \Leftrightarrow (v, w) \in R$ holds for all $v' \in C_v$, $w \in W$.
Then R' is a CIR (CCIR).

Proof
Let o, s and e be a CIR-embedding (CCIR-embedding) of R.
Define

$o' := o$

$s'_{v'} := s_v$, $e'_{v'} := e_v$ for $v \in V$, $v' \in C_v$

Then for all $v \in V$ $v' \in C_v$, $w \in W$:

$(v', w) \in R' \quad \Leftrightarrow \quad (v, w) \in R \quad \Leftrightarrow \quad o(w) \in [s_v, e_v[\quad \Leftrightarrow \quad o'(w) \in [s'_{v'}, e'_{v'}[$

□

Theorem 14 (Right-side Finite Cloning)
Let $R \subseteq V \times W$ be a CIR (CCIR), $R' \subseteq V \times W'$ a relation with $W' = \bigcup_{w \in W} C_w$, where C_w is a *finite* set of clones of w, i.e. $(v, w') \in R' \Leftrightarrow (v, w) \in R$
holds for all $v \in V$, $w' \in C_w$
Then R' is a CIR (CCIR).

Proof
Let o, s and e be a CIR-embedding (CCIR-embedding) of R and $C_w = \left\{ {w'}_{w,0}, \ldots, {w'}_{w,|C_w|-1} \right\}$
Define

$$o'({w'}_{w,i}) := \left(\sum_{\substack{p \in W \\ o(p) < o(w)}} |C_p| \right) + i$$

and
if $o^{-1}\left[[s_v, e_v[\right] = \emptyset$:

$s'_v := 0$, $e'_v := 0$

if $o^{-1}\left[[s_v, e_v[\right] = W, s_v > e_v$:

$s_v := 0$, $e_v := -1$

if $o^{-1}\left[[s_v, e_v[\right] \neq \emptyset, o^{-1}\left[[s_v, e_v[\right] \neq W$:

$$s'_v := \left(\sum_{\substack{p \in W \\ o(p) < s_v}} |C_p| \right) \qquad e'_v := \left(\sum_{\substack{p \in W \\ o(p) < e_v}} |C_p| \right)$$

This is only defined when $|C_p| < \infty$ forall $p \in W$

Show $C_w \subseteq o^{-1}\left[[s'_v, e'_v[\right] \quad \Leftrightarrow \quad w \in o^{-1}\left[[s_v, e_v[\right]$

$$o(w) \geq s_v \quad \Leftrightarrow \quad s'_v = \left(\sum_{\substack{p \in W \\ o(p) < s_v}} |C_p| \right) \leq \left(\sum_{\substack{p \in W \\ o(p) < o(w)}} |C_p| \right) \leq o'\left({w'}_{w,i} \right)$$

$$o(w) < e_v \quad \Leftrightarrow \quad e'_v = \left(\sum_{\substack{p \in W \\ o(p) < e_v}} |C_p| \right) \geq \left(\sum_{\substack{p \in W \\ o(p) < o(w)}} |C_p| \right) + |C_w| > o'\left({w'}_{w,i} \right)$$

□

Definition 15 (Graph, Transitive Closure, Reflexive and Transitive Closure)

- A relation $G \subseteq V \times V$ is called a *Graph*.
- The *transitive closure* G^+ of a Graph $G \subseteq V \times V$ is the smallest transitive relation over $V \times V$ which contains G.
- The *reflexive and transitive closure* G^* of a Graph $G \subseteq V \times V$ is the smallest reflexive and transitive relation over $V \times V$ which contains G.

Definition 16 (CIG, CCIG)

- If G is a CIR it is called *Continuous Image Graph* (CIG).
- If G is a CCIR it is called *Circular Continuous Image Graph* (CCIG).

Definition 17 (Strongly Connected Components)
Let $G \subseteq V \times V$ be a graph and G^* its reflexive and transitive closure.
Then

$$v_1 \sim_s v_2 \quad :\Leftrightarrow \quad (v_1, v_2) \in G^*, (v_2, v_1) \in G^*$$

is an equivalence relation over V.
The equivalence classes of $\sim_s$ are called the *Strongly Connected Components* of G.

Lemma 18
Let $G \subseteq V \times V$ be a graph and $v_1, v_2 \in V$.
$[v]$ denotes the equivalence class of v with respect to $\sim_s$ for all $v \in V$.

1. If $[v_1] = [v_2]$ then $(v_1, v_2) \in G^*$.
2. If $[v_1] = [v_2]$ and $v_1 \neq v_2$ then $(v_1, v_2) \in G^+$.
3. If $[v_1] = [v_2]$ and $|[v_1]| > 1$ then $(v_1, v_2) \in G^+$.

Proof

1. $(v_2, v_2) \in G^*$ and $v_2 \sim_s v_1$.
2. $(v_1, v_2) \in G^*$ because of (1.) and $v_1 \neq v_2$.
3. If $v_1 \neq v_2$ we are finished because of (2.). Else there is $v'' \in [v_1]$ with $v'' \neq v_1 = v_2$. (2.) implies $(v_1, v') \in G^+$ and $(v', v_2) \in G^+$. So $(v_1, v_2) \in G^+$ as G^+ is transitive.

□

Definition 19 (Homomorphism)

- Let $R \subseteq V \times W$ and $R' \subseteq V' \times W'$ be relations.

  $$\begin{array}{lccc} h: & V \times W & \rightarrow & V' \times W' \\ h: & (v, w) & \mapsto & (h_1(v), h_2(w)) \end{array}$$

  is called a *homomorphism* from R to R' iff

  $$(v, w) \in R \quad \Leftrightarrow \quad (h_1(v)), (h_2(w)) \in R'$$

- If $G \subseteq V \times V$ and $G' \subseteq V' \times V'$ are graphs then
  $h : V \to V'$ is called *graph homomorphism* iff
  $$(v_1, v_2) \mapsto (h(v_1), h(v_2))$$
  is an homomorphism from G to G'.

Notation 20
Let $G \subseteq V \times V$, $G' \subseteq V' \times V'$ be graphs and h a graph homomorphism from G to G', then

- $v_1 \sim_{h_1} v_2 \quad :\Leftrightarrow \quad h_1(v_1) = h_1(v_2)$ is an equivalence relation on V.
- $w_1 \sim_{h_2} w_2 \quad :\Leftrightarrow \quad h_2(w_1) = h_2(w_2)$ is an equivalence relation on W.

Definition 21 (Second Finite Homomorphism)

- A homomorphism h from $R \subseteq V \times W$ to $R' \subseteq V' \times W'$ is called *second finite* if all equivalence classes $Y \in W/\sim_{h_2}$ are finite.
- A graph homomorphism h from $G \subseteq V \times V$ to $G' \subseteq V' \times V'$ is called *second finite* if $(v_1, v_2) \mapsto (h(v_1), h(v_2))$ is a second finite homomorphism.

Please note that the number of equivalence classes $Y \in W/\sim_{h_2}$ may be infinite.

Theorem 22
Let $R \subseteq V \times W$ and $R' \subseteq V' \times W'$ be relations and h a second finite homomorphism from R to R'. If R' is a CIR (CCIR) then R is a CIR (CCIR), too.

Proof

1. If R' is a CIR then $R'|_{h[V]\times h[W]}$ is a CIR because of Thm. 7 and
   h is a second finite surjective homomorphism from R to $R'|_{h[V]\times h[W]}$.
2. Define
   $$\tilde{V} := \bigcup_{X \in V/\sim_{h_1}} \{v_X\}$$
   where each v_X is some representative of the equivalence class X and
   $$\tilde{W} := \bigcup_{Y \in W/\sim_{h_2}} \{w_Y\}$$
   where each w_Y is some representative of the equivalence class Y.

   Then R is a CIR if $R|_{\tilde{V}\times\tilde{W}}$ is CIR because
   for all $v \in [\tilde{v}]$, $w \in [\tilde{w}]$, $\tilde{v} \in \tilde{V}$, $\tilde{w} \in \tilde{W}$
   $$(\tilde{v}, \tilde{w}) \in R \overset{Hom.}{\Leftrightarrow} (h_1(\tilde{v}), h_2(\tilde{w})) \in R' \overset{\substack{v\in[\tilde{v}]\\ w\in[\tilde{w}]}}{\Leftrightarrow} (h_1(v), h_2(w)) \in R' \overset{Hom.}{\Leftrightarrow} (v, w) \in R$$
   and $|[w]| < \infty$ for all $w \in W$ as h is second finite.
   So Thm. 13 and Thm. 14 are applicable.
3. $h|_{\tilde{V}\times\tilde{W}}$ is a bijective homomorphism, and therefore an isomorphism,
   from $R|_{\tilde{V}\times\tilde{W}}$ to $R'|_{h[V]\times h[W]}$. So $R|_{\tilde{V}\times\tilde{W}}$ is a CIR iff $R'|_{h[V]\times h[W]}$ is a CIR.

□

Corollary 23
Let $G \subseteq V \times V$ and $G' \subseteq V' \times V'$ be graphs and
h a second finite graph homomorphism from G to G'.
If G' is a CIG (CCIG) then G is a CIG (CCIG), too.

Definition 24
Let $G \subseteq V \times V$ be a graph and $\sim$ an equivalence relation on V.
$G/\sim := \{([v_1], [v_2]) \mid (v_1, v_2) \in G\}$

Theorem 25
Let $G \subseteq V \times V$ be a graph and $\sim_s$ the equivalence relation of Def. 17.

- $G^+/\sim_s$ is a CIG (or CCIG) iff G^+ is.
- $G^*/\sim_s$ is a CIG (or CCIG) iff G^* is.

Proof
"$\Rightarrow$"
Define
$h: \; V \;\to\; V/\sim_s$
$h: \; v \;\mapsto\; [v]$
h is a graph homomorphism from G to $G^+/\sim_s$:
$(v_1, v_2) \in G \;\Rightarrow\; (h(v_1), h(v)) \in G^+/\sim_s$ holds by definition of $G^+/\sim_s$.

Let $v_1, v_2 \in V$ and $([v_1], [v_2]) \in G^+/\sim_s$. Then $v_1' \in [v_1]$, $v_2' \in [v_2]$ exist with $(v_1', v_2') \in G^+$.
We show $(v_1, v_2) \in G^+$.
Case $[v_1] = [v_2]$ and $|[v_1]| = 1$:
Then $v_1' = v_1$ and $v_2 = v_2'$.
Case $[v_1] = [v_2]$ and $|[v_1]| > 1$:
See Lem. 18.
Case $[v_1] \neq [v_2]$:
$(v_1, v_2') \in G^*$ because $v_1 \sim_s v_1'$.
Therefore $(v_1, v_2') \in G^+$ as $v_1 \neq v_2'$ ($[v_1] \neq [v_2']$).
Furthermore $(v_1, v_2) \in G^*$ because $v_2 \sim_s v_2'$.
This implies $(v_1, v_2) \in G^+$ as $v_1 \neq v_2$ ($[v_1] \neq [v_2]$).

"$\Leftarrow$"
For each $X \in V/\sim_s$ let v_X be a representative of X. Define
$h': \; V \;\to\; V/\sim_s$
$h': \; v \;\mapsto\; [v]$
Then $h(h'(X)) = X$ for all $X \in V/\sim_s$.

Furthermore

$$(h'(X_1), h'(X_2)) \in G^+ \quad \Leftrightarrow \quad (h(h'(X_1)), h(h'(X_2))) \in G^+/\sim_s$$

for all $X_1, X_2 \in V/\sim_s$ as h is a graph homomorphism. Therefore

$$(h'(X_1), h'(X_2)) \in G^+ \quad \Leftrightarrow \quad (X_1, X_2) \in G^+/\sim_s$$

holds and h' is a graph homomorphism from $G^+/\sim_s$ to G^+.
From Cor. 23 follows that $G^+/\sim_s$ is a CIG if G^+ is a CIG.

□

5. Conclusion

In this diploma thesis we present an interface providing uniform and efficient access to graph data stored in different data structures. We show that an uniform interface eases objective comparisons between different storage schemes. Moreover the interface enables the implementation of efficient evaluation algorithms for queries on graphs which do not depend on the representation of the data. The interchangeability of the data structure used makes the algorithm reusable and allows a close adaption to the data. We have built a framework of abstract classes which leads to simple and comfortable implementations of concrete data structures without affecting efficiency.

We carefully implement the adjacency matrix, the adjacency array and the PPE data structure. Furthermore a first reasonable implementation of SIC is given. We newly introduce and implement MIC, an extension of SIC to arbitrary graphs, providing an interval representation of the graph which is at most 50% from optimal. In addition two storage schemes are defined which exploit the local structure of the graph. The first one, Localized Child, is designed to represent the adjacency relation and uses tree-like structures in the graph to decompose it into components which then can be treated independently. The second one, Localized Descendant, which is meant for the descendant relation, collapses strongly connected components to reduce the size of the graph and stores the descendant relation of the resulting DAG. Both data structures profit from the uniform interface as they may choose the most suitable representation for the components or the DAG respectively.

Finally we proof some interesting properties of CIGs. Particularly we show that homomorphisms preserve CIGs and that collapsing the strongly connected components of a graph is compatible with the CIR representation of its transitive closure or descendant relation.

Future Work.

- The PC-Tree data structure [**?**] is left unimplemented but is necessary for the automatic initialization of SIC.
- A implementation for "minimal cost perfect matching" is needed to improve the compression ratio of MIC. We recommend the integration of an existing freely available implementation like Blossom V [**?**]
- Improved heuristic for choosing which data structure is applied to some graph.
- Experimental evaluation of the implemented data structures.
  - Determine break-even-points (important for the previous point).
  - Examine how many graphs are CIGs or CCIGs. CCIGs extend CIGs by allowing circular intervals. CCIGs are a somehow nicer graph class as they are closed under complement. A corresponding data structure analogous to SIC, SCIC, is already implemented. The automatic initialization depends on the PC-Tree,too.
  - Explore how often the SIC representations of two or more relations of a graph can use the same ordering.

- Implementation of the evaluation algorithm for queries on graphs proposed in [**?**, **?**]. Probably the implementation will need a further (but very natural) compaction operation to be added to the interface for relations on compact integer domains.
- Comparison of the complete system (inclusive evaluation) to existing systems.
- Comparison of [**?**] to the Localized Descendant storage scheme with MIC.
- Improvement of existing and implementation of new data structures
  - Specific implementation of the inverse relation and the complement operation for SIC and MIC
  - Order sharing for different relation represented by SIC
  - Implementation of a data structure for *interval digraphs* [**?**]. Interval digraphs are a superclass of forests, CIGs, CCIGs and interval graphs. Considering [**?**, **?**, **?**] they seem to be the largest polynomial determinable graph class which has an order-based representations that can be stored in linear space and offers constant adjacency testing. The interval digraph encoding can be viewed as an extension of Pre-Post-Encoding. Both use two orderings of the vertices of the graph and use the positions of the vertices in these orderings to determine adjacency. The digraph encoding employs two further integer values for each vertex. However on trees these values collapse with the positions in the two orderings. So on a tree pre- and post-order together with the pre- and post- position values for each vertex also form a digraph encoding for the descendant relation of the tree.

A. Classes Contained in the Implementation

```
.
|-- IntRelations/
|   |-- BinaryIntRelations/
|   |   |-- PrePostEncoding/
|   |   |   |-- ABinaryPrePostRelation.cs
|   |   |   |-- PRE_POST_ENCODING.cs
|   |   |   |-- PrePostAnc.cs
|   |   |   |-- PrePostChild.cs
|   |   |   |-- PrePostDesc.cs
|   |   |   |-- PrePostFollow.cs
|   |   |   |-- PrePostFollowSib.cs
|   |   |   |-- PrePostIntRelationSet.cs
|   |   |   |-- PrePostParent.cs
|   |   |   |-- PrePostPrec.cs
|   |   |   `-- PrePostPrecSib.cs
|   |   |-- AdjacencyMatrix.cs
|   |   |-- And.cs
|   |   |-- Empty.cs
|   |   |-- LocalizedChild.cs
|   |   |-- LocalizedDesc.cs
|   |   |-- Mic.cs
|   |   |-- Or.cs
|   |   |-- OrderedAdjacencyArray.cs
|   |   |-- OrderedAdjacencySingleArray.cs
|   |   |-- OrderedAdjacencySingleArrayTreeChild.cs
|   |   |-- Scic.cs
|   |   |-- Sic.cs
|   |   |-- SimpleTreeChild.cs
|   |   |-- SimpleTreeParent.cs
|   |   |-- UnorderedAddableAdjacencyArray.cs
|   |   |-- UnorderedAdjacencyArray.cs
|   |   `-- UnorderedAdjacencySingleArray.cs
|   |-- IntRelationSets/
|   |   |-- AGraphIntRelationSet.cs
|   |   |-- AIntRelationSet.cs
|   |   |-- AdaptiveGraphIntRelationSet.cs
|   |   |-- CombinedIntRelationSet.cs
|   |   |-- LocalizedChildIntRelationSet.cs
|   |   |-- LocalizedDescIntRelationSet.cs
|   |   |-- MicIntRelationSet.cs
|   |   |-- ScicIntRelationSet.cs
|   |   |-- SicIntRelationSet.cs
|   |   `-- VertexLabel1IntRelationSet.cs
```

```
|   |-- UnaryIntRelations/
|   |   `-- VertexLabel1_Unary.cs
|   |-- ABinaryIntRelation.cs
|   |-- ABinaryLabelRelation.cs
|   |-- AUnaryIntRelation.cs
|   |-- BinaryIntDomain.cs
|   |-- CBM.cs
|   |-- CCBM.cs
|   |-- CCIR.cs
|   |-- CIR.cs
|   |-- CutIntRelation1.cs
|   |-- IBinaryIntRelation.cs
|   |-- IUnaryIntRelation.cs
|   |-- StdBinaryComplementIntRelation.cs
|   |-- StdInverseIntRelation.cs
|   |-- StdUnaryComplementIntRelation.cs
|   |-- TS.cs
|   |-- UnaryEmptyIntRelation.cs
|   `-- UnaryIntDomain.cs
|-- ABinaryDomain.cs
|-- ABinaryRelation.cs
|-- ARelation.cs
|-- AUnaryDomain.cs
|-- AUnaryRelation.cs
|-- ArrayExtensions.cs
|-- BinaryEmptyRelation.cs
|-- EULER.cs
|-- EmptyEnumerator.cs
|-- EmptyRelation.cs
|-- GRAPH.cs
|-- IBinaryRelation.cs
|-- IPair.cs
|-- IRelation.cs
|-- IRelationSet.cs
|-- IUnaryRelation.cs
|-- IntArrayBitExtensions.cs
|-- MATCHING.cs
|-- Pair.cs
|-- SArrayStack.cs
|-- SBitArray.cs
|-- SBitMatrix.cs
|-- SEARCH.cs
|-- SIntInterval.cs
|-- SIntMatrix.cs
|-- SPANNING_TREE.cs
|-- SPair.cs
```

```
|-- STRONG_COMPONENTS.cs
|-- SUncheckedBitArray.cs
|-- SUncheckedIntArray.cs
|-- StdBinaryComplementRelation.cs
|-- StdBinaryCrossProduktDomain.cs
|-- StdBinaryInverseRelation.cs
|-- StdComplementRelation.cs
|-- StdCutRelation1.cs
|-- StdUnaryComplementRelation.cs
`-- UnaryEmptyRelation.cs
```