Bachelor Thesis

Abstract Rewrite Tool

Sebastian Stabinger
sebastian.stabinger@student.uibk.ac.at

4 April 2011

Supervisor: Sarah Winkler, MSc.

Abstract

Abstract rewrite systems allow to capture properties common to different variants of rewrite systems. In this project a tool was developed which allows the creation, manipulation and analysis of such systems. All necessary algorithms to achieve this are introduced and analyzed. The tool is also equipped with a game-like mode intended for students to test their knowledge about properties of abstract rewrite systems. In the course of implementing the Abstract Rewrite Tool it became clear that a method for automatic graph drawing was needed. Thus a graph drawing algorithm was implemented which is also presented in this thesis.

Contents

1 Introduction

Abstract rewrite systems abstract, as the name indicates, some properties which are common to different kinds of rewrite systems like term rewrite systems or string rewrite systems. Certain properties and relationships can be analyzed in abstract rewrite systems and then transferred to other rewrite systems. This is for example done in the introductory course on term rewrite systems at the University of Innsbruck.

To understand abstract rewrite systems it is usually necessary to practice the identification of properties by hand on a sheet of paper. This approach has some drawbacks. For a student who did not grasp some concept about abstract rewrite systems it can be hard to understand why a given answer was wrong even if the correct solution is presented.

On the other hand the preparation of examples for the students to solve can be bothersome because graphs have to be drawn and the correct solution has to be determined which, even if it is not difficult for a skilled person, takes time and is prone to oversight.

To aid this process we have developed the Abstract Rewrite Tool. It allows the creation and manipulation of abstract rewrite systems in an easy and intuitive way. All properties of the resulting system are automatically analyzed and it can either be embedded into a LaTeX document or saved to a file and passed on to the student.

In an additional mode of the tool the student can test her understanding of abstract rewrite systems by identifying properties as she would on paper. In contrast to working on paper the tool will, in case of a wrong answer, give additional information why the input is wrong. Additionally to abstract rewrite systems provided by a tutor it is also possible to practice on randomly generated graphs.

Since there were two main problem areas to tackle while developing the tool this thesis consists of two main parts. The first will introduce abstract rewrite systems and their properties in Chapter 2 and present algorithms that can be used to determine them in Chapter 3. The second part will look at the problem of automatically drawing a graph in a visually pleasing manner and present a solution to this problem with *Graph Drawing by Force-directed Placement* [2] in Chapter 4. In Chapter 5 the Abstract Rewrite Tool will be presented and its usage explained. At the end of this thesis in Chapter 6 we will give a conclusion and some ideas about possible future work in this area.

2 Abstract Rewrite Systems

Since it is necessary to understand abstract rewrite systems (ARSs) for this thesis, a short description of them will now be given. Big parts of this chapter are based on [4] and [1].

2.1 Motivation for ARSs

In mathematics and computer science you can find many so called *rewrite systems* or *reduction systems*. Two examples would be term rewrite systems and string rewrite systems. In both examples part of an object (term or string) can be replaced by another object following certain rules.

We can therefore extract the quintessential parts of a rewrite system (objects are rewritten to other objects) and define abstract rewrite systems in which it is not specified what is rewritten but only that rewriting can take place. This alone is sufficient to define and examine important properties and characteristics common to all rewrite systems.

2.2 Basic Definitions

Definition 2.1. An ARS $\mathcal{A}$ consists of a set of objects A and a binary relation $\rightarrow$ between them. A pair $(a, b) \in \rightarrow$ can also be written as $a \rightarrow b$ and is called a *rewrite step*.

Definition 2.2. A concatenation of zero or more rewrite steps is called a *rewrite sequence*. A rewrite sequence starting at $a \in A$ and ending in $b \in A$ is denoted by $a \rightarrow^* b$ and we can also say "a rewrites to b", "a reduces to b" or "b is a successor of a". A rewrite sequence with one or more rewrite steps is denoted by $a \rightarrow^+ b$.

It is easy to see from Definition 2.1 that an ARS can be interpreted as a directed graph.

Example 2.3. The corresponding graph for an ARS with $A = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and $\rightarrow = \{(\mathsf{a}, \mathsf{b}), (\mathsf{a}, \mathsf{c}), (\mathsf{b}, \mathsf{d}), (\mathsf{c}, \mathsf{d})\}$ can be seen in Figure 2.1.

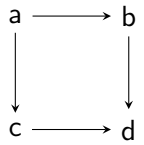


Figure 2.1: Example for a directed graph representation of an ARS.

An ARS can by definition be infinite but in this thesis only finite ARSs will be considered.

Joinability

Definition 2.4. Two elements $a, b \in A$ are *joinable* in $\mathcal{A}$, if both can be reduced to common successor $c \in A$. This relation is denoted by $a \downarrow b$.

Translated to a graph this means that two nodes are joinable if there exists a node that is reachable from both nodes.

Example 2.5. In Figure 2.2 elements a and b are joinable because both can be rewritten to d but b and e are not joinable because they have no common successor.

It should be noted that the common element can be one of the two elements that are being joined. So in Figure 2.2 elements b and d are also joinable because b can be rewritten to d and d does not have to be rewritten.

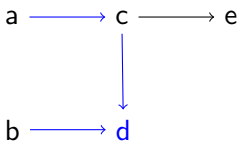


Figure 2.2: Example for joinability.

Normal Form

Definition 2.6. An element $a \in A$ is a *normal form* (NF) of $\mathcal{A}$ if there exists no element $b \in A$ so that a can be rewritten to b in one or more steps.

Given two elements $a, b \in A$ where b is a normal form in $\mathcal{A}$ the fact that a can be rewritten to the normal form b is denoted by $a \to^! b$.

Translated to a graph, this means that a node n is a normal form if and only if there are no nonempty paths beginning at n. In graph theory such a node is usually called a *sink*.

Example 2.7. In Figure 2.3 element b is a normal form, a and c are not. Element c is not in normal form because a and b from Definition 2.6 can of course be the same element. So the path c → c prevents c from being a normal form.

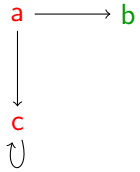


Figure 2.3: Example for a normal form.

Informally this means that a rewrite process terminates once a normal form is reached.

2.3 Properties

Every node of an ARS has a certain set of properties. If a property holds for every node of an ARS the property holds for the ARS itself.

Strong Normalization

Definition 2.8. An element $e \in A$ is *strongly normalizing* (SN) in $\mathcal{A}$ if there exists no infinite rewrite sequence starting from e.

Translated to a graph this means a node is strongly normalizing if there are no cycles reachable from that node.

Example 2.9. In Figure 2.4 element e is not strongly normalizing because it can reach the cycle a → b → c → a. Because a, b and c are part of the cycle themselves those elements are not strongly normalizing either.

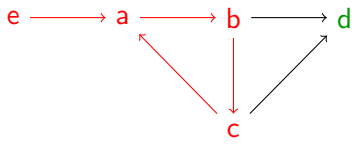


Figure 2.4: Example for an ARS which is not strongly normalizing.

Informally this means that starting from a strongly normalizing element, no matter which rewrite steps are chosen in a rewrite process, it will always terminate at some point. On the other hand it means that if a rewrite sequence starts from an element which is not strongly normalizing there is always at least one way to select a next element so the rewriting process will never terminate.

Weak Normalization

Definition 2.10. An element $a \in A$ is *weakly normalizing* (WN) in $\mathcal{A}$ if there exists at least one rewrite sequence $a \to^! b$ leading to a normal form $b \in A$.

Translated to a graph, this means that if a node n is weakly normalizing there is at least one path leading to a sink.

Example 2.11. In Figure 2.5 elements a and d are weakly normalizing because from both of them a normal form can be reached (in case of element d it is an empty path). Elements b and c are not weakly normalizing since no normal form can be reached from those.

Informally this means that for a rewriting process starting at an element which is weakly normalizing there is at least one way to select the rewrite steps so the process will terminate. It also means, if the rewriting process starts at an element which is not weakly normalizing, no matter in which way the rewrite steps are chosen, the process will never terminate.

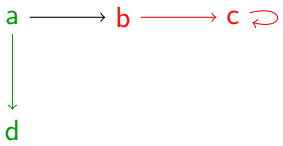


Figure 2.5: Example for weak normalization.

Unique Normal Forms

Definition 2.12. An element $a \in A$ has *unique normal forms* (UN) in $\mathcal{A}$ if the rewrite sequences $a \to^! b$ and $a \to^! c$ imply $b = c$.

Translated to a graph, this means that a node has unique normal forms if there is at most one sink reachable from that node.

Example 2.13. In Figure 2.6 element a does not have unique normal forms since it can be rewritten to the normal forms d and c. Elements b, c and d on the other hand have unique normal forms since they all can only be rewritten to one normal form.

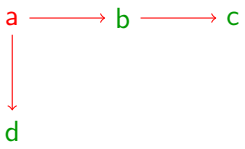


Figure 2.6: Example for unique normal forms.

Informally this means that starting from an element which has unique normal forms the process will always terminate in the same normal form if it terminates at all. On the other hand, a rewrite process starting from an element which has no unique normal forms can terminate in at least two different normal forms.

Church-Rosser Property

An element that has the Church-Rosser property (CR) is also called *confluent*.

Definition 2.14. An element $a \in A$ is *confluent* in $\mathcal{A}$ if for all $b, c \in A$ with $a \to^* b$ and $a \to^* c$ element b can be joined with element c.

Translated to a graph, this means that all reachable nodes have to be pairwise joinable to be confluent.

Example 2.15. In Figure 2.7 element a is confluent. The set of all reachable elements from a is $\{b, c, d, e\}$. So every element from this set has to be joinable with every other element. Elements b and c can be joined via the common reducts $\{c, d, e\}$, b and d can be joined via the common reducts $\{d, e\}$, b and e can be joined via the common reduct e, c and d can be joined via the common reducts $\{d, e\}$, c and e can be joined via the common reduct e and d and e can be joined via the common reduct e. In fact in Figure 2.7 all elements are confluent.

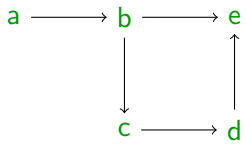


Figure 2.7: Example for Church-Rosser property.

This means in essence that no matter which two rewrite sequences are taken from an element which is confluent, there is always the possibility to reach the same element and if the rewrite process terminates, it will always terminate in the same normal form.

Weak Church-Rosser Property

An element that has the weak Church-Rosser property (WCR) is also called *locally confluent*.

Definition 2.16. An element $a \in A$ is *locally confluent* in $\mathcal{A}$ if for all $b, c \in A$ with $a \to b$ and $a \to c$ element b can be joined with element c.

Translated to a graph, this means that a node is locally confluent, if all pairs of direct successors[1] can be joined.

Example 2.17. In Figure 2.8 node a is locally confluent. The set of direct successors is $\{\mathsf{b}, \mathsf{d}\}$ and b and d can be joined via the common successor d. In fact all elements in Figure 2.8 except c are locally confluent.

This is also a good example of elements which are locally confluent but not confluent. For element a to be confluent d and e would also have to be joinable, which they are not.

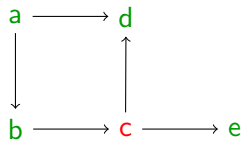
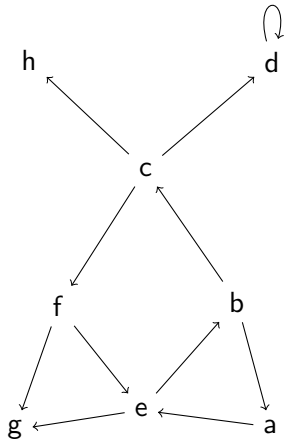


Figure 2.8: Example for weak Church-Rosser property.

[1] All nodes reachable with a path of length one.

Example 2.18. To bring everything together we will now present a graph and all properties of all its elements, see Figure 2.9.



	SN	WN	CR	WCR	UN
a	×	✓	×	✓	×
b	×	✓	×	✓	×
c	×	✓	×	×	×
d	×	×	✓	✓	✓
e	×	✓	×	✓	×
f	×	✓	×	✓	×
g	✓	✓	✓	✓	✓
h	✓	✓	✓	✓	✓

Figure 2.9: An abstract rewrite system and all its properties.

2.4 Relationships Between Properties

Properties and combinations of properties can have certain relationships to one another. We will now investigate some important examples of such relationships for an element $a \in A$.

Lemma 2.19. *SN(a)* $\rightarrow$ *WN(a)*

Proof. Since SN(a) implies that every rewrite sequence starting from a has to terminate in a finite number of steps there has to be at least one normal form reachable from this node which implies that it is also weakly normalizing. □

This relationship does not hold in the other direction though because of the fact that there exists a rewrite sequence which ends in a normal form does not imply that there cannot be other infinite sequences. Element a in Figure 2.9 would be an example for this.

Lemma 2.20. *CR(a)* $\rightarrow$ *UN(a)*

Proof. Since different normal forms are never joinable among each other an element which is confluent can only have one normal form at most. An element which has zero or one normal form has unique normal forms. □

Lemma 2.21. *CR(a)* $\rightarrow$ *WCR(a)*

Proof. Since a is confluent, for all b and c with $b \,{}^*\!\leftarrow a \rightarrow^* c$ we have $b \downarrow c$. This $b \downarrow c$ also holds for all $b \leftarrow a \rightarrow c$ which implies that a is locally confluent. □

Lemma 2.22. *SN(a)* $\wedge$ *UN(a)* $\rightarrow$ *CR(a)*

Proof. Since a is strongly normalizing every element b with $a \to^* b$ can be rewritten to a normal form in a finite number of steps. Since a has unique normal forms $b \to^! c$ and $b \to^! d$ implies $c = d$. Thus for all peaks $e \,{}^*\!\leftarrow a \to^* f$ the elements e and f have unique normal forms e' and f'. Since these are also normal forms of a, $e' = f'$ and thus $e \downarrow f$ needs to hold, which implies that a is confluent.

□

Lemma 2.23. $\neg UN(a) \to WN(a)$

Proof. If an element does not have unique normal forms it has to have at least two different normal forms. Every element which has at least one normal form is weakly normalizing. □

3 Implementation of Abstract Rewrite Systems

This chapter will describe how abstract rewrite systems are represented in the Abstract Rewrite Tool (ART) and how the properties are determined.

3.1 Representation of the ARS

Abstract rewrite systems in the ART are represented as a graph which is implemented using adjacency lists. This means that there exists a list with all nodes that are part of the graph and every node itself holds an adjacency list containing references to all of its direct successors.

File Format

For loading and saving abstract rewrite systems a file format had to be specified. Listing 3.1 shows the specification in extended Backus-Naur form.

```
ars ::= node { \n node }
node ::= ( id ( -> | : ) nodelist ) | id
nodelist ::= id { , id }
digit ::= 0 | 1 | 2 | 3 | ... | 9
alphabetic ::= A | B | ... | Z | a | b | ... | z
alphanum ::= digit | alphabetic
id ::= alphanum [ alphanum ]
```

Listing 3.1: File format description in extended Backus-Naur form.

For an example, see Listing 3.2.

```
1->5
5->2,7
2->3,1
3->6,4
6->5,7
```

Listing 3.2: Content of a file that describes the graph in Figure 3.2.

As can be seen in Listing 3.1 a colon can be used instead of `->`. This is mainly intended to speed up the creation of edges in the parse prompt. See Section 5.3.1 for further information. It can also be seen that nodes can only have names with two symbols at maximum. This is no general limitation of

the ART but names bigger than two symbols will not fit into the circles that represent the nodes (see Figure 3.1) so it was decided to limit the size.

Figure 3.1: Example of a node with a name of length three.

3.2 Algorithms to Decide Properties

3.2.1 Auxiliary Algorithms

Transitive Closure

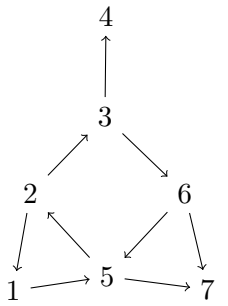
Definition 3.1. For a node n the *transitive closure* contains all nodes that are reachable from n with a path of a length of at least one. The transitive closure of a node n will be denoted by $tc(n)$.

Definition 3.2. For a node n the *reflexive transitive closure* contains all nodes that are reachable from n with a path of a length of at least zero. It is therefore the transitive closure of n extended with the node n itself. The reflexive transitive closure of a node n will be denoted by $rtc(n)$.

To determine the transitive closure Warshall's algorithm [8] is being used.

Definition 3.3. Given a graph G with n nodes, a matrix A of size $n \times n$ is called an *adjacency matrix* of G if $A_{i,j} = 1$ if there exists an edge from node i to j and $A_{i,j} = 0$ if no such edge exists.

Example 3.4. Figure 3.2 gives an example how a graph and its adjacency matrix correlate.



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.2: Correlation between a graph and its adjacency matrix.

The algorithm starts with the adjacency matrix M of size $n \times n$ and creates a working copy T which will contain the result at the end.

It then iterates over every element in the matrix until a 1 is found. If the position is i, j this means that there is an edge from node i to node j but this

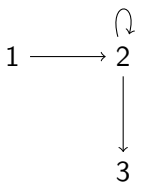
also means that every node which can be reached from j can also be reached from node i via j. To express this, the nodes reachable from j have to be marked reachable for i. This is done by iterating over every element of the j^{th} row in the matrix. If an entry with value 1 can be found at position k (which means that k is reachable from j) the element on position i, k is also set to 1 which represents the notion that k is reachable from i. Because it is equivalent and more efficient than using conditional branching the formula $T_{i,k} = T_{i,k} \vee T_{j,k}$ is used for every $k \in \{1, \ldots, n\}$ where 1 is interpreted as true and 0 as false.

1. for j in 1 to n
2. for i in 1 to n
3. if($T_{i,j} == 1$)
4. for k in 1 to n
5. $T_{i,k} = T_{i,k} \vee T_{j,k}$

Listing 3.3: Warshall's algorithm.

The time complexity of Warshall's algorithm for n nodes is $O(n^3)$ because line 5 is executed n^3 times at most. Since no additional memory is needed the memory complexity is $\Theta(n^2)$ for the matrix itself.

Example 3.5. The algorithm is performed on the following graph:



$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

- $T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$
- $i = 1$ and $j = 1$; $T_{1,1} = 0 \Rightarrow$ do nothing
- $i = 2$ and $j = 1$; $T_{2,1} = 0 \Rightarrow$ do nothing
- $i = 3$ and $j = 1$; $T_{3,1} = 0 \Rightarrow$ do nothing
- $i = 1$ and $j = 2$; $T_{1,2} = 1 \Rightarrow$
  $T_{1,1} \vee T_{2,1} = 0 \vee 0 = 0 \Rightarrow T_{1,1} = 0$
  $T_{1,2} \vee T_{2,2} = 1 \vee 1 = 1 \Rightarrow T_{1,2} = 1$
  $T_{1,3} \vee T_{2,3} = 0 \vee 1 = 1 \Rightarrow T_{1,3} = 1$
  $T = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$
- $i = 2$ and $j = 2$; $T_{2,2} = 1 \Rightarrow$ since i and j are 2, T will not change because $T_{i,k} \vee T_{j,k} = T_{i,k}$ if $i == j$

- $i = 3$ and $j = 2$; $T_{3,1} = 0 \Rightarrow$ do nothing

- $i = 1$ and $j = 3$; $T_{1,3} = 1 \Rightarrow$
  $T_{1,1} \vee T_{3,1} = 0 \vee 0 = 0 \Rightarrow T_{1,1} = 0$
  $T_{1,2} \vee T_{3,2} = 1 \vee 0 = 1 \Rightarrow T_{1,2} = 1$
  $T_{1,3} \vee T_{3,3} = 1 \vee 0 = 1 \Rightarrow T_{1,3} = 1$
  $T = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ (nothing has changed)

- $i = 2$ and $j = 3$; $T_{2,3} = 1 \Rightarrow$
  $T_{2,1} \vee T_{3,1} = 0 \vee 0 = 0 \Rightarrow T_{2,1} = 0$
  $T_{2,2} \vee T_{3,2} = 1 \vee 0 = 1 \Rightarrow T_{2,2} = 1$
  $T_{2,3} \vee T_{3,3} = 1 \vee 0 = 1 \Rightarrow T_{2,3} = 1$
  $T = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ (nothing has changed)

- $i = 3$ and $j = 3$; $T_{3,3} = 0 \Rightarrow$ do nothing

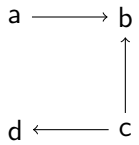
So the only thing that has changed is $T_{1,3} = 1$ which means that node 3 is reachable from node 1 which is obviously correct.

Joinability

Joinability can be directly implemented as it is defined. Two nodes a and b are joinable, if $rtc(a) \cap rtc(b) \neq \varnothing$.

If the transitive closure is already calculated, checking joinability is a fairly efficient approach. If data structures are used that allow a lookup in constant time (for example a hash map) this algorithm has a complexity of $O(n)$ for one pair of nodes and $\binom{n}{2}O(n) = O(n^3)$ for all combinations of pairs.

Example 3.6. Given the following graph, we check the joinability of all nodes.



$$
\begin{aligned}
\mathrm{rtc}(\mathsf{a}) &= \{\mathsf{a},\mathsf{b}\} \\
\mathrm{rtc}(\mathsf{b}) &= \{\mathsf{b}\} \\
\mathrm{rtc}(\mathsf{c}) &= \{\mathsf{c},\mathsf{b},\mathsf{d}\} \\
\mathrm{rtc}(\mathsf{d}) &= \{\mathsf{d}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{a} \downarrow \mathsf{b}: &\quad \{\mathsf{a},\mathsf{b}\} \cap \{\mathsf{b}\} = \{\mathsf{b}\} && \Rightarrow \mathsf{a} \text{ and } \mathsf{b} \text{ are joinable} \\
\mathsf{a} \downarrow \mathsf{c}: &\quad \{\mathsf{a},\mathsf{b}\} \cap \{\mathsf{c},\mathsf{b},\mathsf{d}\} = \{\mathsf{b}\} && \Rightarrow \mathsf{a} \text{ and } \mathsf{c} \text{ are joinable} \\
\mathsf{a} \downarrow \mathsf{d}: &\quad \{\mathsf{a},\mathsf{b}\} \cap \{\mathsf{d}\} = \varnothing && \Rightarrow \mathsf{a} \text{ and } \mathsf{d} \text{ are not joinable} \\
\mathsf{b} \downarrow \mathsf{c}: &\quad \{\mathsf{b}\} \cap \{\mathsf{c},\mathsf{b},\mathsf{d}\} = \{\mathsf{b}\} && \Rightarrow \mathsf{b} \text{ and } \mathsf{c} \text{ are joinable} \\
\mathsf{b} \downarrow \mathsf{d}: &\quad \{\mathsf{b}\} \cap \{\mathsf{d}\} = \varnothing && \Rightarrow \mathsf{b} \text{ and } \mathsf{d} \text{ are not joinable} \\
\mathsf{c} \downarrow \mathsf{d}: &\quad \{\mathsf{c},\mathsf{b},\mathsf{d}\} \cap \{\mathsf{d}\} = \{\mathsf{d}\} && \Rightarrow \mathsf{c} \text{ and } \mathsf{d} \text{ are joinable}
\end{aligned}
$$

3.2.2 Strong Normalization

The question whether a node is strongly normalizing or not can be decided as follows: A node is strongly normalizing if and only if all its successors are strongly normalizing and it is not part of a cycle. This idea can be formalized as follows. Here $\mathrm{succ}(n)$ contains all direct successors of an element $n \in A$.

Definition 3.7.

$$
\mathrm{SN}(n, \mathit{visited}) = \begin{cases} \mathit{false} & \text{if} \quad n \in \mathit{visited} \\ \mathit{true} & \text{if} \quad \mathrm{succ}(n) = \varnothing \\ \mathit{true} & \text{if} \quad \forall s \in \mathrm{succ}(n) : \mathrm{SN}(s, \mathit{visited} \cup \{n\}) = \mathit{true} \\ \mathit{false} & \text{if} \quad \exists s \in \mathrm{succ}(n) : \mathrm{SN}(s, \mathit{visited} \cup \{n\}) = \mathit{false} \end{cases}
$$

Lemma 3.8. *An element $e \in A$ is strongly normalizing in $\mathcal{A}$ iff $\mathrm{SN}(e, \varnothing) = \mathit{true}$.*

Again dynamic programming can be used to develop an efficient algorithm. Instead of actually following all recursions, we use memoization to cache results of function-calls. That guarantees that every node is only processed once. An implementation of the algorithm in Scala can be seen in Listing 3.4. The memoization is implemented by saving the results of every function call in a hash map.

Since we have a depth first search without repetition we get a complexity of $O(|V| + |E|)$ where $|V|$ is the number of nodes and $|E|$ is the number of edges. With n nodes in the worst case we could end up with n^2 edges and the complexity would become $O(n^2)$. The graphs used in the Abstract Rewrite Tool however have usually between n and $2n$ edges which would result in a complexity of $O(n)$.

3.2.3 Weak Normalization and Unique Normal Forms

These two properties can be determined together. An implementation in pseudo code can be seen in Listing 3.5.

```
def getSN() = {
  val sn:Map[Node,Boolean] = Map.empty

  def visit(n:Node,visited:Set[Node]) {
    if(!(sn contains n)) {
      if(visited contains n) sn += (n -> false)
      else {
        for(i <- n.connections) visit(i,visited + n)
        sn += (n -> (n.connections.forall(sn(_) == true)))
      }
    }
  }

  for(i <- nodes if(!(sn contains i))) visit(i,Set.empty)
  sn
}
```

Listing 3.4: Algorithm to calculate strong normalization for all nodes of a graph implemented in Scala.

Since for a graph with n nodes the transitive closure of one node can at most contain n nodes the complexity of the algorithm will be $O(n)$. This of course does not include the calculation of the transitive closure itself.

3.2.4 Church-Rosser Property

The Church-Rosser property for a node n can be determined by checking joinability for all pairs of nodes from the transitive closure of n. The node n has the Church-Rosser property if and only if all pairs are joinable.

In the worst case starting from a node $a \in A$ where A has n nodes, $tc(a)$ will also contain n nodes. To determine whether a is confluent or not the joinability of $\binom{n}{2}$ pairs has to be checked. Testing the joinability of a pair of nodes has a complexity of $O(n)$ so the complexity to test for confluence will be $\binom{n}{2}O(n) = O(n^3)$. So checking the whole system for the Church-Rosser property could have complexity $O(n^4)$. If the results for joinability are memoized this can be brought down to a complexity of $O(n^3)$.

3.2.5 Weak Church-Rosser Property

The weak Church-Rosser property of a node $a \in A$ can be determined by checking the set of successors of a. If all pairs of nodes from that set are joinable, a is locally confluent. Since in the worst case the set of successors could be all nodes of the graph the same arguments as for the Church-Rosser property can be made. We then end up with a complexity of $O(n^4)$ respective $O(n^3)$ with memoization of joinability.

1. $count = 0$
2. for all elements $e \in tc(a)$
3. if e is a normal form
4. $count = count + 1$
5. if $count < 2$
6. $UN(a) = \text{true}$
7. else
8. $UN(a) = \text{false}$
9. if $count > 0$
10. $WN(a) = \text{true}$
11. else
12. $WN(a) = \text{false}$

Listing 3.5: Algorithm to determine WN(a) and UN(a).

3.3 Overall Complexity

We will now look at the complexity of calculating all properties of a graph with n nodes with utilization of memoization. See Table 3.1 for a summary of the complexities of all the used algorithms.

transitive closures	$O(n^3)$
joinability	$O(n^3)$
weak normalization and unique normal forms	$O(n^2)$
strong normalization	$O(n^2)$
Church-Rosser property	$O(n^3)$
weak Church-Rosser property	$O(n^3)$

Table 3.1: Complexities for the algorithms in the ART.

This leads to a total complexity of $O(n^3) + O(n^3) + O(n^2) + O(n^2) + O(n^3) + O(n^3) = O(n^3)$.

3.4 Optimizations

3.4.1 Memoization

Memoization is a method which allows an algorithm to get higher performance by using more memory. This is achieved by remembering the return value of a function in relation to a set of input parameters. This of course only works as long as the internal state of an object stays the same but then it ensures that every computation only has to be done once.

In the ART every function of the objects that represent the graph and the nodes uses memoization. For example if the joinability of a pair of nodes should be checked the responsible function will first look up if a value for this pair was already calculated. If it was this value will be directly returned, if no precalculated value exists it will be calculated, remembered and returned. The next time the joinability of the same two nodes is requested the calculation can be omitted.

Implementation

In the ART memoization is strictly separated from the classes representing graphs and nodes. So the classes `DirectedGraph` and `Node` have an equivalent `MemDirectedGraph` and `MemNode` whose only purpose is to support memoization for every function of the two classes. Functions which do not need parameters are memoized by a simple variable. Functions with parameters are memoized using a hash map with the parameter values as keys. An example implementation of a memoization of the joinability function in Scala looks as follows:

```
var memJoins:HashMap[Node,Boolean] = HashMap.empty
override def joins(node: Node) = memJoins.getOrElseUpdate(node,
    super.joins(node))
```

This example also shows how the functional capabilities of Scala can lead to very natural and concise code. The first line creates a hash map that maps from a node to a boolean value. The second line overwrites the original function `joins` in the class `Node`. The method `getOrElseUpdate` is already provided by `HashMap` and will check if the parameter `node` is already present in the hash map. If it is, the saved value will be returned, if it is not, the second parameter will be returned and at the same time saved in the hash map. Since the second parameter is call-by-name it will only be evaluated if no precalculated result can be found.

Advantages

There are different advantages of the general memoization of all functions. On one hand many algorithms can be implemented directly in a recursive fashion without performance impairments. For example the algorithm to determine strong normalization can omit the dynamic programming approach because the general memoization provides this automatically.

Additionally the code becomes more concise and easier to understand. For example in the ART nodes that are normal forms will be drawn in a different color. Since the information if a node is a normal form or not is needed on every redraw this amounts to approximately sixty requests a second per node. This of course is not practical and it would be necessary to create data structures in the graphical part of the system to hold this information which also would have to be synchronized in certain intervals to the actual values. With memoization the information can be directly acquired from the core classes in a very transparent way without impairing performance.

Internal State

Memoization is of course only possible as long as the internal state of objects does not change. For example if nodes and edges are removed and added in a graph the memoized values may not be correct anymore. To alleviate this problem all memoized values are discarded if the graph is manipulated.

Of course it would be possible to only discard certain memoized values regarding to what has changed in the graph but since the additional performance gain is assumed to be relatively minuscule this possibility was never investigated further.

Performance

To test the performance gained by memoization some benchmarks[1] were taken. The time needed to determine all properties of ten graphs with differing number of nodes was measured. To obtain accurate measurements while using randomly generated graphs actually 100,000 graphs were processed and the results calculated back for ten graphs. The results can be seen in Figure 3.3.

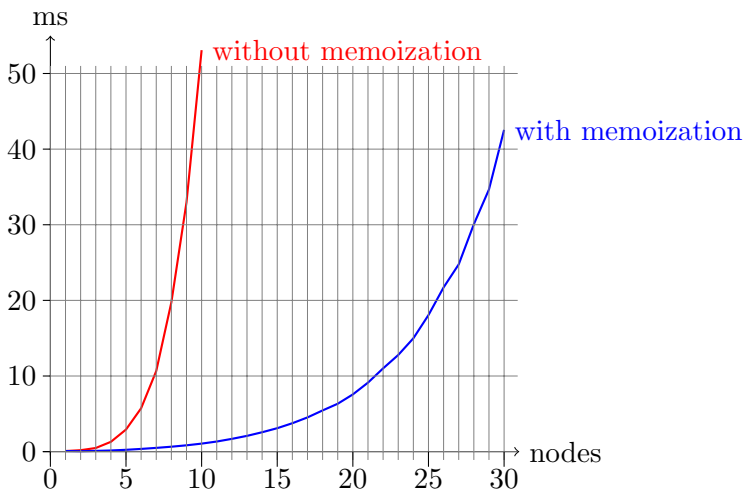


Figure 3.3: Time required to determine all properties of ten random graphs with memoization and without.

It is obvious to what a huge degree memoization increases performance. Although even without memoization the algorithms would likely be fast enough for the average graph used in the Abstract Rewrite Tool, it is clear that the computational complexity would very quickly become unpractical for bigger graphs.

3.4.2 Exploiting Relationships Between Properties

As we have seen in Section 2.4 some properties and combinations of properties have relationships to one another. In theory this can of course be used to in-

[1] For all benchmarks in this thesis one core of an *AMD Phenom II X4 920 2.8 GHz* was used.

crease the performance of the Abstract Rewrite Tool. Since not all relationships between properties actually reduce the computational need we will now analyze which relationships can be harnessed. One thing to keep in mind is that if all properties have to be determined a fixed sequence has to be followed. This alone already limits the relationships that can be used in practice. A good idea to determine the sequence is to evaluate the properties in their order of complexity since we want to omit as many calculations with high complexity as possible. This leads to the following sequence: $WN \ \& \ UN \Rightarrow SN \Rightarrow WCR \Rightarrow CR$.

It should be clear that with that order for example the relationship SN(a) $\rightarrow$ WN(a) is not useful. A list of all relationships that could be used with this sequence looks as following:

1. $\neg$WN(a) $\rightarrow$ $\neg$SN(a) (contrapositive of Lemma 2.19)
2. $\neg$UN(a) $\rightarrow$ $\neg$CR(a) (contrapositive of Lemma 2.20)
3. $\neg$WCR(a) $\rightarrow$ $\neg$CR(a) (contrapositive of Lemma 2.21)
4. SN(a) $\wedge$ UN(a) $\rightarrow$ CR(a) (Lemma 2.22)

The relationship $\neg$UN(a) $\rightarrow$ WN(a) (Lemma 2.23) is not being used since unique normalization and weak normalization are determined together in the same algorithm.

Performance

Since this relationships can only be used in some cases the worst-case-complexity is not reduced and stays at $O(n^3)$. To determine the impact of this optimization in practice some benchmarks were taken. As in the benchmarks for memoization it was measured how long it takes to determine all properties of ten random graphs with differing numbers of nodes. The results can be seen in Figure 3.4 3.4.

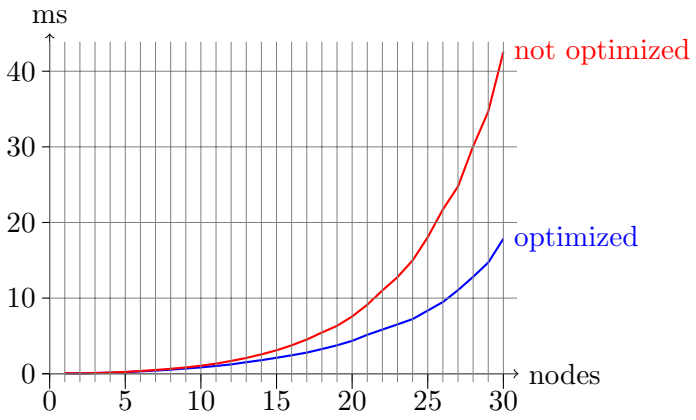


Figure 3.4: Time required to determine all properties of ten random graphs with utilization of relations between properties and without.

It is clear that the optimized version is much more efficient. In another test it could also be determined that most of the performance gain can be attributed to the relation $\neg$UN(a) $\rightarrow$ $\neg$CR(a).

Implementation

If memoization is implemented it is very easy to implement the relationships as well. For example if the function checking for weak normalization is called and it finds that the property does not hold additional to memoizing the result for weak normalization it also memoizes that strong normalization does not hold.

3.5 LaTeX – Output

To use the created graphs in documents it is possible to export them in LaTeX-format. Specifically the TikZ package[2] is being used to draw the graphs in LaTeX.

3.5.1 Conversion of Coordinates

TikZ supports absolute positioning, so it is relatively easy to translate coordinates used for onscreen visualization to coordinates usable in TikZ. There are two issues that have to be considered while converting from ART to TikZ:

- In TikZ a node with a greater value for the y-axis will lie further to the top of the page while it will be further to the bottom of the screen in the ART.
- In TikZ one unit refers by default to one centimeter while it refers to one pixel in the ART.

A good ratio between centimeters and pixels of 1:70 was determined by trial and error. The different orientation of the y-axis can be resolved by negating the y-value. The resulting conversion formulas from ART to TikZ can be seen in Equation 3.1 and 3.2.

$$X_{\text{TikZ}} = \frac{X_{\text{ART}}}{70} \tag{3.1}$$

$$Y_{\text{TikZ}} = -\frac{Y_{\text{ART}}}{70} \tag{3.2}$$

3.5.2 Graph Drawing in TikZ

The graph has to be embedded in a TikZ environment which is opened and closed by `\begin{tikzpicture}` and `\end{tikzpicture}`.

One node can be defined by `\node (`*ref*`) at (`*x-pos*`, `*y-pos*`) {`*text*`};` where *ref* is an identifier that can be used to refer to the node, *x-* and *y-pos* is the absolute position in centimeters and *text* is the text that will be displayed in the document to name the node.

The nodes can then be connected by edges via `\draw[->] (`*from-reference*`) -- (`*to-reference*`);` where *from-* and *to-reference* are two identifiers given to nodes earlier. To draw an edge that has the same origin and destination `\draw[->] (`*reference*`) edge [loop above] (`*reference*`);` can be used.

[2] `http://sourceforge.net/projects/pgf`

For an example how the right hand side graph of Figure 4.1 can be defined in TikZ, see Listing 3.6.

```
\begin{tikzpicture}[scale=1.0]
  %nodes
   \node (d)  at (2.4285715,-0.94285715) {d};
   \node (a)  at (3.5714285,-0.6) {a};
   \node (c)  at (4.6,-0.4) {c};
   \node (f)  at (3.6285715,-2.5285714) {f};
   \node (g)  at (3.0428572,-0.0) {g};
   \node (b)  at (2.7571428,-1.9285715) {b};
   \node (e)  at (3.942857,-1.4571428) {e};
  %edges
   \draw[->] (a) -- (e);
   \draw[->] (c) -- (d);
   \draw[->] (c) -- (f);
   \draw[->] (f) -- (g);
   \draw[->] (f) -- (e);
   \draw[->] (b) -- (a);
   \draw[->] (b) -- (c);
   \draw[->] (e) -- (g);
   \draw[->] (e) -- (b);
\end{tikzpicture}
```

Listing 3.6: TikZ definition of the right-hand side graph in Figure 4.1.

Unfortunately the conversion of ART-coordinates to TikZ coordinates does not always give satisfactory results since the size of a graph in the ART can vary widely and a constant relation between pixels and centimeters does not always suffice. Fortunately TikZ is able to scale a graph by passing `scale=x` to the TikZ environment where `x` should be smaller then one to shrink the graph and bigger than one to enlarge it. So to shrink the size of a graph by 50% the TikZ picture should start with `\begin{tikzpicture}[scale=0.5]`.

4 Graph Drawing Algorithm

Graph drawing algorithms try to generate a pleasing and easy to understand visual representation of a given graph. This usually is done by calculating positions for all nodes of the graph on a 2D plane. What "visually pleasing" and "easy to understand" exactly means may vary from situation to situation but as few intersections of edges as possible and a high symmetry are usually regarded as good properties.

Figure 4.1 illustrates this very well. The two graphs are identical besides the placement of the nodes. On the left side the nodes were put on a random position, on the right side the positions were calculated by the graph drawing algorithm used in the Abstract Rewrite Tool. It should be clear that it is much easier to understand the structure of the graph in the symmetrical and nonoverlapping representation.

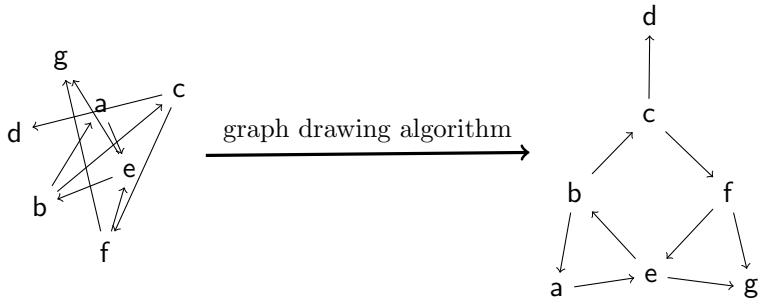


Figure 4.1: Difference between randomly positioned nodes and node positions calculated with a graph drawing algorithm.

Since the Abstract Rewrite Tool should allow interactive editing of graphs the algorithm used had to meet certain additional criteria. It had to be possible to add and remove edges and nodes interactively without disrupting the whole graph and confusing the user by suddenly presenting a completely different representation of the graph.

To accommodate these special requirements it was decided to implement a slightly altered version of *Graph Drawing by Force-directed Placement* [2] in the ART.

Initially it was also considered using JUNG[1] instead which is a graph framework for Java but several problems led to the decision of not using it. For example many supported methods to draw a graph do not allow the direct manipulation of the graph while still automatically arranging the nodes. The

[1] http://jung.sourceforge.net/

method that does support direct manipulation which is based on a spring system seemed not very good at preventing intersecting edges and still seemed very slow and stiff. Additionally JUNG is not only a system to display graphs but also a framework to analyze them. This would have meant that many algorithms and methods described in this thesis would already have been implemented in the framework. Since we did not want to just glue frameworks and libraries together it was ultimately decided to implement everything from scratch.

4.1 Graph Drawing by Force-Directed Placement

Graph Drawing by Force-Directed Placement (GDFP) is based on the simulation of a physical system and uses ideas from *Simulated Annealing* to compensate some shortcomings of the algorithm.

4.1.1 The Physical Component

In GDFP the positions of nodes are determined by running a simulation of a system that consists of particles that repulse each other (for example electrons) and springs connecting them.

It should be noted that what is actually simulated is a very abstracted form of a physical system. For example forces do not influence particles by accelerating them (in fact inertia is not simulated at all). A force in this context just means a movement or displacement to or from the origin of that force. The formulas used to "simulate" repulsion of particles and attraction by springs are also not physically valid formulas.[2] What remains is a simulation that captures the basic premise of a system consisting of electrons and springs and feels natural when working with it interactively.

A system consisting of two nodes will look as shown in Figure 4.2 and there will be two forces acting on both particles. An attracting force f_a and a repulsing force f_r. After a finite number of simulation steps the two nodes will end up at a relative distance where $f_r = f_a$.

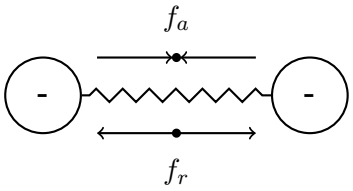


Figure 4.2: Basic model of the particle-spring-system.

Similar to an actual physical system consisting of charged particles and springs the repulsing force gets stronger the closer two nodes get to each other and the attracting force gets stronger the farther apart two nodes get.

[2] It has also to be noted that it would be most difficult to glue springs onto electrons in practice.

This relationship can be seen in Figure 4.3. The distance where the two forces cancel each other out can be changed by setting the constant k.

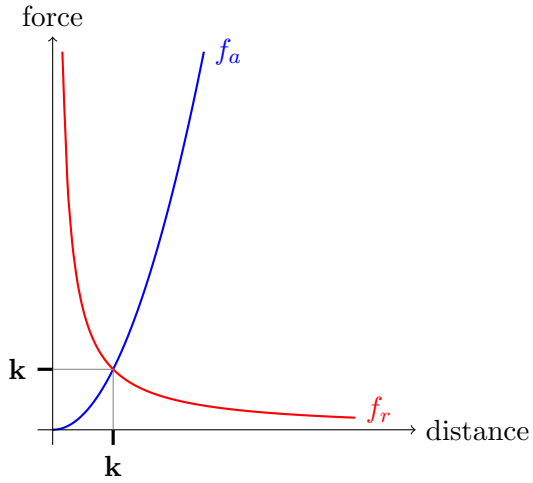


Figure 4.3: Relationship of repulsive and attractive force.

The exact formulas for the forces used in the algorithm can be seen in Equations 4.1 and 4.2 where d represents the distance between two nodes and k the medium distance between nodes that the system should settle on.

$$f_a(d) = \frac{d^2}{k} \tag{4.1}$$

$$f_r(d) = \begin{cases} \frac{k^2}{d} & d \neq 0 \\ \infty & d = 0 \end{cases} \tag{4.2}$$

The conditional branching in Equation 4.2 for f_r is needed to prevent division by zero. If the distance is zero, the repulsive force is assumed to be infinite. In practice the arbitrary (but high enough) value of 1000 is used. This is of course an inaccuracy of the formula but has negligible influence on the final result since forces over 300 are usually capped to keep the system stable. This will be described in more detail shortly.

Application on a whole graph

To translate a whole graph into such a system consisting of electrons and springs each node is converted into a particle and every edge is translated into a spring. The direction of an edge is irrelevant. If there are elements $a, b \in A$ with edges $a \to b$ and $b \to a$, only one of the edges is accounted for. An example for such a translation can be seen in Figure 4.4.

As can be seen in Figure 4.5, this means that the repulsive forces of all other nodes are acting on a single node but only nodes which are connected by an edge act by an attracting force.

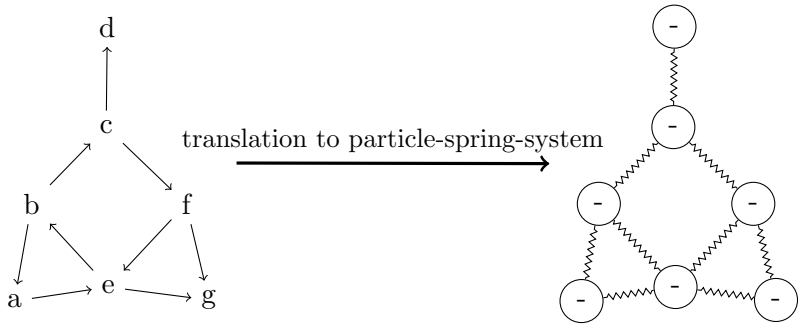


Figure 4.4: Translation of a graph into a particle-spring-system.

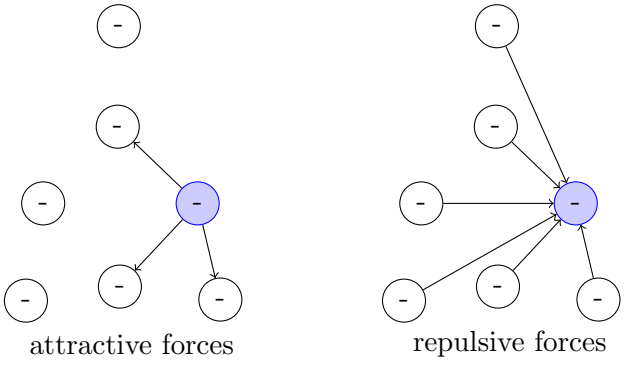


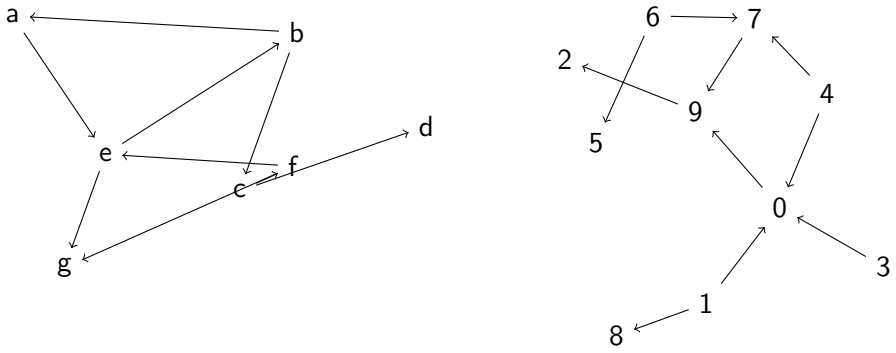
Figure 4.5: Forces on one node in the system.

Problems

In an actual application there are two main problems with this algorithm which arise from the amount of displacement of a node that is allowed in one step of the simulation.

No limitation to displacement. If the amount of displacement per step is not limited at all the algorithm never settles at a stable point. This basically means that the algorithm is not able to do what it is supposed to do. Figure 4.6a shows that the results are not much better than just randomly placing the nodes.

Only a very small displacement is allowed. If the amount of displacement is restricted to a very low amount the system becomes stable, but the time to reach that point is very long since the system can only change by a small amount in each step. There is also the problem that certain intersecting edges cannot be resolved, because one node would have to move through an area of strong repulsive forces. If the displacement is not limited the node often just bypasses that area in one step. In Figure 4.6b nodes 2 and 5 have this problem. To resolve the intersection between $9 \rightarrow 2$ and $6 \rightarrow 5$ nodes 5 and 2 would have to change places. But in the process of doing so they would have to come closer together which is of course prevented by the repelling force between 2 and 5.



a.) no limitation b.) one pixel limitation

Figure 4.6: How different amounts of allowed displacement in one step of the simulation influence the result.

Unfortunately there is not one amount of displacement that would solve the issues. The displacement has to be limited very much for a stable system to arise but it has to be unlimited for disentangling the intersection of edges. Force-directed placement resolves this contradiction by using a method modeled on *simulated annealing*.

4.1.2 Simulated Annealing

Simulated Annealing is a heuristic which combines *hill climbing* with a random element [3, Chapter 5.1]. A global parameter called *temperature* influences the probability that an actually worse state than the current one is chosen next in the search. The temperature is lowered by *cooling* over the length of the search. So simulated annealing gradually moves from random walk to hill climbing.

In force-directed placement the global temperature influences the amount of displacement allowed in each step and is lowered with each iteration by a cooling-function. This leads to disentangling of intersections at the beginning and a stable system at the end.

The analogy is quite striking. At high temperatures the particles move more or less randomly and by lowering the temperature the particles begin to crystallize in an orderly fashion.

It has to be said though that in our opinion the use of the term *simulated annealing* is a bit of a misnomer. Although the analogies used in classical simulated annealing actually fit force-directed placement very well (even better than in the traditional use) the core of classic simulated annealing is very different from the process used here since force-directed placement does not really perform a search.

Simulated Annealing in the ART

In the ART the temperature should never reach too low a value because the graph becomes unresponsive if it does. So it was decided to start the temperature at 300 and lower it to a minimum value of 3. Equation 4.3 shows the used formula, Equation 4.4 shows how it is applied. Figure 4.7 shows the resulting temperature curve.

$$cool(x) = \max\left(\frac{4}{5}x, 3\right) \tag{4.3}$$

$$\text{temperature}_{\text{new}} = cool(\text{temperature}_{\text{current}}) \tag{4.4}$$

Admittedly the formula used for cooling was developed by trial and error and a better one could probably be developed with a systematic approach. *Genetic Programming* could very well be a good method to optimize this function.

4.1.3 Optimization

If the graphs get larger the main performance bottleneck becomes the calculation of the repulsive force because it has to be calculated between all pairs of nodes.

Since nodes that are far apart only apply very small forces on each other they can in practice be neglected without changing the result very much. So for every node instead of calculating the repulsive force to every other node in the graph it suffices to only consider nodes up to a maximal distance.

The difference between the algorithm without and with this optimization can be seen in Figure 4.15 in the benchmark section.

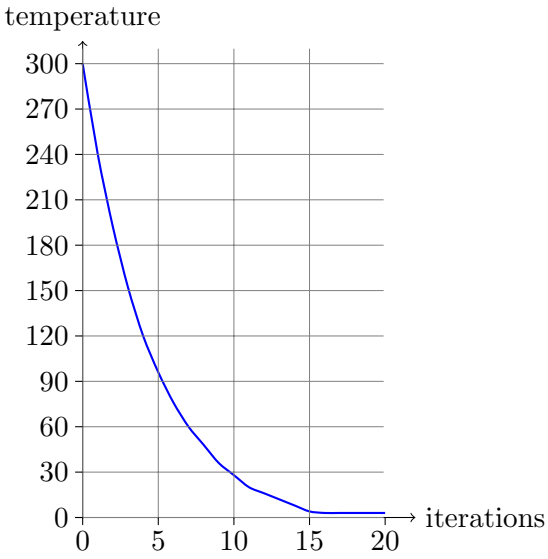


Figure 4.7: Temperature curve resulting from implemented cooling function.

4.1.4 Alterations to the Original Algorithm

Avoidance of Intersections

It can of course happen that even after the system has settled on a stable point some edges remain intersected, be it because the algorithm did not work properly or the graph is not planar.[3]

A practical way to optimize the algorithm in this regard is to run the algorithm again if intersections are detected below a certain temperature. The temperature chosen should represent a point where disentangling of intersected edges becomes very unlikely.

If such a situation is detected, the nodes get new random positions, the system is set to the highest temperature and a counter is decremented. After a certain amount of retries it can be assumed that the graph cannot be drawn without intersections.

To determine the optimal amount of retries an experiment was done. The algorithm ran on randomly generated graphs with 10 nodes[4] on average. The limit of retries was set to 40 which would usually be far too high. It was then determined how many retries were actually needed to acquire a visual representation of the graph without intersecting edges. If the algorithm had to be restarted 40 times it was assumed that the graph could not be drawn without intersections (at least not with the used algorithm).

Figure 4.8 illustrates what percentage of graphs that could be drawn without intersections would still contain intersections after a certain amount of retries. It

[3] Which means, the graph cannot be visualized without intersecting edges in 2 dimensions.

[4] This is the amount of nodes the ART generates for a random graph and seems to be a good size and complexity for graphs the tool will most likely be used for. There are usually a few more edges than nodes.

was decided that 98% visualized graphs without intersection was an acceptable compromise and the amount of retries was set to 6.

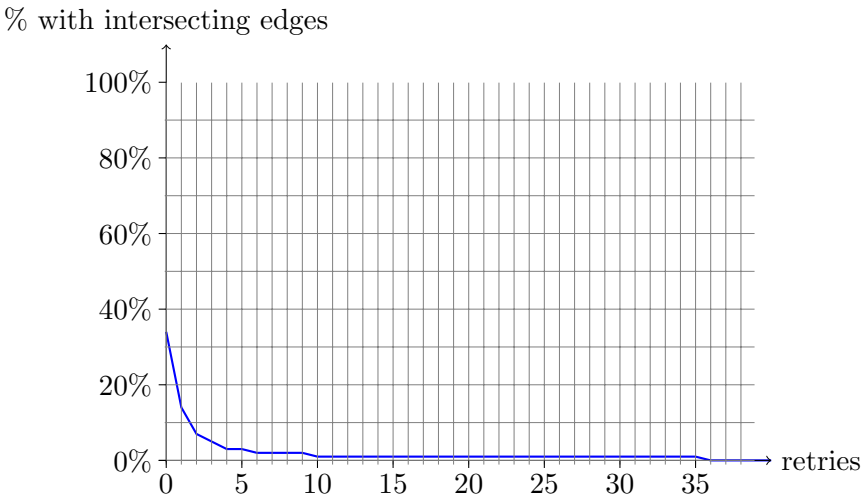


Figure 4.8: Percentage of graphs still containing intersections after a certain amount of retries (10 nodes).

Figure 4.9 shows the same for graphs with 20 nodes. In that case 20 retries would be necessary to acquire 98% non intersecting graphs. This is still feasible but it is clear that for bigger graphs the algorithm itself would have to be improved upon.

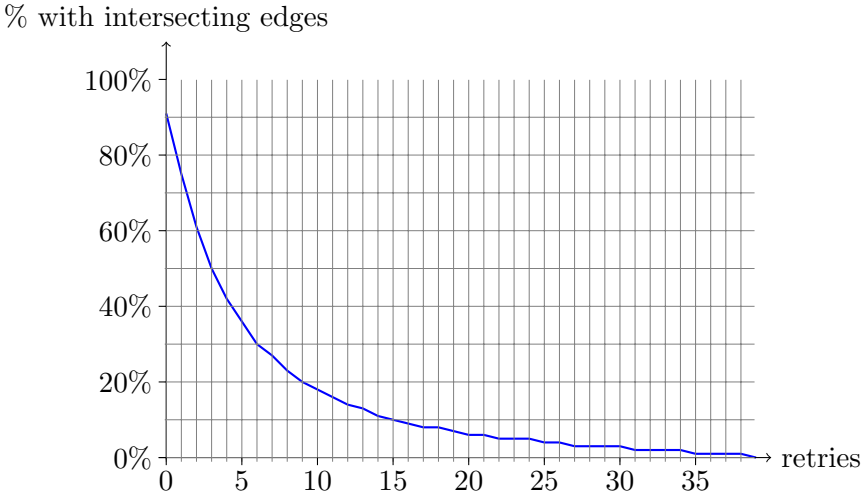


Figure 4.9: Percentage of graphs still containing intersections after a certain amount of retries (20 nodes).

Centering the Graph

It is easy to extend the algorithm in a way, so the graph always tends to stay centered. To achieve this, for every node the distance to the center is determined and a centering force f_c is calculated. The mean value of all displacements that would result from that force is calculated and applied to every node. This leads to a movement of the whole graph to the center without actually disturbing the graph itself.

4.2 Shortcomings of the Algorithm

The algorithm seems to have problems with a certain kind of subgraphs. Figure 4.10 shows such a graph.

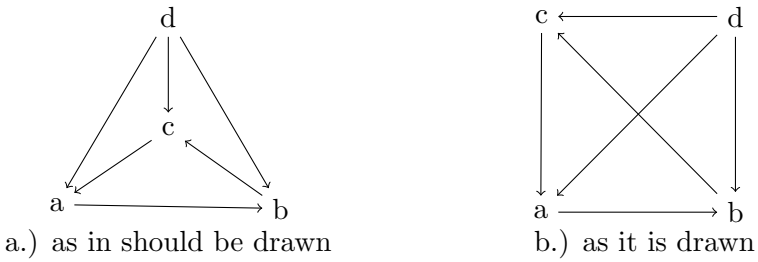


Figure 4.10: Example for a graph the algorithm has difficulty with.

The problem seems to be that the node in the middle is very unstable in that position because of all the repelling forces from the surrounding nodes. So if the node is not very well centered or the nodes that make up the triangle are moved too much, the node is ejected from the middle and the graph moves into a configuration seen in Figure 4.10b which is very stable, but has an intersection.

It is not clear how this behavior of the algorithm could be alleviated and it seems to be the biggest flaw of this graph drawing approach.

4.3 Implementation

In the implementation of the Abstract Rewrite Tool every node in the graph is able to carry a "payload" which in this case only contains four integer values. Two represent the current x- and y-position of the node, the other two stand for the amount of x- and y-displacement in the current step of the simulation.

The additional values for displacement are necessary because one cannot directly change the position of nodes during one iteration. This would lead to the mixing of new and old positions in one step of the simulation. One possible solution would be to create a copy of the graph but since the two integers are a minuscule part of the whole structure by which the graph is represented this would be a waste of memory and processing time.

So it was chosen to not actually change the position during one iteration but merely save the amount of change in two displacement values (x- and y-axis)

and change the position of all nodes once the old positions are no longer needed. This also makes it possible to sum all the displacements from the different forces for each node and limit the amount of total displacement at the end.

4.3.1 Important Formulas and Algorithms

Calculating the Distance

The distance D between two nodes (n_1 and n_2) consisting of an x- and y-component (x_{n_1}, y_{n_1}, etc.) can be very easily calculated via Euclidean distance.

$$D = \sqrt{(x_{n_1} - x_{n_2})^2 + (y_{n_1} - y_{n_2})^2}$$

Calculating the Angle Between Two Nodes

Given two nodes n_1 and n_2 the angle between those nodes is defined as follows. The node n_1 is assumed to lie at the origin of a two dimensional coordinate system. The angle between the x-axis and the line going through the origin and n_2 is the angle we want to determine, see Figure 4.11.

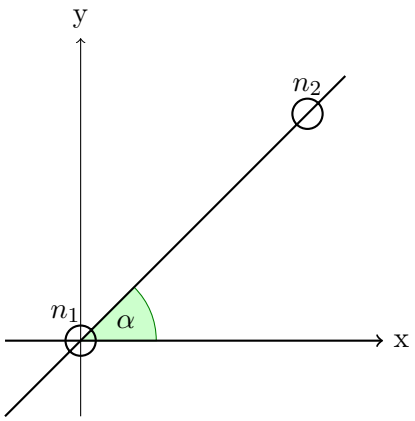


Figure 4.11: The angle between two nodes.

Using basic trigonometry, the angle α is calculated as follows:

$$x = x_{n_1} - x_{n_2} \tag{4.5}$$

$$y = y_{n_1} - y_{n_2} \tag{4.6}$$

$$\alpha = \text{sign}(y) * \arccos\left(\frac{x}{\sqrt{x^2 + y^2}}\right) \tag{4.7}$$

Calculating the Components of a Displacement

Given the nodes n_1, n_2 and a distance D the node n_1 should be displaced by the amount of D along the axis created by n_1 and n_2, see Figure 4.12. So if the distance is positive, n_1 should be moved towards n_2, if the distance is negative it should be moved away from n_2.

So what the formulas should give us is an x- and y-value by which n_1 has to be moved. To calculate the angle α between n_1 and n_2, see Equation 4.7.

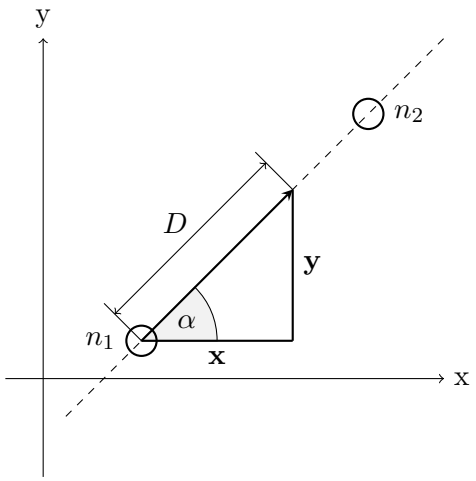


Figure 4.12: Displacement of a node.

The x- and y-values are calculated as follows:

$$x = D * \cos(\alpha) \tag{4.8}$$
$$y = D * \sin(\alpha) \tag{4.9}$$

Checking if Two Lines Intersect Each Other

To avoid intersections, the algorithm first has to be able to detect them. The method used here is based on [7, Chapter 24].

The function *ccw* calculates if a line given by the points p_1 and p_2 has to be rotated clockwise or counterclockwise around p_1 to point in the direction of p_3, see Figure 4.13. The function *ccw* returns $+1$ if p_3 lies counterclockwise and -1 if it lies clockwise. There are also the cases when p_1, p_2 and p_3 are colinear.[5] In that case the results are defined as follows: If p_3 is between p_1 and p_2 *ccw* will return 0, if p_1 is between p_2 and p_3 *ccw* will return -1 and if p_2 is between p_1 and p_3 *ccw* will return $+1$. An implementation of the function *ccw* in Scala can be seen in Listing 4.1.

Given two points p_1 and p_2 on a two dimensional plane, the line connecting those points will be denoted by $\overline{p_1p_2}$.

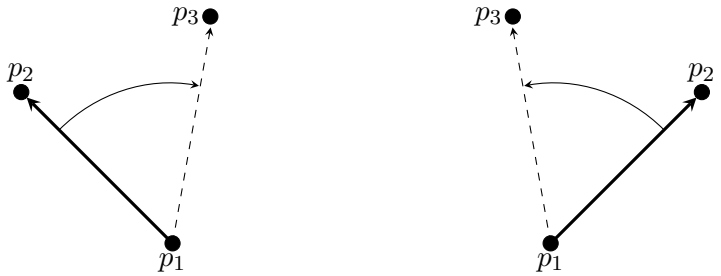
The function *ccw* can now be used to determine if a line $\overline{p_1p_2}$ intersects with a line $\overline{p_3p_4}$. If two endpoints of $\overline{p_1p_2}$ and $\overline{p_3p_4}$ are actually the same point it is assumed that the lines do not intersect.

If the points p_3 and p_4 lie on different sides from $\overline{p_1p_2}$ (so one point has the orientation clockwise and the other one counterclockwise) and p_1, p_2 lie on different sides from $\overline{p_3p_4}$ both lines intersect. Otherwise they do not. An implementation of that algorithm in Scala can be seen in Listing 4.2 .

[5] Which means that p_1, p_2 and p_3 are on one straight line.

```
def ccw(p1: Point, p2: Point, p3: Point): Int = {
  val dx1 = p2.x - p1.x
  val dy1 = p2.y - p1.y
  val dx2 = p3.x - p1.x
  val dy2 = p3.y - p1.y
  if (dx1 * dy2 > dy1 * dx2) return -1
  if (dx1 * dy2 < dy1 * dx2) return 1
  if ((dx1 * dx2 < 0) || (dy1 * dy2 < 0)) return -1
  if ((dx1 * dx1 + dy1 * dy1) < (dx2 * dx2 + dy2 * dy2)) return 1
  return 0
}
```

Listing 4.1: Implementation of *ccw* in Scala.



clockwise: $ccw = -1$ counterclockwise: $ccw = 1$

Figure 4.13: Rotation as it is being defined by the function *ccw*.

4.3.2 Simulation over the Lifetime of a Graph

The lifetime of a graph spans from creation or loading of the graph until another graph is loaded or the program is terminated. In that time different stages in the simulation can be distinguished.

Directly after loading or randomly creating a graph the temperature of the system is very high. This leads to very big changes in every iteration which looks chaotic when visualized. So the simulation runs for 100 iterations before the graph is presented to the user for the first time. At this point the temperature of the system will already have reached the lowest possible point which leads to small changes and a very fluid real time behavior of the graph.

After that we reach the real time state. Here the simulation is run for six

```
def intersects(p1:Point, p2:Point, p3:Point, p4:Point) = {
  if (p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4) false
  else
    ((ccw(p1, p2, p3) * ccw(p1, p2, p4)) <= 0) &&
           ((ccw(p3, p4, p1) * ccw(p3, p4, p2)) <= 0)
}
```

Listing 4.2: Function to check for intersections of two lines in Scala.

iterations for every frame that is shown. The six iterations were chosen by trial and error. It is a good compromise between a high demand for processing power and a quickly reacting graph. If the simulation is only advanced one iteration every frame the graph feels very sluggish. If the number of iterations is too high, it is often difficult to see how the graph changes when edited.

It should also be noted that the number of iterations per frame should be an even number. The simulation can sometimes enter a bistable state where the graph will change between two slightly different visualizations with every iteration. If the number of iterations per frame is odd the graph that will be shown on screen will also switch between those two states which leads to a graph that vibrates with a very high frequency. If an even number of iterations is used, the state that is actually displayed will always be the same.

If the graph is not manipulated with the mouse or edited in some other way the simulation will reach a quasi-stable state within seconds and running the simulation further becomes meaningless. So the simulation will be stopped and only started again for a few seconds once the graph is manipulated in some fashion.

Figure 4.14 shows how a graph transits from an unordered representation to its final form over the course of a simulation.

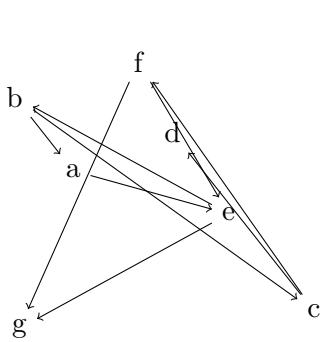
One Iteration in the Simulation

To summarize, one iteration of the simulation proceeds as follows:

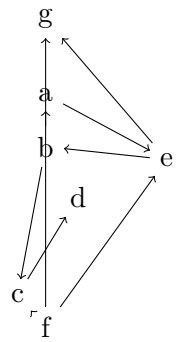
1. Calculate the distance between all pairs of nodes.
2. Calculate the repulsive forces.
3. Displace the nodes according to the calculated forces.
4. If two nodes are connected by an edge:
   a) Calculate the attracting forces
   b) Displace the nodes according to the calculated forces.
5. Calculate the centering force.
6. Displace the nodes according to the calculated force.
7. Limit the displacement of all nodes according to the current temperature of the system.
8. Change the position of all nodes according to the overall displacement.

4.4 Performance

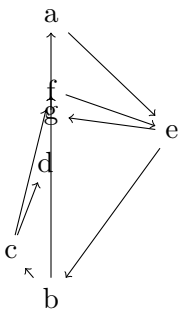
To test the performance of the algorithm benchmark tests were done. The time it takes for 100 iterations in the simulation was measured for graphs of sizes 1 to 40. The influence of the optimization limiting the calculation of repulsing forces discussed in Section 4.1.3 was also tested. The results can be seen in Figure 4.15.



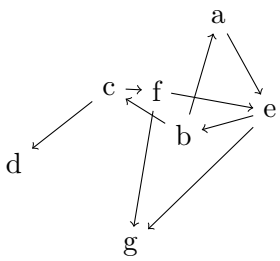
iteration 0



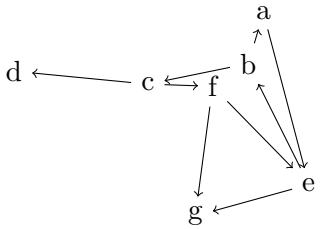
iteration 4



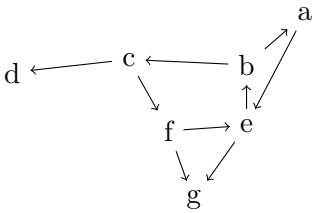
iteration 6



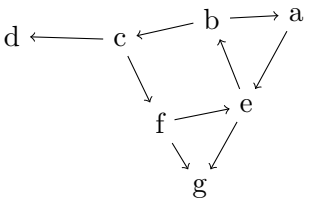
iteration 8



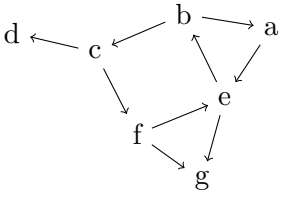
iteration 10



iteration 12



iteration 14



iteration 24

Figure 4.14: Shows how a graph changes its visual appearance over the course of a simulation run.

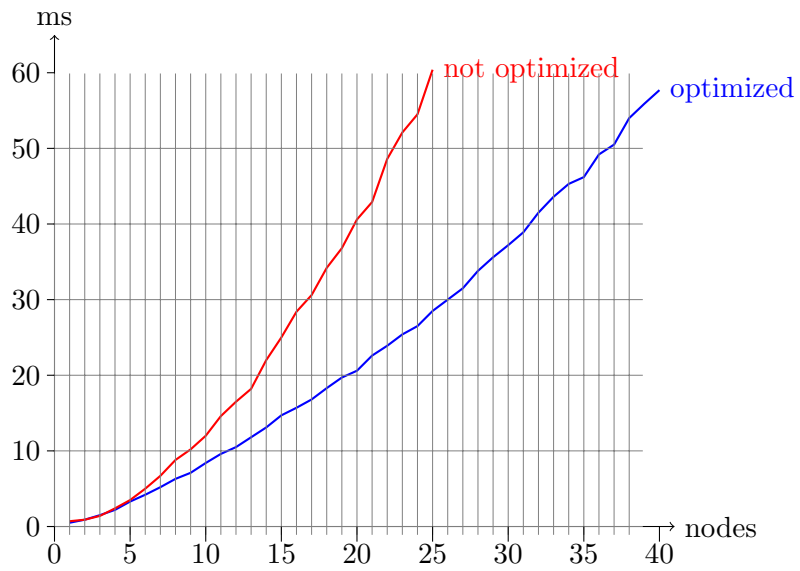


Figure 4.15: Time required for 100 iterations of the simulation depending on the size and complexity of the graph.

5 Usage of the Abstract Rewrite Tool

5.1 Overview

The goal of the Abstract Rewrite Tool was to create a program that on the one hand allowed easy analysis of abstract rewrite systems and on the other hand could be used to test one's own knowledge about such systems in a game-like setting.

5.1.1 Used Tools

Scala

Scala[1] is an object oriented and functional programming language which compiles to *Java Byte Code* and can therefore run on a *Java Virtual Machine*. This also leads to full interoperability between Java-libraries and Scala. For further information about Scala, [6] and [5] can be recommended.

Processing

Processing[2] is a very low level graphics API originally created for designers and artists. It is on one hand a programming language very close to Java but also a framework usable from Java (and therefore Scala). It has the advantages of platform independence and high enough performance for real time applications.

5.2 Installation

The tool can be directly used as a Java-applet in the web browser[3] or downloaded from the tools website.[4] The downloaded version can be started via

```
java -jar art.jar
```

5.3 Usage

The ART offers two modes: The *edit mode* and the *game mode*. To change between the modes press the *g*-key on the keyboard.

[1] `http://www.scala-lang.org/`
[2] `http://www.processing.org/`
[3] `http://www.pi23.net/art2`
[4] `http://www.pi23.net/art2/art.jar`

5.3.1 Edit Mode

In edit mode it is possible to create new and edit existing graphs. To allow an analysis of the graph the properties of all nodes will be shown in a table in the upper-right corner. If the graph is edited the properties will be updated automatically. Additionally nodes which are normal forms have orange text to allow easier analysis of the graph. A screenshot of the ART in edit mode can be seen in Figure 5.1.

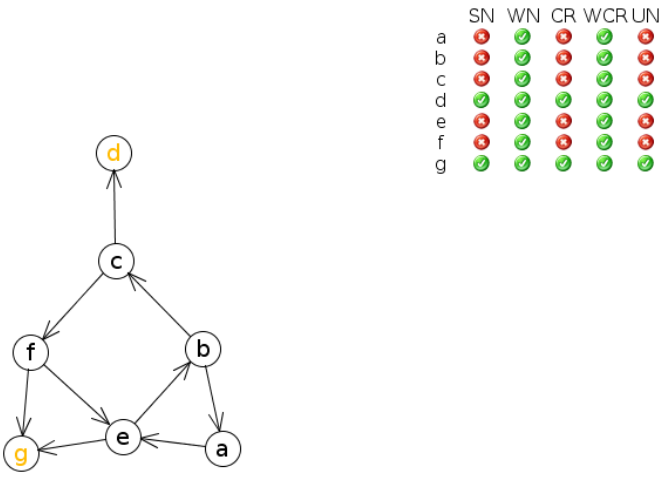


Figure 5.1: Abstract Rewrite Tool in edit mode.

A green checkmark means that the corresponding property holds, a red cross indicates that the property does not hold. A click with the left mouse-button on one of the arks will highlight the selected cell in the table and (for most properties) display an example in the graph why that property does or does not hold. This highlighting will also be automatically updated if the graph is modified. An example for a selection can be seen in Figure 5.2 and the resulting highlighting of the graph in Figure 5.3.

Figure 5.2: Properties in edit mode with strong normalization for node c selected.

Unfortunately it is not always possible to give a meaningful visualization of a property. A description of what is going to be highlighted follows:

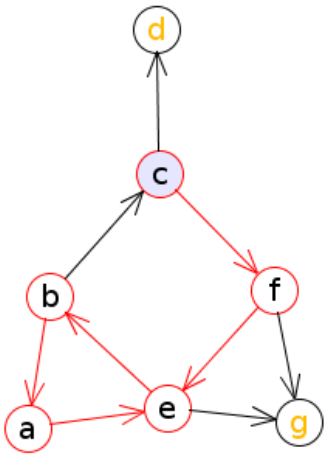


Figure 5.3: Graph in edit mode with highlighting that visualizes why element c is not strongly normalizing.

Strong normalization. If an element is strongly normalizing no meaningful visualization is possible. Otherwise a cycle and a path leading to that cycle will be highlighted.

Weak normalization. If an element is weakly normalizing a path to one normal form will be highlighted. Otherwise all paths to all nodes in the transitive closure will be highlighted. It can be checked that none of the reachable nodes are a normal form.

Church-Rosser property. If the Church-Rosser property holds no meaningful visualization is possible. Otherwise two paths to two nodes that are not joinable will be highlighted.

Weak Church-Rosser property. If the weak Church-Rosser property holds for an element no meaningful visualization is possible. Otherwise two paths to two nodes that are not joinable will be highlighted.

Unique normal forms. In both cases all paths to all nodes in the transitive closure will be highlighted. It can be checked how many normal forms are reachable.

Manipulating the Graph

There are three possibilities to start a new graph. A graph can either be loaded from a file by pressing the l-key on the keyboard, it can be randomly generated by pressing the n-key on the keyboard or an empty graph can be generated for manual creation by pressing the c-key on the keyboard.

It is possible to grab a node and move it around by clicking on it with the left mouse-button and dragging it around. The rest of the graph will dynamically react to the movement.

The editing of a graph can be done by using the mouse and keyboard. The right button on the mouse is the “edit-key”. It is possible to “cut” edges or

nodes by clicking on a free space with the right mouse-button and dragging a line which will cut edges and delete nodes it intersects with after releasing the button. A pair of little scissors at the starting point of the line will indicate that you are about to delete nodes or edges, see Figure 5.4.

Figure 5.4: A line that can be drawn to delete edges and nodes.

To add an edge to the graph right click on a node and drag the mouse to the destination of the edge and release it there. While dragging the line it will actually be displayed as an arrow which indicates that you are about to create a new edge, see Figure 5.5. To create an edge that has the same start- and end-node drag the line far enough away from the node so you can see the arrow and then return to the node and release the mouse-button. If you accidentally started creating an edge just release the mouse-button over an empty area and the command will be omitted.

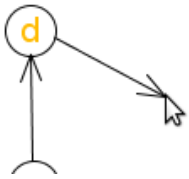


Figure 5.5: A line that can be drawn to add edges.

To create an edge you can also use the so called *parse prompt*. You can open the parse prompt by pressing the *enter*-key on the keyboard. This will open a prompt where you will be able to directly enter a command that will be interpreted and parsed as if it were one line of an ARS-file (Section 3.1), see Figure 5.6. 5

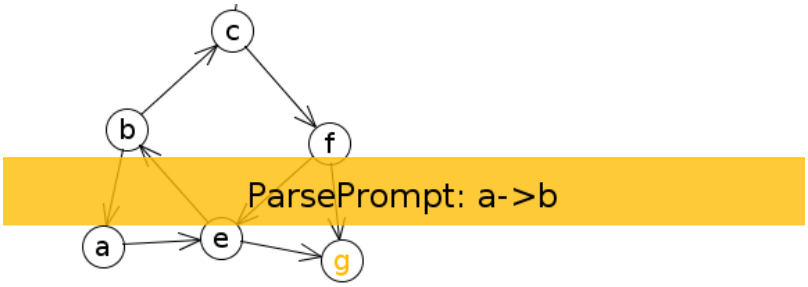


Figure 5.6: A parse prompt with the command `a->b` typed in.

So to create an edge from node a to node b you would enter `a->b` and press

return. To speed up the typing the `->` can be replaced by a colon (`:`) which would lead to `a:b`. It is also possible to create multiple edges at once as long as the origin is the same for all of them. To add an edge from node a to nodes b, c and d the command `a:b,c,d` would suffice. Additionally all nodes which are used in the command and do not exist until this moment will automatically be created.

This also gives us a way to manually create singleton nodes. To create a node enter the parse prompt, type in only the name of the node and press return. Keep in mind that names for nodes must not have more then two symbols otherwise you will get a parse error. To speed up the creation of new nodes you can also use the left mouse-button instead of the return-key to accept the parse prompt.

It is not possible to delete edges with the parse prompt but it is possible to edit a graph completely in a textual form. Pressing the p-key on the keyboard will open a *parse dialog*, see Figure 5.7. Here you will find a textual description of the graph as it would be saved to a file and the graph can be directly altered here. After pressing the OK-button the graph will be revised according to the changes made.

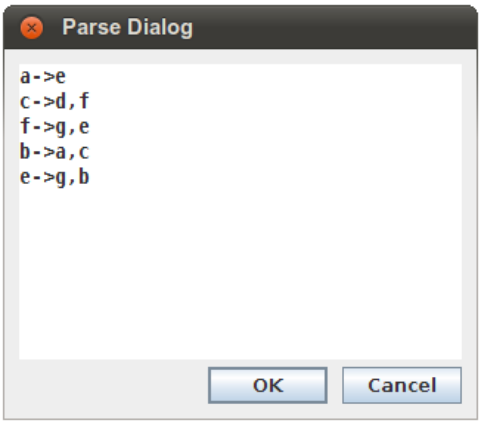


Figure 5.7: A parse dialog containing a textual description of the graph seen in Figure 5.1 and 5.3.

The LaTeX source code of the graph including all properties can be obtained by pressing the x-key on the keyboard. This will open a dialog showing the code.

Edit Mode Cheat-Sheet

Left mouse-button Move nodes.

Right mouse-button Add/delete edges and delete nodes (start dragging at a node to add a new edge).

G Switch to game mode.

ENTER Open parse prompt. You can enter a parsable line (for example `a:b,c,d` to create a node `a` with connections to `b`,`c` and `d`) or enter an identifier to just create a new node. You can accept with enter or the left mouse button.

R Restart automatic alignment of graph.

C Delete the current graph.

N Create new random graph.

L Load file.

S Save file.

P Open parse dialog to copy and paste an ARS or edit the description directly.

F Show frames per second.

X Display the source code for a LaTeX document of the graph.

5.3.2 Game Mode

In game mode you can test your own understanding of abstract rewrite systems. The goal is to decide whether a property holds for a node or not. You should answer this for all properties of all nodes in as short a time as possible and with as few mistakes as you can. If you make a mistake the tool will also give you an explanation why your answer was wrong and it will highlight additional information in the graph itself where it is possible or useful to do so. For highlighting the same examples are used as in edit mode. You can see the number of mistakes and the time you needed until now in the upper portion of the window. A screenshot of the ART in game mode can be seen in Figure 5.8.

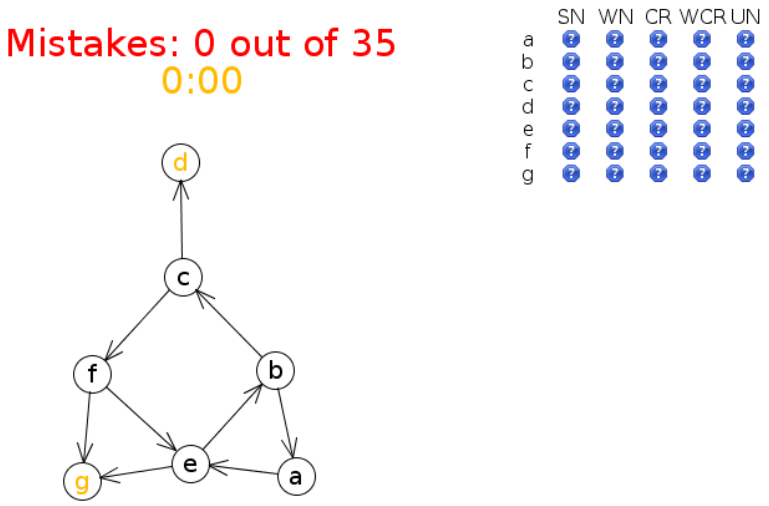


Figure 5.8: Abstract Rewrite Tool in game mode.

If the game is started the upper-right corner holds a table with one node per line and one property per column where all fields are marked with a blue question mark. This means that no answers were given up to this moment. Click with the left mouse-button into one of the fields if you think that this property holds for the corresponding element and click with the right mouse-button if you think the property does not hold. If you answered correctly the field will get a green background and if you gave the wrong answer it will get a red background. Whether you answered correctly or not the right solution will be filled in in the form of a green checkmark if the property holds or a red cross if the property does not hold. Figure 5.9 shows the properties table after answering all properties for all elements. Some of the answers were correct some were not.

Figure 5.9: Completely answered propertyfield in game mode with some correct and some wrong answers.

As was already mentioned in the case of a wrong answer an explanatory message will be shown and (when possible) additional information will be highlighted in the graph. An example for this can be seen in Figure 5.10.

Game Mode Cheat-Sheet

Left mouse-button Move nodes and click to mark a property as true.

Right mouse-button Click to mark a property as false.

G Enter edit mode.

R Restart automatic alignment of graph.

N Create new random graph.

L Load file.

P Open parse dialog to copy and paste an ARS.

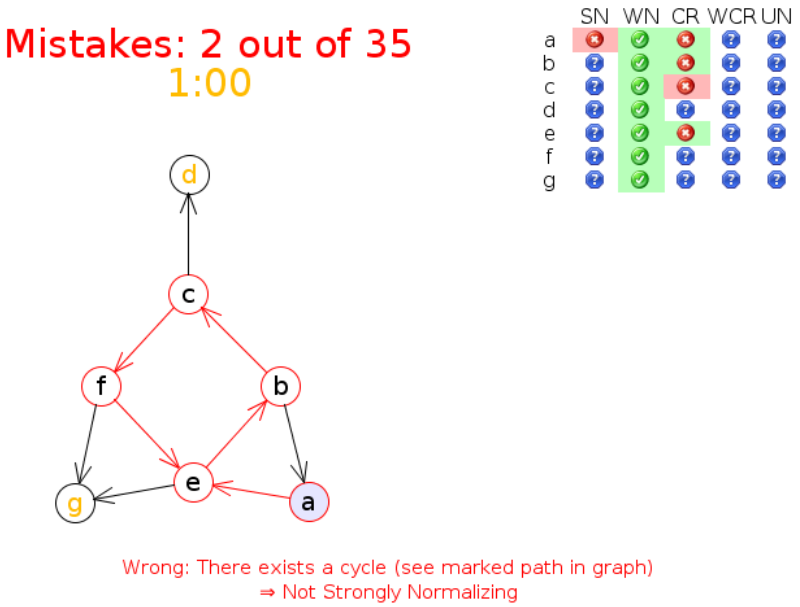


Figure 5.10: Abstract Rewrite Tool in game mode where a wrong answer was given by the user. In this case the user thought that element a was strong normalizing.

6 Conclusion

Summary

Chapter 2 gave an introduction to abstract rewrite systems, their properties and how these properties relate to each other. In Chapter 3 we showed how these properties can be determined algorithmically and discussed some ways in which these algorithms can be optimized. We also presented a method for automatic graph drawing in Chapter 4, possible optimizations and how this method behaves in this context. In Chapter 5 we presented the Abstract Rewrite Tool which was implemented for this project.

We think that the project succeeded in creating a tool that helps teaching abstract rewrite systems to students. It gives them more feedback while learning how to determine properties of such systems and through the LaTeX-output teachers are able to quickly generate examples in an easy way.

Future Work

We do not think that much further work has to be done regarding the Abstract Rewrite Tool itself. One exception would be the evaluation of the possibility of considering infinite abstract rewrite systems in the ART.

On the other hand an extension of the tool's core into a full framework for displaying, editing and manipulating graphs could be valuable. Especially the rigorous utilization of Scala's qualities would most likely result in a framework which would allow the fast and easy creation of graph-based programs without overwhelming the user with a huge system like JUNG.

Bibliography

[1] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.

[2] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[3] Z. Michalewicz and D. Fogel. *How to solve it: modern heuristics.* Springer-Verlag New York Inc, 2004.

[4] A. Middeldorp. *Term Rewriting.* 2009.

[5] M. Odersky. Scala by example (draft). *http://www.scala-lang.org/docu/files/ScalaByExample.pdf*, 2010.

[6] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala.* Artima Inc, 2008.

[7] R. Sedgewick. *Algorithmen in C.* Addison-Wesley, 1992.

[8] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.