Incremental Algorithms and a Minimal Graph Representation for Regular Trees

Matthias Horbach
Sven Woop

Abstract

In this paper we present an efficient way of representing regular trees, which allows operations such as subtree checking and equality tests to be computed efficiently. To do so, we represent regular trees as minimal graphs which are constructed incrementally.

Our work is primarily based on the paper *An incremental unique Representation for Regular Trees* [1] by Laurent Mauborgne, where he studies an algorithm that incrementally finds a unique, minimized representation for regular trees in time $O(n^2)$.

We refine some of his algorithms to gain better complexity, and give a proof of correctness. Furthermore we give our work a theoretical base that allows us to develop useful properties of the trees and graphs we work on, as well as the possibility to use hashing techniques on regular trees.

November 7, 2002
set with $\LaTeX\,2_\varepsilon$

email:
horbach@ps.uni-sb.de
woop@ps.uni-sb.de

Contents

1 Trees and Graphs **3**
1.1 Trees . . . 3
1.2 T-Graphs . . . 4
1.3 Relations on Trees and Graphs . . . 7
1.4 Properties of Minimal Graphs . . . 10
1.5 Natural Order on Trees . . . 11
1.6 Hashing Regular Trees . . . 12

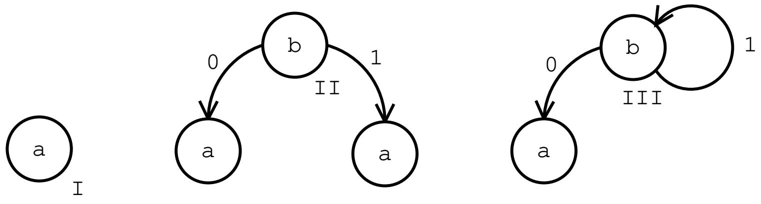
2 The Partitioning Algorithm **15**
2.1 Basics . . . 15
2.2 The Algorithm . . . 17
2.3 Computing the Refinement . . . 19
2.4 Runtime Analysis . . . 20
2.5 Unique Order on the Trees of a Minimal Graph . . . 21

3 Incremental Minimization of Graphs **24**
3.1 Adding a Component . . . 24
3.2 $\mathcal{T}(C) \subset \mathcal{T}(R)$ . . . 25
3.3 $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$ . . . 27
3.4 Optimized Hash Key . . . 28
3.5 Incremental Graph Minimization Algorithm . . . 28
3.6 Runtime Analysis . . . 30
3.7 Building a Regular Tree Database . . . 31
3.7.1 Equality Testing . . . 31
3.7.2 Subtree Testing . . . 31

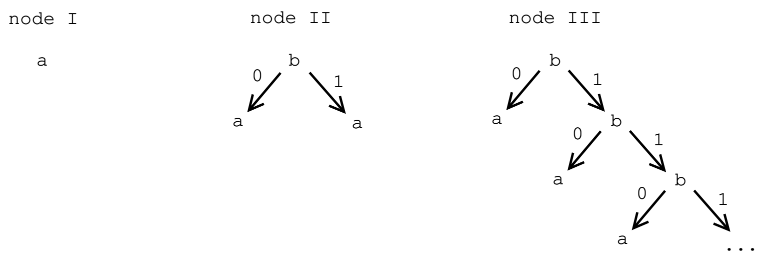
4 Comparison with Mauborgne's Work **31**

1 Trees and Graphs

In this section we will formalize the meaning of trees and graphs and study some relations between both. We will represent regular trees as special graphs like the following ones.



As in automaton theory, where every state corresponds to a regular language, there is a natural notion of the tree defined by a node in the graphs above. For Example the trees of the nodes I, II and III of the graphs above are:



On the other hand, some sets of trees can be represented in graph form. E.g. the sets of trees of the nodes of the example graphs above are represented in the graphs themselves.

Definition 1.1. Let X be a set. Then $\mathcal{P}(X)$ denotes the power set of X. We consider X^* to be the set of *finite strings* over X. The symbol $\varepsilon \in X^*$ denotes the empty string. The *concatenation* pq of two strings p and q over X^* is defined the usual way. If $n > 0$ then $[n]$ is the set $[n] = \{0, \ldots, n-1\}$, and we define $[0] = \emptyset$.

We will now give a precise definition for the notions shown above. For this, assume that there exists a fixed set *Lab* of labels (e.g. the set of integers or strings), and a fixed element $\perp \notin Lab$. For later applications, we will require *Lab* to be provided with a total order.

1.1 Trees

Definition 1.2. A *tree domain* is a subset $D \subseteq N^*$ with the properties

- $\forall p \in \mathbb{N}^*, n \in \mathbb{N} : pn \in D \implies p \in D$
- $\forall p \in \mathbb{N}^* \exists n \in \mathbb{N} : \{m \in \mathbb{N} | pm \in D\} = [n]$

Example 1.3. The set X is a tree domain, the sets $Y_1, \ldots, Y_3$ are not:

$$\begin{aligned} X &= \{\varepsilon, 0, 00, 000, 01, 1\} \\ Y_1 &= \{0, 00, 000, 01, 1\} \\ Y_2 &= \{\varepsilon, 0, 1, 3\} \\ Y_3 &= \{\varepsilon, 0, 1, 2, \ldots\} \end{aligned}$$

Definition 1.4. A *tree* is a partial function $t \in Tree = \mathbb{N}^* \rightharpoonup Lab$ such that $\mathrm{Dom}(t)$ is a tree domain. We write $L(t)$ instead of $t(\varepsilon)$ to access the root of the tree. The *subtree* $t|p$ of t at position p is defined as:

$$t|p \stackrel{\text{def}}{=} sub(t,p) \stackrel{\text{def}}{=} \{(q,l)|(pq,l) \in t\}$$

The set $Sub(t) \stackrel{\text{def}}{=} \{t|p \mid p \in \mathbb{N}^*\}$ is the set of all subtrees of t. A *regular tree* is a tree t such that $Sub(t)$ is finite. When we speak of trees, we will always mean regular trees.

Definition 1.5. Let X be a set of trees. We call X *(subtree) closed*, if

$$\forall t \in X, p \in \mathbb{N}^* : p \in \mathrm{Dom}(t) \implies t|p \in X$$

Example 1.6. The set consisting of the two trees of nodes I and III from above is subtree closed.

1.2 T-Graphs

Definition 1.7. A *T-graph* G is a triple (V, L, E), where

1. V is the set of nodes such that $\perp \notin V$,
2. $L \in V \to Lab$ is a function,
3. $E \in V \times \mathbb{N} \rightharpoonup V$ is a partial function such that $\forall v \in V \exists n \in \mathbb{N} : \{m \in \mathbb{N} | (v,m) \in \mathrm{Dom}(E)\} = [n]$.

Given a Graph $G = (V, L, E)$, V is the set of nodes of G. The function L assigns a label to every such node, and the function E works as a transition function and corresponds to edges between nodes. The arity of a single node u is defined as:

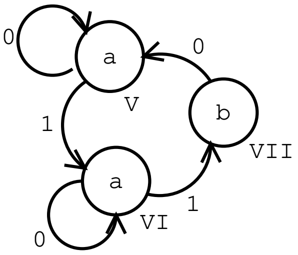
$$arity(u) \stackrel{\text{def}}{=} \#\{n \in \mathbb{N} \mid (u,n) \in \mathrm{Dom}(E)\}$$

The arity of a set X of nodes is the maximal arity of the nodes within the set:

$$arity(X) \stackrel{\text{def}}{=} \max\{arity(u) \mid u \in X\}$$

Definition 1.8. A *finite T-graph* $G = (V, L, E)$ is a T-graph, where the set V of nodes is finite. If nothing else is mentioned, the term *graph* will always refer to a finite T-graph.

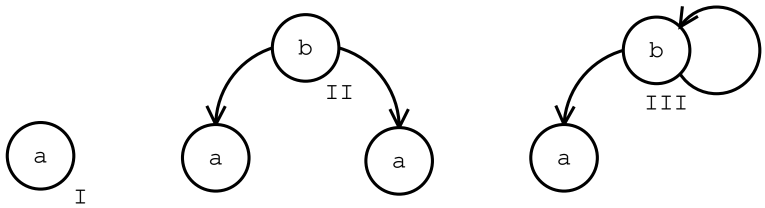
Example 1.9. The following graph is a finite T-graph that we use to illustrate the concepts all over this paper.



This graph has the following finite representation (V, L, E):

$$\begin{array}{rcl} V & = & \{V, VI, VII\} \\ L & = & \{(V, a), (VI, a), (VII, b)\} \\ E & = & \{((V, 0), V), ((V, 1), VI), ((VI, 0), VI), ((VI, 1), VII), ((VII, 0), V)\} \end{array}$$

When we visualize graphs and there is no danger of ambiguity, we will skip the number of each egde and assume edges to be numbered left-to-right. Doing so, the graphs at the beginning of the chapter look like this:



Definition 1.10. Now we define a total infix written extension $. \in V \times \mathbb{N}^* \to (V \cup \bot)$ of the transition function E. Let be $m \in \mathbb{N}$,$p \in \mathbb{N}^*$:

$$\begin{array}{l} v.\varepsilon = v \\ v.mp = \begin{cases} E(v, m).p & \text{if } (v, m) \in \mathrm{Dom}(E) \\ \bot & \text{otherwise} \end{cases} \end{array}$$

In other words $u.p$ is the node that we reach, if we follow the path p starting at u. Note that $\{p \in \mathbb{N}^* | v.p \neq \bot\}$ is a tree domain for every $v \in V$ by the definition of the graph.

Example 1.11. In the graph of example 1.9 we have $V.1 = V.01 = V.001 = V.0^*1 = VI$ and $VII.0110 = V$.

Definition 1.12. Let $X \subseteq V$ be a set of nodes. We call X *closed*, if

$$\forall u \in X, p \in \mathbb{N}^* : u.p \neq \bot \implies t.p \in X$$

Now we can formalize the meaning of the tree defined by a node in any graph.

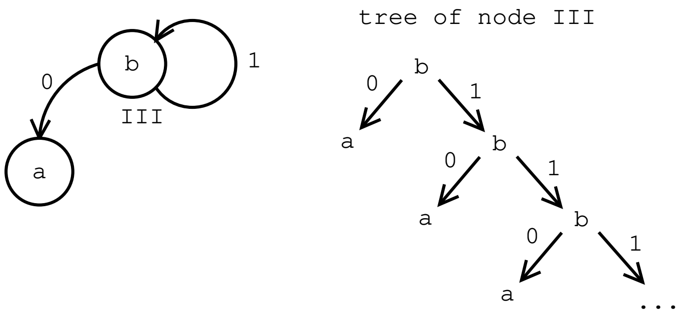
Proposition 1.13. Let $G = (V, L, E)$ be a graph. The function $\mathcal{T} \in (V \cup \{\perp\}) \rightarrow Tree$ defined by

- $\mathcal{T}(\perp) \stackrel{\text{def}}{=} \emptyset$ (the empty function)
- $\mathcal{T}(v)(p) \stackrel{\text{def}}{=} \begin{cases} L(v.p) & \text{if } v.p \neq \perp \\ undefined & \text{if } v.p = \perp \end{cases}$

associates a tree to every node in G, and the empty tree to $\perp$.
If $U \subseteq V$ is a set of nodes, let $\mathcal{T}(U) \stackrel{\text{def}}{=} \{\mathcal{T}(u) \mid u \in U\}$.

Proof. We have to show that the domain of this function is a tree domain. But this is clear, because $\{p \in \mathbb{N}^* | v.p \neq \perp\}$ is a tree domain for every $v \in V$, and $\emptyset$ is a tree domain as well. $\square$

Example 1.14. The tree on the right is the one of node III in the graph on the left.

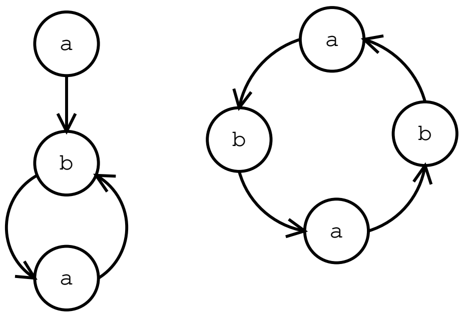


The idea of trees gives a notion of the equivalence of graphs:

Definition 1.15. Two graphs $G = (V, L, E)$ and $G' = (V', L', E')$ are *equivalent*, if the sets of trees defined by the nodes are equal:

$$\mathcal{T}(V) = \mathcal{T}(V')$$

Example 1.16. The following graphs are equivalent:



The opposite way of finding a graph representation for a set of trees only works under special circumstances:

Proposition 1.17. Let $X = t_1, \ldots, t_r$ be a finite, closed set of trees. Then there is a graph $G = (V, L, E)$ such that $\mathcal{T}(V) = X$.

Proof. Let

- $V = X$,
- $L : V \to Lab; \quad L(v) = v(\varepsilon)$, and
- $E : V \times \mathbb{N} \rightharpoonup V; \quad E(v,m) = \begin{cases} v|m & \text{if } m \in Dom(v) \\ \text{undefined} & \text{otherwise} \end{cases}$.

Then $G = (V, L, E)$ is a finite T-graph (the defining properties of G are obviously fulfilled, as well as the finiteness of V). The function E is well defined, as X is subtree closed. We show $\mathcal{T}(u) = u$ by induction on the length of p for each node:

$$\begin{aligned} \mathcal{T}(u)(\varepsilon) &= L(u, \varepsilon) = u(\varepsilon) \\ \mathcal{T}(u)(mp) &= \mathcal{T}(E(u,m))(p) \overset{IH}{=} E(u,m)(p) = (u|m)(p) = u(mp) \end{aligned}$$

The equality $\mathcal{T}(V) = X$ follows directly. □

These connections between the mathematical structures of trees and the implementable data structures of graphs allows us to develop ideas and algorithms on an abstract level. The concrete realization will usually be obvious.

1.3 Relations on Trees and Graphs

Within all the possible relations on graphs, there is one relation that takes a very central role. This relation is the first application of the possibility to connect concrete and abstract views.

Definition 1.18. Let $G = (V, L, .)$ be a graph. Then there is the canonical relation $\sim_{\mathcal{T}} \subseteq V \times V$ defined by

$$u \sim_{\mathcal{T}} v \overset{def}{\iff} \mathcal{T}(u) = \mathcal{T}(v).$$

It is obvious that $\sim_{\mathcal{T}}$ is an equivalence relation. A set X of nodes is *minimal* if $\forall u, v \in X : u \sim_{\mathcal{T}} v \iff u = v$. The graph G is called *minimal* if the set of nodes V is minimal.

Note that this notion of minimality corresponds exactly to the one of automaton or graph minimization, although we avoid playing around with deletion of nodes or similar actions which strongly depend on implementational details.

We consider relations to be partially ordered by inclusion, and refer to this ordering when comparing different relations. Be aware that a *smaller* or *finer* relation in fact yields a *stronger* differentiation and a *greater* set of equivalence classes, whereas a *greater* or *coarser* relation has *less* equivalence classes.

We will usually work on one single graph. In what follows, we always assume that there is one given graph $G = (V, L, E)$.

Definition 1.19. An equivalence relation $\sim \subseteq V \times V$ is a *congruence relation on V* iff the following holds for all $u, v \in V$ and $n \in \mathbb{N}$:

- $u \sim v \implies L(u) = L(v)$
- $u \sim v \wedge u.n \neq \perp \implies v.n \neq \perp \wedge u.n \sim v.n$

Two nodes $u, v \in V$ are called *congruent*, *if* there is a congruence relation $\sim$ such that $(u, v) \in \sim$.

Definition 1.20. An equivalence relation $\sim \subseteq V \times V$ that contains all congruence relations on V ($\forall$ congruences $\sim' \subseteq V \times V : \sim' \subseteq \sim$) is called a *distinction relation*. The set of all distinction relations on V is called $Dist$.
Two nodes $u, v \in V$ are called *distinguishable*, if there is a distinction relation $\sim$ such that $(u, v) \notin \sim$.

Not only among all relations, but especially concerning congruences and distinctions, the relation $\sim_{\mathcal{T}}$ takes a special position:

Proposition 1.21. The equivalence relation $\sim_{\mathcal{T}}$ is both the coarsest congruence relation and the finest distinction relation.

Proof. It is easy to see that $\sim_{\mathcal{T}}$ is a congruence relation. Let $\sim$ be any congruence relation. We have to show that $\sim \subseteq \sim_{\mathcal{T}}$, which means $\mathcal{T}(u) = \mathcal{T}(v)$ whenever $u \sim v$. We show this by induction on the length of paths p in $\mathbb{N}^*$.
Let $u \sim v$ and $p \in \mathbb{N}^*$.
If $|p| = 0$, then $p = \varepsilon$ and $\mathcal{T}(u)(p) = L(u) = L(v) = \mathcal{T}(v)(p)$.
If $|p| > 0$, let $n \in \mathbb{N}, q \in \mathbb{N}^*$ such that $p = nq$. As $u \sim v$, we get (by definition of a congruence relation) that $u.n = v.n = \perp$ or $u.n \sim v.n$. If $u.n = v.n = \perp$, then we are done, since then $u.n.q = v.n.q = \perp$, and hence $\mathcal{T}(u)(p) = \mathcal{T}(v)(p) = \perp$. Otherwise if $u.n \sim v.n$, then we get $\mathcal{T}(u.n)(q) = \mathcal{T}(v.n)(q)$ by induction hypothesis:

$$\begin{aligned} \mathcal{T}(u)(p) &= L(u.p) = L((u.n).q) = \mathcal{T}(u.n)(q) \\ &\stackrel{IH}{=} \mathcal{T}(v.n)(q) = L((v.n).q) = L(v.p) \\ &= \mathcal{T}(v)(p). \end{aligned}$$

So we have $\mathcal{T}(u)(p) = \mathcal{T}(v)(p)$ for all $p \in \mathbb{N}^*$. Hence $\mathcal{T}(u) = \mathcal{T}(v)$ and $u \sim_{\mathcal{T}} v$.

Since $\sim_{\mathcal{T}}$ is the greatest congruence relation, as we just proved, it contains all congruences; so $\sim_{\mathcal{T}}$ is a distinction relation. Any other distinction relation contains all congruences as well, and it contains in paticular $\sim_{\mathcal{T}}$. Therefore $\sim_{\mathcal{T}}$ is the finest distinction relation. ☐

Proposition 1.22. Let $u, v \in V$, then the following statements are equivalent:

1. u and v are congruent
2. u and v are not distinguishable
3. $u \sim_{\mathcal{T}} v$.

Proof. 1. $\iff$ 2.: If u, v are congruent, they are not distinguishable, because the pair (u, v) is contained in one congruence relation and hence in every distinction relation.
Otherwise, the pair (u, v) is contained in no congruence relation. So it is not in $\sim_{\mathcal{T}}$, which makes u and v distinguishable.
2. $\iff$ 3.: This is clear because $\sim_{\mathcal{T}}$ is the finest distinction relation by proposition 1.21. ☐

With these propositions in mind, another characterization of distinction relations is often helpful and easier to verify:

Corollary 1.23. An equivalence relation $\sim$ is a distinction relation iff $\sim \supseteq \sim_{\mathcal{T}}$.

Proof. This follows directly from proposition 1.21. $\square$

Corollary 1.24. If an equivalence relation $\sim$ is both a congruence and a distinction relation, then $\sim = \sim_{\mathcal{T}}$.

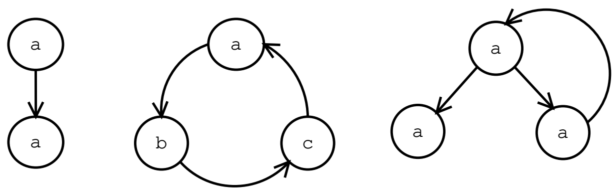
Proof. Proposition 1.21 gives $\subseteq$, because $\sim_{\mathcal{T}}$ is the largest congruence. Corollary 1.23 yields $\supseteq$. $\square$

After all these definitions, we present some basic examples to illustrate the idea of distinction developed above:

Example 1.25.

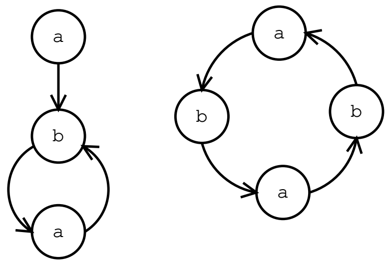
- The finest congruence relation on any graph is defined by $u \sim v \iff u = v$, the coarsest distinction relation by $\sim = V \times V$.

- We will start with some graphs without congruent nodes:

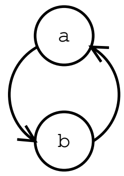


  None of the nodes in each of the presented graphs are congruent, since they differ in label and/or arity and thus denote different trees. This aligns with the intuition that these nodes "mean" something different. Still, for the non-trivial distinction $u \sim v \iff L(u) = L(v)$, nodes with the same label are identified. Note that this relation is no congruence relation.

- In both of the following graphs, all nodes with identical label are congruent:



  A minimal graph equivalent to both is:



1.4 Properties of Minimal Graphs

Proposition 1.26. If G is a minimal graph, then $\mathcal{T}|_V : V \to \mathcal{T}(V)$ defines an isomorphism between nodes and the corresponding trees. So two equivalent minimal graphs are isomorphic, which means that they are identical up to renaming of nodes. Furthermore given a regular tree t, there exists a minimal graph $G = (V, L, E)$ representing t, as $Sub(t)$ is finite and the graph is isomorph to it.

Proof. Let $u, v \in V$ with $u \neq v$. Since G is minimal, it follows that $u \not\sim_{\mathcal{T}} v$, and so $\mathcal{T}(u) \neq \mathcal{T}(v)$. This means that $\mathcal{T}|_V$ is injective (surjective is trivial). Furthermore we have $\mathcal{T}(u.p) = \mathcal{T}(u)|p$ by the following induction on the length of p:

$$\begin{aligned}
\mathcal{T}(u.\varepsilon)(q) &= \mathcal{T}(u)(q) = (\mathcal{T}(u)|\varepsilon)(q) \\
\mathcal{T}(u.np)(q) &= \mathcal{T}(E(u,n).p)(q) \overset{IH}{=} (\mathcal{T}(E(u,n))|p)(q) \\
&= \mathcal{T}(E(u,n))(pq) = \mathcal{T}(u)(npq) = (\mathcal{T}(u)|np)(q)
\end{aligned}$$

As the proof didn't used the minimality of G the equation $\mathcal{T}(u.p) = \mathcal{T}(u)|p$ also holds on arbitrary graphs. ☐

Definition 1.27. Two nodes u, v of a graph are called *strongly connected*, if:

$$(u, v) \in \mathrm{SC} \overset{\text{def}}{\iff} u \sim_{sc} v \overset{\text{def}}{\iff} \exists p, q \in \mathbb{N}^* : u = v.p \wedge v = u.q$$

A set X of nodes is a *strongly connected component*, if for all $u, v \in X$ we have that u and v are strongly connected, and X is maximal with this property. In other words, there exists a node u such that $X = [u]_{SC}$.

Definition 1.28. Two trees t, t' are called *(abstract) strongly connected*, if:

$$(t, t') \in \mathrm{ASC} \overset{\text{def}}{\iff} t \sim_{asc} t' \overset{\text{def}}{\iff} \exists p, q \in \mathbb{N}^* : t = t'|p \wedge t' = t|q$$

Let X be a set of trees, then X is an *(abstract) strongly connected component*, if for all $t, t' \in X$ we have that t and t' are (abstract) strongly connected and X is maximal with this property. In other words, there exists a tree t such that $X = [t]_{ASC}$.

Proposition 1.29. Let u, v be nodes of a graph G. Then $u \sim_{sc} v$ implies $\mathcal{T}(u) \sim_{asc} \mathcal{T}(v)$. We have an equivalence if G is minimal.

Proof. "$\Longrightarrow$": Let $v \sim_{sc} u$. Then there exist $p, q \in \mathbb{N}^*$ such that $u = v.p$ and $v = u.q$. But then we also have $\mathcal{T}(u) = \mathcal{T}(v.p) = \mathcal{T}(v)|p$ and $\mathcal{T}(v) = \mathcal{T}(u.q) = \mathcal{T}(u)|q$ and therefore $\mathcal{T}(v) \sim_{asc} \mathcal{T}(v)$.
If G is minimal, then the equivalence is clear, as V is isomorphic to $\mathcal{T}(V)$ with isomorphism $\mathcal{T}|_V$ by proposition 1.26. ☐

1.5 Natural Order on Trees

In this chapter we define a total order on the set of trees.

Definition 1.30. We call a path $p \in \mathbb{N}^*$ a *separating path* for two trees t_1 and t_2, if $L(t_1|p) \neq L(t_2|p) \vee arity(t_1|p) \neq arity(t_2|p)$.
If $\leq_P \subseteq \mathbb{N}^* \times \mathbb{N}^*$ is total and well founded order on the paths, then we call p the *minimal separating path of* t_1 *and* t_2 *with respect to* $\leq_P$, written $\mathrm{msp}_P(t_1, t_2)$, if p is a separating path that is minimal with this property.

Example 1.31. The length lexicographic order $\leq_{llex} \subseteq \mathbb{N}^* \times \mathbb{N}^*$ defined by the following is a total and well founded order on the set of paths.

$$\begin{aligned} \varepsilon &\leq_{lex} p \\ np \leq_{lex} mq &\overset{\text{def}}{\iff} n \leq m \vee (m = n \wedge p \leq_{lex} q) \\ p \leq_{llex} q &\overset{\text{def}}{\iff} |p| < |q| \vee (|p| = |q| \wedge p \leq_{lex} q) \end{aligned}$$

The order $\leq_{lex}$ alone is *not* well founded, because

$$\ldots <_{lex} 0001 <_{lex} 001 <_{lex} 01 <_{lex} 1.$$

Definition 1.32. Let $\leq_L \subseteq Lab \times Lab$ be a total order on the set of labels. This order can be extended to a partial order $\leq_{LA(L)} \subseteq Tree \times Tree$ on the set of trees:

$$t_1 \leq_{LA(L)} t_2 \overset{\text{def}}{\iff} L(t_1) <_L L(t_2) \vee (L(t_1) =_L L(t_2) \wedge arity(t_1) \leq arity(t_2))$$

This order distinguishes trees by label and arity of the root and depends strongly on $\leq_L$.

Definition 1.33. Let $\leq_P \subseteq \mathbb{N}^* \times \mathbb{N}^*$ be a well founded order on the set of paths and $\leq_L \subseteq Lab \times Lab$ be a total order on the set of labels.
Furthermore let $\leq_{LA} = \leq_{LA(L)} \subseteq Tree \times Tree$ be the partial order defined above. Then $\leq_{LA}$ can be totalized using minimal separating paths. This yields an order $\leq_{\mathcal{T}} \subseteq Tree \times Tree$ on the set of trees as follows:

$$t_1 \leq_{\mathcal{T}} t_2 \overset{\text{def}}{\iff} t_1 = t_2 \vee (t_1 \neq t_2 \wedge p = \mathrm{msp}_P(t_1, t_2) \wedge t_1|p \leq_{LA} t_2|p)$$

Note that $\leq_{\mathcal{T}} = \leq_{\mathcal{T}} (\leq_L, \leq_P)$ depends on both $\leq_L$ and $\leq_P$. For a better readability we usually do not explicitly mention this dependency.

Lemma 1.34. The relation $\leq_{\mathcal{T}}$ is a total order on the set of trees that is uniquely determined by $\leq_P$ and $\leq_L$.

Proof.

1. *reflexive:* $t_1 = t_2 \implies t_1 \leq_{\mathcal{T}} t_2$
2. *antisymmetric:* Assume $t_1 \leq_{\mathcal{T}} t_2 \wedge t_2 \leq_{\mathcal{T}} t_1$ and $t_1 \neq t_2$. Then we have $t_1|p \leq_{LA} t_2|p \wedge t_2|p \leq_{LA} t_1|p \iff L(t_1|p) = L(t_2|p) \wedge arity(t_1|p) = arity(t_2|p)$, in contradiction that p is a separating path.
3. *transitive:* Let $t_1 \leq_{\mathcal{T}} t_2 \wedge t_2 \leq_{\mathcal{T}} t_3$ and all trees be different (otherwise it is trivial). Let p_{12} be the separating path of t_1, t_2, and p_{23} be the one of t_2 and t_3. The case $p_{12} = p_{23}$ is trivial. If $p_{12} <_P p_{23}$, then p_{12}

doesn't separate t_2 and t_3, because p_{23} is minimal. It follows $t_1|p_{12} <_{LA} t_2|p_{12} =_{LA} t_3|p_{12}$. Furthermore p_{12} is the minimal separating path of t_1 and t_3. Otherwise there would be a smaller path p' such that $p' <_P p_{12} <_P p_{23}$ and we would get $t_1|p' \neq_{LA} t_3|p' \overset{p_{23}\,\text{minimal}}{=_{LA}} t_2|p'$, in contradiction to the minimality of p_{23}. The case $p_{12} >_P p_{23}$ is analogous.

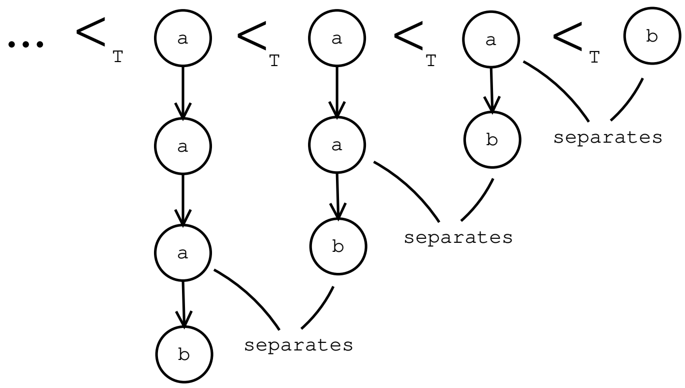
4. *total:* Let t_1, t_2 be two different trees. Then there exists exaxtly one minimal separating path p, as $\leq_P$ is well founded and total. As p separates t_1 and t_2, the partial order $\leq_{LA}$ can compare $t_1|p$ and $t_2|p$.

5. *unique:* Let $\leq'_{\mathcal{T}}$ and $\leq''_{\mathcal{T}}$ be two orders with the property above. If $t_1 = t_2$, we have $t_1 \leq'_{\mathcal{T}} t_2$ and $t_1 \leq''_{\mathcal{T}} t_2$. Otherwise if $t_1 \neq t_2$, then there is a minimal separating path p of t_1 and t_2:

$$\begin{aligned} t_1 \leq'_{\mathcal{T}} t_2 \quad &\iff \quad t_1|p <_{LA} t_2|p \\ &\iff \quad t_1 \leq''_{\mathcal{T}} t_2 \end{aligned}$$

□

If $\leq_P = \leq_{llex}$, then we call $\leq_{\mathcal{T}}$ the *natural tree order*. This order is *not* well founded, like the following example shows:



1.6 Hashing Regular Trees

In this chapter we analyse a method to hash regular trees. The problem is that a regular tree may be infinite in general, so we have to compute a finite representation for it.

Proposition 1.35. The function *key* defined below is injective and assigns a finite tree to each regular tree:

$$\begin{aligned} cycle(t,p) \quad &= \quad \exists q \in \mathbb{N}^* :\ q <_{llex} p \wedge t|q = t|p \\ key(t)(p) \quad &= \quad \begin{cases} t(p) & \text{if } \neg cycle(t,p) \qquad (1) \\ q & \text{if } p = wn \wedge \neg cycle(t,w) \wedge cycle(t,p) \qquad (2) \\ & \wedge\ q = \min_{<_{llex}}\{q' \in \mathbb{N}^* \mid t|q' = t|p\} \\ \text{undefined} & \text{if } p = wn \wedge cycle(t,w) \qquad (3) \end{cases} \end{aligned}$$

Proof. Convince yourself that $key(t)$ defines a tree. We show that key is injective, $t_1 = t_2 \iff key(t_1) = key(t_2)$.

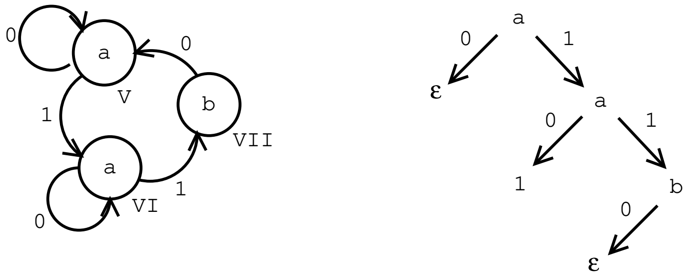
"$\Longrightarrow$": trivial
"$\Longleftarrow$": Let $t_1, t_2 \in Tree$, and p be the separating path of t_1 and t_2 which is minimal with respect to $\leq_{llex}$ and assume $key(t_1) = key(t_2)$. If $key(t_1)(p)$ and $key(t_2)(p)$ are in different cases of the definition of key, then it is clear that we have the contradiction $key(t_1) \neq key(t_2)$, as in case (1) the label of the key is the label of the original tree, in case 2 it is a path and in case 3 it is undefined. Otherwise consider the different cases:

1. If we are in case 1 of the definition, then $key(t_1)(p) = key(t_2)(p) \implies t_1(p) = t_2(p) \wedge arity(t_1|p) = arity(t_2|p)$. But p separates t_1 and t_2. Contradiction.

2. and 3. If we are in case 2 or 3 of the definition, then let q be the longest path such that $p = qq'$, and $key(t_1)(q)$ and $key(t_2)(q)$ are both defined. As q is maximal, $key(t_1)(q)$ and $key(t_2)(q)$ are both in case 2 of the definition. Let $l = key(t_1)(q) = key(t_2)(q)$. Then l is a path with $l <_{llex} q$. Furthermore $t_1|q = t_1|l$ and $t_2|q = t_2|l$. We have $t_1(p) = (t_1|q)(q') = (t_1|l)(q') = t_1(lq')$ and $t_2(p) = t_2(lq')$. But then $lq' <_{llex} p$ is a separating path in contradiction to the minimality of p.

□

Example 1.36. The *key* on the right of the following picture is the *key* of the tree of node V in the left minimal graph. Satisfy yourself that the name of the nodes of the graph have no effect on the key.



To compute a key to a tree t we use a node u in a minimal graph as representation for t. Then the key can be computed using breadth first search. A property of the breadth-first-search algorithm is, that the path p the algorithm follows to a node v is minimal with respect to $\leq_{llex}$. Therefore each time we visit a node v' that we already visited before, we can guarantee that the path p of v is the one the algorithm has to put at the place of v'. Breadth-first-search is in $O(|V|+|E|)$. Consequently the size of the key is also in $O(|V|+|E|)$.

It is easy to see that you can compute the original minimal graph representation out of the key. As the minimal graph is unique for a tree, we defined a injective function between trees and keys. We can compare two trees by comparing the corresponding keys. A key can be easily represented as a finite string and as there exist optimal hashing functions on strings, hashing a tree is no problem after the computation of the key.

2 The Partitioning Algorithm

Since we want to find a unique and efficient representation of trees, we have to remove all redundancy. In terms of graphs this is equivalent to the problem of graph minimization. Hence one of our sub-goals is to minimize a given graph G.

Usually, minimization of graphs (or automata) is done by manipulation of the nodes. We will use a different approach: Successive refinement of a simple distinction relation finally yields $\sim_{\mathcal{T}}$.

In this section, we will give two algorithms to achieve a minimization of graphs, or rather to construct $\sim_{\mathcal{T}}$, a naive and an optimized one, and prove their correctness. The final algorithm runs in $O(n \log n)$, following ideas from Hopcroft [2, 3].

We will usually work on one single graph $G = (V, L, E)$.

2.1 Basics

Before we can continue to the algorithms, we state some further properties of congruence relations, which will be needed to prove the correctness of the algorithms.

Lemma 2.1. Let

$$\begin{aligned} \sim_L \quad &= \{(u, v) \in V \times V \mid L(u) = L(v)\} \\ \sim_{LA} \quad &= \{(u, v) \in V \times V \mid L(u) = L(v) \wedge arity(u) = arity(v)\} \end{aligned}$$

be the relation dividing V by label (and arity). Both $\sim_L$ and $\sim_{LA}$ are distinction relations.

Proof. Both $\sim_L$ and $\sim_{LA}$ are equivalence relations. Furthermore

$$\begin{aligned} u \sim_{\mathcal{T}} v \quad &\Longrightarrow \quad T(u) = T(v) \\ &\Longrightarrow \quad L(u) = L(v) \wedge arity(u) = arity(v) \\ &\Longrightarrow \quad u \sim_{LA} v \implies u \sim_L v, \end{aligned}$$

and so $\sim_{\mathcal{T}} \subseteq \sim_{LA} \subseteq \sim_L$. □

The refinement of distinction relations will be done using the following function:

Definition 2.2. The function $R \in \mathcal{P}(V \times V) \times \mathcal{P}(V) \times \mathbb{N} \to \mathcal{P}(V \times V)$ defined by

$$R(\sim, X, n) := \ (\sim) \cap \{(u, v) \in V \times V | u.n \in X \iff v.n \in X\}$$

is called *refinement*.
Let $\sim, \sim'$ be distinction relations. We write

$$\sim \succ \sim',$$

if there are $X \in V/_\sim$ and $n \in \mathbb{N}$ such that $\sim' = R(\sim, X, n) \neq \sim$.

Lemma 2.3. If $\sim$ is a distinction relation, then $R(\sim, X, n)$ is an equivalence relation with the properties

1. $R(R(\sim, X, n), X, n) = R(\sim, X, n)$ (idempotence)

2. $\forall Y \subseteq V, \forall m \in \mathbb{N} : R(R(\sim, X, n), Y, m) = R(R(\sim, Y, m), X, n)$

3. $X \in V/_\sim \implies R(\sim, X, n) \in \mathit{Dist}$.

Proof. As the intersection of two equivalence relations, $R(\sim, X, n)$ itself is an equivalence relation. For the rest of the proof let $\sim \in \mathit{Dist}$ and $u, v \in V$ be arbitrary.

1. Idempotence:

$$\begin{array}{rl} & (u,v) \in R(R(\sim, X, n), X, n) \\ \iff & (u,v) \in R(\sim, X, n) \wedge (u.n \in X \iff v.n \in X) \\ \iff & (u,v) \in \sim \ \wedge (u.n \in X \iff v.n \in X) \\ \iff & (u,v) \in R(\sim, X, n) \end{array}$$

2. Let $Y \subseteq V, m \in \mathbb{N}$. Then

$$\begin{array}{rl} & (u,v) \in R(R(\sim, X, n), Y, m) \\ \iff & (u,v) \in R(\sim, X, n) \wedge (u.m \in Y \iff v.m \in Y) \\ \iff & (u,v) \in \sim \wedge\ (u.m \in Y \iff v.m \in Y) \\ & \qquad \wedge (u.n \in X \iff v.n \in X) \\ \iff & (u,v) \in R(\sim, Y, m) \wedge (u.n \in X \iff v.n \in X) \\ \iff & (u,v) \in R(R(\sim, Y, m), X, n) \end{array}$$

3. Let $X \in V/_\sim$. Then $\sim_\mathcal{T} \subseteq R(\sim, X, n)$ (remember that $\mathcal{T}(u.n) = \emptyset$ if $u.n = \perp$):

$$\begin{array}{rcl} u \sim_\mathcal{T} v & \implies & \mathcal{T}(u) = \mathcal{T}(v) \\ & \implies & \mathcal{T}(u) = \mathcal{T}(v) \wedge \mathcal{T}(u.n) = \mathcal{T}(v.n) \\ & \implies & u \sim v \wedge u.n \sim v.n \\ & \overset{X \in V/\sim}{\implies} & u \sim v \wedge (u.n \in X \iff v.n \in X) \\ & \implies & (u,v) \in R(\sim, X, n). \end{array}$$

□

Lemma 2.4. Let $\sim \subseteq \sim_L$ be a distinction relation. Then

$$\sim = \sim_\mathcal{T} \iff \forall X \in V/_\sim, n \in \mathbb{N} : \ R(\sim, X, n) = \sim .$$

Proof. "$\implies$": Let be $X \in V/_\sim$ and $n \in \mathbb{N}$. The inclusion $R(\sim_\mathcal{T}, X, n) \subseteq \sim_\mathcal{T}$ is trivial. According to lemma 2.3, $R(\sim_\mathcal{T}, X, n)$ is a distinction relation, because $\sim_\mathcal{T}$ is one. Since $\sim_\mathcal{T}$ is the finest distinction relation by lemma 1.21 we get $\sim_\mathcal{T} \subseteq R(\sim_\mathcal{T}, X, n)$.
"$\impliedby$": For the opposite direction let $R(\sim, X, n) = \sim$ for all $X \in V/_\sim, n \in \mathbb{N}$. As $\sim \subseteq \sim_L$, we get $u \sim v \implies L(u) = L(v)$, which is the first defining property

of a congruence relation. We get the second one the following way: Let $u \sim v$ and $u.n \neq \perp$ then $X = [u.n]_\sim = [u.n]_{R(\sim,X,n)}$ is defined. We get:

$$\begin{aligned} u \sim v &\implies (u,v) \in R(\sim, X, n) \\ &\implies (u.n \in X \Leftrightarrow v.n \in X) \\ &\implies (u.n \in [u.n]_\sim \Leftrightarrow v.n \in [u.n]_\sim) \\ &\implies u.n \sim v.n \end{aligned}$$

Furthermore we get $v.n \neq \perp$, since otherwise $\perp = v.n \in X$. But $\perp \notin V$ as it isn't a node.
Hence $\sim$ is a congruence relation, and by corollary 1.24 we get $\sim = \sim_{\mathcal{T}}$. □

Corollary 2.5. Let

$$\sim_0 \succ \sim_1 \succ \ldots \succ \sim_n$$

be a chain of distinction relations with $\sim_L \supseteq \sim_0$. This chain is finite, and if there is no $\sim_{n+1}$ such that $\sim_n \succ \sim_{n+1}$, then $\sim_n = \sim_{\mathcal{T}}$.

2.2 The Algorithm

A first algorithm for the computation of $\sim_{\mathcal{T}}$ for any given graph G (which is equivalent to finding the minimization of G) follows directly from corollary 2.5 above. But the naive implementation takes $O(n^2)$ time, where $n = |V|$ is the number of nodes of G.

The most easy-to-find starting relation is $\sim_L$, the distinction relation that distinguishes nodes by their labels. Starting the algorithm with $\sim_{LA}$ (to include information on the arity of the nodes) is a simple improvement. In fact, every distinction relation finer than $\sim_L$ can be used as starting relation, so any a priori information can be taken into account.

An even asymptotically improved runtime is reached by an algorithm discovered by Hopcroft in 1971 [2, 3]. It works on the same principle as the naive version, but uses as few and small partitions as possible (only those in the *agenda*) for the refinement process.

Note that we will also use the term *block* for the sets of nodes we will work with, because the relation defining these sets change and so the term *equivalence class* might lead to confusions.

Algorithm 2.6. Let $a = arity(V)$ be the maximal arity of a node in V. Compute a chain $\sim_0 \supseteq \sim_1 \supseteq \ldots$ as follows:

i=0: $\sim_0 = \sim_L$
$agenda_0 = (V/_{\sim_0}) \times \{0, \ldots, a-1\}$
(all pairs of equivalence classes and edge labels are in the agenda)

i → i+1: **if** $agenda_i = \emptyset$ **then return** $\sim_i$.
else There is a pair $(X_i, n_i) \in agenda_i$.
Define $\sim_{i+1} = R(\sim_i, X_i, n_i)$ and update the agenda.

The updating of the agenda is done as follows: Remove the pair (X_i, n_i) from the agenda. Whenever $Y_1, Y_2 \in V/\sim_{i+1}$ with $Y = Y_1 \neq Y_2$ such that $Y_1 \dot\cup Y_2 \in V/_{\sim_i}$, assume $|Y_1| \leq |Y_2|$ and construct the new $agenda_{i+1}$ as follows. For all $0 \leq n < a$ do:

- If $(Y, n) \notin agenda_i$, then add (Y_1, n) to the agenda.
- If $(Y, n) \in agenda_i$, then remove it, and add (Y_1, n) and (Y_2, n).

The algorithm works exactly like the naive version, except that we do not refine using all possible classes. So we only have to show that this suffices, and, since there is a new break condition, that the algorithm terminates after finitely many steps. We will do so after some lemmata needed for the proof.

Definition 2.7. Let $\sim$ be a congruence relation. We call a pair $(X, n) \in V/_\sim \times \mathbb{N}$ *safe with respect to* $\sim$, iff $R(\sim, X, n) = \sim$.

Lemma 2.8. The pair (X, n) is safe with respect to $R(\sim, X, n)$.

Proof. We have to show $R(\sim, X, n) = R(R(\sim, X, n), X, n)$, but this is clear by the idempotence of the refinement. $\square$

Lemma 2.9. Let $(X, n) \in V/_\sim \times \mathbb{N}$ be safe with respect to $\sim$, and $(Y, m) \in V/_\sim \times \mathbb{N}$. Then (X, n) is also safe with respect to $R(\sim, Y, m)$.

Proof. Using "commutativity" and then safeness, we get

$$R(R(\sim, Y, m), X, n) = R(R(\sim, X, n), Y, m) = R(\sim, Y, m).$$

$\square$

Now we can prove the correctness of the algorithm:

Proof. We prove that the following invariant holds at each step of the algorithm:

$$\begin{aligned} &\forall (Y, n) \notin \ agenda_i : \\ &\qquad \exists (Z_1, n), \ldots, (Z_m, n) \in \ agenda_i : \\ &\qquad\qquad (Y \cup Z_1 \cup \ldots \cup Z_m, n) \text{ is safe for } \sim_i \end{aligned}$$

The number m of pairs needed to gain safeness may be 0, in which case (Y, n) itself is safe for $\sim_i$. When the algorithm stops, the agenda is empty, so $m = 0$ for all equivalence classes. This means that all classes are save for the final distinction relation, and so we have reached $\sim_{\mathcal{T}}$ by lemma 2.4.

For simplicity, we will write $Z_{Y,n} := \{Z_1, \ldots, Z_m\}$ within this proof and call it the safety set of (Y, n).

Induction start: At the beginning all pairs $(X, n) \in V/_\sim \times \mathbb{N}$ with $0 \leq n < arity(V)$ are in the agenda. As refinement by an edge not occuring in the graph changes nothing, we are done.

Induction step: Let $\sim = \sim_i$ be the distinction relation at any further step of the algorithm, and let $(X, n) \in agenda$ be the pair with respect to which we refine. We need to show that the invariant holds for the (i+1)th step with $\sim_{i+1} = R(\sim, X, n)$.

In the proof we need the following construction used later:

Construction 1: Let $Z_{Y,n}$ be the safety set of (Y, n) before the refinement on (X, n). We construct a new $Z'_{Y,n}$ by modifing $Z_{Y,n}$ in such a way that we replace each $Y' \in Z_{Y,n}$ that was split in $Y' = Y'_1 \dot\cup Y'_2$ by both Y'_1 and Y'_2. Furthermore we remove X, as (X, n) is no more on the agenda after the refinement. By the definition of the algorithm it is sure that (Y'_1, n) and (Y'_2, n) are on the agenda after the refinement, therefore $Z'_{Y,n}$ can be also found there.

Now lets continue the proof. Let be $(Y_1, n) \notin agenda$:

- If (Y_1, n) is the pair (X, n) we refined on, the invariant is fulfilled as (X, n) is safe for $R(\sim, X, n)$.
- If the equivalence class X we refined on was split itself such that $X = Y_1 \dot\cup Y_2$ and $|Y_1| \leq |Y_2|$, then we can take $Z_{Y_2} = \{Y_1\}$, as (Y_1, n) is now in the agenda.
- If (Y_1, n) was not split by the last refinement, then it fulfills the invariant with the set Z'_{Y_1} of construction 1.
- If (Y_1, n) is a part of a splitted equivalence class such that $Y = Y_1 \dot\cup Y_2$, then (Y_2, n) was put on the agenda. Construction 1 gives us a set $Z'_{Y,n}$ that is on the new agenda. Let $Z_{Y_1,n} = \{Y_2\} \cup Z'_{Y,n}$ and $Z = Y_1 \cup \bigcup\{Y'|Y' \in Z_{Y_1,n}\} = Y \cup \bigcup\{Y'|Y' \in Z_{Y,n}\}$. As $Z_{Y,n}$ is the safety set of (Y, n) we have, that (Z, n) is safe for $\sim$ and also for $R(\sim, X, n)$. But then $Z_{Y_1,n}$ is a valid safety set for (Y_1, n) and we get that the invariant holds for (Y_1, n).

$\square$

2.3 Computing the Refinement

Up to now we have no algorithm for computing the refinement, but this is essential to analyse the runtime of the algorithm. Therefore this section will give an optimized algorithm for computing $R(\sim, X, n)$.

First we need a representation for the distinction relations $\sim$. We represent the equivalence classes (blocks) $V/_\sim$ of $\sim$ using doubly linked lists. Furthermore into each node we save a pointer to the equivalence class it corresponds to and each node has backpointers to its parent nodes accessed by the function *parents* defined as $parents(u, n) = \{v \in V | v.n = u\}$.
The refinement by $(X, n) \in V/_\sim \times \mathbb{N}$ splits equivalence classes B of $\sim$ into $B \cap \{u | u.n \in X\}$ and $B \setminus B \cap \{u | u.n \in X\}$ if none of both are empty.

Proof.

$$
\begin{aligned}
[u]_{(R(\sim,X,n))} &= \{v | (u, v) \in \sim \wedge (u.n \in X \iff v.n \in X)\} \\
&= \{v | (u, v) \in \sim\} \cap \{v | u.n \in X \iff v.n \in X\} \\
&= \begin{cases} [\mathrm{u}]_\sim \cap \{v | v.n \in X\} & \text{if } u.n \in X \\ [\mathrm{u}]_\sim \cap \{v | v.n \notin X\} & \text{if } u.n \notin X \end{cases} \\
&= \begin{cases} [\mathrm{u}]_\sim \cap \{v | v.n \in X\} & \text{if } u.n \in X \\ [\mathrm{u}]_\sim \setminus [\mathrm{u}]_\sim \cap \{v | v.n \in X\} & \text{if } u.n \notin X \end{cases}
\end{aligned}
$$

$\square$

The following algorithm computes this efficiently:

Algorithm 2.10. Refinement

fun refine (X,n)
  $\forall u \in X : \forall v \in parents(u, n)$:
    select the node v

After this you can split each equivalence class in the selected and unselected part if none of both are empty. This can be done in $O(1)$, if the nodes within blocks are stored in a list, and if selected nodes within each block are directly moved into a second list.

2.4 Runtime Analysis

Let $k = arity(V)$. We analyse the number of times the preimage of a fixed node u is computed. This is only the case if we refine on a pair (X, n) with $u \in X$. After this the pair is removed from the agenda. If X was split and X' is the smaller part, we add maximally k pairs (X', n) to the agenda, one for each label occuring in the graph. As the size of the smaller part is less than half of the size of the original block, the preimage of an edge $n \in \mathbb{N}$ of a node can be maximally computed $log_2(|V|)$ times. Let $E_{u,n}$ be the number of incoming edges of u labeled with n. Computing the preimage of a node u and egde n costs $O(E_{u,n})$ and is done maximally $log_2(|V|)$ times. The refinement must be done for each edge and node, therefore we get the following work computing the preimage (visiting nodes is not counted):

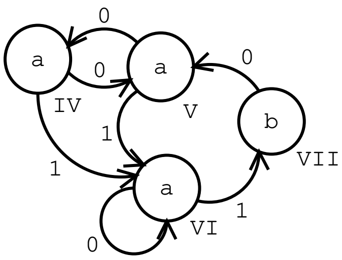
$$\sum_{u \in V} \sum_{n \in \mathbb{N}} O(E_{u,n}) \cdot log_2|V| = O(|E| \cdot log_2|V|)$$

We visit maximally the following number of nodes:

$$\sum_{n=0}^{k} |V| \cdot log_2|V| = k \cdot |V| \cdot \log_2 |V|$$

Therefore the algorithm is in $O((|E|+k\cdot|V|)\cdot\log|V|)$. But we have $|E| \leq k\cdot|V|$, which yields $O(k \cdot |V| \cdot \log|V|)$.

Example 2.11. Now we will minimize the following graph using the partitioning algorithm presented above.



The initial partition using $\sim_L$, that separates only by the label, is $blocks_0 = [\{VII\}, \{IV, V, VI\}]$. Now we construct the agenda and get:

$$\begin{aligned} blocks_0 &= [\{VII\}_A, \{IV, V, VI\}_B] \\ agenda_0 &= [(B, 1), (B, 0), (A, 1), (A, 0)] \end{aligned}$$

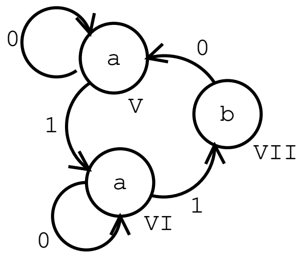
We first refine on the pair $(A, 0)$, which changes nothing, as there is no incoming edge labeled with 0 to the block A. But refinement by $(A, 1)$ splits the block B into $B' = \{VI\}$ and $B'' = \{IV, V\}$. As both $(B, 1)$ and $(B, 0)$ are in the agenda we remove them and add $(B', 1), (B'', 1), (B', 0)$ and $(B'', 0)$.

$$\begin{aligned} blocks_2 &= [\{VII\}_A, \{IV, V\}_{B'}, \{VI\}_{B''}] \\ agenda_2 &= [(B'', 0), (B', 0), (B'', 1), (B', 1)] \end{aligned}$$

Refinements on $(B', 1), (B'', 1), (B', 0)$ and $(B'', 0)$ have no effect on the blocks. Therefore we finally get:

$$\begin{aligned} blocks_6 &= [\{VII\}_A, \{IV, V\}_{B'}, \{VI\}_{B''}] \\ agenda_6 &= [] \end{aligned}$$

The agenda is empty. So we can assure that no refinenent on a pair (X, n) changes anything on the blocks (convince yourself of it). We gain the minimal graph if we replace the node IV by the equivalent node V and get:



2.5 Unique Order on the Trees of a Minimal Graph

Here we present a further application of the partitioning algorithm. The algorithm can be used to compute a unique order on the trees of a minimal graph in $O(|V| \cdot log|V|)$. The order is not the canonical order on the trees, but it is one, that can be computed very fast. Be careful that the order depends on the graph it works on, which means that for two trees t_1,t_2 it could be $t_1 \leq t_2$ using graph G_1 as representation and $t_2 \leq t_1$ using another graph G_2 as representation. For our application of the algorithm, this property is acceptable.

Algorithm 2.12. Let $V/_{\sim_L}$ be the initial partition of the algorithm, $A, B \in V/_{\sim_L}$, and let $L(X)$ return the label of the nodes in the block X, which are all the same. Then the following defines a total order on $V/_{\sim_L}$:

$$A \leq_I B \iff L(A) \leq L(B)$$

We order the initial blocks by this order. Each such order $\leq$ on blocks induces a partial order on the nodes:

$$u \leq v \iff A \leq_I B \quad (\text{where } u \in A \text{ and } v \in B)$$

Furthermore we extend $\leq_I$ to pairs occuring in the agenda as:

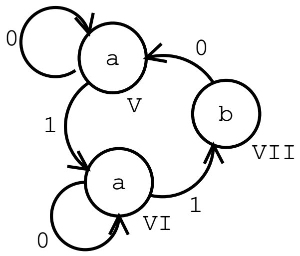
$$(A, n) \leq'_I (B, m) \iff A <_I B \vee (L(A) = L(B) \wedge n \leq m)$$

We use only set theoretic operations to access the blocks. We call the order of the blocks at a specific time in the algorithm $\leq_C$. This order is represented as the order of the blocks in a list. This means that a block occuring on the left of another block in the list is the smaller one. Each time we split a block B by a pair (A, n) into B' and B'' such that $\forall u \in B' : u.n \in A$ we update the order $\leq_C$ putting the block B' directly to the left of B'' at the place B was. More exactly if we had $A \leq_C B \leq_C C$ then we get $A \leq'_C B' \leq'_C B'' \leq'_C C$.
The agenda is implemented as a queue with the special property that old pairs can be deleted if neccessary. Let $a = arity(V)$ be the maximal arity occuring in the graph. We get the following algorithm (*pop* returns and deletes the first element of a queue):

fun compute_order(G) =
  $Blocks := V/\sim_L$ sorted by $\leq_I$
  $agenda = (V/_{\sim_0}) \times \{0, \ldots, a\}$ (initially sorted by $\leq'_I$)
  while $(X, n) = pop(agenda)$ **do**
    Refine the blocks by (X, n) and update the agenda.
  return $Blocks$

Proof. We have to show that the algorithm computes the same order on the trees of two equivalent minimal graphs G_1 and G_2. As G_1 and G_2 are minimal, each node corresponds to exactly one tree. Therefore we can take the set $\mathcal{T}(V_1) = \mathcal{T}(V_2)$ as the set of nodes, such that $\mathcal{T}(u) = u$. But then both graphs are identical. And since blocks are accessed as sets, there is no difference in the state of the algorithm after sorting the initial blocks. As the algorithm is deterministic, it always computes the same order on the nodes, and hence on the trees of the graph. □

Example 2.13. Now we compute the order of the nodes of the minimal graph gained by the last example.



The initial partition is $blocks_0 = [\{V, VI\}, \{VII\}]$ using $\sim_L$ and sorting by the label. Now we construct the agenda and get:

$$\begin{array}{rcl} blocks_0 & = & [\{V, VI\}_A, \{VII\}_B] \\ agenda_0 & = & [(B,1), (B,0), (A,1), (A,0)] \end{array}$$

We first refine on the pair $(A, 0)$, which changes nothing. Refinement by $(A, 1)$ splits the block A into $A' = \{V\}$ and $A'' = \{VI\}$. As $V.1 = VI \in A$, we put the block A' to the left of A''. We get:

$$\begin{array}{rcl} blocks_2 & = & [\{V\}_{A'}, \{VI\}_{A''}, \{VII\}_B] \\ agenda_2 & = & [(A',1), (A',0), (B,1), (B,0)] \end{array}$$

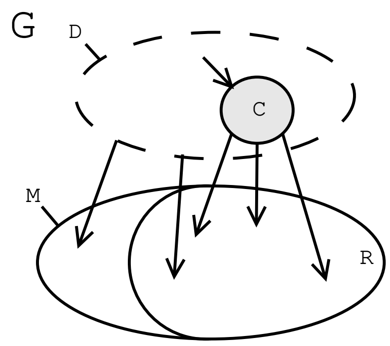
As we only have single node blocks, we are done. The node V is the minimal one in this order.

3 Incremental Minimization of Graphs

The goal of this chapter is to show a method to minimize a graph incrementally. This means to divide the graph into its strongly connected components and then minimize them independently if possible.

3.1 Adding a Component

Assume we have given a graph G and sets of nodes $C \subset D, R \subset M$ as in the figure below. Furthermore $D \dot{\cup} M = V$ and C is a strongly connected component of D at bottom level. Bottom level means that there is no edge (u, v) with $u \in C$ and $v \in D \setminus C$ in the graph. R is the set of nodes of M that are reachable from C. The set $C \cup R$ is closed and M must be minimal and closed. The set D is the rest of the graph, to work on any more and the minimal set M we already worked on.



Our task is to expand the minimal closed set M by the component C, such that the resulting set M' is again minimal and closed. This means to compute the relation $\sim_{\mathcal{T}}$ and then modifiy the graph by means of a minimization of $C \cup M$. By lemma 3.2 only the following cases are possible:

1. $\mathcal{T}(C) \subset \mathcal{T}(R)$
2. $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$
3. $\mathcal{T}(C) \not\subset \mathcal{T}(R)$ and $\mathcal{T}(C) \not\subset \mathcal{T}(M \setminus R)$

To handle the different cases, we use different algorithms explained in the following sections.

The algorithms for the first two cases have to check whether $\mathcal{T}(C) \subset \mathcal{T}(R)$ or $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$, and if so to compute $\sim_{\mathcal{T}}$, as we will want to do a minimization later. Therefore it suffices for the algorithms to return for each node of C the equivalent one in M. If we have this information, we can minimize efficiently if we redirect each incoming edge of a node in C to the equivalent node in M and then delete all nodes of C from the graph (and of course from the set D).

In the third case we can minimize the component C independently of M in $O(n \cdot log\ n)$ using lemma 3.1. We call the minimized component C'. As in this case we have $\mathcal{T}(C) \cap \mathcal{T}(M) = \emptyset$, we can expand M by the minimal component C' and get $M' = C' \dot{\cup} M$. This set M' is minimal and closed again. Furthermore we define $D' = D \setminus C'$. As an enhancement we also assign a unique id to each node of C'. Since we will need some hashing information for the component C' later, we compute the minimal node of C' and add it to a hashing table. This provides a way to find the component C' later.

We call the operation that increases the minimal set M by C' $(M, D) \xrightarrow{C} (M', D')$.

The following lemma is basic for the algorithms.

Lemma 3.1. If $\mathcal{T}(C) \cap \mathcal{T}(R) = \emptyset$, then $C \cup R$ can be minimized in $O(|C| \cdot log(|C|)$ time. The set C does *not* have to be strongly connected.

Proof. (algorithmic) We use the partitioning algorithm to minimize $C \cup R$ with the following initial congruence:

$$\begin{aligned} u \sim v \iff & \quad (u \in C \wedge v \in C \wedge L(u) = L(v)) \\ \vee & \quad (u \in R \wedge v \in R \wedge u = v) \end{aligned}$$

The relation $\sim$ is an equivalence relation. Furthermore it is a distinction relation, because if $T(u) = T(v)$ and $u, v \in C$, then we have $L(u) = L(v)$ and hence $u \sim v$. If $T(u) = T(v)$ and $u, v \in R$, then we have $u = v$ as R is minimal and once more $u \sim v$. Otherwise the nodes cannot be equivalent, because $\mathcal{T}(C) \cap \mathcal{T}(R) = \emptyset$ and therefore $u \not\sim v$.
Since we want to use $\sim$ as the initial partition, we have to show $\sim \subseteq \sim_L$, but this is clear, as $u \sim v \implies L(u) = L(v) \vee u = v$, which implies $L(u) = L(v)$.

Note that the distinction relation puts each node of R into a singleton block (i.e. a block with only one node).

Now we can start the partitioning algorithm on $C \cup R$ and realize the following:

Let be $K = \{v \in R \mid \exists u \in C, \exists n \in \mathbb{N} : v = u.n\}$ (K is the set of nodes of R which have parent nodes in C). It suffices to start the algorithm on the nodes $C \cup K$ and ignore the rest of R. This is possible because of the following:

Let $M = R \setminus K$ be the remaining nodes (those that we ignore), and let $w \in M$. Then the parent nodes of w are all in R, as otherwise there would be a node $s \in C$ and $n \in \mathbb{N}$ such that $s.n = w \in R$, and we would get the contradiction $w \in K$. Therefore the nodes in M can only split the singleton blocks of the nodes in R, but these cannot be split. It follows that the algorithm can be implemented in $O(|C \cup K| \cdot log|C \cup K|)$ time, running on $C \cup K$. The size of K is in $O(|E_C|)$ if $|E_C|$ is the number of edges of C. $\square$

Lemma 3.2. Let G be a graph and $C, D \subseteq V$ be two sets of nodes of G such that C is strongly connected and D is closed. If there are nodes $u \in C$ and $v \in D$ such that $\mathcal{T}(u) = \mathcal{T}(v)$, then there is an equivalent node in D for every node of C.

Proof. Let w be any node of C. As C is strongly connected, there exists a path q such that $u.q = w$. We directly see that $\mathcal{T}(v.q) = \mathcal{T}(u.q) = \mathcal{T}(w)$. And $v.q$ lies in D, as D is closed. $\square$

3.2 $\mathcal{T}(C) \subset \mathcal{T}(R)$

The first case to check is whether $\mathcal{T}(C) \subset \mathcal{T}(R)$, which can be done in two different ways. The following lemma shows a property that can be used to optimize the algorithms.

Lemma 3.3. If $\mathcal{T}(C) \subset \mathcal{T}(M)$, then there exists a strongly connected component C_M in M such that $\mathcal{T}(C) \subset \mathcal{T}(C_M)$. Furthermore if M is reachable from C, then C is directly connected to C_M (i.e. there are nodes $u \in C$ and $v \in M$, and $n \in \mathbb{N}$ with $u.n = v$).

Proof. As $\mathcal{T}(C) \cap \mathcal{T}(M) \neq \emptyset$, we find a tree $t \in \mathcal{T}(C) \cap \mathcal{T}(M)$, and two nodes $u \in C$, $v \in M$, such that $t = \mathcal{T}(u) = \mathcal{T}(v)$. We set $C_M = [v]_{\sim_{sc}}$. Let w be a node of C and w' an equivalent one of M, which exists by lemma 3.2. Then we have:

$$\begin{array}{rcl} u \sim_{sc} w & \stackrel{1.29}{\Longrightarrow} & \mathcal{T}(u) \sim_{asc} \mathcal{T}(w) \\ & \Longrightarrow & \mathcal{T}(v) \sim_{asc} \mathcal{T}(w) \\ & \stackrel{1.29 \text{ on M}}{\Longrightarrow} & v \sim_{sc} w‘ \end{array}$$

This implies $\mathcal{T}(C) \subset \mathcal{T}(C_M)$.

Let M be reachable from C, and let u, v, C, C_M be defined as above. Assume that C is not connected to C_M. Then we find a node $w \in M \setminus C_M$ such that there are paths $p, q \in \mathbb{N}^*$ with $u.p = w$ and $w.q = v$. But now we have $\mathcal{T}(w) = \mathcal{T}(u.p) = \mathcal{T}(u)|p$ and $\mathcal{T}(u) = \mathcal{T}(v) = \mathcal{T}(w.q) = \mathcal{T}(w)|q$ and therefore $\mathcal{T}(v) \sim_{asc} \mathcal{T}(w)$. As M is minimal, we get $v \sim_{sc} w$ by proposition 1.29, but $w \notin C_M$. Contradiction. □

The lemma expresses that equivalent nodes of C and R can *only* be found in the adjoining strongly connected components of C. This fact is used in the following two algorithms. Assume the situation of figure 3.1 and let $C_1, \ldots, C_n$ be the adjoining strongly connected components of C.

Algorithm 3.4. (first version) Take *one* node $u \in C$ and do a naive equality test with each node of the $C_1, \ldots, C_n$ using the following algorithm (let $f \in C \to C_1 \cup \cdots \cup C_n$ be empty at the beginning):

fun equal(u,v)
begin
  if $f(u) = v$ **then return** true
  if $u \in C_1 \cup \cdots \cup C_n$ **then return** u = v

  Let $u_1, \ldots, u_n$ be the children of u
  and $v_1, \ldots, v_m$ be the children of v.

  set $f(u) = v$

  if $n \neq m \vee L(u) \neq L(v)$ **then return** false
  else return $equal(u_1, v_1) \wedge \cdots \wedge equal(u_n, v_n)$
end

This algorithm has a runtime of $O((|C| + |E|) \cdot \sum_i |C_i|)$.

Proof. The correctness of the algorithm is clear, as the node comparision in the second if-clause is allowed, since $C_1 \cup \cdots \cup C_n$ is minimal. □

Algorithm 3.5. (second version)

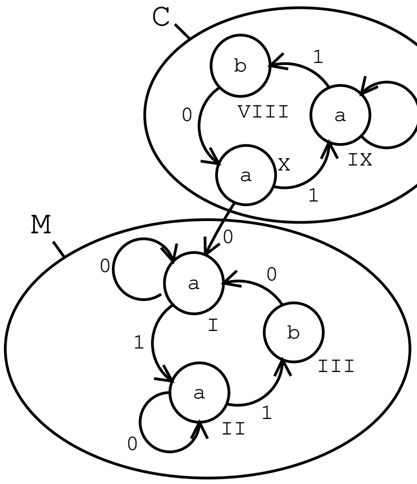
2. Apply the algorithm of lemma 3.1 setting $C_{\text{lemma 3.1}} = C \cup C_1 \cup \cdots \cup C_n$ and the minimal set $R_{\text{lemma 3.1}}$ to $R \setminus (C \cup C_1 \cup \cdots \cup C_n)$.

3. Take the block $B = [u]_\sim$ of a node $u \in C$. Then $\mathcal{T}(C) \subset \mathcal{T}(R)$ iff there is a node of $C_1 \cup \cdots \cup C_n$ in B. Furthermore for each node of C we find an equivalent one in R this way.

Let $X = C \cup C_1 \cup \cdots \cup C_n$. Then the algorithm works in $O(|X| \cdot log(|X|))$.

Omitting constants, the first version of the algorithm is in fact the better choice, if $|C| \leq log_2(\sum_i |C_i|)$. So we use the first version if $|C| \leq log_2(\sum_i |C_i|)$ and the second one otherwise.

Example 3.6. We will apply the algorithm to the following example graph. As the set C is very large, we use the second version.



The initial partition is $blocks_0 = [\{I, II, IX, X\}_A, \{III, VIII\}_B]$ using $\sim$ of lemma 3.1. The lemma is used setting $C_{\text{lemma 3.1}} = C \cup A$ if A are the nodes of the adjoining components of C. But there is only one such component, so we have $A = \{I, II, III\}$. Furthermore $R_{\text{lemma 3.1}}$ is the rest of the set M and therefore empty. Now we refine on the pair $(A, 1)$ and get $blocks_1 = [\{II, IX\}_{A'}, \{I, X\}_{A''}, \{III, VIII\}_B]$. Convince yourself that there is no further pair (X, n) that refines the partition. We investigate block A'' corresponding to node X and find the node $I \in M$ in it. Therefore $\mathcal{T}(C) \subset \mathcal{T}(M)$, and more specifically $\mathcal{T}(I) = \mathcal{T}(X), \mathcal{T}(II) = \mathcal{T}(IX)$ and $\mathcal{T}(III) = \mathcal{T}(VIII)$.

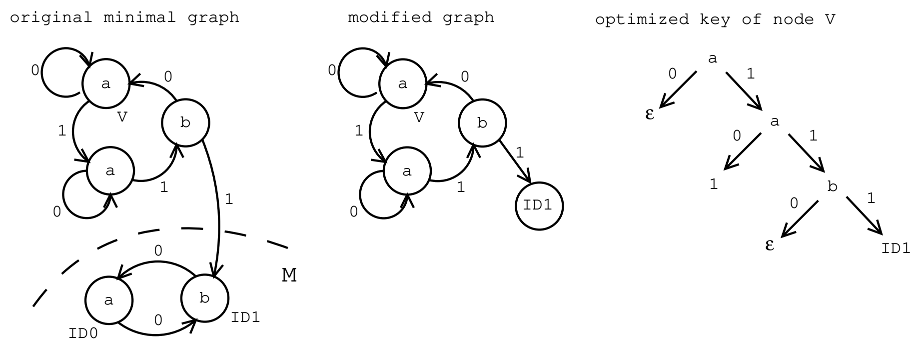
3.3 $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$

We assume not being in the first case, or in other words $\mathcal{T}(C) \not\subset \mathcal{T}(R)$. Then we can compute $\sim_\mathcal{T}$ on C as described in lemma 3.1 with time complexity $O(|C| \cdot log(|C|)$. We use this information to minimize the component C of the graph and obtain a minimal component C'. Now we can test whether $\mathcal{T}(C) = \mathcal{T}(C') \subset \mathcal{T}(M \setminus R)$. We do this by taking the minimal node of C' computed by algorithm 2.12 and then using hashing to find an equivalent one in M. This works, if we save hashing information for each (abstract) component of M, and as $C \cup R$ and M are both minimal and closed. If this equivalent node exists, then we obviously have the case $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$. Up to now we only have the equivalent node for exactly one node of C, but it is easy to gain the other ones using the idea of the proof of lemma 3.2.

3.4 Optimized Hash Key

As we computed id's for each node in M, we can use a key for the hashing algorithm, which is more efficient than the one defined in the section 1.6. The size of this new key is linear in the size of C' and not linear in the size of the graph reachable from C'.
The idea is to replace the nodes at the frontier of the component with their id. More precise, we replace each edge (u, v) such that $u \in C'$ and $v \in M$ with (u, id_v). As the ids are unique, it is trivial to see that this defines an injection. The new key of a node $u \in C'$ is defined as the key of the corresponding node in the modified component.

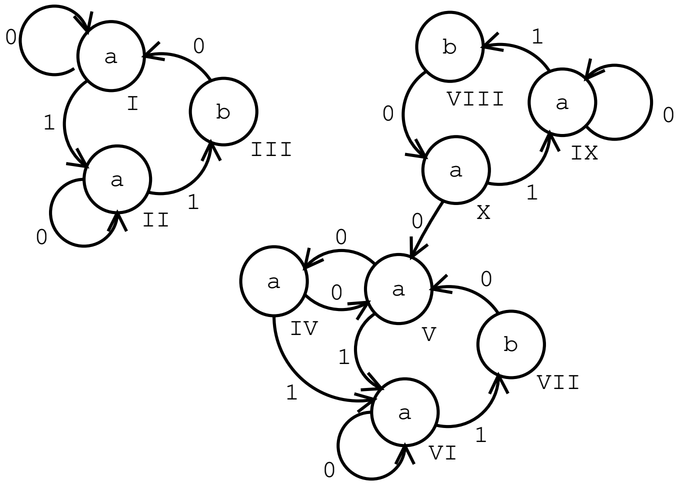
Example 3.7. The following example illustrates the new key.



3.5 Incremental Graph Minimization Algorithm

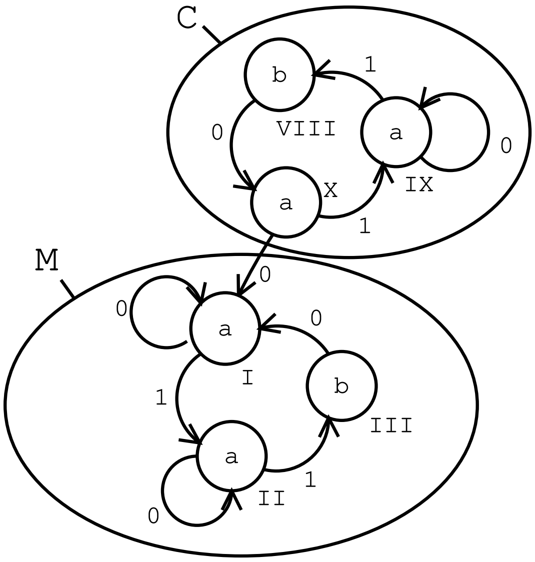
Our goal is to minimize a graph G. Let $M_0 = \emptyset$ at the beginning and $D_0 = V$. We iteratively compute $(M_i, D_i) \xrightarrow{C_i} (M_{i+1}, D_{i+1})$ for a remaining strongly connected component C_i of D_i at bottom level. If D_i is empty, then we are done and M_i contains the nodes of a minimal graph equivalent to G.

Example 3.8. To illustrate the concepts, we present an example applying the incremental algorithm to following not connected graph with 10 nodes.

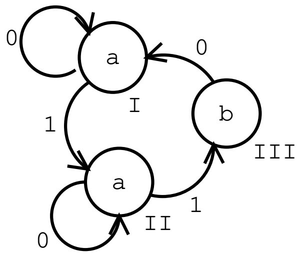


The minimal closed set M is empty at the beginning. First we take the strongly connected component $C_1 = \{I, II, III\}$ at the top left position. From example 2.11 we know that this component is already minimal. There are no adjoining components, so we can directly go to step 2. By example 2.13 we know that the minimal node of this component is the node I, so we have to compute $key(I)$ (done in example 1.36) and search this key in the hash table. since the table is empty at the beginning, we find no node. Therefore we proceed to step 3, where we add $(key(I), I)$ to the hash table and set $M = \{I, II, III\}$.
The next component we can take is $C_2 = \{IV, V, VI, VII\}$. As there are once more no adjoining components, we can minimize C_2 (done in example 2.11) and get $C_2' = \{V, VI, VII\}$. The minimal node of C_2' is V computed in example 2.13. As $key(I) = key(IV)$, we find the previously added node I in the hash table. We replace all nodes of the component C_2' by the equivalent ones in M.

The picture now looks that way:



There is only one component C left. Step 1 of the algorithm yields $\mathcal{T}(C) \subset \mathcal{T}(R) = \mathcal{T}(M)$ computed in example 3.6. We replace the nodes of C by the equivalent ones in M and get the following minimal graph as result:



3.6 Runtime Analysis

For each component C of G, we have to do compute $(M, D) \xrightarrow{C} (M', D')$. Let E_C be the number of edges within C, and let $C_1, \ldots, C_n$ be the adjoining components of C. Then we have the following worst case complexity for the single computation steps:

1. Find C: $O(|C| + E_C)$
2. $\mathcal{T}(C) \subset \mathcal{T}(R)$: $O((|C| + E_C) \cdot \sum_i |C_i|)$
3. $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$: $O(|C| \cdot log(|C|))$

This yields a total runtime of at most $O((|C| + E_C) \cdot \sum_i |C_i|)$ for each component. The whole algorithm is in $O((|V| + |E|)^2)$.

3.7 Building a Regular Tree Database

The algorithms above can be used to compute a database for regular trees. Therefore we only have to save all information of the minimal set M that is expanded during the computation. This information forms the tree database. All information means the edges between the nodes in M, the id's of the nodes, as well as the hashing table that is used.

With these simple changes we have a powerful tree database which assigns an id to each tree that is added. The input trees are given as a node in a finite graph. The resulting id of the tree is the id of the equivalent node in the minimized graph in the database after the computation.

3.7.1 Equality Testing

It follows that equality of trees (that are already represented as an id of the database) can be tested in constant time, if we compare the id's. But observe that the first representation change from the graph representation for the tree to the database representation costs quadratic time.

3.7.2 Subtree Testing

The problem is to decide whether $t_1 \in Sub(t_2)$.

The naive algorithm is in $O(n\ log\ n)$ in the size of the representations of t_1 and t_2.

Working with the database, we can do a little bit better. Assume we already have given the id's id_1 and id_2 of the trees t_1 and t_2. We only have to test, whether the id id_1 is the id of a subtree of t_2. This can be done linear in the size of $Sub(t_2)$.

As you can see, some of the operations on the tree database are very fast, for example equality testing. A problem is that you need to compute the database representation first, which takes quadratic time. But for applications that very often compare regular trees it is more efficient to use the database as representation all over the time. Compilers for example can use the database to represent recursive types, as type comparision is done very often.

4 Comparison with Mauborgne's Work

Apart from

- the developed theoretical background and
- the possibility to define hashing on trees,

we also did some enhancements concerning the concrete algorithms:

- We divided the algorithm into three clearly defined, independent parts ($\mathcal{T}(C) \subset \mathcal{T}(R)$, $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$ and expansion of M).

- We refined the algorithm to test $\mathcal{T}(C) \subset \mathcal{T}(R)$, which Mauborgne calls "ShareWithDone". Our algorithm works much better than the original model, if the component C is very large with respect to R. A disadvantage is, that we have to do much more work if C is very small. But in this case the naive algorithm is the fastest anyway, because its overhead is only minimal.
- The last optimization is that, if we test $\mathcal{T}(C) \subset \mathcal{T}(M \setminus R)$, we compute only the hash key of the minimal node of C. As the computation of the minimal node is in $O(|C|log|C|)$, we have the same complexity here. Mauborgne computes the key of *each* node of C and gets the worse runtime of $O(|C|^2)$.

References

[1] Laurent Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000.

[2] John E. Hopcroft. An $n \cdot \log n$ algorithm for minimizing states in a finite automaton. *Theory Of Machines And Computations, Academic Press*, pages 189–196, 1971.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures And Algorithms.* Computer Science and Information Processing. Addison-Wesley, 1982.

[4] Laurent Mauborgne. *Representation of Sets of Trees for Abstract Interpretation.* PhD thesis, Ecole Polytechnique, 1999.

[5] Kurt Mehlhorn and Stefan Näher. *Leda: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[6] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \cdot \log_2 |V|)$. *Theoretical Computer Science*, 19:85–98, 1982.

[7] Gert Smolka. Higher order recursive types. Notes, 2000.