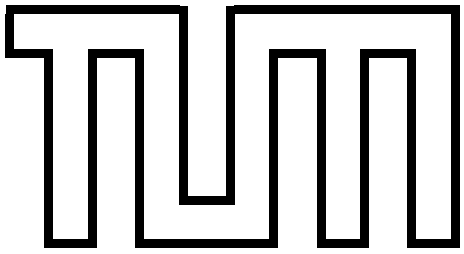


Technische Universität München

Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen

Diploma Thesis in Informatics

Automata-based IP Packet Classification

Benjamin Hummel

Supervisor:	Prof. Dr. Ernst W. Mayr
Advisor:	Dr. Sven Kosub
Date of delivery:	15^{th} July 2006

ACM CCS: F.1.1 Models of Computation – Automata
C.2.6 Internetworking – Routers
General Terms: Algorithms, Theory

AMS MSC: 68Q45 Formal languages and automata
68M12 Network protocols

Declaration: "I hereby declare that this thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration unless stated otherwise."

Munich, Benjamin Hummel

Contents

List of Figures

List of Algorithms

Preface

In this thesis we discuss algorithms and data structures used for classifying IP packets. More specifically we will deal with the longest prefix matching problem which is used in many Internet applications, such as routing, packet filtering, content delivery networks, or deciding quality of service. Contrary to advanced approaches proposed in the literature which are often based on ad-hoc heuristics motivated by observations of practical instances, we base our studies on the fundamentals of language theory, namely finite automata. Therefore several classes of automata useful for representing forwarding tables are studied and minimization algorithms for these are presented.

In Chapter 1 we give a general overview on the IP protocol as well as IP addressing and routing. Based on this we introduce the longest prefix matching problem occurring with IP packet classification. After discussing applications and constraints for this problem we browse the approaches recommended in the literature.

Chapter 2 initiates the theoretical part of the thesis by revisiting notation and definitions used herein. Its purpose is to resolve cases where notation is used differently in the literature and to refresh the details of the definitions required later.

The next chapter introduces partition automata, a generalization of finite automata to decide partitions instead of languages. We rephrase and reprove some of the major results and algorithms from automata theory for these, including the Myhill-Nerode theorem and Hopcroft's $O(n \log n)$ minimization algorithm.

In Chapter 4 we repeat the same generalizations for cover automata which have been introduced recently for the representation of finite languages. Our main contribution besides this generalization is presenting the results of several papers in a larger context using a common notation which allows the simplification of some proofs.

We revisit the idea of specializing automata in Chapter 5 by defining automata for languages and partitions containing only words of the same fixed length, called lookup automata. Unfortunately we do not know an efficient minimization algorithm for lookup automata. Moreover we give evidence that the minimization problem for lookup automata might be computationally hard, so we look into minimization heuristics instead.

After all these theoretical considerations Chapter 6 shows how to solve the longest prefix matching problem using the results from the earlier chapters. The key structure used is the trie in conjunction with suitable transformations.

Finally Chapter 7 gives some practical results on real-world data. We compare these to existing solutions and give an outlook on possible future improvements.

Chapter 1

Introduction

Todays Internet and most of local networks rely on the Internet protocol suite for data transport. The protocols involved are usually organized into four layers as shown in Figure 1.1 (see, *e.g.*, [KR02]). Protocols from higher layers are implemented on top of the layers below in the stack.

The protocols in the link layer forward data within the local network and are tightly coupled to the underlying hardware while the application layer addresses specific user defined problems. Thus there is a large variety in protocols in both of these layers. On the other hand the number of protocols in the transport and network layer building the core of the Internet protocol suite is comparatively limited, as the tasks for these layers are rather focussed. The network layer offers unreliable transport across multiple networks while the transport layer adds reliable transport as well as flow and congestion control. Together they act as the glue between the link and application layer.

As a consequence all data transmitted in the Internet[1] has to pass the network layer and thus be enclosed in a packet of the IP protocol or its designated successor IPv6 [DH98]. The remaining protocols at this layer are not used for carrying explicit data, but are control protocols. This centrality makes the classification of IP packets a central part of many services in the Internet, the most prominent probably being routing but also packet

[1] When talking of the Internet herein, this usually includes other networks based on the Internet protocol suite as well.

	Example protocols
Application layer	HTTP, SMTP, FTP, DNS, IMAP, NFS, SIP
Transport layer	TCP, UDP, RTP
Network layer	IP, IPv6, ARP, RARP, ICMP
Link layer	Ethernet, IEEE 802.11 (wireless), FDDI, PPP

Figure 1.1: The Internet protocol stack

<table>
<tr><td>version</td><td>header length</td><td>type of service</td><td colspan="2">packet length</td></tr>
<tr><td colspan="3">identifier</td><td>flags</td><td>fragment offset</td></tr>
<tr><td colspan="2">time-to-live</td><td>upper layer protocol</td><td colspan="2">header checksum</td></tr>
<tr><td colspan="5">source IP address</td></tr>
<tr><td colspan="5">destination IP address</td></tr>
</table>

Figure 1.2: IP packet header

filtering, quality of service, or content delivery networks can be based upon it as discussed in Section 1.2.

IP packets are usually classified by their header. Although the payload (the transported data) could be included in a classification scheme, this would violate the encapsulation principle stating that the payload of a packet should be opaque to applications working on the network layer. The basic header of an IP packet is shown in Figure 1.2. Optional header fields are not included and discussed here. Some of the fields in the basic header are not suited for a classification, such as the checksum, others allow a simple categorization by thresholds, like the packet length or time-to-live fields. The entries we are most interested in and that are also present in the IPv6 header are the source and destination address. In order to give a useful classification scheme we have to understand IP addressing in more detail which is tackled in the following section.

1.1 IP addressing and routing

To aid our explanation of IP addressing we will give a short (and slightly simplified) description of routing in IP based networks. Assume machine A sends a packet to host B. If A and B are in the same local network this can be handled by the link layer. Otherwise A forwards the packet to a *router*. The router has the task to get the packet as close to B as possible by forwarding the packet to a router that is less distant from B. Once the packet reaches a router in the same network as B, the packet is delivered. All forwarding steps are handled by the link layer, so only routers directly connected (or contained in the same local network) can exchange packets.

One major problem for a router is how to decide where to send a packet next. To support this decision there are two data structures involved. One is the *forwarding table* that decides for a given destination IP address to which router (called *next hop*) to forward the packet. Due to the size of the Internet and frequent changes to the topology (links are added or break down) manual management of this table is not possible. So there is a *routing table* which keeps a restricted view on the topology of the entire network

and allows the forwarding table to be generated from this information. The routing table in turn is built and updated by exchanging messages between adjacent routers. We will not discuss the contents of a routing table or the messages exchanged as this depends on the routing protocol used, *e.g.*, OSPF or BGP (see [KR02]).

Of course the forwarding and routing table are not capable of storing routing information for every possible destination host as there are just too many machines connected to the Internet. But as we have seen above, it is sufficient to get an IP packet to the destination *network* and the delivery to the host is then handled by the link layer. So the forwarding and routing table only contain routing information for every (sub)network. This in turn means that the forwarding table must determine the network from the destination IP address as this is the only information available about the target of the packet. To understand this process we will look at IP addresses in more detail. Although the 32 bits of an IP address are written as four octets which suggests a byte-wise interpretation, for the following discussion IP addresses should be seen as a plain string of bits.

An IP address can be split into a network and a host part. In the initial design of the IP protocol the first few bits of an address determined the *class* of the address and thus the length of the network and host parts. This scheme turned out to be problematic as the classes did not fit well with the network sizes of organizations. So either classes that were to large (*i.e.*, supported too many hosts) for a network were assigned, speeding the exhaustion of the IP address space, or multiple smaller classes were used to cover a single network, leading to a faster increase of used network numbers and thus forwarding table size. To solve this problem of this so called *classfull* addressing a new scheme named *classless inter-domain routing* (CIDR) was introduced in [FLYV93] and is still used today. With CIDR the network part can have any size. As this size is no longer encoded in the IP address, it has to be determined by checking every known network id in the forwarding table to see whether it is a prefix of the given IP address.

In order to get an intuitive idea of IP address assignment and the contents of the forwarding table, we give a simplified example including two small networks. When writing down network numbers we will use the common convention to denote them by the IP address and the number of network bits separated by a slash. So `127.32.0.0/18` means that the top 18 bits (here `01111111001000000`) form the network id and the host `127.32.15.89` would be expected to be part of this network.

The scenario described here is depicted in Figure 1.3. There are two providers $P1$ and $P2$ which were assigned the IP ranges `1.1.0.0/16` respectively `7.5.0.0/16`. The routers $R1$ and $R2$ are operated by these two and they are connected by a third router $R3$ which might be located with a large Tier-1 ISP. $P1$ has two customers $C1$ and $C2$, to each of whom $P1$ delegates a continuous range of its IP addresses (a subnetwork). So the

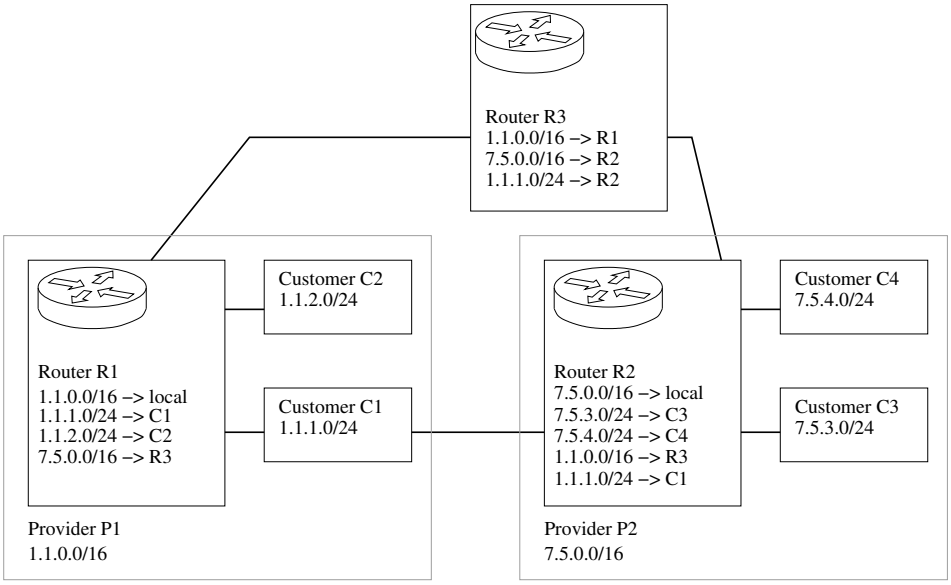


Figure 1.3: Example scenario

forwarding table of $R1$ sends all packets with a destination address whose prefix matches `1.1.1.0/24` to $C1$ and similarly all traffic for `1.1.2.0/24` to $C2$. The remaining packets for `1.1.0.0/16` are distributed locally and all remaining packets (which in this example have to belong to `7.5.0.0/16`) are sent via $R3$. The setup for provider $P2$ with customers $C3$ and $C4$ is analogous, only the last entry in $R2$'s forwarding table is something new and should be ignored by now.

It is no mistake that $R1$ has no next hop for the network `7.5.4.0/24`, as routing will work correctly nonetheless. Assume $C2$ sends a packet to $C4$'s web server which happens to own the address `7.5.4.33`. As this address is not part of the local network the packet is forwarded to the only router reachable from $C2$. $R1$ in turn matches `7.5.4.33` with the network `7.5.0.0/16` and sends the packet to $R3$, which in turn delivers the packet to $R2$. Now $R2$ has a refined view of the network `7.5.0.0/16` and because `7.5.4.33` has the prefix `7.5.4.0/24` the packet is correctly delivered to $C4$. This controlled distribution of network prefixes is called aggregation and is the main tool to slow the growth of the forwarding tables in the Internet when using CIDR.

There is another special case included in our scenario. Customer $C1$ is troubled that its connection to the Internet is not reliable enough, so it leases another line from provider $P2$ to serve as a backup in case $P1$ has technical problems. This makes $C1$ *multi-homed* and explains the last entry in $R2$'s forwarding table which is a redirection to $C1$. So `1.1.1.0/24` has to occur in global forwarding tables ($R3$) as not all routes to $C1$ lead through $P1$'s

network anymore. Which route is chosen by $R3$ depends on the routing protocol. Here $R2$ sends packets to $C1$ via $P2$ as the (physical) path is shorter.

What was to be demonstrated is that finding the network for an IP address is done by testing for every network in the forwarding table if it is a prefix of the address. Additionally we have seen how for a single address multiple networks can match, the main reasons for such entries are multi-homing and changing providers without renumbering [FLYV93]. From the example it should have become clear that in case of multiple matches the most specific network (*i.e.*, longest prefix) should be used.

1.2 Longest prefix matching

The main problem we are dealing with in this thesis is longest prefix matching which is used for mapping IP addresses to networks. Stated more formally we are given a set of (possibly overlapping) strings P and a mapping $f : P \to H$ where H is some arbitrary set. Now we are asked to construct a data structure which for a given string s efficiently determines $f(p)$ where p is the longest string in P being a prefix of s.

As we are using longest prefix matching on source or destination addresses to classify IP packets, P is the set of IP prefixes (network ids) and s is an IP address (interpreted as bit string). The meaning of H depends on the kind of classification we are performing and the constructed data structure is the forwarding table. To flesh out the importance of this problem we will present some applications of longest prefix matching for IP packet classification next.

- *routing*
  This has been thoroughly discussed in the previous section. We use longest prefix matching to find the next hop for an IP packet. As this is probably the most demanding application it will be our standard example throughout this paper.

- *packet filtering*
  Based on a *black list* of untrusted networks we want to discard all packets originating from such a network at a firewall. The network is found by longest prefix matching.

- *quality of service*
  Packets from paying customers (identified by their source network) receive improved service, for example more bandwidth, higher priority, etc. Here H would be a set of service levels.

- *content delivery networks*
  A content delivery network is a set of servers cooperating in delivering

some content (*e.g.*, web pages) to the client host. The main idea is to mirror the content on all participating servers and redirect a clients host to the server nearest to it, reducing transportation cost and time. Finding the nearest server can be based on the source IP address of the client if we have a table mapping all known networks to a position (either geographically or relative to the Internet topology) and thus a best server. Then again we can use longest prefix matching on this table.

To get an impression what efficient means in this context, we will provide some numbers from the context of routing in the Internet backbone. According to [GH04] interface speeds of about 10 to 40 Gbps are common in todays backbone. With an assumed average IP packet size of about 1000 bytes ([SV99] even assumes only 2000 *bits*) about 1.2 to 5 million packets (or even 20 million) must be handled per second, each involving on forwarding table lookup. Additionally the routing table receives up to some hundred updates per second which have to be reflected in the forwarding table. Thus the forwarding table often is organized in a way supporting incremental modification. However [DBCP97] argues that it is sufficient to synchronize the forwarding table with the routing table about once a second, so incremental data structures are not mandatory if the construction of the forwarding table can be performed reasonably fast.

Due to these tight constraints we are not so much interested in the asymptotic complexity of a single lookup, which hides the constants involved, but more in an exact measure of required time. As on todays hardware for most computations the processing time is dominated by delays from memory access, we are especially interested in minimizing two properties of the calculated forwarding table:

1. the number of memory accesses required for one IP lookup,
2. the size of the resulting data structure, as a smaller forwarding table can be kept in a higher and thus faster level of the memory hierarchy (cache memory) reducing the cost of a single memory access.

The actual tradeoff between less memory lookups and reduced memory footprint depends on the exact hardware and router design used, so we also should investigate ways to reduce one of them at the cost of the other one.

Despite the strict timing requirements there are algorithms and data structures which are fast enough handling these, some of them are mentioned in the next section. But the problem is getting more complicated over time for several reasons.

- Due to the growth of the Internet and technological advances in network technology the overall IP traffic and thus the number of packets to be handled by routers grows. Although exact numbers are hard

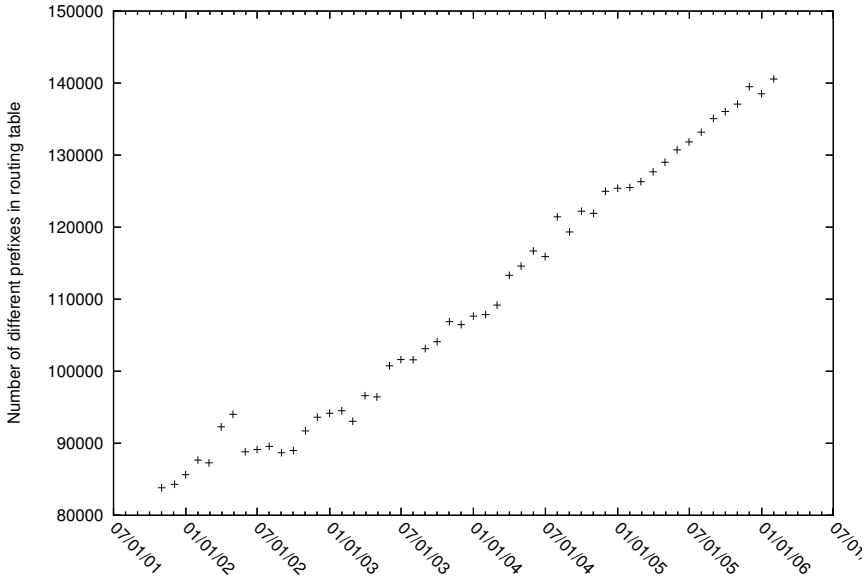


Figure 1.4: Number of known prefixes for route-views2.oregon-ix.net

to come by, an exponential traffic growth is generally assumed. More concrete [Odl03] concludes the Internet traffic to nearly double every year.

- Another factor is the number of network prefixes globally announced which also grows with the extension of the Internet. While exact numbers are router dependent due to issues like prefix aggregation, the general trend should be similar for all backbone routers. Figure 1.4 shows the evolution of the number of known prefixes for a single router[2]. The increase seems to be linear despite an assumed exponential growth of Internet size, showing that the goal of slowing down the growth of routing table size has been achieved by using CIDR.

- The final aspect is the introduction of IPv6 with 128 bit addresses which still requires longest prefix matching in the forwarding table. According to [HD03] the interface (*i.e.*, host) identifier is 64 bit in most cases, leaving at most 64 bit for the network prefix, but still doubling the maximum prefix length.

These observations clearly show the load for backbone routers increasing dramatically over time. As some of these developments are hard to predict, we cannot be sure to compensate the required efficiency gain by hardware

[2]Data extracted from the first routing table dump for each month of the BGP beacon `route-views2.oregon-ix.net` available from `http://www.routeviews.org/`

development (Moore's law) alone, but should also search for novel and improved ways to tackle this problem from the algorithm side.

1.3 Existing solutions

Of course every new solution to a solved problem should be compared with existing ones. To give an overview on the field of "competitors" we present some of the existing solutions for the longest prefix matching problem proposed in the literature. A very detailed overview is already given by [GH04] and [RSBD01] (which also provides a table of worst case complexities), so we will only cover the most important ones. Since the classical trie-based solutions (see Section 6.2) there have been three influential proposals.

- *compressed tries* [DBCP97]
  The version presented in [DBCP97] is also called Luleå-trie. An existing trie is transformed using leaf-pushing (see [SV99]) and alphabet expansion (Section 6.4) to reduce the expected number of memory accesses required to search to a leaf of the trie. The resulting trie is then compressed using a special scheme which does not complicate the trie traversal too much, with the goal to make it fit into processor cache memory. A slightly generalized but similar procedure is also proposed by [NK98], called LC-trie.

- *ranges* [LSV99]
  Another approach is to interpret IP prefixes as address ranges and perform binary search on these intervals. Care must be taken to always find the longest prefix (*i.e.*, the shortest range). To simplify this process the ranges can be organized in a *range tree*.

- *hashing* [WVTP97]
  As the length of the prefix for an IP address is not known in advance, hashing is not easily adjusted to this problem. This is solved by having a separate hash table for each possible prefix length and performing a binary search on the prefix length. To make this work (as there might be gaps in the lengths of the existing prefixes for an address) additional dummy entries have to be inserted into these tables.

It seems that despite their age these are still the basic techniques used, although all of them have been refined, especially towards simplifying updates of the data structure (see [GH04] for details).

In spirit the solution we will be investigating in the following chapters is similar to the trie based approaches from [DBCP97] and [NK98] in that we start with a trie and attempt to reach a compressed representation. But while these solutions use specialized data structures, heavily optimized

towards the expected distribution of IP prefixes and performing lots of "bit-fiddling" to reach a decent level of compression, we are relying on minimization results from automata theory to eliminate redundancies in the forwarding table.

Chapter 2

Preliminaries

This chapter introduces the definitions and notation we will build upon in later chapters. Additionally we prove some technical lemmas needed but not directly related to the topic of the chapter where it is applied.

2.1 Set partitions

Let I be an index set[1], S a set. A tuple $\mathcal{P} = (P_i)_{i \in I}$ is called a *partition of* S iff

1. $\forall i \in I : \emptyset \neq P_i \subseteq S$
2. $\forall i, j \in I : \ i = j \ \vee \ P_i \cap P_j = \emptyset$
3. $\bigcup_{i \in I} P_i = S$.

In the case $|I| = k$ we call $\mathcal{P}$ a *k-partition* and usually assume $I = \{1, \ldots, k\}$. For every $i \in I$ the set P_i is called a *component* of $\mathcal{P}$. Denote by $\chi_{\mathcal{P}}(s)$ the index of the component in $\mathcal{P}$ containing s, *i.e.*,

$$\forall s \in S : \forall i \in I : \chi_{\mathcal{P}}(s) = i \ \Leftrightarrow \ s \in P_i \,.$$

The function $\chi_{\mathcal{P}} : S \to I$ is called the *characteristic function of* $\mathcal{P}$.

To simplify notation we identify the partition $(P_i)_{i \in I}$ with the corresponding set $\{P_i \mid i \in I\}$ and thus for example use $P_i \in \mathcal{P}$ for membership, or $|\mathcal{P}|$ for the number of components. We will also call a set $\mathcal{S} = \{S_i\}_{i \in I}$ a partition if the canonical tuple $(S_i)_{i \in I}$ is a partition. Additionally we introduce the operator ses (*skip empty sets*) which removes all empty sets from a tuple, formally defined as

$$\operatorname{ses}(\mathcal{P}) := (P_i)_{i \in I \wedge P_i \neq \emptyset} \,.$$

[1] In this context an *index set* is just an ordered set of labels used for indexing and is not related to Rice's theorem.

This allows handling tuples that are "nearly" partitions.

If S is a set, $T \subseteq S$, and $\mathcal{P} = (P_i)_{i \in I}$ is a partition of S, denote by $\mathcal{P}|_T$ the *restriction of* $\mathcal{P}$ *to* T defined by

$$\mathcal{P}|_T := \operatorname{ses}\left((P_i \cap T)_{i \in I}\right),$$

which is a partition of T.

Let $\mathcal{P}$ and $\mathcal{R}$ be partitions of some set S. We call $\mathcal{R}$ a *refinement* of $\mathcal{P}$, iff

$$\forall R \in \mathcal{R}\ \exists P \in \mathcal{P} : R \subseteq P\,.$$

Lemma 2.1. *Let $\mathcal{P}$ and $\mathcal{R}$ be partitions of some set S, $\mathcal{R}$ a refinement of $\mathcal{P}$. Then $|\mathcal{R}| \geq |\mathcal{P}|$.*

Proof. Define the function $\iota : \mathcal{R} \to \mathcal{P}$ by $\iota(R) = P$ iff $R \subseteq P$. This is a valid function, as for every $R \in \mathcal{R}$ there is at most one such P due to $\mathcal{P}$ being a partition and at least one P because $\mathcal{R}$ is a refinement of $\mathcal{P}$. We have to show that ι is surjective, *i.e.*, for every $P \in \mathcal{P}$ there is an $R \in \mathcal{R}$ with $\iota(R) = P$, but this is obvious, as $\mathcal{R}$ is a partition and P not empty and the elements from P have to appear in some $R \in \mathcal{R}$. □

2.2 Formal languages

A non-empty set of symbols Σ is called *alphabet* if it is both finite and totally ordered. For an alphabet Σ we write Σ^n for the set of all words over Σ of length n and Σ^* for the set of all words of finite length. The set of words of length at most, at least, shorter than, and longer than n is written as $\Sigma^{\leq n}$, $\Sigma^{\geq n}$, $\Sigma^{<n}$, respectively $\Sigma^{>n}$. The empty word is denoted by ϵ.

For two words v and w we write vw for the concatenated word. Exponentiation is defined by the recursion $w^0 := \epsilon$, $w^{i+1} := w^i w$. Furthermore we identify the alphabet Σ and the set of one letter words Σ^1 which often simplifies notation.

For any word w and letter a we use $\#_a(w)$ for the number of a's in w. So if $w = w_1 w_2 \dots w_n$ with $w_1, w_2, \dots, w_n \in \Sigma$, then

$$\#_a(w) := |\{i \mid w_i = a\}|\,.$$

For any finite set Q, alphabet Σ, and function $\delta : Q \times \Sigma \to Q$ the *canonical expansion of* δ *to* Σ^* is the function $\hat{\delta} : Q \times \Sigma^* \to Q$ inductively defined by

$$\begin{aligned} \forall q \in Q : \quad & \hat{\delta}(q, \epsilon) && := q \\ \forall q \in Q\, \forall w \in \Sigma^*\, \forall a \in \Sigma : \quad & \hat{\delta}(q, wa) && := \delta(\hat{\delta}(q, w), a) \end{aligned}$$

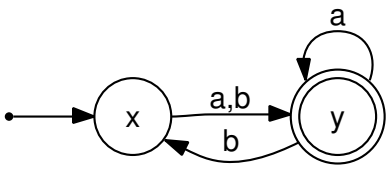


Figure 2.1: Example of a simple DFA

This also implies $\delta(q,a) = \hat{\delta}(q,a)$ for all $q \in Q$ and $a \in \Sigma$, and a simple induction shows $\hat{\delta}(q,uv) = \hat{\delta}(\hat{\delta}(q,u),v)$ for all $q \in Q$ and $u,v \in \Sigma^*$.

Let Σ be an alphabet. A set $L \subseteq \Sigma^*$ is called a *language over* Σ. A k-partition $\mathcal{L}$ of Σ^* is called a *language k-partition over* Σ, a k-partition of $\Sigma^{\leq l}$ is a *language k-partition of order l over* Σ.

A *deterministic finite automaton (DFA)* is defined as a quintuple $A = (\Sigma, Q, q_0, F, \delta)$, with

- alphabet Σ
- finite state set Q
- initial state $q_0 \in Q$
- final states $F \subseteq Q$
- total transition function $\delta : (Q \times \Sigma) \to Q$

The transition function is canonically expanded to Σ^*.

The *accepted language* $L(A)$ for a DFA $A = (\Sigma, Q, q_0, F, \delta)$ is defined as $L(A) := \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$.

When describing DFAs we often use a graphical notation as in Figure 2.1. Each circle represents a state, the labeled edges (transitions) define δ, the start state is marked by an incoming arrow without a source state, and the final states are marked with double circles. So the figure displays the automaton $(\{a,b\}, \{x,y\}, x, \{y\}, \delta)$ with $\delta(x,a) = y$, $\delta(x,b) = y$, $\delta(y,a) = y$, $\delta(y,b) = x$.

2.3 Relations

Let Σ be an alphabet. A relation $\sim \subseteq \Sigma^* \times \Sigma^*$ is called *right invariant* iff

$$x \sim y \;\Rightarrow\; \forall a \in \Sigma : xa \sim ya\,.$$

Obviously for any right invariant relation $\sim \subseteq \Sigma^* \times \Sigma^*$ the following holds by induction:

$$x \sim y \;\Rightarrow\; \forall w \in \Sigma^* : xw \sim yw\,.$$

Every partition $\mathcal{P}$ of a set S induces an equivalence relation $\equiv_{\mathcal{P}}$ over S as follows:

$$s \equiv_{\mathcal{P}} t \;:\Leftrightarrow\; \chi_{\mathcal{P}}(s) = \chi_{\mathcal{P}}(t)\,.$$

A partition $\mathcal{P}$ of Σ^* is called *right invariant* if its induced equivalence relation $\equiv_{\mathcal{P}}$ is right invariant.

Lemma 2.2. *Let Σ be an alphabet, $\mathcal{P}$ a partition of Σ^*. Then $\mathcal{P}$ is right invariant iff*

$$\forall a \in \Sigma\ \forall P \in \mathcal{P}\ \exists P' \in \mathcal{P} : Pa \subseteq P'\,,$$

where $Pa := \{wa \mid w \in P\}$.

Proof. We start with the *only if* case. Let $\mathcal{P}$ be right invariant and choose $a \in \Sigma$, $P \in \mathcal{P}$, and any two elements $x, y \in P$. Set $P' := P_{\chi_{\mathcal{P}}(xa)}$. We have to show $ya \in P'$. For the induced equivalence relation we have $x \equiv_{\mathcal{P}} y$ and due to the right invariance also $xa \equiv_{\mathcal{P}} ya$. But this means $ya \in P'$ which was to be shown.

Now for the *if* part. Let $a \in \Sigma$ and $x, y \in P$, so $x \equiv_{\mathcal{P}} y$. There is a $P' \in \mathcal{P}$ such that $xa, ya \in P'$ thus $xa \equiv_{\mathcal{P}} ya$. So $\equiv_{\mathcal{P}}$ and thus $\mathcal{P}$ are right invariant. □

Let S be a set and $\equiv$ an equivalence relation over S. For each $s \in S$ the set $[s]_{\equiv} := \{t \in S \mid t \equiv s\}$ is called the *equivalence class* of s with respect to $\equiv$. As $\equiv$ is an equivalence relation, for all $s \in S$ and $a, b \in [s]_{\equiv}$ we have $a \equiv b$. Furthermore the set $\{[s]_{\equiv} \mid s \in S\}$ is a partition of S, thus for $s \not\equiv t$ and $a \in [s]_{\equiv}$, $b \in [t]_{\equiv}$ we can conclude $a \not\equiv b$

Let S be a set and $\equiv$ an equivalence relation over S. The cardinal number $|\{[s]_{\equiv} \mid s \in S\}|$, *i.e.*, the number of equivalence classes, is called the *index of* $\equiv$.

Example 2.3. *Define the following relations $\equiv_1, \equiv_2 \subseteq \mathbb{N} \times \mathbb{N}$:*

$$\begin{aligned} x \equiv_1 y &:\Leftrightarrow x \bmod 5 = y \bmod 5 \\ x \equiv_2 y &:\Leftrightarrow \left\lfloor \frac{x}{5} \right\rfloor = \left\lfloor \frac{y}{5} \right\rfloor \end{aligned}$$

The first relation has finite index 5, the equivalence classes being $[1]_{\equiv_1}$, $[2]_{\equiv_1}$, $[3]_{\equiv_1}$, $[4]_{\equiv_1}$, and $[5]_{\equiv_1}$, while the second one has infinite index with the classes $[1]_{\equiv_2}$,$[6]_{\equiv_2}$,$[11]_{\equiv_2}$, ...

2.4 Complete and independent sets

For studying the partitions induced by relations we introduce some more notation.

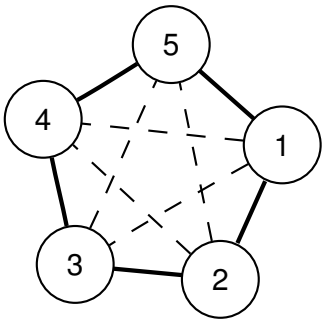


Figure 2.2: A relation where the inequality from Corollary 2.7 is strict

Definition 2.4. *Let $\approx$ be a reflexive symmetric relation over some set S.*

1. *A set $R \subseteq S$ is called* $\approx$-complete *if* $\forall x, y \in R : x \approx y$.
2. *A set $D \subseteq S$ is called* $\approx$-independent *if* $\forall x, y \in D : x \not\approx y$.
3. *A partition $\mathcal{P}$ of S is called* $\approx$-complete partition *if all $P \in \mathcal{P}$ are $\approx$-complete.*

The following is consistent with the definition of the *index* for equivalence relations.

Definition 2.5. *Let $\approx$ be a reflexive symmetric relation over some set S. The* index of $\approx$ *(written as* $\mathrm{index}(\approx)$*) is the cardinality number*

$$\min\{|\mathcal{P}| \mid \mathcal{P} \text{ is a } \approx\text{-complete partition of } S\}\,.$$

There is an obvious dependency between the size of $\approx$-independent sets and $\approx$-complete partitions.

Lemma 2.6. *Let $\approx$ be a reflexive symmetric relation over some set S, the set $D \subseteq S$ $\approx$-independent, and $\mathcal{P}$ a finite $\approx$-complete partition of S. Then $|D| \leq |\mathcal{P}|$.*

Proof. As $\mathcal{P}$ is a partition of S each element $d \in D$ must be contained in some component $P_d \in \mathcal{P}$. But as all elements in P_d are related with respect to $\approx$ no other element from D can be in the same component. So from the pigeon-hole principle we conclude $|D| \leq |\mathcal{P}|$. □

Corollary 2.7. *Let $\approx$ be a reflexive symmetric relation over some set S. Then*

$$\max\{|D| \mid D \subseteq S \text{ is } \approx\text{-independent}\} \leq \mathrm{index}(\approx)\,.$$

The following example shows that the inequality in the previous corollary can be strict.

Example 2.8. *Let* $S := \{1, 2, 3, 4, 5\}$ *and define the relation* $\approx$ *by*

$$x \approx y \;:\Leftrightarrow\; |x - y| \leq 1 \vee \{x, y\} = \{1, 5\},$$

which is both reflexive and symmetric. A graphical representation of this relation omitting self-loops is given in Figure 2.2 where bold lines indicate relatedness while dashed lines are drawn between non-related elements. Obviously we cannot find a $\approx$*-independent set with more than two elements. On the other hand there also is no* $\approx$*-complete set with more than two elements and so every* $\approx$*-complete partition must have at least three components.*

Chapter 3

Partition automata

Deciding languages, *i.e.*, testing whether a given word is in a language, is a problem studied exhaustively in formal language theory. A natural generalization is the following: given a partition of all words, decide which component of the partition contains the word in question. This chapter deals with the adaption of techniques that have been thoroughly investigated in the context of formal languages. We start by looking at finite automata for this problem and giving a characterization of language partitions that can be decided by them. Then we show how to extend known algorithms for automaton minimization to this generalized case.

A partition automaton is the straight forward generalization of a finite deterministic automaton. Instead of a set of final states that correspond to the language accepted we partition the state set related to the language partition to be decided.

Definition 3.1. *A* deterministic finite k-partition automaton ($\mathrm{DP}_k\mathrm{A}$) *is a quintuple* $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$*, with*

- *alphabet* Σ
- *finite state set* Q
- *initial state* $q_0 \in Q$
- k*-partition* $\mathcal{P}$ *of* Q
- *total transition function* $\delta : (Q \times \Sigma) \to Q$

The transition function defines $\hat{\delta}$ *as its canonically expansion.*

We sometimes call the components of $\mathcal{P}$ *state classes*. Usually the first component of $\mathcal{P}$ collects all states that have no deeper meaning to us. This corresponds to the non-final states with DFAs. Obviously every DFA $A = (\Sigma, Q, q_0, F, \delta)$ is equivalent to the $\mathrm{DP}_2\mathrm{A}$ $(\Sigma, Q, q_0, (Q \setminus F, F), \delta)$. Before analyzing the structure of DP_kAs in more detail, we should fix some more

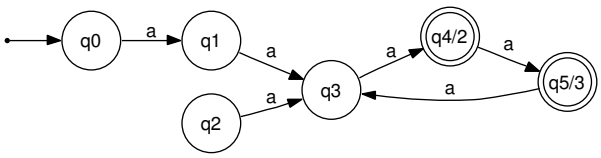


Figure 3.1: An example of a $\mathrm{DP}_3\mathrm{A}$

notation first. For the remainder of this section let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a $\mathrm{DP}_k\mathrm{A}$.

For $R \subseteq Q$ the *accepted language relative to* R denoted by $L_R(A)$ is the set $\{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in R\}$. If $R = \{q\}$ consists of a single state we write $L_q(A)$ instead of $L_{\{q\}}(A)$. $L_{\{q\}}(A)$ is also called the *state language* of q.

Remark 3.2. *The tuples* $\operatorname{ses}(L_P(A))_{P\in\mathcal{P}}$ *and* $\operatorname{ses}(L_q(A))_{q\in Q}$ *are language partitions of* Σ^*. *The partition* $\operatorname{ses}(L_q(A))_{q\in Q}$ *is right invariant.*

Proof. As $\hat{\delta}$ is a total function, *i.e.*, every word is mapped to some state, it is obvious from the definition that those tuples are partitions of Σ^*. For the right invariance let $q \in Q$ and $x, y \in L_q(A)$. Then for any $a \in \Sigma$ we have $\hat{\delta}(q_0, xa) = \delta(\hat{\delta}(q_0, x), a) = \delta(q, a) = \delta(\hat{\delta}(q_0, y), a) = \hat{\delta}(q_0, ya)$ and thus xa and ya are mapped to the same state. Now apply Lemma 2.2. □

The *accepted language partition* $\mathcal{L}(A)$ *of* A is defined as $\operatorname{ses}(L_P(A))_{P\in\mathcal{P}}$. Given a language partition $\mathcal{L}$ and a $\mathrm{DP}_k\mathrm{A}$ A with $\mathcal{L}(A) = \mathcal{L}$, A is said to *accept* or *decide* $\mathcal{L}$. If $\mathcal{L} = (L_1, \ldots, L_k)$ is a partition of order l (that is of $\Sigma^{\leq l}$) for some $l \in \mathbb{N}$ then A *decides* $\mathcal{L}$ if it decides $(L_1 \cup \Sigma^{>l}, L_2, \ldots, L_k)$, *i.e.*, all unspecified words are put to the first component of the partition.

For states $p, q \in Q$ we say p *is reachable from* q iff there is a word $w \in \Sigma^* \setminus \{\epsilon\}$ with $\hat{\delta}(q, w) = p$. The state q is an *acyclic state* if q is not reachable from q. A state $q \in Q$ is called *useful* iff it is reachable from q_0, otherwise we call it *useless*. A state q with $\delta(q, a) = q$ for all $a \in \Sigma$ (one which can never be left) is called a *sink state*.

We use the same graphical notation as for DFAs. States in the first component of $\mathcal{P}$ are shown in a single circle, all other states are displayed using a double circle. If $|\mathcal{P}| > 2$ we write $\chi_{\mathcal{P}}(q)$ to each double circle state q separated by a slash. So Figure 3.1 indicates $\mathcal{P} = (\{q_0, q_1, q_2, q_3\}, \{q_4\}, \{q_5\})$. The state q_2 is useless, all other states are useful. The only acyclic states are q_0, q_1, q_2, the remaining ones are acyclic.

3.1 The Myhill-Nerode theorem

The Myhill-Nerode theorem (see [HU79]) gives a characterization of regular languages and provides a criterion for the minimality of DFAs. The goal of

this section is to rephrase this theorem for DP_kAs. A central part of the theorem is a right invariant equivalence relation connecting words "behaving" the same when extended. So we too start by defining a suitable relation.

Definition 3.3. *Let Σ be an alphabet, $\mathcal{L}$ a language partition of Σ^*. Define the relation $\equiv_{\mathcal{L}} \subseteq \Sigma^* \times \Sigma^*$ by*

$$x \equiv_{\mathcal{L}} y \;:\Leftrightarrow\; \forall w \in \Sigma^* : \chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)\,.$$

Lemma 3.4. *For every alphabet Σ and language partition $\mathcal{L}$ the relation $\equiv_{\mathcal{L}}$ is a right invariant equivalence relation.*

Proof. Obviously $\equiv_{\mathcal{L}}$ is both reflexive and symmetric, so for being an equivalence relation it remains to show transitivity. Let $x, y, z \in \Sigma^*$, $x \equiv_{\mathcal{L}} y$ and $y \equiv_{\mathcal{L}} z$. Then for any $w \in \Sigma^*$ we have $\chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)$ and $\chi_{\mathcal{L}}(yw) = \chi_{\mathcal{L}}(zw)$. Thus $\chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(zw)$ and $x \equiv_{\mathcal{L}} z$.

For the right invariance choose $a \in \Sigma$ and $x, y \in \Sigma^*$ such that $x \equiv_{\mathcal{L}} y$. Then for every word $w \in \Sigma^*$ the words $x(aw) = (xa)w$ and $y(aw) = (ya)w$ are in the same component of $\mathcal{L}$ and thus $xa \equiv_{\mathcal{L}} ya$. □

The relation $\equiv_{\mathcal{L}}$ was chosen such that all words in the same state language $L_q(A)$ are equivalent. Due to the finiteness of the state set this directly limits the language partitions decidable by DP_kAs.

Lemma 3.5. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$. Then for all $q \in Q$ and $x, y \in L_q(A)$ we have $x \equiv_{\mathcal{L}(A)} y$.*

Proof. Let $q \in Q$ and choose any $x, y \in L_q(A)$. We know $\hat{\delta}(q_0, x) = q = \hat{\delta}(q_0, y)$. For any $w \in \Sigma^*$ then $\hat{\delta}(q_0, xw) = \hat{\delta}(\hat{\delta}(q_0, x), w) = \hat{\delta}(\hat{\delta}(q_0, y), w) = \hat{\delta}(q_0, yw)$ and thus $xw, yw \in L_{q'}(A)$ for some $q' \in Q$. So there exists $P \in \mathcal{P}$ with $q' \in P$ which implies $xw, yw \in L_P(A)$ and $\chi_{\mathcal{L}(A)}(xw) = \chi_{\mathcal{L}(A)}(yw)$. According to Definition 3.3 we thus have $x \equiv_{\mathcal{L}(A)} y$. □

Lemma 3.6. *Let A be a DP_kA. Then the relation $\equiv_{\mathcal{L}(A)}$ has finite index.*

Proof. Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$. Due to Lemma 3.5 the language partition $\operatorname{ses}(L_q(A))_{q \in Q}$ is a refinement of the partition $\{[w]_{\equiv} \mid w \in \Sigma^*\}$, because all words in $L_q(A)$ are equivalent and thus are in the same equivalence class. But Q and so also $\{L_q(A) \mid q \in Q\}$ is finite and according to Lemma 2.1 the index of $\equiv_{\mathcal{L}(A)}$ cannot be infinite. □

Now we have everything to prove the main result for this section which is a necessary and sufficient condition for a language partition to be decidable by a DP_kA. The uniqueness result obtained at the same time will be used later in proving the correctness of a DP_kA minimization algorithm.

Theorem 3.7. *Let Σ be an alphabet, $\mathcal{L}$ a language partition of Σ^*. There is a DP$_k$A A accepting $\mathcal{L}$ iff $\equiv_{\mathcal{L}}$ has finite index N.*

The minimal automaton accepting $\mathcal{L}$ is unique up to isomorphic renaming of states and has N states.

Proof. The *only if* case was proven in Lemma 3.6.

For the remainder let $\mathcal{L}$ be a language partition and N the index of $\equiv_{\mathcal{L}}$. Set $Q := \{[w]_{\equiv_{\mathcal{L}}} \mid w \in \Sigma^*\}$, $q_0 := [\epsilon]_{\equiv_{\mathcal{L}}}$. Define $\delta : Q \times \Sigma \to Q, ([w]_{\equiv_{\mathcal{L}}}, a) \mapsto [wa]_{\equiv_{\mathcal{L}}}$. As Q is the induced partition of a right invariant equality relation, it is right invariant and the mapping is unambiguous. The relation $\equiv_{\mathcal{L}}$ ensures that all words in the elements of Q are in the same component of $\mathcal{L}$ and so we can choose $\mathcal{P} := \{P(L) \mid L \in \mathcal{L}\}$ where $P(L) := \{q \in Q \mid \forall w \in q : w \in L\}$. Then $A := (\Sigma, Q, q_0, \mathcal{P}, \delta)$ is a DP$_k$A and accepts $\mathcal{L}$ as can be seen by the fact that $\hat{\delta}(q_0, w) = [w]_{\equiv_{\mathcal{L}}}$ and the way $\mathcal{P}$ was defined. By definition of the index the automaton A has N states.

It remains to show that there is neither a smaller automaton nor one with N states that is not isomorphic to A. In both cases a simple counting argument yields that then there had to be two words $x, y \in \Sigma^*$ with $x \not\equiv_{\mathcal{L}} y$ mapped to the same state in contradiction to Lemma 3.5. □

3.2 Adapting a DFA minimization algorithm

We are often interested in a minimal DP$_k$A for a given language partition. The obvious reason is for saving memory with a more compact representation, but due to the uniqueness of the minimal DP$_k$A is can also be used as a normal form for DP$_k$As allowing for example equivalence testing.

For the problem of DFA minimization a variety of algorithms have been proposed, an overview is given in [Wat94]. The algorithm we are adapting in this section is described in [HU79] and was chosen as it is probably the most well-known algorithm for this problem due to the wide spread of this book.

The algorithm works by merging states that have been identified to be equivalent (have the same behavior). From Theorem 3.7 is seems intuitive to call two states equivalent if all words from their state languages are equivalent.

Definition 3.8. *For a given DP$_k$A $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ we define the relation $\equiv_A \subseteq Q \times Q$ by*

$$q \equiv_A r \;:\Leftrightarrow\; \forall x \in L_q(A)\ \forall y \in L_r(A) : x \equiv_{\mathcal{L}(A)} y\,.$$

Remark 3.9. *The following formulation is easily seen to be equivalent to the definition above for useful states q and r:*

$$q \equiv_A r \;\Leftrightarrow\; \forall w \in \Sigma^* : \chi_{\mathcal{P}}(\hat{\delta}(q, w)) = \chi_{\mathcal{P}}(\hat{\delta}(r, w))$$

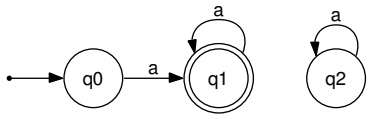


Figure 3.2: A $\mathrm{DP}_k\mathrm{A}$ with a useless state

As $\equiv_A$ was introduced to identify equivalent states we expect $\equiv_A$ to be an equivalence relation. For this to be true we have to restrict ourselves to useful states.

Lemma 3.10. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP$_k$A without useless states. The relation $\equiv_A$ is an equivalence relation.*

Proof. The relation is obviously symmetric. Reflexivity can be seen from Lemma 3.5. For transitivity let $q, r, s \in Q$ and $q \equiv_A r$, $r \equiv_A s$. As A has no useless states there are words $w_q \in L_q(A)$, $w_r \in L_r(A)$, and $w_s \in L_s(A)$. Because of $q \equiv_A r$ and $r \equiv_A s$ we have $w_q \equiv_{\mathcal{L}(A)} w_r$ and $w_r \equiv_{\mathcal{L}(A)} w_s$. As $\equiv_{\mathcal{L}(A)}$ is transitive together with Lemma 3.5 we get $q \equiv_A s$. □

The exclusion of useless states is required, as shown by the next example.

Example 3.11. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP$_k$A with $\Sigma = \{a\}$, $Q = \{q_0, q_1, q_2\}$, $\mathcal{P} = \{\{q_0, q_2\}, \{q_1\}\}$ and δ as implied by Figure 3.2. Obviously q_2 is not useful and so $L_{q_2}(A)$ is empty. Following Definition 3.8 this would intend $q_0 \equiv_A q_2$ and $q_2 \equiv_A q_1$, but as the words from L_{q_0} and L_{q_1} are in different components of $\mathcal{L}(A)$ we have $q_0 \not\equiv_A q_1$, so we lost transitivity here.*

Right invariance was defined for relations on finite words. The relation $\equiv_A$ being based on a right invariant relation has a similar property, namely carrying equivalence over state transitions.

Lemma 3.12. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP$_k$A without useless states and $q, r \in Q$ with $q \equiv_A r$. Then for every $a \in \Sigma$ we have $\delta(q, a) \equiv_A \delta(r, a)$.*

Proof. We know that all words in $L_q(A) \cup L_r(A)$ are equivalent with respect to $\equiv_{\mathcal{L}(A)}$. But as $\equiv_{\mathcal{L}(A)}$ is right invariant for any $a \in \Sigma$ the words from $W := \{wa \mid w \in L_q(A) \cup L_r(A)\}$ are all equivalent, too. As q and r are not useless we have $W \cap L_{\delta(q,a)}(A) \neq \emptyset$ and $W \cap L_{\delta(r,a)}(A) \neq \emptyset$ and so some words in $L_{\delta(q,a)}(A)$ and $L_{\delta(r,a)}(A)$ are equivalent, thus $\delta(q, a) \equiv \delta(r, a)$. □

The actual minimization algorithm consists of two steps. The first phase calculates the equivalence relation $\equiv_A$ on all state pairs, while the second merges all equivalent states into one. Before describing the algorithm in more detail, we prove that the merging phase indeed produces the minimal $\mathrm{DP}_k\mathrm{A}$.

Lemma 3.13. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP$_k$A without useless states. Then the automaton we receive by merging all states in $[q]_{\equiv_A}$ for every $q \in Q$ is a minimal DP$_k$A accepting $\mathcal{L}(A)$.*

Merging here means to replace the occurrence of every $q \in Q$ in the automaton description by $[q]_{\equiv_A}$.

Proof. At first we should check whether δ is still defined properly, that is for any $q \in Q$ and $r, s \in [q]_{\equiv_A}$ we must ensure $\delta(r, a) \equiv_A \delta(s, a)$ for any $a \in \Sigma$, *i.e.*, the possibly different mappings must point into the same state equivalence class. But this is exactly what Lemma 3.12 proved.

The tuple $\mathcal{P}$ is also still a valid partition as two states that are in different components cannot be equivalent.

Next we will show that for $q \in Q$ the set $L_q(A) \subseteq L_{[q]_{\equiv_A}}(A)$. Let $w = a_1 a_2 \dots a_l \in L_q(A)$ with all $a_i \in \Sigma$. Then there are states $q_0, q_1, \dots, (q_l = q)$ such that $\delta(q_{i-1}, a_i) = q_i$ $(1 \leq i \leq l)$. Performing the substitution yields $[q_0]_{\equiv_A}, [q_1]_{\equiv_A}, \dots, ([q_l]_{\equiv_A} = [q]_{\equiv_A})$ with $\delta([q_{i-1}]_{\equiv_A}, a_i) = [q_i]_{\equiv_A}$ $(1 \leq i \leq l)$, so $w \in L_{[q]_{\equiv_A}}$. This together with the fact that the new classes $[q]_{\equiv_A}$ are in the same component in $\mathcal{P}$ as q shows the automaton to still accept the same language partition.

It remains to prove minimality. After merging there are no two states that are equivalent, so for all $x, y \in \Sigma^*$ with $x \equiv_{\mathcal{L}(A)} y$ there is a state $[q]_{\equiv_A}$ such that $x, y \in L_{[q]_{\equiv_A}}(A)$ as otherwise we had two distinct but equivalent states. But this means our states are exactly the equivalence classes of $\equiv_{\mathcal{L}(A)}$ and so the constructed automaton is isomorphic to the minimal one from Theorem 3.7. □

Now we are ready to describe the algorithm for DP$_k$A minimization. Pseudo code is given in Algorithm 3.1. The first step of checking for and removing useless states is not detailed as it is a straight forward application of depth first search (see, *e.g.*, [CLRS01]). The variables $E_{\{q,r\}}$ denote whether the states q and r are equivalent. Initially they are optimistically initialized to **true** unless the states are in different components of $\mathcal{P}$ which obviously prevents them from being equivalent. The next phase searches for proofs that two states q and r are not equivalent which according to above lemma is the case when the successor pair $(\delta(q, a), \delta(r, a))$ is inequivalent for some $a \in \Sigma$. When checking the successor pair it might be the case that this in turn has not yet been proven inequal but will be later. To avoid having to recheck all state pairs again after some E values have changed, we manage lists $S_{\{q,r\}}$ for each state pair containing those pairs of states that are inequivalent if $\{q, r\}$ is inequivalent. So when a state pair is found not to be equivalent, the *mark_inequal* function in Algorithm 3.2 is used to mark all state pair in this list as inequivalent and recursively traverse the lists of those pairs. Finally the equivalent states are merged which is a simple process.

Algorithm 3.1 A minimization algorithm for DP_kAs

Input: A DP_kA $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$
Output: A minimal DP_kA accepting the same language partition as A

if A has useless states **then**
  remove all useless states
  rerun the algorithm on the resulting automaton
end if

for each $(q, r) \in Q \times Q$ **do**
  if $\chi_{\mathcal{P}}(q) = \chi_{\mathcal{P}}(r)$ **then**
    $E_{\{q,r\}} :=$ **true**
  else
    $E_{\{q,r\}} :=$ **false**
  end if
  $S_{\{q,r\}} := \emptyset$
end for

for each $(q, r) \in Q \times Q$ **do**
  for each $a \in \Sigma$ **do**
    if $\neg E_{\{\delta(q,a),\delta(r,a)\}}$ **then**
      mark_inequal (q, r)
    else
      $S_{\{\delta(q,a),\delta(r,a)\}} := S_{\{\delta(q,a),\delta(r,a)\}} \cup \{(q, r)\}$
    end if
  end for
end for

Create the output DP_kA by merging all states q, r with $E_{\{q,r\}} =$ **true** as described in Lemma 3.13.

Algorithm 3.2 The *mark_inequal* procedure used in Algorithm 3.1

Input: States q, r

{We use the same global $E_{\{x,y\}}$ and $S_{\{x,y\}}$ as in Algorithm 3.1}
$E_{\{q,r\}} :=$ **false**
for each $(s, t) \in S_{\{q,r\}}$ **do**
  $S_{\{q,r\}} := S_{\{q,r\}} \setminus \{(s, t)\}$
  mark_inequal (s, t)
end for

After having seen the meaning of the variables and the overall idea we conclude with a formal proof of the correctness and the complexity of the minimization algorithm.

Theorem 3.14. *For a given DP_kA A with alphabet Σ and n states Algorithm 3.1 (using the subroutine in Algorithm 3.2) calculates a minimal DP_kA accepting the same language partition as A in $O(|\Sigma|n^2)$ time and space.*

Proof. Due to the first **if** in the algorithm we may assume that A has only useful states.

For the correctness we will show that when the algorithm terminates we have $E_{\{q,r\}} \Leftrightarrow q \equiv_A r$ and then apply Lemma 3.13. From Lemma 3.12 we know that two states q, r are inequivalent, if for some $a \in \Sigma$ the states $\delta(q, a)$ and $\delta(r, a)$ are not equivalent. The list $S_{\{q,r\}}$ holds all state pairs that have been inspected so far and would be inequivalent if q and r are inequivalent. Additionally we know that states that are in different components of $\mathcal{P}$ cannot be equivalent. So every time we set an $E_{\{q,r\}}$ to **false** we have some proof that $q \not\equiv_A r$ and consequently for $E_{\{q,r\}} =$ **false** the final answer of the algorithm is always correct.

Now assume after execution $E_{\{q,r\}} =$ **true**, but $q \not\equiv_A r$, that is there is a word $w \in \Sigma^*$ such that $\chi_{\mathcal{P}}(\hat{\delta}(q, w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(r, w))$. This implies that for every prefix v of w also $\hat{\delta}(q, v) \not\equiv_A \hat{\delta}(r, v)$. We know that $E_{\{\hat{\delta}(q,w),\hat{\delta}(r,w)\}} =$ **false** as states in different components of $\mathcal{P}$ are initialized this way. So there has to be some prefix w_0 of w and $a \in \Sigma$ such that $s := \hat{\delta}(q, w_0)$, $t := \hat{\delta}(r, w_0)$, $s \not\equiv_A t$, $\delta(s, a) \not\equiv_A \delta(t, a)$ but $E_{\{s,t\}} =$ **true** and $E_{\{\delta(s,a),\delta(t,a)\}} =$ **false**. But then either $E_{\{\delta(s,a),\delta(t,a)\}}$ was already **false** when we inspected the pair (s, t) for a in which case $E_{\{s,t\}}$ would have been set to **false**, or $E_{\{\delta(s,a),\delta(t,a)\}}$ was not yet **false** but then the pair (s, t) would have been added to the list $S_{\{\delta(s,a),\delta(t,a)\}}$ and $E_{\{s,t\}}$ would have been set to **false** when $E_{\{\delta(s,a),\delta(t,a)\}}$ was toggled to **false**. This contradiction concludes the correctness proof.

For the time and space complexity we know that checking the automaton for unused states and fixing it if necessary can be handled by a breadth first search in $O(|\Sigma|n)$ time and space. The $O(|\Sigma|n^2)$ bound has its source in the two nested **for** loops and the fact that every state pair is added to at most $|\Sigma|$ lists which are in turn cleared while they are traversed. □

3.3 Hopcroft's algorithm

While the algorithm from the previous section is simple, with its quadratic running time it is too slow for minimizing large DP_kAs. For the minimization of DFAs the most efficient algorithm currently known is due to John Hopcroft and described in [Hop71]. It can minimize a DFA with n states and alphabet Σ in $O(|\Sigma|n \log n)$ steps using only $O(|\Sigma|n)$ space. In this section we give an adaption of this algorithm for DP_kAs following the description

of [Gri73], where a slightly simplified version of Hopcroft's algorithm is presented. As the changes needed are only minor and a complete analysis requires the description of many implementation details, we will only give a general overview of the algorithm and the major results and refer to the original paper for low level aspects.

The algorithms presented in the previous section worked by modifying a relation until it reached the equivalence relation on states $\equiv_A$ we were looking for. A dual interpretation is to look at the equivalence classes induced by the calculated relation. We start with the partition $\mathcal{P}$ of the automaton and refine it until we reach a partition consisting of the equivalence classes of $\equiv_A$. This is exactly what Hopcroft's algorithm does.

A central operation in this algorithm is *splitting* a set of states with respect to another set. Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a $\mathrm{DP}_k\mathrm{A}$, $X, Y \subseteq Q$ state sets, and $a \in \Sigma$. Splitting X w.r.t. (Y, a) produces two sets $X_1 := \{q \in X \mid \delta(q, a) \in Y\}$ and $X_2 := \{q \in X \mid \delta(q, a) \notin Y\}$. From Lemma 3.12 we know that for all states $x_1 \in X_1$, $x_2 \in X_2$ we have $x_1 \not\equiv_A x_2$. We call X *splittable* w.r.t. (Y, a) if the X_1 and X_2 defined before are both non-empty.

Algorithm 3.3 The trivial splitting algorithm

Input: $\mathrm{DP}_k\mathrm{A}$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$
Output: A partition of Q into equivalence classes for $\equiv_A$

$\mathcal{B} := \mathcal{P}$
while $\exists B_i, B_j \in \mathcal{B},\ a \in \Sigma$ such that B_i is splittable w.r.t. (B_j, a) **do**
  Replace B_i in $\mathcal{B}$ with the two sets from splitting B_i w.r.t. (B_j, a)
end while
return $\mathcal{B}$

The notion of splitting directly leads to Algorithm 3.3 for calculating the equivalence classes of $\equiv_A$. Once the state equivalence classes are known, the construction of the minimal $\mathrm{DP}_k\mathrm{A}$ is straightforward by applying Lemma 3.13. In the remainder of this section we will first prove the correctness of this simple algorithm and then show how to transform it into a more efficient one. We begin with a central invariant.

Lemma 3.15. *For the* ***while*** *loop of Algorithm 3.3 the following invariant holds:*

$$\text{all equivalent states } q \equiv_A r \text{ are in the same component of } \mathcal{B} \tag{3.1}$$

Proof. The invariant is obviously true in the beginning, as states being in different components of $\mathcal{P}$ cannot be equivalent. Now assume we are splitting B_i w.r.t. to (B_j, a) and let $q, r \in B_i$ be equivalent states. Then we know from Lemma 3.12 that $\delta(q, a) \equiv_A \delta(r, a)$, so by the invariant both are in the same component of $\mathcal{B}$ meaning either both $\delta(q, a)$ and $\delta(r, a)$ are in B_j or none of them. Thus splitting does not separate q and r. □

Using this invariant all that remains to be shown is splitting to be sufficient for separating all non-equivalent states.

Lemma 3.16. *In Algorithm 3.3 let $B_k \in \mathcal{B}$ and $q, r \in B_k$ with $q \not\equiv_A r$. Then there are sets $B_i, B_j \in \mathcal{B}$ and $a \in \Sigma$ such that B_i is splittable w.r.t. (B_j, a).*

Proof. Assume there where no such B_i, B_j, and a. Then for every two states s and t in the same component of $\mathcal{B}$ and $c \in \Sigma$ we could conclude $\delta(s, c)$ and $\delta(t, c)$ to be in the same component of $\mathcal{B}$ either, as otherwise the component containing s, t would be splittable w.r.t. the component containing one of $\delta(s, c)$ and $\delta(t, c)$ using c.

From $q \not\equiv_A r$ we know that there is a word $w \in \Sigma^*$ with $\chi_{\mathcal{P}}(\hat{\delta}(q, w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(r, w))$. From our assumption we inductively know that for each prefix v of w both $\hat{\delta}(q, v)$ and $\hat{\delta}(r, v)$ are in the same component of $\mathcal{B}$ and thus by how $\mathcal{B}$ was initialized we gain $\chi_{\mathcal{P}}(\hat{\delta}(q, w)) = \chi_{\mathcal{P}}(\hat{\delta}(r, w))$, a contradiction. □

Invariant 3.1 together with this lemma shows that the algorithm produces the correct result *if* it terminates. Obviously the algorithm has to terminate, as every iteration of the **while** loop increases the size of $\mathcal{B}$ by one and $\mathcal{B}$ cannot have more components as there are states in A.

To transform the algorithm into a more efficient one there are two key observations. One is that once a set B_i has been used together with a letter $a \in \Sigma$ to split *all* other components of $\mathcal{B}$, during the remaining iterations of the algorithm there will never again be a (new) component of $\mathcal{B}$ splittable w.r.t. B_i and a. The second observation is captured in the next lemma.

Lemma 3.17 ([Gri73], Lemma 6). *Let B be a state set split into B_1 and B_2, and $a \in \Sigma$. Then splitting all components of $\mathcal{B}$ w.r.t. all three of B, B_1, B_2 and a has the same outcome as splitting w.r.t. only two of them and a.*

We will not give a formal proof for this here, but the general idea is that from any two of B, B_1, B_2 we get the third either from a union or a set difference. So from a membership test for two of those sets, we can decide membership for the third set. As splitting only involves testing membership, splitting w.r.t. to two of these sets is sufficient.

The improved algorithm now manages a list L of (state set, letter) pairs that still have to be used for splitting all other components of $\mathcal{B}$. Whenever a state set B has been split into B_1 and B_2, we update L. If for a fixed $a \in \Sigma$ the set B still is in L we know from Lemma 3.17 that we can replace (B, a) by (B_1, a) and (B_2, a) without losing information. If $(B, a) \notin L$ then splitting w.r.t. (B, a) must already have occurred, so it suffices to insert only one of (B_1, a) and (B_2, a) into L.

Using all the modifications described so far results in Algorithm 3.4 which is exactly the Hopcroft algorithm adapted for DP_kAs. The only change to

the original algorithm is that we now have k instead of two initial state sets and accordingly we put all of them (for each $a \in \Sigma$) into L (first three lines).

After everything written so far, the algorithm contains little surprise. The only additions not already discussed are the set D which keeps all elements that might be moved when splitting w.r.t. (B_i, a), and the array t used to link a component with its split off *twin component* during a single iteration.

The correctness of Algorithm 3.4 directly results from the correctness of Algorithm 3.3 together with the discussion so far. What remains to be analyzed is the complexity of the algorithm. A not so complicated but substantial part of this analysis consists of checking that most of the operations (even those looking more complicated, such as checking *if there is a* $p \in B_j$ *with* $\delta(p, a) \notin B_i$) can be performed in constant time by managing some additional information. As this is more of an implementation issue both [Hop71] and [Gri73] provide source code (in Algol respectively PL/I) for the exact data structures used. So we refer the reader to these papers when looking for low level details. Instead we will reprove the only lemma from [Gri73] that is influenced by the changes we made to the original algorithm. For the remainder let $n := |Q|$ and $m := |\Sigma|$.

Lemma 3.18 (corresponds to [Gri73], Lemma 8). *The outer* ***while*** *loop in Algorithm 3.4 has at most* $2mn$ *iterations for* A.

Proof. Organize the state sets (we are talking about the sets and not about indices here) occurring during the execution of the algorithm in k trees as follows. The initial sets $B_1, \ldots, B_k$ form the roots of these trees. Each time a state set B is split into $\hat{B}$ and $\tilde{B}$ append $\hat{B}$ and $\tilde{B}$ as children of B. Each of these trees is a binary tree with at most $|B_i|$ leaf nodes, where B_i is the root state set of the i-th tree. So each of these trees has no more than $2|B_i| - 1$ nodes and as the $|B_i|$ sum up to $|Q|$ there will be less than $2|Q|$ different state sets used in the algorithm. As each state set can be inserted into L at most once for every letter from Σ, the number of iterations for the **while** loop is bounded by $2mn$. □

The remaining lemmas and proofs from [Gri73] can be copied verbatim as they are not influenced by the different initialization of L. We try to provide an intuitive understanding for the $O(n \log n)$ time bound here, but refer the reader to the original articles for the formalisms.

Besides bounding the number of iterations of the outer **while** loop as in Lemma 3.18 we want to get an upper bound for the number of elements placed into D over the complete execution time of the algorithm. Therefore we look at fixed $q \in Q$ and $a \in \Sigma$. How often will we retrieve a pair (i, a) from L with $q \in B_i$? After such a pair (i, a) was removed from L, a suitable pair (j, a) can only be added to L as a result of splitting. As after splitting we always add the smaller of the created state sets to L, we know that such

Algorithm 3.4 The Hopcroft algorithm for $\mathrm{DP}_k\mathrm{A}$ minimization

Input: $\mathrm{DP}_k\mathrm{A}$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$
Output: A partition of Q into equivalence classes for $\equiv_A$

$(B_1, \ldots, B_k) := \mathcal{P}$
$L := \{(i, a) \mid 1 \leq i \leq k \ \wedge \ a \in \Sigma\}$
$r := k$

while $L \neq \emptyset$ **do**
  remove a pair (i, a) from L
  $D := \{q \in Q \mid \delta(q, a) \in B_i\}$

  for each $q \in D$ **do**
    let B_j be the component containing q
    if there is a $p \in B_j$ with $\delta(p, a) \notin B_i$ **then**
      if $t[j]$ is not initialized **then**
        $r := r + 1$
        $t[j] := r$
        $B_{t[j]} := \emptyset$
      end if
      move q from B_j to $B_{t[j]}$
    end if
  end for

  for each j where $t[j]$ is initialized **do**
    for each $a \in \Sigma$ **do**
      if $(j, a) \in L$ **or** $|B_j| > |B_{t[j]}|$ **then**
        insert $(t[j], a)$ into L
      else
        insert (j, a) into L
      end if
    end for
    uninitialize $t[j]$
  end for
end while

return $(B_1, \ldots, B_r)$

a pair (i, a) with $q \in B_i$ can be drawn at most $\log n$ times from L. Define $\delta^{-1}(q, a) := \{p \mid \delta(p, a) = q\}$. Then we know that the state q contributes $|\delta^{-1}(q, a)|$ elements to D each time such a pair (i, a) is extracted from L. So the number of elements placed into D during the runtime of the algorithm is

$$\sum_{a \in \Sigma} \sum_{q \in Q} \log n |\delta^{-1}(q, a)| \,.$$

For fixed a we know that $\sum_{q \in Q} |\delta^{-1}(q, a)| = n$ as every state has exactly one successor for the transition on a, simplifying the bound to

$$\sum_{a \in \Sigma} n \log n = mn \log n \,.$$

Thus the complete running time of the first **for each** loop can be bounded by $O(mn \log n)$. The second **for each** loop has a complete running time of $O(mn)$ by a similar argument as in Lemma 3.18. Taking all of this together yields the intended result:

Theorem 3.19 ([Hop71], [Gri73]). *For a DP_kA with alphabet Σ and n states Algorithm 3.4 calculates the partition of states in equivalence classes of $\equiv_A$ and thus a minimal DP_kA in $O(|\Sigma| n \log n)$ steps using $O(|\Sigma| n)$ memory.*

We concealed one problem in this analysis sketch. The set D is actually constructed using $D = \bigcup_{q \in B_i} \delta^{-1}(q, a)$ where the δ^{-1} lists have been calculated during initialization. Thus the time for this step depends not only on the number of states placed into D, but also on the size of B_i, as some of the $\delta^{-1}(q, a)$ can be empty. Of course it can be done in $O(mn \log n)$ steps as well and [Gri73] presents a somewhat lengthy but precise proof for this missing part, so we will not repeat it here.

3.4 Language partitions of finite order

The Hopcroft algorithm for minimizing DFAs is known for 35 years now and no faster algorithm for this problem was suggested so far. This indicates that the algorithm might be optimal, however there is proof for this assumption. For the special case of minimizing DFAs for finite languages there are algorithms working in linear time. As every such algorithm has to read the entire input these algorithms are in fact optimal. We will see later that the language partitions we are considering for the problem of IP packet classification are of finite order, so the goal for this section is deriving an algorithm for minimizing DP_kAs accepting these.

An overview and comparison of algorithms for constructing minimal DFAs accepting finite languages is given in [Dac03]. Those algorithms can be classified into *incremental* or *semi-incremental* algorithms where the words

are added one by one and after each addition the current automaton is transformed to a minimal or near minimal one, and *non-incremental* algorithms which first construct a trie (an automaton that can be constructed in a straight forward fashion, see Section 6.2) from all words and then minimize this trie in one step. As the trie is often notably larger than the resulting and all intermediate minimal automata, the overall memory consumption for the non-incremental methods is higher (by a constant, asymptotically all algorithms require a linear amount of memory). However there is a certain overhead involved in managing the (nearly) minimal automaton during construction and also the algorithms are slightly more complex, so the conclusion of [Dac03] is to stick to a non-incremental algorithm as long as memory is not an issue.

While being large, the language partitions we will construct later will easily fit into the main memory of todays computers. So our algorithm is based on the non-incremental algorithm presented by Dominique Revuz in [Rev91]. The fact exploited by this algorithm (as by most of the minimization algorithms for finite languages) is that the transition graph of a DFA accepting only a finite language is (nearly) a DAG. Before stating this formally for DP_kAs we need some additional concepts.

For a state $q \in Q$ we denote by $W(q)$ the set of all words leading to a state in a component other than the first one, *i.e.*,

$$W(q) := \{w \in \Sigma^* \mid \chi_{\mathcal{P}}(\hat{\delta}(q, w)) \neq 1\}.$$

Define the height function as the longest word in $W(q)$:

$$\mathrm{height}(q) := \sup \{|w| \mid w \in W(q)\}$$

We use the supremum instead of the maximum here as the set is not necessarily compact and $\mathrm{height}(q)$ can well be ∞ (actually $\mathrm{height}(q) = \infty \Leftrightarrow |W(q)| = \infty$). The supremum of the empty set is defined as $-\infty$ here.

Lemma 3.20. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kA, and $p, q \in Q$ with p reachable from q. Then the following holds*

1. $\mathrm{height}(p) = \infty \;\Rightarrow\; \mathrm{height}(q) = \infty$
2. $\mathrm{height}(p) = -\infty \;\Leftarrow\; \mathrm{height}(q) = -\infty$
3. $-\infty < \mathrm{height}(p) < \infty \;\Rightarrow\; \mathrm{height}(p) < \mathrm{height}(q)$

Proof. Let $w \in \Sigma^* \setminus \{\epsilon\}$ such that $\hat{\delta}(q, w) = p$. Every word $u \in W(p)$ can be prepended with w to produce a word $wu \in W(q)$. So if $\mathrm{height}(p) = \infty$ this indicates that $W(p)$ has infinitely many words. Prepending these words with w gives infinitely many words in $W(q)$, so $\mathrm{height}(q) = \infty$ and the first case is shown. Similarly if $\mathrm{height}(q) = -\infty$ then $W(q) = \emptyset$ and thus $W(p) = \emptyset$ and $\mathrm{height}(p) = -\infty$ as otherwise each word from $W(p)$ would

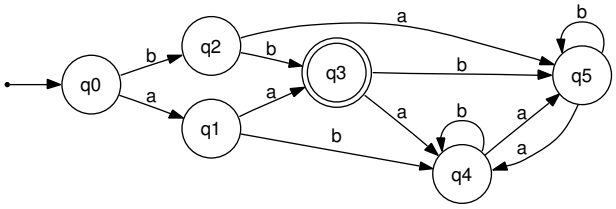


Figure 3.3: An acyclic $\mathrm{DP}_k\mathrm{A}$ accepting $\{aa, bb\}$

yield at least one in $W(q)$. For the last case assume $\text{height}(p) = h$ and let u be a word of length h with $u \in W(p)$. Then $wu \in W(q)$ and as $w \neq \epsilon$ we have $\text{height}(q) \geq |wu| > |u| = h = \text{height}(p)$. □

Earlier we stated that a $\mathrm{DP}_k\mathrm{A}$ A decides a language partition $\mathcal{L}$ of finite order iff all words that are longer than those in $\mathcal{L}$ are decided into the first component. Accordingly we call A *acyclic* if $\mathcal{L}(A) = \{L_1, \ldots, L_k\}$ and there is an $l \in \mathbb{N}$ with $L_i \cap \Sigma^{>l} = \emptyset$ for all $i \geq 2$. This is obviously equivalent to $\text{height}(q_0) < \infty$. The term acyclic is the analogon to "accepts a finite language" for DFAs, and the reason for this name will become clear from the next lemma. An example of an acyclic $\mathrm{DP}_k\mathrm{A}$ is given in Figure 3.3. The height values for q_0 to q_5 are $2, 1, 1, 0, -\infty, -\infty$.

Lemma 3.21. *Let $A = (\Sigma, Q, q_0, \mathcal{P} = (P_1, \ldots, P_k), \delta)$ be a $\mathrm{DP}_k\mathrm{A}$ without useless states. Then A is acyclic iff every $q \in Q$ that is not acyclic has* $\text{height}(q) = -\infty$.

Proof. For the first direction let A be acyclic. Assume q is a state that is not acyclic and has $\text{height}(q) > -\infty$. Then there are words $v, w \in \Sigma^*$ with $v \neq \epsilon$ such that $\hat{\delta}(q, v) = q$ and $\chi_{\mathcal{P}}(\hat{\delta}(q, w)) \neq 1$. But then for every $i \geq 0$ we have $\hat{\delta}(q, v^i w) = \hat{\delta}(q, w)$ and thus $\text{height}(q) = \infty$. As no state is useless, q is reachable from q_0 and Lemma 3.20 gives $\text{height}(q_0) = \infty$, a contradiction.

To prove the reverse direction let A be not acyclic. Then $W(q_0)$ has infinitely many words and as A has only a finite number of states we know there has to be a state $p \notin P_1$ such that the set $\{w \in \Sigma^* \mid \hat{\delta}(q_0, w) = p\}$ is infinite. Especially there must be some word w with $|w| > |Q|$ and $\hat{\delta}(q_0, w) = p$. As every prefix v of w yields a state by $\hat{\delta}(q_0, v)$ from the pigeonhole principle we know that there are prefixes x and xy of w with $|y| > 0$ and $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, xy)$. This implies $\hat{\delta}(\hat{\delta}(q_0, x), y) = \hat{\delta}(q_0, x)$ and furthermore p must be reachable from $\hat{\delta}(q_0, x)$. So $\hat{\delta}(q_0, x)$ is both cyclic and has height $> -\infty$ completing the proof. □

The next two lemmas give an idea in what way the height function can be useful for minimizing $\mathrm{DP}_k\mathrm{A}$s. Both are obvious by looking at the set $W(q)$ respectively its size for the states involved, so a proof is omitted.

Lemma 3.22. *Let A be an acyclic DP_kA. Then all states with* height $= -\infty$ *are equivalent for $\equiv_A$.*

Lemma 3.23. *Let A be an acyclic DP_kA, p, q states of A with* height$(p) \neq$ height(q)*. Then $p \not\equiv_A q$.*

So far we have not talked about how to calculate the height function for a given DP_kA, which can actually be done by a modified depth first search. The function *calc_height* shown in Algorithm 3.5 is initially called with parameters (A, q_0) and puts the height values into the array h. The correctness is proven next.

Algorithm 3.5 The calc_height function

Input: acyclic DP_kA $A = (\Sigma, Q, q_0, \mathcal{P} = (P_1, \ldots, P_k), \delta)$, a state $q \in Q$
Global: array *visited* initially **false**, array h

visited$[q]$:= **true**
if $q \in P_1$ **then**
  $h[q] := -\infty$
else
  $h[q] := 0$
end if

for each $a \in \Sigma$ **do**
  if visited$[\delta(q, a)]$ = **false** **then**
    calc_height$(A, \delta(q, a))$
  end if
  $h[q] := \max\{h[q],\ h[\delta(q, a)] + 1\}$
end for

Lemma 3.24. *Let $A = (\Sigma, Q, q_0, \mathcal{P} = (P_1, \ldots, P_k), \delta)$ be an acyclic DP_kA without useless states. Then calling calc_height(Q, q_0) correctly calculates the value of the* height *function in $h[q]$ for each state q in linear time.*

Proof. Let q be a state with height$(q) = -\infty$. Then we know that every state p reachable from q is in P_1 and also has height$(p) = -\infty$. In the algorithm h will be initialized to $-\infty$ for q and will never be increased, as this would require a state not in P_1 reachable from q, thus $h[q] = -\infty$.

Now assume the algorithm did not work, so there is a state q such that $h[q] \neq$ height(q). From the previous paragraph we know that height$(q) > -\infty$ and thus q is acyclic. Without loss of generality we may assume that q is the *deepest* erroneous state, *i.e.*, every successor of q has the correct height in h, as otherwise we could pick this successor (this picking process terminates, as q is acyclic). So for each $a \in \Sigma$ we know that $h[\delta(q, a)]$ is

correct. The set $W(q)$ contains ϵ iff $q \notin P_1$ and additionally for each $a \in \Sigma$ all words from $\{aw \mid w \in W(\delta(q,a))\}$, nothing more and nothing less. Thus $\text{height}(q) = \max\{\text{height}(\delta(q,a)) + 1 \mid a \in \Sigma\} \cup E$ where E is $\emptyset$ if $q \in P_1$ and $E = \{0\}$ otherwise. But this is exactly how $h[q]$ is calculated and so $h[q] = \text{height}(q)$.

The linear running time is obvious as no state is visited twice due to the way the *visited* array is managed. □

The actual minimization algorithm works by walking the states of a DP_kA in layers defined by the height function and performing merging of states only within these layers. The pseudo code is provided in Algorithm 3.6.

Algorithm 3.6 An algorithm for minimizing acyclic DP_kAs

Input: acyclic DP_kA $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ without useless states and $\Sigma = \{a_1, \ldots, a_s\}$

```
merge all states in {q ∈ Q | height(q) = −∞}
for h := 0 to height(q0) do
  H(h) := {q ∈ Q | height(q) = h}
  for each q ∈ H(h) do
    f(q) := (χ_P(q), δ(q, a1), ..., δ(q, as))
  end for
  merge all p, q ∈ H(h) with f(p) = f(q)
end for
```

Theorem 3.25. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be an acyclic DP_kA without useless states. Then Algorithm 3.6 constructs the minimal equivalent DP_kA for A.*

Proof. The first step of the algorithm is correct according to Lemma 3.22. In the outer **for** loop the following invariant holds:

$$\text{height}(q) < h \;\Rightarrow\; \forall p \in Q \setminus \{q\} : p \not\equiv_A q$$

Obviously this holds in the beginning. To show that it remains true after every iteration we have to see that merging states of the same height does not change the height of the resulting state (which is obvious) and that for finding equivalent states it is sufficient to compare the tuples $f(q)$. For the latter we know from Lemma 3.23 that it is in fact sufficient to search for equivalent states within $H(h)$. Additionally $f(p) \neq f(q)$ implies $p \not\equiv_A q$ as either $\chi_{\mathcal{P}}(p) \neq \chi_{\mathcal{P}}(q)$ or for some $a \in \Sigma$ it is $\delta(p,a) \neq \delta(q,a)$ which results in $\delta(p,a) \not\equiv_A \delta(q,a)$ as both $\text{height}(\delta(p,a)) < h$ and $\text{height}(\delta(q,a)) < h$ (Lemma 3.20) and due to the invariant. On the other hand $f(p) = f(q)$ obviously yields $p \equiv_A q$. From Lemma 3.20 we know that q_0 is the state

with maximal height and thus in the end the invariant holds for all $q \in Q$ proving minimality. □

For the running time of the algorithm it is crucial how to find states p, q with $f(p) = f(q)$. Let $n := |Q|$, $m := |\Sigma|$, and assume we could find all these pairs in time $O(|\Sigma||H(h)|$ then the overall running time of the algorithm would be

$$k_1 mn + \sum_{h=0}^{\text{height}(q_0)} k_2 m|H(h)| = k_1 mn + k_2 mn = O(mn)\,,$$

with k_1, k_2 being constants. This is what we initially promised, so we have to discuss how to find these pairs fast enough. The method proposed by Revuz is to sort the states $q \in H(h)$ by $f(q)$, then finding all states to be merged can be trivially achieved in linear time. Obviously any comparison based sorting algorithm would be too slow, needing at least $O(n \log n)$ steps. In [Rev91] a lexicographic sorting algorithm using bucket sort and the so called *left-right paradigm* is implemented. We will slightly deviate from this and use a relabeling technique instead of the left-right paradigm as it yields the same result and is easier to comprehend.

Pseudo code for bucket sort is given in Algorithm 3.7. It slightly differs from the one presented in [CLRS01] in that we use integer buckets (instead of intervals dividing $[0, 1]$) and the buckets are small enough that the sorting can be accomplished in one step. It should be obvious that the algorithm correctly sorts the elements $q_1, \ldots, q_n$ according to their keys in $O(m + n)$ steps. Furthermore the sorting is stable, *i.e.*, identical elements do not change their relative order. Detailed proofs can be found in [CLRS01].

Algorithm 3.7 The bucket sort algorithm

Input: integer m, items $q_1, \ldots, q_n$
keys $k_1, \ldots, k_n \in \{1, \ldots, m\}$
Output: the q_i in order of increasing k_i

initialize FIFO queues $Q_1, \ldots, Q_m$
for $i := 1$ **to** n **do**
  push q_i into Q_{k_i}
end for
return the concatenation of $Q_1, \ldots, Q_m$

The problem we are facing when using bucket sort, is that the number of keys is $O(n)$ (*i.e.*, all states) which slows down the sorting process too much. One solution is to relabel the relevant states before bucket sorting as shown in Algorithm 3.8.

Algorithm 3.8 An algorithm for sorting states by $f(q)$

Input: acyclic $\mathrm{DP}_k\mathrm{A}$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$
and set $H(h)$ as in Algorithm 3.6

use bucket sort to sort $H(h)$ with keys $\chi_{\mathcal{P}}(q)$
for each $a \in \Sigma$ **do**
  $i := 1$
  for each $q \in H(h)$ **do**
    label$[\delta(q, a)] := i$
    $i := i + 1$
  end for
  use bucket sort to sort $H(h)$ with keys label$[\delta(q, a)]$
end for

Lemma 3.26. *Algorithm 3.8 correctly sorts the state set $H(h)$ such that states with the same $f(q)$ are consecutive in $O(|\Sigma|\,|H(h)|)$ steps.*

Proof. Assume the algorithm does not work, so we have states p, q, r with $f(p) = f(q) \neq f(r)$ and in the output sequence $p < r < q$. Assume $f(q)$ and $f(r)$ are different in the i-th component. Then after sorting for this component (either for $\mathcal{P}$ or some $a \in \Sigma$) we must have either the order p, q, r or r, p, q. The state r cannot be moved between p and q by a later sorting step, as the sorting is stable and thus either the relative order is not influenced (if the tuples are equal for this component) or again r is sorted outside of p and q. Of course this is only true, if the labels are managed correctly, *i.e.*, all "relevant" states $\{\delta(q, a) \mid q \in H(h)\}$ have different labels. But this is obviously true from how the labels were assigned, so the algorithm is correct.

For the running time we note that every iteration of the outer **for each** loop requires $O(|H(h)|)$ steps, as the labels used for bucket sorting are between 1 and $|H(h)|$. Thus the entire loop can be evaluated in $O(|\Sigma|\,|H(h)|)$. The only problem is the first line, which requires time $O(k + |H(h)|$. By using a similar labeling procedure we can ensure that also this step is bounded by $O(|H(h)|)$, completing this proof. □

Corollary 3.27. *Algorithm 3.6 can be implemented to minimize an acyclic DP_kA with n states having an alphabet of size m in time $O(mn)$.*

A slightly different solution for differentiating the states in $H(h)$ is given in [Dac03]. There all states from $H(h)$ are put into a hash table (called *register*) using the keys $f(q)$. States with the same key are hashed into the same position and finding equivalent states is then performed within states put into the same hash table position. Using a suitable hash function or a hash table big enough also yields an expected overall running time of $O(mn)$.

Chapter 4

Cover automata

Cover automata provide a representation for finite languages that is usually much more compact than that of a finite automaton. The idea is to split the task of deciding a finite language L into two separate steps. One is checking membership with a *cover language* of L that possibly contains additional words longer than $\max_{w \in L} |w|$. The other is a length check that can usually be performed easier than with an automaton or is even implicitly given by the application.

While concepts similar to cover automata have been studied before, they were explicitly introduced by [CSY01].

Definition 4.1 ([CSY01]). *Let Σ be an alphabet and $L \subseteq \Sigma^*$ a finite language. Set $l := \max_{w \in L} |w|$. A DFA A is called a* deterministic finite cover automaton (DFCA) for L *iff $L(A) \cap \Sigma^{\leq l} = L$.*

As with DFAs we will look at a generalization that can be used to decide partitions of $\Sigma^{\leq l}$.

Definition 4.2. *Let Σ be an alphabet, $l \in \mathbb{N}$, and $\mathcal{L}$ a partition of $\Sigma^{\leq l}$. A DP$_k$A A is called a* deterministic k-partition l-cover automaton (DP$_k$C$_l$A) for $\mathcal{L}$ *iff $\mathcal{L}(A)|_{\Sigma^{\leq l}} = \mathcal{L}$. The automaton A is said to* cover $\mathcal{L}$.

The goal for the following sections will again be to find a minimal DP$_k$C$_l$A for a given partition $\mathcal{L}$ of order l. But before heading to the algorithms we develop some theory on the structure of cover automata. The presentation in the next few sections roughly follows [CGH05]. We start by discussing similarity relations which replace the equivalence relations used for DP$_k$As. Especially we deal with minimal similarity partitions. Then we extend the results to right invariant similarity partitions which are in turn used to carry minimality results to DP$_k$C$_l$As. After this we leave the trails of above paper and have a deeper look at the size of minimal DP$_k$C$_l$As. Finally we use these results to adapt the minimization algorithms from [CPY02] and [Kör03] towards the context of DP$_k$C$_l$As.

4.1 Similarity relations

With DP_kAs it turns out that studying an equivalence relation on states is the key for characterizing minimal DP_kAs. For $\mathrm{DP}_k\mathrm{C}_l$As so called similarity relations will take this role.

Let Σ be an alphabet. A relation $\sim$ over Σ^* is called *semi-transitive* iff for all $x, y, z \in \Sigma^*$ with $|x| \leq |y| \leq |z|$ the following holds:

1. $x \sim y \wedge y \sim z \Rightarrow x \sim z$
2. $x \sim y \wedge x \sim z \Rightarrow y \sim z$

A relation that is reflexive, symmetric, and semi-transitive is a *similarity relation*. An example of a similarity relation that is not transitive will be given later (Example 4.11).

For a similarity relation $\sim$ we call a $\sim$-complete set *similarity set*, a $\sim$-independent set *dissimilarity set*, and a $\sim$-complete partition is called a *similarity partition*.

As shown in the following lemma, shortest words are important when dealing with similarity relations and similarity sets.

Lemma 4.3 ([CGH05], Lemma 1). *Let $\sim$ be a similarity relation over Σ^*, furthermore S, T non-empty similarity sets, and s_0 and t_0 any shortest word in S respectively T. Then $S \cup T$ is a similarity set iff $s_0 \sim t_0$.*

Proof. If $S \cup T$ is a similarity set, then by definition $s_0 \sim t_0$.

For the reverse case choose any two $s' \in S$ and $t' \in T$. We may assume that $|s_0| \leq |t_0|$ and from the choice of s_0 and t_0 we have $|s_0| \leq |s'|$ and $|t_0| \leq |t'|$. This gives $|s_0| \leq |t_0| \leq |t'|$ and with $s_0 \sim t_0$ and $t_0 \sim t'$ semi-transitivity implies $s_0 \sim t'$. So we have $s_0 \sim t'$ and $s_0 \sim s'$ and either $|s_0| \leq |s'| \leq |t'|$ or $|s_0| \leq |t'| \leq |s'|$ which in any case yields $s' \sim t'$. As s' and t' were chosen arbitrarily, $S \cup T$ is a similarity set. □

Motivated by this lemma, we need a notion for shortest words in a similarity partition. Let $\sim$ be a similarity relation over Σ^*, $\mathcal{P} = (P_1, \ldots, P_k)$ a finite similarity k-partition of Σ^*. The set $C := \{c_1, \ldots, c_k\}$ is called a *minimality cross-section of $\mathcal{P}$* iff

$$\forall 1 \leq i \leq k : \; c_i \in P_i \; \wedge \; |c_i| = \min_{w \in P_i} |w| \,.$$

Obviously for every minimality cross-section C we have $\epsilon \in C$. Using the minimality cross-section we can formulate a simple criterion for checking the minimality of a similarity partition.

Lemma 4.4. *Let $\sim$ be a similarity relation over Σ^*, $\mathcal{P} = (P_1, \ldots, P_k)$ a finite similarity k-partition of Σ^*.*

If there exists *a minimality cross-section of $\mathcal{P}$ that is a dissimilarity set, then $\mathcal{P}$ is minimal,* i.e.*, there is no similarity $(k-1)$-partition.*

If $\mathcal{P}$ is minimal, then every *minimality cross-section is a dissimilarity set.*

Proof. If any minimality cross-section of $\mathcal{P}$ is a dissimilarity set, we have found a dissimilarity set of size k and according to Corollary 2.7 there can be no similarity $(k-1)$-partition.

For the reverse case let $C = \{c_1, \ldots, c_k\}$ be a minimality cross-section of $\mathcal{P}$. For the sake of contradiction assume that C is no dissimilarity set, so there are indices $1 \leq i_1, i_2 \leq k$ with $i_1 \neq i_2$ and $c_{i_1} \sim c_{i_2}$. But then by Lemma 4.3 the set $P_{i_1} \cup P_{i_2}$ is a similarity set and merging the components i_1 and i_2 in $\mathcal{P}$ yields a similarity $(k-1)$-partition violating the minimality of $\mathcal{P}$. □

The following corollary improves upon the results from Corollary 2.7.

Corollary 4.5. *Let Σ be an alphabet, $\sim$ a similarity relation over Σ^* with finite index. Define $\mathcal{D}$ the family of all dissimilarity sets, $\mathfrak{P}$ the family of all similarity partitions of Σ^*. Then*

$$\max_{D \in \mathcal{D}} |D| = \min_{\mathcal{P} \in \mathfrak{P}} |\mathcal{P}| .$$

Proof. From Corollary 2.7 we already know that $\max_{D \in \mathcal{D}} |D| \leq \min_{\mathcal{P} \in \mathfrak{P}} |\mathcal{P}|$, so it suffices to show that from a minimal similarity partition one can construct a dissimilarity set of the same size. But this is what Lemma 4.4 showed, namely the minimality cross-section. □

An interesting property of a minimality cross-section of a minimal similarity partition we will be using later, is that the words in the minimality cross-section are not only the shortest within their component but also shorter that any other word similar to them.

Lemma 4.6. *Let $\sim$ be a similarity relation over Σ^*, $\mathcal{P} = (P_1, \ldots, P_k)$ a minimal finite similarity k-partition of Σ^*, $C = (c_1, \ldots, c_k)$ a minimality cross-section of $\mathcal{P}$, and $c_{i_0} \in C$. Then $|w| \geq |c_{i_0}|$ for every word $w \sim c_{i_0}$.*

Proof. Let $w \in \Sigma^*$ with $w \sim c_{i_0}$ and set $i_w := \chi_{\mathcal{P}}(w)$. If $i_w = i_0$ the claim is true due to the minimality of $|c_{i_0}|$ within P_{i_0}. So let $i_w \neq i_0$ and assume $|w| < |c_{i_0}|$. Then $|c_{i_w}| \leq |w| < |c_{i_0}|$ and $c_{i_w} \sim w$ and $w \sim c_{i_0}$. But this implies $c_{i_w} \sim c_{i_0}$ by semi-transitivity, which is by Lemma 4.4 a contradiction to the minimality of $\mathcal{P}$. □

4.2 Right invariant similarity partitions

We have seen that there is no gap between the size of a maximal dissimilarity set and a minimal similarity partition. The next step will be to show that between minimal similarity partitions and minimal *right invariant* similarity partitions there is no size gap neither. For this we need a rather technical lemma.

Lemma 4.7. *Let $\sim$ be a right invariant similarity relation over Σ^*, $\mathcal{P} = (P_1, \ldots, P_k)$ a minimal finite similarity partition of Σ^*, $C = \{c_1, \ldots, c_k\}$ a minimality cross-section of $\mathcal{P}$. Define a function $\delta : C \times \Sigma \to C$ by choosing $\delta(c_i, a) \in C$ such that*

$$\forall c_i \in C \, \forall a \in \Sigma : \delta(c_i, a) \sim c_i a \, .$$

Then for every word $w \in \Sigma^*$ the following holds for the canonical expansion of δ:

$$\hat{\delta}(\epsilon, w) \sim w$$

Proof. First we should verify that δ can indeed be defined this way, *i.e.*, for every word $w \in \Sigma^*$ there is a $c_i \in C$ with $c_i \sim w$. But as $\mathcal{P}$ is a similarity partition we know $c_{\chi_{\mathcal{P}}(w)} \sim w$.

We will prove this by induction on the length of the word w. The base case is trivial as $\hat{\delta}(\epsilon, \epsilon) = \epsilon \sim \epsilon$. Now let $a \in \Sigma$, $w \in \Sigma^*$, set $\hat{c} := \hat{\delta}(\epsilon, w)$, and assume the induction hypothesis $\hat{c} \sim w$ holds. We will show $\hat{\delta}(\epsilon, wa) \sim wa$. From above definition $\hat{\delta}(\epsilon, wa) = \delta(\hat{\delta}(\epsilon, w), a) = \delta(\hat{c}, a) \sim \hat{c}a$. As $\sim$ is right invariant from $\hat{c} \sim w$ we receive $\hat{c}a \sim wa$. From Lemma 4.6 we know that since $\hat{\delta}(\epsilon, wa) \in C$ it is $|\hat{\delta}(\epsilon, wa)| \leq |\hat{c}a|$ and also $|\hat{c}| \leq |w|$, thus $|\hat{c}a| \leq |wa|$. But these are the prerequisites for semi-transitivity and we conclude $\hat{\delta}(\epsilon, wa) \sim wa$ completing this induction proof. □

Using this lemma we can construct a minimal right invariant similarity partition from any minimal similarity partition as show next.

Theorem 4.8. *Let $\sim$ be a right invariant similarity relation over Σ^*, $\mathcal{P} = (P_1, \ldots, P_k)$ a minimal finite similarity partition of Σ^*. Then there is a similarity k-partition $\mathcal{Q}$ that is right invariant.*

Proof. Let $C = \{c_1, \ldots, c_k\}$ be a minimality cross-section of $\mathcal{P}$ and δ the function from Lemma 4.7. For each $1 \leq i \leq k$ define

$$Q_i := \{w \in \Sigma^* \mid \hat{\delta}(\epsilon, w) = c_i\}$$

and $\mathcal{Q} := (Q_1, \ldots, Q_k)$. We claim that $\mathcal{Q}$ is both a similarity k-partition and right invariant.

For every $c \in C$ we know that $\hat{\delta}(\epsilon, c) = c$ as otherwise (from the mentioned lemma we have $\hat{\delta}(\epsilon, c) \sim c$) the set C was no dissimilarity set. So all

Q_i are non-empty and as $\hat{\delta}$ is a function, *i.e.*, every pair (ϵ, w) is mapped to exactly one element from C, the sequence $\mathcal{Q}$ is a k-partition. Additionally for two words $x, y \in Q_i$ we know that $c_i \sim x$ and $c_i \sim y$ and from Lemma 4.6 either $|c_i| \leq |x| \leq |y|$ or $|c_i| \leq |y| \leq |x|$ and so from semi-transitivity $x \sim y$. Thus each Q_i is a similarity set and $\mathcal{Q}$ is a similarity partition.

It remains to prove the right invariance of $\mathcal{Q}$. We will apply Lemma 2.2 here, so let $a \in \Sigma$, $1 \leq i_0 \leq k$. We have to show that there is i' such that $Q_{i_0}a \subseteq Q_{i'}$. From the definition of δ we know there is i' such that $\delta(c_{i_0}, a) = c_{i'}$. To show that this is the i' we were looking for, choose any $w \in Q_{i_0}$. Then we know that $\hat{\delta}(\epsilon, w) = c_{i_0}$ and $\hat{\delta}(\epsilon, wa) = \delta(\hat{\delta}(\epsilon, w), a) = \delta(c_{i_0}, a) = c_{i'}$ thus $wa \in Q_{i'}$ and we are done. □

4.3 Minimal DP_kC_lAs

The tool for stating the Myhill-Nerode theorem for DFAs was the introduction of a suitable equivalence relation. Similarly to close the gap from right invariant similarly partitions to DP_kC_lAs we just have to define a suitable similarly relation.

Definition 4.9. *Let Σ be an alphabet, $l \in \mathbb{N}$, and $\mathcal{L}$ a partition of $\Sigma^{\leq l}$. Define the relation $\sim_{\mathcal{L}} \subseteq \Sigma^* \times \Sigma^*$ by*

$$x \sim_{\mathcal{L}} y \quad :\Leftrightarrow \quad \forall w \in \Sigma^* : \left((\, |xw| \leq l \wedge |yw| \leq l \,) \Rightarrow (\chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)) \right).$$

Lemma 4.10. *Let Σ be an alphabet, $l \in \mathbb{N}$, and $\mathcal{L}$ a partition of $\Sigma^{\leq l}$. The relation $\sim_{\mathcal{L}}$ is a right invariant similarity relation.*

Proof. Reflexivity and symmetry are obvious from the definition. For semi-transitivity let $x, y, z \in \Sigma^*$ with $|x| \leq |y| \leq |z|$. We rephrase the definition of $\sim_{\mathcal{L}}$ as

$$x \sim_{\mathcal{L}} y \;\Leftrightarrow\; \forall w \in \Sigma^{\leq l - \max\{|x|,|y|\}} : \chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)\,,$$

where we define for negative l the set $\Sigma^{\leq l} := \emptyset$. This is equivalent as exactly the words in $\Sigma^{\leq l - \max\{|x|,|y|\}}$ pass the condition $(\, |xw| \leq l \wedge |yw| \leq l \,)$ in the definition.

For the first part of semi-transitivity we have $x \sim_{\mathcal{L}} y$ and $y \sim_{\mathcal{L}} z$, so $\forall w \in \Sigma^{l-|y|} : \chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)$ and $\forall w \in \Sigma^{l-|z|} : \chi_{\mathcal{L}}(yw) = \chi_{\mathcal{L}}(zw)$. As $|z| \geq |y|$ we have $\Sigma^{\leq l-|z|} \subseteq \Sigma^{\leq l-|y|}$ and thus for all $w \in \Sigma^{\leq l-|z|}$ both $\chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)$ and $\chi_{\mathcal{L}}(yw) = \chi_{\mathcal{L}}(zw)$ hold. Thus by transitivity of $=$ we get $\forall w \in \Sigma^{l-|z|} : \chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw) = \chi_{\mathcal{L}}(zw)$ which is what we wanted to show: $x \sim_{\mathcal{L}} z$. The second condition for semi-transitivity is proven analogously.

Last is right invariance, so let $x \sim_{\mathcal{L}} y$ and $a \in \Sigma$. Then

$$\forall w \in \Sigma^* : \left((|xw| \leq l \wedge |yw| \leq l) \Rightarrow (\chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)) \right)$$

and as w may be any word we may substitute it with aw:

$$\forall w \in \Sigma^* : \left((|xaw| \leq l \wedge |yaw| \leq l) \Rightarrow (\chi_{\mathcal{L}}(xaw) = \chi_{\mathcal{L}}(yaw)) \right).$$

But this means $xa \sim_{\mathcal{L}} ya$ by definition. □

The following example shows that $\sim_{\mathcal{L}}$ is not necessarily transitive.

Example 4.11. *Let $\Sigma := \{a\}$, $l := 4$, and $\mathcal{L} := (L_1, L_2)$ with $L_1 := \{\epsilon, a^3\}$, $L_2 := \{a, a^2, a^4\}$. Then $\mathcal{L}$ is a partition of $\Sigma^{\leq l}$ and both $a \sim_{\mathcal{L}} a^4$ and $a^4 \sim_{\mathcal{L}} a^2$, which can be seen as the only word in $\Sigma^{l-|a^4|}$ is ϵ and all of a, a^2, a^4 are in the second component of $\mathcal{L}$. On the other hand $a \not\sim_{\mathcal{L}} a^2$, as both aa and a^2a are no longer than l but in different components.*

In the next lemma we see that the relation $\sim_{\mathcal{L}}$ was chosen "right" for our purposes.

Lemma 4.12. *Let Σ be an alphabet, $l \in \mathbb{N}$, and $\mathcal{L}$ a partition of $\Sigma^{\leq l}$. Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kC_lA without useless states covering $\mathcal{L}$. Then A induces a right invariant similarity partition in terms of $\sim_{\mathcal{L}}$ by $(L_q(A))_{q \in Q}$.*

Proof. We know from Remark 3.2 that $(L_q(A))_{q \in Q}$ is a right invariant partition. Let $q \in Q$ and $x, y \in L_q(A)$. We have to show that $x \sim_{\mathcal{L}} y$. As words longer than l are similar to all other words we may assume $|x| \leq |y| \leq l$. But as for all words w both $\hat{\delta}(q_0, xw)$ and $\hat{\delta}(q_0, yw)$ are the same, similarity is obvious. □

After all these preparations we can now state the main theorem of this section. For language partitions of finite order we know that there is always a DP_kA accepting them (although this was not formally proven it follows from the trie construction in Section 6.2). This also is a DP_kC_lA covering the finite language, so there is no need for a characterization of coverable language partitions. However the follow theorem resembles the Myhill-Nerode theorem in that it provides a different measure for the number of states in a minimal DP_kC_lA.

Theorem 4.13. *Let Σ be an alphabet, $l \in \mathbb{N}$, $\mathcal{L}$ a k-partition of $\Sigma^{\leq l}$. Every DP_kC_lA covering $\mathcal{L}$ has at least* $\text{index}(\sim_{\mathcal{L}})$ *states and there is a DP_kC_lA covering $\mathcal{L}$ with exactly* $\text{index}(\sim_{\mathcal{L}})$ *states.*

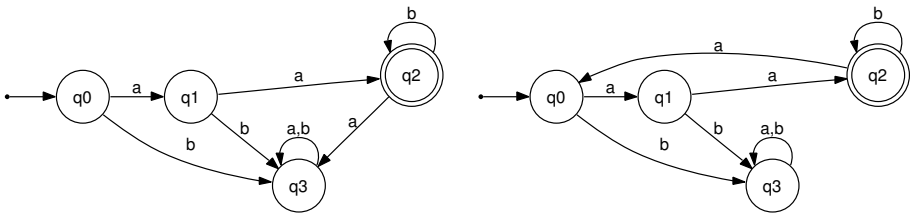


Figure 4.1: Two non-isomorphic minimal $\mathrm{DP}_2\mathrm{C}_4\mathrm{As}$

Proof. As a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ induces a similarity relation by its state languages (Lemma 4.12) it cannot have less than index($\sim_{\mathcal{L}}$) states (Lemma 2.6).

Now for the existence of a suitable $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$. Theorem 4.8 guarantees the existence of a right invariant similarity partition of size index($\sim_{\mathcal{L}}$), which we call $\mathcal{Q}$. The $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ $A := (\Sigma, \mathcal{Q}, \chi_{\mathcal{Q}}(\epsilon), \mathcal{P}, \delta)$ then has the same size and, as we will see, decides $\mathcal{L}$. We have to say first how to choose $\mathcal{P}$ and δ. As $\mathcal{Q}$ is right invariant we may define $\delta(\chi_{\mathcal{Q}}(w), a) := \chi_{\mathcal{Q}}(wa)$ without violating δ being a function. This choice enforces $L_Q(A) = Q$ for all $Q \in \mathcal{Q}$. It remains to show that $\mathcal{Q}|_{\Sigma^{\leq l}}$ is a refinement of $\mathcal{L}$ so we can choose $\mathcal{P}$ accordingly. So let $Q \in \mathcal{Q}$ and $x, y \in L_Q(A) = Q$, $|x| \leq |y| \leq l$. But as x and y are similar (both are in Q) and not longer than l they are by definition in the same component of $\mathcal{L}$. □

This theorem directly yields an easy verifiable condition for checking the minimality of a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$.

Corollary 4.14. *A DP_kC_lA $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ covering $\mathcal{L}$ is minimal iff a minimality cross-section of $(L_q(A))_{q \in Q}$ is a dissimilarity set for $\sim_{\mathcal{L}}$.*

Minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$ (other than with $\mathrm{DP}_k\mathrm{As}$) do not have to be isomorphic, as shown by the next example.

Example 4.15. *Let $\Sigma := \{a, b\}$, $l := 4$, $L := \{aa, aab, aabb\}$, and $\mathcal{L} := (\Sigma^{\leq l} \setminus L, L)$. The set $S := \{\epsilon, b, a, aa\}$ is a dissimilarity set for $\sim_{\mathcal{L}}$ as is easily checked, thus no DP_2C_4A deciding $\mathcal{L}$ can have less than 4 states. Figure 4.1 shows two minimal DP_2C_4As that are not isomorphic (which can be verified by counting the number of in-going transitions for each state) but both deciding $\mathcal{L}$.*

4.4 A lower bound on $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ size

As cover automata were introduced to reduce the size of an automaton representation of a partition, it is legitimate to question the savings achieved. We will give a lower bound on the size of a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ relative to the size of a minimal $\mathrm{DP}_k\mathrm{A}$. But before working out the details, we give an example

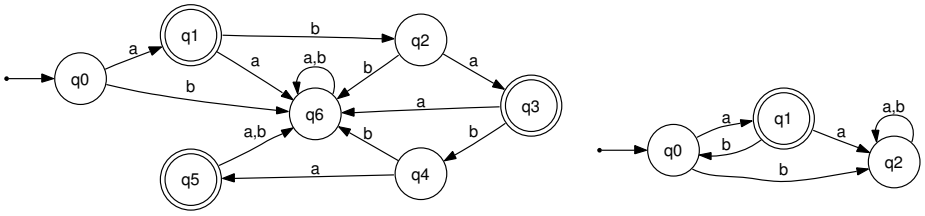


Figure 4.2: A minimal DP_2A and DP_2C_5A for Example 4.16

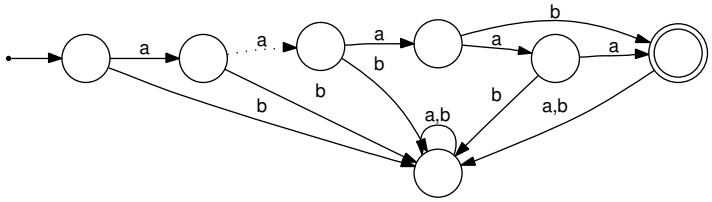


Figure 4.3: The minimal DP_2A for Example 4.18

from [Kör03] showing that the DP_kC_lA can be slightly smaller than the corresponding DP_kA.

Example 4.16. *Let $\Sigma := \{a, b\}$, $l := 5$, $L := \{a, aba, ababa\}$, and $\mathcal{L} := (\Sigma^{\leq l} \setminus L, L)$. Figure 4.2 shows both a minimal DP_2A and a minimal DP_2C_5A having 7 respectively 3 states.*

Before looking at the lower bound we mark down the trivial upper bound. There are actually cases where this bound is reached, *i.e.*, the minimal DP_kC_lA has as many states as the minimal DP_kA. A family of language partitions for which this bound is tight is presented in the example thereafter.

Remark 4.17. *Let $\mathcal{L}$ be a partition of $\Sigma^{\leq l}$. Every DP_kA deciding $\mathcal{L}$ is also a DP_kC_lA covering $\mathcal{L}$. Thus a minimal DP_kC_lA never has more states than a minimal DP_kA for the same language partition.*

Example 4.18. *Let $\Sigma := \{a, b\}$, $n \in \mathbb{N}$ arbitrarily chosen, $l := n + 2$, $L := \{a^n b, a^n aa\}$, and $\mathcal{L} := (\Sigma^{\leq l} \setminus L, L)$. The minimal DP_2A deciding $\mathcal{L}$ is shown in Figure 4.3 and has $n + 4$ states. The set $\{\epsilon, b, a, a^2, \ldots, a^n, a^{n+1}, a^{n+2}\}$ is a dissimilarity set for $\sim_{\mathcal{L}}$ of size $n + 4$. Thus the presented automaton is also a minimal DP_kC_lA. For the proof of dissimilarity first note that b is dissimilar to all a^i $(0 \leq i \leq n + 2)$ as those can be completed to a word in L, contrary to b. For $0 \leq i < j \leq n + 2$ we have $a^i \not\sim_{\mathcal{L}} a^j$ because appending the word a^{n+2-j} results in words in different components of $\mathcal{L}$.*

The tool for our lower bound proof is the automaton product known from DFAs. For DPAs it has to be slightly adjusted to handle the state partition $\mathcal{P}$ correctly.

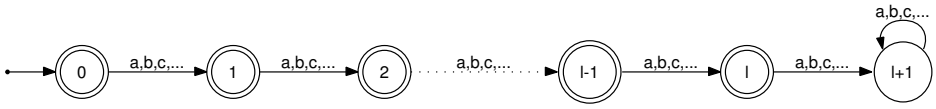


Figure 4.4: The DFA B used in Theorem 4.21

Definition 4.19. *Let $A := (\Sigma, Q_A, q_A, (P_1, \ldots, P_k), \delta_A)$ be a DP_kA and $B := (\Sigma, Q_B, q_B, F, \delta_B)$ a DFA on the same alphabet. The* product automaton *$A \times B$ is defined as the DP_kA $(\Sigma, Q_A \times Q_B, (q_A, q_B), (\hat{P}'_1, P'_2, \ldots, P'_k), \delta')$ with*

- $\delta'((q_1, q_2), a) := (\delta_A(q_1, a), \delta_B(q_2, a)) \qquad (a \in \Sigma,\ q_1 \in Q_A,\ q_2 \in Q_B)$
- $P'_i := \{(q_1, q_2) \mid q_1 \in P_i \wedge q_2 \in F\} \qquad (1 \leq i \leq k)$
- $\hat{P}'_1 := P'_1 \cup \{(q_1, q_2) \mid q_1 \in Q_A \wedge q_2 \in (Q_B \setminus F)\}$

Lemma 4.20. *Let $A := (\Sigma, Q_A, q_A, (P_1, \ldots, P_k), \delta_A)$ be a DP_kA and $B := (\Sigma, Q_B, q_B, F, \delta_B)$ a DFA on the same alphabet. Then the language partition $\mathcal{L}(A \times B) = \operatorname{ses}(\hat{L}_1, L_2, \ldots, L_k)$ where*

- $L_i := L_{P_i}(A) \cap L(B) \qquad (1 \leq i \leq k)$
- $\hat{L}_1 := L_1 \cup (\Sigma^* \setminus L(B))$

Proof. Let δ' be the transition function of $A \times B$. By induction we know that a word w is mapped by $\hat{\delta}'$ to $(\hat{\delta}_A(\epsilon, w), \hat{\delta}_B(\epsilon, w))$, so for $q_1 \in Q_A$, $q_2 \in Q_B$ we have $L_{(q_1,q_2)}(A \times B) = L_{q_1}(A) \cap L_{q_2}(B)$. This together with the state partition for $A \times B$ from Definition 4.19 gives exactly the $\mathcal{L}(A \times B)$ from above. □

Now we are ready to prove the main theorem of this section, showing that a DP_kC_lA covering a partition of $\Sigma^{\leq l}$ can be up to about l times smaller than a minimal DP_kA deciding the same partition.

Theorem 4.21. *Let $A = (\Sigma, Q, q_0, \mathcal{P} = (P_1, \ldots, P_k), \delta)$ be a DP_lC_kA without useless states covering the partition $\mathcal{L}$ of $\Sigma^{\leq l}$. Then there is a DP_kA with at most $l|Q| - |P_1| + 2$ states deciding $\mathcal{L}(A)$.*

Proof. Let $B := (\Sigma, \{0, \ldots, l+1\}, 0, \{0, \ldots, l\}, \delta_B)$ with $\delta_B(i, a) := \max\{i + 1, l + 1\}$ $(a \in \Sigma,\ 1 \leq i \leq l + 1)$ be a DFA (see Figure 4.4). Obviously B has $l + 2$ states and $L(B) = \Sigma^{\leq l}$. Let $C := A \times B$. From Lemma 4.20 we know that $\mathcal{L}(C) = ((L_{P_1}(A) \cap \Sigma^{\leq l}) \cup \Sigma^{>l}, L_{P_2}(A) \cap \Sigma^{\leq l}, \ldots, L_{P_k}(A) \cap \Sigma^{\leq l})$, so C decides $\mathcal{L}$. According to Definition 4.19 the automaton C has $(l+2)|Q|$ states. Due to the structure of B (no backward transitions) the states $(q, 0)$ with $q \in Q \setminus \{q_0\}$ are not reachable and the states $(q, l + 1)$ with $q \in Q$ are all in the first component of C's state partition and will not reach any

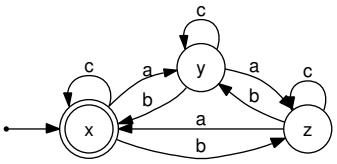


Figure 4.5: The minimal DP_2C_4A for the language M

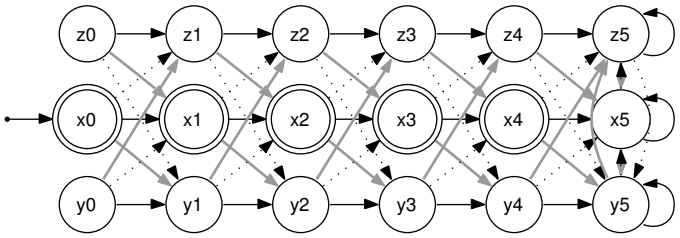


Figure 4.6: The DP_2C_lA from Figure 4.5 after "unrolling"

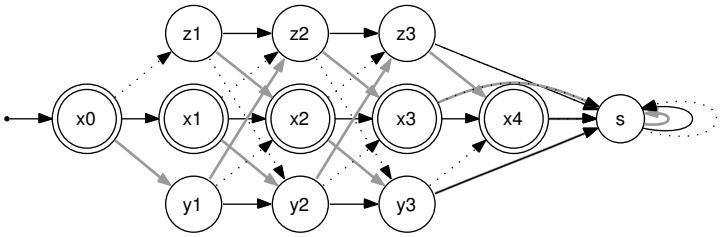


Figure 4.7: The DP_2A from Figure 4.6 after minimization

state (q, i) with $i \leq l$. Additionally all "uninteresting" states in level l ((q, l) with $q \in P_1$) can only have forward transitions to $(q, l+1)$ and so also are in the first component of C's state partition. Thus removing the $|Q| - 1$ unreachable states $(q, 0)$ and merging the $|Q| + |P_1|$ states $(q, l+1)$ and $(q \in P_1, l)$ into one state results in a DP_kA that decides $\mathcal{L}$ and has only $(l+2)|Q| - (|Q| - 1) - (|Q| + |P_1| - 1) = l|Q| - |P_1| + 2$ states. □

We conclude this section with an example showing this lower bound to be tight.

Example 4.22. *Let* $\Sigma = \{a, b, c\}$, $l \in \mathbb{N}$, $M := \{w \in \Sigma^{\leq l} \mid \#_a(w) \equiv \#_b(w) \pmod 3\}$, *and* $\mathcal{L} := (\Sigma^{\leq l} \setminus M, M)$. *Thus* M *contains words such as aaccbcaa, cabab, or aaa.*

The following steps are visualized for the case $l = 4$. *As the association of labels with edges is complicated in dense graphs, some of them represent* a, b, *and* c *transitions by bold gray, dashed, respectively plain solid lines.*

The minimal DP_2C_lA for $\mathcal{L}$ consists of 3 states (it is easy to check, that $\{a^i \mid 0 \leq i < 3\}$ is a dissimilarity set for $\sim_{\mathcal{L}}$), the automaton is shown in Figure 4.5. Building the product as in Theorem 4.21 produces the DP_2A from Figure 4.6 which has obviously $3(l+2)$ states. Removing the unreachable states (in the figure these are $y0$ and $z0$) and merging the "right outer" states (in the figure these are $y4$, $z4$, $x5$, $y5$, $z5$) into a new one (s) yields an automaton with $3(l+2)-(3-1)-(3+3-1)+1 = 3l$ states shown in Figure 4.7. This is exactly the value we wanted to accomplish. All that remains is to prove the minimality of the created graph, as otherwise the bound was not necessarily tight. For this for any two states q_1, q_2 we have to find a word w such that $\hat{\delta}(q_1, w)$ and $\hat{\delta}(q_2, w)$ are in different classes. While it is possible to check this for all possible combinations, this is a tedious task with many different cases giving little insight and so is not performed here. The reader is invited to verify the claim at least for some state pairs in Figure 4.7.

4.5 Towards a minimization algorithm

After having seen some theory on cover automata including a characterization of minimal ones and bounds that imply their usefulness, we would like to be able to construct minimal cover automata for a given partition of order l. Following we are working towards an adaption of the algorithm presented in [CPY02] for constructing minimal cover automata which has some parallels to the algorithm presented in Section 3.2. The input partition for this algorithm will be given as a $\mathrm{DP}_k\mathrm{A}$ deciding this partition, *i.e.*, all words from $\Sigma^{>l}$ are classified into the first component of the state partition.

As we have seen before, shortest words take a central position with cover automata. To simplify working with them, we first introduce some notation. Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a $\mathrm{DP}_k\mathrm{A}$. For a useful state $q \in Q$ we denote by $\mathrm{level}(q)$ the length of a shortest word in $L_q(A)$:

$$\mathrm{level}(q) := \min\{|w| \mid w \in L_q(A)\}\,.$$

If $L_q(A) = \emptyset$ define $\mathrm{level}(q) := \infty$.

Remark 4.23. *For a given DP_kA A the* level *function for all its states can be computed in linear time using breadth first search.*

Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$. For states $p, q \in Q$ define the range function as

$$\mathrm{range}(p, q) := l - \max\{\mathrm{level}(p), \mathrm{level}(q)\}\,.$$

Intuitively two states p, q can be merged as with $\mathrm{DP}_k\mathrm{A}$ minimization if they "behave the same" for all words that are not longer than $\mathrm{range}(p, q)$, which is covered by the following definition.

Definition 4.24. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kC_lA without useless states. We define the relation $\sim_A \subseteq Q \times Q$ by*

$$q \sim_A p \;:\Leftrightarrow\; \forall w \in \Sigma^{\leq \mathrm{range}(p,q)} : \chi_{\mathcal{P}}(\hat{\delta}(p,w)) = \chi_{\mathcal{P}}(\hat{\delta}(q,w))\,.$$

Actually this definition is deeply linked to the similarity relation $\sim_{\mathcal{L}(A)}$ as shown below.

Lemma 4.25. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kC_lA without useless states covering the partition $\mathcal{L}$ of $\Sigma^{\leq l}$, $p, q \in Q$, and $x_p \in L_p(A)$, $x_q \in L_q(A)$ such that $|x_p| = \mathrm{level}(p)$ and $|x_q| = \mathrm{level}(q)$. Then $p \sim_A q$ iff $x_p \sim_{\mathcal{L}} x_q$.*

Proof. Let $m := l - \max\{|x_p|, |x_q|\}$. In the proof of Lemma 4.10 we have seen that condition $x_p \sim_{\mathcal{L}} x_q$ is equivalent to

$$\forall w \in \Sigma^m : \chi_{\mathcal{L}}(x_p w) = \chi_{\mathcal{L}}(x_q w)$$

which because of $\chi_{\mathcal{L}}(w) = \chi_{\mathcal{P}}(\hat{\delta}(q_0, w))$ for $|w| \leq l$ is the same as

$$\forall w \in \Sigma^m : \chi_{\mathcal{P}}(\hat{\delta}(q_0, x_p w)) = \chi_{\mathcal{P}}(\hat{\delta}(q_0, x_q w))\,.$$

But as $\hat{\delta}(q_0, x_p w) = \hat{\delta}(\hat{\delta}(q_0, x_p), w) = \hat{\delta}(p, w)$ and analogous for x_q and q this means $p \sim_A q$. All steps were equivalence transformations, so both directions of the claim are valid. □

Corollary 4.26. *Having the preconditions from Lemma 4.25, $p \sim_A q$ iff $L_p(A) \cup L_q(A)$ is a similarity set for $\sim_{\mathcal{L}}$.*

Proof. If $L_p(A) \cup L_q(A)$ is a similarity set, then $x_p \sim_{\mathcal{L}} x_q$ and Lemma 4.25 gives $p \sim_A q$. On the other hand if $p \sim_A q$ and thus $x_p \sim_{\mathcal{L}} x_q$ then according to Lemma 4.12 and Lemma 4.3 $L_p(A) \cup L_q(A)$ is a similarity set. □

Based on this relation on states we can formulate a minimization algorithm. We will not provide information on how to calculate the relation, as this will be detailed later. The pseudo-code for DP_kC_lA minimization is provided in Algorithm 4.1.

The algorithm operates in two steps. The first phase up to the end of the **while** loop calculates a minimal $\sim_A$-complete partition of Q, the second phase then merges all states that are in the same partition into one. We will prove the correctness of both steps separately.

Lemma 4.27. *Let $A := (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kC_lA. Then $(N_r)_{r \in Q'}$ as calculated in the first part of Algorithm 4.1 is a minimal $\sim_A$-complete partition of Q.*

Algorithm 4.1 A generic minimization algorithm for $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$

Input: A $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ $A = (\Sigma, Q, q_0, \mathcal{P} = (P_1, \ldots, P_k), \delta)$, the relation $\sim_A$ from Definition 4.24
Output: A minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ covering the same language partition as A

$R := Q$
$Q' := \emptyset$
while $R \neq \emptyset$ **do**
  choose $r \in R$ such that $\mathrm{level}(r) = \min_{q \in R} \mathrm{level}(q)$
  $Q' := Q' \cup \{r\}$
  $N_r := \{q \in R \mid q \sim_A r\}$
  $R := R \setminus N_r$
end while

for $i := 1$ **to** k **do**
  $P'_i := P_i \cap Q'$
end for
for each $q' \in Q'$, $a \in \Sigma$ **do**
  {replace every state with its representative in δ}
  $\delta'(q', a) := \chi_{(N_r)_{r \in Q'}}(\delta(q', a))$
end for
return $A' := (\Sigma, Q', q_0, (P'_1, \ldots, P'_k), \delta')$

Proof. Due to the way the N_r have been chosen it is obvious that $(N_r)_{r \in Q'}$ is a $\sim_A$-complete partition of Q. It remains to prove minimality. Assume there is a $\sim_A$-complete partition $\mathcal{B}$ of Q with less than $|Q'|$ components. Then there must be $p', q' \in Q'$ that are in the same component of $\mathcal{B}$ and thus $p' \sim_A q'$. But then one of p' and q' has been drawn of R before the other and we either have $p' \in N_{q'}$ or $q' \in N_{p'}$, a contradiction. □

For the proof of the second phase we need a technical lemma showing that the way we merge states is "correct". This is where we need the representatives of each component to have minimal level within the states of the component.

Lemma 4.28. *Let $A := (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a $DP_k C_l A$ covering some language partition $\mathcal{L}$, $Q' \subseteq Q$ a set of representative states, $\mathcal{N} := (N_r)_{r \in Q'}$ a (not necessarily minimal) $\sim_A$-complete partition of Q such that for each $q' \in Q'$ the level of q' is minimal within all states of $N_{q'}$, and $A' = (\Sigma, Q', q_0, \mathcal{P}' = (P'_1, \ldots, P'_k), \delta')$ the automaton calculated from the second phase of Algorithm 4.1.*

Then A' is defined properly and for every $q' \in Q'$ and any word $w \in$

$\Sigma^{\leq l-\text{level}_A(q')}$ *we have*

$$\chi_{\mathcal{P}'}(\hat{\delta}'(q', w)) = \chi_{\mathcal{P}}(\hat{\delta}(q', w)) .$$

Proof. States from different components of $\mathcal{P}$ having level $\leq l$ cannot be in the same component of $\mathcal{N}$. Additionally for each $P_i \in \mathcal{P}$ there has to be a state $q \in P_i$ with level $\leq l$ as otherwise the i-th component of $\mathcal{L}$ would be empty (and thus $\mathcal{L}$ not a proper partition). This implies that all the P'_i are non-empty and together partition Q'. The starting state q_0 is the only state with level $= 0$, so it has to be in Q'. Consequently the definition of A' is consistent.

For the second part assume that there is some shortest word w such that there is a state $q' \in Q'$ with $w \in \Sigma^{\leq l-\text{level}_A(q')}$ and $\chi_{\mathcal{P}'}(\hat{\delta}'(q', w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q', w))$. As q' is in the same state class in both A and A' we know $w \neq \epsilon$. So let $a \in \Sigma$ and $x \in \Sigma^*$ with $ax = w$. Set $p := \delta(q', a)$, $p' := \delta'(q', a)$. From the initialization of δ' we know that $p \in N_{p'}$ and by the choice of the representative states $\text{level}_A(p') \leq \text{level}_A(p)$. Furthermore $\text{level}_A(p) \leq \text{level}_A(q') + 1$, thus

$$l - \text{level}_A(q') - 1 \leq l - \text{level}_A(p') = l - \min\{\text{level}_A(p), \text{level}_A(p')\}$$

and so $x \in \Sigma^{\leq \text{range}(p,p')}$. From $p \in N_{p'}$ we know $p \sim_A p'$ and Definition 4.24 gives $\chi_{\mathcal{P}}(\hat{\delta}(p, x)) = \chi_{\mathcal{P}}(\hat{\delta}(p', x))$. As w was chosen minimal within the "failing" words the claim of this lemma must be true for x and so $\chi_{\mathcal{P}'}(\hat{\delta}'(p', x)) = \chi_{\mathcal{P}}(\hat{\delta}(p', x))$. But now we have $\hat{\delta}(q', w) = \hat{\delta}(\delta(q', a), x) = \hat{\delta}(p, x)$ and $\hat{\delta}'(q', w) = \hat{\delta}'(p', x)$ which we have shown to be in the same state class. A contradiction to our initial assumption. □

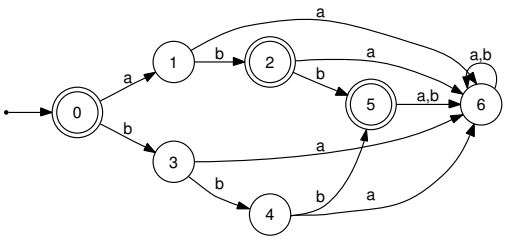
Applying this lemma we can prove the second phase to be correct, *i.e.*, the automaton produced is both minimal and covers $\mathcal{L}$. This is a major result, as all known minimization algorithms for DP_kC_lAs are relying on this phase.

Theorem 4.29. *Let $A := (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kC_lA with n states covering some language $\mathcal{L}$, $Q' \subseteq Q$ a set of representative states, $\mathcal{N} := (N_r)_{r \in Q'}$ a* minimal $\sim_A$ *complete partition of Q such that for each $q' \in Q'$ the level of q' is minimal within all states of $N_{q'}$.*

Then the automaton calculated in the second phase of Algorithm 4.1 is a minimal DP_kC_lA covering $\mathcal{L}$ and the process requires $O(|\Sigma|n)$ time and space.

Proof. From Lemma 4.28 we know that A' is a valid DP_kC_lA and, as q_0 is the only state with level $= 0$, for every $w \in \Sigma^{\leq l}$ we have $\chi_{\mathcal{P}'}(\hat{\delta}'(q_0, w)) = \chi_{\mathcal{P}}(\hat{\delta}(q_0, w))$. So A' covers the same language as A.

For the minimality first note that for all $q' \in Q'$ the set $L_{q'}(A) \neq \emptyset$, as otherwise $\text{level}_A(q') = \infty$ and thus every state in $N_{q'}$ would be similar to



	0	1	2	3	4	5	6
0	$\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$
1	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$
2	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$
3	$\not\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$	$\sim$
4	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$
5	$\sim$	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$
6	$\not\sim$	$\not\sim$	$\not\sim$	$\sim$	$\not\sim$	$\not\sim$	$\sim$

Figure 4.8: The automaton from Example 4.31 and its similarity table

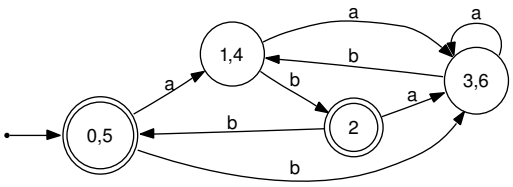


Figure 4.9: The minimized automaton from Example 4.31

every other state in Q. Thus merging $N_{q'}$ with N_{q_0} would yield a smaller $\sim_A$ complete partition. Denote by $x_{q'}$ a shortest word in $L_{q'}(A)$ for each $q' \in Q'$. We know that two states $p', q' \in Q'$ with $p' \neq q'$ are not similar, as otherwise $N_{p'} \cup N_{q'}$ would be $\sim_A$-complete (seen from Lemma 4.3 and Corollary 4.26 as $x_{q'}$ is a shortest word in $\bigcup_{r \in N_{q'}} L_r(A)$) contradicting the minimality of $\mathcal{N}$. Lemma 4.25 yields that $\{x_{q'} \mid q' \in Q'\}$ is a dissimilarity set for $\sim_{\mathcal{L}}$ of size $|Q'|$. As A' has exactly $|Q'|$ states we know from Corollary 4.14 that A' is minimal.

The time and space requirements are obvious. □

To show the correctness of our generic minimization algorithm we just have to stick the proofs of both phases together.

Corollary 4.30. *Let $A := (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a DP_kC_lA with n states covering some language $\mathcal{L}$. We assume the relation $\sim_A$ can be evaluated in constant time, for example has been pre-calculated. Then Algorithm 4.1 returns a minimal DP_kC_lA covering $\mathcal{L}$, and requires only $O(n^2 + |\Sigma|n)$ time and $O(|\Sigma|n)$ space.*

Proof. The correctness is just the combination of Lemma 4.27 and Theorem 4.29. Time and space consumption are those from Theorem 4.29 with $O(n^2)$ from the first **while** loop combined. □

The following example shows how Algorithm 4.1 clusters the states of an input automaton.

Example 4.31. *Let* $\Sigma := \{a, b\}$, $L := \{\epsilon, ab, abb, bbb\}$, *and* $\mathcal{L} := (\Sigma^{\leq 3} \backslash L, L)$. *A* DR_2C_lA *covering* $\mathcal{L}$ *is shown in Figure 4.8 (actually it is the minimal* DR_2A *accepting* $\mathcal{L}$*). Next to it is the tabulated state similarity relation. Applying Algorithm 4.1 on this automaton produces the set* $Q' = \{0, 1, 3, 2\}$ *with* $N_0 = \{0, 5\}$, $N_1 = \{1, 4\}$, $N_3 = \{3, 6\}$, *and* $N_2 = \{2\}$. *The resulting* DR_2C_3A *can be seen in Figure 4.9.*

So far we did not talk about how to decide whether a pair of states is in the relation $\sim_A$. Applying Lemma 4.25 and Definition 4.9 to the problem yields a straight forward method to compute the relation for two states p and q, but the running time could be up to $O(|\Sigma|^{\text{range}(p,q)})$. The method presented in [CPY02] is based on the gap function we will introduce next. Our definition differs slightly from the one in [CPY02] simplifying the proofs and algorithms somewhat.

Definition 4.32. *Let* $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ *be a* DR_kC_lA. *For two states* $p, q \in Q$ *the* gap *function is defined as the shortest word that can prove* $p \not\sim_A q$*:*

$$\text{gap}(p, q) := \min \left\{ |w| \mid w \in \Sigma^* \wedge \chi_{\mathcal{P}}(\hat{\delta}(p, w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q, w)) \right\} .$$

The minimum of the empty set is assumed to be $+\infty$ *as usual.*

Remark 4.33. *Obviously* $p \not\sim_A q$ *iff* $\text{gap}(p, q) \leq \text{range}(p, q)$.

The following lemma gives a hint on how to calculate the gap function.

Lemma 4.34. *Let* $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ *be a* DR_kC_lA. *For two states* $p, q \in Q$ *with* $\chi_{\mathcal{P}}(p) = \chi_{\mathcal{P}}(q)$

$$\text{gap}(p, q) = 1 + \min_{a \in \Sigma} \text{gap}(\delta(p, a), \delta(q, a)) .$$

Proof. We start with proving $\text{gap}(p, q) \leq 1 + \text{gap}(\delta(p, a), \delta(q, a))$ for all $a \in \Sigma$. Let $a \in \Sigma$, $r := \delta(p, a)$, $t := \delta(q, a)$. If $\text{gap}(r, t) = \infty$ the inequality is correct, so let $\text{gap}(r, t) = m$ and w a word with $|w| = m$ and $\chi_{\mathcal{P}}(\hat{\delta}(r, w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(t, w))$ which has to exist due to the definition of the gap function. But then aw is a word of length $m + 1$ and $\chi_{\mathcal{P}}(\hat{\delta}(p, aw)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q, aw))$ so $\text{gap}(p, q) \leq m + 1 = 1 + \text{gap}(r, t)$.

It remains to show that there always is an $a \in \Sigma$ such that $\text{gap}(p, q) \geq 1 + \text{gap}(\delta(p, a), \delta(q, a))$. For $\text{gap}(p, q) = \infty$ the inequality is again true. So let $\text{gap}(p, q) = m$ and $w \in \Sigma^m$ with $\chi_{\mathcal{P}}(\hat{\delta}(p, w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q, w))$. As $\chi_{\mathcal{P}}(p) = \chi_{\mathcal{P}}(q)$ we know that $m > 0$ and so there are $a \in \Sigma$ and $x \in \Sigma^{m-1}$ such that $w = ax$. Let again $r := \delta(p, a)$, $t := \delta(q, a)$. Then $\chi_{\mathcal{P}}(\hat{\delta}(r, x)) \neq \chi_{\mathcal{P}}(\hat{\delta}(t, x))$ and thus $\text{gap}(r, t) \leq m - 1$ which is exactly what we had to prove. □

We will present an algorithm for calculating the gap function for DP_kAs next. The algorithm exploits the fact that every DAG can be topologically sorted (see [CLRS01]). To simplify the presentation we assume the input DP_kA to be minimal. The details are shown in Algorithm 4.2, the now easy correctness proof follows.

Algorithm 4.2 An algorithm for calculating the gap function

Input: A minimal DP_kA $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ with n states,
Output: the gap function for all state pairs

rename the states as $\{q_0, q_1, \ldots, q_{n-1}\}$ such that $\mathrm{height}(q_{n-1}) = -\infty$ and for every transition $\delta(q_i, a) = q_j$ either $i = n-1$ or $i < j$.

for each $0 \leq i, j < n$ **do**
  if $\chi_{\mathcal{P}}(q_i) = \chi_{\mathcal{P}}(q_j)$ **then**
    $\mathrm{gap}(q_i, q_j) := \infty$
  else
    $\mathrm{gap}(q_i, q_j) := 0$
  end if
end for

for $i := n-1$ **down to** 0 **do**
  for $j := n-1$ **down to** 0 **do**
    for each $a \in \Sigma$ **do**
      {Invariant: $\mathrm{gap}(\delta(q_i, a), \delta(q_j, a))$ has already been calculated}
      $\mathrm{gap}(q_i, q_j) := \min\{\mathrm{gap}(q_i, q_j),\ 1 + \mathrm{gap}(\delta(q_i, a), \delta(q_j, a))\}$
    end for
  end for
end for

Theorem 4.35. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a minimal DP_kA for a language partition of finite order. Then Algorithm 4.2 correctly calculates the* gap *function for each state pair of A using only $O(|\Sigma|n^2)$ time and $O(n^2)$ memory.*

Proof. The first observation for the algorithm is that renaming the states appropriately is possible. Due to Lemma 3.22 for a minimal DP_kA there is at most one state with height $= -\infty$ and Lemma 3.21 yields that every other state is acyclic. As every state has outgoing transitions we know from graph theory, that the induced transition graph must have a cycle, so there is at least one state with height $= -\infty$. Removing this states makes the transition graph a DAG and the desired order is achieved in linear time by topological sorting.

	0 (0)	1 (1)	2 (2)	3 (1)	4 (2)	5 (3)	6 (2)
0 (0)	∞ (3)	0 (2)	1 (1)	0 (2)	0 (1)	2 (0)	0 (1)
1 (1)	0 (2)	∞ (2)	0 (1)	1 (2)	2 (1)	0 (0)	1 (1)
2 (2)	1 (1)	0 (1)	∞ (1)	0 (1)	0 (1)	1 (0)	0 (1)
3 (1)	0 (2)	1 (2)	0 (1)	∞ (2)	1 (1)	0 (0)	2 (1)
4 (2)	0 (1)	2 (1)	0 (1)	1 (1)	∞ (1)	0 (0)	1 (1)
5 (3)	2 (0)	0 (0)	1 (0)	0 (0)	0 (0)	∞ (0)	0 (0)
6 (2)	0 (1)	1 (1)	0 (1)	2 (1)	1 (1)	0 (0)	∞ (1)

Figure 4.10: The gap (range) table from Example 4.36

In the algorithm the first **for each** loop sets an upper bound on the gap value while the second one refines this bound. The invariant required for seeing the correctness is given as a comment in the algorithm. For the case $i = j = n - 1$ the gap value calculated is correct, as q_{n-1} is the sink state. So let either $i < n - 1$ or $j < n - 1$, $a \in \Sigma$ and $q_{i'} := \delta(q_i, a)$, $q_{j'} := \delta(q_j, a)$. The states have been renamed such that $i' \geq i$ and $j' \geq j$ and as one of q_i, q_j is not the sink state one of both inequalities is strict. But due to the order used for iterating on i and j and the invariant we know that $\mathrm{gap}(q_{i'}, q_{j'})$ has already been calculated. The correctness of the value retrieved for $\mathrm{gap}(q_i, q_j)$ is then provided by Lemma 4.34 if q_i and q_j are in the same component of $\mathcal{P}$, or by the initialization in the first **for** loop if they are not.

Looking at the algorithm the time and space complexity is easily observed from the nested loops. □

Before having a look at an example we should denote two implementation issues. By exploiting the symmetry of the gap function about half of the steps in Algorithm 4.2 can be omitted. The other useful observation is that the maximal range for two different states is $l - 1$ due to the fact that there is exactly one state with level 0, all other states having level ≤ 1. As we only need the gap value for comparison to the range we can use the value l instead of ∞ for practical purposes.

Example 4.36. *We revisit Example 4.31 here. The minimal DP_2A was given in Figure 4.8. Figure 4.10 provides a table giving for each state pair the values for* gap *and in parentheses for* range*. The parenthesized values next to each state name is the* level *of the state. The reader is encouraged to check those values and verify that they match with the similarity table shown in Figure 4.8 according to Remark 4.33.*

Finally we have collected all the building blocks allowing us the construction of a minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ for a given language k-partition of finite order. Although they are probably obvious by now they are composed into the last result of this section.

Corollary 4.37. *Let Σ be an alphabet, $l \in \mathbb{N}$, $\mathcal{L}$ a k-partition of $\Sigma^{\leq l}$, and A a DP_kA with n states accepting $\mathcal{L}$. Then A can be transformed into a minimal DP_kC_lA within $O(|\Sigma|n^2)$ time and $O(n^2 + |\Sigma|n)$ space.*

Proof. We construct a minimal $\mathrm{DP}_k\mathrm{A}$ B from A with any algorithm from Chapter 3. The algorithm from Theorem 4.35 calculates the gap function for all state pairs of B. The level_B can be calculated in linear time, so the similarity relation $\sim_B$ for all state pairs p, q of B can be retrieved in $O(n^2)$ steps by comparing $\mathrm{gap}_B(p, q)$ to $\mathrm{range}_B(p, q)$ (Remark 4.33). With this information Algorithm 4.1 gives the minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ for $\mathcal{L}$. The time and space complexity is easily derived by summing up the complexity of the individual steps. □

The algorithm we constructed so far only works for minimal DP_kAs. There a two ways to extend it to $\mathrm{DP}_k\mathrm{C}_l$As. One is to use the transformation from Section 4.4 to make a DP_kA first, the other is to modify Algorithm 4.2 for $\mathrm{DP}_k\mathrm{C}_l$As, which would involve using multiple "rounds" similar to the Bellman-Ford algorithm (for the Bellman-Ford algorithm see, *e.g.*, [CLRS01]). Both approaches will obviously increase the worst-case running time of the algorithm. The algorithm presented in the next section uses a different approach and has a better running time for constructing minimal $\mathrm{DP}_k\mathrm{C}_l$As from both DP_kAs and $\mathrm{DP}_k\mathrm{C}_l$As, so we will not go into detailing those proposed extensions here.

4.6 An $O(n \log n)$ minimization algorithm

For minimizing large $\mathrm{DP}_k\mathrm{C}_l$As the quadratic running time of the presented algorithm is too slow. The asymptotically best minimization algorithm for cover automata having a time bound of $O(n \log n)$ was proposed by Heiko Körner in [Kör03]. It has a similar structure as Hopcroft's algorithm for minimizing DFAs and the adaption towards $\mathrm{DP}_k\mathrm{C}_l$As is similarly simple as the adaption of Hopcroft's algorithm in Section 3.3. Like there we will only present the modified algorithm and discuss the high level results, but not delve into the implementation details that are needed for a thorough complexity analysis, as these are even more complex than for the Hopcroft algorithm and are not influenced by our modifications. For these the reader should consult [Kör03] where also a reference implementation in C++ is provided.

Körner's algorithm calculates a minimal $\sim_A$-complete partition of the states for a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ A. From this we can construct the minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ in $O(|\Sigma|n)$ time and space using the method from Theorem 4.29. The general idea is again to start with the state partition $\mathcal{P}$ provided by the automaton A and refine it until a $\sim_A$-complete partition is reached. Refinement is again achieved by splitting. The splitting step is however a bit more involved as

with the Hopcroft algorithm, so we present the algorithm first and later prove some invariants showing the validity of these splittings.

The pseudo code is provided as Algorithm 4.3. When compared to Körner's original algorithm the only difference besides adjusting some notation is in the initialization (first three lines), where the states are partitioned according to $\mathcal{P}$ instead of by being final or not as with plain cover automata. Additionally we introduced the variables γ_i which have no use in the algorithm but our proofs will benefit from them.

Before giving any proofs we want to point out the similarities and differences to Hopcroft's algorithm. In both algorithms a queue of sets still to be used for splitting is managed. These sets are used to split the existing components of the partition and after successfully splitting a component in two, the smaller part of these is again appended to the list. The difference is in how splitting is performed and what is managed in the list. While for the Hopcroft algorithm we had pairs of component indices and letters, we now put the component itself into the list (the index would not suffice as the component could be modified by further splits) together with some parameter t which indicates how the states in the component in question can be differentiated from all others.

In the remainder of this section let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a fixed $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ and $\mathcal{L}$ the partition of $\Sigma^{\leq l}$ covered by A. Denote by (S_i, t_i) the i-th pair added to the queue T. Obviously every S_i was inserted either from one component of the initial partition or as the new component created by splitting. Thus S_i captures the state of B_i when it was introduced for the first time and there is exactly one pair (S_i, t_i) for every B_i in the algorithm. We will see that the B_i and (S_i, t_i) are somehow linked and the t_i follow a certain pattern.

Lemma 4.38 ([Kör03], Lemma 4). *The sequence $t_1, \ldots, t_r$ produced by Algorithm 4.3 is non-decreasing.*

Proof. We prove this by induction on m, the length of a prefix of above sequence. The statement is trivially true in the beginning ($m = k$), as all t_i are initialized to 0. It also holds for the first appended t ($m = k + 1$), as it will have value 1. Now assume the sequence $t_1, \ldots, t_m$ is non-decreasing ($r > m \geq k + 1$). For the sequence $t_1, \ldots, t_m, t_{m+1}$ let i_j be the index of the pair (S, t) which lead to the addition of t_j. As $m \geq k + 1$ both i_m and i_{m+1} are defined. From the way the pairs (S, t) are managed (FIFO queue) we know that $i_m \leq i_{m+1} \leq m$, and so $t_{i_m} \leq t_{i_{m+1}}$. But as $t_m = t_{i_m} + 1$ and $t_{m+1} = t_{i_{m+1}} + 1$ this yields $t_m \leq t_{m+1}$ concluding the proof. □

To better understand the splitting process we introduce three invariants which we will prove next.

Lemma 4.39. *The following invariants hold for the outer **while** loop in*

Algorithm 4.3 The Körner algorithm for $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization

Input: A $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$
Output: A minimal $\sim_A$-complete partition of Q

$(B_1, \ldots, B_k) := \mathcal{P}$
$r := k$
push all pairs $(B_i, 0)$ with $(1 \leq i \leq k)$ into FIFO queue T
$\gamma_i := l$ for each $1 \leq i \leq k$

while $T \neq \emptyset$ **do**
  remove first pair (S, t) from T
  for each $a \in \Sigma$ **do**
    $X := \{q \in Q \mid \delta(q, a) \in S \wedge \mathrm{level}(q) < l - t\}$
    $Y := \{q \in Q \mid \delta(q, a) \notin S \wedge \mathrm{level}(q) < l - t\}$
    for $i := r$ **down to** 1 **do**
      if $B_i \cap X \neq \emptyset$ **and** $B_i \cap Y \neq \emptyset$ **then**
        $Z := B_i \cap X$
        $r := r + 1$
        if $|Z| \leq |B_i \setminus Z|$ **then**
          $B_r := Z$
          $B_i := B_i \setminus Z$
        else
          $B_r := B_i \setminus Z$
          $B_i := Z$
        end if
        $\gamma_i := l - (t + 1)$
        $\gamma_r := l - (t + 1)$
        push $(B_r, t + 1)$ into T
      end if
    end for
  end for
end while

return $(B_1, \ldots, B_r)$

Algorithm 4.3 for every $1 \leq i < j \leq r$*:*

$$\exists q \in B_i \quad : \quad \text{level}(q) \leq \gamma_i \tag{4.1}$$

$$\forall p \in B_i,\, q \in B_j \quad : \quad \text{level}(p) \leq \gamma_i \;\wedge\; \text{level}(q) \leq \gamma_j \;\Rightarrow\; \text{gap}(p,q) \leq l - \max\{\gamma_i, \gamma_j\} \tag{4.2}$$

$$\forall p \in S_i,\, q \in Q \setminus S_i \quad : \quad \text{range}(p,q) \geq t_i \;\Rightarrow\; \text{gap}(p,q) \leq t_i \tag{4.3}$$

Proof. We prove all of them at once. As the language partition $\mathcal{L}$ must have at least one word of length $\leq l$ in each component, there also has to be a state q with $\text{level}(q) \leq l$ in each component of $\mathcal{P}$. As the first B_i are chosen as components of $\mathcal{P}$ and the matching γ_i are l, Invariant 4.1 holds in the beginning. Similarly Invariants 4.2 and 4.3 obviously hold due to the initialization by $\mathcal{P}$.

It remains to show that splitting (the inner **for** loop) does not invalidate the invariants. So assume all three invariants hold before execution of the loop and let (S_p, t_p) be the pair drawn from the queue T, and $a \in \Sigma$ the letter in the **for each** loop. As in the algorithm let $X := \{q \in Q \mid \delta(q,a) \in S_p \wedge \text{level}(q) < l - t_p\}$ and $Y := \{q \in Q \mid \delta(q,a) \notin S_p \wedge \text{level}(q) < l - t_p\}$, and let B_i be the component we are splitting, so $B_i \cap X \neq \emptyset$ and $B_i \cap Y \neq \emptyset$. Set $Z := B_i \cap X$ and assume $|Z| \leq |B_i \setminus Z|$ (the opposite case is symmetric and will not be shown). We prime a variable to indicate its value after the splitting has occurred, so x' is the value of x after splitting B_i w.r.t. (S_p, t_p) and a. Variables that do not change once initialized (like t_i) are not primed. The situation now is as follows:

- $r' = r + 1$
- $t_{r'} = t_p + 1$
- $B_j = B'_j$ for every $j \leq r$ and $j \neq i$
- $B_i = B'_i \cup B'_{r'}$
- $\gamma_j = \gamma'_j$ for every $j \leq r$ and $j \neq i$
- $\gamma'_i = \gamma'_{r'} = l - (t_p + 1)$

Invariant 4.1 holds for all $j \leq r$ and $j \neq i$ as then $B'_j = B_j$ and $\gamma'_j = \gamma_j$. Due to the definition of X and Y both B'_i and $B'_{r'}$ will include states with level $\leq l - t_p - 1 = \gamma'_i = \gamma'_{r'}$ completing the first invariant.

For Invariant 4.2 choose arbitrary $j_1 \neq j_2$ and let $p \in B'_{j_1}$, $q \in B'_{j_2}$ with $\text{level}(p) \leq \gamma'_{j_1}$ and $\text{level}(q) \leq \gamma'_{j_2}$. We have to show

$$\text{gap}(p,q) \leq l - \max\{\gamma'_{j_1}, \gamma'_{j_2}\}\,.$$

As our setup is symmetric for j_1 and j_2 (*i.e.*, exchanging both does not invalidate it) is suffices to check the next three cases:

- $\{j_1, j_2\} \cap \{i, r'\} = \emptyset$
  This case directly follows from Invariant 4.2 as $B'_{j_1} = B_{j_1}$, $B'_{j_2} = B_{j_2}$, $\gamma'_{j_1} = \gamma_{j_1}$, and $\gamma'_{j_2} = \gamma_{j_2}$.

- $j_1 \in \{i, r'\}$ and $j_2 \notin \{i, r'\}$
  We know that either $\gamma_i = l$ from the initialization or $\gamma_i = l - (t_m + 1)$ for some $m \leq p$ as this is the only way γ_i is changed. But as for $m \leq p$ we know $t_p \geq t_m$ and also $t_p \geq 0$, so obviously $l - (t_p + 1) = \gamma'_{j_1} \leq \gamma_i$.

  Every element from B'_{j_1} has been in B_i before, and as $\text{level}(p) \leq \gamma'_{j_1} \leq \gamma_i$ we can apply Invariant 4.2 yielding $\text{gap}(p, q) \leq l - \max\{\gamma_i, \gamma'_{j_2}\}$. Additionally from $\gamma'_{j_1} \leq \gamma_i$ we know $\max\{\gamma'_{j_1}, \gamma'_{j_2}\} \leq \max\{\gamma_i, \gamma'_{j_2}\}$ and so $\text{gap}(p, q) \leq l - \max\{\gamma'_{j_1}, \gamma'_{j_2}\}$.

- $j_1 = i$ and $j_2 = r'$
  This is the interesting case, as it highlights the relationship between the components just split. It is easily seen that if X and Y are "useful" (*i.e.*, non-empty), (X, Y) is a partition of the set $\{q \in Q \mid \text{level}(q) \leq l - (t_p + 1)\}$. So every $p \in B'_i$ with $\text{level}(p) \leq \gamma'_i = l - (t_p + 1)$ has to be in Y and every $q \in B'_{r'}$ with $\text{level}(q) \leq \gamma'_{r'}$ has to be in X. This yields $\delta(p, a) \notin S_p$ and $\delta(q, a) \in S_p$ by definition of X and Y. From $\text{level}(\delta(p, a)) \leq 1 + \text{level}(p) \leq l - t_p$ and $\text{level}(\delta(q, a)) \leq l - t_p$ we receive $\text{range}(\delta(p, a), \delta(q, a)) \geq l - (l - t_p) = t_p$, so Invariant 4.3 gives $\text{gap}(\delta(p, a), \delta(q, a)) \leq t_p$ and thus $\text{gap}(p, q) \leq 1 + t_p = l - (l - (t_p + 1)) = l - \max\{\gamma'_i, \gamma'_{r'}\}$ which was to be shown.

For Invariant 4.3 we only have to look at the new pair $(S_{r'}, t_{r'})$ as all other S_j and t_j are not modified. Now let $p \in S_{r'} = B'_{r'}$ and $q \in Q \backslash S_{r'}$, so there is a $j < r'$ with $q \in B'_j$. From the way the γ are managed we know that $\gamma'_{r'} \leq \gamma'_j$. Additionally let $\text{range}(p, q) \geq t_{r'}$, then we have to show $\text{gap}(p, q) \leq t_{r'}$. We know that $\max\{\text{level}(p), \text{level}(q)\} \leq l - \text{range}(p, q) \leq l - t_{r'} = \gamma'_{r'}$, so $\text{level}(p) \leq \gamma'_{r'}$ and $\text{level}(p) \leq \gamma'_{r'} \leq \gamma'_j$ and we can apply the results from Invariant 4.2 just shown. Thus $\text{gap}(p, q) \leq l - \max\{\gamma'_{r'}, \gamma'_j\} = l - \gamma'_j \leq l - \gamma'_{r'}$ completing also the correctness of the third invariant. □

Basically those invariants show that no superfluous splits are performed. On the other hand we have to see all dissimilar states separated by splitting.

Lemma 4.40 ([Kör03], Lemma 5). *Let $p, q \in Q$ with $p \not\sim_A q$,* i.e., *there is $0 \leq m \leq l$ with $\text{gap}(p, q) \leq m \leq \text{range}(p, q)$. Then there is a pair (S, t) in T separating p and q and a pair (S', t') with $|\{p, q\} \cap S'| = 1$ and $t' \leq m$ is added to T when splitting the set containing p and q (or p, q are initially separated and such a pair (S', t') exists from initialization).*

Proof. We prove this by induction on m. The case $m = 0$ is obviously true from the initialization phase of the algorithm, as then $\text{gap}(p, q) = 0$, so

$\chi_{\mathcal{P}}(\hat{\delta}(p,\epsilon)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q,\epsilon))$, that is p and q are already in different components of $\mathcal{P}$. The pair (S',t') exists from the way T is initialized.

Assume the statement holds for all $m \leq m'$ and let $m = m'+1$. As $\text{gap}(p,q) \leq m$ there is a word $w \in \Sigma^{\leq m}$ with $\chi_{\mathcal{P}}(\hat{\delta}(p,w)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q,w))$. The case $w = \epsilon$ is trivial (the same as $m = 0$), so let $w = av$ with $a \in \Sigma$, $v \in \Sigma^{\leq m'}$, and $p' := \delta(p,a)$, $q' := \delta(q,a)$. Then we know $\text{range}(p',q') \geq \text{range}(p,q) - 1 \geq m'$ and as $\chi_{\mathcal{P}}(\hat{\delta}(p',v)) \neq \chi_{\mathcal{P}}(\hat{\delta}(q',v))$ also $\text{gap}(p',q') \leq |v| \leq m'$. So by induction after p' and q' have been separated, there was a pair (S,t) in T with $t \leq m'$ and $|\{p',q'\} \cap S| = 1$. From $\text{range}(p,q) \geq m$ we know that $\max\{\text{level}(p), \text{level}(q)\} \leq l - m = l - (m'+1) < l - t$ but only one of $\delta(p,a)$ and $\delta(q,a)$ is in S, so when splitting w.r.t. (S,t) and a either $p \in X$ and $q \in Y$ or the other way round. But this means if p and q are in the same component when (S,t) is drawn from T they will be separated and the pair (S',t') generated has either $p \in S'$ or $q \in S'$ but not both and $t' = t+1 \leq m'+1 = m$. □

For the correctness proof we now only have to collect the results seen so far.

Theorem 4.41 ([Kör03], Theorem 2). *On input of a DP_kC_lA $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ Algorithm 4.3 computes a minimal $\sim_A$-complete partition of Q.*

Proof. From the way the B_i are managed it should be obvious that $\mathcal{B} := (B_1, \ldots, B_r)$ is a partition of Q. From Lemma 4.40 we know that dissimilar states are always separated (every pair (S,t) in T is used for splitting) and as the components B_i are only further fragmented but never merged, $\mathcal{B}$ has to be $\sim_A$-complete.

For minimality denote by b_i a state of minimal level in B_i, that is $b_i \in B_i$ and $\text{level}(b_i) = \min_{q \in B_i} \text{level}(q)$. From Invariant 4.1 we know $\text{level}(b_i) \leq \gamma_i$ for every $1 \leq i \leq r$. But then for any $1 \leq i < j \leq r$ Invariant 4.2 gives $\text{gap}(b_i, b_j) \leq l - \max\{\gamma_i, \gamma_j\} \leq l - \max\{\text{level}(b_i), \text{level}(b_j)\} = \text{range}(b_i, b_j)$ and thus $b_i \not\sim_A b_j$. So $D := \{b_i \mid 1 \leq i \leq r\}$ is $\sim_A$-independent and Lemma 2.6 yields the minimality of $\mathcal{B}$ as $|\mathcal{B}| = |D|$. □

After having seen the correctness of the algorithm we will continue by discussing the complexity of a suitable implementation. Let n be the number of states and m the size of the alphabet for the input automaton. The initialization part can obviously be implemented in linear time, so we will only talk about the main **while** loop. As every B_i always contains at least one state, r cannot exceed n. For fixed i the set B_i never grows, so always $|B_i| \leq |S_i|$. Assume the set B_i is just split resulting in the new set B_j. As always the smaller part is used for the new set, $2|S_j| \leq |S_i|$. Consequently a fixed state q can be moved to a new component at most $\log n$ times and the overall number of state moves is bounded by $n \log n$.

The part that turns out to be the most expensive is the calculation of the set X (although in the implementation it is never represented explicitly). As with the Hopcroft algorithm we use the (pre-calculated) lists $\delta^{-1}(q,a) := \{p \in Q \mid \delta(p,a) = q\}$, *i.e.*, for each $q \in S$ and every $p \in \delta^{-1}(q,a)$ we add p to X if it satisfies the level constraint. So the overall number of steps for the construction of X summed over all loop iterations is

$$k_1 \sum_{i=1}^{r} \sum_{a \in \Sigma} \sum_{q \in S_i} |\delta^{-1}(q,a)|\,,$$

where k_1 is a constant. As explained before, no state q appears in more than $\log n$ of the S_i, so above expression is bounded by

$$k_1 \log n \sum_{q \in Q} \sum_{a \in \Sigma} |\delta^{-1}(q,a)|\,,$$

which again simplifies to $O(mn \log n)$ due to $\sum_{q \in Q} \sum_{a \in \Sigma} |\delta^{-1}(q,a)| = mn$ as illustrated in Section 3.3.

While this demonstrates where the $n \log n$ have their origin in, there are still many more details needing illumination. One of these is that storing all of the S_i in the queue T would require $\Omega(n \log n)$ memory, which can be avoided by maintaining for each B_i a history of components having been split off. Another aspect is that the test *if* $B_i \cap X \neq \emptyset$ *and* $B_i \cap Y \neq \emptyset$ can be performed in constant time, if a bunch of arrays indicating the number of elements in certain sets is managed.

For a complete complexity analysis all these details would have to be outlined and again shown to be possible *in time*. However none of these is really influenced by our minor modifications and so the reader is kindly asked to consult [Kör03] for implementation details (down to the source code level) and a thorough analysis of the running time. The analogon to our final result is also found there.

Theorem 4.42 ([Kör03], Theorem 3). *For a $DP_k C_l A$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ with $|\Sigma| = m$ and $|Q| = n$ Algorithm 4.3 determines a minimal $\sim_A$-complete partition of Q in time $O(mn \log n)$ and space $O(mn)$.*

Corollary 4.43. *For a $DP_k C_l A$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ with $|\Sigma| = m$ and $|Q| = n$ the minimal $DP_k C_l A$ can be constructed in time $O(mn \log n)$ and space $O(mn)$.*

Chapter 5

Lookup automata

In the previous chapter we have seen how limiting the class of words to be decided can lead to a more compact automaton. The words in the context of our application will be IP addresses, which are not only finite but are guaranteed to be of the same size. Thus it seems natural to tighten the constraint of deciding only words of size *at most* l even more to requiring input words to be of length *exactly* l. We will call automata for this restricted class of words *lookup automata*.

Definition 5.1. *Let Σ be an alphabet, $l \in \mathbb{N}$, $\mathcal{L}$ a k-partition of Σ^l. A DP$_k$A A is called a* deterministic k-partition l-lookup automaton (DP$_k$L$_l$A) for $\mathcal{L}$ *iff $\mathcal{L}(A)|_{\Sigma^l} = \mathcal{L}$.*

We will start the chapter by visiting some examples of DP$_k$L$_l$As. Then we look at how to decide the equivalence of two DP$_k$L$_l$As and the duality between independent sets and right invariant equivalence partitions for a relation on words induced by lookup automata. Following the pattern of the previous chapters we would study minimization algorithms next, but as it turns out not to be easily solved, we give evidence for its potential hardness instead and provide heuristics which at least produce small (instead of minimal) DP$_k$L$_l$As.

5.1 An example

To introduce lookup automata and to highlight some interesting properties of DP$_k$L$_l$As, we will look at the family of DP$_2$L$_l$As which recognize the word 0^n1^n. More formally for a given n we are interested in DP$_2$As on alphabet $\{0,1\}$ accepting a language partition (L_1, L_2) with $L_2 \cap \{0,1\}^{2n} = \{0^n1^n\}$.

To start with something well known, we should construct the minimal DP$_2$As recognizing 0^n1^n. The construction is fairly simple and shown in Figure 5.1. Obviously the minimal DP$_2$A accepting only 0^n1^n has exactly $2n + 2$ states. We can do somewhat better by using a cover automaton.

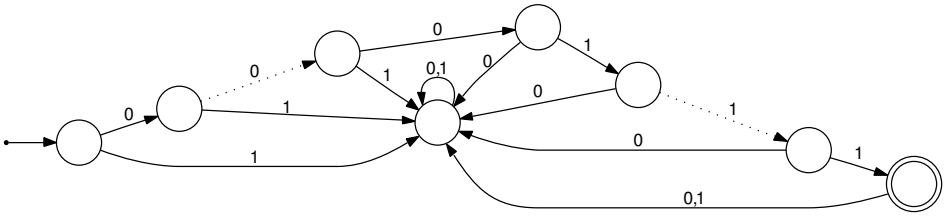


Figure 5.1: The minimal DP_2A recognizing 0^n1^n

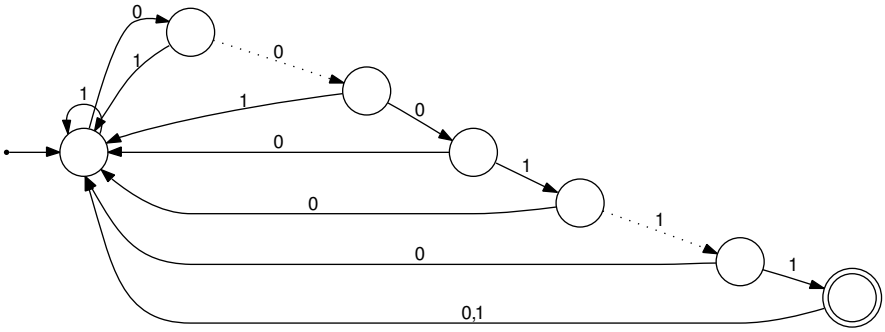


Figure 5.2: The minimal $DP_2C_{2n}A$ recognizing 0^n1^n

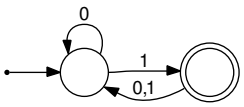


Figure 5.3: The minimal DP_2L_2A recognizing 01

Following the ideas from the previous chapter we see the sink state and the initial state to be similar, but all other state pairs are dissimilar. So the minimal $DP_2C_{2n}A$ for our example (Figure 5.2) is found by merging the initial and the sink state in the minimal DP_2A and thus has $2n + 1$ states.

We introduced DP_kL_lAs hoping for reduced automaton size and therefore expect to construct a suitable automaton using less than $2n + 1$ states. Indeed we can, but as the automata are not easily described we start with small fixed values of n.

The first case is $n = 1$. Obviously there is no solution with only one state, as the words 01 and 00 have to go into separate states. But for two states with some trial-and-error we easily find the automaton shown in Figure 5.3 which has to be the minimal DP_2L_2A for the partition $(\{00, 10, 11\}, \{01\})$.

Now for the case $n = 2$. After some thinking on how DP_kL_lAs work and how to exploit this, one might find one of the DP_2L_4As shown in Figure 5.4

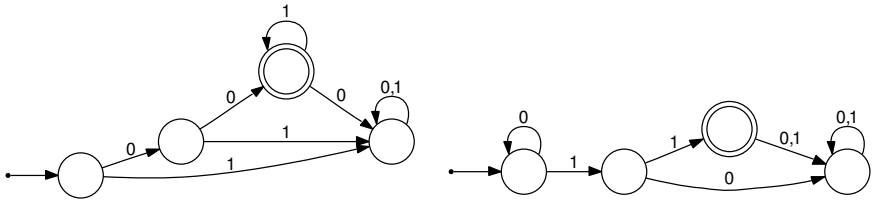


Figure 5.4: Two DP_2L_4As for 0011

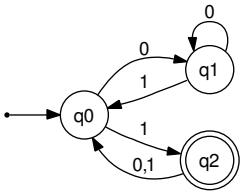


Figure 5.5: The minimal DP_2L_4A recognizing 0011

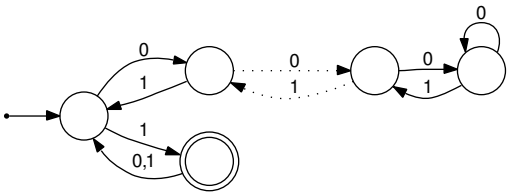


Figure 5.6: A $DP_2L_{2n}A$ recognizing 0^n1^n

having 4 states. However these are not minimal, as by using a brute-force program which enumerates all automata with less than 4 states we find the unique minimal DP_2L_4A from Figure 5.5. To see, why from all words of length 4 only 0011 is accepted by this automaton, we observe that for all words w of even length we would have $\hat{\delta}(q_0, w) = q_0$ if the 0-loop at state q_1 would not exist. So to reach the accepting state q_2 in 4 steps we have to use this transition. The shortest possible word including this transition is 00 and the shortest word to get from q_1 to q_2 is 11 which happen to give the required 0011.

This construction principle can be generalized by "prolonging" the path to the 0-loop and back as in Figure 5.6. Our first case of 0^11^1 also fits into this pattern. This automaton uses only $n+1$ states opposed to the $2n+1$ of the minimal $DP_2C_{2n}A$. Using our brute-force code we can show that these automata are minimal for $n \in \{1, 2, 3, 4\}$ and even unique (up to isomorphic state renaming).

For $n \in \{5, 6\}$ above automaton is still minimal, but we can find non-

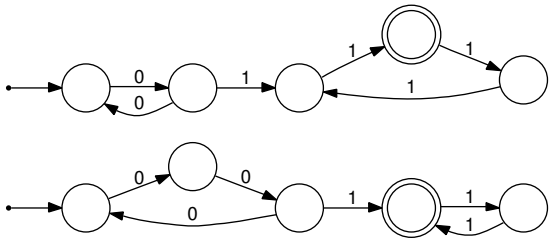


Figure 5.7: Two $\mathrm{DP_2L_{10}As}$ recognizing $0^5 1^5$ (without sink state)

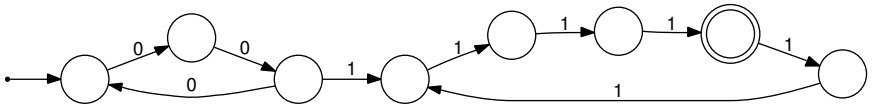


Figure 5.8: A $\mathrm{DP_2L_{28}A}$ recognizing $0^{14}1^{14}$ (without sink state)

isomorphic $\mathrm{DP_2L_{2n}As}$ with the same number of states $(n+1)$. The two additional automata for $n = 5$ are displayed in Figure 5.7 but omitting the sink state (so every missing transition is towards some state $\perp$).

Unfortunately our brute-force approach is not efficient enough to continue enumerating minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ for larger n within reasonable time. However from the last example we find another construction principle. While our first attempts of finding minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$ in Figure 5.4 had a single cycle on the path from the initial to the accepting state, the automata in Figure 5.7 contain *two* cycles.

We will give a final example for this two-cycle approach. Let $n = 14$, then the automaton from Figure 5.8 is a $\mathrm{DP_2L_{28}A}$ accepting only $0^{14}1^{14}$. To see why this holds, we observe that each word reaching the accepting state has to traverse the first cycle x times and the second cycle y times. Additionally as this is a $\mathrm{DP_2L_{28}A}$ only words of length 28 are acceptable, so for x and y we get the formula $2 + 3x + 4 + 5y = 28$. The integral solutions of this equation are given by $(x, y) \in \{(4 - 5u, 2 + 3u) \mid u \in \mathbb{Z}\}$. As both x and y have to be non-negative, the only valid solution is $x = 4$, $y = 2$ and thus the only accepted word is $0^{2+3\cdot4}1^{4+5\cdot2} = 0^{14}1^{14}$. We do not know, if this $\mathrm{DP_2L_{28}A}$ is minimal, but with only 9 states (remember the sink state) we are far better than the $n + 1 = 15$ state construction from above.

To complete this example we summarize our observations so far.

Observation 5.2. *1. The minimal DP_kL_lA for a given partition of Σ^l is not necessarily unique.*

2. The minimal DP_kL_lA for a given partition of Σ^l is often smaller than a minimal DP_kA or DP_kC_lA for that language partition.

3. *Given a family of language partitions for which the minimal DP_kAs and minimal DP_kC_lAs follow a simple pattern, the minimal DP_kL_lAs do not seem to follow a single pattern at all.*

For the size of a minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ (second item) we have the trivial upper bound of the size of a minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ (and thus $\mathrm{DP}_k\mathrm{A}$).

5.2 Equivalence testing

Before handling $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ minimization we will investigate the related problem of automaton equivalence. Two automata are equivalent, if they recognize the same language, so given $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$ $A = (\Sigma, Q, q_0, \mathcal{P}_Q, \delta)$ and $B = (\Sigma, R, r_0, \mathcal{P}_R, \gamma)$ we want to know whether $\mathcal{L}(A)|_{\Sigma^l} = \mathcal{L}(B)|_{\Sigma^l}$. Following let $n_A := |Q|$, $n_B := |R|$, and $m := |\Sigma|$, so the size of the input to the problem is bounded by $O(m(n_A + n_B) + \log l)$.

There is a rather simple algorithm for this problem running in pseudo-polynomial time. Let C be the minimal DFA accepting the language Σ^l. C is easily seen to have $l + 2$ states. Then the $\mathrm{DP}_k\mathrm{As}$ $A \times C$ and $B \times C$ accept the languages $\mathcal{L}(A)|_{\Sigma^l}$ respectively $\mathcal{L}(B)|_{\Sigma^l}$. Both of these automata are acyclic, so we can use the algorithm from Section 3.4 to minimize them. Then testing equivalence can be performed by a simple depth first search as the minimal $\mathrm{DP}_k\mathrm{A}$ is unique. Multiplying a $\mathrm{DP}_k\mathrm{A}$ with C increases its size at most by a factor $l + 2$ and this process can be performed in time linear in the size of the resulting automaton. These ideas show the correctness of the following lemma.

Lemma 5.3. *Let A and B be two DP_kL_lAs with n_A respectively n_B states on an alphabet of size m. The equivalence of A and B can be decided in time $O(lm(n_A + n_B))$.*

The problem with this approach is the linear dependence on l, while l accounts only logarithmically to the input size of the equivalence problem. As long as l is bounded by a polynomial in the size of the two input $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$ (as it is for example in the context of routing tables) this is not a problem. However in general we could have $\log l = \Omega(m(n_A + n_B))$ for which above algorithm degenerates to an exponential one, so we will work towards a refined algorithm working in polynomial time.

For every pair $(q, r) \in Q \times R$ and $i \in \mathbb{N}_0$ denote by

$$S_i(q, r) := \bigcup_{w \in \Sigma^i} \left\{ (\hat{\delta}(q, w), \hat{\gamma}(r, w)) \right\}$$

the set of state pairs from $Q \times R$ which are reachable from (q, r) with the same word of length i. The usefulness of this definition for the equivalence problem is shown next.

Lemma 5.4. *Let $A = (\Sigma, Q, q_0, \mathcal{P}_Q, \delta)$ and $B = (\Sigma, R, r_0, \mathcal{P}_R, \gamma)$ be two DP_kL_lAs. A and B are equivalent, iff for all $(q, r) \in S_l(q_0, r_0)$ we have $\chi_{\mathcal{P}_Q}(q) = \chi_{\mathcal{P}_R}(r)$.*

Proof. A and B are equivalent iff $\mathcal{L}(A)|_{\Sigma^l} = \mathcal{L}(B)|_{\Sigma^l}$. This can be rephrased as $\forall w \in \Sigma^l : \chi_{\mathcal{L}(A)}(w) = \chi_{\mathcal{L}(B)}(w)$, which in turn is the same as $\forall w \in \Sigma^l : \chi_{\mathcal{P}_Q}(\hat{\delta}(q_0, w)) = \chi_{\mathcal{P}_R}(\hat{\gamma}(r_0, w))$. Substituting q for $\hat{\delta}(q_0, w)$ and r for $\hat{\gamma}(r_0, w)$ yields $\forall (q, r) \in \left\{ \left(\hat{\delta}(q_0, w), \hat{\gamma}(r_0, w) \right) \mid w \in \Sigma^l \right\} : \chi_{\mathcal{P}_Q}(q) = \chi_{\mathcal{P}_R}(r)$ which reduces to $\forall (q, r) \in S_l(q_0, r_0) : \chi_{\mathcal{P}_Q}(q) = \chi_{\mathcal{P}_R}(r)$. □

So if we are given $S_l(q_0, r_0)$ we can check the equivalence of A and B in time $O(n_A n_B)$ as $S_l(q_0, r_0)$ contains at most that many state pairs. Now we only have to find an efficient method for calculating $S_l(q_0, r_0)$. For small i we can use above definition of $S_i(q, r)$, but as there are m^i words of length i this does not scale well. However there is a simple formula to calculate $S_{i+j}(q, r)$ if we already know S_i and S_j:

$$\begin{aligned} S_{i+j}(q,r) &= \bigcup_{w \in \Sigma^{i+j}} \left\{ (\hat{\delta}(q,w), \hat{\gamma}(r,w)) \right\} \\ &= \bigcup_{u \in \Sigma^i} \bigcup_{v \in \Sigma^j} \left\{ (\hat{\delta}(q,uv), \hat{\gamma}(r,uv)) \right\} \\ &= \bigcup_{u \in \Sigma^i} S_j(\hat{\delta}(q,u), \hat{\gamma}(r,u)) \\ &= \bigcup_{(s,t) \in S_i(q,r)} S_j(s,t) \end{aligned}$$

In conjunction with a trick often referred to as *repeated squaring* which is used for efficient exponentiation, this results in a polynomial time algorithm for the $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ equivalence problem.

Theorem 5.5. *Let A and B be two DP_kL_lAs with n_A respectively n_B states with an alphabet of size m. The equivalence of A and B can be decided in time $O(mn_A^2n_B^2 + n_A^3n_B^3 \log l)$.*

Proof. We assume l to be given in binary, *i.e.*, $l = \sum_{i=0}^{r} l_i 2^i$ with each $l_i \in \{0, 1\}$ and $r = \lceil \log_2(l + 1) \rceil - 1$. Furthermore let $Q = \{q_0, \ldots, q_{n_A - 1}\}$, $R = \{r_0, \ldots, r_{n_B - 1}\}$, so we can represent each $S_i(q, r)$ as a matrix from $\{0, 1\}^{n_A \times n_B}$ where a 1 in row x and column y indicates $(q_x, r_y) \in S_i(q, r)$. We write S_i for the set $\{S_i(q, r) \mid (q, r) \in Q \times R\}$. Clearly we can initialize S_0 and S_1 in time $O(mn_A^2n_B^2)$.

Given S_i and S_j we can calculate S_{i+j} using the formula derived before in $O(n_A^3n_B^3)$ steps. Thus determining S_{2^i} for all $1 \leq i \leq r$ takes $O(n_A^3n_B^3r)$ steps using $S_{2^i} = S_{2^{i-1}+2^{i-1}}$. Now S_l can be written as $S_{l_0 2^0 + \cdots + l_r 2^r}$ and as

all required S_{2^i} are known this can be done in $O(n_A^3 n_B^3 r)$ as well. To test equivalence we use Lemma 5.4, and the overall complexity follows as the sum of these individual steps. □

5.3 Duality gap

When developing an algorithm for minimizing $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ we would like to prove its correctness. Therefore we need some criterion for the minimality of an $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$. What was used for $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$ (and also for $\mathrm{DP}_k\mathrm{As}$, but as it was less explicit there, we will take $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$ for reference here), were sets of words from which no two can be in the same state language (dissimilarity sets). Obviously a minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ must have at least as many states as the size of the largest of these sets, so the problem of finding a maximal dissimilarity set is dual to the $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization problem. We put quite some effort in proving that the size of a *maximal* dissimilarity set and the the size of a *minimal* $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ are the same (Corollary 4.5, Theorems 4.8 and 4.13), as then we can prove a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimal by finding a dissimilarity set of the same size. Unfortunately we cannot use the same approach for $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$ as there is a gap between the size of a minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ and a maximal set of words not allowed in the same state language, which will be shown in this section.

An essential step in the previous chapters has been the definition of a suitable relation indicating if two words are "compatible", *i.e.*, given a language partition $\mathcal{L}$, these words could be placed into the same state of an automaton, without making $\mathcal{L}$ impossible to be recognized by it. The canonical definition in the $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ context is the following, which should be compared to Definition 4.9.

Definition 5.6. *Let Σ be an alphabet, $l \in \mathbb{N}$, and $\mathcal{L}$ a partition of Σ^l. Define the relation $\simeq_{\mathcal{L}} \subseteq \Sigma^* \times \Sigma^*$ by*

$$x \simeq_{\mathcal{L}} y \quad :\Leftrightarrow \quad \forall w \in \Sigma^* : \left((\, |xw| = l \wedge |yw| = l \,) \Rightarrow (\chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)) \right).$$

Above definition is better understood when split into two cases. If $|x| \neq |y|$ or $|x| > l$ or $|y| > l$, then always $x \simeq_{\mathcal{L}} y$, as the condition $|xw| = l \wedge |yw| = l$ will never hold for the same w. For the remaining case ($|x| = |y| \leq l$) we get

$$x \simeq_{\mathcal{L}} y \;\Leftrightarrow\; \forall w \in \Sigma^{l-|x|} : \chi_{\mathcal{L}}(xw) = \chi_{\mathcal{L}}(yw)\,.$$

From this it is easily seen, that for any $r \geq 0$ the relation $\simeq_{\mathcal{L}} \cap (\Sigma^r \times \Sigma^r)$ is an equivalence relation. So the following lemma follows without further proof.

Lemma 5.7. *Let Σ be an alphabet, $l > 0$, $\mathcal{L}$ a k-partition of Σ^l, and $D \subseteq \Sigma^*$ an $\simeq_{\mathcal{L}}$-independent set. Then all words in D are of the same length.*

This lemma helps us in finding maximal $\simeq_{\mathcal{L}}$-independent sets as shown next.

Lemma 5.8. *Let Σ be an alphabet with $|\Sigma| \geq 2$, $l \geq 1$, and $\mathcal{L} = (L_1, L_2)$ a 2-partition of Σ^l separating a single word,* i.e., *$|L_2| = 1$. Then a maximal $\simeq_{\mathcal{L}}$-independent set has size 2.*

Proof. Let $u \in L_1$, and v the only word from L_2, then obviously $\{u, v\}$ is a $\simeq_{\mathcal{L}}$-independent set of size 2. Now assume there is a bigger $\simeq_{\mathcal{L}}$-independent set $D = \{d_1, \ldots, d_r\} \subseteq \Sigma^*$ with $r > 2$ and all d_i pairwise distinct. From Lemma 5.7 we know $|d_1| = \ldots = |d_r|$ and $|d_1| \leq l$, as otherwise some $d_i \simeq_{\mathcal{L}} d_j$ $(i \neq j)$, contradicting D's $\simeq_{\mathcal{L}}$-independence. Thus at most one element in D can be a prefix of v (the only word from L_2) and there must be two elements $\hat{d} \neq \bar{d}$ in D not being a prefix of v. But then for every word $w \in \Sigma^{l-|d_1|}$ both $\hat{d}w \neq v$ and $\bar{d}w \neq v$. As v was the only word in L_2 we get $\forall w \in \Sigma^{l-|d_1|} : \chi_{\mathcal{L}}(\hat{d}w) = 1 = \chi_{\mathcal{L}}(\bar{d}w)$ and thus $\hat{d} \simeq_{\mathcal{L}} \bar{d}$, a contradiction. □

Now we are ready to give a counter example, showing the gap we initially mentioned.

Theorem 5.9. *For each alphabet Σ with $|\Sigma| \geq 2$ and each $l > 6$ there is a 2-partition $\mathcal{L}$ of Σ^l such that the size of a maximal $\simeq_{\mathcal{L}}$-independent set is strictly smaller than the size of a minimal DP_2L_lA deciding $\mathcal{L}$.*

Proof. We will assume $|\Sigma| = 2$ and generalize to larger alphabets in the end. For this alphabet there are at most $n^{2n}2^n$ $\mathrm{DP_2L}_l\mathrm{As}$ with exactly n states, as each of the $2n$ transitions can be chosen from n target states, the 2^n results from partitioning n states into final and non-final states. Thus there are at most $n^{2n}2^n$ different 2-partitions of Σ^l recognizable by an n-state $\mathrm{DP_2L}_l\mathrm{A}$, more specific no more than $2^{2\cdot 2}2^2 = 64$ different 2-partitions can be recognizes by all $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$ with 2 states. On the other hand we know that for each 2-partition $\mathcal{L}$ with only a single word in the second component the maximal $\simeq_{\mathcal{L}}$-independent set has size 2 and there are $2^l > 2^6 = 64$ such partitions. Thus there has to be at least one 2-partition fulfilling the claim of this theorem.

If we add a new letter to Σ the claim obviously stays valid, as now matter how we "wire" the transitions for this new letter, the recognized partition restricted to the old alphabet does not change. Thus by induction we can make the alphabet arbitrarily large. □

We only proved the existence of a gap, but said nothing about its size. Using larger l we can clearly "beat" even larger automata, while the maximal $\simeq_{\mathcal{L}}$-independent set still has size 2. Thus the same idea can be used to prove an arbitrarily large gap. On the other hand this gap also appears for smaller l as shown by the example in Section 5.1 ($L_2 = \{0^2 1^2\}$).

5.4 Negative results

In analogy to the previous two chapters we would like to present an efficient algorithm for minimizing a given $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$. Unfortunately the methodology used for partition and cover automata fails for lookup automata, which makes us wonder whether the problem is computationally intractable. In this section we will formalize the problem of $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ minimization and collect evidence for its computational hardness.

Problem: Minimum Equivalent $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$

Instance: Positive integers l and S, $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ A.

Question: Is there a $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ B with at most S states equivalent to A, *i.e.*, with $\mathcal{L}(A)|_{\Sigma^l} = \mathcal{L}(B)|_{\Sigma^l}$?

The problem discussed in the literature most similar to ours above is probably the Minimum Consistent DFA problem (also called Minimum Inferred Finite State Automaton in [GJ79]):

Problem: Minimum Consistent DFA

Instance: Finite alphabet Σ, finite sets $S, T \subseteq \Sigma^*$, positive integer K.

Question: Is there a DFA A with at most K states and both $S \subseteq L(A)$ and $T \cap L(A) = \emptyset$?

This problem is known to be NP-complete [Gol78] but solvable in polynomial time if $S \cup T = \Sigma^{\leq l}$ for some l as seen in Chapter 4. It remains NP-complete when restricted to the case $S \cup T \subseteq \Sigma^{\leq l}$ with $|\Sigma^{\leq l} \setminus (S \cup T)| \leq |\Sigma^{\leq l}|^\epsilon$ for some fixed $\epsilon > 0$ [Ang78]. Additionally for any constant k there can be no polynomial-time approximation algorithm finding a consistent DFA with less than opt^k states unless P = NP, where opt is the number of states of a minimum consistent DFA [PW93].

Our problem is similar to the case $S \cup T = \Sigma^l$ (for given l). However the words from S and T are not given explicitly but instead as a finite automaton which makes the problem not easier, as a suitable automaton can be constructed from given S and T in polynomial time. Being a special case of Minimum Consistent DFA does not help in deciding whether Minimum Equivalent $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ is NP-hard, polynomial time solvable, or in between (unless of course P = NP, which would make this entire section obsolete), but it does help in finding suitable literature where similar problems are handled.

A first classification is easily performed. From the section before we know how to test equivalence of $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$s in polynomial time, so there is a simple nondeterministic polynomial time *guess-and-check* algorithm for the Minimum Equivalent $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ problem.

Lemma 5.10. MINIMUM EQUIVALENT $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ *is in NP.*

Additionally a similar construction as in the proof of Theorem 4.21 can be used to show the minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ for a language partition $\mathcal{L}$ of Σ^l to be at most about l times smaller than a minimal $\mathrm{DP}_k\mathrm{A}$. Thus converting the given $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ to a $\mathrm{DP}_k\mathrm{A}$ using a product construction and minimizing the $\mathrm{DP}_k\mathrm{A}$ with any algorithm from Chapter 3 yields a simple approximation algorithm.

Lemma 5.11. *The optimization version of the* MINIMUM EQUIVALENT $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ *problem can be approximated in polynomial time by a factor of* $l+1$ *if* l *is given in unary.*

Next we will show that the class of algorithms used for $\mathrm{DP}_k\mathrm{As}$ and $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$ cannot be used for $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$. Therefore we introduce what we call *partition-and-merge* algorithms. These are algorithms on an automaton $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ working in two steps. The first phase calculates a partition $\mathcal{R} = (R_1, \ldots, R_r)$ of Q, while the second phase merges the states from each component of R into one new state. Merging here means that the new automaton is $(\Sigma, \mathcal{R}, R_1, \mathcal{P}_R, \gamma)$ where $q_0 \in R_1$ and $\mathcal{P}_R$ chosen arbitrarily. For γ we require *consistence*, *i.e.*, for given $a \in \Sigma$ the successor for an R_i must be chosen from the representatives of the successor states of one of its contained states, formally

$$\gamma(R_i, a) = R_j \;\Rightarrow\; \exists s \in R_i, t \in R_j : \delta(s, a) = t\,.$$

Looking back, all minimization algorithms presented so far are from this class. Contrary the MINIMUM EQUIVALENT $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ problem cannot be solved this way.

Lemma 5.12. *The* MINIMUM EQUIVALENT $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ *problem cannot be solved by a partition-and-merge algorithm.*

Proof. Consider the case $\Sigma = \{0, 1\}$, $l = 4$, $\mathcal{L} = (\Sigma^l \setminus \{0011\}, \{0011\})$. We know from Section 5.1 that the minimum $\mathrm{DP}_2\mathrm{L}_4\mathrm{A}$ for $\mathcal{L}$ has 3 states and is unique (shown in Figure 5.5). So if there is a partition-and-merge algorithm, each of the $\mathrm{DP}_2\mathrm{L}_4\mathrm{As}$ from Figure 5.4 with 4 states has to be reducible to the minimum one by merging. As they only have to loose one state, two states must be merged while for the other two states the transitions are fixed by the consistence requirement. However it is easily checked, that for any two states and any selection of transitions kept after merging, the result is never isomorphic to the minimum $\mathrm{DP}_2\mathrm{L}_4\mathrm{A}$ for $\mathcal{L}$. □

Another hint on the hardness of the MINIMUM EQUIVALENT $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ problem was already given in Section 5.3. For many problems the first step towards an efficient algorithm is some kind of duality theorem proving

equality for the optima between the primal and the dual problem. The most prominent example for this is probably the *max-flow min-cut theorem* for network flows, and here Theorem 4.13 can be interpreted as such a duality result. On the other hand adding restrictions to a problem, which force a gap between these optima, often makes the problem hard. An example is the *linear programming* problem, which is solvable in polynomial time but becomes NP-complete when searching for the integer optimum (known as *integer linear programming*). Besides becoming NP-complete also a duality gap appears[1]. So the gap identified in Theorem 5.9 could be another hint on the hardness of MINIMUM EQUIVALENT DP_kL_lA .

We finish this section by showing that for the case of IP addresses ($\Sigma = \{0,1\}$, $l \in \{32, 128\}$) the problem is (at least in theory) trivial.

Lemma 5.13. *For fixed alphabet Σ and fixed l the* MINIMUM EQUIVALENT DP_kL_lA *problem can be solved in constant time.*

Proof. Let $m := |\Sigma|$, then there is a DP_kA with at most $1 + \sum_{i=0}^{l} m^i \leq \max\{m^{l+1}, l+2\} =: N$ states. This is easily seen, as all words longer than l are equivalent and thus can be collected in a single state. For the remaining words we need at most one state per word. So it is sufficient to enumerate all DP_kL_lA with less than N states.

For a given number of states n we have n^{mn} ways to define the δ function. Given the state set Q and δ we can test the equivalence to the input DP_kL_lA by evaluating δ for each word in Σ^l for both automata and comparing the state class of the states reached. This also implies the state partition $\mathcal{P}$ which was not decided in advance.

Overall we have to check $\sum_{i=0}^{N-1} i^{mi}$ automata where each check needs $O(lm^l)$ steps, but this together is an (albeit large) constant. □

5.5 Heuristically reducing DP_kL_lA size

From the discussion in the previous section the DP_kL_lA minimization problem does not seem to be efficiently solvable. Nonetheless we are interested in reducing the size of DP_kL_lAs when using them for the representation of routing tables. While a minimal DP_kL_lA would be nice to have, for real world applications it might be sufficient to just calculate a *smaller* DP_kL_lA. So in this section we present a heuristic approach for transforming a given DP_kL_lA into an equivalent one with possibly less states. However we can give no guarantees on the amount of states lost here. To get an idea on the quality of these heuristics we apply them on some real world instances in Chapter 7.

The first idea might be to modify one of the partition-and-merge algorithms from Chapters 3 and 4. As we know from Lemma 5.12 such an

[1] Details on the mentioned problems and theorems can be found in [PS82]

algorithm will not be able to find a minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$, but maybe the resulting $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ is "small enough". Unfortunately the canonical relations on the states of a $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$, which are the core of such an algorithm, either are not "strong" enough (the resulting algorithm would be the same as $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization) or they do not have the required properties exploited by these algorithms (namely the right-invariance on the state level). Even if we could find a suitable relation, we still had to struggle to make an $O(n \log n)$ or even linear time algorithm of it, if this is possible at all, as the simpler quadratic algorithms are definitely to slow for the intended application with routing tables, where we have automata with more than 10^6 states. So instead we describe a different approach using the existing $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization algorithms. The key idea is captured in the next lemma.

Lemma 5.14. *Let Σ be an alphabet, l a positive integer, and $\mathcal{L}$ be a language k-partition of Σ^l. Then there is a k-partition $\hat{\mathcal{L}}$ of $\Sigma^{\leq l}$ such that the minimal DP_kC_lA for $\hat{\mathcal{L}}$ is a minimal DP_kL_lA for $\mathcal{L}$.*

Proof. Denote by $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ the minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ for $\mathcal{L}$. This A defines a component for each $w \in \Sigma^{<l}$ by $\chi_{\mathcal{P}}(\hat{\delta}(q_0, w))$ and thus the partition $\hat{\mathcal{L}}$. Obviously A is a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ for $\hat{\mathcal{L}}$, and as $\hat{\mathcal{L}}|_{\Sigma^l} = \mathcal{L}$ each $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ for $\hat{\mathcal{L}}$ is a $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ for $\mathcal{L}$. Thus A is also a minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ as otherwise a smaller $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ would exist. □

From the lemma we know that we can find the minimal $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ for a partition $\mathcal{L}$ by "guessing" the right component for each word shorter than l. This of course will not work for us, as not only we have no idea how to perform this guesswork, but also there are $\sum_{i=0}^{l-1} |\Sigma|^i$ words to be guessed which is usually exponential on the size of the automaton. However the words from $\Sigma^{<l}$ are already implicitly clustered by the state languages $L_q(A)$ of the given automaton. So the idea is to "guess" a new component in $\mathcal{P}$ for each state in Q and then use $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization on the resulting automaton. Of course this is only a rough approximation of what happens in Lemma 5.14, but we are just looking for a heuristic solution anyway. There are two aspects to be considered: we may not change the component for a state q which is reachable from q_0 by some word w of length l as this would affect the language partition recognized, and we have to define how guessing of the new component works.

For the first part we need another definition. We say some state q is *r-distant*, if there is a word $w \in \Sigma^r$ with $\hat{\delta}(q_0, w) = q$. The set of all r-distant states is denoted by $D_r(A)$. As said before, we may only modify the component for states from $Q \setminus D_l(A)$.

Lemma 5.15. *Let A be a finite automaton with n states on alphabet Σ, $r > 0$. We can calculate the sets $D_0(A), \ldots, D_r(A)$ in $O(|\Sigma| n r)$.*

Proof. We know $D_0(A)$ is just the initial state of A. Calculating $D_i(A)$ from $D_{i-1}(A)$ can be performed in $O(|\Sigma|n)$ in a straight forward fashion. □

Using a similar technique as in Section 5.2 the set D_r can also be calculated in $O(|\Sigma|n + n^3 \log r)$. However we are only interested in l-distant states, and as l is much smaller than n for IP routing tables, we prefer the method presented in the lemma above.

Now that we know how to identify the states we may not modify, we have to decide on how to guess the new component for each state. The simplest method would be to choose for each state the target component randomly from all available, but this does not work too well when there are many components. A better approach is to select one of the successor states and use the same component, thus allowing the $\mathrm{DP_kC}_l\mathrm{A}$ minimizer (which only merges states from the same component) to combine them. Additionally we handle the states "back-to-front", *i.e.*, in the order they appear in the sets $D_{l-1}(A), \ldots, D_0(A)$. The details are outlined in Algorithm 5.1.

Algorithm 5.1 A heuristic approach for $\mathrm{DP_kL}_l\mathrm{A}$ size reduction

Input: A $\mathrm{DP_kL}_l\mathrm{A}$ $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$
Output: A $\mathrm{DP_kL}_l\mathrm{A}$ B equivalent to A which is often smaller

Calculate $D_0(A), \ldots, D_l(A)$
$S := D_l(A)$
for $i = l - 1$ **down to** 0 **do**
  for each $q \in D_i(A) \setminus S$ **do**
    choose random $a \in \Sigma$ uniformly distributed
    move q to the same component of $\mathcal{P}$ as $\delta(q, a)$
  end for
  $S := S \cup D_i(A)$
end for
return the minimal equivalent $\mathrm{DP_kC}_l\mathrm{A}$ for the modified automaton

Lemma 5.16. *Let A be a DP_kL_lA with n states with an alphabet of size m. Algorithm 5.1 calculates an equivalent DP_kL_lA with at most n states in $O(mn(l + \log n))$ steps.*

Proof. The equivalence is seen from the fact that only states not in $D_l(A)$ are moved to another component, and the $\mathrm{DP_kC}_l\mathrm{A}$ minimizer preserves the assignment of all words in $\Sigma^{\leq l}$ which includes the words in Σ^l. Furthermore no step in the algorithm adds any states.

For the complexity we just combine Lemma 5.15 with Körner's minimization algorithm (Corollary 4.43). □

As it is hard to judge the quality of our heuristic approach from its description alone, we provide some results on routing table instances in

Section 7.2. As the algorithm has a $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ both as input and output we can apply it multiple times, and the results indicate that two or three repetitions are a good idea.

Chapter 6

Longest prefix matching using automata

Until now we have studied finite automata for several classes of partitions, but did not describe the connection between automata and the longest prefix matching problem on IP addresses, which we initially intended to solve. Closing this gap is the goal of this chapter, where we present several approaches of how a forwarding table can be represented as a finite automaton and how these can be constructed efficiently.

In Section 1.2 the input for constructing a forwarding table was given as a set of prefixes P, the next hops H, and a mapping $f : P \to H$. To simplify the construction of an automaton we assume $H = \{2, 3, \ldots, k\}$, reserving the answer 1 for the failed lookup, *i.e.*, if no matching prefix is in P. Next hop queries are only run on IP addresses, so the alphabet $\Sigma = \{0, 1\}$ and all query strings will have the same length l. Consequently we may assume all strings in P having length at most l as well.

To clarify the differences between the presented methods we use a common example throughout the entire chapter. The prefixes and their next hop are shown in Figure 6.1. To keep the example manageable we assume $l = 4$, so we can manually enumerate all possible IP addresses.

In the following sections we show how a finite automaton can be used as a forwarding table and address the problem of constructing such an automaton. The remaining sections describe a useful automaton transformation and

prefix	next hop
00	2
1	3
101	2
100	3

Figure 6.1: Prefixes and next hops for the example

IP address	next hop
0000	2
0001	2
0010	2
0011	2
0100	1
0101	1
0110	1
0111	1

IP address	next hop
1000	3
1001	3
1010	2
1011	2
1100	3
1101	3
1110	3
1111	3

Figure 6.2: Expanded IP table for the example

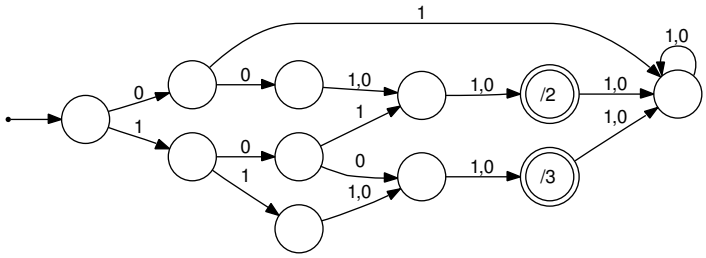


Figure 6.3: Expanded automaton for the example

a common trick to speed up the lookup process. Both of those techniques are rather simple but technical, so we will only give the intuitive idea and omit the algorithmic details and proofs.

As a side note we should mention, that we expect the transition function δ to be represented as a plain table, so we can manipulate it in constant time. Furthermore the automata constructed in this chapter usually are not minimal, but they can be used with the algorithms from the previous chapters to receive minimal ones.

6.1 Representing forwarding tables as automata

Basically there are two ways for storing the next hop information in an automaton. The more obvious one is probably just inserting the expanded IP table (see Figure 6.2) into an automaton, which we will call the *expanded automaton*. So the language to be decided by the automaton is $\mathcal{L} = (L_1, \ldots, L_k)$ where $L_i := \{w \in \Sigma^l \mid \text{next hop for } w \text{ is } i\}$. A suitable automaton is shown in Figure 6.3. The advantage of such an automaton is the simplified lookup. If the automaton is given by $(\Sigma, Q, q_0, \mathcal{P}, \delta)$ for an IP query w we just have to evaluate $\chi_{\mathcal{P}}(\hat{\delta}(q_0, w))$. However it is not obvious how to construct this automaton without determining the expanded IP table whose number of entries is exponential in l. Later we will see how this can be achieved in polynomial time.

An alternative approach is to only insert the prefixes into the automaton

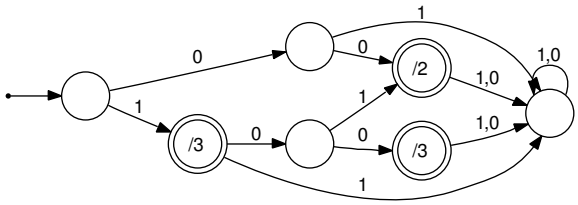


Figure 6.4: Prefix automaton for the example

Algorithm 6.1 Forwarding table lookup for a prefix automaton

Input: prefix automaton $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$,
IP address $w = w_1 w_2 \dots w_l$
Output: next hop for w (including 1 for unknown)

$q := q_0$
$h := \chi_{\mathcal{P}}(q)$
for $i := 1$ **to** l **do**
  $q := \delta(q, w_i)$
  if $\chi_{\mathcal{P}}(q) \neq 1$ **then**
    $h := \chi_{\mathcal{P}}(q)$
  end if
end for
return h

(here called *prefix automaton*), thus deciding $\mathcal{L} = (L_1, L_2, \dots, L_k)$ where $L_i := \{w \in P \mid f(w) = i\}$ for $i \geq 2$ and $L_1 := \Sigma^{\leq l} \setminus \bigcup_{i=2}^{k} L_i$. This is the method chosen by the existing trie-based algorithms although there are many deviations from the basic scheme. The advantage here is the often reduced size of such an automaton (Figure 6.4) over the expanded automaton. However we now have to evaluate $\chi_{\mathcal{P}}(\hat{\delta}(q_0, v))$ for every prefix v of a given IP address w. Of course some efficiency is gained by using intermediate results when using a method as the one shown in Algorithm 6.1.

Before heading to the construction algorithms we have to talk about the efficiency measure used. For algorithms which build or transform automata, we will use the usual asymptotic notation (Landau symbols). However for the lookup complexity we are more interested in the exact number of memory accesses required in the worst-case as explained in Section 1.2. We will later implement automata using a lookup table (two-dimensional array) for δ and an array containing the state class of each state for $\mathcal{P}$. Thus both a single transition $\delta(\cdot, \cdot)$ and a state component lookup $\chi_{\mathcal{P}}(\cdot)$ require a single memory access. We are discussing both operations separately nonetheless

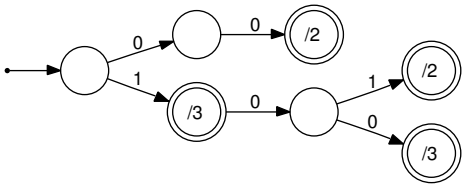


Figure 6.5: The trie for the example prefixes

as it aids in understanding which operation contributes most to the lookup time. For the expanded automaton a prefix lookup requires l transitions and one component lookup, while the prefix automaton requires l transitions and l component lookups.

6.2 Automaton construction using tries

Up to now all automata presented in this text simply appeared without any hint on how to construct them systematically. In this section we will address this issue. Our tool of choice is the *trie*, which was described in [Fre60] as a way to organize memory for efficient re*trie*val of data indexed by strings. Today the term trie is usually used for ordered trees where all edges are labeled with a character from some alphabet Σ ([GT01], pp. 429). The definition of a trie given here is motivated by the intended usage as an "incrementally constructed automaton".

Definition 6.1. *An automaton $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ is called* trie*, if there is a unique sink state $\perp \in P_1$ and for each $q \in Q \setminus \{\perp\}$ we have $|L_q(A)| = 1$.*

When drawing tries we will usually omit the state $\perp$ and all transitions to it, as the automaton then is much simpler to draw.

Remark 6.2. *The transition graph induced by the states $Q \setminus \{\perp\}$ of a trie is a directed tree with root q_0.*

We give no proof for this, but it is easy to see that if the transition graph was no tree, there had to be an undirected cycle which in turn would lead to at least one state q having $|L_q(A)| \neq 1$. This is best seen at the trie for the prefixes in our example as shown in Figure 6.5.

What makes tries especially useful is that inserting a word to a trie is as easy as performing a lookup. The only difference is the eventual addition of new states. This straight forward method is shown in Algorithm 6.2, the formal description follows.

Lemma 6.3. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a trie, $w \in \Sigma^*$ a word of length m, and $1 \leq i \leq |\mathcal{P}| + 1$. Applying Algorithm 6.2 on A, w, and i results in a trie $A' = (\Sigma, Q', q_0, \mathcal{P}', \delta')$ such that*

Algorithm 6.2 Adding a word to a trie

Input: Trie $(\Sigma, Q, q_0, \mathcal{P}, \delta)$ with sink state $\perp$
word $w = w_1 w_2 \dots w_m$ and desired component index i

$q := q_0$
for $j := 1$ **to** m **do**
  if $\delta(q, w_j) = \perp$ **then**
    {create new state q'}
    denote by q' some new state $q' \notin Q$
    $Q := Q \cup \{q'\}$, $P_1 := P_1 \cup \{q'\}$
    $\delta(q', a) := \perp$ for all $a \in \Sigma$
    $\delta(q, w_j) := q'$
  end if
  $q := \delta(q, w_j)$
end for
move state q to component P_i

1. $\forall v \in \Sigma^* \setminus \{w\} : \chi_{\mathcal{P}'}(\hat{\delta}'(q_0, v)) = \chi_{\mathcal{P}}(\hat{\delta}(q_0, v))$
2. $\chi_{\mathcal{P}'}(\hat{\delta}'(q_0, w)) = i$

thereby adding at most m new states and needing $O(m)$ steps.

Proof. From the way new states are initialized, it is obvious that once we start inserting states, we will never reach an "old" state $q \in Q$ but keep adding new states. Let $q_w := \hat{\delta}'(q_0, w)$. If $q_w \in Q$ (*i.e.*, q_w is not a new state), then all that changes is moving q_w to P_i. As A is a trie, $|L_{q_w}(A)| = 1$ and so $\chi_{\mathcal{P}}(\hat{\delta}(q_0, v)) \neq \chi_{\mathcal{P}'}(\hat{\delta}'(q_0, v))$ only for $v = w$.

Now let $q_w \in Q' \setminus Q$, so there were new states added. The first observation is that new states are added in a way which does not violate the trie criterion, so A' is still a trie. As all added states replace a transition to the sink state $\perp \in P_1$ and are also put into P_1 the accepted language is not modified, until we finally move the state in the last line. But this case was already discussed above. □

Corollary 6.4. *Let Σ be an alphabet, $P \subseteq \Sigma^*$ a given set of words, $\mathcal{H} = (H_1, \dots, H_k)$ a k-partition of P, and $n := \sum_{w \in P} |w|$. We can construct a trie A with $\mathcal{L}(A) = (H_1 \cup (\Sigma^* \setminus P), H_2, \dots, H_k)$ with at most $2 + n$ states using $O(n)$ time.*

Proof. Start with the empty trie $(\Sigma, \{q_0, \perp\}, q_0, (\{q_0, \perp\}), \delta)$ with $\delta(q, a) = \perp$ for all $q \in \{q_0, \perp\}$ and $a \in \Sigma$. Then use Algorithm 6.2 to add the words one by one. □

If the prefix automaton is represented as a trie, we can improve on the number of operations required for a single longest prefix match. As every

state except the sink state is only reachable from one path, we can easily determine for every state q the single word w with $\hat{\delta}(q_0, w) = q$, and adjust the state class of q to be the same as the result of a longest prefix matching of w. For example in Figure 6.5 we would move the state $\hat{\delta}(q_0, 01)$ to state class 3. This preprocessing step is called *leaf-pushing* in [SV99] and can be performed in linear time using depth first search on the transition graph. The benefit from this operation is a reduced number of operations for a single longest prefix match. For an IP address $w = w_1 \dots w_l$ now the result is either $\chi_{\mathcal{P}}(\hat{\delta}(q_0, w))$ if $\hat{\delta}(q_0, w) \neq \perp$ or $\chi_{\mathcal{P}}(q')$, where q' is the last state in $\hat{\delta}(q_0, w_1), \hat{\delta}(q_0, w_1 w_2), \dots, \hat{\delta}(q_0, w_1 w_2 \dots w_l)$ which is not $\perp$. So we can easily adjust Algorithm 6.1 to perform the lookup using at most l transitions and a single component lookup (opposed to l lookups without leaf-pushing).

6.3 Level shifting

After having seen how to efficiently construct the prefix automaton using a trie, we now look into constructing the expanded automaton by transforming the prefix automaton. We call the technique used *level shifting* as it modifies the levels of the automaton which carry information.

Definition 6.5. *An acyclic DP$_k$A $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ is called* leveled modulo m*, iff for each relevant state $q \in Q \setminus P_1$*

$$\text{level}(q) \equiv 0 \pmod{m}.$$

The goal of *level shifting modulo m* is to transform an arbitrary acyclic automaton A into an automaton B which is leveled modulo m and yields the same result as A for longest prefix matching of all words of length l. Obviously this is impossible if $m \nmid l$, as can be seen by having two prefixes of length l which only differ in the last character but are in different state classes.

We can easily implement level shifting as follows. Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a minimal acyclic DP$_k$A with sink state $\perp$ and $\text{level}(q) \leq l$ for all $q \in Q \setminus \{\perp\}$. Visit each state $q \in Q \setminus \{\perp\}$ in topological sorted order (so no state has a transition to one of the states already processed). If $\text{level}(q) \not\equiv 0 \pmod{m}$ and $q \notin P_1$ we have to move q to P_1. To ensure the longest prefix lookup to behave the same, we inspect the successor states for each letter $a \in \Sigma$. There are three cases (denote by P_q the component of $\mathcal{P}$ containing q):

1. $\delta(q, a) \in P_1 \setminus \{\perp\}$:
   Propagate the state class of q by moving $\delta(q, a)$ to P_q.

2. $\delta(q, a) \notin P_1$:
   There is nothing to do, as a longest prefix matching along this path would use the result of $\delta(q, a)$ anyway.

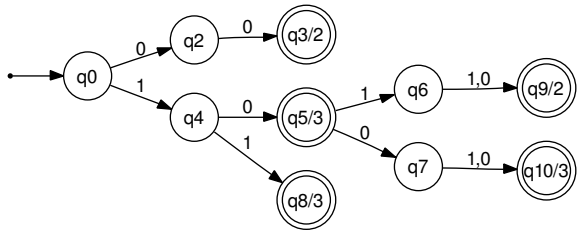


Figure 6.6: The trie after level shifting modulo 2

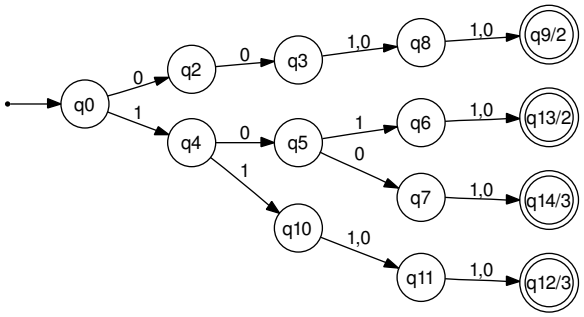


Figure 6.7: The trie after level shifting modulo 4

3. $\delta(q, a) = \perp$:
   Insert a new state q' to P_q with $\delta(q', b) = \perp$ for each $b \in \Sigma$ and set $\delta(q, a) = q'$.

It is not hard to verify that following these steps ensures the resulting automaton to be both leveled modulo m and equivalent according to longest prefix matching. As an example we applied level shifting to the trie from Figure 6.5. The results are shown in Figures 6.6 and 6.7. Looking at the way new states are added, it can be seen that they form chains of length at most $(m-1)$ leading to some state $q \notin P_1$. Thus managing these states carefully and reusing them where possible, allows level shifting to be performed with little overhead in the number of states added. This is captured by the next lemma which follows from the discussion above.

Lemma 6.6. *Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be a minimal acyclic DP_k^*A with n states. Level shifting modulo m can be performed in time $O((|\Sigma|+m)n)$ and by adding at most $(|\mathcal{P}|-1)(m-1)$ new states.*

Our initial task of transforming the prefix automaton to an expanded automaton is just level shifting modulo l. So as long as l is bounded by a polynomial in the size of the automaton, which is a valid assumption in the context of IP addresses, we have an efficient algorithm for constructing the expanded automaton.

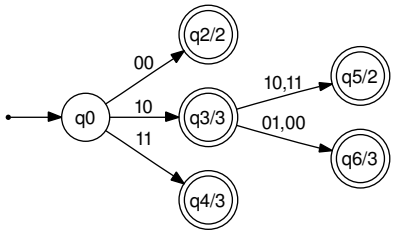


Figure 6.8: The trie after level shifting and alphabet expansion with $r = 2$

6.4 Alphabet expansion

Initially we have seen longest prefix matching on an expanded automaton to take up to l transitions and a single component lookup. Using leaf-pushing we reach these numbers also for the prefix automaton. So for the component lookups we are already optimal and to optimize further we have to address the number of transitions needed. The idea here is to interpret an IP address not as a sequence of l bits, but as a sequence of words in $\{0,1\}^r$. In the routing community this technique is labeled *multibit trie* ([RSBD01]), but as we apply it to general automata and it is essentially a modification of the alphabet we call it *alphabet expansion* here.

The actual operation is fairly easy to describe. Let $A = (\Sigma, Q, q_0, \mathcal{P}, \delta)$ be the input automaton and let $r \in \mathbb{N}$ be a compression factor. Alphabet expansion generates the automaton $B = (\Sigma^r, Q, q_0, \mathcal{P}, \delta')$ with $\delta'(q, w) = \hat{\delta}(q, w)$ for each $w \in \Sigma^r$ and $q \in Q$. The automaton B potentially includes many unused states which should be eliminated afterwards.

To be of any use for longest prefix matching we have to ensure $r \mid l$ as otherwise we will not be able to query IP addresses, and the automaton should be leveled modulo r so we do not loose information. Alphabet expansion for our running example is displayed in Figure 6.8.

After applying alphabet expansion on our prefix or expanded automaton, we can now perform longest prefix matching using only $\frac{l}{r}$ transitions and a single component lookup. Additionally the number of states is not increased from alphabet expansion (if we omit the growth from the preceding level shifting). However r has to be chosen carefully as the size of the alphabet grows exponentially in r, which in turn contributes linearly to the size of the automaton for many representations.

Chapter 7

Experimental results

To get an idea of the practical relevance of the theoretical work presented so far, we will provide some computational results on real world routing tables. Our main interest is the size of the resulting automaton after applying combinations of the preprocessing steps from the previous chapter and using the minimization algorithms for $\mathrm{DP}_k\mathrm{As}$, $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$, and $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$. We will also discuss the number of memory lookups required in the worst case for a single longest prefix matching.

It would be interesting to have results on the average lookup speed for real IP traffic as well, but unfortunately traffic data in conjunction with the routing table used is not publicly available. Additionally our approach relies on getting the automaton small enough to fit into cache memory. But the to get suitable measurements, the code has to be optimized for the cache architecture of the hardware used, which was not explored.

Another issue not yet addressed is the representation of an automaton in memory which of course influences its size. We will use a very simple model here, where $\Sigma = \{0, \ldots, m-1\}$ and $Q = \{0, \ldots, n-1\}$. The transition function δ then can be realized as a two-dimensional array storing the next state for each state letter pair, and the state partition $\mathcal{P}$ is a plain array, giving for each state the index of the component it belongs to. So for storing an automaton in memory we need $c(mn + n)$ bits, where c is the number of bits used for one array entry. Using this model every state transition and state class lookup translates to exactly one memory lookup. For this to work, the array elements might need further alignment, contributing to the size of the automaton even further, but as this again is machine dependant we will ignore it herein.

Next we give an overview on the data used for testing, followed by experimental results on this data, including a comparison with existing similar data structures. Then we close this chapter by interpreting these results and giving possible directions for further development.

Network	prefixes	next hops
AS 553 (BelWue)	115347	194
AS 852 (Telus - East Coast)	176985	1
AS 852 (Telus - West Coast)	177040	1
AS 3257 (Tiscali)	176736	1
AS 3561 (Savvis Communications)	192200	2
AS 3741 (Internet Solutions)	844	1
AS 4323 (Time Warner Telecom)	180022	8
AS 5388 (Planet Online)	177456	1
AS 5511 (Opentransit)	180211	10
AS 6539 (GT Group Telecom)	178511	14
AS 6648 (Bayantel Inc.)	1974	1
AS 6667 (Eunet Finland)	176840	1
AS 6730 (Sunrise)	89761	51
AS 6939 (Hurricane Electric)	170807	393
AS 7474 (Optus Route Server Australia)	177945	10
AS 7911 (Wiltel)	142038	625
AS 8220 (Colt Internet)	187117	1
AS 9132 (Broadnet Mediascape Communications AG)	177190	413
AS 12312 (Tiscali Germany)	175896	1
AS 13645 (Host.net)	177404	2
AS 15290 (Allstream - East)	184812	7
AS 15290 (Allstream - West)	184743	8

Figure 7.1: Characteristics of the real-world routing tables used

7.1 Test data

We intend to evaluate the algorithms on real-world data, so we extracted routing tables from the routers listed at [Ker06]. All tables were collected at 3/31/2006 and preprocessed to eliminate duplicate entries. An overview on the characteristics of these tables is given in Figure 7.1. Routing tables listed in [Ker06] which are not included here could not be collected due to technical problems such as time-outs.

To get an idea of the distribution of prefix lengths in a typical routing table, we counted the number of prefixes for each possible length over all those routing tables. The result is shown in Figure 7.2 using a logarithmic scale. Obviously most prefixes have a length between 16 and 24 with notable peaks at lengths 8, 16, and 24, which are the sizes of the network part for class A, B, and C networks used before the introduction of CIDR.

Additionally we are using the routing tables from [NK98] which are also similar to those from [DBCP97]. They are listed in Figure 7.3 and allows us to compare our algorithms with theirs without having access to their code.

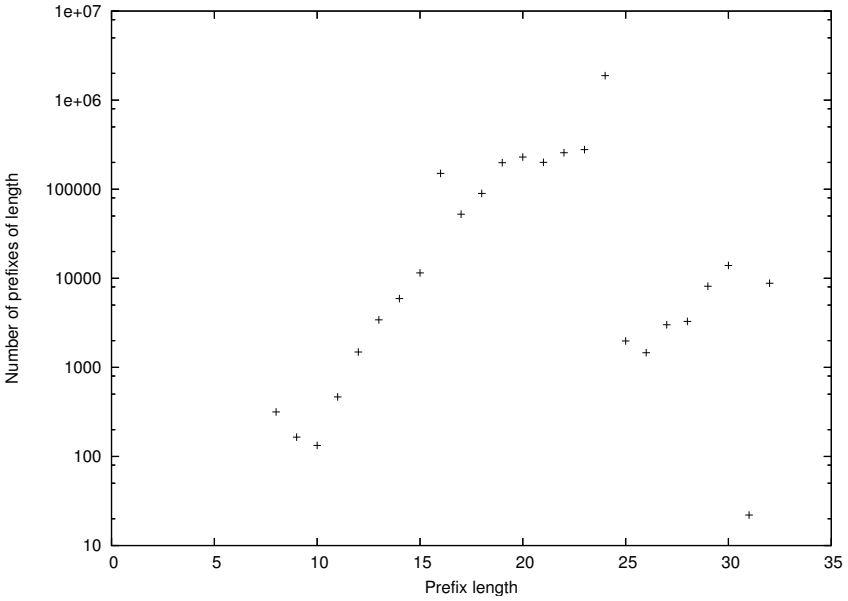


Figure 7.2: Number of prefixes of given length, aggregated over all real-world routing tables used, logarithmically scaled

Site	collection date	prefixes	next hops
AADS	08/24/97	20218	19
Mae East	10/30/97	38094	59
Mae West	10/30/97	14960	57
Pac Bell	08/24/97	20533	3

Figure 7.3: Characteristics of the routing tables used in [NK98]

7.2 Results

All previous chapters provided us with a variety of transformations and algorithms we can apply to a forwarding table when represented as a finite automaton. Chapter 6 introduced the differentiation into prefix automata and expanded automata which is reflected in separate tables here. For the prefix automata we always use leaf-pushing, so the number of worst-case memory lookups is the same for both prefix and expanded automaton and only depends on r, the level of alphabet expansion used (see Figure 7.4). As l is considered constant, for p given prefixes all tries used can be constructed in $O(2^r p)$ steps, as indicated by the results in Chapter 6. For minimizing these tries we always used the most efficient algorithm available, *i.e.*, $O(2^r n)$ for $\mathrm{DP}_k\mathrm{As}$, $O(2^r n \log n)$ for $\mathrm{DP}_k\mathrm{C}_l\mathrm{As}$, and the $O(2^r n \log n)$ heuristic for $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$, where $n = O(p)$ is the number of states of the constructed trie.

For each automaton we give both the number of states and the size

Level of alphabet expansion	number of memory lookups (worst case)
none ($r = 1$)	33
$r = 2$	17
$r = 4$	9
$r = 8$	5

Figure 7.4: Number of memory lookups required for a forwarding table after alphabet expansion (IPv4)

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AS 553	Trie	326948	3832	100.00%	179788	3512	100.00%	106937	7102	100.00%	69860	70133	100.00%
	DPA	77034	903	23.56%	34692	678	19.30%	14977	995	14.01%	5400	5422	7.73%
	DPCA	77030	903	23.56%	34692	678	19.30%	14972	995	14.00%	5397	5419	7.73%
AS 3741	Trie	3563	42	100.00%	1923	38	100.00%	1104	74	100.00%	728	731	100.00%
	DPA	1422	17	39.91%	675	14	35.10%	339	23	30.71%	131	132	17.99%
	DPCA	1421	17	39.88%	674	14	35.05%	339	23	30.71%	131	132	17.99%
AS 5511	Trie	499947	5859	100.00%	278942	5449	100.00%	171199	11369	100.00%	119203	119669	100.00%
	DPA	96662	1133	19.33%	42714	835	15.31%	19554	1299	11.42%	6490	6516	5.44%
	DPCA	96625	1133	19.33%	42702	835	15.31%	19547	1299	11.42%	6487	6513	5.44%
AS 6667	Trie	488439	5724	100.00%	272438	5322	100.00%	166816	11078	100.00%	115602	116054	100.00%
	DPA	72601	851	14.86%	30269	592	11.11%	12480	829	7.48%	3806	3821	3.29%
	DPCA	72601	851	14.86%	30269	592	11.11%	12480	829	7.48%	3806	3821	3.29%
AS 7911	Trie	410468	4811	100.00%	228084	4455	100.00%	138076	9170	100.00%	94471	94841	100.00%
	DPA	128348	1505	31.27%	61429	1200	26.93%	28280	1878	20.48%	8201	8234	8.68%
	DPCA	128348	1505	31.27%	61429	1200	26.93%	28280	1878	20.48%	8201	8234	8.68%

Figure 7.5: Results for a subset of the test-data (prefix automaton)

of a memory representation in kB. Additionally we give the relative size to the simple trie. For calculating the memory requirements we assume the representation mentioned in the beginning of this chapter, using 32-bit integer values as table entries. So the value of the *Mem* column is $\lceil \frac{n2^r+n}{256} \rceil$, where n is the number of states and r the level of alphabet expansion.

We will give results only for a subset of the routing tables here (those written in italics in Figure 7.1), so we can be more focused on them. The instances were chosen to have some variety in the number of prefixes and next hops. Complete results for all instances are provided in Appendix A.

For the prefix automata we applied both $\mathrm{DP}_k\mathrm{A}$ and $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization to the trie for several levels of alphabet expansion. The value l is chosen as $32/r$, where r denotes the level of prefix expansion. In Figure 7.5 the corresponding numbers are shown.

Using this data we observe several trends, most of them not too surprising. At first, routing tables with a little number of next hops (AS 5511, AS 6667) can be compressed better than those with large variety in the next hops (AS 553, AS 7911), as states are more "likely" to be equivalent in the former case. Furthermore sparsely populated tables (AS 3741) do not compress as well as the other denser routing tables. Finally compressibility of the automata seems to improve with alphabet expansion. Probably this operation has some kind of a "smoothing" effect on an automaton. While the difference in size between the trie and the minimal $\mathrm{DP}_k\mathrm{A}$ is pretty ob-

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AS 553	Trie	329906	3867	100.00%	181141	3538	100.00%	107566	7144	100.00%	70111	70385	100.00%
	DPA	56577	664	17.15%	28352	554	15.65%	13969	928	12.99%	5579	5601	7.96%
	DPLA1	50236	589	15.23%	25732	503	14.21%	12987	863	12.07%	5276	5297	7.53%
	DPLA2	48878	573	14.82%	25328	495	13.98%	12905	857	12.00%	5276	5297	7.53%
	DPLA3	48030	563	14.56%	25137	491	13.88%	12888	856	11.98%	5276	5297	7.53%
	DPLA4	47518	557	14.40%	25050	490	13.83%	12875	855	11.97%	5276	5297	7.53%
AS 3741	Trie	3582	42	100.00%	1932	38	100.00%	1108	74	100.00%	730	733	100.00%
	DPA	1277	15	35.65%	625	13	32.35%	327	22	29.51%	126	127	17.26%
	DPLA1	1149	14	32.08%	570	12	29.50%	306	21	27.62%	123	124	16.85%
	DPLA2	1125	14	31.41%	543	11	28.11%	300	20	27.08%	123	124	16.85%
	DPLA3	1124	14	31.38%	541	11	28.00%	298	20	26.90%	123	124	16.85%
	DPLA4	1083	13	30.23%	541	11	28.00%	298	20	26.90%	123	124	16.85%
AS 5511	Trie	500152	5862	100.00%	279037	5450	100.00%	171243	11372	100.00%	119222	119688	100.00%
	DPA	64155	752	12.83%	32115	628	11.51%	17442	1159	10.19%	6400	6425	5.37%
	DPLA1	60663	711	12.13%	30580	598	10.96%	17016	1130	9.94%	6362	6387	5.34%
	DPLA2	59171	694	11.83%	30122	589	10.79%	16907	1123	9.87%	6355	6380	5.33%
	DPLA3	58476	686	11.69%	29862	584	10.70%	16866	1121	9.85%	6354	6379	5.33%
	DPLA4	57383	673	11.47%	29741	581	10.66%	16847	1119	9.84%	6354	6379	5.33%
AS 6667	Trie	488463	5725	100.00%	272449	5322	100.00%	166821	11078	100.00%	115604	116056	100.00%
	DPA	33490	393	6.86%	17265	338	6.34%	9241	614	5.54%	3710	3725	3.21%
	DPLA1	32371	380	6.63%	16742	327	6.15%	8988	597	5.39%	3701	3716	3.20%
	DPLA2	31301	367	6.41%	16433	321	6.03%	8919	593	5.35%	3701	3716	3.20%
	DPLA3	30829	362	6.31%	16357	320	6.00%	8888	591	5.33%	3701	3716	3.20%
	DPLA4	30542	358	6.25%	16263	318	5.97%	8869	589	5.32%	3701	3716	3.20%
AS 7911	Trie	417843	4897	100.00%	231491	4522	100.00%	139650	9274	100.00%	95150	95522	100.00%
	DPA	115599	1355	27.67%	57997	1133	25.05%	28487	1892	20.40%	8766	8801	9.21%
	DPLA1	101552	1191	24.30%	52260	1021	22.58%	26512	1761	18.98%	8004	8036	8.41%
	DPLA2	99029	1161	23.70%	51520	1007	22.26%	26420	1755	18.92%	8004	8036	8.41%
	DPLA3	97672	1145	23.38%	51165	1000	22.10%	26391	1753	18.90%	8004	8036	8.41%
	DPLA4	97018	1137	23.22%	51019	997	22.04%	26375	1752	18.89%	8004	8036	8.41%

Figure 7.6: Results for a subset of the test-data (expanded automaton)

vious, unfortunately the minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ only saves a couple of states (if any) over the $\mathrm{DP}_k\mathrm{A}$. It seems that the very regular structure exploited by a $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ is rarely seen in practice (at least for routing tables).

The next table (Figure 7.6) summarizes the results for the expanded automaton as both a trie and minimal $\mathrm{DP}_k\mathrm{A}$. The DPLAn rows represent the automaton after n iterations of the $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ minimizing heuristic from Section 5.5. Values for a minimal $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ were omitted here, as the structure of the expanded automaton (all states with level $< l$ are in the first component of $\mathcal{P}$) does not permit any further compression than the $\mathrm{DP}_k\mathrm{A}$ (only the sink state can be merged in some cases) and thus provides no new information.

Basically we encounter the same trends as for the prefix automaton (compression factor increases with density and level of alphabet expansion, but decreases with number of next hops). What we are more interested in here, are the results for the $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$s. The numbers support the usefulness of $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ as they indeed can reduce automaton size somewhat. Multiple iterations of the proposed heuristic approach yield better compression, although the first iteration is the most effective. The amount of $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ compression decreases with increasing level of alphabet expansion, as then the value of $l = 32/r$ is reduced resulting in less "inter-level" state merges. It is interesting that while constructing the expanded automaton by level shifting increases the size of the trie somewhat, the corresponding minimal $\mathrm{DP}_k\mathrm{A}$ often is remarkably smaller than the one for the prefix automaton.

As announced we want to compare our results to the approaches from

Approach used	memory lookups	AADS (in kB)	Mae East (in kB)	Mae West (in kB)	Pac Bell (in kB)
Luleå-trie [DBCP97]	12	28	160	86	99
LC-trie [NK98]	6	672	1024	553	680
DP_kA, $r = 2$	17	112	223	137	89
DP_kA, $r = 4$	9	201	391	243	157
DP_kA, $r = 8$	5	1191	2285	1503	885
DP_kA, $r = 2$, BV	10	240	351	265	217
DP_kA, $r = 4$, BV	6	329	519	371	285

Figure 7.7: Comparison of results with [NK98] and [DBCP97] (the DP_kA is a minimal prefix automaton, BV = with base vector). The number of memory lookups denotes the worst-case.

[DBCP97] (Luleå-trie) and [NK98] (LC-trie) which are structurally similar. Therefore we calculated the tables for prefix and expanded automata of the routing tables from Figure 7.3. Here will only use selected numbers, the full results can be found in Appendix A. The first two rows of Figure 7.7 give the number of memory lookups and the sizes of the forwarding tables as provided in the original papers[1]. Next are the values for minimal DP_kAs of alphabet expanded prefix automata. As the number of states permits this, we are assuming 16-bit integers here, so the memory consumption is about half of that shown in the appendix. The description of the remaining lines follows later.

Apparently none of our automata is smaller than the Luleå-trie, although we easily can beat it in terms of memory lookups. To have less lookups than the LC-trie we have to perform alphabet expansion with $r = 8$ which makes our automata larger than them. To understand the reason for this, we should look at the techniques used in these papers.

The Luleå-trie heavily relies on efficiently packing all values into bit-level structures which are suitably aligned in memory. While this obviously gives good compression results, it makes scaling very hard, as for larger routing tables not only a single parameter, but the entire algorithm has to be modified. We could try to find a more memory conserving representation for an automaton, but this is likely to increase the number of memory lookups we have to perform, so we did not explore it in more detail.

To reduce the number of memory lookups the LC-tries are using another trick. Speaking in the terms defined here, they apply a different level of alphabet expansion to different depths of the trie. So for the first depth level a large alphabet is used allowing a large step, while the memory cost is relatively small as only one state is affected. For the next level a smaller alphabet is used as more states are at this level. Taken to the extreme

[1]The numbers for [DBCP97] refer to slightly older versions of the tables used. In [NK98] no exact size in bytes is given, but using the provided data is it easily calculated.

we could replace the initial state of an automaton with a lookup table for all possible 16-bit prefixes (called *base vector* in [NK98]), then precalculate and store the resulting state in there. For an automaton with alphabet expansion of level r this will replace the first $16/r$ transitions with a single lookup, saving $16/r - 1$ memory lookups. This increase in lookup speed is paid for with additional 128 kB (16-bit state number for each of 2^{16} prefixes). The results we would retrieve when including a base vector are reported in the last two lines of Figure 7.7. We could even have saved some memory by removing all states no longer needed. But even without removing them, we now easily beat the LC-trie in size while having the same low number of memory lookups.

7.3 Conclusion

In this thesis we have seen several types of automata, studied their structure, and found algorithms for their minimization (or at least size reduction in the case of $\mathrm{DP}_k\mathrm{L}_l\mathrm{As}$). Together with the transformations introduced in Chapter 6 we have a rich toolkit for solving the longest prefix matching problem, which is the core problem for IP packet classification. Using these building blocks we can tailor the construction of forwarding tables, finding a compromise between memory size and lookup speed.

It is hard to decide whether the $\mathrm{DP}_k\mathrm{A}$ or the often smaller $\mathrm{DP}_k\mathrm{L}_l\mathrm{A}$ should be used for practical applications as the worst-case lookup time is the same for both. The decision depends on whether the reduced automaton size justifies the increased running time of a minimization algorithm ($O(n)$ versus $O(n \log n)$), which in turn is influenced by technical prerequisites and requirements. Our results indicate both approaches to yield a more compact representation of forwarding tables than the LC-trie having comparable worst-case lookup times when used in conjunction with a base vector.

Further studies could target several directions. At first it would be interesting to compare the average-case lookup times between the different automata presented so far. This requires either access to a large sequence of real world prefix lookups or a suitable model for the distribution of host lookups in a router.

Additionally it might be worthwhile to find and apply more compact memory representations for the given automata. Due to the requirement of reducing the number of memory lookups this might include adaptions on a given caching architecture.

The poor compression results for the $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ on concrete instances despite its theoretical superiority to the $\mathrm{DP}_k\mathrm{A}$ are unfortunate. Maybe we can modify the automaton without affecting the forwarding table it represents in a similar manner as in Section 5.5 to make it more accessible to $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization.

Finally it remains an open problem to decide the complexity for MINIMUM EQUIVALENT DP_kL_lA . From the results seen so far we doubt it to be solvable in polynomial time, however an NP-completeness proof was not found either.

Bibliography

[Ang78] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[CGH05] J.-M. Champarnaud, F. Guingne, and G. Hansel. Similarity relations and cover automata. *Theoretical Informatics and Applications*, 39(1):115–123, 2005.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[CPY02] C. Câmpeanu, A. Paun, and S. Yu. An efficient algorithm for constructing minimal cover automata for finite languages. *International Journal of Foundations of Computer Science*, 13(1):83–97, 2002.

[CSY01] C. Câmpeanu, N. Sântean, and S. Yu. Minimal cover-automata for finite languages. *Theoretical Computer Science*, 267(1-2):3–16, 2001.

[Dac03] J. Daciuk. Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. In *Proceedings of the 7th International Conference on Implementation and Application of Automata (CIAA '02)*, volume 2608 of *Lecture Notes in Computer Science*, pages 255–261. Springer-Verlag, 2003.

[DBCP97] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '97)*, pages 3–14. ACM Press, 1997.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[FLYV93] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519, September 1993.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[GH04] J.-C. Grégoire and Angèle M. Hamel. You can get there from here: Routing in the internet. In *1st Workshop on Combinatorial and Algorithmic Aspects of Networking (CAAN 2004)*, volume 3405 of *Lecture Notes in Computer Science*, pages 173–182. Springer-Verlag, 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[Gri73] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[GT01] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples.* John Wiley & Sons, 2001.

[HD03] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513 (Proposed Standard), April 2003.

[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations.*, pages 189–196. Academic Press, 1971.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley Publishing Company, 1979.

[Ker06] T. Kernen. traceroute.org web site. http://www.traceroute.org, 2006. Route Servers section.

[Kör03] H. Körner. A time and space efficient algorithm for minimizing cover automata for finite languages. *International Journal of Foundations of Computer Science*, 14(6):1071–1086, 2003.

[KR02] J. F. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison-Wesley Publishing Company, 2002.

[LSV99] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking*, 7(3):324–334, 1999.

[NK98] S. Nilsson and G. Karlsson. Fast address look-up for internet routers. In *Proceedings of the 4th International Conference on Broadband Communications (BC '98)*, pages 11–22. Chapman & Hall, Ltd., 1998.

[Odl03] A. M. Odlyzko. Internet traffic growth: sources and implications. In *Proceedings of Optical Transmission Systems and Equipment for WDM Networking II*, volume 5247, pages 1–15. SPIE The International Society for Optical Engineering, 2003.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: Algorithms and complexity.* Prentice-Hall, 1982.

[PW93] L. Pitt and M.K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the ACM*, 40(1):95–142, 1993.

[Rev91] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92:181–189, 1991.

[RSBD01] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2), 2001.

[SV99] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, 1999.

[Wat94] B. W. Watson. A taxonomy of finite automata minimization algorithms. Technical report, Eindhoven University of Technology, 1994.

[WVTP97] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing table lookups. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 25–36, September 1997.

Appendix A

Detailed results

In this appendix we provide the detailed results omitted from Section 7.2. All numbers given there are also covered in these tables to provide a complete overview.

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AADS	Trie	99398	1165	100.00%	53530	1046	100.00%	30690	2039	100.00%	20131	20210	100.00%
	DPA	22397	263	22.53%	11465	224	21.42%	6039	402	19.68%	2372	2382	11.78%
	DPCA	22397	263	22.53%	11465	224	21.42%	6039	402	19.68%	2372	2382	11.78%
Mae East	Trie	172215	2019	100.00%	93065	1818	100.00%	53718	3568	100.00%	35777	35917	100.00%
	DPA	44479	522	25.83%	22736	445	24.43%	11753	781	21.88%	4551	4569	12.72%
	DPCA	44479	522	25.83%	22736	445	24.43%	11753	781	21.88%	4551	4569	12.72%
Mae West	Trie	80864	948	100.00%	43064	842	100.00%	24119	1602	100.00%	15128	15188	100.00%
	DPA	26906	316	33.27%	14028	274	32.57%	7294	485	30.24%	2994	3006	19.79%
	DPCA	26906	316	33.27%	14028	274	32.57%	7294	485	30.24%	2993	3005	19.78%
Pac Bell	Trie	99073	1162	100.00%	53496	1045	100.00%	30840	2048	100.00%	20418	20498	100.00%
	DPA	17653	207	17.82%	9050	177	16.92%	4712	313	15.28%	1762	1769	8.63%
	DPCA	17653	207	17.82%	9050	177	16.92%	4712	313	15.28%	1762	1769	8.63%

Figure A.1: Results for routing tables from [NK98] (prefix automaton)

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AADS	Trie	99735	1169	100.00%	53682	1049	100.00%	30761	2043	100.00%	20159	20238	100.00%
	DPA	23683	278	23.75%	11771	230	21.93%	6143	408	19.97%	2393	2403	11.87%
	DPLA1	21363	251	21.42%	11018	216	20.52%	5923	394	19.25%	2355	2365	11.68%
	DPLA2	20959	246	21.01%	10811	212	20.14%	5885	391	19.13%	2355	2365	11.68%
	DPLA3	20438	240	20.49%	10752	210	20.03%	5874	391	19.10%	2355	2365	11.68%
	DPLA4	20311	239	20.36%	10729	210	19.99%	5866	390	19.07%	2355	2365	11.68%
Mae East	Trie	173132	2029	100.00%	93484	1826	100.00%	53913	3581	100.00%	35856	35997	100.00%
	DPA	47033	552	27.17%	23501	460	25.14%	11928	793	22.12%	4621	4640	12.89%
	DPLA1	43296	508	25.01%	22116	432	23.66%	11488	763	21.31%	4513	4531	12.59%
	DPLA2	41204	483	23.80%	21711	425	23.22%	11441	760	21.22%	4511	4529	12.58%
	DPLA3	40697	477	23.51%	21558	422	23.06%	11430	760	21.20%	4511	4529	12.58%
	DPLA4	40533	475	23.41%	21461	420	22.96%	11427	759	21.20%	4511	4529	12.58%
Mae West	Trie	81797	959	100.00%	43485	850	100.00%	24312	1615	100.00%	15203	15263	100.00%
	DPA	31178	366	38.12%	15552	304	35.76%	7688	511	31.62%	3071	3083	20.20%
	DPLA1	27498	323	33.62%	14151	277	32.54%	7258	482	29.85%	2966	2978	19.51%
	DPLA2	26381	310	32.25%	13853	271	31.86%	7193	478	29.59%	2965	2977	19.50%
	DPLA3	25903	304	31.67%	13714	268	31.54%	7181	477	29.54%	2965	2977	19.50%
	DPLA4	25616	301	31.32%	13647	267	31.38%	7177	477	29.52%	2965	2977	19.50%
Pac Bell	Trie	99134	1162	100.00%	53524	1046	100.00%	30853	2049	100.00%	20423	20503	100.00%
	DPA	17293	203	17.44%	8720	171	16.29%	4646	309	15.06%	1762	1769	8.63%
	DPLA1	16498	194	16.64%	8376	164	15.65%	4517	300	14.64%	1754	1761	8.59%
	DPLA2	15741	185	15.88%	8202	161	15.32%	4497	299	14.58%	1754	1761	8.59%
	DPLA3	15715	185	15.85%	8187	160	15.30%	4486	298	14.54%	1754	1761	8.59%
	DPLA4	15706	185	15.84%	8184	160	15.29%	4483	298	14.53%	1754	1761	8.59%

Figure A.2: Results for routing tables from [NK98] (expanded automaton)

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AS 553	Trie	326948	3832	100.00%	179788	3512	100.00%	106937	7102	100.00%	69860	70133	100.00%
	DPA	77034	903	23.56%	34692	678	19.30%	14977	995	14.01%	5400	5422	7.73%
	DPCA	77030	903	23.56%	34692	678	19.30%	14972	995	14.00%	5397	5419	7.73%
AS 852 (east)	Trie	489218	5734	100.00%	273012	5333	100.00%	167196	11103	100.00%	115949	116402	100.00%
	DPA	72648	852	14.85%	30254	591	11.08%	12455	828	7.45%	3738	3753	3.22%
	DPCA	72647	852	14.85%	30253	591	11.08%	12455	828	7.45%	3738	3753	3.22%
AS 852 (west)	Trie	489323	5735	100.00%	273073	5334	100.00%	167236	11106	100.00%	115984	116438	100.00%
	DPA	72661	852	14.85%	30259	591	11.08%	12457	828	7.45%	3737	3752	3.22%
	DPCA	72660	852	14.85%	30258	591	11.08%	12457	828	7.45%	3737	3752	3.22%
AS 3257	Trie	487945	5719	100.00%	272343	5320	100.00%	166851	11080	100.00%	115749	116202	100.00%
	DPA	72402	849	14.84%	30140	589	11.07%	12401	824	7.43%	3732	3747	3.22%
	DPCA	72402	849	14.84%	30140	589	11.07%	12401	824	7.43%	3732	3747	3.22%
AS 3561	Trie	528531	6194	100.00%	295656	5775	100.00%	180242	11970	100.00%	124443	124930	100.00%
	DPA	78313	918	14.82%	32617	638	11.03%	13387	889	7.43%	4024	4040	3.23%
	DPCA	78182	917	14.79%	32598	637	11.03%	13380	889	7.42%	4024	4040	3.23%
AS 3741	Trie	3563	42	100.00%	1923	38	100.00%	1104	74	100.00%	728	731	100.00%
	DPA	1422	17	39.91%	675	14	35.10%	339	23	30.71%	131	132	17.99%
	DPCA	1421	17	39.88%	674	14	35.05%	339	23	30.71%	131	132	17.99%
AS 4323	Trie	494317	5793	100.00%	276274	5396	100.00%	169702	11270	100.00%	118277	118740	100.00%
	DPA	101376	1188	20.51%	45187	883	16.36%	21287	1414	12.54%	7007	7035	5.92%
	DPCA	101376	1188	20.51%	45187	883	16.36%	21287	1414	12.54%	7007	7035	5.92%
AS 5388	Trie	490557	5749	100.00%	273736	5347	100.00%	167690	11136	100.00%	116332	116787	100.00%
	DPA	72867	854	14.85%	30377	594	11.10%	12507	831	7.46%	3791	3806	3.26%
	DPCA	72863	854	14.85%	30375	594	11.10%	12507	831	7.46%	3791	3806	3.26%
AS 5511	Trie	499947	5859	100.00%	278942	5449	100.00%	171199	11369	100.00%	119203	119669	100.00%
	DPA	96662	1133	19.33%	42714	835	15.31%	19554	1299	11.42%	6490	6516	5.44%
	DPCA	96625	1133	19.33%	42702	835	15.31%	19547	1299	11.42%	6487	6513	5.44%
AS 6539	Trie	492772	5775	100.00%	275085	5373	100.00%	168580	11195	100.00%	117188	117646	100.00%
	DPA	97216	1140	19.73%	43009	841	15.63%	19668	1307	11.67%	6634	6660	5.66%
	DPCA	97216	1140	19.73%	43009	841	15.63%	19668	1307	11.67%	6634	6660	5.66%
AS 6648	Trie	6425	76	100.00%	3637	72	100.00%	2270	151	100.00%	1664	1671	100.00%
	DPA	2077	25	32.33%	989	20	27.19%	488	33	21.50%	224	225	13.46%
	DPCA	2077	25	32.33%	989	20	27.19%	488	33	21.50%	224	225	13.46%
AS 6667	Trie	488439	5724	100.00%	272438	5322	100.00%	166816	11078	100.00%	115602	116054	100.00%
	DPA	72601	851	14.86%	30269	592	11.11%	12480	829	7.48%	3806	3821	3.29%
	DPCA	72601	851	14.86%	30269	592	11.11%	12480	829	7.48%	3806	3821	3.29%
AS 6730	Trie	256529	3007	100.00%	142997	2793	100.00%	86967	5776	100.00%	60847	61085	100.00%
	DPA	65309	766	25.46%	29975	586	20.96%	14216	945	16.35%	3779	3794	6.21%
	DPCA	65309	766	25.46%	29975	586	20.96%	14216	945	16.35%	3779	3794	6.21%
AS 6939	Trie	476489	5584	100.00%	266021	5196	100.00%	162757	10809	100.00%	112822	113263	100.00%
	DPA	94969	1113	19.93%	42067	822	15.81%	18787	1248	11.54%	6138	6162	5.44%
	DPCA	94969	1113	19.93%	42067	822	15.81%	18787	1248	11.54%	6138	6162	5.44%
AS 7474	Trie	491517	5760	100.00%	274217	5356	100.00%	167880	11149	100.00%	116131	116585	100.00%
	DPA	117863	1382	23.98%	54896	1073	20.02%	28083	1865	16.73%	8592	8626	7.40%
	DPCA	117861	1382	23.98%	54896	1073	20.02%	28083	1865	16.73%	8592	8626	7.40%
AS 7911	Trie	410468	4811	100.00%	228084	4455	100.00%	138076	9170	100.00%	94471	94841	100.00%
	DPA	128348	1505	31.27%	61429	1200	26.93%	28280	1878	20.48%	8201	8234	8.68%
	DPCA	128348	1505	31.27%	61429	1200	26.93%	28280	1878	20.48%	8201	8234	8.68%
AS 8220	Trie	515156	6037	100.00%	286924	5604	100.00%	173883	11547	100.00%	118394	118857	100.00%
	DPA	76153	893	14.78%	31699	620	11.05%	13155	874	7.57%	4012	4028	3.39%
	DPCA	76130	893	14.78%	31693	620	11.05%	13155	874	7.57%	4012	4028	3.39%
AS 9132	Trie	488899	5730	100.00%	273127	5335	100.00%	167375	11115	100.00%	116173	116627	100.00%
	DPA	121241	1421	24.80%	56183	1098	20.57%	25312	1681	15.12%	7850	7881	6.76%
	DPCA	121240	1421	24.80%	56183	1098	20.57%	25312	1681	15.12%	7850	7881	6.76%
AS 12312	Trie	485997	5696	100.00%	271107	5296	100.00%	165944	11020	100.00%	114927	115376	100.00%
	DPA	72260	847	14.87%	30097	588	11.10%	12394	824	7.47%	3728	3743	3.24%
	DPCA	72260	847	14.87%	30097	588	11.10%	12394	824	7.47%	3728	3743	3.24%
AS 13645	Trie	490587	5750	100.00%	273584	5344	100.00%	167409	11118	100.00%	116001	116455	100.00%
	DPA	73053	857	14.89%	30458	595	11.13%	12547	834	7.49%	3819	3834	3.29%
	DPCA	73045	856	14.89%	30456	595	11.13%	12547	834	7.49%	3819	3834	3.29%
AS 15290 (east)	Trie	507030	5942	100.00%	282456	5517	100.00%	171224	11371	100.00%	117043	117501	100.00%
	DPA	118309	1387	23.33%	55096	1077	19.51%	27574	1832	16.10%	8731	8766	7.46%
	DPCA	118291	1387	23.33%	55085	1076	19.50%	27572	1831	16.10%	8731	8766	7.46%
AS 15290 (west)	Trie	506960	5941	100.00%	282397	5516	100.00%	171185	11368	100.00%	116996	117454	100.00%
	DPA	118185	1385	23.31%	55079	1076	19.50%	27594	1833	16.12%	8698	8732	7.43%
	DPCA	118169	1385	23.31%	55069	1076	19.50%	27593	1833	16.12%	8698	8732	7.43%

Figure A.3: Results for the test-data (prefix automaton)

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AS 553	Trie	329906	3867	100.00%	181141	3538	100.00%	107566	7144	100.00%	70111	70385	100.00%
	DPA	56577	664	17.15%	28352	554	15.65%	13969	928	12.99%	5579	5601	7.96%
	DPLA1	50236	589	15.23%	25732	503	14.21%	12987	863	12.07%	5276	5297	7.53%
	DPLA2	48878	573	14.82%	25328	495	13.98%	12905	857	12.00%	5276	5297	7.53%
	DPLA3	48030	563	14.56%	25137	491	13.88%	12888	856	11.98%	5276	5297	7.53%
	DPLA4	47518	557	14.40%	25050	490	13.83%	12875	855	11.97%	5276	5297	7.53%
AS 852 (east)	Trie	489242	5734	100.00%	273023	5333	100.00%	167201	11104	100.00%	115951	116404	100.00%
	DPA	33424	392	6.83%	17234	337	6.31%	9228	613	5.52%	3710	3725	3.20%
	DPLA1	32387	380	6.62%	16773	328	6.14%	8960	595	5.36%	3701	3716	3.19%
	DPLA2	31001	364	6.34%	16526	323	6.05%	8884	590	5.31%	3701	3716	3.19%
	DPLA3	30599	359	6.25%	16293	319	5.97%	8863	589	5.30%	3701	3716	3.19%
	DPLA4	30431	357	6.22%	16234	318	5.95%	8857	589	5.30%	3701	3716	3.19%
AS 852 (west)	Trie	489347	5735	100.00%	273084	5334	100.00%	167241	11106	100.00%	115986	116440	100.00%
	DPA	33424	392	6.83%	17234	337	6.31%	9228	613	5.52%	3709	3724	3.20%
	DPLA1	32209	378	6.58%	16617	325	6.08%	8958	595	5.36%	3701	3716	3.19%
	DPLA2	31848	374	6.51%	16387	321	6.00%	8893	591	5.32%	3700	3715	3.19%
	DPLA3	30983	364	6.33%	16301	319	5.97%	8868	589	5.30%	3700	3715	3.19%
	DPLA4	30888	362	6.31%	16239	318	5.95%	8859	589	5.30%	3700	3715	3.19%
AS 3257	Trie	487969	5719	100.00%	272354	5320	100.00%	166856	11081	100.00%	115751	116204	100.00%
	DPA	33421	392	6.85%	17233	337	6.33%	9227	613	5.53%	3715	3730	3.21%
	DPLA1	32270	379	6.61%	16843	329	6.18%	8963	596	5.37%	3707	3722	3.20%
	DPLA2	31970	375	6.55%	16629	325	6.11%	8892	591	5.33%	3706	3721	3.20%
	DPLA3	31364	368	6.43%	16484	322	6.05%	8872	590	5.32%	3706	3721	3.20%
	DPLA4	31042	364	6.36%	16436	322	6.03%	8864	589	5.31%	3705	3720	3.20%
AS 3561	Trie	528569	6195	100.00%	295673	5775	100.00%	180250	11970	100.00%	124446	124933	100.00%
	DPA	35303	414	6.68%	18157	355	6.14%	9645	641	5.35%	3958	3974	3.18%
	DPLA1	33782	396	6.39%	17430	341	5.90%	9285	617	5.15%	3911	3927	3.14%
	DPLA2	33436	392	6.33%	17178	336	5.81%	9199	611	5.10%	3902	3918	3.14%
	DPLA3	33248	390	6.29%	17006	333	5.75%	9177	610	5.09%	3899	3915	3.13%
	DPLA4	33158	389	6.27%	16955	332	5.73%	9168	609	5.09%	3898	3914	3.13%
AS 3741	Trie	3582	42	100.00%	1932	38	100.00%	1108	74	100.00%	730	733	100.00%
	DPA	1277	15	35.65%	625	13	32.35%	327	22	29.51%	126	127	17.26%
	DPLA1	1149	14	32.08%	570	12	29.50%	306	21	27.62%	123	124	16.85%
	DPLA2	1125	14	31.41%	543	11	28.11%	300	20	27.08%	123	124	16.85%
	DPLA3	1124	14	31.38%	541	11	28.00%	298	20	26.90%	123	124	16.85%
	DPLA4	1083	13	30.23%	541	11	28.00%	298	20	26.90%	123	124	16.85%
AS 4323	Trie	494484	5795	100.00%	276351	5398	100.00%	169738	11272	100.00%	118293	118756	100.00%
	DPA	71657	840	14.49%	35580	695	12.87%	19448	1292	11.46%	6924	6952	5.85%
	DPLA1	68153	799	13.78%	34111	667	12.34%	19068	1267	11.23%	6903	6930	5.84%
	DPLA2	66689	782	13.49%	33526	655	12.13%	18988	1261	11.19%	6903	6930	5.84%
	DPLA3	65835	772	13.31%	33286	651	12.04%	18953	1259	11.17%	6903	6930	5.84%
	DPLA4	64997	762	13.14%	33181	649	12.01%	18943	1258	11.16%	6903	6930	5.84%
AS 5388	Trie	490581	5749	100.00%	273747	5347	100.00%	167695	11136	100.00%	116334	116789	100.00%
	DPA	33545	394	6.84%	17295	338	6.32%	9262	616	5.52%	3728	3743	3.20%
	DPLA1	32276	379	6.58%	16852	330	6.16%	8997	598	5.37%	3716	3731	3.19%
	DPLA2	31244	367	6.37%	16556	324	6.05%	8925	593	5.32%	3713	3728	3.19%
	DPLA3	30803	361	6.28%	16377	320	5.98%	8899	591	5.31%	3712	3727	3.19%
	DPLA4	30524	358	6.22%	16294	319	5.95%	8887	591	5.30%	3711	3726	3.19%
AS 5511	Trie	500152	5862	100.00%	279037	5450	100.00%	171243	11372	100.00%	119222	119688	100.00%
	DPA	64155	752	12.83%	32115	628	11.51%	17442	1159	10.19%	6400	6425	5.37%
	DPLA1	60663	711	12.13%	30580	598	10.96%	17016	1130	9.94%	6362	6387	5.34%
	DPLA2	59171	694	11.83%	30122	589	10.79%	16907	1123	9.87%	6355	6380	5.33%
	DPLA3	58476	686	11.69%	29862	584	10.70%	16866	1121	9.85%	6354	6379	5.33%
	DPLA4	57383	673	11.47%	29741	581	10.66%	16847	1119	9.84%	6354	6379	5.33%
AS 6539	Trie	493023	5778	100.00%	275199	5375	100.00%	168631	11199	100.00%	117208	117666	100.00%
	DPA	65247	765	13.23%	32608	637	11.85%	17807	1183	10.56%	6557	6583	5.59%
	DPLA1	62110	728	12.60%	31253	611	11.36%	17416	1157	10.33%	6527	6553	5.57%
	DPLA2	60463	709	12.26%	30780	602	11.18%	17318	1151	10.27%	6527	6553	5.57%
	DPLA3	59714	700	12.11%	30535	597	11.10%	17293	1149	10.25%	6527	6553	5.57%
	DPLA4	58809	690	11.93%	30390	594	11.04%	17280	1148	10.25%	6527	6553	5.57%

Figure A.4: Results for the test-data (expanded automaton, part 1)

Alphabet expansion		none			2			4			8		
Instance	Type	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size	States	Mem	Rel. size
AS 6648	Trie	6441	76	100.00%	3644	72	100.00%	2273	151	100.00%	1665	1672	100.00%
	DPA	1834	22	28.47%	907	18	24.89%	485	33	21.34%	222	223	13.33%
	DPLA1	1679	20	26.07%	835	17	22.91%	454	31	19.97%	220	221	13.21%
	DPLA2	1647	20	25.57%	815	16	22.37%	449	30	19.75%	219	220	13.15%
	DPLA3	1554	19	24.13%	806	16	22.12%	449	30	19.75%	219	220	13.15%
	DPLA4	1543	19	23.96%	804	16	22.06%	449	30	19.75%	219	220	13.15%
AS 6667	Trie	488463	5725	100.00%	272449	5322	100.00%	166821	11078	100.00%	115604	116056	100.00%
	DPA	33490	393	6.86%	17265	338	6.34%	9241	614	5.54%	3710	3725	3.21%
	DPLA1	32371	380	6.63%	16742	327	6.15%	8988	597	5.39%	3701	3716	3.20%
	DPLA2	31301	367	6.41%	16433	321	6.03%	8919	593	5.35%	3701	3716	3.20%
	DPLA3	30829	362	6.31%	16357	320	6.00%	8888	591	5.33%	3701	3716	3.20%
	DPLA4	30542	358	6.25%	16263	318	5.97%	8869	589	5.32%	3701	3716	3.20%
AS 6730	Trie	257201	3015	100.00%	143306	2799	100.00%	87114	5785	100.00%	60912	61150	100.00%
	DPA	51072	599	19.86%	25433	497	17.75%	13339	886	15.31%	3784	3799	6.21%
	DPLA1	47887	562	18.62%	24122	472	16.83%	13044	867	14.97%	3714	3729	6.10%
	DPLA2	46848	549	18.21%	23769	465	16.59%	13014	865	14.94%	3714	3729	6.10%
	DPLA3	46197	542	17.96%	23617	462	16.48%	13000	864	14.92%	3714	3729	6.10%
	DPLA4	45788	537	17.80%	23521	460	16.41%	12994	863	14.92%	3714	3729	6.10%
AS 6939	Trie	481647	5645	100.00%	268386	5242	100.00%	163865	10882	100.00%	113262	113705	100.00%
	DPA	67183	788	13.95%	33741	660	12.57%	17500	1163	10.68%	6527	6553	5.76%
	DPLA1	58949	691	12.24%	30083	588	11.21%	16034	1065	9.78%	6034	6058	5.33%
	DPLA2	57351	673	11.91%	29573	578	11.02%	15972	1061	9.75%	6034	6058	5.33%
	DPLA3	56346	661	11.70%	29387	574	10.95%	15943	1059	9.73%	6034	6058	5.33%
	DPLA4	55856	655	11.60%	29306	573	10.92%	15929	1058	9.72%	6034	6058	5.33%
AS 7474	Trie	491724	5763	100.00%	274313	5358	100.00%	167925	11152	100.00%	116150	116604	100.00%
	DPA	99927	1172	20.32%	49176	961	17.93%	26715	1775	15.91%	8471	8505	7.29%
	DPLA1	95331	1118	19.39%	47175	922	17.20%	26289	1746	15.66%	8442	8475	7.27%
	DPLA2	93275	1094	18.97%	46534	909	16.96%	26180	1739	15.59%	8442	8475	7.27%
	DPLA3	91997	1079	18.71%	46153	902	16.82%	26138	1736	15.57%	8442	8475	7.27%
	DPLA4	91193	1069	18.55%	45944	898	16.75%	26119	1735	15.55%	8442	8475	7.27%
AS 7911	Trie	417843	4897	100.00%	231491	4522	100.00%	139650	9274	100.00%	95150	95522	100.00%
	DPA	115599	1355	27.67%	57997	1133	25.05%	28487	1892	20.40%	8766	8801	9.21%
	DPLA1	101552	1191	24.30%	52260	1021	22.58%	26512	1761	18.98%	8004	8036	8.41%
	DPLA2	99029	1161	23.70%	51520	1007	22.26%	26420	1755	18.92%	8004	8036	8.41%
	DPLA3	97672	1145	23.38%	51165	1000	22.10%	26391	1753	18.90%	8004	8036	8.41%
	DPLA4	97018	1137	23.22%	51019	997	22.04%	26375	1752	18.89%	8004	8036	8.41%
AS 8220	Trie	515180	6038	100.00%	286935	5605	100.00%	173888	11548	100.00%	118396	118859	100.00%
	DPA	35427	416	6.88%	18253	357	6.36%	9658	642	5.55%	3979	3995	3.36%
	DPLA1	33973	399	6.59%	17641	345	6.15%	9335	620	5.37%	3946	3962	3.33%
	DPLA2	33394	392	6.48%	17377	340	6.06%	9253	615	5.32%	3938	3954	3.33%
	DPLA3	33200	390	6.44%	17073	334	5.95%	9220	613	5.30%	3938	3954	3.33%
	DPLA4	33010	387	6.41%	16992	332	5.92%	9207	612	5.29%	3938	3954	3.33%
AS 9132	Trie	494750	5798	100.00%	275804	5387	100.00%	168619	11198	100.00%	116674	117130	100.00%
	DPA	98764	1158	19.96%	49554	968	17.97%	24657	1638	14.62%	8237	8270	7.06%
	DPLA1	87847	1030	17.76%	45045	880	16.33%	23023	1529	13.65%	7638	7668	6.55%
	DPLA2	85602	1004	17.30%	44461	869	16.12%	22949	1524	13.61%	7638	7668	6.55%
	DPLA3	84466	990	17.07%	44202	864	16.03%	22928	1523	13.60%	7638	7668	6.55%
	DPLA4	83951	984	16.97%	44029	860	15.96%	22918	1522	13.59%	7638	7668	6.55%
AS 12312	Trie	486021	5696	100.00%	271118	5296	100.00%	165949	11021	100.00%	114929	115378	100.00%
	DPA	33386	392	6.87%	17221	337	6.35%	9224	613	5.56%	3710	3725	3.23%
	DPLA1	32329	379	6.65%	16764	328	6.18%	8952	595	5.39%	3702	3717	3.22%
	DPLA2	31760	373	6.53%	16449	322	6.07%	8891	591	5.36%	3701	3716	3.22%
	DPLA3	31039	364	6.39%	16304	319	6.01%	8869	589	5.34%	3701	3716	3.22%
	DPLA4	30844	362	6.35%	16223	317	5.98%	8858	589	5.34%	3701	3716	3.22%
AS 13645	Trie	490626	5750	100.00%	273602	5344	100.00%	167417	11118	100.00%	116004	116458	100.00%
	DPA	33755	396	6.88%	17406	340	6.36%	9314	619	5.56%	3731	3746	3.22%
	DPLA1	32242	378	6.57%	16929	331	6.19%	9032	600	5.39%	3718	3733	3.21%
	DPLA2	31857	374	6.49%	16771	328	6.13%	8965	596	5.35%	3714	3729	3.20%
	DPLA3	31429	369	6.41%	16618	325	6.07%	8939	594	5.34%	3713	3728	3.20%
	DPLA4	31045	364	6.33%	16409	321	6.00%	8931	594	5.33%	3713	3728	3.20%
AS 15290 (east)	Trie	507190	5944	100.00%	282530	5519	100.00%	171258	11373	100.00%	117057	117515	100.00%
	DPA	99156	1162	19.55%	48800	954	17.27%	26295	1747	15.35%	8588	8622	7.34%
	DPLA1	94186	1104	18.57%	46505	909	16.46%	25749	1710	15.04%	8538	8572	7.29%
	DPLA2	92266	1082	18.19%	45771	894	16.20%	25636	1703	14.97%	8533	8567	7.29%
	DPLA3	90559	1062	17.86%	45420	888	16.08%	25595	1700	14.95%	8531	8565	7.29%
	DPLA4	89822	1053	17.71%	45230	884	16.01%	25573	1699	14.93%	8531	8565	7.29%
AS 15290 (west)	Trie	507119	5943	100.00%	282471	5518	100.00%	171219	11371	100.00%	117010	117468	100.00%
	DPA	99111	1162	19.54%	48799	954	17.28%	26316	1748	15.37%	8570	8604	7.32%
	DPLA1	94362	1106	18.61%	46539	909	16.48%	25775	1712	15.05%	8524	8558	7.28%
	DPLA2	92343	1083	18.21%	45779	895	16.21%	25671	1705	14.99%	8517	8551	7.28%
	DPLA3	90418	1060	17.83%	45438	888	16.09%	25635	1703	14.97%	8514	8548	7.28%
	DPLA4	89504	1049	17.65%	45259	884	16.02%	25610	1701	14.96%	8512	8546	7.27%

Figure A.5: Results for the test-data (expanded automaton, part 2)

Appendix B

Implementation

All algorithms described in this thesis have been implemented in Java (with the exception of the brute force DP_kL_lA minimizer which was written in C++) to gain some experience with the algorithms and perform correctness tests on them. Additionally the code provided a platform for experimenting with different algorithmic ideas in the context of finite automata (*e.g.*, when developing heuristics for DP_kL_lA size reduction) as well as testing different setups for the creation and minimization of forwarding tables. Finally the programs were used to calculate the results from Section 7.2 and Appendix A.

With more than 6000 lines of code (including about 1000 lines of testing code) the sources can not be included here. Instead they are available upon request from the author (or can be found on the accompanying CD for the printed version). The intention of this section is to give an overview on the available source code, and to serve as a starting point for users and developers working with the code.

The code is organized as several loosely coupled Java packages within the `edu.tum.cs.flang` hierarchy (*flang* was chosen as a short form of *finite language* although the DP_kA code can be used with infinite regular languages as well).

core
: Definition of the `IFiniteAutomaton` interface, as well as basic implementations of this interface, including the `Trie` and the `TreelikeAutomaton` (a special automaton slightly more general than the trie which supports leaf-pushing and level shifting as operations).

dpa
: Implementations of the textbook, Hopcroft, and acyclic DP_kA minimization algorithms (Algorithms 3.1, 3.4, and 3.6). Furthermore code for alphabet expansion of DP_kAs as well as various tool code (such as testing whether two DP_kAs are isomorphic) are located here.

dpca

Implementations of the quadratic and Körner DP_kC_lA minimization algorithms (Algorithms 4.1 with 4.2, and 4.3), as well as a DP_kL_lA equivalence test (in `DPCATools`).

dpla

Besides the heuristic described in Section 5.5 (`DPLAOrderedFlippingMinimizer`) an older (less efficient) version of the heuristic assigning arbitrary state classes (`DPLARandomFlippingMinimizer`) and DP_kL_lA equivalence testing code (the non-polynomial time version) are included. Additionally the `ILPTransformer` can create an LP file containing an *integer linear program* for the minimization problem of a given DP_kL_lA. We hoped that in conjunction with commercial ILP solvers such as CPLEX[1], larger instances as with our simple brute force solver could be optimized. This did not work as expected, which might both be inherent to the DP_kL_lA minimization problem and our LP formulation.

io

This package has code for reading and writing DP_kAs using a proprietary file format. The `DotWriter` produces input files for the dot[2] program, which was also used to create the automata images in this paper. Being able to visualize medium sized automata easily this way proved to be invaluable for debugging.

rtable

Code for dealing with routing tables, such as conversions from different formats to a simple proprietary one, and building a trie from a routing table.

bin

Command-line tools for converting and minimizing automata, and creating statistics tables such as those given in Appendix A.

util

Utility code not directly related to the project, but used from some of the other classes.

For further details on the classes and implementation the reader is encouraged to look into the actual code or the generated JavaDoc documentation coming with it.

[1] CPLEX is a commercial mixed integer linear programming solver from ILOG (`http://www.ilog.com`)

[2] dot is part of the Open Source graph layout package GraphViz available at `http://www.graphviz.org/`

We want to loose a final word on testing. Testing complex algorithms is usually complicated as corner cases are overlooked and manually checking non-trivial test cases is often nearly impossible. An advantage of our implementation is the presence of at least two different algorithms for both $\mathrm{DP}_k\mathrm{A}$ and $\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$ minimization. So besides the usual simple test cases we included concurrent testing, where larger automata are randomly generated and minimized using *different* algorithms. From the theoretical results we know that the minimal automata produced by different algorithms for the same problem must have the same size and are isomorphic ($\mathrm{DP}_k\mathrm{A}$) or equivalent ($\mathrm{DP}_k\mathrm{C}_l\mathrm{A}$) which both can be checked programmatically. Thus the algorithms are used to verify each other increasing the "trust level" of the implementation.