Constructing factor oracles

Citation for published version (APA):

Document status and date:
Published: 01/01/2004

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Constructing Factor Oracles

Loek Cleophas¹, Gerard Zwaan¹, and Bruce W. Watson¹,²

¹ Department of Mathematics and Computer Science, Technische Universiteit
Eindhoven, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
² Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa
loek@loekcleophas.com, g.zwaan@tue.nl, bruce@bruce-watson.com

January 5, 2004

Abstract. A factor oracle is a data structure for weak factor recognition. It is an automaton built on a string $p$ of length $m$ that is acyclic, recognizes at least all factors of $p$, has $m+1$ states which are all final, and has $m$ to $2^m - 1$ transitions. In this paper, we give two alternative algorithms for its construction and prove the constructed automata to be equivalent to the automata constructed by the algorithms in [1]. Although these new $O(m^2)$ algorithms are practically inefficient compared to the $O(m)$ algorithm given in [1], they give more insight into factor oracles. Our first algorithm constructs a factor oracle based on the suffixes of $p$ in a way that is more intuitive. Some of the crucial properties of factor oracles, which in [1] need several lemmas to be proven, are immediately obvious. Another important property however becomes less obvious. A second algorithm gives a clear insight in the relationship between the trie or dawg recognizing the factors of $p$ and the factor oracle recognizing a superset thereof. We conjecture that an $O(m)$ version of this trie-based algorithm exists.

1 Introduction

A factor oracle is a data structure for weak factor recognition³. It can be described as an automaton built on a string $p$ of length $m$ that (a) is acyclic, (b) recognizes at least all factors of $p$, (c) has $m+1$ states (which are all final), and (d) has $m$ to $2m - 1$ transitions (cf. [1]). Some example factor oracles are given in Figure 1.

Factor oracles are introduced in [1] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. In such algorithms, a window on a text is read backward while attempting to match a keyword factor. When this fails, the window is shifted using the information on the longest factor matched and the mismatching character.

³ See Subsection 1.2 for the definition of a factor.
Instead of an automaton recognizing exactly the set of factors of the keyword, it is possible to use a factor oracle: although it recognizes more strings than just the factors and thus might read backwards longer than necessary, it cannot miss any matches. The advantage of using factor oracles is that they are easier to construct and take less space to represent compared to the automata that were previously used in these factor-based algorithms, such as suffix, factor and subsequence automata. The latter automata lack one of the properties (c) \((m+1)\) states or (d) \(m\) to \(2m-1\) transitions) and in fact have more states or transitions. The factor oracle on the other hand satisfies both and therefore takes less memory space.

The factor oracle is introduced in [1] by means of an \(O(m^2)\) construction algorithm that is used as its definition. Furthermore, an \(O(m)\) sequential construction algorithm is described. It is not obvious by just considering the algorithms that it recognizes at least all factors of \(p\) and has \(m\) to \(2m-1\) transitions (i.e. that (b) and (d) hold). For both algorithms, a number of lemmas are needed to prove this. In this paper, we give two alternative algorithms for the construction of a factor oracle.

Our first algorithm, in Section 2, constructs a factor oracle based on the suffixes of \(p\). This algorithm is \(O(m^2)\) and thus not of practical interest, but it is more intuitive to understand and properties (b) and (d)—two important properties of factor oracles—are immediately obvious from the algorithm. The acyclicity of the factor oracle however—corresponding to property (a)—is not immediately obvious. Our proof of this property (part of Property 6) is rather involved, whereas the property is immediately obvious from the algorithms in [1]. We prove that the alternative construction algorithm and those given in [1] construct equivalent automata in Section 3.

Section 4 shows that the language of a factor oracle is prefix-, suffix- and therefore factor-closed. A precise characterization of the language (independent of the automaton itself) is still open. We show that the occurrence of repetitions and the occurrence of states (other than the start state) with at least 2 outgoing transitions are necessary for the language to contain strings other than factors.
In Section 5 we present our second algorithm, which constructs a factor oracle from the trie recognizing the factors of $p$. Although this algorithm is $O(m^2)$ as well, it gives a clear insight in the relationship between the trie and dawg recognizing the factors of $p$ and the factor oracle recognizing a superset thereof. In addition, we conjecture that an $O(m)$ trie-based algorithm exists, although we have not investigated this due to lack of time.

Finally, Section 6 gives a summary and overview of future work.

1.1 Related Work

An earlier version of this report appears as [4, Chapter 4]. That thesis also discusses pattern matching algorithms—among them those using factor oracles—and the implementation of the factor oracle as part of the SPARE TIME pattern matching toolkit, a revised and extended version of SPARE PARTS ([11]).

A shorter version of this report—excluding Section 4 on the language of factor oracles—was presented at the 2003 Prague Stringology Conference ([3]).

As mentioned before, factor oracles were introduced in [1] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. A pattern matching algorithm using the factor oracle is described in that paper as well.

Apart from their use in pattern matching algorithms, factor oracles have been used in a heuristic to compute repeated factors of a string [8] as well as to compress text [9]. An improvement for those uses of factor oracles is introduced in [10] in the form of the repeat oracle.

Related to the factor oracle, the suffix oracle—in which only those states corresponding to a suffix of $p$ are marked final—is introduced in [1]. In [2] the factor oracle is extended to apply to a set of strings.

1.2 Preliminaries

A string $p = p_1 \ldots p_m$ of length $m$ is a sequence of characters from an alphabet $V$. A string $u$ is a factor (resp. prefix, suffix) of a string $v$ if $v = sut$ (resp. $v = ut$, $v = su$), for $s, t \in V^*$. We will use $\text{fact}(p)$, $\text{pref}(p)$ and $\text{suff}(p)$ for the set of factors, prefixes and suffixes of $p$ respectively. A factor (resp. prefix or suffix) is a proper factor (resp. prefix or suffix) of a string $p$ if it does not equal $p$. We write $u \leq_s v$ to denote that $u$ is a suffix of $v$, and $u <_s v$ to denote that $u$ is a proper suffix of $v$. 

3
2 Construction based on suffixes

Our first alternative algorithm for the construction of a factor oracle constructs a ‘skeleton’ automaton for \( p \)—recognizing \( \text{pref}(p) \)—and then constructs a path for each of the suffixes of \( p \) in order of decreasing length, such that eventually at least \( \text{pref}(\text{suff}(p)) = \text{fact}(p) \) is recognized. If such a suffix of \( p \) is already recognized, no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized there is a longest prefix of such a suffix that is recognized. A transition on the next, non-recognized symbol is then created, from the state in which this longest prefix of the suffix is recognized, to a state from which there is a path leading to state \( m \) that spells out the rest of the suffix.

**Build Oracle.2** (\( p = p_1p_2...p_m \))

1: for \( i \) from 0 to \( m \) do
2: Create a new final state \( i \)
3: for \( i \) from 0 to \( m - 1 \) do
4: Create a new transition from \( i \) to \( i + 1 \) by \( p_{i+1} \)
5: for \( i \) from 2 to \( m \) do
6: Let the longest path from state 0 that spells a prefix of \( p_i...p_m \) end in state \( j \) and spell out \( p_i...p_k \) \( (i - 1 \leq k \leq m) \)
7: if \( k \neq m \) then
8: Build a new transition from \( j \) to \( k + 1 \) by \( p_{k+1} \)

Note that this algorithm is \( O(m^2) \) (since the operation on line 6 can be implemented using a while loop). The factor oracle on \( p \) built using this algorithm is referred to as Oracle(\( p \)) and the language recognized by it as \( \text{factoracle}(p) \).

The first two properties we give are obvious given our algorithm. They correspond to (b) and (c)-(d) respectively as mentioned in Section 1.

**Property 1.** \( \text{fact}(p) \subseteq \text{factoracle}(p) \).

**Proof:** The algorithm constructs a path for all suffixes of \( p \) and all states are final.

**Property 2.** For \( p \) of length \( m \), Oracle(\( p \)) has exactly \( m + 1 \) states and between \( m \) and \( 2m - 1 \) transitions.

**Proof:** States can be constructed in steps 1-2 only, and exactly \( m + 1 \) states are constructed there. In step 4 of the algorithm, \( m \) transitions are created. In steps 5-8, at most \( m - 1 \) transitions are created.

**Property 3 (Glushkov’s property).** All transitions reaching a state \( i \) of Oracle(\( p \)) are labeled by \( p_i \).
Proof: The only steps of the algorithm that create transitions are steps 4 and 8. In both, transitions to a state \( i \) are created labeled by \( p_i \).

Property 4 (Weak determinism). For each state of Oracle(\( p \)), no two outgoing transitions of the state are labeled by the same symbol.

Proof: The algorithm never creates an outgoing transition by some symbol if such a transition already exists.

We now define function \( \text{poccur}(u, p) \) to give the end position of the leftmost occurrence of \( u \) in \( p \) (equivalent to the same function in [1]):

**Definition 1.** Function \( \text{poccur} \in V^* \times V^* \rightarrow \mathbb{N} \) is defined as

\[
\text{poccur}(u, p) = \min\{|tu|, p = tv\} \quad (p, t, u, v \in V^*)
\]

Note that if \( u \notin \text{fact}(p) \), \( \text{poccur}(u, p) = \infty \).

**Property 5.** For suffixes and prefixes of factors we have:

\[
\begin{align*}
uv & \in \text{fact}(p) \Rightarrow \text{poccur}(v, p) \leq \text{poccur}(uv, p) \quad (p, u, v \in V^*) \\
uv & \in \text{fact}(p) \Rightarrow \text{poccur}(u, p) \leq \text{poccur}(uv, p) - |v| \quad (p, u, v \in V^*)
\end{align*}
\]

We introduce \( \text{min}(i) \) for the minimum length string recognized in state \( i \)—either in a partially constructed or in the complete automaton.

In the following property, we use \( j_i \) and \( k_i \) to identify the values \( j \) and \( k \) attain when considering suffix \( p_i \ldots p_m \) of \( p \) in steps 5-8 of the algorithm.

**Property 6.** For the partial automaton constructed according to algorithm **Build-Oracle_2** with all suffixes of \( p \) of length greater than \( m - i + 1 \) already considered in steps 5-8 (\( 2 \leq i \leq m + 1 \)), we have that

i. it is acyclic

ii. for each \( h \) with \( 1 \leq h < i \), all prefixes of \( p_h \ldots p_m \) are recognized

iii. for each state \( n \) and outgoing transition to a state \( q \neq n + 1 \), \( q \leq k_{\text{max}} + 1 \) holds where \( k_{\text{max}} = \max\{k_h, 1 < h < i \land k_h < m\} \)

iv. for each state \( n \), \( \text{min}(n) \) is an element of \( \text{fact}(p) \), \( \text{min}(n) \) is a suffix of each string recognized in \( n \), and \( n = \text{poccur}(\text{min}(n), p) \)

v. if \( u \in \text{fact}(p) \) is recognized, it is recognized in a state \( n \leq \text{poccur}(u, p) \)

vi. for each state \( n \) and each symbol \( a \) such that there is a transition from \( n \) to a state \( q \) by \( a \), \( \text{min}(n) \cdot a \in \text{fact}(p) \) and \( q = \text{poccur}(\text{min}(n) \cdot a, p) \)
vii. for each pair of states \( n \) and \( q \), if \( \min(n) \leq \min(q) \), then \( n \leq q \), and as a result, if \( \min(n) < \min(q) \), then \( n < q \).

viii. if \( w \) is recognized in state \( n \), then for any suffix \( u \) of \( w \), if \( u \) is recognized, it is recognized in state \( q \leq n \).

**Proof:** See Appendix A. \( \square \)

Note that Property 6, i. corresponds to property (a) in Section 1.

### 3 Equivalence to Original Algorithms

A factor oracle as introduced in [1] is built by the following algorithm:

**Build Oracle**\( (p = p_1 p_2 \ldots p_m) \)

1. for \( i \) from 0 to \( m \) do
2. Create a new final state \( i \)
3. for \( i \) from 0 to \( m - 1 \) do
4. Create a new transition from \( i \) to \( i + 1 \) by \( p_{i+1} \)
5. for \( i \) from 0 to \( m - 1 \) do
6. Let \( u \) be a minimal length word in state \( i \)
7. for all \( \sigma \in \Sigma, \sigma \neq p_{i+1} \) do
8. if \( u\sigma \in \text{Fact}(p_{i-|u|+1} \ldots p_m) \) then
9. Build a new transition from \( i \) to \( \star \)

To prove the equivalence of the automata constructed by the two algorithms, we need the following properties.

**Property 7.** For any state \( i \) of both Oracle\((p) \) (i.e. the factor oracle constructed according to algorithm **Build**\_Oracle\_2] and the factor oracle constructed according to algorithm **Build**\_Oracle), if \( u = \min(i) \) then
\[
\begin{align*}
u\sigma & \in \text{fact}(p_{i-|u|+1} \ldots p_m) \equiv u\sigma \in \text{fact}(p)
\end{align*}
\]

**Proof:** \( \Rightarrow \): Trivial. \( \Leftarrow \): By Property 6, iv. (for **Build**\_Oracle\_2] and [1, Lemma 1] (for **Build**\_Oracle), \( i = pocc\_{occur}(u, p) \). By Property 5, \( pocc\_{occur}(u\sigma, p) \geq i \), hence \( u\sigma \in \text{fact}(p_{i-|u|+1} \ldots p_m) \). \( \square \)

**Property 8.** For any state \( i \) of an automaton constructed by either algorithm, if \( u = \min(i) \) and \( u\sigma \in \text{fact}(p) \) then
\[
\begin{align*}
i - |u| + pocc\_{occur}(u\sigma, p_{i-|u|+1} \ldots p_m) &= pocc\_{occur}(u\sigma, p)
\end{align*}
\]

* Note that in [1] the term \(-|u|\) is missing in the algorithm, although from the rest of the paper it is clear that it is used in the construction of the automata.
Proof:
\[
i - |u| + \text{poccur}(u\sigma, p_i - |u|+1...p_m)
\]
\[
= \{ \text{ definition } \text{poccur } \}
\]
\[
i - |u| + \min\{ |tu\sigma|, p_i - |u|+1...p_m = tu\sigma v \}
\]
\[
= \{ u = \min(i), \text{ hence recognized in } i = \text{poccur}(u, p) \}
\]
\[
i - |u| + \min\{ |tu\sigma| - (i - |u|), p = tu\sigma v \}
\]
\[
= \{ u\sigma \in \text{fact}(p), \text{ property of } \min \}
\]
\[
i - |u| + \min\{ |tu\sigma|, p = tu\sigma v \} - (i - |u|)
\]
\[
= \{ \text{ calculus, definition } \text{poccur } \}
\]
\[
\text{poccur}(u\sigma, p)
\]}

Property 9. The algorithms Build Oracle 2 and Build Oracle construct equivalent automata.

Proof: We prove this by induction on the states. Our induction hypothesis is that for each state \( j \) (0 \( j < i \)), \( \min(j) \) is the same in both automata, and the outgoing transitions from state \( j \) are equivalent for both automata.

If \( i = 0 \), \( u = \min(i) = \varepsilon \) in both automata. Consider a transition created by Build Oracle 2, say to state \( k \) by \( \sigma \neq p_{i+1} \). Since this transition exists, \( u\sigma \in \text{fact}(p) \) and \( k = \text{poccur}(u\sigma, p) \) (due to Property 6, vi.). Using Properties 7 and 8, such a transition was created by Build Oracle as well. Similarly, consider a transition created by Build Oracle, say to state \( k \) by \( \sigma \). This transition, say on symbol \( \sigma \), leads to state \( k = i - |u| + \text{poccur}(u\sigma, p_i - |u|+1...p_m) \) and was created since \( u\sigma \in \text{fact}(p_i - |u|+1...p_m) \) (see the algorithm). Using Properties 7 and 8, such a transition was created by Build Oracle 2 as well.

If \( i > 0 \), using the induction hypothesis and acyclicity of the automata, \( i \) has the same incoming transitions and as a result \( \min(i) \) is the same for both automata. Using the same arguments as in case \( i = 0 \), the outgoing transitions from state \( i \) are equivalent for both automata.

As a result, the two automata are equivalent.

\[ \square \]

4 Language of Factor Oracles

A definition of the language \text{factoracle}(p) not employing an automaton and its construction algorithm was left as an open question in [1]. It is
straightforward to see that it is bounded from above by $\text{seq}(p)$, the set of all subsequences of $p$: factor oracles are acyclic, all transitions go from a state $i$ to $j > i$, and all transitions going to some state $j$ are labeled by $p_j$.

In this section, we show that the language is prefix-, suffix- and hence factor-closed. We also give two properties showing sufficient conditions for $\text{factoracle}(p) \supset \text{fact}(p)$ to hold.

**Property 10.** The language $\text{factoracle}(p)$ is prefix-closed:

$$w \in \text{factoracle}(p) \Rightarrow \text{pref}(w) \subseteq \text{factoracle}(p) \quad (p, w \in V^*)$$

**Proof:** Follows directly from the fact that all states of Oracle($p$)—and thus all states on the path from state 0 spelling $w$—are final.

In [1, Lemma 5], the authors prove that the language recognized by a factor oracle is suffix-closed. Their proof is rather hard to follow and we therefore give a slightly different, longer version here.

**Property 11.** The language $\text{factoracle}(p)$ is suffix-closed:

$$w \in \text{factoracle}(p) \text{ is recognized in state } n \Rightarrow \text{suff}(w) \subseteq \text{factoracle}(p) \text{ and } u \in \text{suff}(w) \text{ is recognized in state } o \leq n \quad (p, u, w \in V^*)$$

**Proof:**

By induction on $|w|$. It is true if $|w| = 0$ or $|w| = 1$. Assume $|w| \geq 2$ and that it is true for all strings $x$ such that $|x| < |w|$. We show that it is also true for $w$, recognized in $n$.

Let $w = xa \ (x \neq \varepsilon)$, $x$ is recognized in $h \ (0 < h < n)$. Consider a proper suffix of $w$. It either equals $\varepsilon$ and is recognized in state 0 to $n$ or it can be written as $vu$ where $v < x$.

According to the induction hypothesis, $v$ is recognized in state $l \leq h$. Let $\bar{x} = \text{min}(h)$ and $\bar{v} = \text{min}(l)$. Due to Property 6, iv., $\bar{x} \leq_s x$ and $\bar{v} \leq_s v$. We now prove that $\bar{v} \leq_s \bar{x}$. If $l = h$, then $\bar{v} = \bar{x}$. Now consider the case $l < h$. Since $v \leq_s x$ and $\bar{v} \leq_s v$, $\bar{v} \leq_s x$. Due to Property 6, vii., $\bar{x} \nleq_s \bar{v}$. Thus, since $\bar{v}$ and $\bar{x}$ both are suffixes of $x$, $\bar{v} \leq_s \bar{x}$. Since $\bar{x}$ is recognized in $h$ and there is a transition by $a$ from $h$, by Property 6, vi. we have that $\bar{x}a \in \text{fact}(p)$ and $n = \text{poccur}(\bar{x}a, p)$. Since $\bar{x}a \in \text{fact}(p)$, $\bar{x}az \leq_s p$ for some $z \in V^*$, hence $\bar{v}az \leq_s p$ as well, hence $\bar{v}a$ is recognized. Since $\bar{v}$ is recognized in $l$, there is a transition by $a$ from $l$ to a state $o$. We know that $o = \text{poccur}(\bar{v}a, p)$ due to Property 6, vi. Since $\bar{v}a \leq_s \bar{x}a$, $\text{poccur}(\bar{v}a, p) \leq \text{poccur}(\bar{x}a, p)$ due to Property 5 and hence $o \leq n$. □
Property 12. The language factoracle is factor-closed:

\[ w \in \text{factoracle}(p) \Rightarrow \text{fact}(w) \in \text{factoracle}(p) \quad (p, w \in V^*) \]

**Proof:** Follows from Properties 10 and 11 and \( \text{fact}(w) = \text{pref}(\text{suff}(w)) \).

\[ \square \]

**Remark 1.** For \( \text{fact}(p) \), the equality \( \text{fact}(p)^r = \text{fact}(p^r) \) holds for all \( p \in V^* \). The equality \( \text{factoracle}(p)^r = \text{factoracle}(p^r) \) does not hold for all \( p \in V^* \). An example of a \( p \) for which the equality does not hold is \( p = \text{baabba} \). The factor oracles for \( p \) and \( p^r \) are given in Figure 2. It is clear that \( \text{bab} \in \text{factoracle}(\text{baabba})^r \) but \( \text{bab} \notin \text{factoracle}((\text{baabba})^r) \).

\[ \square \]

Due to Property 12, it must be possible to characterize a function \( \text{skip}(p) \) returning a set of strings such that \( \text{factoracle}(p) = \text{fact}(\text{skip}(p)) \). The exact definition of this function \( \text{skip} \) is still unclear, but the following two properties give some insight in the language \( \text{factoracle}(p) \):

**Property 13 (Relationship between non-factor strings and repetitions).** If there are no repetitions of any symbol \( a \) in \( p \), \( \text{factoracle}(p) \subseteq \text{fact}(p) \).

**Proof:** We prove this based on an induction hypothesis on the partial factor oracle constructed according to the algorithm, with the suffixes of \( p \) upto \( p_{i-1} \ldots p_m \) (i.e. of length \( \geq m-i+1 \)) already added to the automaton. Our induction hypothesis is that the transitions in this partial factor oracle are exactly those from \( j \) to \( j+1 \) on \( p_{j+1} \) for \( 0 \leq j < m \) and those from 0 to \( j \) on \( p_j \) for \( 1 \leq j \leq i \) and that the language recognized by this partial factor oracle is a subset of or equal to \( \text{fact}(p) \).

In case \( i = 1 \), the automaton clearly recognizes \( \text{pref}(p) \), and \( \text{pref}(p) \subseteq \text{fact}(p) \).
Assume that the induction hypothesis holds for $0 \leq j \leq i$. The algorithm will construct a transition from 0 to $i+1$ by $p_{i+1}$ in step 8 due to the absence of repetitions in $p$. New strings recognized will be $\text{pref}(p_{i+1}...p_m)$, and $\text{pref}(p_{i+1}...p_m) \subseteq \text{fact}(p)$. Thus, the language recognized is a subset of or equal to $\text{fact}(p)$ and the transitions are exactly those from $j$ to $j+1$ for $0 \leq j < m$ and those from 0 to $j$ for $1 \leq j \leq i+1$. \hfill \square

Property 14. If there is no state $i > 0$ in the factor oracle on $p$ with at least 2 outgoing transitions, $\text{factoracle}(p) = \text{fact}(p)$.

Proof: In this case every path from state 0 to state $m$ is labeled by a suffix of $p$. \hfill \square

5 Construction based on Trie

There is a close relationship between the data structures Trie($\text{fact}(p)$)—the trie ([7]) on $\text{fact}(p)$—recognizing exactly $\text{fact}(p)$, DAWG($\text{fact}(p)$)—the directed acyclic word graph ([6, 5]) on $\text{fact}(p)$—recognizing exactly $\text{fact}(p)$, and Oracle($p$)—the factor oracle on $p$—which recognizes at least $\text{fact}(p)$.

It is a well known fact that DAWG($\text{fact}(p)$) can be constructed from Trie($\text{fact}(p)$) by merging states whose right languages are identical (see for example [6, 5]). The factor oracle as defined by Oracle($p$) can also be constructed from Trie($\text{fact}(p)$), by merging states whose right languages have identical longest strings (which are suffixes of $p$). An example of a trie, DAWG and factor oracle for the factors of $abbc$ can be seen in Figure 3.

Definition 2. We define Trie($S$) as a 5-tuple $<Q, V, \delta, \varepsilon, F>$ where $S$ is a finite set of strings, $Q = \text{pref}(S)$ is the set of states, $V$ is the
alphabet, $\delta$ is the transition function, defined by

$$
\delta(u,a) = \begin{cases} 
ua & \text{if } ua \in \text{pref}(S) \\
\bot & \text{if } ua \notin \text{pref}(S)
\end{cases} \quad (u \in \text{pref}(S), a \in V),
$$

$\varepsilon$ is the single start state and $F = S$ is the set of final states. \qed

Property 15. For $u, v \in \text{fact}(p)$ we have :

$$
uv \in \text{fact}(p) \land (\forall w : uw \in \text{fact}(p) : |w| \leq |v|) \Rightarrow uv \in \text{suff}(p)
$$

$$
uv_1 \in \text{fact}(p) \land (\forall w : uw \in \text{fact}(p) : |w| \leq |v_1|) \\
\land uv_2 \in \text{fact}(p) \land (\forall w : uw \in \text{fact}(p) : |w| \leq |v_2|) \Rightarrow v_1 = v_2
$$

\qed

Property 16. For $u \in \text{fact}(p)$ and $C \in \mathbb{N}$,

$$(\forall w : uw \in \text{fact}(p) : |w| \leq C) \equiv (\forall w : uw \in \text{suff}(p) : |w| \leq C)$$

Proof: $\Rightarrow$: trivial. $\Leftarrow$: Let $ux \in \text{fact}(p)$, then $(\exists y : : uxy \in \text{suff}(p))$, hence $(\exists y : : |xy| \leq C)$, and since $|y| \geq 0$, $|x| \leq C$. \qed

Using Properties 15 and 16, $\max_p(u)$ can be defined as the unique longest string $v$ such that $uv \in \text{suff}(p)$:

**Definition 3.** Define $\max_p(u) = v$ where $v$ is such that

$$
uv \in \text{suff}(p) \land (\forall w : uw \in \text{suff}(p) : |w| \leq |v|)$$

\qed

We now present our trie-based construction algorithm for factor oracles:

**Trie_To_Oracle($p = p_1p_2...p_m$)**

1. Construct Trie($\text{fact}(p)$)
2. for $i$ from 2 to $m$ do
3. Merge all states $u$ for which $\max_p(u) = p_{i+1}...p_m$ into the single state $p_1...p_i$

The order in which the values of $i$ are considered is not important. In addition, note that it is not necessary to consider the states $u$ for which $\max_p(u) = p_2...p_m$ since there is precisely one such state $u$ in Trie($\text{fact}(p)$), $u = p_1$. Due to Property 15, it is sufficient to only consider suffixes of $p$ as longest strings.
Also note that the intermediate automata may be nondeterministic, but the final automaton will be weakly deterministic (as per Property 4).

The above algorithm has complexity $O(m^2)$ (assuming that $\max_p(u)$ was computed during construction of the trie). The construction of a Trie can be done in $O(m)$ time however, and the merging of the states is similar to minimization of an acyclic automaton, which can also be done in $O(m)$. We therefore conjecture that an $O(m)$ trie-based factor oracle construction algorithm exists, although we have not investigated this due to lack of time.

To prove that algorithm Trie-To-Oracle constructs Oracle($p$), we define a partition on the states of the trie, induced by an equivalence relation on the states.

**Definition 4.** Relation $\sim_p$ on states of Trie($\text{fact}(p)$) is defined by

$$ t \sim_p u \equiv \max_p(t) = \max_p(u) \quad (t, u \in \text{fact}(p)) $$

*Note that relation $\sim_p$ is an equivalence relation.*

We now show that the partitioning into sets of states of Trie($\text{fact}(p)$) induced by $\sim_p$, is the same as the partitioning of Trie($\text{fact}(pa)$) induced by $\sim_{pa}$, restricted to the states of Trie($\text{fact}(p)$), i.e.

**Property 17.**

$$ t \sim_p u \equiv t \sim_{pa} u \quad (t, u \in \text{fact}(p), a \in V) $$

*Proof:*

\[ t \sim_p u \]

\[ \equiv \quad \{ \text{definition } \sim_p \} \]

\[ \max_p(t) = \max_p(u) \]

\[ \equiv \quad \{ \} \]

\[ \max_p(t)a = \max_p(u)a \]

\[ \equiv \quad \{ (\star) \} \]

\[ \max_{pa}(t) = \max_{pa}(u) \]

\[ \equiv \quad \{ \text{definition } \sim_{pa} \} \]

\[ t \sim_{pa} u \]

where we prove (\star) by
\[
v = \max_{pa}(u)
\]
\[
\equiv \{ \text{definition } \max_{pa} \}
\]
\[
uv \in \text{suff}(pa) \land (\forall w : uw \in \text{suff}(pa) : |w| \leq |v|)
\]
\[
\equiv \{ u \in \text{fact}(p), \text{ hence } (\exists x : ux a \in \text{suff}(pa)), \text{ hence } |xa| > 0 \text{ and } |v| > 0; \text{suff}(pa) = \text{suff}(p)a \cup \{ \epsilon \} \}
\]
\[
uv \in \text{suff}(p)a \land (\forall w : uw \in \text{suff}(pa) : |w| \leq |v|)
\]
\[
\equiv \{ |v| > 0, \text{ introduction } v' \}
\]
\[
uv \in \text{suff}(p)a \land (\forall w : w \neq \epsilon \land uw \in \text{suff}(pa) : |w| \leq |v|) \land v = v'a
\]
\[
\equiv \{ \text{suff}(pa) = \text{suff}(p)a \cup \{ \epsilon \} \}
\]
\[
uv \in \text{suff}(p)a \land (\forall w : w \neq \epsilon \land uw \in \text{suff}(pa) : |w| \leq |v|) \land v = v'a
\]
\[
\equiv \{ w = w'a \}
\]
\[
uv \in \text{suff}(p)a \land (\forall w' : uw'a \in \text{suff}(pa) : |w'| \leq |v'|) \land v = v'a
\]
\[
\equiv \{ \}
\]
\[
uv \in \text{suff}(p)a \land (\forall w' : uw' \in \text{suff}(p) : |w'| \leq |v'|) \land v = v'a
\]
\[
\equiv \{ \text{definition } \max_p \}
\]
\[
v' = \max_p(u) \land v = v'a
\]
\[
\equiv \{ \text{elimination } v' \}
\]
\[
v = \max_p(u)a
\]

\[\square\]

Property 18. Algorithm Trie\_To\_Oracle constructs Oracle(p).

Proof: By induction on \(|p| = m\). If \(m = 0\), \(p = \epsilon\), and Trie(fact(\epsilon)) = Oracle(\epsilon). If \(m = 1\), \(p = a\) (\(a \in V\)), and Trie(fact(\epsilon))=Oracle(a). If \(m > 1\), \(p = xa\) (\(x \in V^*, a \in V\)), and we may assume the algorithm to construct part Oracle(x) of Oracle(xa) correctly (using fact(ua) = fact(u) \cup suffix(u)a, Trie(fact(xa)) being an extension of Trie(fact(x)), and Oracle(xa) being an extension of Oracle(x) (which is straightforward to see from algorithm Build\_Oracle\_2 as well as [1, page 57, after Corollary 4]), and Property 17). Now consider the states of this partially converted automaton in which suffixes of \(x\) are recognized. By construc-
tion of the trie, there are transitions from these states by $a$. The factor oracle construction according to algorithm Oracle_Sequential in [1] creates $\text{Oracle}(xa)$ from $\text{Oracle}(x) + a$ (i.e. the factor oracle for $x$ extended with a single new state $m$ reachable from state $m - 1$ by symbol $p_m = a$) by creating new transitions to state $m$ from those states in which suffixes of $x$ are recognized and that do not yet have a transition on $a$. Since Trie_To_Oracle merges all states $t$ for which $\max_{xa}(t) = a$ into the single state $m$, Oracle($xa$) is constructed correctly from Trie($\text{fact}(xa)$).

\section{Conclusions and Future Work}

We have presented two alternative construction algorithms for factor oracles and shown the automata constructed by them to be equivalent to those constructed by the algorithms in [1]. Although both our algorithms are $O(m^2)$ and thus practically inefficient compared to the $O(m)$ sequential algorithm given in [1], they give more insight into factor oracles.

Our first algorithm is more intuitive to understand and makes it immediately obvious, without the need for several lemmas, that the factor oracle recognizes at least $\text{fact}(p)$ and has $m$ to $2m - 1$ transitions.

Our second algorithm gives a clear insight into the relationship between the trie or dawg recognizing $\text{fact}(p)$ and the factor oracle recognizing a superset thereof. We conjecture that an $O(m)$ trie-based algorithm for the construction of factor oracles exists, although we have not investigated this due to lack of time.

Although an automaton-independent characterization of the language $\text{factoracle}(p)$ remains to be defined, we have given clear proofs that $\text{fact}(p) \subseteq \text{factoracle}(p)$. In addition, we have shown some sufficient conditions for $\text{fact}(p) \subset \text{factoracle}(p)$ to hold.

We are still working on an automaton-independent characterization of the language. Such a characterization would enable us to calculate how many strings are recognized that are not factors of the original string. This could be useful in determining whether to use a factor oracle-based algorithm in pattern matching or not.

As stated in [1], the factor oracle is not minimal in terms of number of transitions among the automata with $m + 1$ states recognizing at least $\text{fact}(p)$. We note that it is not even minimal among the subset of such automata having Glushkov’s property (see Figure 4).
Fig. 4. Factor oracle recognizing a superset of \text{fact}(p)$ (including for example $cace \notin \text{fact}(p)$) and alternative automaton with $m+1$ states satisfying Glushkov’s property yet recognizing a different superset of $\text{fact}(p)$ (including for example $acacdace \notin \text{factoracle}(p)$, but not $cace$) and having less transitions, for $p = abcacdace$.

References

A Proof of Property 6

We first consider the automaton constructed in steps 1-4 of the algorithm. It is straightforward to verify that the properties hold for \( i = 2 \).

Now assume that the properties hold for the automaton with all suffixes of \( p \) of length greater than \( m - i + 1 \) already considered. We prove that they also hold for the automaton after the suffix of length \( m - i + 1 \), \( p_i \ldots p_m \), has been considered.

If \( k = m \) in step 6, suffix \( p_i \ldots p_m \) is already recognized, no new transition will be created, the automaton does not change and the properties still hold.

If \( k < m \), then we need to prove that each of the properties holds for the new automaton.

Ad i: By v., string \( p_i \ldots p_k \) is recognized in state \( j \leq \text{poccur}(p_i \ldots p_k, p) \).

Since \( p_i \ldots p_k \leq_s p_i \ldots p_k \) and \( \text{poccur}(p_i \ldots p_k, p) = k \), \( \text{poccur}(p_i \ldots p_k, p) \leq k \) due to Property 5. Since \( j \leq k \), the transition created from \( j \) to \( k + 1 \) is a forward one.

Ad ii: Trivial.

Ad iii: We prove that the property holds for the new automaton by showing that \( k = k_i \geq k_{\text{max}} \), i.e. \( k \) will become the new \( k_{\text{max}} \).

If \( k_{\text{max}} = -\infty \), \( k \geq k_{\text{max}} \) clearly holds.

If \( k_{\text{max}} > -\infty \), assume that \( k_{\text{max}} > k \), then there is an \( h \) such that \( 1 < h < i \land k_h < m \land k_h = k_{\text{max}} \). Factor \( p_h \ldots p_k \) is recognized in \( g \leq k \) due to ii. and v.

If \( g = k \), then \( p_h \ldots p_k \) is recognized in \( k \) and \( p_h \ldots p_m \) is recognized in \( m \); so \( k_h = m \) which contradicts \( k_h < m \).

If \( g < k \), then \( p_h \ldots p_k \) is recognized in \( g < k \). Since \( p_i \ldots p_k \) is recognized in \( j = j_i \) and \( p_i \ldots p_k \leq_s p_i \ldots p_k \), due to viii., \( j \leq g \).

If \( j = g \), then \( p_h \ldots p_k \) is the longest prefix of \( p_h \ldots p_m \) recognized by the old automaton, which contradicts ii.

If \( j < g \), then \( j < g < k \). We know that \( \text{min}(g) \leq_s p_h \ldots p_k \) (using iv.), \( \text{min}(j) \leq_s p_h \ldots p_k \) (using iv. and \( p_i \ldots p_k \leq_s p_h \ldots p_k \) and therefore that \( \text{min}(j) < s \text{min}(g) \) (due to vii.). Let \( l \) be the state to which the transition by \( p_{k+1} \) from \( g \) leads, i.e. \( l \) is the state in which \( p_h \ldots p_{k+1} \) is recognized. Using vi., we have that \( l = \text{poccur}(\text{min}(g) \cdot p_{k+1}, p) \). Using Property 5 we have that \( l \leq \text{poccur}(p_h \ldots p_{k+1}, p) \) and the latter is \( \leq k + 1 \) due to the definition of \( \text{poccur} \) (since \( k + 1 \) marks the end of an occurrence of \( p_h \ldots p_{k+1} \)).

We have \( \text{poccur}(\text{min}(j) \cdot p_{k+1}, p) \leq \text{poccur}(\text{min}(g) \cdot p_{k+1}, p) = l \) since \( \text{min}(j) \leq_s \text{min}(g) \). We want to prove that \( k + 1 \leq \text{poccur}(\text{min}(j) \cdot p_{k+1}, p) \).

Assume that \( \text{poccur}(\text{min}(j) \cdot p_{k+1}, p) < k + 1 \). If the first occurrence of
min(j) · pk+1 starts before position i of p, then it is a prefix of a suffix of p longer than pl...pm and thus by ii. min(j) · pk+1 is recognized. Since min(j) is recognized in j, a transition from j by pk+1 must exist and we have a contradiction. If the first occurrence of min(j) · pk+1 starts at or after position i of p, then there exists a shortest string x such that x · min(j) · pk+1 ∈ pref(pl...pk) and x · min(j) · pk+1 is recognized in a state ≤ j. But then x · min(j) is recognized in a state n < j. By viii., since min(j) ≤ x · min(j), this means that min(j) is recognized in state s ≤ n < j and we have a contradiction. Thus k + 1 ≤ poccur(min(j) · pk+1, p) ≤ l and therefore, since l ≤ k + 1 holds, l = k + 1. In that case, ph...pm is recognized in l = k + 1 and ph...pm is recognized in m. But then kh = m, and we have a contradiction.

Thus, kmax = kh ≤ k = ki and iii. holds for the new automaton.

Ad iv: Let s = min(j), t = min(k+1) and u = min(h) (k+1 ≤ h ≤ m) respectively in the old automaton. Due to the proof of iii., k = ki ≥ kmax and therefore a unique path between k + 1 and h exists, labeled r, and—due to iv—u ≤ s tr.

If |spk+1r| ≥ |u|, u remains the minimal length string recognized in state h. Since s ≤ s pl...pk, spk+1r ≤ s pl...pk+1r. Since u ≤ s tr, tr ≤ s p1...pk+1r and |spk+1r| ≥ |u|, u ≤ spk+1r and—due to iv—u ≤ s spk+1r as well for any s′ recognized in state j.

If |spk+1r| < |u|, spk+1r is the new minimal length string recognized in state h. Since s ≤ s pl...pk, spk+1r ≤ s pl...pk+1r. Since u ≤ s tr, tr ≤ s p1...pk+1r and |spk+1r| < |u|, spk+1r ≤ s u and—due to iv—spk+1r ≤ s s′pk+1r as well for any s′ recognized in state j.

Since pl...pk+1r was not recognized before, it is not a prefix of p, p2...pm, ... pl−1...pm (using ii.), hence poccur(pl...pk+1r, p) = k + 1 + |r|. Since s ≤ s pl...pk, poccur(spk+1r, p) ≤ k + 1 + |r|. Assume that poccur(spk+1r, p) < k + 1 + |r|, then pl...pk+1r = uspk+1r (u, v ∈ V*, v ≠ ε, |u| minimal), since spk+1r cannot start before pl because in that case it would have already been recognized by the old automaton. Factor us is recognized in state g < j (using i.) and—since viii. holds—s ≤ s us is recognized in a state o ≤ g < j. This contradicts s being recognized in j. As a result poccur(spk+1r, p) = k + 1 + |r|.

Ad v: Any new factor of p recognized after creation of the transition from j to k + 1 has the form vpk+1r and is recognized in k + 1 + |r| with v ∈ fact(p) recognized in state j. Since k + 1 + |r| = poccur(min(k+1)r, p) (using iii., iv. holding for the new automaton plus the fact that k is the new kmax) and min(k+1) · r ≤ s vpk+1r due to iv. holding for the new automaton, k + 1 + |r| ≤ poccur(vp+1r, p) using Property 5.
Ad vi: The states \( n \) we have to consider are \( n = j \) and \( n = h \) for \( k + 1 \leq h \leq m \).

For \( n = j \), a new transition to \( k + 1 \) is created and by iv., \( \min(j) \leq s p_1 \ldots p_k \), hence we have \( \min(j) \cdot p_{k+1} \leq s p_1 \ldots p_{k+1}, p_{k+1} \cdot k+1 = \min(j) \cdot p_{k+1}, \min(j) \cdot p_{k+1} + 1 \in \text{fact}(p) \) and \( \text{poccur}(\min(j) \cdot p_{k+1}) \leq k + 1 \). Since \( \min(j) \cdot p_{k+1} \) is recognized in state \( k + 1 \), due to v. for the new automaton, \( k + 1 \leq \text{poccur}(\min(j) \cdot p_{k+1}). \) Therefore \( k + 1 = \text{poccur}(\min(j) \cdot p_{k+1}). \)

For \( n = h \) with \( k + 1 \leq h \leq m \), \( \min(h) \) changes to \( sp_{k+1}r \) if and only if \( |sp_{k+1}r| < |u| \) (with \( r, s, u \) as in the proof of iv.). We know that \( u \in \text{fact}(p) \) and \( q = \text{poccur}(ua, p) \). Since \( sp_{k+1}r \leq s u, sp_{k+1}ra \leq s u, \) hence \( sp_{k+1}ra \in \text{fact}(p) \) as well and \( \text{poccur}(sp_{k+1}ra, p) \leq \text{poccur}(ua, p) = q, \) but due to v., \( q \leq \text{poccur}(sp_{k+1}ra, p) \) hence \( q = \text{poccur}(sp_{k+1}ra, p) \).

Ad vii: Assume \( \min(n) \leq s \min(q) \). We have \( \text{poccur}(\min(n), p) \leq \text{poccur}(\min(q), p) \) due to Property 5, which according to iv. is equivalent to \( n \leq q \).

Ad viii: By induction on \(|w|\). It is true if \(|w| = 0 \) or \(|w| = 1 \). Assume that it is true for all strings \( x \) such that \(|x| < |w| \). We will show that it is also true for \( w \), recognized in \( n \).

Let \( w = xa \) \((x \neq \varepsilon)\), \( x \) is recognized in \( h \) \((0 < h < n) \). Consider a proper suffix of \( w \), recognized in state \( q \). It either equals \( \varepsilon \) and is recognized in state \( 0 \leq n \) or it can be written as \( va \) where \( v <_s x \).

Suffix \( va \) of \( w \) is recognized, therefore suffix \( v \) of \( x \) is recognized and according to the induction hypothesis, \( v \) is recognized in state \( l \leq h \). Let \( \bar{x} = \min(h) \) and \( \bar{v} = \min(l) \). Due to iv. for the new automaton, \( \bar{x} \leq_s x \) and \( \bar{v} \leq_s v \). We now prove that \( \bar{v} \leq_s \bar{x} \). If \( l = h \), then \( \bar{v} = \bar{x} \). Now consider the case \( l < h \). Since \( v \leq_s x \) and \( \bar{v} \leq_s v, \bar{v} \leq_s x \). Due to vii., \( x \leq_s \bar{x} \). Thus, since \( \bar{v} \) and \( \bar{x} \) both are suffixes of \( x \), \( \bar{v} \leq_s \bar{x} \). Since \( \bar{x} \) is recognized in \( h \) and there is a transition by \( a \) from \( h \) to \( n \), by vi. for the new automaton we have that \( \bar{x}a \in \text{fact}(p) \) and \( n = \text{poccur}(\bar{x}a, p) \). Since \( \bar{v} \) is recognized in \( l \) and there is a transition by \( a \) from \( l \) to \( q, \bar{v}a \in \text{fact}(p) \) and \( q = \text{poccur}(\bar{v}a, p) \) due to vi. for the new automaton. Since \( \bar{v}a \leq_s \bar{x}a \), \( \text{poccur}(\bar{v}a, p) \leq \text{poccur}(\bar{x}a, p) \) due to Property 5 and hence \( q \leq n \).

We have shown that the properties hold for every partial automaton during the construction. Consequently, they hold for the complete automaton Oracle(\(p) \).

\( \square \)