Three New Algorithms for Regular Language Enumeration

Margareta Ackerman[1] and Erkki Mäkinen[2]

[1] University of Waterloo, Waterloo ON, Canada
[2] University of Tampere, Tampere, Finland em@cs.uta.fi

Abstract. We present new and more efficient algorithms for regular language enumeration problems. The *min-word problem* is to find the lexicographically minimal word of length n accepted by a given NFA, the *cross-section problem* is to list all words of length n accepted by an NFA in lexicographical order, and the *enumeration problem* is to list the first m words accepted by an NFA according to length-lexicographic order. For the min-word and cross-section problems, we present algorithms with better asymptotic running times than previously known algorithms. Additionally, for each problem, we present algorithms with better practical running times than previously known algorithms.

1 Introduction

We would like to explore the language accepted by a given NFA by obtaining a sample of words from that language. The *min-word problem* is to find the lexicographically minimal word of length n accepted by a given NFA. The *cross-section problem* is to list all words of length n accepted by an NFA in lexicographical order. The *enumeration problem* is to list the first m words accepted by an NFA according to length-lexicographic order [3] (sorting a set of words according to length-lexicographic order is equivalent to sorting words of equal length according to lexicographic order and then sorting the set by length using a stable sort.)

We can use algorithms for the above problems to test the correctness of NFAs and regular expressions. (If a regular language is represented via a regular expression, we first convert it to an NFA.) While such a technique provides evidence that the correct NFA or regular expression has been found, an algorithm for the enumeration problem can also be used to fully verify correctness once sufficiently many words have been enumerated [1, p.11].

An algorithm for the cross-section problem can be used to decide whether every word accepted by a given NFA on s states is a power (a string of the form x^ℓ, for $|x| \geq 1$ and $\ell \geq 2$). It was shown by Anderson et. al. [2] that if every word accepted by the NFA is a power, then the NFA accepts no more than $7s$ words of each length, and further, if it accepts a non-power, it must accept a non-power of length less than $3s$. Using these results, Anderson et. al. [2] get an efficient algorithm for determining whether every word accepted by an NFA is a power by enumerating all the words of length $1, 2, \ldots, 3s-1$ and testing if each word is a power, stopping if the length of any cross-section exceeds $7s$.

The cross-section problem also leads to an alternative solution to the next k-subset of an n-set problem. The problem is, given a set $T = \{e_1, e_2, \ldots, e_n\}$, to enumerate all k-subsets of T in alphabetical order. See [3] for details of the solution.

For the min-word and cross-section problems, we present algorithms with better asymptotic and practical running times than previously known algorithms. In addition, for the enumeration problem, we present an algorithm with better practical running time than previously known algorithms, and the same asymptotic running time as the optimal previously known algorithm.

[3] Length-lexicographic order is also known as "radix order" and "pseudo-lexicographic order."

We analyze the algorithms in terms of their output size, the parameters of the NFA, and the length of words in the cross-section for the cross-section enumeration algorithms. The output size, t, is the total number of characters over all words enumerated by the algorithm. An NFA is a five-tuple $N = (Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the alphabet, δ is the transition function, q_0 is the start state, and F is the set of final states. In our analysis we consider $s = |Q|$, $\sigma = |\Sigma|$, and d, the number of transitions in the NFA. We assume that the graph induced by the NFA is connected (otherwise, we can preprocess the NFA in $O(s + d)$ operations.) Our algorithms are more efficient than previous algorithms in terms of the number of states and transitions of the NFA, without compromising efficiency on the other parameters.

First we present some previous work on regular language enumeration, as well as set up the framework which we will use for most of our new algorithms. Next, we present our first set of algorithms, which we refer to as the AMSorted enumeration algorithms, with better asymptotic running times than previous algorithms for the min-word and cross-section problems. Next, we present the AMBoolean enumeration algorithms, with even better asymptotic running times for the min-word and cross-section problems. Lastly, we present a very simple set of algorithms, which we call IntersectionEnumeration, which gives a min-word and a cross-section algorithm with the same asymptotic running time as the AMBoolean algorithms (however, the IntersectionEnumeration algorithms are inefficient in practice). We perform rigorous testing of our algorithms and previous enumeration algorithms. We find that the AMSorted algorithm for min-word and the AMBoolean algorithm for the cross-section and enumeration problems have the best practical running times.

2 Previous Work

The lookahead-matrix cross-section enumeration algorithm presented by Ackerman and Shallit [3] is the most efficient previously known algorithm in terms of n, the length of the words in the cross-section. As shown in [3], the cross-section lookahead-matrix algorithm, crossSectionLM is $O(s^{2.376}n + \sigma s^2 t)$ [3]. The min-word lookahead-matrix algorithm, minWordLM, finds the minimal word of length n in $O(s^{2.376}n)$ time and $O(s^2 n)$ space [3]. The algorithm enumLM, the lookahead-matrix algorithm for the enumeration problem, uses $O(s^{2.376}c + \sigma s^2 t)$ operations, where c is the number of cross-sections encountered throughout the enumeration.

We now analyze these algorithms with respect to d, the number of edges in the NFA. The lookahead-matrix algorithms use the framework in Section 3. As described in Section 3, after the minimal word in the cross-section has been found, to enumerate a cross-section we need to examine the transitions from all the states currently on the state stack (of which there are at most d). Thus, enumCrossSection is $O(s^{2.376}n + dt)$. Similarly, enumLM is $O(s^{2.376}c + dt)$.

Another previous set of algorithms is Mäkinen's algorithms, originally presented in [4] and analyzed in the unit-cost model, where it is linear in n. In the bit-complexity model, Mäkinen's cross-section enumeration algorithm is quadratic in n. In [3], Ackerman and Shallit discuss the theoretical and practical performance of two versions of Mäkinen's algorithms, MäkinenI (MI) and MäkinenII (MII). The main difference between these two algorithms is the way how they determine when a given cross-section has been fully enumerated. The MI cross-section algorithm precomputes the maximal word in a given cross-section, and terminates when the maximal word is found in the enumeration. The MII cross-section algorithm terminates when the state stack is empty (the same termination method is used in cross-section in Section 3). While the two versions of Mäkinen's algorithm have the same asymptotic running time, the MII cross-section algorithm has better

practical performance. The asymptotic running time of Mäkinen's min-word algorithm is $\Theta(s^2n^2)$, as shown in [3]. Mäkinen's cross-section algorithms are $O(s^2n^2+\sigma s^2t)$, and Mäkinen's enumeration algorithms are $O(\sigma s^2t+s^2e)$, where e is the number of empty cross-sections encountered throughout the enumeration [3]. If we analyze these algorithms with respect to d, the number of edges in the NFA, we find that Mäkinen's min-word algorithms are $\Theta(dn^2)$, the cross-section algorithms are $O(d(n^2+t))$ and the enumeration algorithms are $O(d(e+t))$. When referring to Mäkinen's algorithm, we use the MII versions, calling them minWordM, crossSectionM, and enumM.

The Grail computation package [5] includes a cross-section enumeration algorithm, fmenum, that uses a breadth-first-search approach on the tree of all possible paths from the start state of the NFA [5]. The algorithm is exponential in n [3].

It was found that crossSectionM and enumM usually have the best running time in practice. In special cases, where the quadratic running time of Mäkinen's cross-section algorithm is reached, lookahead-matrix performs better [3].

For all algorithms, we assume that the characters on the transitions from state A to state B are sorted, for all states A and B. Then the running time is independent of alphabet size. Otherwise, we can sort the characters on transitions between all pairs of states in $O(\sigma \log \sigma s^2)$ operations.

3 Algorithm Framework

We present the general framework commonly used for cross-section and enumeration algorithms. This framework was used for the lookahead-matrix algorithms and Mäkinen algorithms [3]. We use this framework for most of our new algorithms.

For cross-section enumeration, we first find the minimal word $w = a_1a_2\cdots a_n$ in the cross-section with respect to length-lexicographic order, or determine that the cross-section is empty.

Definition 1 (i-complete). *We say that state q of NFA N is* i-complete *if starting from q there is a path in N of length exactly i ending at a final state of N.*

Let $S_0 = \{q_0\}$ and $S_i = \cup_{q\in S_{i-1}}\delta(q, a_i) \cap \{q \mid q \text{ is } (n-i)\text{-complete}\}$, for $1 \le i < n$. That is, S_i is the set of $(n-i)$-complete states reachable from the states in S_{i-1} on a_i. We find w while storing the sets of states $S_0, S_1, S_2, \ldots, S_{n-1}$ on the *state stack*, S, which we assume is global.

We assume that there is some implementation of the method minWord(n, N), which returns the minimal word w of length n accepted by NFA N starting from one of the states on top of the state stack, or returns NULL if no such word exists. Different algorithms use different approaches for finding minimal words. To find the next word, we scan the minimal word $a_1a_2\cdots a_n$ from right to left, looking for the shortest suffix that can be replaced such that the new word is in $L(N)$. It follows that the suffix $a_i\cdots a_n$ can be replaced if the set of $(n-i)$-complete states reachable from S_{i-1} on any symbol greater than a_i is not empty. As we search for the next word of length n, we update the state stack. Therefore, each consecutive word can be found using the described procedure. The algorithm is outlined in detail in nextWord. Note that the algorithms use indentation to denote the scope of loops and if-statements.

Algorithm 1 nextWord(w,N)
INPUT: A word $w = a_1a_2\cdots a_n$ and an NFA N.
OUTPUT: Returns the next word in the n^{th} cross-section of $L(N)$ according to length-lexicographic order if it exists. Otherwise, returns NULL. Updates S for a potential subsequent call to nextWord or minWord.

```
FOR i ← n, ..., 1
    S_{i-1} = top(S)
    R = {v ∈ ∪_{q∈S_{i-1}, a∈Σ} δ(q,a) | v is (n−i)-complete}
    A = {a ∈ Σ | ∪_{q∈S_{i-1}} δ(q,a) ∩ R ≠ ∅}
    IF for all a ∈ A, a ≤ a_i
       pop(S)
    ELSE
       b_i = min{a ∈ A | a > a_i}
       S_i = {v ∈ ∪_{q∈S_{i-1}} δ(q,b_i) | v is (n−i)-complete}
       IF i ≠ n
          push(S,S_i)
       w' = w[1 ··· i−1] · b_i · minWord(n − i, N)
       RETURN w'
RETURN NULL
```

Algorithm 2 crossSection(n,N)
INPUT: A nonnegative integer n and an NFA N.
OUTPUT: Enumerates the n^{th} cross-section of $L(N)$.

```
S = empty stack
push(S, {q_0})
w = minWord(n,N)
WHILE w ≠ NULL
   visit w
   w = nextWord(w,N)
```

The algorithms nextWord and crossSection can be used in conjunction with any algorithms for `minWord` and for determining if a state is i-complete.

To get an enumeration algorithm, find the minimal word in each cross-section and call nextWord to get the rest of the words in the cross-section, until the required number of words is found.

Algorithm 3 enum(m,N)
INPUT: A nonnegative integer m and an NFA N.
OUTPUT: Enumerates the first m words accepted by N according to length-lexicographic order, if there are at least m words. Otherwise, enumerates all words accepted by N.

```
i = 0
numCEC = 0
len = 0
WHILE i < m AND numCEC < s DO
    S = empty stack
    push(S, {q_0})
    w = minWord(len,N)
    IF w = NULL
       numCEC = numCEC+1
    ELSE
        numCEC = 0
        WHILE w ≠ NULL AND i < m
            visit w
            w = nextWord(w,N)
            i = i + 1
    len = len+1
```

The variable *numCEC* counts the number of consecutive empty cross-sections. If the count ever hits s, the number of states in N, then all the words accepted by N have been visited [3].

4 New Enumeration Algorithms

We present three sets of algorithms. Our algorithms improve on the asymptotic time over previous algorithms for the min-word and cross-section problems. In addition, for all of the enumeration problems discussed (namely min-word, cross-section, and enumeration), at least one of our algorithms has better practical running time than previous algorithms.

4.1 Ackerman-Mäkinen Sorted Algorithms

We present a modification of minWordM that runs in time linear in n, the length of the words in the cross-section. The original algorithm appears in [4]. Mäkinen's original algorithm is linear in n in the unit-cost model, and as shown in [3], the algorithm is quadratic in n in the bit complexity model. Here we present a modification of minWordM which is linear in n in the bit-complexity model. We also present a cross-section enumeration algorithm using this modification of minWordM that is linear in n. This algorithm is more efficient than the Lookahead-Matrix cross-section algorithm in terms of the number of states in the NFA, without compromising efficiency in any of the other parameters.

We now describe the setup for the modification of Mäkinen's algorithm. The algorithm builds a table for each state, representing the minimal words of lengths 1 through n that can be accepted from that state. In particular, $A^{min}[1]$ stores the minimal character that occurs on a transition from A to a final state; if no such transitions exists, then $A^{min}[1]$ is NULL. Index i in table A^{min} stores a pair (a, B), with symbol a and state B, such that the minimal word of length i that occurs on a path from A to a final state is a concatenated with the minimal word of length $i-1$ appearing on a path from state B to a final state; if no such path exists, then $A^{min}[i]$ is NULL.

We define an order on the set $\{A^{min}[i] \mid A \in Q\}$ as follows. If $A^{min}[i] = \text{NULL}$ and $B^{min}[i] \neq \text{NULL}$, then $A^{min}[i] < B^{min}[i]$. In addition, if $A^{min}[1] = a$ and $B^{min}[1] = b$ where $a < b$, then $A^{min}[1] < B^{min}[1]$. For $i \geq 2$, if $A^{min}[i] = (a, A')$ and $B^{min}[i] = (b, B')$ where $a < b$, or $a = b$ and $A'^{min}[i-1] < B'^{min}[i-1]$, then $A^{min}[i] < B^{min}[i]$.

Algorithm 4 minWordAMSorted(n, N)
INPUT: A positive integer n and an NFA N.
OUTPUT: Table $A^{min}[1, \ldots, n]$ for each state $A \in Q$ where $A^{min}[i] = (a, B)$ and the minimal word of length i that occurs on a path from A to a final state is a concatenated with the minimal word of length $i-1$ appearing on a path from state B to a final state.

```
FOR each A ∈ Q
    IF for all a ∈ Σ, δ(A, a) ∩ F = ∅
        A^min[1] = NULL
    ELSE
        A^min[1] = min{a ∈ Σ | δ(A, a) ∩ F ≠ ∅}
FOR i ← 2, ..., n
    Sort the set {A^min[i − 1] | A ∈ Q}
    FOR each A ∈ Q
        a = min{a ∈ Σ | B^min[i − 1] ≠ NULL, B ∈ δ(A, a)}
        min = NULL
        FOR each B ∈ Q such that B ∈ δ(A, a)
            IF B^min[i − 1] ≠ NULL
                IF B^min[i − 1] < min OR min = NULL
                    min ← B^min[i − 1]
        A^min[i] = (a, min)
RETURN {A^min | A ∈ Q}
```

Notice that sorting $\{A^{min}[i-1] \mid A \in Q\}$ takes $s \log s$ operations, since the sorting consists of sorting the pairs by their values of $\{A^{min}[i-2] \mid A \in Q\}$, for which the order has already been computed, followed by sorting the symbols in the pairs.

Since we assume that the characters on the transitions between states are sorted, minWordAMSorted uses $O(n(s \log s + d))$ operations. Observe also that minWordAMSorted uses $O(sn)$ space.

Theorem 1. *The algorithm* minWordAMSorted *uses $O((s \log s + d)n)$ operations and $O(sn)$ space to find the minimal word of length n accepted by an NFA, where s is the number of states and d is the number of transitions in the NFA.*

There are two main differences between this version of Mäkinen's algorithm and the original. The first is the mode of storage, as the original algorithm stored the minimal word of length i that occurs on a path from A to a final state in cell $A^{min}[i]$. The second modification is the sorting of the set $\{A^{min}[i] \mid A \in S\}$. These changes eliminate the need for expensive comparisons.

We can use minWordAMSorted as part of a cross-section enumeration algorithm, by replacing minWord with minWordAMSorted, giving algorithm crossSectionAMSorted. To determine if a state A is i-complete, we simply check whether $A^{min}[i] \neq$ NULL. The algorithm crossSectionAMSorted finds the minimal word in the cross-section in $O((s \log s + d)n)$ operations, and finds the remaining words in $O(dt)$ where t is the output size. Therefore, crossSectionAMSorted uses $O(ns \log s + dt)$ operations.

Theorem 2. *The algorithm* crossSectionAMSorted *uses $O(ns \log s + dt)$ operations to enumerate the n^{th} cross-section of a given NFA, where s is the number of states and d is the number of transitions in the NFA and t is the number of characters in the output.*

We can also use minWordAMSorted as part of an enumeration algorithm, by replacing minWord with minWordAMSorted giving algorithm enumAMSorted. After enumerating the n^{th} cross-section we have a table $A^{min}[1, \ldots, n]$ for each state A. To improve the performance of enumAMSorted, when minWordAMSorted is called for $n+1$, we reuse these tables, extending the tables by index $n+1$. Therefore, each call to minWordAMSorted(i, S) costs $O(s \log s + d)$. Finding the rest of the words in the cross-section costs $O(dt)$.

For each empty cross-section, the algorithm does $O(s \log s + d)$ operations. Therefore, enumAMSorted uses $O(c(s \log s + d) + dt))$ operations, where c is the number of cross-sections encountered through the enumeration. The running time is independent of alphabet size since the characters on transitions between every pair of states are sorted, so for every state on the state stack we can keep a pointer to the last characters used, and progress it when the state is revisited. When a state is removed from the state stack then the pointer is reset.

Theorem 3. *The algorithm* enumAMSorted *uses $O(c(s \log s + d) + dt))$ operations, where c is the number of cross-sections encountered throughout the enumeration.*

4.2 Ackerman-Mäkinen Boolean Algorithms

We introduce another modification of Mäkinen's algorithms which yields algorithms that are efficient both theoretically and practically. These algorithms are more efficient than the AMSorted algorithms in terms of the number of states in the NFA, without compromising efficiency in any of

the other parameters. The min-word and cross-section algorithms in this section have better asymptotic running times than previous algorithms for these problems. In addition, the cross-section and enumeration algorithms presented in this section have the best practical running times.

The main function of the lookup tables constructed by Mäkinen's algorithm is to enable quick evaluation of i-completeness. Instead of storing characters, we propose storing boolean values in the lookup table. That is, $A^{min}[i]$ is true if and only if state A is i-complete.

Algorithm 5 preprocessingAMBoolean(n, N)
INPUT: A positive integer n and an NFA N.
OUTPUT: Table $A^{min}[1, \ldots, n]$ for each state $A \in Q$ where $A^{min}[i]$ is true if and only if A is i-complete.

```
FOR each A ∈ Q
    IF for all a ∈ Σ, δ(A, a) ∩ F = ∅
        A^min[1] = FALSE
    ELSE
        A^min[1] = TRUE
FOR i ← 2, ..., n
    FOR each A ∈ Q
        A^min[i] = FALSE
        FOR each B ∈ δ(A, a) for any a ∈ Σ
            IF B^min[i − 1] = TRUE
                A^min[i] = TRUE
RETURN {A^min | A ∈ Q}
```

The preprocessing takes $O((s + d)n)$ operations. Now, to find the minimal word of a given length n as well as the rest of the words in the cross-section, we use the state stack (as described in Section 3), which takes $O(dt)$. This gives a $O(d(n + t))$ algorithm for cross-section enumeration (recall that we assume that the graph induced by the NFA is connected, and thus $s \leq d$.) We call this algorithm crossSectionAMBoolean.

Theorem 4. *The algorithm* crossSectionAMBoolean *uses $O(d(n + t))$ operations to enumerate the n^{th} cross-section, where d is the number of transitions in the NFA and t is the number of characters in the output.*

Similarly, to use this approach for enumerating the first m words accepted by an NFA, we extend the tables by one for each consecutive cross-section in $O(s + d)$ operations, and use the state stack to enumerate each cross-section. This gives a $O(d(c + t))$ time algorithm, where c is the number of cross-sections encountered throughout the enumeration. Or, since each non-empty cross-section adds at least one to t, we can express the asymptotic running time in terms of e, the number of empty cross-sections encountered by the algorithm, giving $O(d(e + t))$.

Theorem 5. *The algorithm* enumAMBoolean *uses $O(d(e+t))$ operations, where c is the number of empty cross-sections found throughout the enumeration, d is the number of transitions in the NFA, and t is the number of characters in the output.*

Note also, that we can terminate crossSectionAMBoolean after the first word is found, giving a $O(dn)$ running time algorithm for min-word - we call this algorithm minWordAMBoolean.

4.3 Intersection Cross-Section Enumeration Algorithm

We introduce a new cross-section enumeration algorithm, that is both very simple and has good asymptotic running time. The main idea of the algorithm is to create an NFA, such that when crossSectionEnum traverses the NFA looking for i-complete states, all reachable states are i-complete.

Let All_n be the minimal DFA that accepts all words over Σ^n.

Algorithm 6 minWordIntersection(n, N)
INPUT: A positive integer n and an NFA N.
OUTPUT: The minimal word of length n accepted by N.

1. Find NFA $C = N \times All_n$.
2. Perform a breadth-first-search starting from the final states of C using reverse transitions, and remove all unvisited states.
3. Find the minimal word of length n accepted by C by traversing C from the start state following minimal transitions.

The NFA C, the cross-product of N with the NFA that accepts all words of length n, can be constructed by concatenating n copies of N. Notice that C has $O(dn)$ transitions. Step 2 and 3 run in time proportional to the size of C. Therefore, minWordIntersection uses $O(dn)$ operations.

Theorem 6. *The algorithm* minWordIntersection *uses $O(dn)$ to find the minimal word of length n accepted by a given NFA, where d is the number of transitions in the NFA.*

To find all the words in the cross-section, perform breadth-first search, recording all the words of length n occurring on paths of length n starting from the start state. That is, if we store the sets of states visited by minWordIntersection on a state stack, and use minWordIntersection with enumCrossSection, we get algorithm crossSectionIntersection.

We can use minWordIntersection to make a cross-section enumeration algorithm, crossSectionIntersection. That is, we can use the enumCrossSection algorithm on C instead of N, with minWordIntersection. All paths starting from the start state in C lead to a final state in n states; therefore, testing for i-completeness is not needed.

The algorithm crossSectionIntersection uses $O(dn)$ operations to find the minimal word of length n, and $O(dt)$ to find every other character in the cross-section. Therefore, crossSectionIntersection uses $O(d(n+t))$ operations.

Theorem 7. *The algorithm* crossSectionIntersection *uses $O(d(n+t))$ operations to enumerate the n^{th} cross-section in a given NFA, where d is the number of transitions in the NFA, and t is the number of characters in the output.*

5 Experimental Results

We compare the practical performance of the new algorithms with the best previously known algorithms. From [3], we know that among the Grail enumeration algorithms, the lookahead-matrix algorithms, and two versions of Mäkinen's algorithm, the Mäkinen algorithms tend to have the best practical performance. In some cases, Lookahead-Matrix performs better than Mäkinen's algorithm [3]. Therefore, we compare the new algorithms presented here with Mäkinen's and the lookahead-matrix algorithms.

A large body of tests was randomly generated. Most tests follow the following format: 100 NFAs were randomly generated with a bound on the number of vertices and alphabet size. The probability of placing an edge between any two states was randomly generated. The probability of any state being final or the number of final states was randomly generated within a specified range. The algorithms were tested on NFAs with differing number of states, varying edge densities, various alphabet sizes, and different proportions of final states.

In addition, we tested the algorithms on NFAs on which crossSectionLM performs better than crossSectionM. An example of this type of NFAs is presented in Figure 1. To see why crossSectionM has quadratic running time on these type NFA, see [3]. All but the tests on the L_m NFAs (see Figure 1) are on sets of randomly generated NFAs with up to 10 nodes.

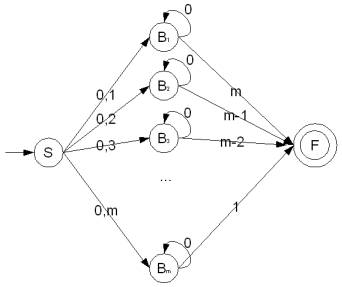

Fig. 1. NFA L_m.

We perform three groups of tests. The first group is aimed at determining the practical efficiency of the new min-word algorithms. We compared minWordM, minWordAMSorted, minWordAMBoolean, and minWordLM. In most of these tests, minWordAMSorted algorithm performs best. The only tests in which minWordAMSorted did not have the best performance was in some of the tests with the L_m NFAs (see Figure 1), but even on these NFAs this algorithm's performance was close to the performance of the fastest algorithm for these tests.

The second set of tests evaluates the new cross-section enumeration algorithms. We compare crossSectionM, crossSectionAMSorted, crossSectionAMBoolean, crossSectionLM, and crossSectionIntersection. We found that, in general, crossSectionAMBoolean and crossSectionM perform better than the other algorithms, with their performance very close to each other. On NFAs on which crossSectionM is quadratic in the size of the cross-section (the L_m automata), crossSectionAMBoolean outperforms the other algorithms. Thus, crossSectionAMBoolean has the overall best asymptotic and practical running time. In addition, we found that crossSectionIntersection is significantly slower than the other algorithms.

In the third set of tests we compared the practical performance of enumM, enumAMSorted, enumAMBoolean, and enumLM. The algorithm enumAMBoolean has the best running time. In a few cases, enumAMBoolean was outperformed by a very small amount by enumM or enumAMSorted. Due to the nature of the algorithms, it appears that the cases when enumAMBoolean is outperformed can be attributed to language or implementation specific details. Thus, the AMBoolean algorithms have the best practical performance for both the cross-section and the enumeration problem.

The tests were written in C# 3.0 and run on Microsoft Windows Vista Business, Intel(R) Core(TM)2 Duo CPU 1.80GHz, 1.96 GB of RAM. We summarize our experiments in Tables 2-4 in the appendix.

6 Summary and Future Work

We presented three sets of algorithms: The AM-Sorted min-word, cross-section and enumeration algorithms, the AM-Boolean min-word, cross-section and enumeration algorithms, and the intersection-based min-word and cross-section algorithms. We then compared the practical performance of these algorithms with the two best previously known sets of algorithms, lookahead-matrix and Mäkinen. We found that minWordAMSorted, crossSectionAMBoolean, and enumAMBoolean have the best practical running times.

Table 4 summarizes the asymptotic running times of the new and old algorithms. Recall that s is the number of states and d is the number of transitions in the NFA, t is the number of characters in the output, c is the number of cross-sections encountered throughout the enumeration, and e is the number of empty cross-sections encountered throughout the enumeration.

	AMSorted	AMBoolean	Intersection	Mäkinen	lookahead-matrix
Min-word	$O((s \log s + d)n)$	$O(dn)$	$O(dn)$	$O(s^2n^2)$	$O(s^{2.376}n)$
Cross-section	$O(ns \log s + dt)$	$O(d(n+t))$	$O(d(n+t))$	$O(d(n^2+t))$	$O(s^{2.376}n + dt)$
Enumeration	$O(c(s \log s + d) + dt)$	$O(d(e+t))$	x	$O(d(e+t))$	$O(s^{2.376}c + dt)$

Table 1. Asymptotic performances of the algorithms.

This shows that $O(dn)$ is the best asymptotic running time for the min-word problem, $O(d(n+t))$ is the best running time for the cross-section problem, and $O(d(e+t))$ is the best running time for the enumeration problem - all achieved by the AMBoolean algorithms. It would be interesting to try to find more efficient algorithms for these problems, or explore the question of whether these are optimal by proving lower bounds.

7 Acknowledgments

We would like to thank Moshe Vardi for a very helpful discussion in which he proposed the idea behind the intersection-based cross-section algorithm.

References

1. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall, London (1971)
2. Anderson, T., Rampersad, N., Santean, N., Shallit, J.: Finite automata, palindromes, patterns, and borders. CoRR **abs/0711.3183** (2007)
3. Ackerman, M., Shallit, J.: Efficient enumeration of regular languages. LNCS **4783** (2007) 226–242
4. Mäkinen, E.: On lexicographic enumeration of regular and context-free languages. Acta Cybernet **13** (1997) 55–61
5. Department of Computer Science, U.o.W.O.: Grail+ (2008) `http://www.csd.uwo.ca/Research/grail/index.html`.

8 Appendix

We summarize our experiments, recording time measured in seconds, in Tables 2–4. When an "x" appears, the test case has not been run to completion due to a high running time.

	n	Mäkinen	AMSorted	AMBoolean	Lookahead-Matrix
L_2	5000	3.120	**0.031**	**0.031**	**0.031**
L_2	50,000	x	**2.777**	3.229	3.307
L_2	10,000	x	**18.283**	35.864	29.047
L_9	2000	4.336	0.0468	**0.0156**	0.0780
L_9	20000	x	0.624	**0.468**	0.889
L_9	50,000	x	**3.4164**	3.7908	4.0872
L_9	80,000	x	18.782	**17.257**	14.6016
L_9	100,000	x	**18.938**	36.722	28.018
L_9	120,000	x	47.252	50.575	**42.713**
Alp. size 2, $\leq$ 10 nodes	3	1.045	**0.8976666**	1.6160000	2.7340000
Alp. size 2, 1 final	5	0.083	**0.078**	0.109	0.114
Alp. size 2, 1 final	7	0.114	**0.088**	4.066	5.699
Alp. size 2, 1 final	8	0.114	**0.099**	36.462	48.599
Alp. size 3, 1 final	7	0.047	**0.042**	3.6244000	4.8932000
Alp. size 3, 1 final	8	0.078	**0.042**	33.285	43.597
Alp. size 3, 1 final	500	42.713	**0.905**	x	x
Alp. size 15, 1 final	100	2.002	**0.608**	x	x
Alp. size 15, 1 final	500	40.092	**2.938**	x	x
Alp. size 5	200	8.611	**0.619**	x	x
Alp. size 10	200	6.839	**0.816**	x	x
Alp. size 10	500	39.359	**1.950**	x	x
Alp. size 10	600	57.164	**2.335**	x	x

Table 2. The performances of the Min-word algorithms.

	n	Mäkinen	AMSorted	AMBoolean	Lookahead-Matrix	Intersection
L_2	1000	0.153	0.009	**0.006**	0.008	x
L_2	5000	3.617	0.092	**0.057**	0.063	x
L_2	10000	14.191	0.217	**0.157**	0.183	x
L_9	1000	1.15	0.045	**0.026**	0.053	x
L_9	5000	28.371	0.387	**0.307**	0.413	x
Alp. size 2, dense graph	5	1.576	**1.538**	1.539	1.722	21.461
Alp. size 2, dense graph	6	**24.163**	24.597	24.424	28.126	x
Alp. size 2, dense graph	7	0.191	0.197	**0.186**	0.226	1.782
Alp. size 2, sparse graph	10	**10.322**	10.385	10.379	12.908	2:45.835
Alp. size 2, all final	5	1.630	1.632	**1.617**	1.748	41.961
Alp. size 10, sparse graph	7	**1.591**	1.580	1.602	1.714	11.302

Table 3. The performances of the Cross-section algorithms.

	n	Mäkinen	AMSorted	AMBoolean	Lookahead-Matrix
L_2	100	0.003	**0.002**	**0.002**	0.003
L_2	1000	0.181	**0.161**	0.162	0.210
L_2	10000	38.681	37.286	**36.896**	49.048
L_9	100	0.004	**0.001**	**0.001**	**0.001**
L_9	1000	0.034	0.029	**0.025**	0.037
L_9	10000	2.999	2.702	**2.678**	0.3461
L_9	20000	14.635	**12.751**	13.027	16.221
Alp. size 2	10	3.966	3.796	**3.262**	7.677
Alp. size 2	12	5.085	4.882	**4.575**	8.995
Alp. size 2, dense graph	12	0.046	0.041	**0.036**	0.047
Alp. size 2, dense graph	20	0.089	0.102	**0.083**	0.089
Alp. size 2, dense graph	30	0.159	0.159	**0.152**	0.167
Alp. size 2, dense graph	100	**15.882**	16.487	16.589	18.477
Alp. size 2, sparse graph	10	0.031	0.028	**0.023**	0.029
Alp. size 2, sparse graph	20	0.036	0.034	**0.028**	0.037
Alp. size 2, sparse graph	30	0.041	0.041	**0.036**	0.044
Alp. size 2, sparse graph	100	0.155	0.146	**0.131**	0.172
Alp. size 2, sparse graph	1000	12.55	12.504	**12.137**	13.944
Alp. size 2, all final.	10	0.039	0.038	**0.032**	0.037
Alp. size 2, all final.	15	0.049	0.052	**0.043**	0.05
Alp. size 2, all final.	20	0.1	0.096	**0.086**	0.098
Alp. size 2, all final.	30	2.511	2.497	**2.426**	3.588
Alp. size 10, sparse graph	100	0.064	0.062	**0.058**	0.066
Alp. size 2, sparse graph	1000	**2.843**	2.844	2.862	3.082
Alp. size 2, sparse graph	2000	**17.010**	17.163	17.685	17.264

Table 4. The performances of the Enumeration algorithms.