Efficient Enumeration of Regular Languages

Margareta Ackerman and Jeffrey Shallit

University of Waterloo, Waterloo ON, Canada
mackerma@uwaterloo.ca, shallit@graceland.uwaterloo.ca

Abstract. The cross-section enumeration problem is to list all words of length n in a regular language L in lexicographical order. The enumeration problem is to list the first m words in L according to radix order. We present an algorithm for the cross-section enumeration problem that is linear in n. We provide a detailed analysis of the asymptotic running time of our algorithm and that of known algorithms for both enumeration problems. We discuss some shortcomings of the enumeration algorithm found in the Grail computation package. In the practical domain, we modify Mäkinen's enumeration algorithm to get an algorithm that is usually the most efficient in practice. We performed an extensive performance analysis of the new and previously known enumeration and cross-section enumeration algorithms and found when each algorithm is preferable.

1 Introduction

Given an NFA N, we wish to enumerate the words accepted by N. By "enumerate" we mean list the words, as opposed to only counting them. Given words $u = u_1u_2\cdots u_n$ and $v = v_1v_2\cdots v_m$, $u < v$ according to *radix order* if $n < m$ or if $n = m$, $u \neq v$, and $u_i < v_i$ for the minimal i where $u_i \neq v_i$. Sorting a set S of words according to radix order is equivalent to sorting words in S of equal length according to lexicographic order and then sorting S by length. Given an NFA accepting a language L, the *enumeration problem* is to enumerate the first m words in L according to their radix order. Let the n^{th} *cross-section* of a language $L \subseteq \Sigma^*$ be $L \cap \Sigma^n$. Given an NFA accepting language L, the *cross-section enumeration problem* is to enumerate the n^{th} cross-section of L in lexicographical order.

Enumeration algorithms enable correctness testing 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, the technique can also be used to fully verify correctness once sufficiently many words have been enumerated [1, p.11].

In addition, regular language enumeration leads to an alternative solution to the next k-subset of an n-set problem. The problem is, given a set $T = \{e_1, e_2, \ldots, e_n\}$, we wish to enumerate all k-subsets of T in alphabetical order. Nijenhuis and Wilf provide a solution to this problem [5, p. 27]. A cross-section enumeration algorithm yields an alternative solution, as follows. Construct an NFA N over the alphabet $\{0, 1\}$ that accepts all words with exactly k 1s. The n^{th} cross-section of N is in bijection with the set of k subsets of T via the function that takes a word $w = a_1a_2\cdots a_n$ in the n^{th} cross-section of N to the k-subset $\{e_i \mid a_i = 1\}$. Therefore, we enumerate the n^{th} cross-section of $L(N)$, which is in bijection with the set of k-subsets of T.

Our contributions are two-fold. On the complexity theoretic side, we give a cross-section enumeration algorithm, `crossSectionLM`, with running time linear in n, the length of words in the cross-section. The best previously known algorithm is quadratic in n. This cross-section enumeration algorithm has a corresponding enumeration algorithm, `enumLM`. To refer to both algorithms

together, we call them the *lookahead-matrix* algorithms. In addition, we perform a theoretical analysis of the previously known algorithms and our algorithms. 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 $s = |Q|$ and $\sigma = |\Sigma|$.

In the practical domain, we give enumeration algorithms, `crossSectionMäkinenII` and `enumMäkinenII`, both of which usually perform better than the other discussed algorithms for their respective problems. The algorithms `crossSectionMäkinenII` and `enumMäkinenII` are a combination of Mäkinen's algorithm [4] and the lookahead-matrix enumeration algorithms. We perform extensive performance analysis of both previous enumeration algorithms and the algorithms presented here, and find when each algorithm performs well. For example, one of our findings is a set of regular languages on which `crossSectionLM` outperforms `crossSectionMäkinenII`.

Here is an outline of the paper. We first introduce the general framework for the enumeration algorithms, after which we describe enumeration algorithms based on Mäkinen's regular language enumeration algorithm [4]. Then we introduce the lookahead-matrix algorithms. Next, we discuss an enumeration algorithm found in the symbolic computation environment, Grail+ 3.0 [3], list a few bugs, and provide a theoretical analysis of the algorithm. We conclude with an analysis and comparison of how these algorithms perform in practice.

2 Enumeration Algorithms

2.1 Enumerating the n^{th} Cross-Section

We introduce a general framework for enumerating the n^{th} cross-section of a language accepted by an NFA, N. First, we find the minimal word $w = a_1 a_2 \cdots a_n$ in the cross-section with respect to radix order, or determine that the cross-section is empty. We say that state q is *i-complete* if starting from q in N there is a path of length i ending at a final state. 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 $1 \leq i < n$. 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 present two methods for finding the minimal word in the following two sections. For now 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. To find the next word, we scan the minimal word $a_1 a_2 \cdots a_n$ 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` $w = a_1 a_2 \cdots a_n$ `and an NFA` N`.`
`OUTPUT: Returns the next word in the` n^{th} `cross-section of` $L(N)$ `according to radix order, if it`

```
        exists. Otherwise, return 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 `enumCrossSection(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 `enumCrossSection` can be used in conjunction with any algorithms for `minWord` and for determining if a state is i-complete. We will use `nextWord` and `enumCrossSection` to form enumeration algorithms based on Mäkinen's algorithm. We will also use these algorithms to form the basis for the lookahead-matrix enumeration algorithms.

2.2 Enumerating the First m Words

We provide a structure for an algorithm that enumerates the first m words accepted by an NFA. The algorithm `enum` finds the minimal word in each cross-section and calls `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 radix 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. This bound is tight, as it is reached in the NFA consisting of a cycle of states, with the start state final. The proof for the following lemma appears in the appendix.

Lemma 1. *Let N be an NFA with s states accepting an infinite language L. The maximum number of consecutive empty cross-sections in L is $s - 1$.*

We will use `enum` as a base for creating a number of enumeration algorithms.

3 Mäkinen's Algorithm

Mäkinen [4] presented a cross-section enumeration algorithm. His algorithm assumes that the language is represented by a regular grammar. A regular grammar is equivalent to an NFA and these representations have the same space complexity. For consistency, we present and analyze Mäkinen's algorithm on NFAs. The algorithm is separated into two parts: finding the first word of length n and finding the remaining words of length n. The original algorithm for finding the remaining words applies only to DFAs, and so the NFA has to be converted to a DFA before the algorithm can be applied. By using `enumCrossSection`, we demonstrate an enumeration algorithm that uses parts of Mäkinen's algorithm and works directly on NFAs, without incurring the exponential size blow-up of subset construction.

To find the minimal word of length n, first find the lexicographically minimal words of length 1 through $n - 1$ starting at each state, via dynamic programming. Theorem 3.2 in [4] states that the minimal and maximal words of length n can be found in $O(n)$ time and space. Mäkinen analyzes the algorithm in the unit-cost model, treating the size of the grammar as a constant. In the unit-cost model all operations, regardless of the size of the operands, have a cost of 1. Since Mäkinen's algorithm uses operands of length n, this model does not fully capture the complexity of the problem. We analyze the algorithm in the bit-complexity model and also take into account the number of states in the NFA.

Algorithm 4 `minWordMäkinen(`n`, `N`)`

```
INPUT: A positive integer n and an NFA N.
OUTPUT: Table A^min[1···n] for each state A ∈ Q where A^min[i] is the minimal word of length i starting
        at state A.

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
    FOR each A ∈ Q
        min = NULL
        FOR each B ∈ Q and minimal a ∈ Σ such that B ∈ δ(A, a)
            IF B^min[i − 1] ≠ NULL
                IF aB^min[i − 1] < min OR min = NULL
                    min ← aB^min[i − 1]
        A^min[i] = min
RETURN {A^min | A ∈ Q}
```

We assume that the complexity of comparison of two words of length n is in the order of the position of the first index where the words differ. We can store the NFA as an adjacency list, keeping track of the edge with the minimal character between any pair of states, which adds constant time and linear space to the implementation of the NFA. Therefore, the running time of `minWordMäkinen` is independent of the alphabet size. The following theorem is proved in the appendix.

Theorem 1. *The algorithm* `minWordMäkinen` *uses* $\Theta(sn)$ *space and* $\Theta(s^2n^2)$ *operations in the worst case.*

The algorithm `minWordMäkinen` finds the minimal word of length n in linear time on DFAs, since the determinism causes all words compared by the algorithm to differ on the leftmost character.

In all variations of Mäkinen's algorithm, to determine if a state is i-complete we check if $A^{min}[i]$ is not `NULL`. To use `minWordMäkinen` with `enumCrossSection` and `enum`, store the sets of states $S_0, S_1, \ldots, S_{n-1}$ on the state stack and then invoke `nextWord`. We know that a cross-section has been fully enumerated when the state stack is emptied, that is, when `nextWord` returns `NULL`. We call this the `enumCrossSection` *cross-section termination method*. Mäkinen introduces an alternative method for determining when a cross-section has been fully enumerated. In addition to finding the minimal word, his algorithm finds the maximal word in the cross-section in a method similar to finding the minimal word. When the maximum word in the cross-section is found we know that the cross-section has been fully enumerated. We call this method *Mäkinen's cross-section termination method*.

When `enumCrossSection` is used with `minWord` replaced by `minWordMäkinen` and Mäkinen's cross-section termination method, we get the algorithm `crossSectionMäkinenI`. When instead of Mäkinen's cross-section termination method the `enumCrossSection` cross-section termination method is used, we get the algorithm `crossSectionMäkinenII`. Similarly, `enumMäkinenI` is `enum` with `minWord` replaced by `minWordMäkinen` and Mäkinen's cross-section termination method. The function `enumMäkinenII` is the same as `enumMäkinenI`, except that it uses the `enumCrossSection` cross-section termination method.

Consider Mäkinen's cross-section termination method. Finding the maximal words adds an overhead of $\Theta(s^2n^2)$ in the worst case and $\Theta(sn)$ in the best case. The `enumCrossSection` cross-section termination method recognizes that a cross-section has been enumerated when the first character of the last word found cannot be replaced. This takes $\Theta(s^2n)$ time in the worst case and constant time in the best case. Recall that the output size, t, is the total number of characters over all words enumerated by an algorithm. With either termination method, once the first word in the cross-section is found, the rest of the work is $O(\sigma s^2 t)$. Therefore, `crossSectionMäkinenI` and

`crossSectionMäkinenII` use $O(s^2n^2 + \sigma s^2 t)$ operations. The difference in the best and worst case performance between these two versions is significant for practical purposes, as will be discussed in Section 6.2.

Theorem 2. *The algorithms* `crossSectionMäkinenI` *and* `crossSectionMäkinenII` *use* $O(s^2n^2 + \sigma s^2 t)$ *operations.*

In the algorithms `enumMäkinenI` and `enumMäkinenII`, after enumerating the n^{th} cross-section we have a table $A^{min}[1 \cdots n]$ for each state A. To improve the performance of these algorithms, when `minWord` is called for $n+1$, we reuse these tables, extending the tables by index $n+1$. Therefore, each call to `minWord`(i, S) costs $O(s^2 i)$. Finding the rest of the words in the cross-section costs $O(\sigma s^2 t)$. For each empty cross-section, the algorithm does $O(s^2)$ operations. Therefore, `enumMäkinenI` and `enumMäkinenII` have $O(\sigma s^2 t + s^2 e)$ operations, where e is the number of empty cross-sections found throughout the enumeration.

Theorem 3. *The algorithms* `enumMäkinenI` *and* `enumMäkinenII` *are* $O(\sigma s^2 t + s^2 e)$*, where* e *is the number of empty cross-sections encountered throughout the enumeration.*

4 Lookahead-Matrix Algorithm

To find the minimal words of length n, Mäkinen's algorithms find the minimal words of length 1 through $n-1$. An alternative approach for finding the minimal word of length n is to generate the characters of the word one at a time, while avoiding going down paths that would not lead to a final state within the required number of transitions. To do so, we need a method of quickly determining whether a word of length i can be completed to a word of length n in $n-i$ steps.

Given an NFA, we precompute M, the adjacency matrix of the NFA; $M_{p,q} = 1$ if there is a transition from state p to state q, and $M_{p,q} = 0$ otherwise. Then compute $M^2, M^3, \ldots, M^{n-1}$ using boolean matrix multiplication. Observe that $M^i_{p,q} = 1$ if and only if there is a path from state p to state q of length exactly i. Note that M^0 is the identity matrix.

To find the minimal word of length n, find the set of $(n-1)$-complete states, S_1, reachable from the start state on the minimal possible symbol a_1. Then find the set of $(n-2)$-complete states, S_2, reachable from any state in S_1 on the minimal symbol. Continue this process for a total of n iterations. Then $a_1 a_2 \cdots a_n$ is the minimal word of length n. The algorithm `minWordLM(`n`,`N`)` finds the minimal word of length n starting from a state in the set of states on top of the state stack S and ending at a final state, or determines that no such word exists. To find the minimal word of length n accepted by N, place $S_0 = \{q_0\}$ on the state stack S and call `minWordLM`.

Algorithm 5 `minWordLM(`n`,`N`)`

```
INPUT: A nonnegative integer n and an NFA N.
OUTPUT: The minimal word of length n accepted by N. Updates state stack S for a potential
        subsequent call to minWord or nextWord.

Compute M, M^2, ..., M^n, if they have not been precomputed
S_0 = top(S)
IF M^n_{q,f} = 0 for all f ∈ F, q ∈ S_0
   return NULL
w = empty word
FOR i ← 0 ··· n − 1
```

$a_{i+1} = \min(a \in \Sigma \mid \exists u \in S_i, f \in F$ where $M_{v,f}^{n-1-i} = 1$ for some $v \in \delta(u,a))$
$w = wa_{i+1}$
$S_{i+1} = \{v \in \cup_{u \in S_i} \delta(u, a_{i+1}) \mid M_{v,f}^{n-1-i} = 1$ for some $f \in F\}$
`IF` $i \neq n-1$
`push(`S`,`S_{i+1}`)`
`return` w

Since the matrices require $O(s^2n)$ space, `minWordLM` uses $O(s^2n)$ space. Finding each character of the minimal word can be implemented in a way that uses $O(s^2)$ operations. The standard matrix multiplication algorithm is $O(s^3)$. Strassen's matrix multiplication algorithm has $O(s^{2.81})$ operations [6]. The best bound on the matrix multiplication problem is $O(s^{2.376})$ [2]. All other operations in the algorithm cost $O(s^2n)$. Therefore, `minWordLM` can be made to run in $O(s^{2.376}n)$ operations. However, the matrices have to be unreasonably large before the differences in these multiplication methods become apparent in practice.

Theorem 4. *The algorithm* `minWordLM` *finds the minimal word of length* n *in* $O(s^{2.376}n)$ *time and* $O(s^2n)$ *space.*

Note that `minWordLM` can be easily modified to find the maximal word of length n. In the bit-complexity model, `minWordMäkinen` is quadratic in n. The algorithm `minWordLM` is linear in n in the bit-complexity model. Theorem 3.2 of [4] states that the minimal and maximal words of length n in a regular language can be found in linear time in n in the unit-cost model. The algorithm `minWordLM` proves that this is also true in the bit-complexity model.

Replace `minWord` by `minWordLM` in `nextWord` and use the matrices to determine i-completeness to get the method `nextWordLM`. Then using `nextWordLM` instead of `nextWord`, we get modified versions of `enumCrossSection` and `enum`, which we call `crossSectionLM` and `enumLM`, respectively. See appendix for details. Looking for the minimal word costs $O(s^{2.376}n + \sigma sn)$ and finding all consecutive words costs $O(\sigma s^2 t)$. Therefore `crossSectionLM` costs $O(s^{2.376}n + \sigma s^2 t)$.

Theorem 5. *The algorithm* `crossSectionLM` *uses* $O(s^{2.376}n + \sigma s^2 t)$ *operations.*

If an empty cross-section is encountered in `enumLM`, the algorithm performs $O(s^{2.376})$ operations to determine that. Therefore, `enumLM` uses $O(s^{2.376}(m+e)+\sigma s^2 t)$ operations, where e is the number of empty cross-sections encountered during the enumeration. Note that if the total number of cross-sections encountered by the algorithm is c, then the running time of `enumLM` is $O(s^{2.376}c + \sigma s^2 t)$.

Theorem 6. *The algorithm* `enumLM` *uses* $O(s^{2.376}(m+e)+\sigma s^2 t)$ *operations, where* e *is the number of empty cross-sections encountered throughout the enumeration.*

5 Grail Enumeration Algorithm

The symbolic computation environment Grail+ 3.0 has an `fmenum` function that finds the m lexicographically first words accepted by an NFA. Consider the potentially infinite tree of paths that can be traversed on an NFA. The function `fmenum` performs breadth first search (BFS) on that tree until the required number of words is found. More precisely, it looks for all words of length n by searching all paths of length n starting at the start state, distinguishing paths that terminate at a final state. It searches for words of length $n+1$ by completing the paths of length n. Based on the Grail algorithm, we present a cross-section enumeration algorithm `crossSectionBFS`.

Let the n^{th} *NFA-cross-section* be the set of all words appearing on paths of length n of an NFA that start at the start state. Given all words in the $(n-1)^{st}$ NFA-cross-section and the ends states of the corresponding paths, `nextCrossSection` finds all words in the n^{th} NFA-cross-section as well as the end states of the corresponding paths. To find all words of length n accepted by an NFA, `fmenum` finds the words in the n^{th} NFA-cross-section and selects all words for which there is a path that ends at a final state.

Algorithm 6 `nextNFACrossSection(`N`, prevSec, prevSecStates)`

```
INPUT: NFA N. The set, prevSec, of all words of some length l ≥ 0 that occur on paths in N
       starting at s_0. An array, prevSecStates, where prevSecStates[w]= δ(q_0, w) for all
       w ∈ prevSec.
OUTPUT: Returns the pair (nextSec, nextSecStates), where nextSec is the set of all words
        in L(N) of length l + 1 and nextSecStates[w]= δ(q_0, w) for all w ∈ nextSec.

nextSec = ∅
FOR i ← 1 ··· size(prevSec)
    currWord = prevSec[i]
    currNodes = prevSecStates[currWord]
    FOR each currNode in currNodes
        FOR each edge adjacent to currNode
            newWord = currWord + value(edge)
            IF newWord ∉ nextSec
               nextSec = nextSec ∪ newWord
               nextSecStates[newWord] = ∅
            nextSecStates[newWord] = nextSecStates[newWord] ∪ destination(edge)
RETURN (nextSec, nextSecStates)
```

Algorithm 7 `crossSectionBFS(`n`, `N`)`

```
INPUT: A nonnegative integer n and an NFA N.
OUTPUT: Visits all words of length n accepted by N in lexicographical order.

FOR each state in N
    sort outgoing edges
words = ∅
emptyWord = ""
crossSec = {emptyWord}
crossSecStates[emptyWord] = {q_0}
IF n = 0
   IF q_0 ∈ F
      visit emptyWord
ELSE
   FOR i ← 1 ··· n
      (crossSec, crossSecStates) = nextNFACrossSection(N, crossSec, crossSecStates)
   sort(crossSec)
   FOR each word in crossSec
       IF crossSectionStates[word]∩F ≠ ∅
          visit word
```

The BFS enumeration algorithm, `enumBFS`, calls `crossSectionBFS` until the required number of words is found. When we refer to our implementation of the BFS enumeration algorithm we call it `enumBFS` and when we refer to the original Grail implementation, we call it `fmenum`.

We found a number of bugs in `fmenum`. The function does not always display words in the cross-sections in radix order (equivalently, lexicographic order). When asked to enumerate the first two words accepted by the NFA in Figure 1(a), the following input to `fmenum` results in an output of 1 followed by a 0.
(START) |− 0
0 1 1
0 0 2
1 −| (FINAL)
2 −| (FINAL)

In addition, `fmenum` does not always display all words it should. When `fmenum` is called with $n = 1000$ and a DFA that accepts words over $(0 + 1)^*$ such that the number of 1s is congruent to 0 mod 3 (see Figure 1(b)), `fmenum` is missing 11000000000001.

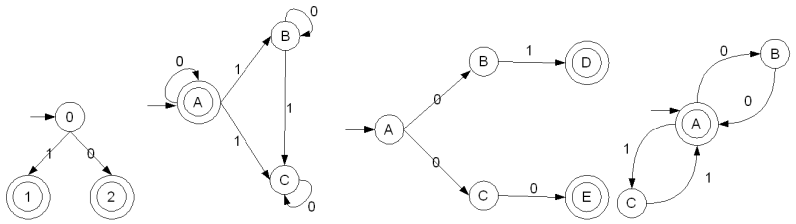

Fig. 1.

Without explicit sorting of the words, words found by BFS algorithms will likely not be visited in radix order. Sorting the edges based on their alphabet symbol reduces the frequency of the problem, but does not eliminate it. If we call `enumBFS` on the NFA in Figure 1(c), then while enumerating words of length 2 we attempt to complete the string "0", which was found while enumerating the previous cross-section. Since both states B and C are reached on the symbol 0, state B may be chosen to complete the word. Thus, the algorithm finds "01" before it finds "00". To solve this problem, we sort the words after they are found.

The algorithm `crossSectionBFS` may do exponential work in n for empty output. Consider the NFA in Figure 1(d). If we enumerate the n^{th} cross-section of this NFA for $n = 2j + 1$, $j \geq 0$, the algorithm performs over $(2j) \cdot 2^j \in \Theta(n2^{n/2})$ operations and has empty output. Note that the NFA in Figure 1(d) is minimal. On non-minimal NFAs, the running time could be worse. The running time of the BFS enumeration algorithms depends on the structure of the NFA. The i^{th} NFA-cross-section contains at most σ^i words and the algorithm does $O(\sigma s^2)$ operations per word in the i^{th} NFA-cross-section when enumerating the $(i+1)^{st}$ NFA-cross-section. Therefore, the algorithm performs $O(\sigma^i s^2)$ operations for the i^{th} NFA-cross-section. Therefore, `crossSectionBFS` is $O(s^2\sigma^n)$. The algorithm `enumBFS(`m, N`)` is bounded by $O(s^2\sigma^k)$, where k is the length of the last cross-section examined. Further, $k \leq ms$ as for every word enumerated by `enumBFS` there are at most s empty cross-sections examined.

Theorem 7. *The algorithm* `crossSectionBFS`(n, N) *has* $O(s^2\sigma^n)$ *operations. The algorithm* `enumBFS`(m, N) *has* $O(s^2\sigma^k)$ *operations, where* $k \leq ms$ *is the length of the last cross-section examined.*

6 Experimental Results

6.1 Implementation

We discussed the following algorithms: `enumMäkinenI`, `crossSectionMäkinenI`, `enumMäkinenII`, `crossSectionMäkinenII`, `enumLM`, `crossSectionLM`, `enumBFS`, and `crossSectionBFS`. We also introduce the naive algorithm, `enumNaive`, which generates words over Σ^* in radix order and checks which are accepted by the NFA, until the required number of words is found or it is determined by the `enumCrossSection` cross-section termination method that the language is finite. The algorithm `enumNaive` has a corresponding cross-section enumeration algorithm, `crossSectionNaive`, that generates all words over Σ^n in radix order and checks which are accepted by the NFA. The running time of `crossSectionNaive` is $O(s^2\sigma^n)$, since the algorithm may have to do up to s^2 operations for every character in a word in Σ^n. The algorithm `enumNaive`(m, N) costs $O(s^2\sigma^k)$, where k is the length of the last word examined by the algorithm. As in `enumBFS`, $k \leq ms$. We implemented these algorithms and compared their performance. We represent NFAs as adjacency-lists. To improve performance, edges adjacent to each vertex are sorted based their associated Σ symbols.

6.2 Performance Comparison

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. Each algorithm was run between 1 and 10 times on each NFA, and the average running time was recorded.

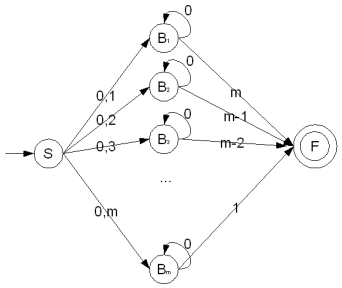

Fig. 2. NFA L_m.

In addition to randomly generated tests, the algorithms were tested on the DFA that accepts the language 1^*, the DFA that accepts the language $(0 + 1)^*$, and some NFAs from the set $L =$

$\{L_m \mid m \geq 2\}$, found in Figure 2. The NFAs in L are important because they take quadratic time on `crossSectionMäkinen`.

The naive algorithms perform reasonably well on small NFAs when the alphabet is of size less than 3, but even in these cases they tend to be slower than the other algorithms. With an alphabet size greater than 3, the naive algorithms are unreasonably slow. For large values of s, the naive algorithms are very slow, even on NFAs over the binary alphabet. The only case found in which the naive algorithms outperform the other algorithms is on NFAs with a unary alphabet where all states are final.

The BFS algorithms tend to perform well on small NFAs for small values of n. The algorithms `enumBFS` and `crossSectionBFS` outperform the other enumeration algorithms on 1^*, and `crossSectionBFS` outperforms the other algorithms for L_2 and L_9 (see Figure 2). In addition, the BFS algorithms are faster than the naive algorithms. However, `enumBFS` and `crossSectionBFS` are significantly slower than both Mäkinen and lookahead-matrix on most test cases.

Mäkinen and lookahead-matrix were slower than BFS on the language 1^*, for which Mäkinen and lookahead-matrix are particularly poorly suited. After each minimal word is found, the Mäkinen and lookahead-matrix algorithms go through the state stack searching for a character that can be replaced, finding none. Both algorithms end up performing a lot of redundant work on this particular language.

The efficiency of Mäkinen and lookahead-matrix on any NFA N can be estimated by the average length of the common prefix between two consecutive words in the same cross-section of $L(N)$. Therefore, Mäkinen and lookahead-matrix are particularly well suited for dense languages. This is confirmed in practice, as the performance of the Mäkinen and lookahead-matrix algorithms improves as the alphabet size increases. Performance improves further when, in addition to a larger alphabet, the number of edges or final states increases.

The top competing algorithms on almost all randomly generated tests are MäkinenII and lookahead-matrix. As the alphabet size increases, the difference between the efficiency of the algorithms decreases. On average, MäkinenII performs best. The performance of lookahead-matrix is consistently close to that of MäkinenII. Lookahead-matrix overtakes MäkinenII on some test cases where there is only a single final state.

As expected, MäkinenII is significantly more efficient than MäkinenI on NFAs with unary alphabets, due to the overhead in MäkinenI of searching for the maximal word in each cross-section where all cross-sections have a unique element. MäkinenII is also much more efficient on NFAs corresponding to sparse graphs. While on a few other test cases there is a significant disparity in the performance of the algorithms, their performance is similar on average, with MäkinenII performing a little better on most tests.

The algorithm `minWordMäkinen` is $O(s^2n^2)$ in the worst case and $O(sn)$ in the best case. We implemented the lookahead-matrix algorithms with the standard $O(s^3)$ matrix multiplication algorithm. Therefore, our implementation of `minWordLM` is $O(s^3n)$. Finding the rest of the words is $O(s^2t)$ for both algorithms. All other operations in the algorithms are identical. The performance difference in the average case can be explained by the additional factor of s in lookahead-matrix when searching for the minimal word of length n and the hypothesis that the worst case of $O(s^2n^2)$ for `minWordMäkinen` is not usually reached on random NFAs. This provides a theoretical basis for the proposition that the slightly faster average performance of `crossSectionMäkinenII` and `enumMäkinenII` over that of the lookahead-matrix algorithms is not symptomatic of a larger problem with the lookahead-matrix algorithms.

On NFAs where Mäkinen's cross-section enumeration algorithms are quadratic in n, `crossSectionLM` performs significantly better than Mäkinen's cross-section algorithms. On L_9, `crossSectionLM` runs over 50 times faster than `crossSectionMäkinenI` and `crossSectionMäkinenII`. Note also that it is sufficient for an NFA to have an NFA in L as a subgraph in order for the Mäkinen algorithms to have a quadratic running time in n.

From these results, we find that on typical data, Mäkinen algorithms with the `enumCrossSection` cross-section termination method tend to perform slightly faster than all other algorithms. However, in applications where a bounded worst case running time is essential, the lookahead-matrix algorithms are preferable.

The tests were run on Microsoft Windows XP Professional Version 2002 Service Pack 2, AMD Sempron(tm) 1.89 GHz, 1.00 GB of RAM.

7 Acknowledgements

We would like to thank David Loker for proofreading this paper and for his helpful suggestions.

References

1. Conway, John Horton. *Regular Algebra and Finite Machines.* London, Chapman and Hall, (1971).
2. Coppersmith D. and Winograd, S. "Matrix multiplication via arithmetic progressions." *J. Symbolic Comput.* **9** (1990), no. 3, 251–280.
3. http://www.csd.uwo.ca/Research/grail/index.html. Department of Computer Science, University of Western Ontario, Canada.
4. Mäkinen, Erkki. "On lexicographic enumeration of regular and context-free languages." *Acta Cybernetica*, **13** (1997), 55–61.
5. Nijenhuis, Albert. Wilf, Herbert. "Combinatorial algorithms. For computers and calculators." Second edition. Computer Science and Applied Mathematics. Academic Press, Inc. Harcourt Brace Jovanovich, Publishers, New York-London, 1978.
6. Strassen, Volker. "Gaussian elimination is not optimal." *Numer. Math.* **13** (1969), 354–356.

8 Appendix

Lemma 1. *Let N be an NFA with s states accepting an infinite language L. The maximum number of consecutive empty cross-sections in L is $s-1$.*

Proof. Suppose L is infinite but contains s consecutive empty cross-sections, say of length $m, m+1, \ldots, m+s-1$. Let $w = a_1 a_2 \cdots a_r$ be a shortest word in L of length $\geq m+s$. Such a word exists because L is infinite. Consider the accepting configuration $q_0, q_1, \ldots, q_r$ of w in N. Now look at the sequence of states $q_m, q_{m+1}, \ldots, q_{m+s-1}$. None of these s states are accepting, since otherwise there would be a word in the associated cross-section. But there is at least one accepting state in N. So there are at most $s-1$ distinct states in the sequence. Therefore some state is repeated. If we cut out the loop, we get either a shorter word in L of length $\geq m+s$ or a word of length between m and $m+s-1$.

Theorem 1. *The algorithm* `minWordMäkinen` *uses $\Theta(sn)$ space and $\Theta(s^2n^2)$ operations in the worst case.*

Proof. The two expensive operations are concatenation and comparison of words. Concatenation of words can be performed in constant time by changing the mode of storage: Instead of storing a word w of length i in $A^{min}[i]$, store the pair (a, B) such that $w = aB^{min}[i-1]$. With this modification, `minWordMäkinen` uses $\Theta(sn)$ space.

The time complexity of comparing two words of length n is $O(n)$. The number of symbols compared throughout the algorithm is $O(n^2)$. Since the states of an NFA can form a complete graph, the worst case running time is $O(s^2n^2)$. This bound is reached on the NFA in Figure 3.

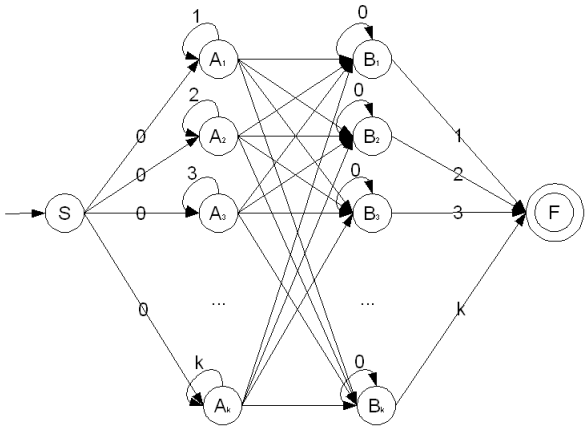

Fig. 3. $\delta(A_i, a_i) = \{B_1, B_2, \ldots, B_k\}$ for all distinct a_i.

To fill in $A_j^{min}[i]$, which represents the minimal word of length i starting at state A_j, the minimal words of length $i-1$ starting at states $B_1, B_2, \ldots, B_{\frac{s-2}{2}}$ are compared. The minimal word of length $i-1$ starting at state B_l is $0^{i-2}l$. Therefore, comparing each pair of minimal words requires $i-1$ steps, and there is a total of $\frac{s-2}{2} - 1$ comparisons. So filling all the tables A_i^{min} takes $\Theta(s^2n^2)$ operations.

Algorithm 8 crossSectionLM(n,N)

```
INPUT: A nonnegative integer n and an NFA N.
OUTPUT: Enumerates the n^th cross-section of L(N).

Find M, M^2, ..., M^n
S = empty stack
push(S, {q_0})
w = minWordLM(n,N)
WHILE w ≠ NULL
   visit w
   w = nextWordLM(w,N)
```

Algorithm 9 enumLM(m, N)

```
INPUT: A nonnegative integer m and an NFA N.
OUTPUT: Enumerates the first m words accepted by N according to radix order,
        if there are at least m words. Otherwise, enumerates all words accepted by N.

i = 0
len = 0
numCEC = 0
WHILE i < m AND numCEC < s DO
   IF len > 1
      find M^i
   S = empty stack
   push(S, {q_0})
   w = minWordLM(len,S)
   IF w = NULL
      numCEC = numCEC+1
   ELSE
      numCEC = 0
      WHILE w ≠ NULL AND i < m
         visit w
         w = nextWordLM(w,N)
         i = i+1
   len = len+1
```

The following tables list some of the running times of the algorithms. The first table lists the results of the described enumeration algorithms and the second lists the results of the cross-section enumeration algorithms. When an "x" appears, the test case has not been run to completion due to an unreasonably high running time.

NFA type	num NFAs	num runs	n	crossSectionBFS	crossSectionMäkinenII	crossSectionMäkinenI	crossSectionLM	naive
1^* dfa	1	5	10000	0.2	0.24	0.43	0.26	0.33
1^* dfa	1	5	50000	8.48	9.11	21.27	8.95	18.47
$(0+1)^*$ dfa	1	10	15	0.71	0.18	0.15	0.16	0.42
$(0+1)^*$ dfa	1	10	20	x	5.11	5.17	5.43	16.86
L_2	1	5	1000	0.05	0.05	0.19	0.02	x
L_2	1	1	10000	3.2	21.75	21.91	0.53	x
L_9	1	5	1000	0.32	1.48	1.46	0.1	x
L_9	1	1	10000	6.3	52.69	54.45	1.02	x
Alp. size 2, $\leq$ 10 nodes	100	5	4	1.99	0.23	0.24	0.24	0.27
Alp. size 2, $\leq$ 10 nodes	100	5	5	35.69	2.72	2.79	3.03	3.26
Alp. size 5, $\leq$ 10 nodes	100	5	4	x	7.48	7.51	7.49	x
Alp. size 10, $\leq$ 10 nodes	100	1	4	x	01:04.27	01:03.78	01:08.28	x
Alp. size 20, $\leq$ 10 nodes, all final	100	5	3	x	15.98	15.97	17.13	x

NFA type	num NFAs	num runs	m	enumBFS	enumMäkinenII	enumMäkinenI	enumLM	naive
1^* dfa	1	10	1000	0.01	1.78	2.91	1.93	1.4
1^* dfa	1	3	2000	0.02	10.91	16.69	12.15	9.82
$(0+1)^*$ dfa	1	10	1000	0.01	0.01	0	0	0.01
$(0+1)^*$ dfa	1	10	2000	0.02	0.01	0	0.01	0.02
$(0+1)^*$ dfa	1	5	100000	1.79	0.53	0.48	0.5	1.25
$(0+1)^*$ dfa	1	5	1000000	18.48	4.73	4.75	5.35	14.68
L_2	1	5	1000	0.02	0.33	0.41	0.41	x
L_2	1	5	5000	0.25	15.84	20.39	18.91	x
L_2	1	5	10000	0.59	01:31.84	01:58.11	01:57.20	x
L_9	1	5	1000	0.01	0.05	0.05	0.07	x
L_9	1	5	5000	0.05	1.31	1.36	1.7	x
L_9	1	5	10000	0.18	6.17	6.48	8.33	x
Alp. Size 1, $\leq$ 10 nodes, 1 final	100	1	7	03:30.41	00:21.03	00:46.28	00:30.20	01:02.72
Alp. Size 1, $\leq$ 10 nodes, all final	100	1	8	43.33	14.17	29.13	17.06	12.34
Alp. Size 1, $\leq$ 10 nodes	100	1	8	55.23	15.98	34.25	19.75	17.05
Alp. size 2, $\leq$ 10 nodes	100	10	10	0.2	0.08	0.08	0.08	4.6
Alp. size 2, $\leq$ 10 nodes	100	3	50	35.36	1.74	1.94	2.13	x
Alp. size 2, $\leq$ 10 nodes	100	1	100	x	38.77	40.09	44.33	x
Alp. size 5, $\leq$ 10 nodes	100	10	100	4.63	0.31	0.33	0.32	x
Alp. size 5, $\leq$ 10 nodes	100	5	1000	x	43.94	50.61	47.66	x
Alp. size 10, $\leq$ 10 nodes	100	5	100	1.14	0.17	0.16	0.16	16.6
Alp. size 10, $\leq$ 10 nodes	100	1	500	24.84	1.03	1.17	1.03	x
Alp. size 10, $\leq$ 10 nodes	100	5	1000	40.42	3.15	4.19	3.32	x
Alp. size 10, $\leq$ 10 nodes	19	1	5000	06:02.81	02:02.22	03:12.94	02:23.81	x
Alp. 2, $\leq$ 10 n, 1 final state	100	20	100	x	0.18	0.18	0.17	x
Alp. 5, $\leq$ 10 n, 1 final state	100	5	100	x	0.27	0.26	0.25	x
Alp. 5, $\leq$ 10 n, 1 final state	100	5	1000	x	24.21	28.5	27.43	x
Alp.10, $\leq$ 10 n, 1 final state	100	20	100	x	0.17	0.16	0.16	x
Alp.10, $\leq$ 10 n, 1 final state	100	5	1000	x	1.43	1.44	1.52	x
Alp.10, $\leq$ 10 n, 1 final state	100	5	5000	x	29.64	29.75	32.72	x
Alp. size 2, $\leq$ 10 nodes, dense grph	100	20	100	x	22.45	24.19	26.33	x
Alp. size 10, $\leq$ 10 nodes, dense grph	100	5	100	x	0.19	0.19	0.18	x
Alp. size 10, $\leq$ 10 nodes, dense grph	100	5	1000	x	1.79	1.81	1.86	x
Alp. size 10, $\leq$ 10 nodes, dense grph	100	5	5000	x	37.71	37.79	40.18	x
Alp. size 2, $\leq$ 10 nodes, sparse grph	100	5	100	x	0.28	0.33	0.32	x
Alp. size 2, $\leq$ 10 nodes, sparse grph	100	5	1000	x	26.99	40.18	31.08	x
Alp. size 10, $\leq$ 10 nodes, sparse grph	100	5	100	x	0.13	0.15	0.14	x
Alp. size 10, $\leq$ 10 nodes, sparse grph	100	5	1000	x	7.58	12.28	8.58	x
Alp. size 5, $\leq$ 10 nodes, all final	100	10	100	2.35	0.25	0.27	0.26	x
Alp. size 5, $\leq$ 10 nodes,all final	100	5	500	01:36.57	4.11	4.68	4.29	x
Alp. size 2, $\leq$ 10 nodes, all final	100	5	30	5.35	3.12	6.31	5.79	x
Alp. size 5 $\leq$ 3n, all final	100	5	100	0.26	0.18	0.21	0.2	x
Alp. size 5 $\leq$ 3n, all final	100	5	500	2.53	3.26	4.9	3.63	x
Alp. size 5 $\leq$ 3n, all final	100	5	1000	31.32	17.01	29.07	20.18	x
Alp. size 5 $\leq$ 20n, all final	100	5	100	31.95	1.52	1.56	1.56	x
Alp. size 5 $\leq$ 20n, all final	100	5	150	30.4	1.97	2.07	2.06	x
Alp. size 20 $\leq$ 10 nodes, all final	100	5	100	3.29	0.28	0.29	0.28	0.49
Alp. size 20 $\leq$ 10 nodes, all final	100	5	1000	x	1.97	2.01	2.07	x