

15-453: FLAC  
Computational Complexity of  $K_n$   
Final Project Report  
Due: 5/3/2007  
Jeremiah Blocki  
Advisor: Leonid Kontorovich

## Definitions

Given a fixed alphabet  $\Sigma$  and  $x, y \in \Sigma^*$  define:

1.  $DFA(n, \Sigma) := \{M : M \text{ is a DFA with } n \text{ states and alphabet } \Sigma\}$  (1)
2.  $DFS(n, \Sigma) := \{M : M \text{ is a semi-automaton with } n \text{ states and alphabet } \Sigma\}$  (1)  
Note that a semi-automaton is a DFA without the labeling of final states.
3.  $T^{(n)} := \{M : M \text{ is a Turing Machine with } n \text{ states}\}$
4.  $K_n(x, y) := |\{M : M \in DFA(n, \Sigma) \text{ and } \{x, y\} \subset L(M)\}|$  (1)
5.  $S_n(x, y) = \{M \in DFS(n, \Sigma) : \delta_M(x) = \delta_M(y)\}$  (1)

Clearly,  $K_n$  is well defined and computable. It is an important kernel because it "can be easily adapted to render all the regular languages linearly separable." If  $K_n$  were efficiently computable then it becomes easy to learn any regular language. Unfortunately, there is no known way to compute  $K_n$  efficiently. Given a DFA with  $n$  states  $\{q_0, \dots, q_{n-1}\}$ , designate  $q_0$  as the start state. Hence, there are  $n^{n\Sigma}$  distinct transition functions and  $2^n$  labellings of final states:

$$|DFA(n, \Sigma)| = 2^n |DFS(n, \Sigma)| = 2^n n^{n\Sigma} \quad (1)$$

This means that brute force computation of  $K_n$  is only tractable for small values of  $n$ . However, there is an efficient  $\epsilon$  approximation for  $K_n$  (2). The purpose of this paper is to investigate the computational complexity of this universal kernel  $K_n$ .

Let  $M \in DFS(n, \Sigma)$ ,  $x, y \in \Sigma^*$

If  $\delta_M(x) = \delta_M(y)$  then exactly half ( $2^{n-1}$ ) of the labelings of final states in  $M$  produce a DFA that accepts both  $x$  and  $y$ .

Else if  $\delta_M(x) \neq \delta_M(y)$  then exactly one-fourth ( $2^{n-2}$ ) of the labelings of final states in  $M$  which produce a DFA accepting both  $x$  and  $y$ .

Therefore,  $K_n(x, y) = 2^{n-1}|S_n(x, y)| + 2^{n-2}(n^{|\Sigma|} - |S_n(x, y)|)$   
Hence, computing  $K_n$  and  $|S_n|$  are equivalent problems.

## Brute Force Computation of $|S_n(x, y)|$

Fix  $\Sigma = \{0,1\}$  and represent a DFS with  $n$  states as a 2 by  $n$  array of integers (the transition function). Then there is a 1-1 mapping between integers and semi-automata defined as follows:

```
// Assuming the DFA is "small enough" (<= 6 states)
// At 7 states already  $7^{(2*7)} > 2^{32} - 1$ , takes in an input
// array of dimensions: ALPHABET_SIZE x MAX_STATES, but only
// considers the first n states to compress the DFA
inline unsigned int compressDFS(int Delta[ALPHABET_SIZE][MAX_STATES], int n) {
    unsigned int total = 0;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        for (int j = 0; j < n; j++) {
            total *= n;
            total += Delta[i][j];
        }
    }
    return total;
}

// Only works for n <= 6, see note above
//
// Only fills the first n states, decompresses based on n
inline void decompressDFS(unsigned int compressedDFA, int n,
                        int DFS[ALPHABET_SIZE][MAX_STATES]) {
    unsigned int comp = compressedDFA;
    for (int i = ALPHABET_SIZE-1; i >= 0; i--){
        for (int j = n-1; j >= 0; j--) {
            DFS[i][j] = comp % n;
            comp /= n;
        }
    }
}
```

Computing  $|S_n(x, y)|$  is then a simple matter of looping through all integers less than  $n^{2^n}$ , obtaining the corresponding semi-automaton ( $M$ ) and running it on  $x$  and  $y$  to see if  $\delta_M(x) = \delta_M(y)$ . Using this code I was able to compute  $K_1, K_2, K_3, K_4, K_5$  by brute force for all  $x, y \in \Sigma^*$  such that  $|x|, |y| \leq 5$  within a day.

Conclusion: Brute-Force computation is feasible only for semi-automata with fewer than 7 states, though even computing  $S_6(x, y)$  takes several hours.

## Closed forms for $K_n$ given $|x|, |y| \leq 2$

Let  $\sigma_1, \sigma_2, \sigma \in \Sigma$  with  $\sigma_1 \neq \sigma_2$   $K_n(x, x) = \frac{1}{2} 2^n n^{|\Sigma|}$

Claim:  $K_n(01, 10) = \frac{1}{4} 2^n n^{2n} (1 - \frac{1}{n^2} - \frac{1}{n^3} + \frac{3}{n})$

Proof:

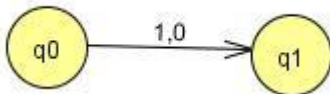
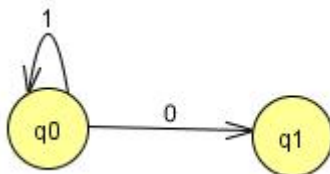
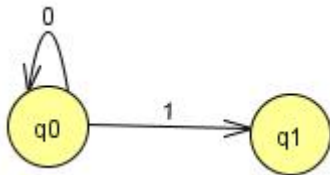
Consider the transitions  $\delta(q_0, 0), \delta(q_0, 1)$

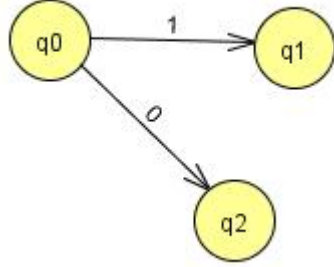
1. Case 1 (Probability  $\frac{1}{n^2}$ )



$$Pr[\delta(q_0, 01) = \delta(q_0, 10)] = 1$$

2. Cases 2,3,4,5: (Probability:  $1 - \frac{1}{n^2}$ )





$$Pr[\delta(q_0, 01) = \delta(q_0, 10)] = \frac{1}{n}$$

$$\therefore K_n(01, 10) = 2^{n-1} \left( \frac{1}{n^2} * 1 + \left(1 - \frac{1}{n^2}\right) * \frac{1}{n} \right) + 2^{n-2} \left( \left(1 - \frac{1}{n^2}\right) \frac{n-1}{n} \right) = \frac{1}{4} 2^n n^{2n} \left( 1 + \frac{1}{n} + \frac{1}{n^2} - \frac{1}{n^3} \right)$$

Using similar analysis it can be shown that:

1.  $K_n(x, x) = \frac{1}{2} 2^n n^{|\Sigma|^n}$
2.  $\sigma_1, \sigma_2 \in \Sigma (\sigma_1 \neq \sigma_2) \rightarrow K_n(\sigma_1, \sigma_2) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 2 * \frac{1}{n} + \frac{n-1}{n} \right) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( \frac{n+1}{n} \right)$
3.  $\sigma \in \Sigma \rightarrow K_n(\epsilon, \sigma) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( \frac{n+1}{n} \right)$
4.  $\sigma \in \Sigma \rightarrow K_n(\epsilon, \sigma\sigma) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 2 * \frac{1}{n} + 2 * \frac{1}{n} * \left( \frac{n-1}{n} \right) + \left( \frac{n-1}{n} \right)^2 \right)$
5.  $\sigma_1, \sigma_2 \in \Sigma, (\sigma_1 \neq \sigma_2) \rightarrow K_n(\epsilon, \sigma_1\sigma_2) = K_n(\epsilon, \sigma_2\sigma_1) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( \frac{n+1}{n} \right)$
6.  $K_n(\sigma, \sigma\sigma) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 2 * \frac{1}{n} + 2 * \left( \frac{n-1}{n^2} \right) + \left( \frac{n-1}{n} \right)^2 \right) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 1 + \frac{2n-1}{n^2} \right)$
7.  $K_n(\sigma_1, \sigma_2\sigma_1) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 2 * \frac{1}{n^2} + 2 * \frac{n-1}{n^2} + 2 * \frac{n-1}{n^2} + \left( \frac{n-1}{n} \right)^2 \right)$
8.  $K_n(\sigma_1, \sigma_2\sigma_2) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 2 * \frac{1}{n^2} + \frac{n-1}{n^2} + 2 * \frac{n-1}{n^2} + \left( \frac{n-1}{n} \right)^2 \right)$
9.  $K_n(\sigma_1, \sigma_1\sigma_2) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( \frac{n+1}{n} \right)$
10.  $K_n(\sigma_1\sigma_2, \sigma_2\sigma_1) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 1 + \frac{1}{n} + \frac{1}{n^2} - \frac{1}{n^3} \right)$
11.  $K_n(\sigma_1\sigma_1, \sigma_2\sigma_2) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 1 - \frac{1}{n^2} - \frac{1}{n^3} + \frac{3}{n} \right)$
12.  $K_n(\sigma_1\sigma_2, \sigma_2\sigma_2) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( 2 * \frac{1}{n^2} + 2 * \frac{n-1}{n^2} + \frac{(n-1)(n+1)}{n^2} \right)$
13.  $K_n(\sigma_1\sigma_1, \sigma_1\sigma_2) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( \frac{n+1}{n} \right)$
14.  $K_n(\sigma_2^k \sigma_1, \sigma_2^j) = \frac{1}{4} 2^n n^{|\Sigma|^n} \left( \frac{n+1}{n} \right)$
15. Also Note:  $K_n(x, y) \leq K_n(xz, yz)$

The closed form formula's above suggests that  $K_n(x, y)$  can be computed efficiently if  $|x|, |y|$  are both small.

### Algorithm to Compute $K_n(x, y)$ efficiently if $|x|, |y|$ are small.

1. Start with a uninitialized DFS (M) of n states (no transitions defined), with only  $q_0$  marked.
  2. return  $FractionDFSsEndingOnSameState(x, y, M)$
- ```

FractionDFSsEndingOnSameState(x1, y1, DFA) {
  run the DFS on x1 and y1 on the DFA until both hit an uninitialized transition
  ( if both x1 and y1 are in the same state and x1.nextChar == y1.nextChar then
  we only need to decide 1 transition)
  enumerate separate cases for the 2 transitions as well as their
  probabilities ie. for each marked state both transitions go there with
  probability:  $1/n^2$ 
    for each pair of marked states the first transition goes to the first
    state the second transition goes to the second state with
    probability:  $1/n^2$ 
    for each pair of marked states the second transition goes to the
    first state/the first transition goes to the second state with
    probability:  $1/n^2$ 
    both transitions go to a different unmarked state with probability
    (note in this case we mark the new states)
     $\#unmarked * (\#unmarked - 1) / n^2$ 
    both transitions go to the same unmarked state with probability
    (note in this case we mark the new states)
     $\#unmarked / n^2$ 
  recursively compute  $FractionDFSsEndingOnSameState$  for each case and
  multiply by the probabilities for each case
  Sum the results for each case
}

```

$$K_n(x, y) = \frac{1}{4} 2^n n^{\Sigma |x|} * (1 + FractionDFSsEndingOnSameState(x, y))$$

Note that the running time is still exponential on  $|x|, |y|$ , but it shows whenever  $|x|, |y| \ll n$  we can compute  $K_n(x, y)$  efficiently.

### Proof that any algorithm to compute $K_n(x, y)$ must examine both input strings the entirely

$$\text{Let } x = 1^n, y = 1^{n+n!}$$

Consider a DFS (M) with  $n$  states, after input  $1^n$  we must be on a cycle by the pigeonhole principle. Let  $k$  ( $1 \leq k \leq n$ ) be the length of this cycle. Because  $k|n!$ ,  $\delta_M(1^n 1^{n!}) = \delta_M(1^n)$ . Consider a DFS whose cycle has length  $k > 1$ . Then  $\delta(1^n 1^{n!}) \neq \delta(1^n 1^{n!+1})$ , hence  $K_n(x, y) \neq K_n(x, 1^{n+n!+1})$   $\square$

Intuitively, this means that it is impossible for any DFA with  $n$  states to distinguish  $1^n$  from  $1^{n+n!}$ . Let  $P = \{1^n\}$  (Positive Set), and  $N = \{1^n 1^{n!}\}$  (Negative Set). The minimum consistent DFA must have at least  $n$  states to distinguish  $1^n$  from  $1^{n!}$ .

## Modified Problem Is as Hard as Factoring

Let  $A(n, x) = K_n(1^{n+x}, 1^n)$ ,

Claim: Computing  $A(n, x)$  is as hard as factoring

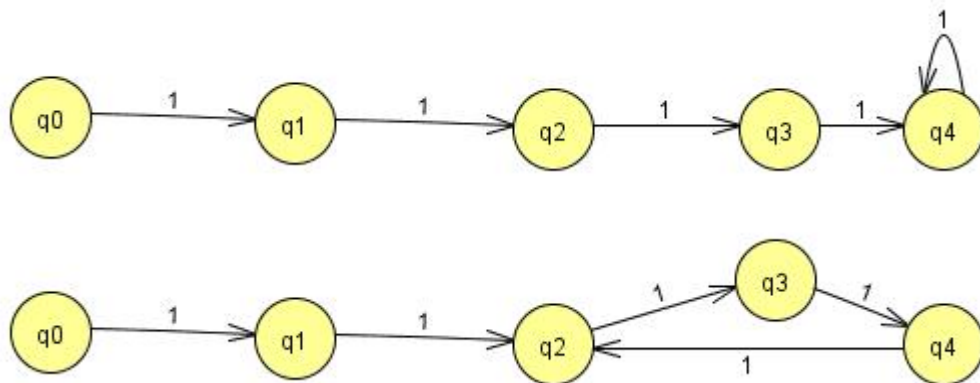
Cautionary Note: the size of the input to  $A$  is  $\log(n) + \log(x)$ , while in the original problem the length of the words is  $1^{n+x}, 1^n$ . Therefore, proving that  $A(n, x)$  is as hard as factoring does not imply that  $K_n(1^{n+x}, 1^n)$  is as hard as factoring.

Let  $M \in \mathbb{N}$  be given and let  $n = \lceil \sqrt{M} \rceil$ .

Claim 1:

$$A(n, M) = A(n, 1) \leftrightarrow K_n(1^{n+M}, 1^n) = K_n(1^{n+1}, 1^n) \leftrightarrow \forall j \in [2, \dots, n], \neg j|M$$

Proof:  $D_1, D_j \in \text{DFS}(n, \Sigma)$  be given such that the cycle on  $D_1$  has length 1 and the cycle on  $D_j$  has length  $j > 1$ . Examples ( $j = 3$ )



Note first that  $\forall x \in \mathbb{N}$ ,  $D_1$  does not distinguish  $1^{n+x}$  and  $1^n$ . Second, note that

$D_j$  does distinguish  $1^n, 1^{n+1}$ . Finally, note that  $\forall x \in \mathbb{N} D_j$  does not distinguish  $1^n$ , from  $1^{n+x} \leftrightarrow j|x$ .

←) If there is some  $j \in [2, \dots, n]$  such that  $j|m$  then  $\exists D_j \in DFS(n, \Sigma)$ , which distinguishes  $1^n, 1^{n+1}$  but does not distinguish  $1^n, 1^{n+M}$ . Hence,  $K_n(1^n, 1^{n+M}) > K_n(1^n, 1^{n+1})$ .

→) Suppose  $K_n(1^n, 1^{n+M}) = K_n(1^n, 1^{n+1})$ , then  $\forall j, D_j \in DFA(n, \Sigma)$  with cycle length  $j > 1, D_j$  must distinguish  $1^n, 1^{n+M} \rightarrow j$  does not divide  $M$ .  $\square$

Conclusion: Any efficient algorithm to compute  $A(n, x)$  would yield an efficient algorithm to factor  $M \in \mathbb{Z}$  by performing a binary search for the smallest factor.

## Proof that the corresponding Turing Machines Kernel is Not Computable

There is no reason to limit the kernel to DFA's. A universal kernel could also be used to learn context free languages, or even languages recognized by Turing Machines. Clearly, the corresponding kernel for Push-Down Automata is well defined and is computable by brute force, although it is not necessarily tractable. Is it possible that there is a computable kernel to learn any language recognized by a Turing Machines?

Given a fixed alphabet  $\Sigma$  and  $x, y \in \Sigma^*$ ,

Define:

$T^{(n)}(x, y) := \{ \langle H \rangle : H \text{ is a Turing Machine with } n \text{ states st. } H \text{ accepts } x \text{ and } y \}$

$K_n^{(TM)}(x, y) := |T^{(n)}(x, y)|$

Claim:  $K_n^{(TM)}(x, y)$  is NOT computable.

Proof: Suppose that there was some Turing Machine  $T_K$  to compute  $K_n^{(TM)}(x, y)$ . Then we could decide  $A_{TM}$  as follows:

$H =$  "On input  $\langle M \rangle, w$

1. Enumerate all Turing Machines with  $n = |M|$  states (there are finitely many such turing machines, and  $M$  is one of them)
2. Use  $T_k$  to compute  $K_n^{(TM)}(w, w)$
3. Set finishedCounter = 0
4. Run each enumerated machine on  $w$  in parallel (each one step at a time)
5. If  $M$  ever Accepts/Rejects then Accept/Reject
6. If some machine finishes (accepts or rejects) increment finishedCounter
7. If  $finishedCounter == K_n^{(TM)}(x, y)$  and  $M$  has not halted then Reject (There are only  $K_n^{(TM)}(w, w)$  turing machines with  $n$  states that accept  $w$  and hence

$M$  is not one of them). Note that  $K_n^{(TM)}(x, y)$  of the Turing Machines MUST accept  $w$  after finitely many steps.

This is a contradiction:  $A_{TM}$  is not decidable. Hence,  $K_n^{(TM)}(x, y)$  is not computable.

## Summary Of Results:

1. C++ code to compute  $K_n$  ( $\Sigma = \{0, 1\}$ ) for  $n \leq 6$  by brute force
2. Closed form expressions for  $K_n(x, y)$ ,  $\forall x, y \in \Sigma^*$ ,  $|x|, |y| \leq 2$
3. Algorithm to efficiently generate a closed form expression for  $K_n(x, y)$  given "small" words  $x, y \in \Sigma^*$  (the running time of the algorithm grows exponentially with  $|x|$  and  $|y|$  but is independent of  $n$ )
4. Proof that any algorithm  $A$  to compute  $K_n(x, y)$  must at least examine both words  $x, y$  completely. Hence,  $T(A) = \Omega(|x| + |y|)$
5. Proof that  $A(n, x)$ , a modified version of  $K_n$ , is as hard as factoring
6. Proof that the corresponding kernel for Turing Machines is not computable

## References:

1. Corinna Cortes, Leonid Kontorovich and Mehryar Mohri  
"A Universal Kernel for Learning Regular Languages"  
(paper in progress)
2. "Kernel Methods for Learning Languages"  
Talk by Leonid Kontorovich  
Joint Work with Corinna Cortes and Mehryar Mohri