Separating Three Words by Finite Automata

Fulai Lu, Kaiyang Teng, Nicolas Tran

2024 summer

Abstract

The problem of distinguishing between binary strings using deterministic finite automata (DFA) has been extensively studied for two words. However, the extension to three words introduces additional complexities. In this paper, we explore the problem of separating three distinct binary strings using the smallest possible DFA. We present new findings, including an upper bound for strings of different lengths and a construction method for strings of equal length, offering practical insights into the three-word separation problem.

1 Introduction

The word separation problem traditionally focuses on distinguishing between two binary strings using a minimal DFA. When extended to three words, the problem becomes more intricate as the DFA must distinguish each pair of strings uniquely. In this paper, we analyze the conditions under which a DFA can effectively separate three distinct binary strings and provide bounds on the number of states required.

2 Upper Bound for Different Lengths

In the study of DFA, a fundamental problem is determining the minimum number of states required to distinguish between multiple binary strings. When the strings have distinct lengths, the problem can be approached by leveraging properties of prime numbers and logarithmic functions to establish an upper bound on the size of the DFA. Given three binary strings x, y, and z where the lengths of the strings are different ($|x| \neq |y| \neq |z|$), the challenge is to design a DFA that can separate these strings—meaning that the DFA must transition through states such that each string ends up in a unique final state. The question is: how many states are necessary to guarantee that the DFA can distinguish between these three strings based on their lengths?

Proposition 1. Let x, y, z be binary strings whose lengths are all different. There exists a DFA with $13.2 \ln(\max(|x|, |y|, |z|))$ to separate x, y, and z.

Proof. Recall that if $n > m$ are integers, then there is a prime $p \leq 4.4 \ln n$ such that $n \not\equiv m \pmod{p}$. Now suppose by contradiction that for every prime $p \leq 13.2 \ln(\max(|x|, |y|, |z|))$:

1. either $|x| \equiv |y| \pmod{p}$; or
2. $|x| \equiv |z| \pmod{p}$; or
3. $|y| \equiv |z| \pmod{p}$.

By the pigeonhole principle, there must be at least $4.4 \ln \max(|x|, |y|, |z|)$ primes that satisfy one of the above cases, so either $|x| = |y|$ or $|x| = |z|$ or $|y| = |z|$, contradicting the hypothesis. □

3 Separation of Equal-Length Strings

When x, y, and z have the same length, the problem requires a different approach. We propose a method to construct a DFA with a minimal number of states that can separate three strings of equal length.

Proposition 2. Let x, y, z be distinct binary strings of the same length $n > 2$. There exists a DFA with $n + 1$ states to separate x, y, and z.

Proof. Let $c_{x,y}$ be the longest common prefix of x and y, and $c_{y,z}$ be the longest common prefix of y and z. Because there are only two symbols in the alphabet, $c_{x,y}$ and $c_{y,z}$ cannot have the same length, or else one of them is not the longest common prefix. Without loss of generality, let $n > \alpha = |c_{x,y}| > |c_{y,z}| = \beta$.
Define DFA $M = \langle \{0, 1, 2, \ldots, n, n+1\}, \{a, b\}, 0, \delta \rangle$, where

$$
\begin{aligned}
&1.\delta(0, x_0) = 1, \delta(1, x_1) = 2, \ldots, \delta(n-2, x_{n-2}) = n-1, \delta(n-1, x_{n-1}) = 0,\\
&2.\delta(\alpha, y_\alpha) = \alpha,\\
&3.\delta(\beta, z_\beta) = n+1,\\
&4.\delta(n+1, a) = \delta(n+1, b) = n+1,\\
&5.\delta(\cdot, \cdot) = n-1 \quad \text{for the remaining transitions.}
\end{aligned}
$$

M is well-defined, and it is clear that $\delta(0, x) = 0$, $\delta(0, z) = n + 1$, and $\delta(0, y) \notin \{0, n+1\}$. Hence, M separates x, y, z with $n + 1$ states. □

4 Mod DFA in Separating Words with Exactly One '1'

In the context of DFA, distinguishing binary strings based on specific patterns is crucial for various applications. One particular scenario involves distinguishing binary strings that contain exactly one '1'. This case is interesting because it limits the number of transitions and states needed, given that each string can be

differentiated by the position of the single '1' relative to the other symbols, which are all '0's. The key idea is to map the positions of the '1' in each binary string to distinct remainders when divided by a certain modulus m. This modulus is carefully chosen to ensure that each unique binary string ends up in a different state within the DFA.

Proposition 3. For any combination of three unique binary strings, each of length $n \geq 3$ and containing exactly one '1', the mod DFA will always process these strings to different final states.

Proof: We can distinguish these strings by constructing a DFA based on modular arithmetic. Let the three binary strings have lengths n with a, b, and c representing the number of '0's before the '1' in the respective strings. When the DFA reads '1', it returns to the start state and continues counting the number of '0's that follow.

Assume the DFA has m states and is constructed such that $a \mod m \neq b \mod m \neq c \mod m$. Then the equation still holds for $n-a \mod m \neq n-b \mod m \neq n-c \mod m$. Here, $n-a$, $n-b$, and $n-c$ are the number of 0s after 1 in the string. Due to the existence of the loop in the DFA and the different results of the modulus, the three strings will end up in different states. □

Proposition 4. For any arbitrary number k of '0's added after the three binary strings, the same DFA will still process them to different final states.

Proof:

- **Base Case ($k = 0$):** This was proven in Part 1 where we showed that the DFA distinguishes the three original strings (without any additional '0's).
- **Inductive Hypothesis:** Assume that for $k = x$, the DFA correctly distinguishes the three binary strings each with n additional '0's appended. Equation: $n-a+x \mod m \neq n-b+x \mod m \neq n-c+x \mod m$.
- **Inductive Step ($k = x+1$):** We need to prove that the DFA will also distinguish the three strings if one more '0' is added, resulting in $x+1$ additional '0's. After processing the extra '0', these states transition to: $n-a+(n+1) \mod m \neq n-b+(n+1) \mod m \neq n-c+(n+1) \mod m$. The equation ensures that these results are distinct because the modular arithmetic maintains the distinctness of these values, and the DFA has been designed to map different mod results to different states. Hence, the three strings will end up in different final states even after adding one more '0'. □

5. Experimental Results

The experimental results on separating three binary strings using deterministic finite automata (DFA) show that as string length increases, the DFA generally requires more states. For shorter strings (up to length 4), only 3 states are needed, but the complexity increases with string length, as evidenced by the growing number of worst-case tuples. From length 5 to 7, 4 states are sufficient, though the number of worst-case tuples rises significantly. At length 8, the DFA requires 5 states, a need that persists through length 14, with the worst-case tuples fluctuating and peaking at length 12. For lengths 15 and 16, 6 states are needed, and from length 17 onward, 7 states are required, with the number of worst-case tuples peaking at lengths 21 and 23. These results highlight the increasing complexity and state requirements as string length grows.

String length	**Worst Case**	**Total worst case count**
N=2	3 states	4 tuples total
N=3	3 states	56 tuples total
N=4	3 states	560 tuples total
N=5	4 states	4 tuples total
N=6	4 states	160 tuples total
N=7	4 states	1914 tuples total
N=8	5 states	4 tuples total
N=9	5 states	20 tuples total
N=10	5 states	96 tuples total
N=11	5 states	398 tuples total
N=12	5 states	1490 tuples total
N=13	5 states	47 tuples total
N=14	5 states	65 tuples total
N=15	6 states	2 tuples total
N=16	6 states	6 tuples total
N=17	7 states	2 tuples total
N=19	7 states	8 tuples total
N=21	7 states	23 tuples total
N=23	7 states	23 tuples total

6 Code Appendix

6.1 DFA.h

```
#pragma once
#include <vector>
#include <string>
#include <cassert>
#include <iostream>

//Represent a single state in DFA
struct state
{
        //Trasition from current state on input 0
        size_t t_0 = 0;

        //Trasition from current state on input 1
        size_t t_1 = 0;

};


class DFA
{

public:

        //Constructors

        //Default Constructor
        DFA()
        {
                size = 0;
                init_state_x = 0;
                init_state_y = 0;
                init_state_z = 0;
                final_state_x = 0;
                final_state_y = 0;
                final_state_z = 0;
                states = std::vector<state>();
        }

        //Constructor that takes in the state number of the DFA
        DFA(size_t n)
        {
                size = n;
                init_state_x = 0;
                init_state_y = 0;
                init_state_z = 0;
                final_state_x = 0;
                final_state_y = 0;
                final_state_z = 0;
                for (int i = 0; i < n; i++)
                {
                        states.push_back(state());
                }
        }
```

```
DFA(const DFA& D)
{
        size = D.size;
        init_state_x = D.init_state_x;
        init_state_y = D.init_state_y;
        init_state_z = D.init_state_z;
        final_state_x = D.final_state_x;
        final_state_y = D.final_state_y;
        final_state_z = D.final_state_z;
        for (int i = 0; i < D.states.size(); i++)
        {
                states.push_back(D.states[i]);
        }
}

//Accesors

//Seperation
//For a common start state
//M ends in different states after reading in x, y and z
bool seperation(const std::string& x, const std::string& y,
    red↪  const std::string& z)
{
        assert(x.size() == y.size() && y.size() == z.size()
            red↪ );

        if (x == y || y == z || x == z)
        {
                return false;
        }

        final_state_x = init_state_x;
        final_state_y = init_state_y;
        final_state_z = init_state_z;
        for (int i = 0; i < x.size(); i++)
        {
                final_state_x = x[i] == '0' ? states[
                    red↪ final_state_x].t_0 : states[
                    red↪ final_state_x].t_1;
                final_state_y = y[i] == '0' ? states[
                    red↪ final_state_y].t_0 : states[
                    red↪ final_state_y].t_1;
                final_state_z = z[i] == '0' ? states[
                    red↪ final_state_z].t_0 : states[
                    red↪ final_state_z].t_1;
        }

        return (final_state_x != final_state_y) && (
            red↪ final_state_y != final_state_z)
        && (final_state_x != final_state_z) ? true : false;
}
```

```
//E-Seperation
//For some common start state
//M ends in different states after reading in x, y and z
bool E_seperation(const std::string& x, const std::string&
    red↪ y, const std::string& z)
{
        bool flag = false;
        for (int i = 0; i < size; i++)
        {
                init_state_x = i;
                init_state_y = i;
                init_state_z = i;
                if (seperation(x, y, z))
                {
                        flag = true;
                }
        }

        return flag;
}

std::string get_DFA()
{
        std::string s = "";
        for (int i = 0; i < size; i++)
        {
                s += char(states[i].t_0 + '0');
                s += char(states[i].t_1 + '0');
        }
        return s;
}

//Modifier

//Only used for testing
//Add a new state to an DFA
void add_state(state n)
{
        states.push_back(n);
        size++;
}

//Increment the DFA
DFA& operator++(int)
{
        size_t carry = 0;
        states[0].t_0++;
        for (int i = 0; i < size; i++)
        {
                states[i].t_0 += carry;
                if (states[i].t_0 == size)
                {
                        states[i].t_0 = 0;
                        carry = 1;
                }
                else
```

```
                        {
                                carry = 0;
                        }

                        states[i].t_1 += carry;
                        if (states[i].t_1 == size)
                        {
                                states[i].t_1 = 0;
                                carry = 1;
                        }
                        else
                        {
                                carry = 0;
                        }
                }

                return *this;
        }

public:

        //Number of states
        size_t size;

        //Initial state for x
        size_t init_state_x;

        //Initial state for y
        size_t init_state_y;

        //Initial state for z
        size_t init_state_z;

        //Final state for x
        size_t final_state_x;

        //Final state for y
        size_t final_state_y;

        //Final state for z
        size_t final_state_z;

        //A set of states
        std::vector<state> states;

};
```

6.2 main.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <string>
#include <unordered_map>
#include <map>
#include "DFA.h"
```

```
using namespace std;

DFA find_result
(const string& str1, const string& str2, const string& str3, size_t
    red↪ & val)
{
        size_t n = 3;
        while (true)
        {
                DFA m(n);
                for (int i = 0; i < pow(n, 2 * n) - 1; i++)
                {
                        if (m.E_seperation(str1, str2, str3))
                        {
                                val = i;
                                return m;
                        }
                        m++;
                }
                n++;
        }
}


void str_generator
(int len,int target,std::string&path,std::vector<std::string>& ans)
{
    if(len==target)
    {
        ans.push_back(path);
        return;
    }
    path+='0';
    str_generator(1+len,target,path,ans);
    path.pop_back();
    path+='1';
    str_generator(1+len,target,path,ans);
    path.pop_back();
}

void str_generator_exactly_N_one
(int len,int count,int target,std::string&path,std::vector<std::
    red↪ string>& ans)
{
    if(len==target)
    {
        if(count==0)ans.push_back(path);
        return;
    }
    path+='0';
    str_generator_exactly_N_one(1+len,count,target,path,ans);
    path.pop_back();
    if(count!=0)
        {
                path+='1';
                str_generator_exactly_N_one(1+len,count-1,target,
                    red↪ path,ans);
```

```
                path.pop_back();
        }
}

int get0num(string&s)
{
        int sum=0;
        int i=0;
        while(s[i++]!='1') sum++;
        return sum;
}

int main()
{
        int str_len=14;
        DFA m;
        size_t val;
        std::ofstream ofs;
        ofs.open("N"+to_string(str_len)+"_Result.txt");
        vector<string> N;
        std::string path="";
        //str_generator(0,str_len,path,N);
        str_generator_exactly_N_one(0,1,str_len,path,N);
        int maxstate=-1;

        ofs << "str1" << '\t' << '\t' << "str2" << '\t' << '\t' <<
            red↪ "str3" << '\t' << '\t'
        << "val1" << '\t' << "val2" << '\t' << "val3" << '\t' << "
            red↪ dist" << '\t' << "result"
        << '\t' << '\t' << "val" << std::endl;
        int count=0;
        int countof5=0;

        unordered_map<std::string,int> worstcase_counter;

        for (int i = 0; i < N.size(); i++)
        {
                for (int j = i + 1; j < N.size(); j++)
                {
                        for (int k = j + 1; k < N.size(); k++)
                        {
                                ofs << N[i] << '\t' << N[j] << '\t'
                                    red↪  << N[k] << '\t' << i
                                << '\t' << j << '\t' << k << '\t'
                                    red↪ << m.size << '\t';
                                for (int l = m.states.size() - 1; l
                                    red↪  >= 0; l--)
                                {
                                        ofs << m.states[l].
                                            red↪ t_1 << m.
                                            red↪ states[l].
                                            red↪ t_0;
                                }
                                ofs << '\t' << '\t' << val << std::
                                    red↪ endl;
                        }
```

```
                }
        }
        return 0;
}
```