Shortest Synchronizing Sequences in Finite Automatas

Engincan Varan: evaran@sabanciuniv.edu Computer Science / FENS, 2020

Nilay İrem Güçin: ngucin@sabanciuniv.edu Computer Science

Hüsnü Yenigün: Computer Science

Kamer Kaya: Computer Science

Abstract:

We created hundreds of automata and their shortest sequences. We checked if any of the pieces of the shortest sequence can be generated by the Greedy algorithm. At the end, it turns out that relatively small pieces can be generated by the Greedy. However our research is still going on.

Introduction

Starting this project, our goal was to find a hybrid model to find the shortest sequence of any given automaton. A shortest sequence is described as a sequence of strings which transitions the set of states that the automaton has, to 1 from the original number. Finding the shortest sequence is an important task of modern life however there is no efficient way of doing this. To find the real shortest sequence takes a lot of time because it actually requires to check every single transition at every single state and the time complexity increases exponentially as the number of states increases linearly. Other methods, which are called *heuristics* due to their practicality, use different techniques. However they do not guarantee to return the shortest sequence.

Considering what makes the heuristics practical and time efficient, we planned to divide the real the shortest sequence into pieces and decide which piece can be generated by one of the heuristics.

Shortest Sequence Generators:

Since our aim is to find a correlation between these subsequences and the algorithms provided, first we wanted to have a code that generates the correct synchronizing sequence. Therefore, we used a ready code that generates a randomized automaton. This ready code was written by our supervisors earlier and uses brute force to search the shortest synchronizing sequence. After we have the synchronizing sequences, we write a code to divide the sequence into subsequences input by input and track the number of states in order to compare the algorithms. Then we are ready to compare and track how different algorithms behave for that particular subsequence. However, in these reports we were only able to analyze the greedy algorithm and how it behaves in the subsequences we provided. Real shortest sequence is generated with a code using branching which will be elaborated in the next section.

Randomized Automata Generator:

We had a pre-ready code that generates a finite complete automaton with the given inputs with given inputs such as number of states, alphabet size and randomized seed etc…. For example when we run the code with the inputs 82,2,1 (82 as the number of states, 2 as the alphabet size, 1 for the random seed), we should end up with a complete finite automata, and the resulted text file should look like this:

```
Automata 82 2 1 ---------------------
letter a:  11   2    8    12   7    16   0    14   9    0    18   4
0          5    5    19   15   5    34   35   27   3    33   10   40
29         22   21   12   21   39   30   44   45   6    9    47   48
49         8    42   21   15   50   53   10   1    2    9    10   17
56         18   54   41   29   59   60   61   0    4    24   10   19
```

```
59         22         14         11         10         3          77         9          29         37         68         73         39
9          55         66         7          51         73         29         68         56         35         44
letter b:  1          3          6          13         2          10         1          23         4          20         17         28
7          26         31         25         32         24         22         36         37         38         9          14         15
0          19         43         4          30         41         24         3          46         1          27         8          6
25         13         52         16         29         42         55         17         18         4          51         13         57
39         20         51         33         58         26         54         26         50         53         62         56         25
54         29         5          70         36         12         12         44         36         4          34         62         53
81         17         47         71         16         69         19         43         9          40         10
```

Here we can observe, we have 2 lists of numbers, since we entered 2 as our input size. Each index denotes the state we are in, and the value inside the index means that with the given input, indices will move on to that state. For instance, let's take a look at the bolded and underlined 11. Since we are in list "letter a" this means that we are taking the input "a". Index is 0 which means we are state 0. So, when we apply "a" to this state, our next state will be 11. This goes on like this.

Brute Force Algorithm to Find the Shortest Synchronizing Sequence :

Now we have the complete finite automaton, we need to have a working code that always produces the shortest synchronizing sequence in order to have a reliable controlled sequence. We used a brute force algorithm to find this shortest sequence. This algorithm has an exponential time complexity so it produces correct results but it takes a long time when the input size is increased. However, since we are trying to compare the algorithms, we need to have a controlled group. The algorithm nearly tries every possible sequence to find the shortest sequence. In the end it generates the shortest sequence for us. For example, if we run the code with the inputs 156, 2, 1 (the seed is important in our case), the shortest synchronizing sequence would be, "011111100110101100111010110" where 0 denotes "a" and 1 denotes "b".

Sequence Tracking

Our project actually starts here. All the codes and explanations provided above were actually written by our supervisors.

So, now we have the shortest synchronizing sequence and the automata for it. We can start dividing this sequence into parts and analyze it. However, in order to analyze the sequences we needed the number of states for each subsequence. Therefore we write a code to generate them. An example output is shown below:

```
(14) : 2 3 4 18 22 23 25 28 32 33 35 38 78 100
the sequence necessary to go from 14 states to 12 states: 1
(12) : 0 6 8 12 16 31 47 49 57 59 65 73
(12) : 4 10 14 15 17 27 28 34 49 55 85 97
the sequence necessary to go from 12 states to 9 states: 10
(9) : 2 7 12 14 18 19 26 80 106
```

```
(9) : 12 13 32 33 39 42 48 58 69
the sequence necessary to go from 9 states to 8 states: 01
(8) : 10 14 15 29 31 43 65 70
```

With this output, we now know how to generate the shortest synchronizing sequence and how the states are changing. We randomly created thousands of complete automatas and found the shortest sequence with the brute-force algorithm. Then, we divide each shortest sequence and states and check whether we are dividing it correctly by comparing them with each other. We have done this testing many times in order to check if we are dividing the states and inputs correctly and we have decided that our outputs are indeed correct.

Algorithm 1: Greedy's Behavior

Greedy algorithms are the first obvious solution to many problems. These algorithms always choose the most optimal solution locally in order to create a global optimum output. In our case, the greedy algorithm would always choose a sequence that merges more states than the other sequences. For example, if we have 15 states, the greedy algorithm tries every possible input combination (0,1, 00, 01, 10, 11, 000…) and chooses directly the one that decreases the number of states. So, let's assume inputs 0 and 1 do not decrease the number of states, and if 00 decreases it to 14, the greedy algorithm would choose this subsequence. However, choosing the local optimum does not always provide the optimal solution because sometimes choosing other options creates more optimal solutions later on.

So, we wanted to understand when to choose local optimum and when not to choose it. We implemented a code that mimics the behavior of the greedy algorithm and analyzes each subsequence with it. We compare the outputs of the greedy algorithm and brute-force algorithm for each subsequence we have. A friendly reminder that we analyze the subsequences of inputs that are bigger than 1. An example output that greedy algorithm could generate:

```
From (#2) states with input (110) we have the states: 4 6
To (#1) states with input (110) we have the states: 3
--- CHECK TIME ---
We have to check for these sequences: 0 1 00 10 01 11
    Trying for: 0
        With input (0) we have the states: 22 27
        Size Comparison:    We had (2), we can go for min (1), yet we have (2).
    Trying for: 1
        With input (1) we have the states: 14 28
        Size Comparison:    We had (2), we can go for min (1), yet we have (2).
    Trying for: 00
        With input (0) we have the states: 22 27
        Size Comparison:    We had (2), we can go for min (1), yet we have (2).
        With input (0) we have the states: 9 44
        Size Comparison:    We had (2), we can go for min (1), yet we have (2).
    Trying for: 10
        With input (1) we have the states: 14 28
        Size Comparison:      We had (2), we can go for min (1), yet we have (2).
        With input (0) we have the states: 0 7
        Size Comparison:      We had (2), we can go for min (1), yet we have (2).
    Trying for: 01
        With input (0) we have the states: 22 27
        Size Comparison:      We had (2), we can go for min (1), yet we have (2).
        With input (1) we have the states: 0 4
        Size Comparison:      We had (2), we can go for min (1), yet we have (2).
    Trying for: 11
        With input (1) we have the states: 14 28
        Size Comparison:      We had (2), we can go for min (1), yet we have (2).
        With input (1) we have the states: 5 20
        Size Comparison:      We had (2), we can go for min (1), yet we have (2).
```

Here we can observe that the greedy algorithm could generate this subsequence, because there is no shorter input sequence that decreases the number of states. We have to use at least 3 inputs to decrease the number of states, in our it was 110. We mimic the greedy algorithm here and try all possible combinations.

For another example that greedy cannot generate:

```
From (#32) states with input (10) we have the states: 0 5 6 7 8 10 14 15 19 21 22 23 24 27 29 30 31 40 45 48 53 64 68 72 74 82 90 92 93 97 106
110
To (#26) states with input (10) we have the states: 0 2 3 4 7 9 10 12 14 18 19 21 27 30 40 45 54 68 73 74 77 85 90 92 110 111
--- CHECK TIME ---
We have to check for these sequences: 0 1
    Trying for: 0
        With input (0) we have the states: 0 2 3 5 6 8 9 13 15 18 19 20 24 25 27 31 33 43 45 47 51 61 62 68 93 94 96 103
        Size Comparison:      We had (32), we can go for min (26), yet we have (28).

!!!GREEDY CANNOT DO THIS!!!
```

Here we can observe that, when we mimic the greedy algorithm, it will choose the input "0" only since it already decreases the number of states and it seems like the local optimum. However, the brute-force algorithm chooses "10" as the input sequence and decreases the number of states much more than the greedy algorithm.

We analyzed this on many different automata and sequences in order to have an idea about how the greedy algorithm behaves and when to behave like the greedy algorithm.

Discussions & Conclusions:

During our mission to find a hybrid algorithm, we discovered new methods to find the actual shortest sequence. Dividing the problem and later on attaching the results is a good method for our goal and we practiced it throughout the semester. However we didn't come to a point where we can show polished results as in this research is still going on. While working on algorithms and automata we also learned alternate ways to increase efficiency and/ or accuracy. In computer science it's about the balance and this project taught us to capture this balance.

In terms of research, we found out that most of the pieces in the shortest sequence can be generated by the greedy algorithm. In fact in rare cases all of the sequence can be generated by the greedy. Shorter the sequence is more likely that the greedy can find it.

From where we're standing, our research seems hopeful. We have to include other heuristics to the picture as we compare the pieces and find new ways to blend these algorithms. In addition, we should perfecten the algorithm for dividing the sequence into pieces and find new ways to categorize pieces in terms of which can be generated by which algorithm.

References:

1. Adam Roman, Synchronizing finite automata with short reset words, Applied Mathematics and Computation 209 (2009), no. 1, 125 – 136, Special Issue International Conference on Computational Methods in Sciences and Engineering 2005 (ICCMSE-2005).