Multi-threaded Automata Generation and Testing CLI Application

Mitchell Jacobs
SUNY Polytechnic Institute

May 2021

1 Abstract

This paper presents an Automata generation and testing CLI application. The algorithm for generating Automata is provided, along with an adaptation of the algorithm for testing Automata[Rus21]. The CLI application is shown to scale linearly.

2 Introduction

The purpose of this paper is to extend the Automata generation and testing algorithms[Rus21]. The Automata processed by this algorithm are assumed to be finite, deterministic, and possibly weighted. Due to the theorem proven by[Ric07; Epp90], that every non-deterministic finite Automata can be represented as a deterministic finite Automata, the paper's scope can be reduced to only deterministic finite Automata.

A weighted deterministic finite Automata M is defined as following.

$$\begin{aligned}
&M = (Q, \Sigma, \delta, s, A, \gamma)\\
&Q\ is\ a\ finite\ set\ of\ states\\
&\Sigma\ is\ the\ finite\ input\ alphabet\\
&\delta : Q * K \rightarrow Q\\
&s \in Q\ is\ the\ starting\ state\\
&A \subseteq Q\ is\ the\ set\ of\ accepting\ states\\
&\gamma : \Sigma \rightarrow \mathbb{N}
\end{aligned}$$

The input to a deterministic finite Automata is any regular language L. L contains synchronizing and non-synchronizing words. A Synchronizing word is defined as a $w \in L \subseteq \{word : DFA(word) \rightarrow True\}$. Where *DFA(x)* is a deterministic finite Automata that takes an input word x and returns *True* or *False*.

The goal of this application is to use heuristics to prune the regular language L so as to return only one synchronizing word.

3 Generating Automata

3.1 Automata Format

In order to generate Automata we must first choose a format to store them in. CSV and XML filetypes are not well suited for this. As a result, the Generate Automata algorithm uses a custom format.

$$[weight_0, ..., weight_N] : [state_0 0, ..., state_0 M]|[state_N 0, ..., state_N M] \tag{1}$$

Automata in this format can have up to N possible symbols and up to M states per symbol.

3.2 Generate Automata

To run tests using the application it is necessary to generate Automata. It is possible to provide the application with an Automaton through the command line. Additionally, it is possible to batch test Automata using a file.

A Python implementation generating *automata* random Automata is as follows.

```python
for i in range(automata):
    states = random.randint(3,maxStates)
    symbols = random.randint(2,maxSymbols)
    weights = '['+ ','.join([str(random.randint(1,maxSymbols)) for i in range(symbols)]) + ']'
    transition = '[' + '|'.join(['[' + ','.join([str(random.randint(0,maxStates)) for i in range(states)]) + ']' for j in range(symbols)]) + ']'
    line = ':'.join([weights,transition])+'\n'
```

Each *line* generated from this for loop will contain one random Automata. The Automata generated from this loop are not guaranteed to be valid. To determine validity, the Automata must be tested.

4 Testing Automata

Testing Automata for validity involves determining wether the Automata contains a path from the starting state to the ending state. This is determined by the *testAutomata* function.

$$testAutomata \rightarrow automata \subseteq \{A : A \in \{\epsilon, Automaton\}\} \tag{2}$$

The Python implementation is as follows.

```python
tests = list()
if (inputFile is not None and inputFile.exists()):
    with mp.Pool(processes=amtWorkers) as pool:
        tests = pool.map(addAutomaton, getAutomataFromFile(inputFile))
```

pool.map maps the *addAutomaton* function with an Automata from *getAutomataFromFile* to each worker in the pool.

4.1 Computing Potential Words

Once the validity of a given Automaton has been determined, testing can begin. First, all the potential words in the language L that the Automaton accepts must be computed.

The Python implementation for this is as follows.

```python
1  def compute_shortest_words(self,starting_vertex:int) -> dict:
2  
3      #Check if we have already calculated for this starting_vertex
4      if (starting_vertex in self._shortestWords):
5          return self._shortestWords[starting_vertex]
6  
7      #We haven't already calculated shortest words
8      distances = self._maxGraphDistances
9      distances[starting_vertex] = (0,"")
10 
11     pq = [(0, starting_vertex, "")]
12     while len(pq) > 0:
13         current_distance, current_vertex, current_word = heapq.heappop(pq)
14 
15         if current_distance > distances[current_vertex][0]:
16             continue
17 
18         for neighbor, (weight,symbol) in self._graph[current_vertex].items():
19             distance = current_distance + weight
20 
21             if distance < distances[neighbor][0]:
22                 distances[neighbor] = (distance, symbol + current_word)
23                 heapq.heappush(pq, (distance, neighbor, symbol + current_word))
24 
25     # Store a copy of the shortest words from a given distance so that we can preform a look up for later calls
26     self._shortestWords[starting_vertex] = distances
27     return distances
```

Lines 8-23 are adapted[Rus21].

4.2 Heuristics

After calculating the potential words in the language L accepted by the given Automaton, it is necessary to begin pruning our language. Each heuristic in *testHeuristics* utilizes a different heuristic for pruning. Additionally, the shortest word is computed and stored in *shortest*. The heuristics used in *testHeuristics* are explained in [Rus21].

```python
def testHueristics(automaton:Automata) -> tuple:
    h1 = automaton.approximate_weighted_synch(4,automaton.t1,
    automaton.h1)
    h2 = automaton.approximate_weighted_synch(4,automaton.t1,
    automaton.h2)
    h3 = automaton.approximate_weighted_synch(4,automaton.t3,
    automaton.h3)
    h4 = automaton.approximate_weighted_synch(4,automaton.t1,
    automaton.h4)
    shortest = automaton.compute_shortest_word()
    return (h1,h2,h3,h4,shortest)
```

approimate_weighted_synch as implemented in Python below is adapted[Rus21].

```python
def approximate_weighted_synch(self,m:int, t, H):
    # Check if we should time
    if (self._shouldTime):
        startTimeAWS = time.time()

    # Set the power set
    self._power = m

    # Generate power automaton
    self._compute_automaton_m()
    aut = (self.powerTransitionFunction,self.keyToStateDict)

    # Run graph so that our graph has the values for the power
    automaton
    self.graph(aut[0])

    # Get dummy longest words
    shortest_words_with_inf = self.compute_shortest_words(0)

    shortest_words = dict()
    for i in range(1,len(self._transitionFunction[0])):

        # Get the shortest words from the starting state i
        tmp_words = self.compute_shortest_words(i)
        for el in tmp_words:
            if shortest_words_with_inf[el][0] > tmp_words[el][0]:
                shortest_words_with_inf[el] = tmp_words[el]

    # Copy over our computed shortest words
    for el in shortest_words_with_inf:
        if shortest_words_with_inf[el][0] != float('infinity'):
            shortest_words[el] = shortest_words_with_inf[el]

    # The total states in our automaton
    T = tuple(range(len(self._transitionFunction[0])))
    u = ""
    while len(T) > 1:
        current_shortest = float('infinity')
        w = ""
        for el in shortest_words:

            # The key represenation of the temp word
            P = set(aut[1][el])

            if t(P, T, m):
                tmp = H(P,T,shortest_words[el])

                # Check if the new weight is smaller
                if tmp < current_shortest:
                    current_shortest = tmp
                    w = shortest_words[el][1]

        # Check if we found any synchronizing words
        if w == "":
            return None

        # Add our new word and reduce the remaining states
        u = u + w
        T = self.compute_image_by_word(T, w)

    if (self._shouldTime):
        self._totalTests += 1
        endTimeAWS = time.time()
    return Result(word=u, wordWeight=self.word_weight(u),time=
    endTimeAWS - startTimeAWS,hCount=self._totalTests)
```

5 Optimization

The application provided is written using the multiprocessing module in Python. The implementation of this module allows for a significant increase in testing capabilities. As follows is the comparison of the application's execution time with vs. without the multiprocessing module.

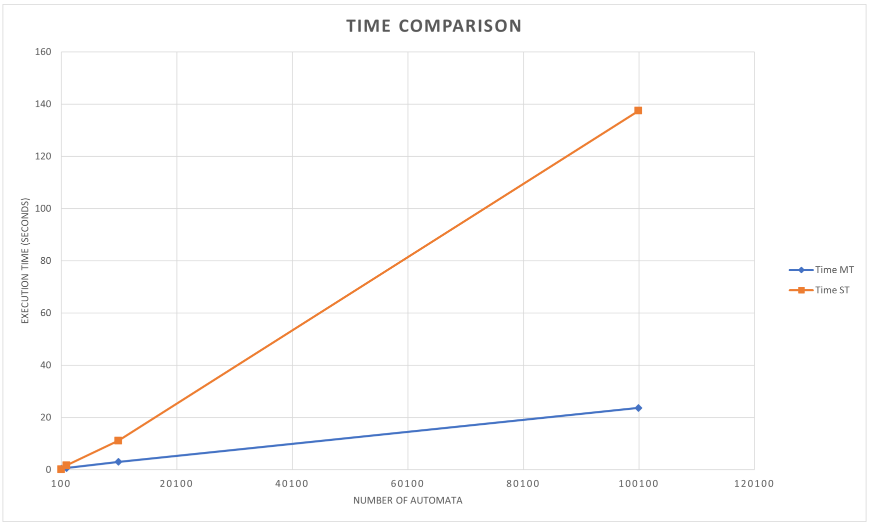

Figure 1: Time comparison

Time MT is the timing with the multiprocessing module and Time ST is the time without. The Python implementation of *testHeuristics* using the multiprocessing module is as follows.

```
e = list()
for x in range(len(tests)):
    if (tests[x] is not None):
        e.append(tests[x])
tests = e

heuristics = list()
with mp.Pool(processes=amtWorkers) as pool:
    heuristics = pool.map(testHueristics, tests)

e = list()
for x in range(len(tests)):
    if (None not in heuristics[x]):
        e.append((tests[x],heuristics[x]))
heuristics = e
```

To determine the time complexity of the Automata testing application, the application was tested on different numbers of Automata. As follows is a logrithmic graph of the Automata timing results.

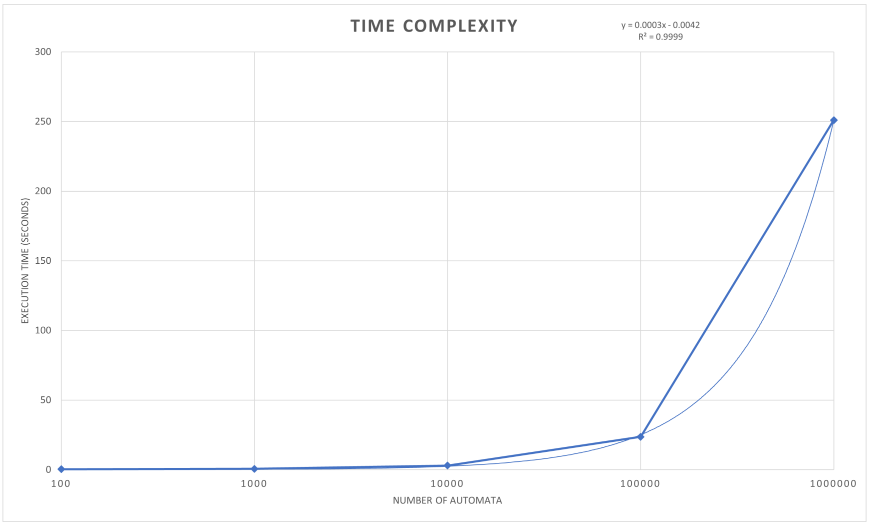

Figure 2: Time complexity

6 Conclusion

The application described in this paper sufficiently generates and tests Automata. Additionally, the application has linear time complexity allowing for significant testing capabilities. A possible improvement to the application would be implementing NVIDIA's CUDA or other GPU acceleration modules.

The full implementation is hosted on GitHub.

References

[Epp90] D. Eppstein. "Reset Sequences for Monotonic Automata". In: *SIAM J. Comput.* 19 (1990), pp. 500–510.

[Ric07] Elaine Rich. *Automata, Computability and Complexity: Theory and Applications*. Pearson, 2007. ISBN: 9780198520115.

[Rus21] Jakub Ruszil. "Approximation algorithm for finding short synchronizing words in weighted automata". In: *CoRR* abs/2103.16185 (2021). arXiv: 2103.16185. URL: https://arxiv.org/abs/2103.16185.