An Implementation of Product Construction on Deterministic Finite Automata in Python

Evan Childers
University of Alabama
Tuscaloosa, Alabama, United States
epchilders@crimson.ua.edu

August Connors
University of Alabama
Tuscaloosa, Alabama, United States
ahconnors@crimson.ua.edu

Kai NeSmith
University of Alabama
Tuscaloosa, Alabama, United States
klnesmith@crimson.ua.edu

***Abstract*—In this paper, we discuss the theory and implementation of product construction between two deterministic finite automata (DFAs). In doing so, we discuss the creation of a DFA data structure, the product construction algorithm itself and methods of verifying the product logic through experimentation and visual mediums. We also discuss additional related topics including identifying potentially unreachable states in a resultant DFA as well as the distinction between the intersection and union operations and how they affect the accepting state set.**

I. Introduction

In automata theory, product construction describes the method by which two DFAs are combined into a single DFA to accept a language that is derived from a certain boolean operation on the languages of the original DFAs such as intersection and union. This method plays a pivotal role in many applications in computing theory and formal language processing. In particular, product construction provides a means for which relationships between regular languages can be explored. By constructing a product DFA, one can effectively evaluate whether two DFAs accept overlapping inputs, meet certain logical requirements, or satisfy a desired system behavior.

II. Problem Statement

While the process of DFA product construction is a well understood concept in automata theory, the manual computation of a product DFA can become extremely complex and error-prone as the size of the DFAs involved increase. Additionally, the intricacy of these product DFAs can often complicate the process of verifying logical operations like the intersection and union of different automata. This sparks the need for a programmatic solution capable of systematically identifying the product states and transitions, simulating the traversal of a DFA given an input string and presenting the resulting automaton in a clear and interpretable form. This work addresses the lack of available resources regarding this issue by detailing the implementation and formal proof for product construction in Python, along with its potential use cases in verifying logical expressions.

III. Implementation

The implementation of the product construction for two deterministic finite automata (DFAs) was developed in Python, utilizing a custom DFA class to encapsulate the automaton's components and facilitate the construction process. In the scope of this paper, the case of both input DFAs sharing an alphabet was considered for simplicity. The DFA class is initialized with a constructor that accepts five parameters: a list of states, a list of alphabet symbols, a dictionary of transitions mapping state-symbol pairs to destination states, an initial state, and a list of accepting states. This structured representation ensures efficient access to DFA components, simplifying the product construction and state reachability analysis. For the discussion of the product algorithm, the input DFAs will be represented by the tuples

$$D_1 = (Q_1, \Sigma_1, \delta_1, s_1, F_1)$$

$$D_2 = (Q_2, \Sigma_2, \delta_2, s_2, F_2)$$

and the product DFA represented by

$$D' = (Q', \Sigma', \delta', s', F')$$

where Q denotes the list of states, Σ the alphabet, δ the transition function, s the start state, and F the list of accepting states.

The product construction function is primarily based on the formal definition, particularly adapted from Sipser's interpretation for union [1]. The function takes two DFA objects as input and generates a new DFA representing either the intersection or union of their languages. In addition to the two DFAs, the function's input includes a flag indicating whether the intersection or union is desired. The function constructs the state set of the product DFA as the Cartesian product of the input DFAs' state lists, forming pairs (q_1, q_2) where $q_1 \in Q_1, q_2 \in Q_2$. The alphabet is inherited from the input DFAs, as both share the same alphabet. The transition function is defined using the input DFAs' transition dictionaries: for a state pair $(q_1, q_2) \in Q'$ and symbol a, the transition is to $(\delta_1[(q_1, a)], \delta_2[(q_2, a)])$. The initial state of the product DFA is the pair (s_1, s_2). The list of accepting states is the only attribute that differs in the product DFA based on the intersection/union flag. For an intersection, the accepting states are pairs (q_1, q_2) where $q_1 \in F_1$ and $q_2 \in F_2$. For a union, the accepting states are pairs (q_1, q_2) where $q_1 \in F_1$ or $q_2 \in F_2$. These components are then assembled into a new DFA object, leveraging the class's structured format. The psuedocode for this function is provided in Algorithm 1.

Algorithm 1 Product Construction of Two DFAs

```
$\Sigma' \leftarrow \Sigma_1$
$s' \leftarrow s_1 + "," + s_2$
for $q_1 \in Q_1$ do
    for $q_2 \in Q_2$ do
        $q' \leftarrow q_1 + "," + q_2$
        $Q'$.append($q'$)
        if intersection then
            if $q_1 \in F_1$ and $q_2 \in F_2$ then
                $F'$.append($q'$)
            end if
        end if
        if union then
            if $q_1 \in F_1$ or $q_2 \in F_2$ then
                $F'$.append($q'$)
            end if
        end if
        for $a \in \Sigma'$ do
            $q_t \leftarrow \delta_1[(q_1, a)] + "," + \delta_2[(q_2, a)]$
            $\delta'[(q', a)] \leftarrow q_t$
        end for
    end for
end for
return DFA($Q', \Sigma', \delta', s', F'$)
```

To identify unreachable states in the product DFA, a breadth-first search (BFS) algorithm was implemented. Starting from the initial state pair, the BFS explores reachable state pairs by iterating over the alphabet and querying the transition dictionary. A queue manages states to be visited, and a set tracks visited states to avoid cycles. If a state returned by the querying of the transition dictionary is not already in the visited states set, it is added to both this set and the queue. The BFS algorithm keeps searching until the queue is empty. At the end of the search, the BFS has compiled a list of reachable states, and any state pairs in the product DFA's state set not visited are identified as unreachable. This list of unreachable states is output alongside the product DFA, providing insight into the automaton's structure without modifying its state set.

This implementation benefits from the DFA class's clear organization, which streamlines the product construction and enables efficient BFS-based analysis, ensuring accurate computation of the DFA product and identification of unreachable states. Additional functions were used for testing and visualizing the DFAs involved in the product construction algorithm. Two functions of the DFA class were used to output a transition table and to output a graphical representation of the DFA. The function for the graphical representation utilized Python's automathon library [2] to simplify its implementation. These functions allow for a visual comparison between the input DFAs and their product. Further, a function which takes a string as input and outputs whether the string is accepted or rejected by a DFA was implemented in the DFA class. This function iterates through the characters of an input string and uses the transition dictionary to navigate through the states of the DFA, returning true if the final state is accepting and false if otherwise. This was utilized to verify the properties of the union and intersection of the languages recognized by the input DFAs.

IV. Experiments

In this section, we will explore how product construction can be applied to a variety of different DFAs with a focus on the differences between intersection and union as well as identifying the causes of unreachable states in the resulting automaton

A. Experiment 1

In this experiment, we started with two DFAs to product construct in order to test the basic capabilities of our implementation. The first DFA, whose transition table is shown in Table I and diagram shown in Fig. 1, accepts the language that contains strings of length at least two that have a 1 in their second position.

TABLE I
Experiment 1's DFA 1 Transition Table

	0	1
$\rightarrow a_0$	a_1	a_1
a_1	a_r	a_2
$*a_2$	a_2	a_2
a_r	a_r	a_r

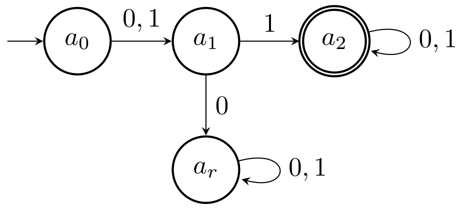

Fig. 1. Experiment 1's DFA 1

The second DFA, whose transition table is shown in Table II and diagram shown in Fig. 2, accepts the language that contains strings that contain at least one 0.

TABLE II
Experiment 1's DFA 2 Transition Table

	0	1
$\rightarrow b_0$	b_1	b_0
$*b_1$	b_1	b_1

When computing the product of these two DFAs, the resulting transition table is shown in Table III and diagram shown in Fig. 3.

The algorithmically determined unreachable states are (a_0, b_1) and (a_r, b_0). When analyzing these state pairs in the context of their original DFAs, we can understand why they

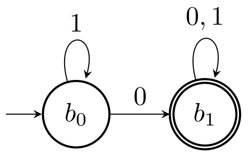
Fig. 2. Experiment 1's DFA 2

TABLE III
EXPERIMENT 1'S PRODUCT DFA TRANSITION TABLE

	0	1
$\to a_0, b_0$	a_1, b_1	a_1, b_0
a_0, b_1	a_1, b_1	a_1, b_1
a_1, b_0	a_r, b_1	a_2, b_0
a_1, b_1	a_r, b_1	a_2, b_1
a_2, b_0	a_2, b_1	a_2, b_0
$*a_2, b_1$	a_2, b_1	a_2, b_1
a_r, b_0	a_r, b_1	a_r, b_0
a_r, b_1	a_r, b_1	a_r, b_1

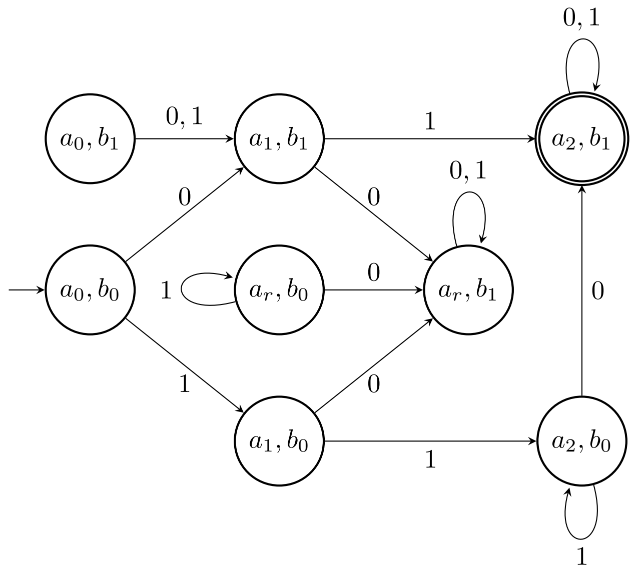
Fig. 3. Experiment 1's Product DFA

are unreachable in the product DFA. For the state pair (a_0, b_1), note that b_1 can only be reached from DFA 2's start state by a transition on input 0. However, in DFA 1, a_0 transitions to a different state on input 0 and does not allow any transitions that lead back to a_0. Therefore, there is no string that leads to the simultaneous presence of a_0 and b_1, rendering this state pair unreachable in the product. A similar rationale applies to (a_r, b_0). The state a_r in DFA 1 is reached via a transition on input 0, while b_0 in DFA 2 transitions away from itself on input 0 and does not return. Thus, there is no string in the input alphabet that can result in the system being in the state (a_r, b_0), making it unreachable as well.

In order to verify the intersection and union properties, we use the `accepts_string` method to test string acceptance in the product DFA. The results, seen in Table IV, demonstrate the logical operations between the original DFAs.

TABLE IV
BOOLEAN ANALYSIS OF EXPERIMENT 1'S PRODUCT DFA OPERATIONS

DFA Type	**11**	**00**	**01**	**1**
DFA 1	True	False	True	False
DFA 2	False	True	True	False
Union DFA	True	True	True	False
Intersection DFA	False	False	True	False

As shown in table, the union DFA accepts a string if either original DFA accepts it ($\text{DFA}_1 \cup \text{DFA}_2$), while the intersection DFA only accepts strings accepted by both ($\text{DFA}_1 \cap \text{DFA}_2$). The results confirm the correct implementation of these Boolean operations.

B. Experiment 2

In this experiment, we started with a pair of slightly larger DFAs in order to further test the capabilities of our product construction implementation. The first DFA, whose transition table is shown in Table V and diagram shown in Fig. 4, accepts the language that contains strings that have exactly two 0s.

TABLE V
EXPERIMENT 2'S DFA 1 TRANSITION TABLE

	0	1
$\to a_0$	a_1	a_0
a_1	a_2	a_1
$*a_2$	a_3	a_2
a_3	a_3	a_3

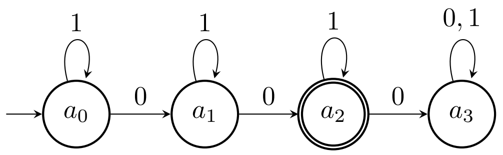
Fig. 4. Experiment 2's DFA 1

The second DFA, whose transition table is shown in Table VI and diagram shown in Fig. 5, accepts the language that contains strings that contains at least two 1s.

TABLE VI
EXPERIMENT 2'S DFA 2 TRANSITION TABLE

	0	1
$\to b_0$	b_0	b_1
b_1	b_1	b_2
$*b_2$	b_2	b_2

When computing the product of these two DFAs, the resulting transition table is shown in Table VII and diagram shown in Fig. 6.

Notably, this experiment differs from Experiment 1 in that it generates no unreachable states, due to the behavior of the

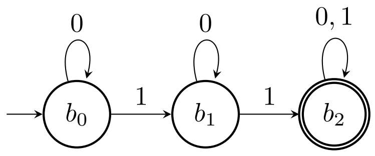

Fig. 5. Experiment 2's DFA 2

two separate DFAs. Since there are only transitions to different states on a 0 input in DFA 1, and only transitions to different states on a 1 input in DFA 2, the resulting product constructed DFA exhibits a lattice-like structure, while ensuring that no resultant states are unreachable (this essentially functions like a coordinate grid, meaning that you can reach any resultant state from the start state on an appropriate input since no transitions in the original DFAs preempt another transition).

TABLE VII
EXPERIMENT 2'S PRODUCT DFA TRANSITION TABLE

	0	1
$\rightarrow a_0, b_0$	a_1, b_0	a_0, b_1
a_0, b_1	a_1, b_1	a_0, b_2
a_0, b_2	a_1, b_2	a_0, b_2
a_1, b_0	a_2, b_0	a_1, b_1
a_1, b_1	a_2, b_1	a_1, b_2
a_1, b_2	a_2, b_2	a_1, b_2
a_2, b_0	a_3, b_0	a_2, b_1
a_2, b_1	a_3, b_1	a_2, b_2
$*a_2, b_2$	a_3, b_2	a_2, b_2
a_3, b_0	a_3, b_0	a_3, b_1
a_3, b_1	a_3, b_1	a_3, b_2
a_3, b_2	a_3, b_2	a_3, b_2

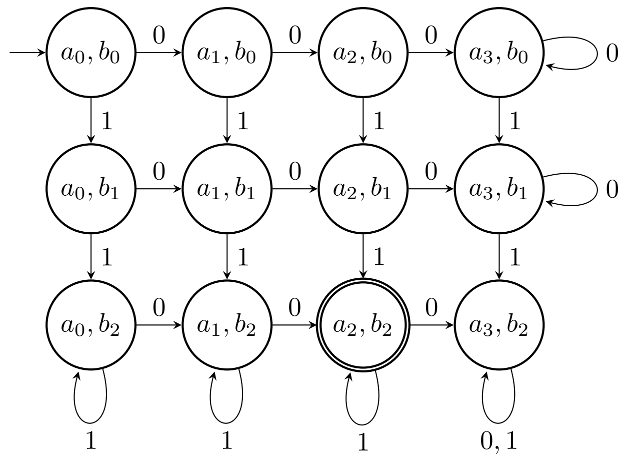

Fig. 6. Experiment 2's Product DFA

In order to verify the intersection and union properties, we use the `accepts_string` method to test string acceptance in the product DFA. The results, seen in Table VIII, demonstrate the logical operations between the original DFAs. As was the case in Experiment 1, the union and intersection DFAs act as expected for the respective operations, thereby supporting our confidence in the implementation.

TABLE VIII
BOOLEAN ANALYSIS OF EXPERIMENT 2'S PRODUCT DFA OPERATIONS

DFA Type	**000**	**00**	**11**	**0101**
DFA 1	False	True	False	True
DFA 2	False	False	True	True
Union DFA	False	True	True	True
Intersection DFA	False	False	False	True

C. Experiment 3

In this experiment, we started with two DFAs to product construct in order to test and verbosely demonstrate the difference between union and intersection product constructions. As discussed in the Implementation Details section, the only difference between union and intersection operations when performing product construction is the way the accepting states set is constructed. By creating a test to demonstrate this prominently, we would be able to effectively evaluate whether our process successfully constructed the accepting states set. The first DFA, whose transition table is shown in Table IX and diagram shown in Fig. 7, accepts all strings (it is purposefully very simple).

TABLE IX
EXPERIMENT 3'S DFA 1 TRANSITION TABLE

	0	1
$\rightarrow *q_0$	q_0	q_0

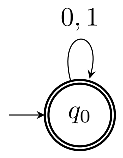

Fig. 7. Experiment 3's DFA 1

The second DFA, whose transition table is shown in Table X and diagram shown in Fig. 8, was designed simply for its complexity; we did not consider the language it accepts, as it is only meant to be dense.

TABLE X
EXPERIMENT 3'S DFA 2 TRANSITION TABLE

	0	1
$\rightarrow a_0$	a_7	a_9
a_1	a_5	a_4
a_2	a_8	a_2
a_3	a_5	a_6
a_4	a_6	a_5
a_5	a_{10}	a_7
$*a_6$	a_8	a_3
a_7	a_2	a_7
$*a_8$	a_0	a_{10}
a_9	a_4	a_1
a_{10}	a_{10}	a_2

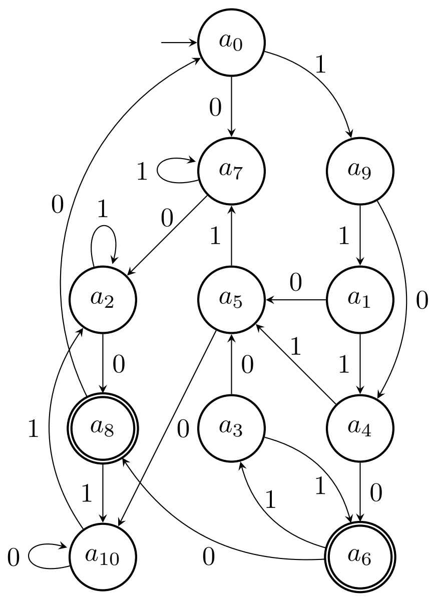

Fig. 8. Experiment 3's DFA 2

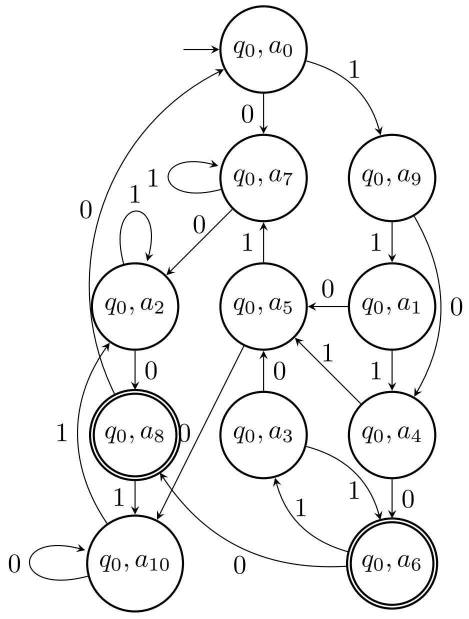

Fig. 9. Experiment 3's Intersection Product DFA

When computing the product of these two DFAs using the intersection method (which has been used in the preceding experiments), the resulting transition table is shown in Table XI and diagram shown in Fig. 9. Note that the transition table and diagram are essentially the same as the original DFA 2. Since DFA 1 is only one state, no "new" states are generated as a result of taking the Cartesian product of the two sets of states (all of DFA 2's states are preserved, just with q_0 prepended). Additionally, since we chose to use the intersection method, only state pairs that are both accepting are marked as accepting in the resulting DFA, meaning that DFA 2's original accepting states are also preserved. Therefore, the intersection product DFA is functionally the same as DFA 2.

TABLE XI
EXPERIMENT 3'S INTERSECTION PRODUCT DFA TRANSITION TABLE

	0	1
$\rightarrow q_0, a_0$	q_0, a_7	q_0, a_9
q_0, a_1	q_0, a_5	q_0, a_4
q_0, a_2	q_0, a_8	q_0, a_2
q_0, a_3	q_0, a_5	q_0, a_6
q_0, a_4	q_0, a_6	q_0, a_5
q_0, a_5	q_0, a_{10}	q_0, a_7
$*q_0, a_6$	q_0, a_8	q_0, a_3
q_0, a_7	q_0, a_2	q_0, a_7
$*q_0, a_8$	q_0, a_0	q_0, a_{10}
q_0, a_9	q_0, a_4	q_0, a_1
q_0, a_{10}	q_0, a_{10}	q_0, a_2

On the other hand, when computing the product of these two DFAs using the union, the resulting transition table is shown in Table XII and diagram shown in Fig. 10. Note that once again, as in the intersection case, the states and transitions of DFA 2 are preserved and prepended with q_0, but now all states are marked as accepting. This is due to the behavior of DFA 1; since all states in the product DFA contain at least one original accepting state, all states accept via union logic.

TABLE XII
EXPERIMENT 3'S UNION PRODUCT DFA TRANSITION TABLE

	0	1
$\rightarrow *q_0, a_0$	q_0, a_7	q_0, a_9
$*q_0, a_1$	q_0, a_5	q_0, a_4
$*q_0, a_2$	q_0, a_8	q_0, a_2
$*q_0, a_3$	q_0, a_5	q_0, a_6
$*q_0, a_4$	q_0, a_6	q_0, a_5
$*q_0, a_5$	q_0, a_{10}	q_0, a_7
$*q_0, a_6$	q_0, a_8	q_0, a_3
$*q_0, a_7$	q_0, a_2	q_0, a_7
$*q_0, a_8$	q_0, a_0	q_0, a_{10}
$*q_0, a_9$	q_0, a_4	q_0, a_1
$*q_0, a_{10}$	q_0, a_{10}	q_0, a_2

The preceding results can be further verified by picking strings that are accepted and using the `accepts_string` method to test them using the two product DFAs. The results, seen in Table XIII, demonstrate the logical operations between the original DFAs. Of important note, the table is truncated, as there is no case in which DFA 1 will not accept, so such cases were removed from the table. As can be seen in both the diagrams and from reasoning about the two original DFAs, the union and intersection results are as expected, thereby demonstrating the difference between union and intersection while also verifying an edge case of a singular state.

V. CONTRIBUTIONS

A. Evan Childers

For implementation, Evan created the `accepts_string` and `print_transition_table` functions, and provided minor fixes to the data cleaning process in

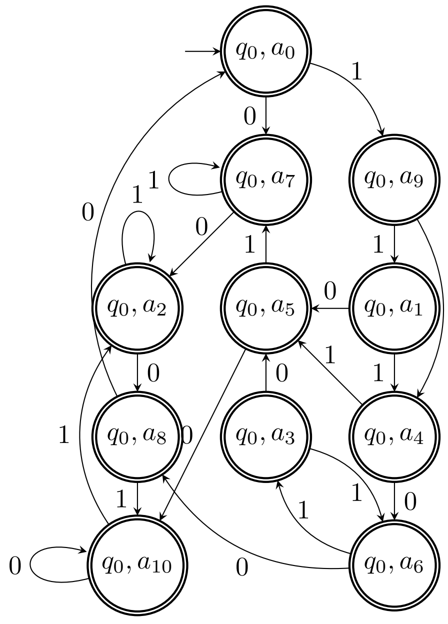

Fig. 10. Experiment 3's Union Product DFA

TABLE XIII
BOOLEAN ANALYSIS OF EXPERIMENT 3'S PRODUCT DFA OPERATIONS

DFA Type	10010	10011
DFA 1	True	True
DFA 2	False	True
Union DFA	True	True
Intersection DFA	False	True

the `read_dfa_file` method. He also implemented the `visualize_dfa` function, which applies the pre-existing *Automathon* Python library. These functions were used to validate the `product_construction` method by writing test cases for the involved DFAs in the `main` method of the code.

Outside of coding, Evan wrote the Introduction and Problem Statement sections of the paper. He also collaborated with Kai to develop and format the Experiments subsection, including the tables and automaton diagrams of the experimental results. He also assisted in writing the Abstract.

B. August Connors

August implemented the product construction algorithm. This involved adapting the union algorithm from the Sipser book to use both union and intersection. Further, August created the `find_unreachable_states` function, which utilized a BFS to find all reachable states, and then use that to derive the unreachable states list.

In addition to the coding contributions, August wrote the implementation section, covering the DFA class, product construction function, the BFS, and the testing functions. Furthermore, August included a pseudocode algorithm to help portray the method of product construction used.

C. Kai NeSmith

In terms of implementation, Kai created the underlying data structure for handling DFAs in Python, alongside providing the design and saving method, `save_dfa_file`, for the text file format to read and write DFA data to. They also created and maintained the README file of the repository, improved the `main` method of the code to allow for command line input, and implemented a GitHub Action that would validate and build the PDF version of this paper in order to validate simultaneous edits.

Outside of code, Kai worked with Evan to test the program by creating experiments and documenting them. In the case of the paper, they formatted the code output into LaTeX-compatible tables and automaton diagrams in order to deliver a polished final product, while providing the majority of content for Experiments 2 & 3. They also assisted in writing the conclusion and abstract of the paper.

VI. CONCLUSION

In this paper, we described the implementation of a Python-based product construction program which accepts two DFAs as input and produces a product-constructed DFA as output. For the implementation, we developed a basic data structure and built upon it by providing a variety of output options, ranging from the transition table, to a complete text representation, to graphs used to visually check results. We also devised a series of experiments in order to help guide our development and to test the final product and identify problems with the code. In all, we found our implementation to be satisfactory in representing the product construction process, thereby providing a solution to the problem posed to us. The source code repository for this project is available on GitHub: https://github.com/resistiv/CS575-S25-Group3FinalProject.

REFERENCES

[1] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.
[2] R. Quintero, "rohaquinlop/automathon," GitHub, Apr. 04, 2024. https://github.com/rohaquinlop/automathon