jsFLAP: A JavaScript Formal Languages and Automata Package for Computer Science Education

by
Elijah Cirioli

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science
(Honors Scholar)

Presented March 2, 2023
Commencement June 2023

AN ABSTRACT OF THE THESIS OF

Elijah Cirioli for the degree of Honors Baccalaureate of Science in Computer Science presented on March 2, 2023. Title: jsFLAP: A JavaScript Formal Languages and Automata Package for Computer Science Education.

Abstract approved:_____

Mike Rosulek

Visualization and simulation software serves an important role in education, especially in the education of abstract topics. The field of computational theory, and specifically the topics of formal languages and finite automata are well suited to visualization. When done properly, this improves the learning experience for both students and educators. The JavaScript Formal Languages and Automata Package (jsFLAP) was created to facilitate the construction and simulation of different types of finite state automata. Following best practices for visualization design, jsFLAP was designed to be accessible and easy to use. It aims to reduce frustration for students, allowing them to focus on engaging directly with the underlying concepts, and to require as little technological overhead as possible for instructors wishing to integrate it into their curriculum. In doing this, jsFLAP can serve an important role in the education of computing theory, replacing existing automata construction tools or reaching audiences that those tools could not.

Keywords: computer science, education, visualization, formal languages, automata

Corresponding e-mail address: ciriolie@oregonstate.edu

jsFLAP: A JavaScript Formal Languages and Automata Package for Computer Science Education

by
Elijah Cirioli

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science
(Honors Scholar)

Presented March 2, 2023
Commencement June 2023

Honors Baccalaureate of Science in Computer Science project of Elijah Cirioli presented on March 2, 2023.

APPROVED:

_____

Mike Rosulek, Mentor, representing Department of Electrical Engineering and Computer Science

_____

Christopher Hundhausen, Committee Member, representing Department of Electrical Engineering and Computer Science

_____

Julianne Coffman, Committee Member, representing Department of Electrical Engineering and Computer Science

_____

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, Honors College. My signature below authorizes release of my project to any reader upon request.

_____

Elijah Cirioli, Author

Background

Interactive visualizations serve an important role in the education of math and computer science topics. When done correctly, these visualizations can complement the other course materials to provide students with an opportunity to engage with the course concepts directly, bringing abstract and theoretical topics into a more understandable plane. In the paper, *Visualization as effective instructional and learning tools in the Computer Science Curriculum*, authors M. Quweider and F. Khan introduce the concept of Visualization-Reinforced Instruction (VRI), defined as "a form of active learning, in which traditional instructional material is complemented and enhanced with carefully designed animations and simulations of key concepts, models, and algorithms" [1]. Visualization-Reinforced Instruction does not seek to replace traditional learning materials, but rather to supplement them with student-driven simulations and visualizations of the core concepts.

This method of teaching has been shown to benefit student understanding of complex topics, especially ones in math and computer science that can so often be abstract and difficult to visualize or conceptualize without help. In a survey of students educated in computer science theory using VRI, 76% of them responded that the visualization helped them to understand the underlying concepts [1]. Furthermore, the researchers found that the visualizations allowed the educators to better understand student misconceptions and misunderstandings which the educators were then able to correct. Engaging with the content in a hands-on manner tests students' conceptual understanding and allows them to experiment with how modifying different parameters can alter the execution or result.

This is not to say that simply displaying a visual representation of a theoretical algorithm or computing model is enough to improve student understanding. In the paper *Exploring the Role of Visualization and Engagement in Computer Science Education*, Naps et al. [2] discuss both the benefits and challenges of implementing visualizations in computer science education. The authors claim that the two key obstacles to widespread adoption of visualization tools in computer science education are that the learner may not find them to be educationally beneficial, and the educator may find them to incur too much technological and structural overhead to be worthwhile. A good visualization tool must help students to understand the course material, and it must be easy enough for the instructor to integrate into the course itself. The latter is dependent on how the tool operates as a technology, but the former depends on how it is designed from the perspective of a user.

The researchers devise a set of best practices for designing such tools wherein they make it critically important that interactivity and engagement are more important than just animation when it comes to actual educational benefit. Broadly, these best practices focus on the ability of students to modify the visualizations and execute them at their own pace. Specifically, some best practices listed are to "include execution history", "support flexible execution control", "support learner-built visualizations", and "support custom input sets" [2]. All of this is to say that the visualizations are not simply an animation or picture representing the relevant concept, but are instead a tool through which the student can actively engage with the concept, experimenting and executing it themselves on their own schedule. When visualization tools are done well, the researchers again espouse their benefits for computer science education. All survey respondents they studied agreed with the statement that "using visualizations can help students learn

computing concepts". It is not just beneficial for student understanding, either. The researchers list many other benefits including an improved level of student participation, a more enjoyable teaching experience, and overall anecdotal evidence that the class was more fun for students.

The effectiveness of a visualization can be difficult to quantify, especially because what qualifies as a visualization is itself a large spectrum. One way this can be done is by applying Bloom's taxonomy of the cognitive domain: a hierarchical framework that is used to classify a learner's depth of comprehension along six increasingly sophisticated levels [3] shown in Fig. 1. The first level is knowledge and it is characterized by factual recall without any deeper understanding. At the next level, the comprehension level, the learner is able to understand the deeper meaning behind the facts. At the application level, the learner can take what they have learned and apply it to new concrete scenarios. Next, at the analysis level, the learner is able to understand the relationship between different ideas and break down problems into their component parts. When a learner reaches the synthesis level they are able to generalize from the facts that they have learned and draw new conclusions by combining multiple prior concepts. Finally, at the evaluation level, the learner is able to assess the value and effectiveness of the different materials they have learned in order to make reasoned choices about what to do when facing new scenarios. A properly engaging visualization tool adhering to the best design practices should allow students to at least reach the application level by having them apply the concepts they have learned to new scenarios or challenges. An even better tool would bring students to the synthesis level by allowing the combination of several concepts in order to achieve more complex results.

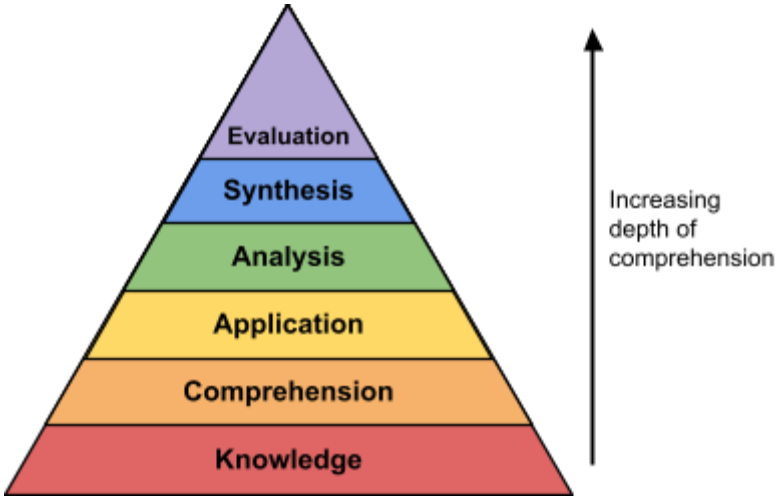

Fig. 1. The original six levels of Bloom's taxonomy

The field of computer science theory is a prime candidate for visualization tools. The computational model of finite automata and the formal languages associated with them can be difficult to understand from a mathematical perspective, but they are much more accessible when approached in a graphical manner. There is great potential for a tool that allows the construction and simulation of finite state automata in the education of computing theory. Students could be tasked with constructing a finite automaton to accept a certain language, they could be asked to analyze a given automaton to determine the language it accepts, or they could compare the equivalence of different automata. These examples all fall within the application level of Bloom's taxonomy, but students can reach the synthesis level through the combination of different computational theory topics. This includes tasks such as converting between regular expressions, grammars, deterministic and non-deterministic finite state automata, or comparing the computational power of models like the deterministic finite automata to that of the pushdown stack finite automata. The potential benefit for students learning about these topics is clear, and

the need for a tool that will allow them to construct and simulate different types of finite state automata is present.

Given that the need for such a tool is apparent, one of them already exists. The Java Formal Languages and Automata Package (JFLAP) [4] is a graphical application for the construction and execution of finite state automata. It supports regular and context-free languages through pushdown automata, as well as recursively enumerable languages through the implementation of Turing machines. Development for JFLAP started in 1990 at Rensselaer Polytechnic Institute, and it has received numerous updates and improvements over the past three decades. It has been downloaded over 64,000 times by people in 161 countries, and it is used in over 20,000 college courses. JFLAP is a powerful tool that has served an important role in educating thousands of people about computer science theory over multiple decades, but it is not without its flaws.

The core functionality that JFLAP provides is good, but it suffers from a level of inaccessibility. It is a Java program and thus requires all users to have the Java Virtual Machine installed on their computer. This also means that it cannot be used on Chromebooks or mobile devices. Generally speaking, requiring all students to install any piece of software can sometimes result in too much technological overhead for educators trying to integrate a new tool into their courses. When the software has been installed, using it can be difficult and cumbersome. The user interface of JFLAP reflects the period in which it is created, and simple tasks such as creating multiple transitions between the states of a finite automaton could be characterized as “clunky”. There is usually only one specific way to achieve any desired outcome in the tool, and

some useful features are not present at all. It lacks some educationally useful algorithms like product automata and cycle detection, and it lacks quality-of-life features such as keyboard shortcuts, editor tabs, or good algorithms to reorganize automata in a visually understandable way.

A perfect tool for educating about computer science theory using finite state automata would be easy to access and integrate into a course, and students using it would not have to struggle with using the tool itself, but could rather focus on the concepts they are trying to learn. This thesis represents an attempt to create one that is closer to that idealized goal: an accessible and intuitive educational tool for constructing and executing different types of finite state automata to visualize the abstract topics of computer science theory.

Implementation

This thesis regards the creation of the JavaScript Formal Languages and Automata Package (jsFLAP), a browser-based tool for constructing and simulating finite state automata for the purposes of computer science education. It is implemented with the modern tools of HTML5, CSS3, and JavaScript, with the addition of the JavaScript library jQuery for easier manipulation of DOM elements. All of the JavaScript code in jsFLAP is run in the client's browser, meaning that no external server or internet connection is required. This was done intentionally so that jsFLAP would be as easy as possible to integrate into existing course materials. It can be packaged into an executable for students to download, embedded within an existing website, or hosted statically. Students using jsFLAP only need a modern-enough web browser and potentially an internet connection depending on how the tool is hosted. There are no requirements for students to download any additional programs, and all operating systems capable of displaying a website are automatically supported. Because it is built this way, jsFLAP should be easy to access for both students and instructors who wish to use it in their course. As discussed by Naps et al., a visualization tool has reduced effectiveness when it incurs significant technological overhead [2], so this convenience is of critical importance for jsFLAP.

The design philosophy of high user flexibility extends to the coded implementation itself. The overall structure of jsFLAP allows for great extensibility with compartmentalized, object-oriented modules that can be built upon or replaced to suit the needs of the specific course where it is being used. The code is open source and freely available on GitHub under the MIT license, and it is documented well enough that other developers should find themselves able to understand its internal workings. The goal of this again is to reduce the overhead required for

integrating jsFLAP into a school's existing curriculum. If someone wishes to modify jsFLAP to connect it to their own autograder or to implement a specific feature that they need, a reasonably-capable programmer should be able to do it with modest effort.

The development of jsFLAP began in October of 2021 and lasted approximately 11 months. Further development may occur to add additional features or fix issues, but for the time being jsFLAP is in a finished state with all of the originally-intended functionality present. The goal of jsFLAP's implementation was to utilize the best practices of software development in order to be modern, modular, and extensibile enough to stand the test of time. Even with large breaks in between future development cycles, the hope is that jsFLAP will remain easy to improve and expand upon for years to come.

Features

The core functionality of jsFLAP is centered around the construction of finite state automata using states and transitions which are visually analogous to the nodes and edges of a directed graph. The user first selects which type of automaton they wish to construct: a standard finite state automaton for parsing regular languages, a pushdown automaton for parsing context-free languages, or a Turing machine for parsing recursively enumerable languages. They are then free to use the mouse to create automata states and connect them with transition arrows. The states can be configured to be an initial state that acts as an entry point for parsing, a final state that indicates language acceptance, neither, or both. The transitions for the standard finite state automata are configured with alphabet characters that can be consumed by the automaton when changing states as shown in Fig. 2. The transitions for the Pushdown automata and Turing machines instead use tuples to indicate not only the character that will be consumed, but also the stack or tape transformation that will occur as part of the transition as shown in Fig. 3. All of this serves as a visual way of representing the agreed-upon mathematical definitions for each of these models, and it is consistent with the visual representations of finite state machines across educational literature.

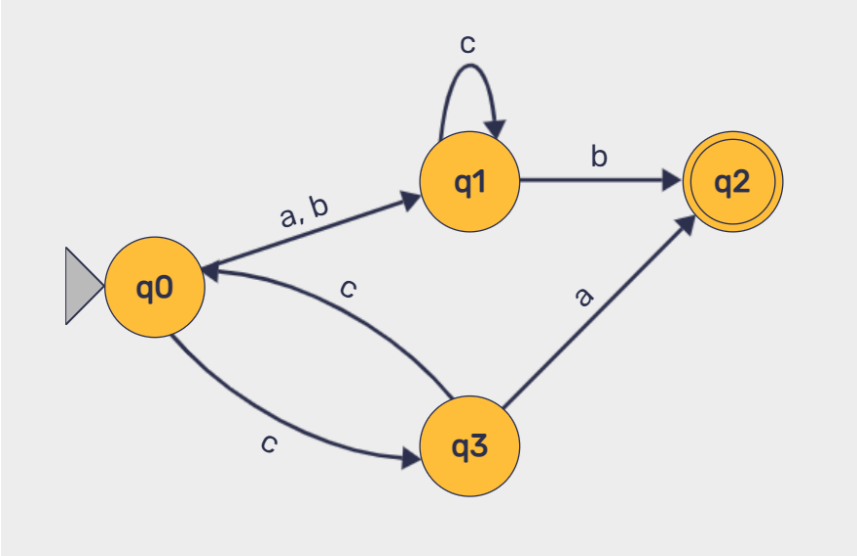

Fig. 2. A Non-deterministic finite automaton created in jsFLAP

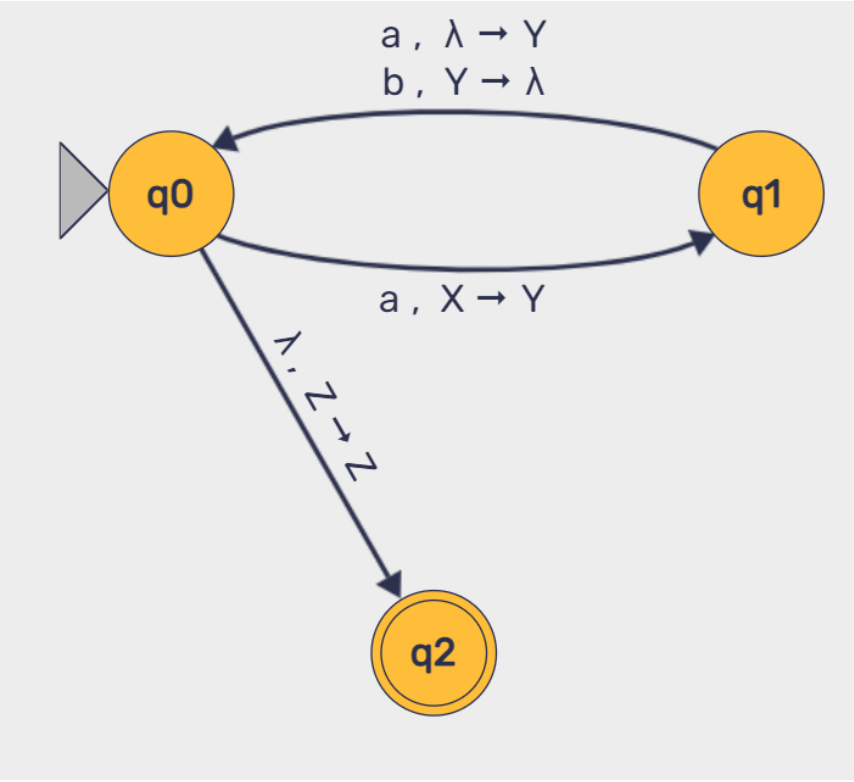

Fig. 3. A Non-deterministic pushdown automaton created in jsFLAP

Once an automaton has been constructed, it can be simulated in one of two ways. The outcome of the simulation is dependent on what type of automaton is being simulated. The standard finite state automata and pushdown automata are language acceptors, so they take potential words as inputs and return whether or not they are part of the language. The default mode of simulation for these types of automata allows the user to enter as many words as they wish to test and they will receive a visual indication of whether or not the automaton they constructed has accepted each one as shown in Fig. 4. This is an important feature for users who wish to quickly test whether their automata meets their desired criteria by entering words they know should or should not be in the language. A distinguishing feature that separates jsFLAP from other similar programs such as JFLAP is that these simulations happen in real time. When the user enters a new letter or modifies their automaton the tests outputs will automatically update to reflect the changes. The simple change of not requiring users to manually run the simulation means that users can experiment with their automaton and see how modifying it changes the language that it accepts in real time. A potential use for this is to define a series of test cases before beginning the automaton construction and then watching as more words get accepted when new states and transitions are added.

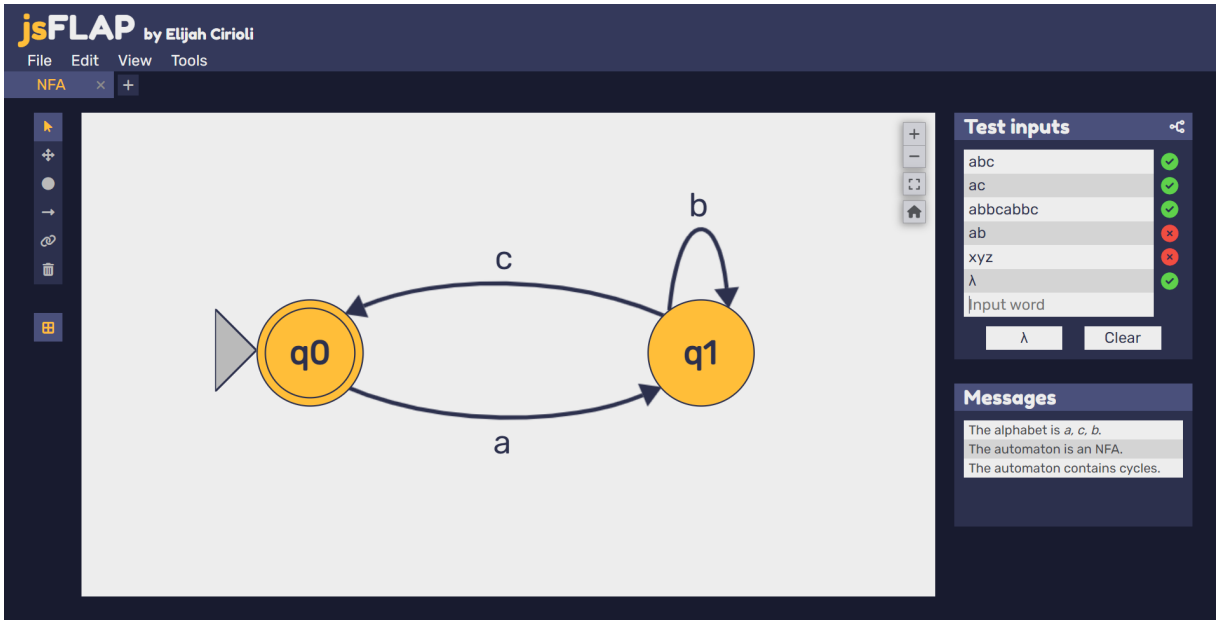

Fig. 4. Testing multiple inputs using an NFA

The second mode of simulation is step-by-step parsing. In this mode, the user enters a single word they wish to parse and then they click through each step of the parsing process as the automaton moves between states as shown in Fig. 5. When non-determinism is involved, the parse tree that is displayed will branch to represent the different routes that the automaton can take. When it reaches a state with no more usable transitions, the branch ends and the leaf node is colored based on whether the word was accepted or not. The user can navigate through this parse tree backwards and forwards, navigating down different branches to better understand exactly how the automaton is operating. The current state of the automaton is highlighted and the portion of the input word yet to be consumed is displayed in a table format.

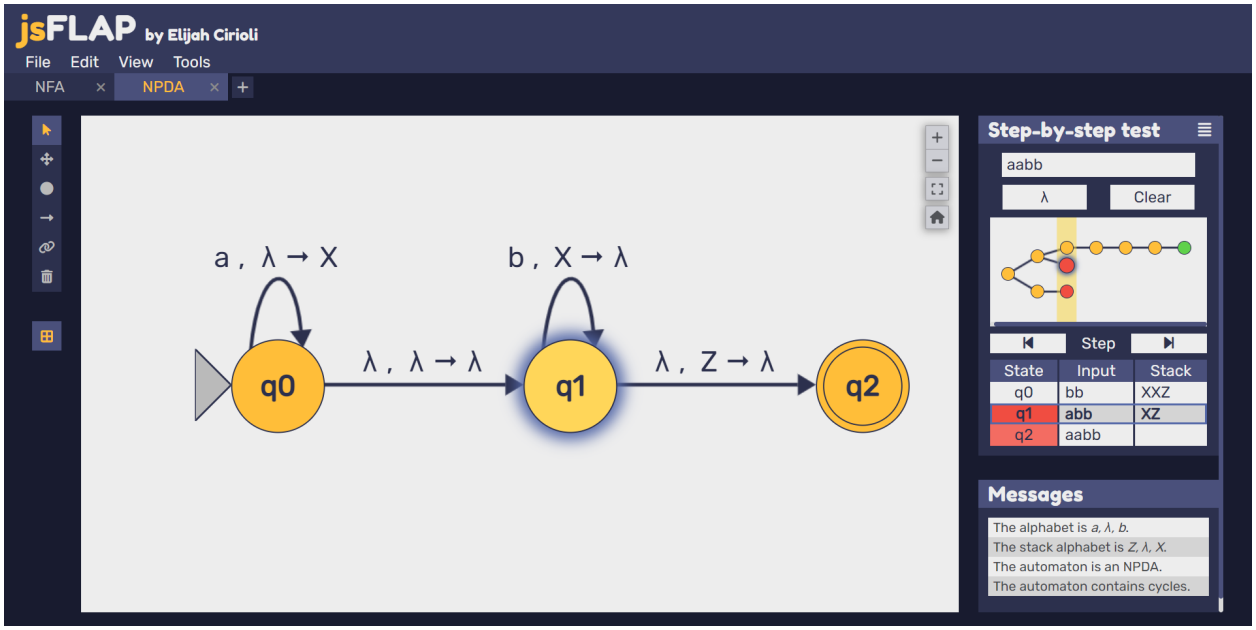

Fig. 5. Parsing a single input step-by-step using an NPDA

The simulation of Turing machines operates on the same general principle, with a few key differences. They are implemented as a-machines described by Alan Turing in the paper *On computable numbers, with an application to the Entscheidungsproblem* [5], wherein the finite state automata does not accept a word as input, but rather operates as a computer with access to an infinite writable tape. The machine can read and write a single cell of the tape at a given time, and the logic of the finite state machine is used to determine what to write, where to move the read/write head next, and whether the machine should halt. The machine's input is written on the tape by the user prior to the start of execution, and the output can be read off of the same tape. Implementing this functionally necessitated a tape interface in jsFLAP where users can enter the initial tape data with infinite blank spaces extending off on either end. The result of parsing is no longer a boolean value of whether the word was accepted, but rather the final contents of the tape and the current position of the read/write head. Additionally, a boolean visual indicator is used to

indicate whether the Turing machine reached a halt state, or whether its execution was stopped after reaching a user-defined maximum number of iterations. The step-by-step parsing mode now displays the current tape contents, as well as the position of the read/write head as shown in Fig. 6. The maximum number of alphabet characters, state machine states, and tape size have no hard value, but are rather a function of the computer that is running jsFLAP's resources. Thus it is trivial to prove the Turing-completeness of jsFLAP as it can be used to simulate any Turing machine and is therefore capable of solving all computable problems that other universal Turing machines can.

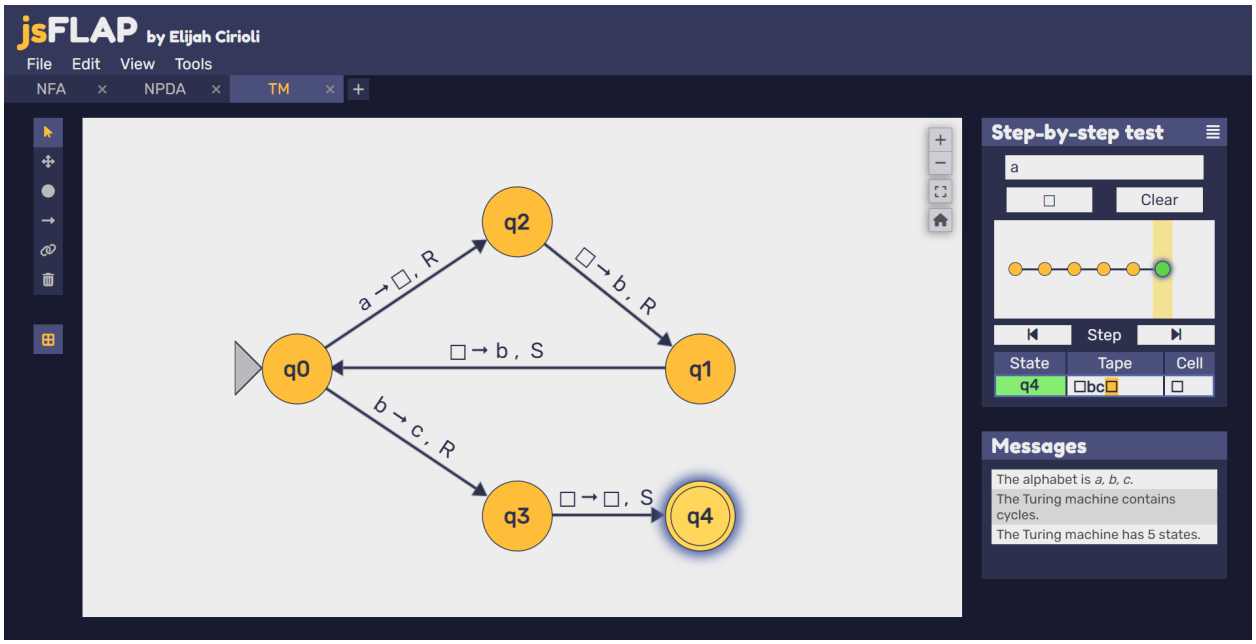

Fig. 6. Parsing a single input step-by-step using a Turing machine

Through the construction and simulation of these different types of finite state automata, jsFLAP should allow users to reach the application level of Bloom's taxonomy. Users can construct finite state automata to accept a certain language and then provide their own inputs to test whether they have succeeded. This, along with the flexible execution control and execution

history offered by step-by-step parsing, brings jsFLAP in line with the visualization best practices described by Naps et al. [2]. The versatility of the construction tools allow jsFLAP to be used for a wide variety of purposes such as lecture demonstrations, homework assignments, and in-class activities. On top of the basic construction and simulation functionality present, additional tools and features were implemented for both quality-of-life and to promote more complex synthesis education.

While it may be easy to discount features that slightly reduce user frustration, their role in educational tools is an important one. The goal of such tools is to engage students with the content, not to make them struggle with an unwieldy user interface. There are many features in jsFLAP that seek to address this. The editor features a “messages” window that updates in real time as an automaton is constructed with information that the user may find helpful. If their automaton lacks an initial or final state, or if any of the states are unreachable, a warning message will appear to inform them. This feature acts as a first line of defense against some of the common mistakes that beginners to finite state automata might make. Additional messages contain information such as the current alphabet, the number of states, and whether non-determinism is present. These messages can help to confirm a student’s basic understanding of what they are creating, or act as confirmation that an automaton design fulfills some expected criteria. More features focused on improving usability include copying and pasting, undo and redo functionality, keyboard shortcuts for all major actions, informative tooltips when hovering over buttons, automatic saving of progress, customizable color themes, and automatic layout algorithms to reformat automata into more organized shapes. When combined together, these small features help to eliminate many sources of annoyance and frustration for users of the

program. Making jsFLAP a more enjoyable program to use allows those using it to focus more on what they are doing and less on the program itself.

Many of the more complex features and algorithms implemented in jsFLAP are not for user convenience, but are instead to aid in deeper conceptual understanding of computing theory. A key topic in the theory of computation is the computational power of these different models, and the equivalence that exists between some of them. Any non-deterministic finite state automaton can be represented as an equivalent deterministic finite state automaton, and jsFLAP can demonstrate this with its feature for converting NFAs to DFAs. The equivalence of any two NFAs can be tested which could be a helpful tool for grading when students are asked to create a finite automata that accepts a certain language. This feature can also play a more important role in allowing students to prove NFA-DFA equivalence to themselves and to explore the infinite possibilities for constructing finite automata that accept a single language.

The concept of representing the same regular language in different forms is further demonstrated by the regular expression to NFA converter. With this tool, students enter a regular expression into jsFLAP and it constructs an NFA that accepts the same language. These conversion and equivalence tools allow students to demonstrate the relationships between different models of computing, which may bring them to the synthesis level of Bloom's taxonomy where they are generalizing and combining several different concepts they have learned to solve new problems.

Another feature of jsFLAP that brings together multiple concepts of computing theory is the product automata tool. This tool allows the user to construct the cartesian product of two finite state automata as shown in Fig. 7. With this tool, students can perform set operations on automata to perform those same operations on the languages they accept. For example, a cross product of two finite state automata could be created to accept the intersection of them both. It then follows that the language defined by this new automaton would accept all words that exist in both the component languages, which is the language intersection. The same concept can be applied to find the union of two languages, as well as the difference which acts as another way to prove equivalence. If the union of the languages created by taking the difference of two automata in both possible orders is the empty set, then the two automata must be equivalent. Seeing this demonstrated visually and being able to step through the parsing manually may allow students to better understand this somewhat complex merging of concepts. They are deepening the connection between finite automata and the languages that they represent as well as solving existing problems in new ways by choosing from a set of provided tools. This again should help bring students towards the synthesis level of Bloom's taxonomy. When students are able to combine multiple tools and concepts they have learned about to solve problems then they have shown themselves to reach the synthesis stage [3].

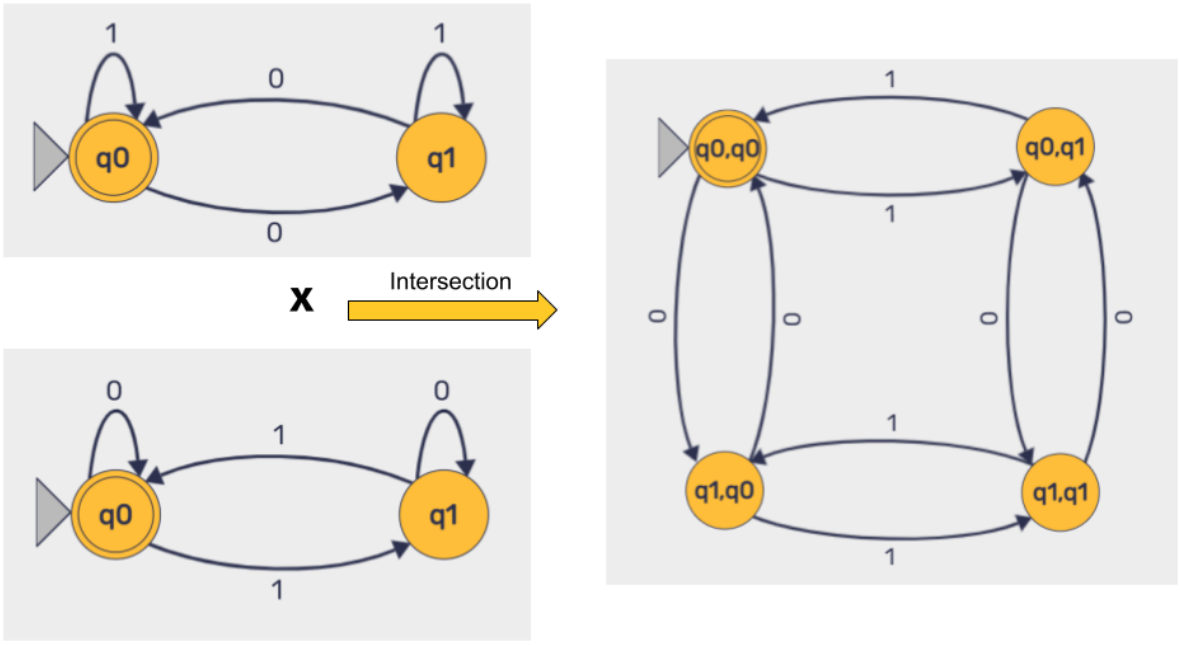

Fig. 7. The product of two DFAs representing the intersection of their two languages

Potential for Future Improvement

The open source nature of jsFLAP means that anyone could create improvements for it in the future. There are many areas that can be expanded and reworked to add functionality and create a better overall user experience. Great work was put in to make jsFLAP accessible to as many people as possible, but expanding this must be a key area of focus for future improvements. Button tooltips and color customization help users with vision impairments and all actions are timed and driven purely by the user. There are no time-sensitive events or unprompted animations that users may find challenging as discussed by Naps et al. [2]. In its current state, jsFLAP depends heavily on the mouse for input. It is technically possible to use it with a touchscreen, but this is somewhat cumbersome. Ideally, jsFLAP should support a fully keyboard-based input method that would help users with certain motor impairments, but this is easier said than done because the whole tool is designed around drawing graphs. Additionally, work could be done to improve the experience of using jsFLAP with a screen reader. The basic functionality for that is present, but it is far from refined.

After accessibility has been expanded, there is more potential for improvement through added functionality. Grammars are another common way of representing formal languages and they are often used in computing theory courses. Adding grammar functionality to jsFLAP would require a whole new suite of interfaces as they are not graph based like automata are. Adding the functionality to define grammars and convert between them and automata would allow jsFLAP to play a role in more course instruction, and it could deepen students' understanding of the fundamental equivalence between the different computing models.

Another feature that could improve students' understanding is explicit proof visualizations. Currently, jsFLAP allows students to demonstrate and prove concepts to themselves, but more formalized proofs that walk the student through the process could expand the role of jsFLAP from a tool used within an educational context to an educational platform in and of itself. For example, students could see the pumping lemma used to prove that a language they defined is not regular, or get a visual demonstration of the halting problem for Turing machines. To properly transform jsFLAP into an educational platform, functionality would have to be added to formally evaluate student work and provide them with challenges. Lessons could be implemented that introduce a concept and then challenge students to create an NFA using that concept, which is then automatically evaluated for correctness. These lessons could be hardcoded into jsFLAP, or defined by instructors using some specialized file format. There is great potential here, but it is outside of the scope of this thesis. As it exists right now, jsFLAP has the functionality necessary for educational adoption, it is just a matter of getting it into the hands of students.

Educational Usage

A key feature of jsFLAP that should aid in its adoption is its interoperability with existing tools. JFLAP is used in over 20,000 college courses, and jsFLAP has near feature parity while being easier both to access and to use. Realistically, jsFLAP could be used to replace JFLAP in many of these courses and it could potentially lead to a better learning experience for the students taking them. To make this process easier for educators, jsFLAP allows the importing of JFLAP files. This will let instructors use their existing examples and assignments in jsFLAP without having to recreate anything.

Even in courses that do not already use a similar tool, jsFLAP could be integrated into the course material to improve student engagement. As discussed by M. Quweider and F. Khan, the majority of students report that effective visualizations help them to better understand the underlying concepts [1]. This makes a compelling argument for giving students learning computer science theory the opportunity to use jsFLAP as part of their classes. In addition to the primary use of jsFLAP by students, educators may find it helpful for creating finite state automata graphics or for providing live demonstrations of topics. It can be used for a wide range of courses extending into advanced topics involving Turing machines, but the target use case for jsFLAP is in introductory level courses where students are first being introduced to these new topics.

Despite its recent release and lack of formal literature, jsFLAP has already found a use in education. A professor at Harvey Mudd College has integrated jsFLAP into their introductory computer science course for their unit on finite automata starting in the 2022 school year. They

previously used JFLAP for this course, and by switching to jsFLAP they have proven it to be usable as a modern replacement. Additionally, they have connected jsFLAP to their normal auto-grading system for assignments which is a concrete demonstration of jsFLAP's extensibility. There is no reason that Oregon State University could not follow the lead of Harvey Mudd College by replacing JFLAP with jsFLAP as part of the CS 321 Theory of Computation undergraduate course. Doing so would benefit students by giving them a more accessible and usable tool for visualizing and simulating the fundamental concepts of the course. It would also benefit educators by eliminating the need for them to install any programs or make students install them for the purposes of the course.

Conclusion

Effective and engaging visualization tools help both students and instructors in the education of complex topics, especially abstract ones such as computational theory. This theory is fundamental to much of the field of computer science, so it is critical that students have a solid comprehension of it. To aid in some part of this, jsFLAP was created as a tool to construct and simulate different types of finite state automata. It provides an easy way for students to engage with the material in a hands-on manner, working through misconceptions and applying practically what was once highly theoretical. This can help to deepen understanding of computational theory which will benefit students at all levels of computer science education.

While not the first visualization tool for this purpose, jsFLAP is more accessible, making it easier for educators to integrate it into their curriculum. It is also easier to use than existing tools which allows users to spend less time worrying about how to use it, and more time focusing on the underlying concepts that they are learning. The convenience of jsFLAP, as well as its potential to be adapted and customized to different application environments, make it a powerful tool for education in the field of computational theory. If jsFLAP is used for computer science education at the Oregon State University and other universities like it, it has the potential to benefit thousands of engineering students.

Works Cited

[1] M. Quweider and F. Khan, “Visualization as effective instructional and learning tools in the Computer Science Curriculum,” *2017 ASEE Annual Conference & Exposition Proceedings*, 2017.

[2] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide, “Exploring the role of visualization and engagement in Computer Science Education,” *ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 131–152, 2003.

[3] B. S. Bloom, *Taxonomy of educational objectives: The classification of educational goals*. New York City, New York: Longman, 1984.

[4] S. Rodger, “Learning automata and formal languages interactively with JFLAP,” *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education - ITICSE '06*, 2006.

[5] A. Turing, “On computable numbers, with an application to the Entscheidungsproblem (1936),” *The Essential Turing*, 2004.