Finite State Machine Educational Tools: Implementing Equivalence Testing and State Minimization Algorithms

Final Project Report

Author: Sten Arthur Laane

Supervisor: Dr Agi Kurucz

Student ID: 1857758

April 2021

Abstract

Deterministic Finite Automata (DFAs) find widespread use in modern technology, yet current interactive software for learning about and working with them is dated and unintuitive. This problem is especially acute for DFA state minimization and equivalence testing, for which algorithms have existed for decades, yet are not available in most DFA software, and no solutions are web-based. This project addressed the problem by creating an educational open-source web application based on modern design principles. Three popular algorithms, The Table-Filling Algorithm, The *n lg n* Hopcroft Algorithm, and The (Nearly) Linear Algorithm, were implemented and visualized step-by-step. Custom datasets that cause worst-case performance were created for each algorithm using in-depth analysis. Users can utilize these datasets and run algorithms in headless mode without visualizations to compare their results and running times. Compared to similar existing systems, the software is more interactive, user-friendly, and lightweight.

Originality Avowal

Sten Arthur Laane

April 2021

Acknowledgements

This section is dedicated to dr Agi Kurucz, the project's wonderful supervisor. Her guidance and feedback for every part of the project from idea to execution has been invaluable. Her expertise and insight into the world of informatics has been inspiring and greatly contributed to the success of this project.

Table of Contents

Abstract..2
Originality Avowal..3
Acknowledgements..4
Introduction..7
1.1. Aims and Objectives..7
1.2. Scope..8
Background..9
2.1. Automata Theory..9
2.2. Analysis of Similar Systems..12
2.2.1. Native Applications..12
2.2.2. Web Applications..17
2.2.3. Comparison and Learnings..19
2.3. Algorithms..20
2.3.1. The Table-Filling Algorithm..20
2.3.2. The *n lg n* Hopcroft Algorithm..26
2.3.3. The (Nearly) Linear Algorithm..31
2.3.4. State minimization..36
2.4. Datasets..38
Requirements and Specification..42
3.1. Brief..42
3.2. Requirements..42
3.2.1. Front End Requirements..43
3.2.2. Environment Requirements..44
3.2.3. Algorithm and Dataset Requirements..44
Design..47
Implementation..50
5.1. Methodology & Tooling..50
5.2. Common Front-End..52

5.3. Algorithms and Datasets 56
5.4. Testing 62
5.5. Optimization 64
Professional and Ethical Issues 65
6.1. Ethical Concerns 65
6.2. British Computing Society Code of Conduct 65
Experimentation 67
7.1. Datasets 67
7.2. Comparison of Algorithm Running Times 74
Evaluation 76
8.1. Software 76
8.1.1. Brief 76
8.1.2. Requirements 76
8.2 Experimentation 77
Conclusion and Future Work 78
9.1. Conclusion 78
9.2. Future Work 79
References 80

Chapter 1

Introduction

The notion of finite automata was first introduced in the 1940s and the mathematical concept behind it was formalized in the beginning of the 1950s. Finite automata can be viewed as a model of computing with a finite set of states and fixed transitions from one state to another based on an input string. (Constable 1980)

Since then, finite automata have found use in the fields of text processing, mathematical verification, hardware design, biology, genetics, and others (Kryvyi 2011). The practical aspects of automata necessitate the use of minimization to create an equivalent automaton that uses less resources (physical or virtual) while providing the same functionality (Constable 1980).

State minimization also gives a way to check the equivalence of two automata, since the process of minimization produces a single unique (minimal) automaton. Because this automaton is unique, two automata with an identical minimized form are equivalent. This kind of equivalence testing has applications in the field of formal program verification and reliability. (*ibid.*)

1.1. Aims and Objectives

The goal of this project is to create software that will be used in educational contexts, *e.g.* teaching computer science students about deterministic finite automata (DFAs), their equivalence and DFA state minimization. To achieve this, the software will implement and visualize state minimization and equivalence testing algorithms, working through them step-by-step and providing detailed information that students can use to internalize the workings of the algorithms.

The software will come with custom example datasets that showcase differences between the implemented algorithms and highlight their performance in worst-case scenarios. These datasets will be created by data generation scripts made specifically to exploit each algorithm's weaknesses. Example data will be sufficiently large (>1000 states) to show how worst-case examples significantly impact run time of the algorithms chosen and can be used to compare algorithms with each other.

The software will be a helpful addition to the core informatics curriculum to expand curious students' knowledge about finite automata and algorithm time complexity while also providing utility as a tool that can be used to quickly minimize automata or check their equivalence.

Additionally, the DFA state minimization and equivalence testing tools currently available on the internet are severely lacking. None of the algorithms used in this project have been implemented on online tools. This project will provide an accessible way to run these algorithms without the need for a custom software installation or compilation by creating a web application.

1.2. Scope

This software created in this project will implement and provide optional visualization for three DFA state minimization and equivalence testing algorithms: The Table-Filling Algorithm, The *n lg n* Hopcroft Algorithm, and The (Nearly) Linear Algorithm. Besides implementing the algorithms in their original forms, additionally variants of them will also be implemented, namely ones producing a witness string that indicates why two automata being compared are not equivalent.

These algorithms were chosen because they are covered in-depth in Daphne A. Norton's 2009 M.S. Project Report „Algorithms for Testing Equivalence of Finite Automata, with a Grading Tool for JFLAP", which will act as the main bibliographic source for this project. Norton implemented and visualized the algorithms using Java and JFLAP. These implementations will be used as a reference and comparison point for this project.

In terms of target platforms (hardware and OS configurations) the project will run on, there will be no specific limitations. Since the aim of the project is to produce a web application, it will only depend on the JavaScript (JS) language and browser functionality. Because modern browsers run on most hardware/OS platforms, the resulting software will not be limited and will run equally well on MacOS, Windows, Linux, and other modern operating systems.

Chapter 2

Background

2.1. Automata Theory

John von Neumann recognized computers as "more than a high-speed calculator", noting that a computer would allow „a new form of scientific modeling, and it constituted an object of scientific inquiry in itself". With this work, he laid the foundations for the formal study of computers as "artificial organisms", *i.e.* the theory of automata. (Mahoney 2010)

In the 1950s, Noam Chomsky began studying formal grammars in the field of linguistics (Hopcroft, Motwani, Ullman 2006). He described grammars as a construct for enumerating sentences for formal languages, which are "collections of sentences ... constructed from a finite alphabet of symbols" (Chomsky 1959). Chomsky classified these formal grammars into a hierarchy of four distinct types and associated each class with languages and automata capable of recognizing the languages (*ibid.*). Using his model, automata can be described as abstract computing models capable of recognizing and accepting or rejecting some formal language.

This project will focus on deterministic finite state automata (DFAs). These are defined in Chomsky's classification hierarchy as machines capable of recognizing Type-3 grammars and their languages, all regular languages – languages that can be constructed from regular expressions (*ibid.*).

Formally defined, a deterministic finite automaton A is a five-tuple

$$A := (Q, \Sigma, \delta, q_0, F)$$

where

1. Q is a finite set of all states in the DFA.
2. Σ is the alphabet of the DFA, a finite set of input symbols.
3. δ is a transition function that takes a state and an input symbol as arguments and returns a state. $\delta: Q \times \Sigma \rightarrow Q$.
4. q_0 is the starting state of the DFA, one of the states in Q.
5. F is the set of final (accepting) states. $F \subseteq Q$.

DFAs are capable of processing strings in what is called a computation (sometimes also called simulating a run). When a DFA is given an input w, a string consisting of letters in its alphabet, a computation starts by finding the starting state of the DFA, looking at the first symbol w_0 of the input, and following the transition $\delta(q_0, w_0) = q_i$ to the some state q_i. The next step of the computation is to look at the second symbol in the input and follow the transition function from q_i on that symbol. This process continues until the end of the input is reached, in which case the DFA will reach some state q_f. If q_f is an accepting state ($q_f \in F$), the DFA accepts the input w, otherwise it rejects it. (Hopcroft *et al.* 2006)

The language of an automaton, L, consists of all the strings the automaton accepts. DFAs only accept regular languages, and all regular languages can be described using a regular expression. This implies that every DFA has an equivalent regular expression describing the language it accepts. (*ibid.*)

Regular expressions can be viewed as rules for creating strings. They are made up of three distinct components: symbols, the Kleene star *, and the union operator ∪. Specific symbols not followed by a Kleene star or union operator mean the symbol must be present in the position it appears in the expression. A Kleene star next to an expression (consisting of symbols, Kleene stars and unions) means the expression can be repeated 0 to infinity times in accepted strings. Finally, a union can be constructed from two expressions, A and B, indicating that a choice of either A or B is accepted.

For example, the expression 010 describes the language accepting only the string "010", while 01*0 describes the language accepting "00", "010", "0110", "01110", *etc.* up to a string with an infinite number of 1s in the middle. The expression $10 \cup 01$ describes the language accepting both the strings "10" and "01" and nothing else.

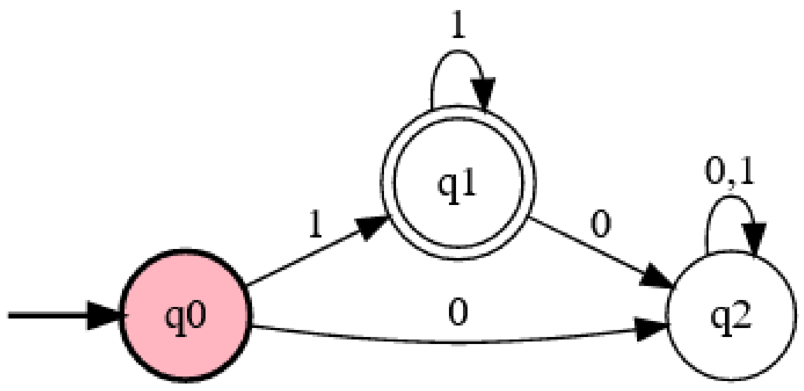


Figure 1. Graphical representation of a DFA accepting the language of 1.*
Source: Zuzak, Jankovic 2017

Pictured in Figure 1 is a graphical representation of a DFA accepting the language 1*. The states of the DFA are represented as graph vertices, the starting state is indicated in red and final states are distinguished with a double circle. Directed edges between vertices represent transitions from one state to another and are labeled with the input symbol that is used for the transition. In this example,

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- $\delta = \{(q_0, 0) \rightarrow q_2;\ (q_0, 1) \rightarrow q_1; (q_1, 0) \rightarrow q_1;\ (q_1, 1) \rightarrow q_2;\ (q_2, 0) \rightarrow q_2;\ (q_2, 1) \rightarrow q_2\}$
- $q_0 = q_0$
- $F = \{q_1\}$

The transition function is usually specified in the form of a table. For the DFA in Figure 1, it is visualized in Table 1.

	0	1
q_0	q_2	q_1
q_1	q_2	q_1
q_2	q_2	q_2

Table 1. Transition function for DFA in Figure 1.

The DFA in Figure 1 accepts all strings consisting only of ones. Whenever it encounters a 0 in its input, it transitions to q_2, from which it can only transition to q_2. Since q_2 is non-accepting and there is no way to get to other states from it, it is called a trap state (Norton 2009).

For convenience, we define an extended transition function $\hat{\delta}$ that takes a state q and a string w as input and returns the state that the automaton reaches when starting in state q and processing the sequence of inputs w (Hopcroft *et al.* 2006). For the DFA in Figure 1, $\hat{\delta}(q_0, 1111) = q_1$, after processing "1111" starting in state q_0, the DFA will reach state q_1. Additionally, $\hat{\delta}(q_0, 10111) = q_2$, after processing "10111" starting in state q_0, the DFA will reach state q_2.

Running a computation of a DFA for some string w is equivalent to checking if applying the extended transition function on the starting state q_0, $\hat{\delta}(q_0, w) = q_f$, returns a state that is accepting. For example, to compute if the DFA in Figure 1 accepts the string "10", we can compute $\hat{\delta}(q_0, 10) = q_2$. Since q_2 is not an accepting state, the DFA rejects "10". We can check the correctness of this by stepping through the transitions one-by-one. When the DFA is in state q_0 and encounters the first symbol of the input, 1, it will transition to q_1. Now, when it encounters 0, it transitions to q_2. Since the whole input has now been processed and q_2 is not an accepting state, the DFA rejects the string.

Two DFAs are considered equivalent if they accept the same language, otherwise they are not equivalent. To show that DFAs are equivalent, one must show that they have indistinguishable start states. Two states of some DFA are distinguishable if and only if there exists some input that takes one of the states to an accepting state and not the other. This means that if one can produce an input that one DFA would accept but not the other, then the DFAs are not equivalent, because both DFAs must start computation at their start states, which are shown to be distinguishable. Formally, two DFAs A and B are equivalent if their starting states are indistinguishable and two states p and q are indistinguishable if for all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is also an accepting state. (Hopcroft *et al.* 2006)

An input that demonstrates two DFAs' non-equivalence is called a witness string (Norton 2009).

A consequence of there being a way to test DFA equivalence is that there is a way to minimize a DFA, meaning to find an equivalent DFA that has the minimum number of states but still recognizes the same language (Hopcroft *et al.* 2006). The minimal DFA is unique for every language, although its states may be labeled with arbitrarily different names (Norton 2009). A minimal DFA can be reached from a non-minimal one by eliminating all states that cannot be reached from the start state (unreachable states) and then partitioning all remaining states into groups of indistinguishable ones and combining these blocks into one single state (*ibid.*).

2.2. Analysis of Similar Systems

2.2.1. Native Applications

There are many native applications for working with DFAs, with varying capabilities and aims. Many of these apps have been unmaintained for years (or even decades), with some having out-of-date binaries, which means users must compile these apps locally from sources instead.

Below is an overview of the more popular and powerful native DFA applications, analyzing their capabilities and paying special attention to the state minimization and equivalence testing algorithms implemented (if any exist).

JFLAP

JFLAP is Java software that is used as a toolkit for working with formal languages, deterministic and non-deterministic finite automata, pushdown automata, Turing machines and other parts of automata theory. Additionally, JFLAP functions as a program with a graphical user interface (GUI) that can be used to learn about various automata and visualize how they work. On the JFLAP website, it is described as “a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory” (JFLAP Version 7.1 2018).

JFLAP has found broad usage in the academic community, with tens of papers published that either use it or modify it to implement custom algorithms. It has also been the subject of over 15 conference talks. JFLAP was recognized as a finalist candidate in the NEEDS Premier Award for Excellence in Engineering Education Courseware competition in recognition of its effort to teach computer science students about automata theory. (*ibid.*)

JFLAP provides functionality for DFA state minimization, both by pre-built GUI-based ways and by exposing application programming interfaces (APIs) to do it in code. Both ways are implemented using an algorithm that partitions states into groups of distinguishable ones using a tree and then constructs a minimized DFA by combining the groups of indistinguishable states into single states. This implementation is similar to the Table-Filling Algorithm described in section 2.3.1. According to Daphne A. Norton, who did an analysis of the time complexity of this implementation, it “is some polynomial on n – a polynomial far greater than it truly should be” because of an inefficient implementation (Norton 2009). There are no other state minimization algorithms natively available in JFLAP software.

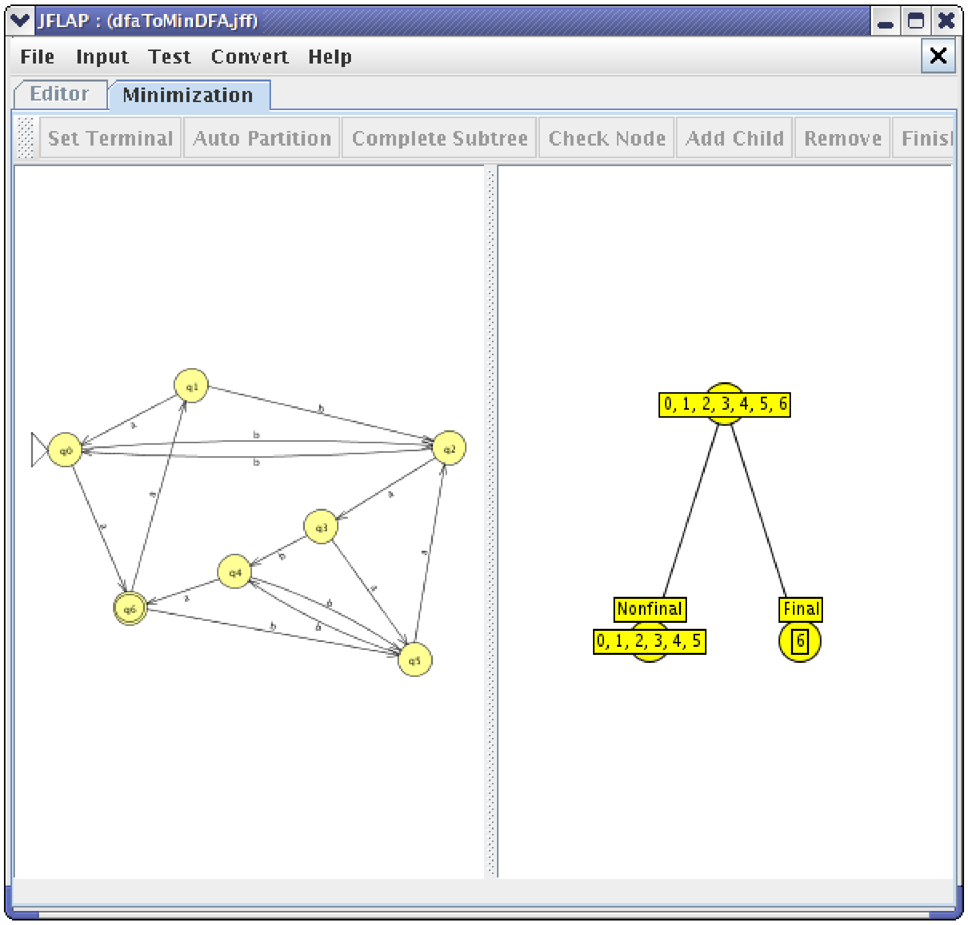

JFLAP : (dfaToMinDFA.jff)
File Input Test Convert Help
Editor Minimization
Set Terminal Auto Partition Complete Subtree Check Node Add Child Remove Finis
q0
q1
q2
q3
q4
q5
q6
a
b
0, 1, 2, 3, 4, 5, 6
Nonfinal
0, 1, 2, 3, 4, 5
Final
6

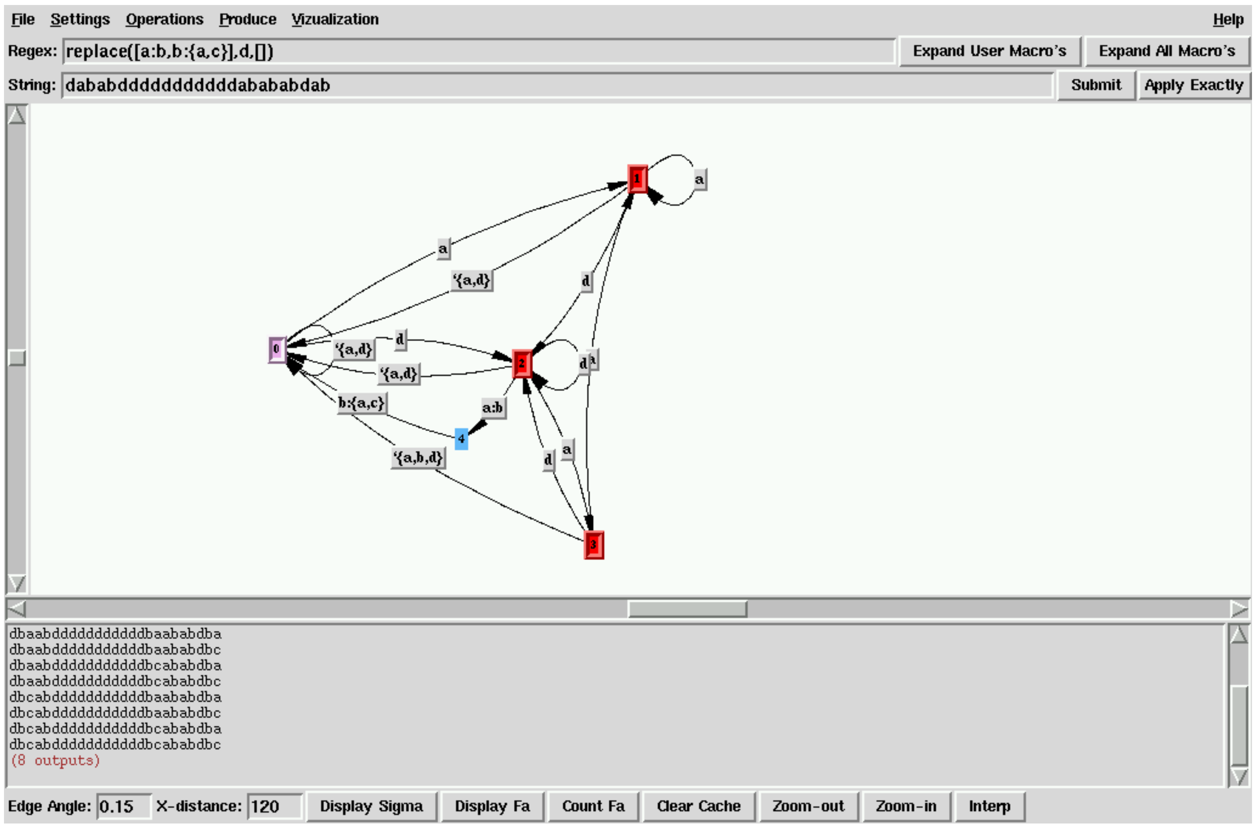
Figure 2. State minimization using JFLAP.
Source: JFLAP Version 7.1 2018

JFLAP also supports DFA equivalence testing, which can be done either in code or using the GUI. Equivalence testing is implemented by minimizing the two DFAs being compared and then doing a basic recursive isomorphism check on them to find indistinguishable states. Since the minimal DFA for a language is unique, the algorithm simply tries to find a mapping for each state of one minimal DFA's states to the other minimal DFA's states and returns whether all states have a counterpart. This algorithm "runs rapidly enough to work for typical school assignments containing only a handful of states, although ... faster methods are available to handle large automata", which JFLAP does not use (Norton 2009). JFLAP does not provide witness strings to indicate why two automata are different.

In 2009, Norton authored her M.S. Project "Algorithms for Testing Equivalence of Finite Automata, with a Grading Tool for JFLAP" in which she implemented two additional state minimization and equivalence testing using JFLAP, the Hopcroft *n lg n* algorithm (described in section 2.3.2), and the (Nearly) Linear Algorithm (described in section 2.3.3). These algorithms have significantly faster run times than the existing JFLAP implementation and can be used for minimizing and equivalence checking large automata. Additionally, Norton's implementation the algorithms provide a witness string when doing equivalence checking between DFAs.

The Finite State Automata Utilities Toolbox (FSAU)

The Finite State Automata Utilities Toolbox (FSAU) is software written in Prolog that supports working with regular expressions, finite state automata and finite state transducers. On the FSA Utilities toolbox website it is marketed as "a collection of utilities" for use in Automata Theory (van Noord 2020). It can be used as APIs that can be programmed against or alternatively as a GUI program similar to JFLAP.



Regex: replace([a:b,b:{a,c}],d,[])

String: dababdddddddddddabababdab

```
dbaabdddddddddddbaababdba
dbaabdddddddddddbaababdbc
dbaabdddddddddddbcababdba
dbaabdddddddddddbcababdbc
dbcabdddddddddddbaababdba
dbcabdddddddddddbaababdbc
dbcabdddddddddddbcababdba
dbcabdddddddddddbcababdbc
(8 outputs)
```

Edge Angle: 0.15 X-distance: 120

Figure 3. Example of using the FSAU toolkit GUI.
Source: van Noord 2020

FSAU was last updated in 2006. Since it is written in Prolog, which is not cross-platform, users must either download a pre-built binary or download the sources and compile FSAU themselves (*ibid.*). The binaries provided on FSAU's official website are aged and the Windows ones specifically do not work on modern Windows 10, thus users are forced to compile the program locally.

FSAU provides a variety of methods for manipulating automata. For state minimization it offers two different algorithm implementations, though neither are visualized. The algorithms are The *n lg n* Hopcroft Algorithm (described in section 2.3.2.) and the Brzozowski Reversing Minimization Algorithm. The Reversing Minimization Algorithm is a very inefficient algorithm, having exponential worst-case time complexity (Norton 2009). The state minimization facilities are difficult to use since users must know Prolog and program using specific Prolog APIs to apply the algorithms to their DFAs.

FSAU does not support equivalence checking between two DFAs, though the implemented algorithms can be extended to provide this since sources for FSAU are available.

The Finite State Automaton Applet (FSAA)

The Finite State Automaton Applet (FSAA) is a Java applet for simulating DFA runs. FSAA is basic in its functionality, it provides no ways to manipulate DFAs (therefore no minimization or equivalence testing). FSAA's only functionality is creating DFAs in a GUI and testing custom string inputs on them to see whether they are accepted or rejected.

FSAA was last updated in 2000. The underlying Java Applet technology it uses has been deprecated, removed from the Java Language Specification, and browsers have removed support for it as well (Oracle 2018). In 2020, FSAA can only be run in custom applications made for legacy software.

jFAST

The Java Finite Automata Simulation Tool (jFast) is educational Java software intended to be used "particularly for less advanced computer science students". In the research paper outlining the creating of jFast, it is described as "as an easy-to-use, easy-to-learn software tool for teachers and students for discovery and exploration of finite state machines". jFast was designed to be complementary to JFLAP, acting as a simplified version of it. (White, Way 2006)

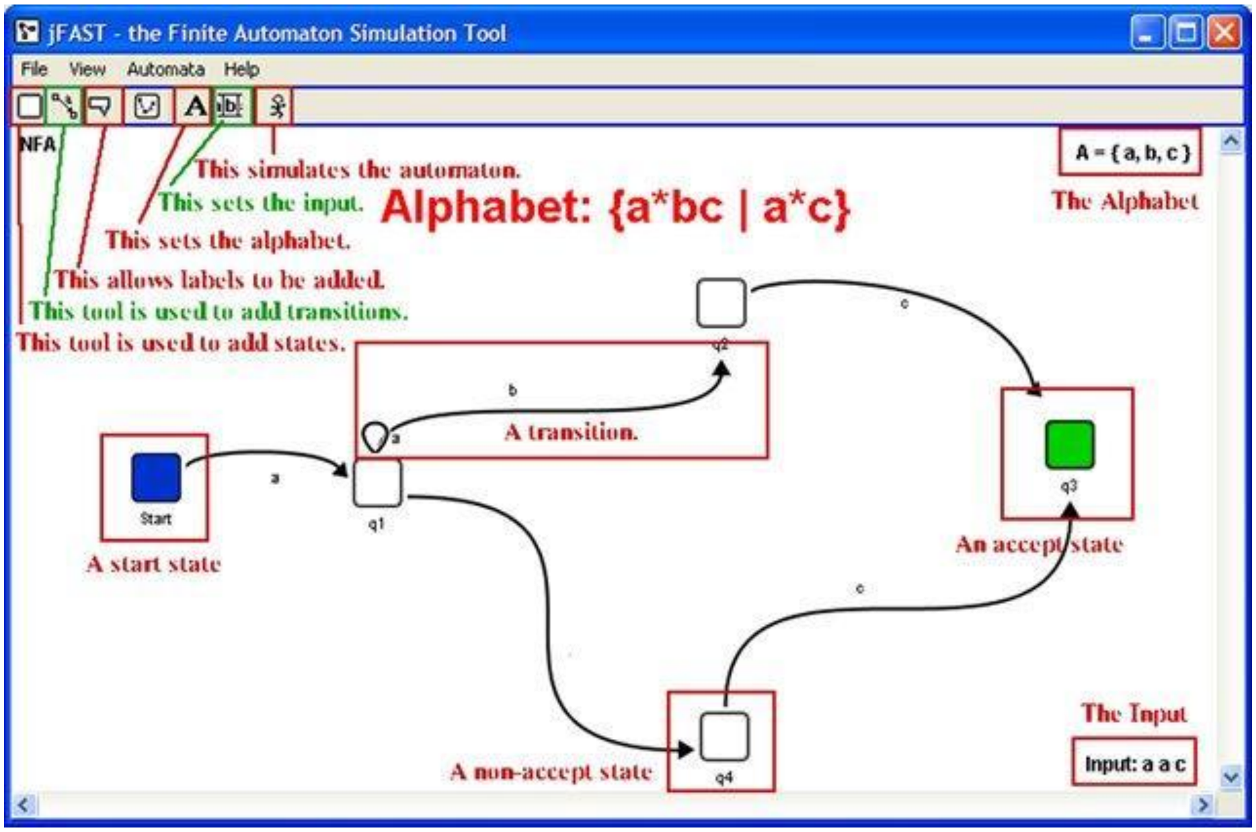


Figure 4. Overview of jFast GUI.
Source: White, Way 2006

jFast supports creating, editing, and simulating various finite automata like DFAs, non-deterministic finite automata (NFAs), and pushdown automata (PDAs), however it does not do state minimization or equivalence testing. jFast was last updated in 2006.

2.2.2. Web Applications

Besides native applications, there are many web-based toolkits and applications for automata theory that do not require installation or compilation and instead can be run in users' browsers.

While the native applications described in section 2.2.1. usually have some kind of academic background or research paper outlining their creation, the most popular web applications for DFAs are instead hobby projects of casual developers. Because of this, most of these applications were developed informally and quickly and were abandoned just as fast, as seems to be the case in the following examples.

In contrast to native applications, the webapps described often provide less functionality but do so with more aesthetics. More specifically, they usually have a more minimal and modern look and feel.

Noam & FSM simulator (FSMS)

Noam is a JavaScript-based library used for simulating and manipulating automata and formal grammars for regular and context-free languages. It provides the necessary tools for working with DFAs, NFAs, and PDAs.

Noam was last updated in 2017 and its Git repository describes it as being in pre-alpha status with a long to-do list of missing features (Zuzak, Budiselic 2017). Despite this, Noam's authors have created many web applications using the current implementation.

The Noam-based app most relevant to this project is FSM Simulator (FSMS). It is a web application where users can either input a regular expression or data about a finite automaton and the app will generate a graph of the automaton and users can test whether it accepts a custom input or not. FSMS does not offer any DFA manipulation in the way of minimization or equivalence testing.

FSMS has modern, accessible, and minimalistic graph visualizations of DFAs, and for this reason, the example DFAs used in this project are visualized using FSMS.

The Noam library could potentially be used for implementing the algorithms chosen for this project, but considering that Noam was last updated in 2017 and has thousands of lines of code that are irrelevant to this project (regular expressions, DFAs, PDAs, *etc.*), a better alternative is needed.

UC Davis Automaton Simulator (UCDAS)

Automaton Simulator (UCDAS) is software written in Elm used for teaching and grading the Theory of Computing module at University of California, Davis (Doty 2020).

Automaton Simulator provides a modern editor for creating and simulating DFAs, NFAs, and Turing Machines (TMs), including small pre-generated examples. For DFAs, UCDAS provides computation and state minimization functionality. There is almost no visualization in UCDAS, in the case of state minimization it simply replaces an existing DFA description with a minimized one without showing any work done or saying which algorithm is used.

UCDAS does not have original sources available, instead using preprocessed minified JavaScript files as sources. These are near-impossible to decipher and thus there is no way to know what state minimization algorithm is used and whether it is even correctly implemented.

Automaton Simulator (AS)

Automaton Simulator (AS) is described in its source repository as a “website that simulates various finite state machines: DFA, NFA, PDA” (Dickerson 2018). Automaton Simulator provides a GUI to create automata from graphs and functionality to test either individual inputs on the automaton or to test many inputs in bulk and reports the results.

AS has three different example automata users can use to get started and users can additionally save and load automata using external files. AS does not support DFA state minimization or equivalence testing.

2.2.3. Comparison and Learnings

The qualitative analysis of existing systems can be found summarized in Table 2 below, with columns representing the capabilities of the systems. The columns were chosen based on the necessary functionality of the software produced in this project (state minimization, equivalence testing, save/load functionality, visualization) outlined in chapter 4 (Requirements) as well as an extra column for DFA computations to highlight that this project is not concerned with computations, whereas all existing systems do support it.

The ’Last updated’ column was included to highlight that most of these projects are at best outdated and at worst deprecated and unmaintained. Because of this, when using any of the existing toolkits/utilities for this project, it would result in upstream dependency bugs being difficult or even impossible to fix.

An extra row was added to the table signifying the intended software outcome of this project. The only project that already provides the necessary functionality is JFLAP, but JFLAP is a native application so it can not be applied to web contexts. Additionally, JFLAP is too general in that it implements many things not needed for this project’s outcomes.

Of existing web-based projects, Noam would be the closest fit to achieving this project’s aims as it is a toolkit that can be extended with state minimization and equivalence testing functionality. However, Noam is unmaintained and has been in pre-alpha status for more than two years and it contains extra functionality (NFAs, PDAs) not necessary for DFA equivalence testing that would result in a bloated application.

Based on the previous analysis, a custom software solution will be needed to achieve the Aims and Objectives outlined in section 1.1.

Tool	Last updated	State minimization	Equivalence testing	DFA run simulation	Visualization	Save and load

JFLAP	2018	✓	✓	✓	extensive	✓
FSAU	2006	✓	X	✓	extensive	✓
FSAA	2000	X	X	✓	basic	✓
jFast	2006	X	X	✓	extensive	✓
FSMS	2017	X	X	✓	basic	X
UCDAS	2019	✓	X	✓	none	✓
AS	2018	X	X	✓	basic	✓
Proposed solution	2021	✓	✓	X	extensive	✓

Table 2. Comparison of existing finite automata systems with proposed software solution of this project.

2.3. Algorithms

2.3.1. The Table-Filling Algorithm

The Table-Filling Algorithm (also called Moore's Algorithm) was devised in 1956 by Edward F. Moore for DFA minimization. It works by identifying all pairs of indistinguishable states of a DFA and combining them into a single state. (Moore 1956)

Although it was devised for DFA minimization, it can be extended to test equivalence of two DFAs by combining them into a single automaton, identifying the indistinguishable states of that automaton, and checking whether the states corresponding to the original DFAs' starting states have been marked as distinguishable or not.

The following description of the algorithm is based on Norton's 2009 paper and the Hopcroft *et al.* 2006 textbook.

The algorithm works by assuming all states of a DFA are indistinguishable, identifying some pairs of distinguishable states and using these pairs to discover new distinguishable pairs (since reaching distinguishable states from as-of-yet indistinguishable ones means the states are actually distinguishable) until all possible pairs have been discovered. The first pairs that are marked as distinguishable are all those in which one state is a final state and the other is not, since in those cases one state will accept the empty string ε and the other will not.

The algorithm is called "table-filling" since the pairs of states being compared are stored in a table that is progressively filled in as new distinguishable pairs are identified.

Consider the following application of the Table-Filling Algorithm: testing the equivalence of the DFA in Figure 5, which accepts the language of 01^*, and the DFA in Figure 6, which accepts the language 01.

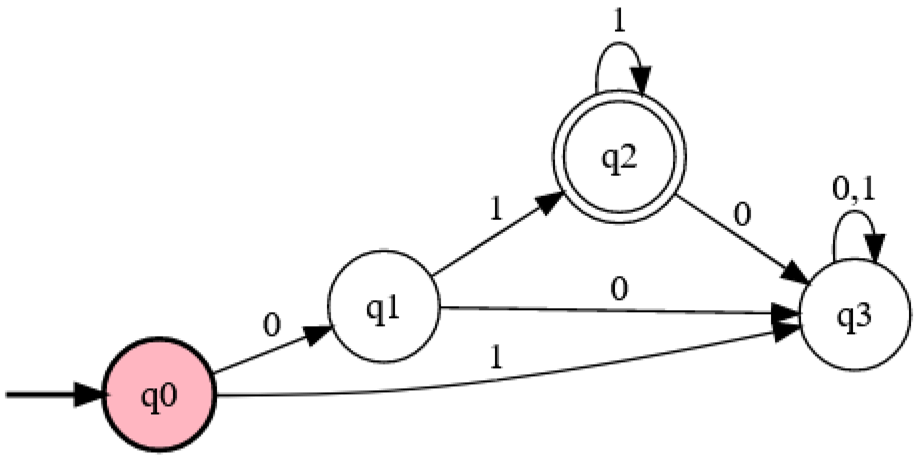


Figure 5. A DFA accepting the language of 01.*
Source: Zuzak, Jankovic 2017

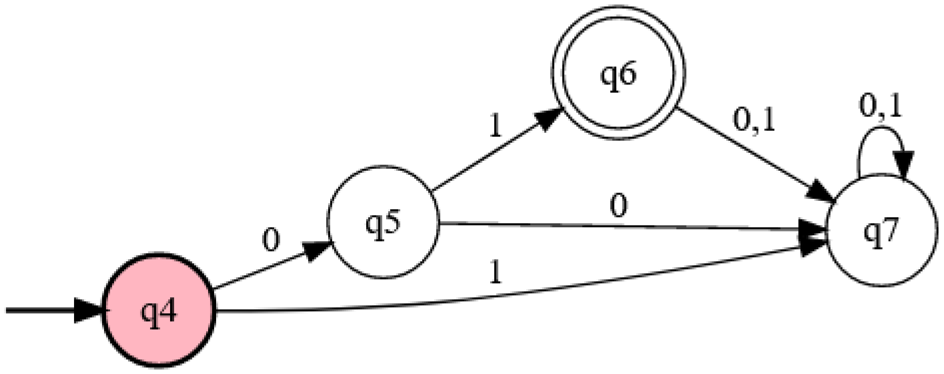


Figure 6. A DFA accepting the language of 01.
Source: Zuzak, Jankovic 2017

First, we construct a table listing all the states in the two automata in both the rows and columns of the table. The cells of the table represent pairs of states that will marked by the algorithm. Only the lower half of the table will be used since pairs need to be compared only once, otherwise each pair would be duplicated. Initially all states are assumed to be indistinguishable so the table is empty.

q1							
q2	X	X					
q3			X				
q4			X				
q5			X				
q6	X	X		X	X	X	
q7			X				X
	q0	q1	q2	q3	q4	q5	q6

Figure 7. Table after distinguishing initial pairs in The Table-Filling Algorithm.

To generate initial distinguishable states, all pairs where one state is an accepting state and the other is not are marked as distinguishable. In this example, $q2$ and $q6$ are the only accepting states so $(q2, q0)$, $(q6, q0)$, $(q2, q1)$, $(q6, q1)$, *etc.* (all pairs containing $q2$ or $q6$ except for $(q2, q6)$) will be marked as distinguishable, represented by an X in the table. The state of the table after doing this is visualized in Figure 7.

Next, we will loop through all unmarked pairs of states. For each pair, consisting of states p and q, we will consider which states they can transition to for each symbol in the alphabet Σ. If there exists some symbol $a \in \Sigma$ such that $\delta(p, a) = p'$ and $\delta(q, a) = q'$, where p' and q' are distinguishable states (with a witness string w), then p and q must also be distinguishable (with a witness string aw).

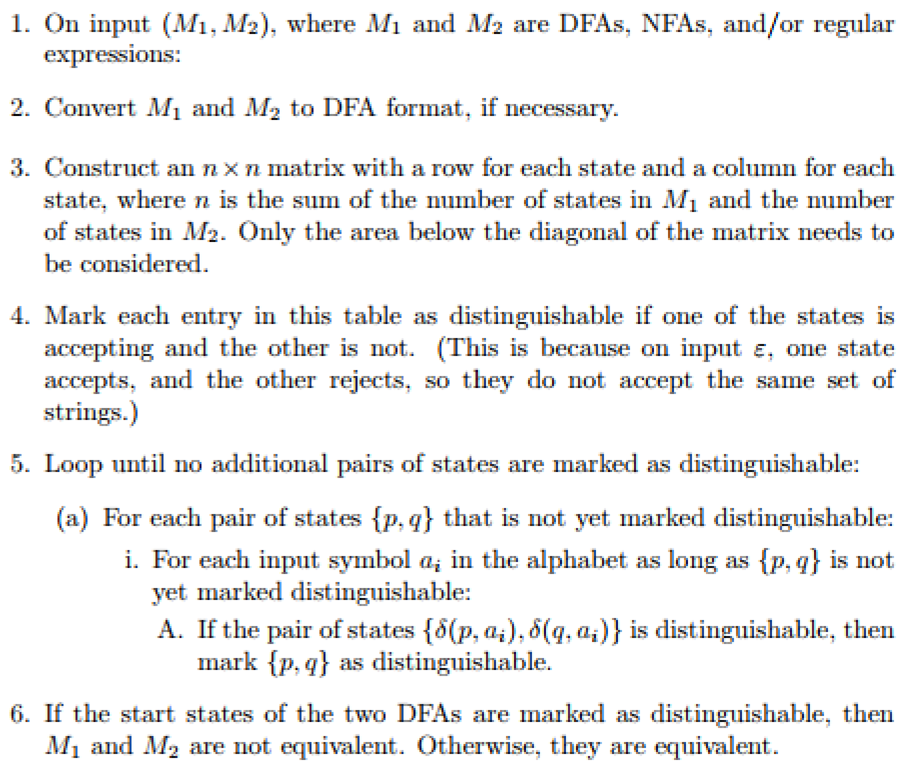
In our example, we will start with the pair $(q1, q0)$. For input 0, $\delta(q1, 0) = q3$ and $\delta(q0, 0) = q1$. The pair $(q3, q1)$ has not been marked as distinguishable, therefore we will do nothing yet. For input 1, $\delta(q1, 1) = q2$ and $\delta(q0, 1) = q3$. The pair $(q2, q3)$ has been already marked, so $(q1, q0)$ is also marked. Next, we look at $(q3, q0)$. $\delta(q3, 0) = q3$, $\delta(q0, 0) = q1$, $\delta(q3, 1) = q3$, $\delta(q0, 1) = q3$. The pair $(q3, q1)$ has not been marked and $(q3, q3)$ can never be distinguishable, because a state can never be distinguishable with itself (because of this, these symmetric pairs are not even in the table). Therefore, $(q3, q0)$ will not be marked. This process continues for all unmarked pairs, the next pairs being marked during this iteration being $(q5, q0)$, $(q3, q1)$, $(q4, q1)$, $(q7, q1)$, $(q6, q2)$, $(q5, q3)$, $(q5, q4)$, $(q7, q5)$. The state of the table after the first iteration is visualized in Figure 8.

	q0	q1	q2	q3	q4	q5	q6
q1	X						
q2	X	X					
q3		X	X				
q4		X	X				
q5	X		X	X	X		
q6	X	X	X	X	X	X	
q7		X	X			X	X

Figure 8. Table after first iteration of The Table-Filling Algorithm.

Since we discovered new distinguishable pairs during the first iteration, we must re-check the previous pairs that were not marked before, because the pairs they transition to might now be distinguishable. During the second iteration, the pairs $(q3, q0)$, $(q7, q0)$, $(q5, q1)$, $(q4, q3)$, $(q7, q4)$ are marked. We marked more pairs during this iteration, so a third iteration must also be run. This time a single pair $(q4, q0)$ is marked. Notice that $q4$ and $q0$ are the starting states of the DFAs being compared and since they are distinguishable, the DFAs are, by definition, non-equivalent. The algorithm could stop execution now as an optimization, but for completion a fourth and final iteration is run where no new pairs are marked.

Pseudocode for The Table Filling Algorithm is shown in Figure 9.



```
1. On input $(M_1, M_2)$, where $M_1$ and $M_2$ are DFAs, NFAs, and/or regular
   expressions:
2. Convert $M_1$ and $M_2$ to DFA format, if necessary.
3. Construct an $n \times n$ matrix with a row for each state and a column for each
   state, where $n$ is the sum of the number of states in $M_1$ and the number
   of states in $M_2$. Only the area below the diagonal of the matrix needs to
   be considered.
4. Mark each entry in this table as distinguishable if one of the states is
   accepting and the other is not. (This is because on input $\varepsilon$, one state
   accepts, and the other rejects, so they do not accept the same set of
   strings.)
5. Loop until no additional pairs of states are marked as distinguishable:
    (a) For each pair of states $\{p, q\}$ that is not yet marked distinguishable:
        i. For each input symbol $a_i$ in the alphabet as long as $\{p, q\}$ is not
           yet marked distinguishable:
            A. If the pair of states $\{\delta(p, a_i), \delta(q, a_i)\}$ is distinguishable, then
               mark $\{p, q\}$ as distinguishable.
6. If the start states of the two DFAs are marked as distinguishable, then
   $M_1$ and $M_2$ are not equivalent. Otherwise, they are equivalent.
```

Figure 9. Pseudocode for The Table-Filling Algorithm.
Source: Norton 2009

The Table-Filling Algorithm has a worst-case time complexity of $O(n^4)$, where n is the sum of the number of states in the two DFAs being compared. If there are n states, then the number of pairs is $n * (n - 1)/2$ so simply looping through all pairs would take $O(n^2)$ time. In the worst case, only a single pair is marked as distinguishable in every iteration of the loop in step 5 of the algorithm, and the pair containing the starting states of the DFAs is the final pair that is distinguished. In this case, the loop will be executed $O(n^2)$ times, and since iterating also takes $O(n^2)$, the worst-case complexity is the result of the multiplication of these two complexities, $O(n^4)$.

Witness variation

In its original form, the Table-Filling Algorithm does not produce a witness string that indicates how two DFAs are non-equivalent. With a simple modification of keeping track of which input symbol distinguishes states and storing that in the table (instead of 'X' like in the previous examples), a witness string can be constructed.

The modified algorithm starts out by differentiating pairs that contain an accepting and non-accepting state, as before. However, this time the symbol distinguishing them (the empty string ε) is marked in the table instead. After that all unmarked pairs are looped through, as before.

In our example, when starting with the pair $(q1, q0)$, again, $\delta(q1, 0) = q3$ and $\delta(q0, 0) = q1$. The pair $(q3, q1)$ has not been marked yet, therefore we will do nothing. For input 1, $\delta(q1, 1) = q2$ and $\delta(q0, 1) = q3$. The pair $(q2, q3)$ has been already marked, so $(q1, q0)$ must be also marked with the symbol distinguishing them, in this case 1.

The algorithm continues as before, with the same pairs being marked each iteration, though being differentiated with the symbols distinguishing them. The state of the table after completion of the algorithm is represented in Figure 10.

	q0	q1	q2	q3	q4	q5	q6
q1	1						
q2	ε	ε					
q3	0	1	ε				
q4	0	1	ε	0			
q5	1	1	ε	1	1		
q6	ε	ε	1	ε	ε	ε	
q7	0	1	ε		0	1	ε

Figure 10. Table after finishing The Table-Filling Algorithm with witness.

Using this table, we will construct a witness string as follows: for the starting states of the DFAs, $q0$ and $q4$, look up the input symbol distinguishing them and append it to the empty witness string. After this, follow the transition from both states on that symbol, *i.e.* compute $\delta(q0,0) = q1$ and $\delta(q4,0) = q5$. Now, look up the symbol distinguishing $q1$ and $q5$, append it to the witness string, and follow the transitions from $q1$ and $q5$ on the symbol. Doing this we have a witness string of 01 and reach states $q2$ and $q6$. This process continues until one of states reached is an accepting state and the other is not, since this means the one automaton will accept the string while the other will reject it. $q2$ and $q6$ are distinguished by 1, $\delta(q2,1) = q2$ and $\delta(q6,1) = q7$. Since $q2$ is an accepting state and $q7$ is not, we now have a complete witness string of 011. Indeed, the first DFA accepts it, and the second one does not.

The time complexity of this algorithm is still $O(n^4)$ since the only extra steps done are storing the distinguishing symbols in the table (which does not impact run time) and constructing a witness string, which in the worst case will visit every single pair of states once and take $O(n^2)$ time.

2.3.2. The *n lg n* Hopcroft Algorithm

The *n lg n* Hopcroft Algorithm was devised by John Hopcroft as an algorithm for minimizing DFAs, but it can also be extended to DFA equivalence testing. The algorithm works by partitioning states into blocks, where all states inside a block are indistinguishable from others inside it, but distinguishable with all states in other blocks. These blocks are then split into smaller ones until all indistinguishable blocks have been discovered. (Hopcroft 1971).

Equivalence testing using the algorithm works by combining all states of two DFAs being compared into a single set of states, running the algorithm, and seeing if the starting states of the DFAs end up in the same block (meaning they are indistinguishable) or not.

The following description of The *n lg n* Hopcroft Algorithm is based on Norton's 2009 paper and Hopcroft's original 1971 specification of the algorithm.

At the start of the algorithm, an inverse transition function of the DFAs is constructed. This function $\delta^{-1}(s, a)$ returns the set of states which transition to a state s on the input symbol a. This function is stored in the form a table.

Secondly, for each symbol a in the alphabet, an initially empty to-do list (implemented as an unordered set) L_a is constructed.

Then the initial distinguishable blocks are created. We can identify two distinct sets of states from all states of the DFAs – those which are final states – and those which are not. These sets can be distinguished on the empty string ε. The two blocks are marked as $B_1 = F$ and $B_2 = S - F$, where S is the set of all states.

Now, for each input symbol a, for each block B_i, using the inverse transition function we construct a set $a_a(i)$ of the states in B_i that have predecessors on input a. Using these sets, for each input symbol a, we choose the set that has the least amount of states with predecessors (breaking ties by picking the lower-numbered set) $a_a(n)$, and add n to the to-do list L_a.

These to-do lists are used for splitting blocks into smaller ones. We will start by picking a to-do list L_a and removing an element n from it. n indicates which block's predecessors will be split. For all predecessors of states in block B_n, that is, states s for which $\delta(s, a) \in a_a(n)$, we will mark the block B_i that contains s as a block to be split. Additionally, we will add s to the set B'_i, creating it if necessary. After marking blocks for every predecessor of B_n and adding them to B_i', we are ready to start splitting blocks.

For each block B_i to be split, we will check if $|B_i'| < B_i$. If that is the case, then there exist some states in B_i that transition to block B_n on input a and some states which do not. These two groups of states can be distinguished on the input a since the blocks they transition to are themselves distinguishable. We split B_i into two blocks, B_i', and all the other states in the block, $B_i - B_i'$. For these new blocks, we will once again calculate $a_a(i)$ for each symbol a in the alphabet and add the index i of the smaller non-empty set to the processing list L_a.

This process of removing elements from the to-do list and splitting blocks repeats until all to-do lists are empty. By that point all distinguishable blocks have been found and we can determine the equivalence of two DFAs by checking whether their starting states are in the same block.

A more detailed explanation of this algorithm is available in David Gries's paper "Describing an Algorithm by Hopcroft" which walks through its application step-by-step (Gries 1972).

We will explore an application of the *n lg n* Hopcroft algorithm using the same example as for the Table-Filling algorithm, determining the equivalence of DFAs from Figure 5 and 6.

Initially, we will construct the inverse transition function. It is visualized as a table in Figure 11.

	0	1
q0	-	-
q1	q0	-
q2	-	q1, q2
q3	q1, q2, q3	q0, q3
q4	-	-
q5	q4	-
q6	-	q5
q7	q5, q6, q7	q4, q6, q7

Figure 11. Inverse transition function for DFAs in Figures 5 and 6.

After that we create the two initial blocks of accepting states and non-accepting ones. $B_1 = \{q2, q6\}$ and $B_2 = \{q0, q1, q3, q4, q5, q7\}$. For each symbol in the alphabet $a \in \Sigma$ (in this example, 0 and 1), we will create the sets $a_a(i)$ indicating which states in block i have predecessors on input a. $a_0(1) = \{\}$, $a_0(2) = \{q1, q3, q5, q7\}$, $a_1(1) = \{q2, q6\}$, and $a_1(2) = \{q3, q7\}$.

Now, we can create the to-do lists. Each to-do list is initialized containing the number of the block that has the least amount of states with predecessors on the input a. Since $|a_0(1)| = 0$, and $|a_0(2)| = 4$, we will initialize the to-do list for 0 with the block number 1, $L_0 = \{1\}$. For input

1, $|a_1(1)| = 2$ and $|a_1(2)| = 2$. Since we have to break the tie, we choose the block with the lower number, so $L_1 = \{1\}$.

Now we will start marking blocks to be split based on the to-do lists. L_0 only contains a single element to process, the number 1. We remove this from L_0 so now $L_0 = \{\}$. Using the inverse transition function, we see that no states lead to $q2$ or $q6$ (the contents of block 1) on the input 0, so there are no predecessors to split. We move on to the next to-do list L_1.

L_1 contains a single element, the number 1. We remove it from L_1 and try to split the predecessors of block 1 again. On the input 1, it is possible to get to states $q2$ and $q6$ from $q1$, $q2$, and $q5$. For each of the states, we mark the block B_i they are currently in and add them to a new block B_i'. After doing this, we have marked both B_1 and B_2 and created $B_1' = \{q2\}$ and $B_2' = \{q1, q5\}$.

Now we must split B_1 and B_2 into two blocks each. Starting with B_2, we can split it because we have shown that it is possible to get to B_1 from the states in B_2', but not from the other states in B_2, therefore these two blocks can be distinguished.

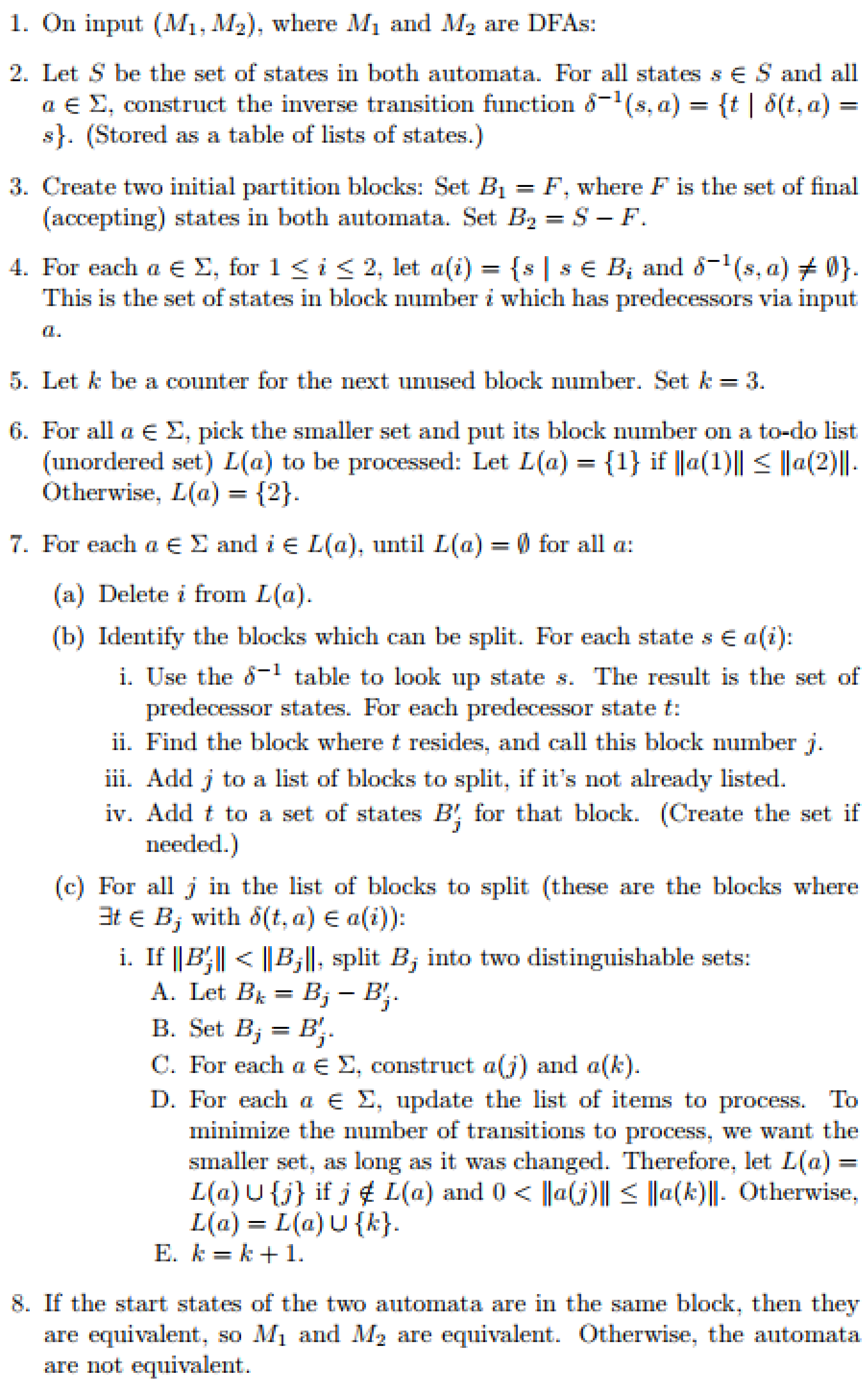
We will define $B_3 = B_2 - B_2' = \{q0, q3, q4, q7\}$ and $B_2 = B_2' = \{q1, q5\}$. Since we changed our blocks and, we need to update the sets $a_a(i)$. $a_0(2) = \{q1, q5\}$, and $a_0(3) = \{q3, q7\}$. Using the new sets, we will update the to-do list for input 0 with the number of the smaller non-empty block (in this case we have a tie again, so we choose the block with smaller number) $L_0 = \{2\}$. Doing the same for input 1, $a_1(2) = \{\}$ and $a_1(3) = \{q3, q7\}$. We now update $L_1 = \{3\}$, because $a_1(2)$ is empty and no new blocks can be created from its predecessors.

Doing the same splitting for B_1, we get $B_4 = \{q6\}$ and $B_1 = \{q2\}$. After updating $a_a(n)$ with the new blocks, we have $a_0(1) = \{\}$, $a_0(4) = \{\}$, $a_1(1) = \{q2\}$, $a_1(4) = \{q6\}$. Based on this, we update our to-do lists such that $L_0 = \{2,4\}$ and $L_1 = \{3,1\}$.

Continuing with the same process, the next blocks to be split are predecessors of B_1 on the input 0. This results in B_3 being split into $B_3 = \{q0, q4\}$ and $B_5 = \{q3, q7\}$. After this, splitting the predecessors of B_1 on 1 will result in the blocks $B_2 = \{q1\}$ and $B_6 = \{q5\}$. Finally, after splitting the predecessors of B_2 on input 0, we will get $B_3 = \{q0\}$ and $B_7 = \{q4\}$.

After this no more blocks are split, and the to-do lists will eventually be empty. Since $q0$ and $q4$ are in different blocks (B_3 and B_7) the two automata being compared are not equivalent.

Pseudocode for the *n lg n* Hopcroft algorithm is outlined in Figure 12.



```
1. On input $(M_1, M_2)$, where $M_1$ and $M_2$ are DFAs:
2. Let $S$ be the set of states in both automata. For all states $s \in S$ and all
   $a \in \Sigma$, construct the inverse transition function $\delta^{-1}(s,a) = \{t \mid \delta(t,a) = s\}$.
   (Stored as a table of lists of states.)
3. Create two initial partition blocks: Set $B_1 = F$, where $F$ is the set of final
   (accepting) states in both automata. Set $B_2 = S - F$.
4. For each $a \in \Sigma$, for $1 \le i \le 2$, let $a(i) = \{s \mid s \in B_i \text{ and } \delta^{-1}(s,a) \neq \emptyset\}$.
   This is the set of states in block number $i$ which has predecessors via input $a$.
5. Let $k$ be a counter for the next unused block number. Set $k = 3$.
6. For all $a \in \Sigma$, pick the smaller set and put its block number on a to-do list
   (unordered set) $L(a)$ to be processed: Let $L(a) = \{1\}$ if $\|a(1)\| \le \|a(2)\|$.
   Otherwise, $L(a) = \{2\}$.
7. For each $a \in \Sigma$ and $i \in L(a)$, until $L(a) = \emptyset$ for all $a$:
   (a) Delete $i$ from $L(a)$.
   (b) Identify the blocks which can be split. For each state $s \in a(i)$:
       i. Use the $\delta^{-1}$ table to look up state $s$. The result is the set of
          predecessor states. For each predecessor state $t$:
       ii. Find the block where $t$ resides, and call this block number $j$.
       iii. Add $j$ to a list of blocks to split, if it's not already listed.
       iv. Add $t$ to a set of states $B'_j$ for that block. (Create the set if
           needed.)
   (c) For all $j$ in the list of blocks to split (these are the blocks where
       $\exists t \in B_j$ with $\delta(t,a) \in a(i)$):
       i. If $\|B'_j\| < \|B_j\|$, split $B_j$ into two distinguishable sets:
          A. Let $B_k = B_j - B'_j$.
          B. Set $B_j = B'_j$.
          C. For each $a \in \Sigma$, construct $a(j)$ and $a(k)$.
          D. For each $a \in \Sigma$, update the list of items to process. To
             minimize the number of transitions to process, we want the
             smaller set, as long as it was changed. Therefore, let $L(a) =
             L(a) \cup \{j\}$ if $j \notin L(a)$ and $0 < \|a(j)\| \le \|a(k)\|$. Otherwise,
             $L(a) = L(a) \cup \{k\}$.
          E. $k = k + 1$.
8. If the start states of the two automata are in the same block, then they
   are equivalent, so $M_1$ and $M_2$ are equivalent. Otherwise, the automata
   are not equivalent.
```

Figure 12. Pseudocode for the n lg n Hopcroft Algorithm.
Source: Norton 2009

As its name implies, the *n lg n* Hopcroft algorithm has a worst-case time complexity of $O(n \lg n)$, where n is the sum of the number of states in the two DFAs being compared. All the initialization steps 1-6 in the pseudocode run in $O(n)$. The loop in step 7 runs a maximum of n times, with the amount of work done in an iteration being $O(\lg n)$. A detailed breakdown of the algorithm's runtime can be found in Gries's 1972 article (Gries 1972).

Witness variation

The *n lg n* Hopcroft algorithm can be modified to produce a witness string in a similar vein to how The Table-Filling Algorithm was modified. A table is constructed where the rows of the table represent all states in one DFA and all columns represent states in the other.

Whenever a block B_i is split into two blocks B_i' and $B_i - B_i'$ based on the symbol a, for all pairs of states p and q, in which one state will be placed in B_i and the other $B_i - B_i'$, we will mark the cell in the table representing the pair with the symbol a.

After the initial states of the DFAs have been distinguished, constructing a witness string is possible. Initially, we must look up the symbol a distinguishing the two starting states of the DFAs and append it to the witness string. Then we must calculate the transition for those states on a and check if the output for the pair consists of an accepting and non-accepting state. If it does not, we must again look up which symbol distinguishes the pair and append it to the witness and transition to the next states. This process continues until we reach a point where one of the states reached is accepting and the other is not. The witness string is the transitions that had to be followed to reach these states.

In our example of equivalence testing the DFAs in Figures 5 and 6, when we first split B_2 into $B_3 = \{q0, q3, q4, q7\}$ and $B_2 = \{q1, q5\}$ on the input 1, we would mark the pairs $(q0, q5)$, $(q3, q5)$, $(q4, q1)$, $(q7, q1)$ in the table with a 1. After splitting B_1 on the input 1 into $B_4 = \{q6\}$ and $B_1 = \{q2\}$, we would mark the pair $(q2, q6)$ with a 1. After splitting B_3 on 0 into $B_3 = \{q0, q4\}$ and $B_5 = \{q3, q7\}$, we would further mark the pairs $(q0, q7)$ and $(q4, q3)$ with a 0. The next splits are B_2 on the input 0 into $B_2 = \{q1\}$ and $B_6 = \{q5\}$, for which mark the pair $(q1, q5)$, and B_3 on the input 1 into $B_3 = \{q0\}$ and $B_7 = \{q4\}$, after which we can mark the pair of starting states $(q0, q4)$ with 0. The state of the table after this process is shown in Figure 13.

Using the table, we construct the witness string as follows: starting from the pair of starting states $(q0, q4)$, which are distinguished by 0, we follow $\delta(q0,0) = q1$ and $\delta(q4,0) = q5$. After this, we follow the transition distinguishing $(q1, q5)$, marked by 1 in the table. From this we reach $(q2, q6)$, which is distinguished by a 1. After this we reach $(q2, q7)$, where $q2$ is an accepting state, but $q7$ is not. This means we have reached a point where one DFA will accept the witness string and the other will not. The witness string consists of the transitions from the starting states to the end, 011. Indeed, the DFA in Figure 5 accepts the string and the one in Figure 6 does not.

	q4	q5	q6	q7
q0	0	1		0
q1	1	1		1
q2			1	
q3	0	1		

Figure 13. Filled in witness table for comparing DFAs in Figures 5 and 6 using the n lg n Hopcroft Algorithm.

The complexity of the *n lg n* Hopcroft Algorithm with witness is $O(n^2)$. This is because maintaining the witness table takes extra work. A detailed analysis of this is found in Norton's 2009 paper. (Norton 2009)

2.3.3. The (Nearly) Linear Algorithm

The (Nearly) Linear Algorithm was explicitly designed for DFA equivalence testing, unlike The Table-Filling and *n lg n* Hopcroft algorithms, which were intended for state minimization and later extended to do equivalence testing. The (Nearly) Linear Algorithm works by merging states into indistinguishable sets, in contrast to the previous algorithms which work by distinguishing states from initially indistinguishable pairs or blocks. (Hopcroft, Karp 1971)

The following description of The (Nearly) Linear Algorithm is based on Norton's 2009 paper and Hopcroft and Karp's original 1971 specification of the algorithm.

Before any processing starts, all states in the two DFAs being compared are assumed to be distinguishable and each state is placed in a distinct set indicating it is distinguishable from all others. Secondly, a processing stack is created for pairs of states whose successors will be merged.

The merging process starts by assuming the starting states of the DFAs, p and q, are indistinguishable. They are merged into a single indistinguishable set $\{p, q\}$ and the pair (p, q) is added to the processing stack to merge their successors.

After this is done, we pop the same pair (p, q) off the processing stack. Since p and q were assumed to be indistinguishable, then for all inputs a, $\delta(p, a) = p'$ and $\delta(q, a) = q'$ must also be indistinguishable. Therefore, the sets containing them can safely be merged into a single set $\{p', q'\}$, and (p', q') will be added to the processing stack to merge their successors.

Next, we pop (p', q') off the stack and look at their successors for every symbol a, respectively p'' and q'', and merge them into the same set. If it happens that p'' (or symmetrically q'') has already been merged into some other set S, then q'' is merged into S instead, because p'' is indistinguishable with both the contents of S and q'' and therefore q'' must also be indistinguishable with the states in S. Since we merged sets, we must now add a pair of states from the original sets to the processing stack to process their successors. However, this time, we can add any pair consisting of q'' and a state $r \in S$ to the processing stack. The state r is called the representative of S.

The reason we can process (q'', r) is that if p'' has already been merged into S then there is some pair (p'', s) where $s \in S$ that is currently on the stack or has already been processed. Because eventually both (q'', r) and (p'', s) are processed and the successors of p'', q'', r and s are merged into the same set, adding (q'', r) to the stack and processing it will have the same effect as adding (q'', p''). Analogously, if both p'' and q'' are already in sets $p'' \in S'$ and $q'' \in S''$, then representatives of these sets can be added to the stack instead of the pair (p'', q'').

Since initially each state is the only element in its set, all states that have been on the stack will have had their transitions followed at least once, and because we start from the initial states of the DFAs, all reachable states will eventually be processed on the stack. Because all reachable states will be processed, by the end of the processing all sets of states that are indistinguishable are identified.

Using this knowledge, we can determine equivalence of the DFAs, by looking for inconsistencies in the sets. This is similar to a proof by contradiction, where our initial assumption is that the starting states of the DFAs are indistinguishable, and we see if this would result in a situation where accepting and non-accepting states can be found in the same set. Since accepting and non-accepting states are always distinguishable, we must have made a mistake in our assumption and the DFAs being compared cannot be equivalent.

To illustrate the use of The (Nearly) Linear Algorithm, consider the example of determining equivalence of the DFAs in Figures 5 and 6.

Initially, we create sets for each state in the DFAs, $\{q0\}$, $\{q1\}$, $\{q2\}$, $\{q3\}$, $\{q4\}$, $\{q5\}$, $\{q6\}$, $\{q7\}$, and an empty processing stack $S = []$

After this, we assume the starting states of the DFAs, $q0$ and $q4$, to be indistinguishable and merge them into the set $\{q0, q4\}$. The pair $(q0, q4)$ is then pushed to the processing stack.

Now, $(q0, q4)$ is popped off the stack and for each input symbol, sets containing their successors on that symbol are merged. For input 0, $\delta(q0,0) = q1$ and $\delta(q4,0) = q5$. The sets $\{q1\}$ and

$\{q5\}$ are merged into $\{q1, q5\}$ and the pair of representatives from each set is added to the stack. In this case since $\{q1\}$ and $\{q5\}$ both only had one element, these sets must be represented by that single element and $(q1, q5)$ is added to the stack. Analogously, for the input 1, $q1$ and $q4$ will respectively transition to $q3$ and $q7$, which are merged into $\{q3, q7\}$ and $(q3, q7)$ is added to the stack. Now, $S = [(q1, q5), (q3, q7)]$.

We are done processing $(q0, q4)$, so $(q3, q7)$ is popped off the stack. These are both trap states, and on both inputs 0 and 1, they only lead to the same pair $(q3, q7)$. Since these states are already in the same set $\{q3, q7\}$, nothing is merged or added to the stack. The stack is now $S = [(q1, q5)]$.

$(q1, q5)$ is popped off the stack next. On input 0, the states respectively transition to $(q3, q7)$, which are already in the same set. On input 1, $q1$ transitions to $q2$ and $q5$ transitions to $q6$. These are currently in the sets $\{q2\}$ and $\{q6\}$, therefore they must be merged into $\{q2, q6\}$ and the pair $(q2, q6)$ is added to the stack.

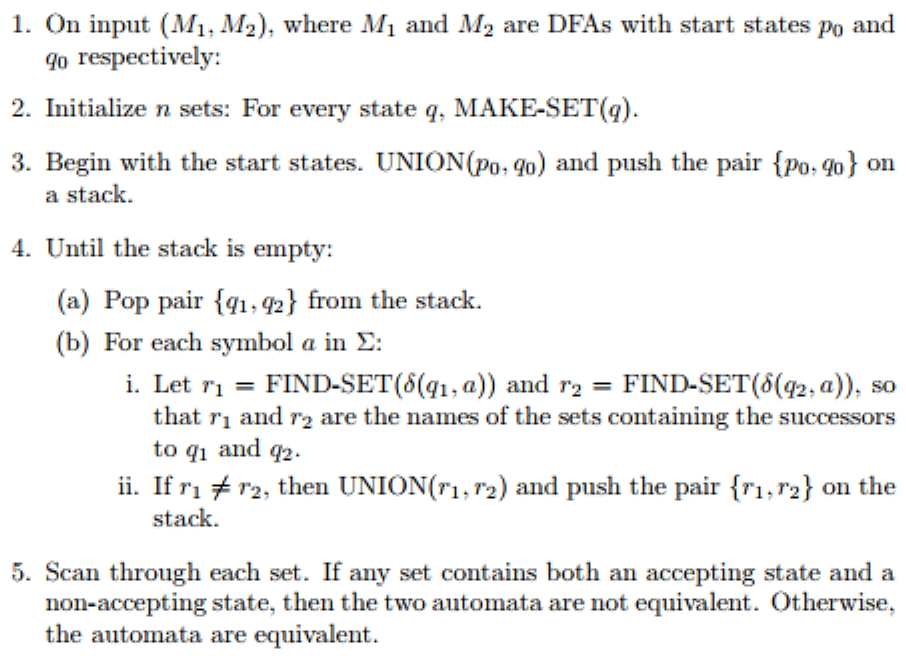
After this, $(q2, q6)$ is popped off the stack. On input 0 they transition to the already merged pair $(q3, q7)$, but on input 1 they respectively transition to states $q2$ and $q7$. The sets containing $q2$ and $q7$, $\{q2, q6\}$ and $\{q3, q7\}$, are distinct, and must be merged into one set $\{q2, q3, q6, q7\}$.

After merging, we must add a pair consisting of the pair of representatives from the original sets to the stack. Since this time both sets contain two elements, we must pick one from each as a representative. We will arbitrarily choose the lowest-numbered state in each set as the representative and add $(q2, q3)$ to the stack.

Finally, $(q2, q3)$ is popped off the stack. On input 0, the states transition to $(q3, q3)$, which are, by definition, already in the same set. On input 1, they transition to $(q2, q3)$, which are also already in the same set. Nothing is added to the stack and it is now empty, $S = []$.

Since the processing stack is now empty, we have now inspected all reachable states and their transitions and have identified which states would be indistinguishable if the starting states were indistinguishable. These sets are $\{q0, q4\}$, $\{q1, q5\}$ and $\{q2, q3, q6, q7\}$. To determine the validity of the initial assumption, we will scan the sets for contradictions, *i.e.* if any of them contain both accepting and non-accepting states. In the first set, both $q0$ and $q4$ are non-accepting. In the second set, both $q1$ and $q5$ are non-accepting. However, in the final set there exist contradictions. For example, it contains the accepting state $q2$ and the non-accepting state $q3$. This means our initial assumption must have been wrong and the two DFAs are not equivalent.

Pseudocode for The (Nearly) Linear Algorithm is outlined in Figure 14.



```
1. On input $(M_1, M_2)$, where $M_1$ and $M_2$ are DFAs with start states $p_0$ and
   $q_0$ respectively:
2. Initialize $n$ sets: For every state $q$, MAKE-SET($q$).
3. Begin with the start states. UNION($p_0, q_0$) and push the pair $\{p_0, q_0\}$ on
   a stack.
4. Until the stack is empty:
   (a) Pop pair $\{q_1, q_2\}$ from the stack.
   (b) For each symbol $a$ in $\Sigma$:
       i. Let $r_1$ = FIND-SET($\delta(q_1, a)$) and $r_2$ = FIND-SET($\delta(q_2, a)$), so
          that $r_1$ and $r_2$ are the names of the sets containing the successors
          to $q_1$ and $q_2$.
       ii. If $r_1 \neq r_2$, then UNION($r_1, r_2$) and push the pair $\{r_1, r_2\}$ on the
           stack.
5. Scan through each set. If any set contains both an accepting state and a
   non-accepting state, then the two automata are not equivalent. Otherwise,
   the automata are equivalent.
```

Figure 14. Pseudocode for The (Nearly) Linear Algorithm.
Source: Norton 2009

The running time of the algorithm is dominated by the loop in step 4, iterations of which are dominated by the FIND-SET and UNION operations, which are applications of the Union-Find algorithm, detailed in Aho, Hopcroft, and Ullman's "The Design and Analysis of Computer Algorithms" (Aho, Hopcroft, Ullman 1974). These operations are bounded by the inverse Ackermann function α, a function that grows so slowly, that for all practical purposes it is constant (Norton 2009). The loop itself is executed a maximum of n times (*ibid.*).

Because of these set operations, The (Nearly) Linear Algorithm has a worst-case time complexity that is nearly linear, $O(n * \alpha(n))$, where n is the sum of the number of states in the two DFAs being compared. Because the time complexity is bounded by α, the algorithm technically does not have linear time complexity, but it is very close to it. (Norton 2009)

Witness variation

The main difference between the witness producing variation of The (Nearly) Linear Algorithm and its original version is that the witness variation keeps track of the symbols on which states are established to be indistinguishable inside a map. Whenever we inspect a pair of states (p, q) and merge their successors (p', q') into a single set, we add an entry in the map consisting of the key (p', q') and the value (p, q, a), where p and q are the predecessors of p' and q' and a is the symbol that is used to transition to p' and q'.

A second difference is that instead of maintaining a processing stack, a FIFO queue is maintained instead. This is done to output the shortest possible witness string. Using the FIFO queue data structure, the search for a contradiction takes the form a breadth-first search (BFS), where pairs of states closest to the starting states are evaluated first and if a contradiction exists, it is found as close to the starting states as possible.

Finally, whenever the sets of some states p and q are merged, the pair (p, q) is enqueued, instead of a pair consisting of representatives of the sets p and q are in. This is done to "strictly and systematically follow both of the DFAs' transition functions … in order to produce an accurate witness" (Norton 2009). After adding this entry to the map, it is checked if one of p and q is an accepting state and the other is not.

If that is the case, we have found the contradiction and can now construct the witness string. To do this we repeat the following process until p and q are the starting states of the DFAs: look up the entry for (p, q) in the map with the value (p', q', a), prepend a to the beginning of the witness string, assign $p = p'$ and $q = q'$ and repeat. By the end of this process we will have witness string that is obtained by following the transitions from the starting states until the point where a contradiction occurs. When this contradiction happens, the starting states are shown to be distinguishable, and the witness can be used to run a computation that proves it.

If the queue is fully processed and no contradiction arises, the two DFAs are equivalent and no witness can be produced.

To illustrate this variation of the algorithm, consider the example of determining equivalence of the DFAs in Figures 5 and 6.

As in the previous example, initially the starting states are merged into one set $\{q0, q4\}$ and the pair $(q0, q4)$ is enqueued in the processing queue.

Next, the same pair is dequeued and its successors are evaluated. On the input 0, the states transition to $(q1, q5)$, which in added to the queue and the sets containing $q1$ and $q5$ are merged into $\{q1, q5\}$. Now, an entry is added to the map consisting of the key $(q1, q5)$ and the value $(q0, q4,0)$. On input 1, $q0$ and $q4$ respectively transition to $q3$ and $q7$, which are merged into the set $\{q3, q7\}$ and the pair $(q3, q7)$ is added to the queue. Again, an entry is added to the map, consisting of the key-value pair $(q3, q7)$: $(q0, q4,1)$. After this, we are done processing the starting states. Currently, for the witness map M and the processing queue Q , $M = \{(q1, q5): (q0, q4,0), (q3, q7): (q0, q4,0)\}$ and $Q = [(q1, q5), (q3, q7)]$.

Now, $(q1, q5)$ is dequeued and its successors are processed. On input 0, the states transition to $(q3, q7)$, which are already in the same set, so nothing is done. On input 1, they transition to

$(q2, q6)$, which means that the sets $\{q2\}$ and $\{q6\}$ are merged into $\{q2, q6\}$ and $(q2, q6)$ is added to the queue. Additionally, a new entry is added in the map, $(q2, q6)$: $(q1, q5, 1)$.

After this, $(q3, q7)$ is dequeued. Since $q3$ and $q7$ are trap states that only lead to themselves, no sets are merged and nothing is added to the queue or map. $Q = [(q2, q6)]$.

Finally, $(q2, q6)$ is dequeued. On input 0, these states again transition to $(q3, q7)$, which are already in the same set. On input 1, these states transition to $(q2, q7)$. Because $q2$ and $q7$ are in different sets, we must merge them into $\{q2, q3, q6, q7\}$ and add $(q2, q7)$ to the queue. We also add the entry $(q2, q7)$: $(q2, q6, 1)$ to the map. Since $q2$ is an accepting state and $q7$ is not, we have found the contradiction and are ready to start witness construction. The map is now $M = = \{(q1, q5): (q0, q4, 0), (q3, q7): (q0, q4, 0), (q2, q6): (q1, q5, 1), (q2, q7): (q2, q6, 1)\}$.

To start constructing the witness, we will assign p and q to be the states which were reached to create the contradiction, $p = q2$ and $q7$. We look up the entry $(q2, q7)$ in the map and get the value $(q2, q6, 1)$. Using this value, we add 1 to the beginning of the witness string and assign $p = q2, q = q6$. Now, we look up $(q2, q6)$ in the map and get $(q1, q5, 1)$. Again, we prepend 1 to the witness and assign $p = q1, q = q5$. Finally, we look up $(q1, q5)$ and get $(q0, q4, 0)$. We prepend 0 to the witness so it is now 011, and assign $p = q0, q = q4$. Since these are the starting states of the DFAs, we are done constructing the witness and output 011.

Indeed, the DFA in Figure 5 accepts the string, but the DFA in Figure 6 rejects it.

The time complexity of the witness variation of The (Nearly) Linear Algorithm is still the same as its original version, $O(n * \alpha(n))$. The only new operations are adding to the map, which using the Union-Find algorithm is near-linear, and looking up entries in the map, which also takes near-linear time. Since the loop to create the witness is executed at most n times, witness construction takes $O(n * \alpha(n))$ time and the overall complexity of the algorithm remains unchanged (Norton 2009).

2.3.4. State minimization

A DFA can be minimized by partitioning its states into indistinguishable groups of states and combining these groups into single states (Hopcroft *et al.* 2006).

Each partition P is combined into a state s, where

- s will be a starting state if one the states in P is the starting state of the DFA
- s will be an accepting state if one of the states in P is an accepting state

- the transitions from s on some character a can be created by inspecting some state in P, $q \in P$. If q transitions to a state that is in some partition P' that has been (or will be) combined into s'', then $\delta(s, a) = s''$. Since all states in a partition are indistinguishable, it does not matter which q we pick because all states in P will transition to states in P' anyway.

A minimized DFA can now be created by taking the original DFA and replacing all states in each partition P with the combined state s.

The Table-Filling Algorithm

As The Table-Filling Algorithm identifies which states in a DFA are indistinguishable with each other, it can be used for DFA state minimization.

Instead of constructing the table from the states of two automata, it would only include a single DFA's states. After running the algorithm, all states in the DFA which are indistinguishable with each other are identified (they remain unmarked in table). These equivalent states can simply be grouped into partitions by iterating through the cells in the table and for each unmarked pair adding its states into a single set. If the states of a pair have already been merged into different sets, then these sets are merged into one larger set. The resulting sets will be partitions of indistinguishable states, which can be combined into a single state using the process described above.

The *n lg n* Hopcroft Algorithm

The *n lg n* Hopcroft Algorithm works by partitioning states into blocks, each block representing states which are indistinguishable with each other. A DFA can be minimized by using these blocks as the partitions that are combined using the process described above.

The (Nearly) Linear Algorithm

The (Nearly) Linear Algorithm can not be used for state minimization (Bonchi, Pous 2013). The algorithm can only output whether some two states being compared are equivalent or not.

To try and use it for minimizing a single DFA, we could try comparing the starting state of the DFA with itself, but this would only result in the creation of sets containing two copies of each state in the DFA and indistinguishable states within the DFA would not be identified.

2.4. Datasets

To showcase the running times of the algorithms implemented in this project, example datasets are needed. To generate good datasets, we must first analyze how the different parts of a DFA can influence the algorithms and also probe into each algorithm individually to discover what kinds of DFAs are needed to make it run in worst-case time.

DFAs are defined of five-tuples, consisting of states, an alphabet, the transition function, a starting state, and a set of final states.

Obviously, the number of states will influence the run time of the algorithms. Minimization needs to identify all indistinguishable states, and if there are more states, more operations will be done to partition these states. Equivalence testing needs to only identify whether starting states are distinguishable, and the only way to find out is to follow transitions until accepting states are reached. If there are more states, there are more transitions to follow and thus run time increases. Because run time needs to sufficiently large to highlight the different growth rates of the algorithms, the example datasets will need to contain a large number of states.

The transition function of a DFA also influences algorithm run time. Since the it determines the paths an algorithm can take to compare states, the run time heavily depends on the transition function and the way it arranges states. For some algorithms, a favorable transition function might decrease run time by a significant amount while for others the same transition function might increase run time. Because of this, the example datasets will need various different transition functions. Specific configurations that influence each algorithm's run time are explored below.

The size of a DFA's alphabet influences how many transitions there are in total since each state needs one transition for each symbol in the alphabet. A larger alphabet will increase the number of possible transitions, and because algorithms work by following transitions, the size of a DFA's alphabet is correlated to the running time of the algorithms. However, in time complexity analysis, the size of the alphabet is considered a constant and therefore it is not counted in big-oh notation (Norton 2009). Because alphabet size increases run time by a constant factor, having different alphabets for datasets is not helpful. Instead, datasets will have small variability in alphabet sizes.

For minimization algorithms, the starting state does not significantly influence the running time, because all indistinguishable states must be identified and thus the amount of operations will stay the same. For equivalence testing, starting states determine how many transitions need to be followed to reach a final state and thus significantly influence running times. To induce worst-case performance in equivalence testing, the start state must be initialized as far as possible from final states.

The number of accepting states also influences running time since accepting and non-accepting can easily be distinguished, giving initial partitions of states. Different algorithms use accepting states in different ways, so no generalization can be made about the number of accepting states relation to algorithm run time. The datasets will include examples with varying numbers of final states.

To conclude, the example datasets will need to feature a large number of states to increase runtime, as well various transition functions to exploit the weaknesses of each algorithm. Starting states will be far from final states, and the number of final states will vary. The alphabets used will only have small variations.

The Table-Filling Algorithm

The Table-Filling Algorithm's time complexity is dominated by the process of marking states as indistinguishable (step 5 in the pseudocode). To run the algorithm with the worst-case performance, all iterations of this loop must be as long as possible, and the number of iterations must also be the maximum number possible. In this worst-case scenario, both the amount of iterations and the work done in each iteration scale with $O(n^2)$ (Norton 2009).

Looping stops whenever no pairs are marked in an iteration. Therefore, to maximize the amount of iterations, in each iteration the minimal number of pairs possible must be marked. Additionally, to have iteration continue as long as possible, all states must eventually be distinguished.

Each iteration only does work on unmarked pairs. To maximize the amount of work done, the number of unmarked pairs must be as high as possible in each iteration. Therefore, the number of pairs marked each iteration must again be minimal.

Details on creating DFAs for the algorithms can be found in chapters 5 (Implementation) and 7 (Experimentation).

The *n lg n* Hopcroft Algorithm

The *n lg n* Hopcroft Algorithm's time complexity is dominated by the loop of partitioning blocks into smaller ones (step 7 in the pseudocode). The number of iterations done is proportional to n and the amount of work done each iteration is proportional to $\lg n$. To induce worst-case performance, both parts must be maximized.

Looping stops when all to-do lists are empty, and that happens when there are no more blocks to be split. Therefore, to maximize the number of iterations, blocks must be split for as long as possible, *i.e.* until every single block only contains a single state.

The amount of work done per iteration depends on how many states with predecessors the block removed from a to-do list has (the size of $a_a(i)$). Whenever we split a block B into two smaller ones B_1 and B_2, we always add the block with the least amount of states with predecessors to the to-do list. Therefore, in the next iteration we will process, at most, half of the states in $a_a(i)$. To maximize the amount of work done, $a_a(i)$ needs to be as large as possible and each block needs to be split exactly in half per iteration.

An example DFA capable of inducing worst-case performance from this algorithm would have all states be distinguishable (to maximize iterations) and each state would have predecessors on all input symbols in the alphabet (to maximize size of $a_a(i)$). Additionally, blocks must be split in half each iteration, which can be achieved by having the initial blocks be equally sized (having the same number of accepting and non-accepting states), as well as arranging transitions in such a way that for each symbol half of the states will always transition to one block and half to the other.

The (Nearly) Linear Algorithm

The (Nearly) Linear Algorithm's time complexity is dominated by the loop combining sets (step 4 in the pseudocode). The number of iterations done is proportional to n and the amount of work done each iteration is proportional to $\alpha(n)$.

Since the amount of work done per iteration near-constant, maximizing it has virtually zero (observable) effect. Thus, the number of iterations must be maximized instead.

Iteration stops whenever the processing stack is empty. The processing stack can only be empty once all reachable states have been processed, which implies it does not matter how they are arranged, as they will all be processed at some point anyway.

However, as an optimization, the algorithm can be stopped when any pair with a contradiction is merged into the same set (this is done in the witness variation of the algorithm). In this case, the contradictory pair must be the last pair evaluated. This can be achieved by exploiting the fact that the algorithm uses a stack to keep track of states to be processed. Because of this, iteration takes the form of a depth-first search (DFS). To maximize the numbers of states that are evaluated in DFS, a DFA could be constructed in a tree-like fashion, where the the final pair of the final branch being evaluated consists of an accepting and non-accepting pair.

For the witness variation of the algorithm, a queue is maintained, and the search takes the form of a breadth-first search. Again, we can maximize the number of iterations by having the final pair evaluated be the pair with the contradiction. An example DFA that achieves this would be a tree where the contradictory pair is the pair furthest away from starting state.

Chapter 3

Requirements and Specification

This chapter contains an overview of the core requirements for the software produced in this project. All the requirements were specified to support the brief in section 3.1., which outlines the main intended outcome of the project.

The requirements were generated using the process of requirements engineering, mostly with the techniques of brainstorming and whiteboarding.

Since the software was developed using agile methodology (described more in chapter 5, Implementation), the requirements changed often and were iterated on during the lifetime of the project.

3.1. Brief

The primary goal of this project is to create a piece of educational software that end-users can use to learn about and run DFA equivalence testing and state minimization algorithms and also demonstrate those algorithms' worst-case time complexities using concrete examples.

3.2. Requirements

The software can be split into three overarching categories, each with their own set of requirements:

- The overall front end of the software, consisting of the UI users will see. This category includes common functionality shared between the algorithms and all the elements used to navigate the app
- The environment. This includes resource constraints for the application as well browser compatibility requirements.
- The algorithms and datasets. This category is concerned with the implementation of specific algorithms, including correctness and how they are visualized. Additionally, this includes the datasets that show worst-case complexity and their generation.

Some of the requirements have been marked as optional to indicate that while they would enhance user experience, they are not explicitly necessary to achieve the outcome in the brief.

3.2.1. Front End Requirements

- **F1:** When first navigating to the application, users are given overview of the app's purpose and functionality
- **F2:** Users are given a choice of running any of the six state minimization and equivalence testing algorithms described in section 2.3.
- **F3:** For algorithms which support both state minimization and equivalence testing, users can choose which mode to run the algorithm in
- **F4:** Users can choose whether to run equivalence testing with witness string generation or not
- **F5:** To run algorithms, users can use pre-generated example inputs
- **F6:** Whenever a user selects a pre-generated input, it is visualized to them and they can modify it
- **F7:** Users can run algorithms using custom inputs
- **F8:** Users can save input to a file
- **F9:** Users can load input from files
- **F10:** Input DFAs are visualized to users as they enter them
- **F11:** Every algorithm will be runnable in two distinct modes: visual mode and headless mode
- **F12:** Running in visual mode visualizes the working process of the algorithms
- **F13:** Visual mode: users can step forward through the execution of an algorithm
- **F14:** Visual mode: users can skip to the end of an algorithm's execution
- **F15:** Visual mode: users can reset execution to the algorithm's initial state
- **F16:** Visual mode: currently executed stage of an algorithm is shown
- **F17:** Visual mode: there is a dedicated section for state and variables of an algorithm
- **F18:** Visual mode: the input to an algorithm is shown throughout execution
- **F19:** Visual mode: every algorithm will have a text-based representation of the working process
- **F20:** (OPTIONAL) Visual mode: every algorithm will have a graphic visualization
- **F21:** Visual mode: users can go back from an algorithm's visualization to modify the input data at any time
- **F22:** Visual mode: users can switch from an algorithm's visualization to any other algorithm's visualization or headless mode with the same input
- **F23:** Running in headless mode computes the algorithms' results without any visuals or stepping
- **F24:** Headless mode: users can select one or many algorithms to run on the same input

- **F25:** Headless mode: algorithms are run sequentially, and results are reported as soon as they come in
- **F26:** Headless mode: when all algorithms finish running, a report is produced indicating the result and run time of each algorithm
- **F27:** Headless mode: users can select any algorithm that was run and visualize it's working process using the input to headless mode
- **F28:** If an algorithm is run in state minimization mode, the resulting DFA visualized
- **F29:** Users can save the result of state minimization to a file
- **F30:** Every algorithm will have a brief description including complexity, with guidance on finding additional information
- **F31:** The application will contain a help page providing an in-depth guide of using the application, a description of the project, and link to the source code of the application
- **F32:** The application will have an accessible design, conforming with the Web Content Accessibility Guidelines (WCAG) and achieving an accessibility rating of at least A.

3.2.2. Environment Requirements

- **E1:** The application will run in all modern browser (supporting >99% of current browser usage)
- **E2:** The application will have a maximum idle memory usage of 10 MB (excluding memory used by algorithms)
- **E3:** The application will have a maximum compressed bundle size of <1 MB
- **E4:** The application must be runnable in any user's local browser without the need for a back-end server
- **E5:** The application will not have dependencies on non-standardized browser or OS features. Only standardized JS and browser APIs will be used
- **E6:** (OPTIONAL) The application will scale to match the different aspect ratios of desktop computers, tablets, and mobile devices

3.2.3. Algorithm and Dataset Requirements

The Table-Filling Algorithm

- **A1:** Algorithm correctly implemented for both state minimization and equivalence testing and runs in $O(n^4)$ time
- **A2:** Witness variation of equivalence testing mode correctly implemented and runs in $O(n^4)$ time
- **A3:** Visual mode: table construction (empty table) shown

- **A4:** Visual mode: whether a pair has been distinguished in the current is clearly indicated
- **A5:** Visual mode: each step is equivalent one iteration in the main loop of the algorithm
- **A6:** Visual mode, witness variation: table indicates which input distinguishes pairs
- **A7:** Visual mode, witness variation: witness string construction shown

The *n lg* n Hopcroft Algorithm

- **A8:** Algorithm correctly implemented for both state minimization and equivalence testing and runs in $O(n \lg n)$ time
- **A9:** Witness variation of equivalence testing mode correctly implemented and runs in $O(n^2)$ time
- **A10:** Visual mode: all partition blocks shown
- **A11:** Visual mode: inverse transition function visualized as a table
- **A12:** Visual mode: sets of states with predecessors on certain input ($a_a(i)$) from blocks shown
- **A13:** Visual mode: to-do lists L_a shown
- **A14:** Visual mode: comparisons of $a_a(i)$ and adding elements to to-do lists shown
- **A15:** Visual mode: process of marking and partitioning blocks shown
- **A16:** Visual mode, witness variation: symbols table shown
- **A17:** Visual mode, witness variation: witness construction shown

The (Nearly) Linear Algorithm

- **A18:** Algorithm correctly implemented for equivalence testing and runs in $O(\alpha(n))$ time
- **A19:** Witness variation of algorithm correctly implemented and runs in $O(\alpha(n))$ time
- **A20:** Visual mode: current sets of distinguishable sets shown
- **A21:** Visual mode: processing stack shown
- **A22:** Visual mode: comparison of successors for pairs in stack shown
- **A23:** Visual mode, witness variation: FIFO processing queue shown
- **A24:** Visual mode, witness variation: transition map shown
- **A25:** Visual mode, witness variation: witness construction shown

Datasets

- **A26:** The application will provide example datasets for running algorithms, which demonstrate how an algorithm works and are intended to be run in visual mode
- **A27:** Each algorithm will have specific example dataset tailored to exploit its weaknesses
- **A28:** Datasets are generated by reusable dataset generators
- **A29:** Users can create own datasets using the dataset generators

- **A30:** Dataset generations support configuring the number of states, size of an alphabet, and number of accepting states
- **A31:** Users can select from different types of transition functions to use for generated datasets specifically designed for each algorithm
- **A32:** User-generated datasets are visualized as graphs
- **A33:** Users can directly use generated data as input to algorithms or headless mode
- **A34:** Users can save generated data to file

Chapter 4

Design

Due to the long list of requirements and technical nature of this project, the application produced will be a large and complex one. To manage this complexity, the design phase plays an important role in limiting potential troubles in implementation and testing.

A way to reduce complexity and ensure scalability of web applications is by using the TypeScript (TS), a strongly typed language that compiles into regular JavaScript. JavaScript is dynamically typed and using it can result in run time type errors, but using TypeScript, all type checking can happen at compile-time, which means programs will be free of type errors. Secondly, TS types act as a form of specification, which developers can use to familiarize themselves with library APIs quicker. Finally, using types integrated development environments (IDEs) can provide autocomplete suggestions which increases productivity. For these reasons, the project will use TS instead of regular JS. (TypeScript 2021)

To create complex client-side web applications, developers reduce cognitive load and reuse code by using web frameworks. Web frameworks are extensive libraries that provide useful reusable abstractions (*e.g.* state management or event handling) for app development. Web frameworks form a standard part of modern companies' tooling and are extensively used in modern front-end web development. They provide developers with "tried and tested tools for building scalable, interactive web applications", which are also needed for this project. (Understanding client-side JavaScript frameworks 2020)

The most popular web frameworks in 2020 are React, Ember, Angular, and Vue (*ibid.*). From this selection, the project will use React because of its large developer community and mature ecosystem, as well the author's previous familiarity with it.

React is described on its homepage as a "library for building user interfaces" (React 2020). It is a declarative component-based framework, where UI elements are reusable encapsulated components that manage their own state (*ibid.*). Because of React's component-based nature, it is suitable for this project, as visualizing algorithms will often reuse the same code for shared functionality (displaying algorithm variables, visualizing DFAs).

Because frameworks are so influential in the development lifecycle of an app, the decision to use React was made early and the application will be built around its functionality. In this case it means that algorithm implementations and visualization will be designed in a way that they may be encapsulated in components.

For example, consider the simplified component hierarchy in Figure 16 meant to represent visualizing an algorithm Algorithm1. Visualizations of different algorithm can share the StateSection and StepControls components, while the actual algorithm's executed is in the Algorithm1Visualization component. The wrapping VisualizationContainer component would be reusable and adapt to different algorithms being passed in by changing the nested AlgorithmXVisualization component.

```
<VisualizationContainer algorithm={new Algorithm1()}>
    <StateSection/>
    <Algorithm1Visualization/>
    <StepControls/>
</VisualizationContainer>
```

Figure 15. Simplified React component hierarchy for visualizing an algorithm.

A design challenge for this project is implementing algorithms in a way that they can be run in both headless and visual modes using only one codebase. Implementing headless mode would be straightforward, consisting of simply translating algorithms' pseudocode to actual code. For visual mode, however, execution must happen in single steps so users can take steps through the visualization.

To enable this behavior, the algorithm implementations will follow a modified version of the State design pattern, which allows "an object to alter its behavior when its internal state changes" (Gamma, Helm, Johnson, Vlissides 1994). The idea behind the pattern is keeping track of an object's state in a variable and switching its behavior based on that variable. In the case of algorithms, the state would be the current step the algorithm is in and a single step would be defined as running some part of the algorithm and modifying the state variable to a new one. Example code of this pattern is shown in Figure 16.

```
class Algorithm1 {
    state = "X"

    step() {
        switch (this.state) {
            case "X":
                this.doX();
                break;
            case "Y":
                this.doY();
                break;
            case "Final":
                console.log("Finished executing")
                return;
        }
    }

    doX() {
        // do X
        this.state = "Y"
    }

    doY() {
        // do Y
        this.state = "Final"
    }
}
```
Figure 16. Example algorithm implementation using the modified State pattern.

Visualization components could access the state field and visualize the algorithms accordingly, with step controls invoking a single step in the algorithm. Running in headless mode would still be possible by looping steps until the “Final” state is reached.

Chapter 5

Implementation

5.1. Methodology & Tooling

The project was developed using an iterative agile software development lifecycle model. Agile is a way of developing software in which change is seen as an unavoidable part of the process, and therefore it is accounted for and integrated into the process itself. Agile approaches are supported by a myriad of frameworks (Scrum, Extreme Programming) and techniques (pair programming, sprints, stand-ups) which are all joined by adherence to the Agile manifesto. The manifesto lays out the fundamental principles of agile, emphasizing the importance of regular, small releases, valuing product over process, and accounting for change. (Agile 101 2021)

In the implementation process, no specific agile framework was used, instead the general agile principles were followed. For example, releases were small, self-contained, and regular; rigorous automated testing and tooling was used to ensure code quality; feedback from stakeholders was gathered early and often.

Pull requests

The project was implemented as a Git repository hosted on GitHub. Development utilized GitHub pull requests (PRs), in which a request is made to merge Git commits from one Git branch into another (parent) branch. Feature development was done by branching off the default "main" branch, implementing the feature in a series of commits and then submitting a PR to merge the commits from the feature branch to the main branch.

Using this, commits with similar domains (*e.g.* a new feature being implemented or a bug being fixed) were grouped together into a single logical whole (the commits in the pull request) that represented a working version of the software. Therefore, each pull request was a self-contained release of the software that could be tested, run, and deployed using CI/CD.

CI/CD

Continuous Integration (CI) is an automation process in software development where changes to software are automatically and frequently built, tested, and merged. Continuous Delivery/Deployment (CD) builds upon the CI process and extends it by automatically creating a release after passing the CI process and deploying the software to a production environment, where the software can be used by end-users. (What is CI/CD? 2021)

The agile approach of the project was supported by a strong CI/CD pipeline. Whenever a pull request was created or new commits were pushed to the source branch of the PR, the CI system was run. It consisted of first running the TypeScript compiler for static type checking and identifying any syntax errors, then running ESLint to do linting and checking for code anti-patterns, then running Prettier to check that code formatting is consistent, and finally running automated regression tests.

After successfully passing these the CI checks, the pull request would become eligible for merging. After merging it to the main branch, the CD system would start execution by creating a minimized (in terms of bundle size) and optimized (in terms of code speed) build using Webpack. This build was automatically deployed to the web server hosting the code using secure copy protocol (scp) which made the latest version of the application publicly available without any effort from the developer.

The CI/CD pipeline was implemented in GitHub Actions, which defines CI/CD in terms of jobs, which consist of steps. The aforementioned CI flow consisted of two jobs – "build" and "test" and the CD flow consisted of a single "publish" job. A graphical representation of a successful CI/CD flow can be seen in Figure 17.

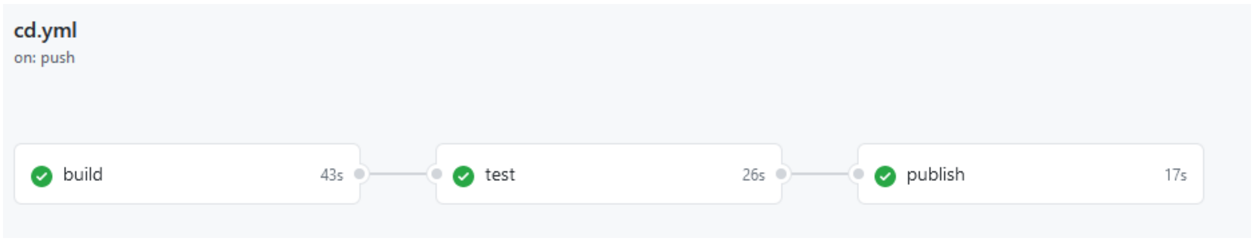


Figure 17. GitHub Actions CI/CD flow.

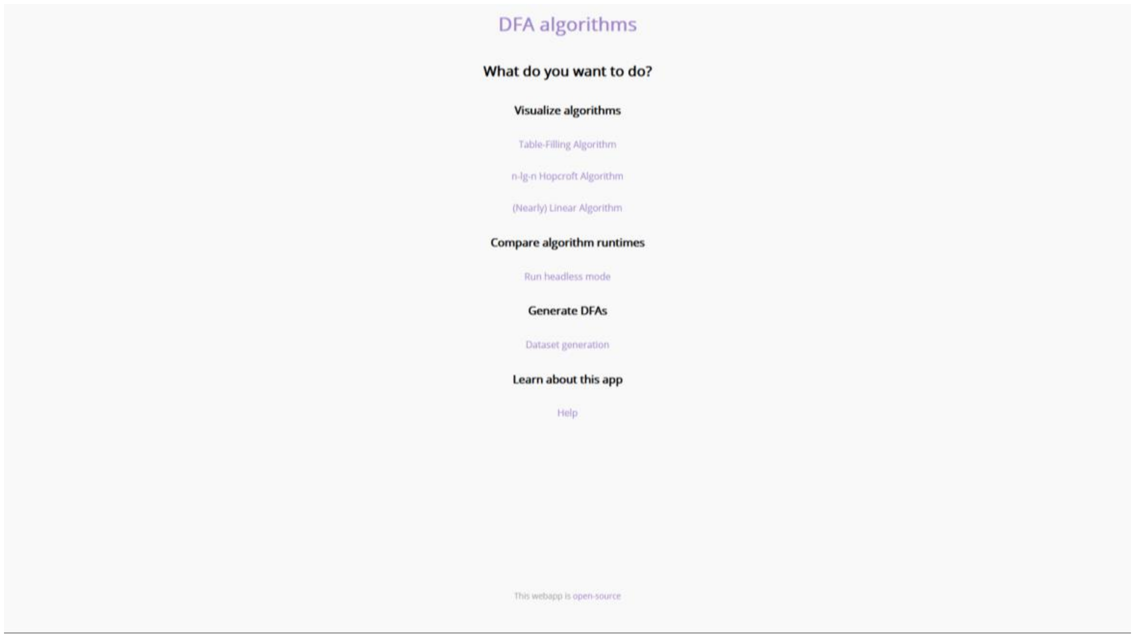
Tooling

To ensure code quality, extend language functionality, and automate manual processes, various software tools were used:

- tsc, the TypeScript compiler, to ensure type safety
- ESLint, a JavaScript linter, to lint the code and check for the occurrence of anti-patterns
- Prettier, a code formatter, to maintain consistency in code style
- husky and pretty-quick, to create Git hooks that automatically run Prettier on all committed code

- Sass, a Cascading Style Sheets (CSS) extension, to add options to apply styling in a more concise fashion
- Webpack, a build tool, to minimize and optimize the source code into deployable static files

5.2. Common Front-End


DFA algorithms

What do you want to do?

Visualize algorithms

Table-Filling Algorithm

n-lg-n Hopcroft Algorithm

(Nearly) Linear Algorithm

Compare algorithm runtimes

Run headless mode

Generate DFAs

Dataset generation

Learn about this app

Help

This webapp is open-source

Figure 18. Screenshot of the landing page.

Although the project's main goal is to implement and visualize DFA algorithms, doing so needs common infrastructure to serve as a foundation for the algorithm implementations. This foundation is auxiliary to the algorithms and is needed to ensure a smooth end-user experience.

The common front-end consists of the landing page, the navigation infrastructure, data input systems, algorithm visualization helpers, and the headless mode runner.

The landing page was designed to be minimal and easily accessible and to give users a clear overview of the application's capabilities. The user is given a list of actions to choose from with possible actions clearly highlighted in a different color than non-interactive text. A screenshot of the landing page can be found in Figure 18.

The navigation infrastructure provides functionality for users to reuse the same input in different contexts. For example, users can directly transition from one algorithm's visualization to another algorithm or headless mode without having to reinput their data. It works by serializing input

DFAs to JSON format and passing them as HTTP parameters to other pages, which then decode the JSON.

The data input system enables users to either select a pre-generated input or manually input their own custom DFAs to run algorithms on. Using pre-generated inputs is straightforward, whenever a user clicks on a button the manual input section is filled in with the data associated with the pre-generated input and users can inspect or modify it before running an algorithm.

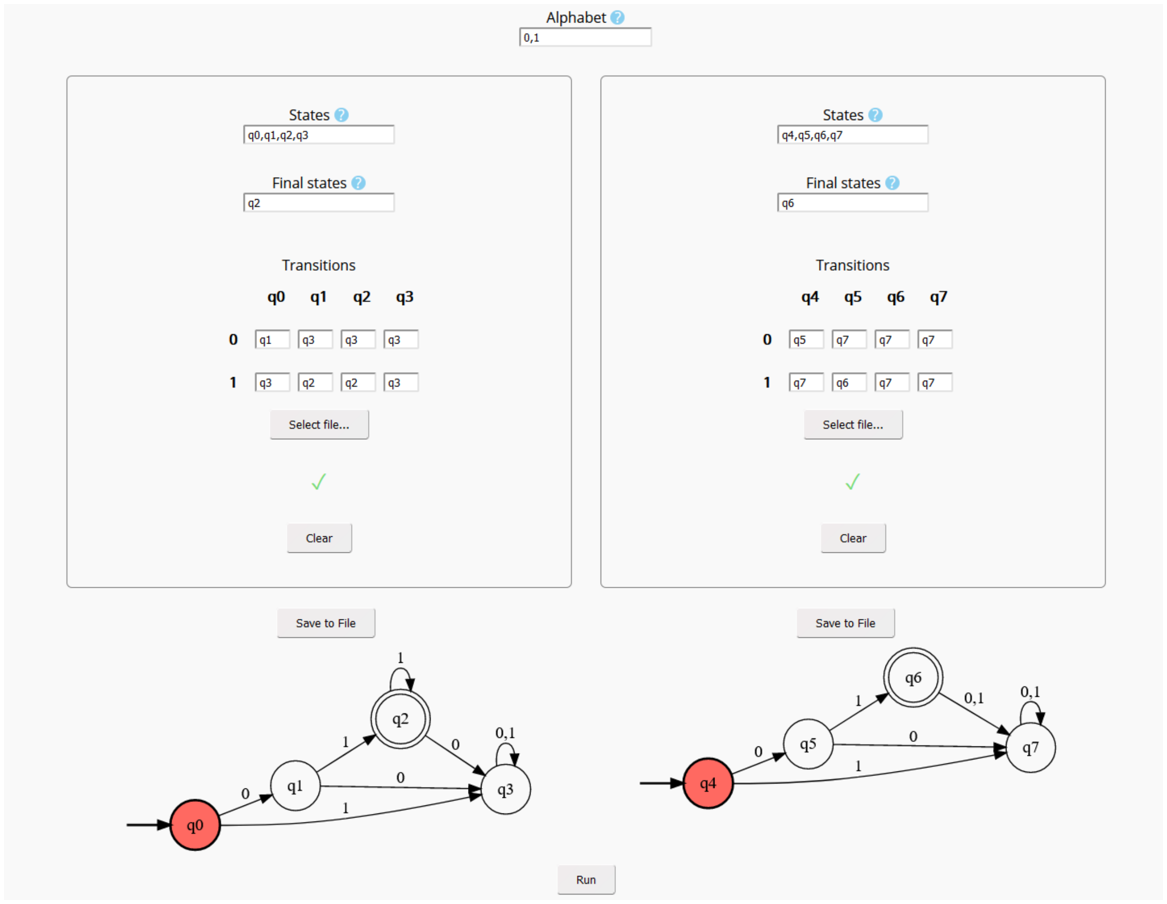
The custom input system is, however, a complex mix of components that aims to be as dynamic and user-friendly as possible. Users are given fields for a DFA's alphabet, states, and final states with tooltips that describe each field's purpose and expected data format. Validation is also run on the fields and fields are highlighted in red if they have missing or invalid values. For example, when the final states field contains a state that is not declared in the states field, it is highlighted to ensure that the user corrects the input.

Based on the DFA's alphabet and states, a table is dynamically generated to represent the transition function of the DFA. Columns of the table correspond to states, rows correspond to symbols in the alphabet, and each cell is one entry in the transition function. Whenever new states are added or the alphabet is extended with new symbols, new rows and columns are created in the table. Whenever states or symbols are deleted, their rows and columns are also removed, while existing values are preserved.

Finally, when the user has input a valid DFA (or selected a pre-generated input), a visualization of the DFA is generated. This visualization uses a module in the noam library described in chapter 2 and is designed to be minimal and accessible to help users get a clear overview of their DFAs.

Alongside the visualization, the "Save to File" and "Run" buttons are enabled. Clicking "Save to File" saves the current DFA configuration as a JSON file to the user's computer, which they can later use to run other algorithms or headless mode using the "Select file…" button. Clicking "Run" initializes a new algorithm instance with the specified input. A screenshot of the data input system can be found in Figure 19.

The data input system was packaged into reusable React components that were utilized for both algorithm visualizations and headless mode.


Alphabet: 0,1

States: q0,q1,q2,q3

Final states: q2

Transitions

	q0	q1	q2	q3
0	q1	q3	q3	q3
1	q3	q2	q2	q3

States: q4,q5,q6,q7

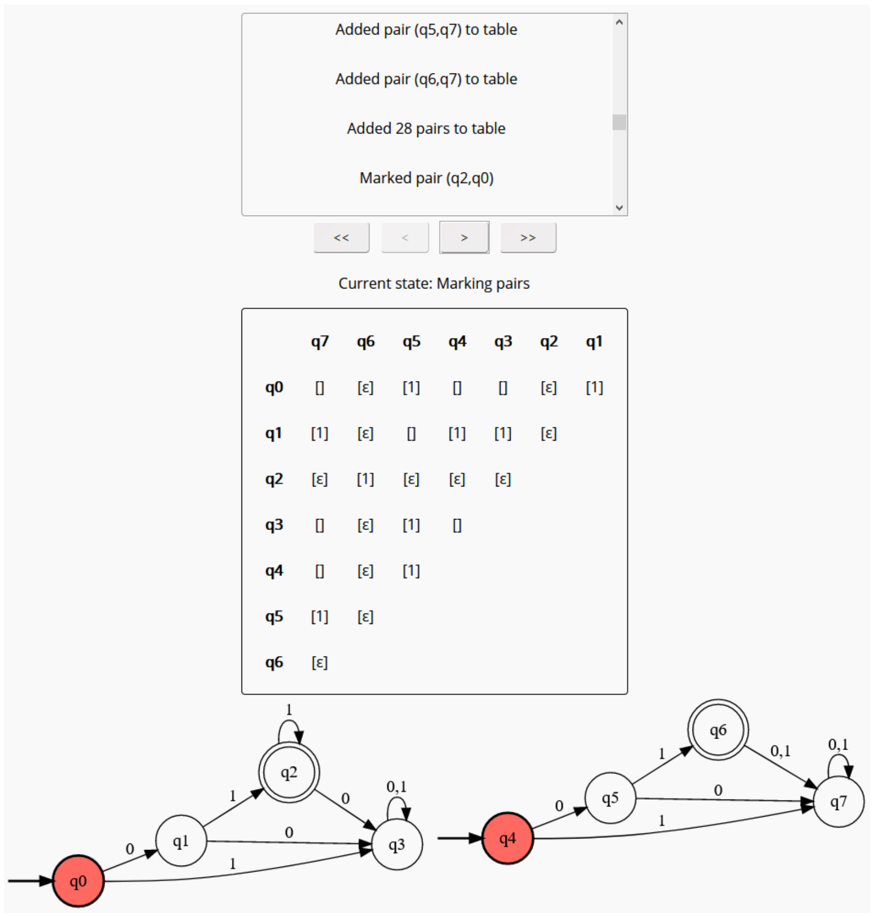
Final states: q6

Transitions

	q4	q5	q6	q7
0	q5	q7	q7	q7
1	q7	q6	q7	q7

Figure 19. Screenshot of data input system.

Algorithm visualization helpers consist of the AlgorithmVisualization, AlgorithmLog, and AlgorithmStepControls React components. AlgorithmVisualization is a component that acts as a wrapper around a specific algorithm's visualization and adds extra functionality to the visualization in the form of a log, where the algorithm can output messages describing its actions, and step controls, which users can use to step through the algorithm. Additionally, AlgorithmVisualization visualizes the inputs to the algorithm so users can see exactly which transitions and states algorithms use. An example of the Table-Filling Algorithm utilizing the visualization helpers can be seen in Figure 20.



Added pair (q5,q7) to table

Added pair (q6,q7) to table

Added 28 pairs to table

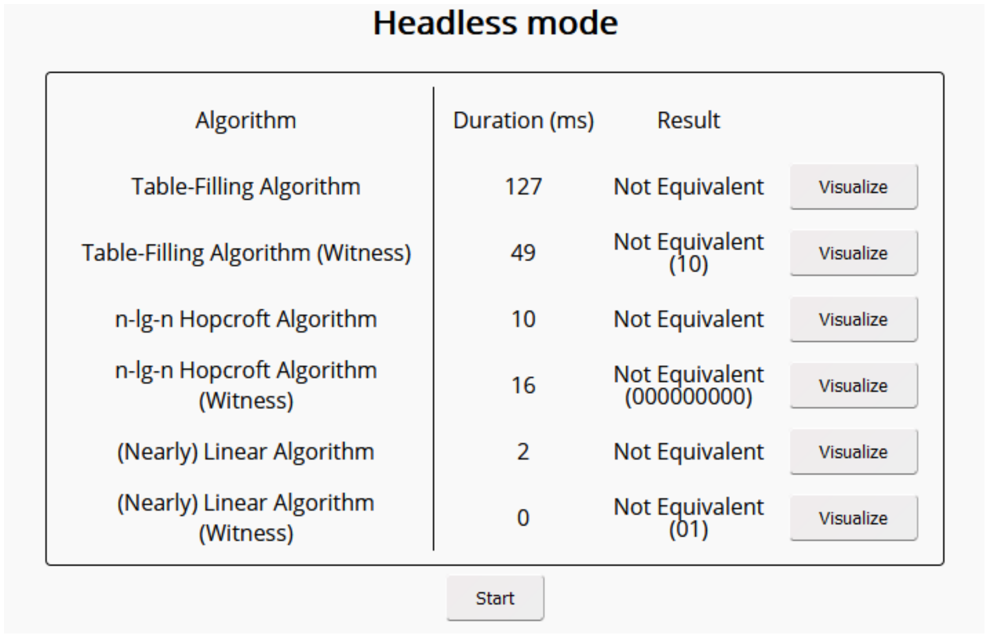
Marked pair (q2,q0)

Current state: Marking pairs

	q7	q6	q5	q4	q3	q2	q1
q0	[]	[ε]	[1]	[]	[]	[ε]	[1]
q1	[1]	[ε]	[]	[1]	[1]	[ε]	
q2	[ε]	[1]	[ε]	[ε]	[ε]		
q3	[]	[ε]	[1]	[]			
q4	[]	[ε]	[1]				
q5	[1]	[ε]					
q6	[ε]						

Figure 20. Screenshot of visualization of The Table-Filling Algorithm (Witness) including algorithm visualization helpers.

Finally, the headless mode runner acts as a container for algorithm instances, which can be run by clicking a “Start” button, after which each algorithm is run with the same input and results are reported back to the user. Users can then compare these results and visualize individual algorithms. A screenshot of the headless mode runner can be seen in Figure 21.



Headless mode

Algorithm	Duration (ms)	Result	
Table-Filling Algorithm	127	Not Equivalent	Visualize
Table-Filling Algorithm (Witness)	49	Not Equivalent (10)	Visualize
n-lg-n Hopcroft Algorithm	10	Not Equivalent	Visualize
n-lg-n Hopcroft Algorithm (Witness)	16	Not Equivalent (000000000)	Visualize
(Nearly) Linear Algorithm	2	Not Equivalent	Visualize
(Nearly) Linear Algorithm (Witness)	0	Not Equivalent (01)	Visualize

Figure 21. Screenshot of headless mode runner.

5.3. Algorithms and Datasets

All algorithms were implemented as JS classes that implemented a Typescript Algorithm interface. This interface grouped common behavior between the algorithms and forced them to implement a step-based execution model corresponding to the modified State pattern discussed in the Design chapter.

Actual algorithm implementations overrode some of the Algorithm interface's properties with more specific types, for example the (Nearly) Linear Algorithm has a result property of type 'DFA' for the because it cannot do equivalence testing.

```
interface Algorithm {
    type: AlgorithmType;
    state: AlgorithmState;
    run: () => void;
    step: () => void;
    reset: () => void;
    log?: Log;

    input1: DFA;
    input2?: DFA;

    mode: AlgorithmMode;
    produceWitness: boolean;
    witness: string;
    result: EquivalenceTestingResult | DFA;
}

enum EquivalenceTestingResult {
    NOT_AVAILABLE,
    EQUIVALENT,
    NON_EQUIVALENT,
}

enum AlgorithmMode {
    EQUIVALENCE_TESTING,
    STATE_MINIMIZATION,
}
```
Figure 22. TypeScript Algorithm interface.

DFAs were implemented as a TypeScript interface with each DFA storing an array of states, a set of final states, an alphabet, and a distinct starting state. Final states are stored in a set data structure because the algorithms require looking up whether certain states are final or not and by using a set that can be done in constant $O(1)$ time. Transitions were represented as a map from strings (representing symbols in the alphabet) to states because this also enables constant $O(1)$ lookup time needed to check the successors of states in the algorithms.

```
interface DFA {
    states: State[];
    startingState: State;
    finalStates: Set<State>;
    alphabet: string[];
}

type Transitions = Map<string, State>;

interface State {
    name: string;
    transitions: Transitions;
}
```
Figure 23. TypeScript DFA interface

The Table-Filling Algorithm

The Table-Filling Algorithm was the first algorithm to be implemented. The core of the algorithm, the table itself, is stored as a map data structure to ensure that successor pairs can be looked up in constant time $O(1)$ as to not increase the algorithm's $O(n^4)$ time complexity. The map uses pairs of states (implemented as a JS array with type [State, State]) as keys and strings as values. The value of an entry is the symbol distinguishing the pair if one exists, otherwise it is empty.

The built-in JavaScript Map class was not suitable for the map since it uses object identity for inclusion checks but keeping track of and reusing the same array for every single possible pair of states would result in unnecessarily complex and unmaintainable code. Therefore, a dependency of "hashmap" was used instead, which defines a map that does inclusion checks by comparing a hash of the key. During development a bug was found in hashmap's TypeScript definitions and was fixed and merged into the upstream definitions (Laane 2021).

The algorithm was implemented using the modified State pattern, with the states being INITIAL_STATE, EMPTY_TABLE, MARKING_PAIRS, ALL_PAIRS_MARKED, CONSTRUCTING_WITNESS, INDISTINGUISHABLE_STATE_GROUPS_IDENTIFIED, FINAL_STATE.

Visualizing the Table-Filling Algorithm was done by rendering the pairs stored in the map as a table and displaying the values distinguishing pairs in the table's cells. Each time AlgorithmStepControls were used to run a step in the algorithm, the table was re-rendered with updated pairs. A screenshot of the Table-Filling Algorithm's visualization can be seen in Figure 20.

The *n lg n* Hopcroft Algorithm

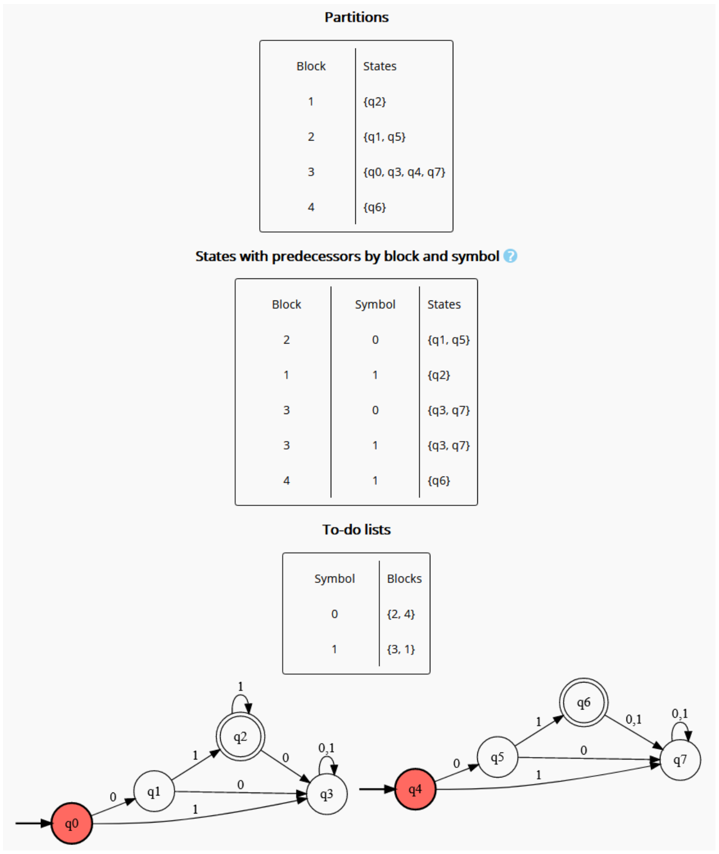
The *n lg n* Hopcroft Algorithm was the second algorithm to be implemented and it was the algorithm with the longest and most complex implementation, coming in at over 450 lines of TypeScript. Each instance of the algorithm needed to store an inverse transition function, partitions of states, sets of states with predecessors, to-do lists, (optionally) the witness table and update them throughout execution, which needed complex and long code. To optimize access times, all of these properties are stored as either as maps or sets.

Execution of the algorithm happens in steps, with the algorithm defining the following states: INITIAL_STATE, INVERSE_TRANSITION_FUNCTION_CREATED, INITIAL_PARTITIONS_CREATED, SETS_OF_STATES_WITH_PREDECESSORS_CREATED, PARTITIONING_BLOCKS, ALL_BLOCKS_PARTITIONED, CONSTRUCTING_WITNESS, FINAL_STATE.

The algorithm's visualization consists of different tables, where for each state tables relevant to it are shown. Specific tables corresponding to states can be seen in Table 3. A screenshot of the algorithm's visualization can be seen in Figure 24.

State	Visualization
INITIAL_STATE	-
INVERSE_TRANSITION_FUNCTION_CREATED	Inverse transition function
INITIAL_PARTITIONS_CREATED	Blocks (partitions)
SETS_OF_STATES_WITH_PREDECESSORS_CREATED	Blocks, states with predecessors by block and symbol
PARTITIONING_BLOCKS	Blocks, states with predecessors by block and symbol, to-do lists
ALL_BLOCKS_PARTITIONED	Blocks
CONSTRUCTING_WITNESS*	Witness table
FINAL_STATE	Blocks, witness table*

*Table 3. Visualizations of the n lg n Hopcroft Algorithm's states. Entries marked with * signify they are only visualized in witness mode.*


Partitions

Block	States
1	{q2}
2	{q1, q5}
3	{q0, q3, q4, q7}
4	{q6}

States with predecessors by block and symbol

Block	Symbol	States
2	0	{q1, q5}
1	1	{q2}
3	0	{q3, q7}
3	1	{q3, q7}
4	1	{q6}

To-do lists

Symbol	Blocks
0	{2, 4}
1	{3, 1}

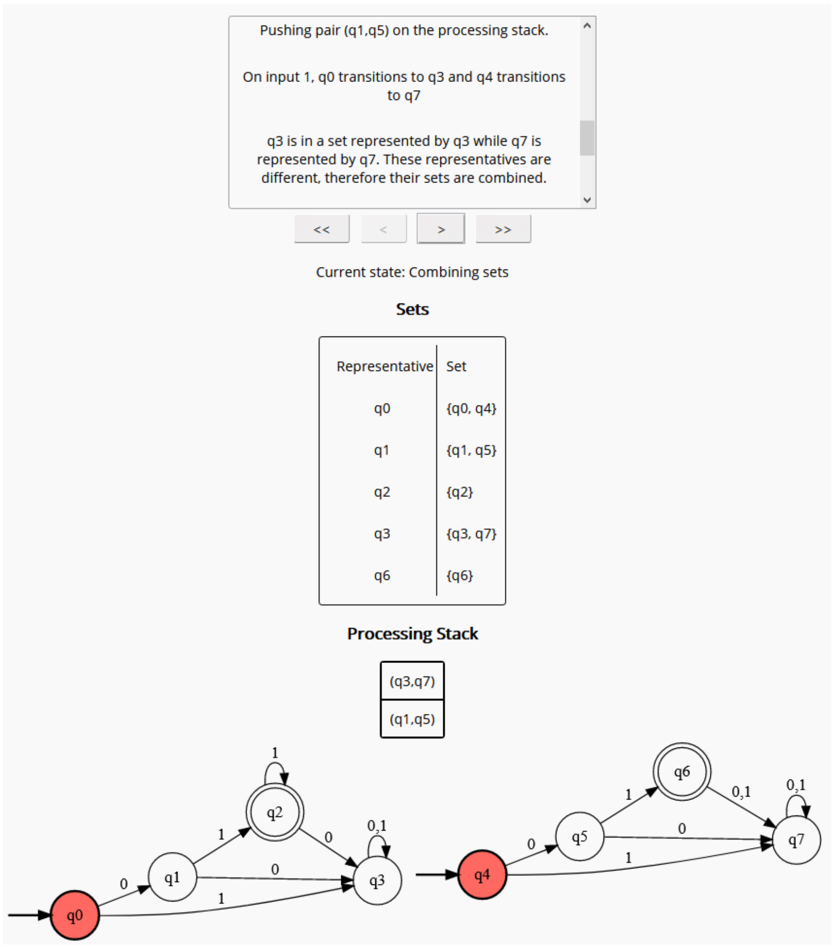
Figure 24. Visualization of the n lg n Hopcroft Algorithm.

The (Nearly) Linear Algorithm

The (Nearly) Linear algorithm was implemented using the disjoint set data structure. This data structure keeps track of all the sets and their representatives and enables combining sets in (nearly) constant time. Each state in the input DFAs is initialized as a distinct set and during processing the disjoint set data structure handles combining indistinguishable sets. The processing stack/queue is maintained using corresponding data structures and the witness map is maintained as a hashmap to enable looking up pairs by hash instead of identity.

The (Nearly) Linear Algorithm was implemented with the following states: INITIAL_STATE, SETS_INITIALIZED, COMBINING_SETS, ALL_SETS_COMBINED, CONSTRUCTING_WITNESS, FINAL_STATE.

The algorithm visualization displays all sets with representatives and the processing stack (or queue in witness mode). In witness mode, the witness map is also shown. A screenshot of the algorithm's visualization can be seen in Figure 25.


Pushing pair (q1,q5) on the processing stack.

On input 1, q0 transitions to q3 and q4 transitions to q7

q3 is in a set represented by q3 while q7 is represented by q7. These representatives are different, therefore their sets are combined.

Current state: Combining sets

Sets

Representative	Set
q0	{q0, q4}
q1	{q1, q5}
q2	{q2}
q3	{q3, q7}
q6	{q6}

Processing Stack

(q3,q7)
(q1,q5)

Figure 25. Visualization of The (Nearly) Linear Algorithm.

Datasets

As well as implementing algorithms, this project aims to compare them, highlighting their worst-case time complexities. To achieve this, datasets of large DFAs (hundreds or thousands of states) were needed.

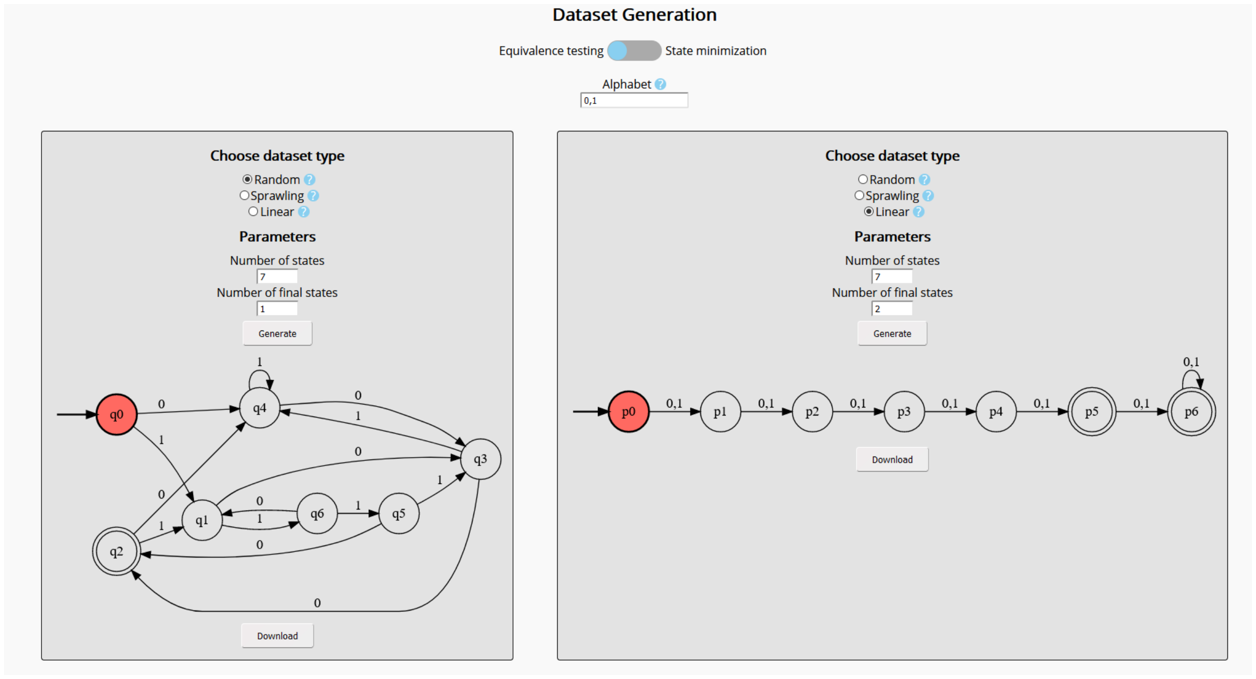
To create these datasets, a DatasetGenerator TypeScript interface was defined. This interface specified the parameters each generator would take to produce a DFA, such as the number of states in the DFA, its alphabet, and the number of final states.

```
type DatasetGenerator = (
    statesCount: number,
    alphabet: string[],
    finalStatesCount: number,
    statePrefix?: string
) => DFA;
```

Figure 26. DatasetGenerator TypeScript interface.

This interface was implemented by 4 functions, each creating DFAs from templates that were designed to exploit the implemented algorithms. These datasets are detailed in chapter 8 (Experimentation).

As well as using these generators for example datasets, users also have the ability to generate custom datasets in the application. User-generated datasets are visualized to give a graphical overview of what kind of DFAs they represent. Additionally, each dataset type has an in-depth tooltip explaining in natural language how it is generated and what algorithms have it as the worst-case input.



Dataset Generation

Equivalence testing / State minimization

Alphabet: 0,1

Choose dataset type
● Random
○ Sprawling
○ Linear

Parameters
Number of states: 7
Number of final states: 1

Choose dataset type
○ Random
○ Sprawling
● Linear

Parameters
Number of states: 7
Number of final states: 2

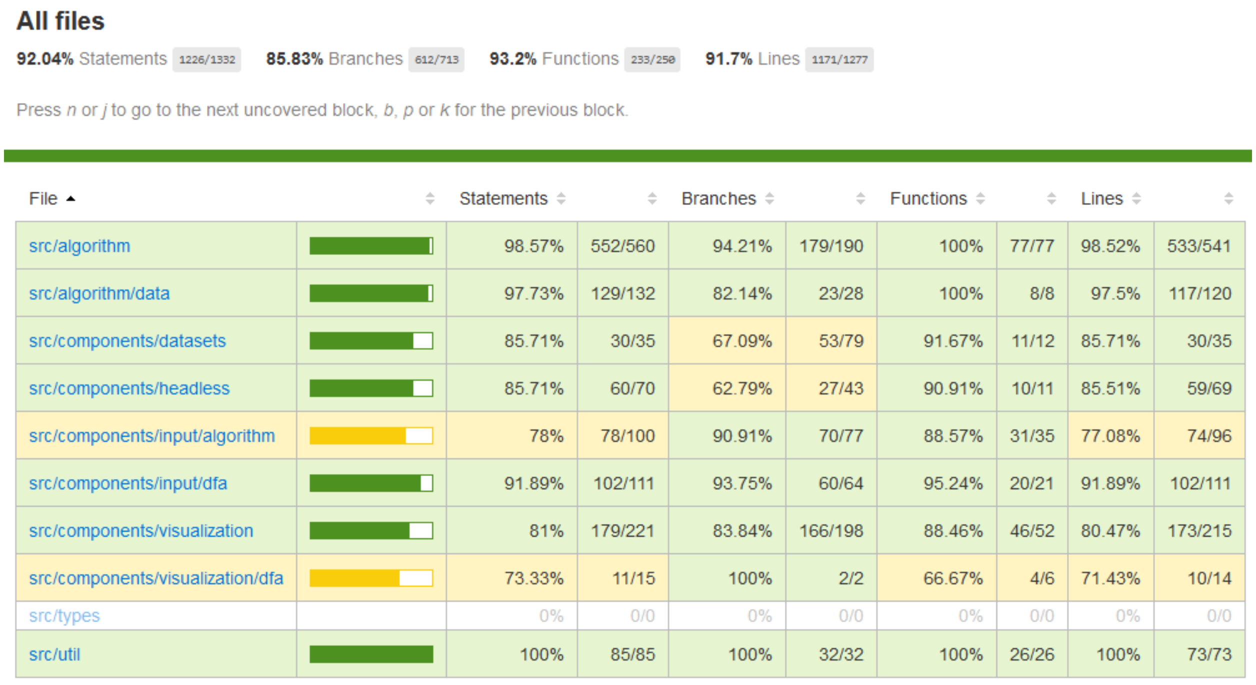
Figure 27. Screenshot of dataset generation.

5.4. Testing

The software was extensively tested, in-line with the agile principles. Whenever new features were added or bugs fixed, corresponding unit tests were written to ensure the new code worked as intended. Besides unit tests, component-level tests were also used to validate the complex interactions between individual components and ensure that data flows correctly between code units. In total, the codebase contains 33 test suites consisting of 84 individual tests.

Tests were implemented using the Jest test runner, which is an assertion-based framework with mocking features. Jest runs tests in parallel, which results in faster execution times and quicker feedback to developers. To test React components, the Enzyme testing framework was used. Enzyme enables creating React components by mounting them on a virtual page that only exists in memory and interacting with this page through code. React components were unit tested using Enzyme's shallow wrappers and larger component tests were done by mounting complex structures using Enzyme's mount wrappers.

After running tests, test coverage statistics were automatically generated using Istanbul. This coverage was manually checked to identify and fill any gaps in testing. In total, test coverage came out to 92% of statements and 85% of branches tested, well above the industry standard of 80%.



All files

92.04% Statements 1226/1332 **85.83%** Branches 612/713 **93.2%** Functions 233/250 **91.7%** Lines 1171/1277

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File ▲		Statements		Branches		Functions		Lines	
src/algorithm		98.57%	552/560	94.21%	179/190	100%	77/77	98.52%	533/541
src/algorithm/data		97.73%	129/132	82.14%	23/28	100%	8/8	97.5%	117/120
src/components/datasets		85.71%	30/35	67.09%	53/79	91.67%	11/12	85.71%	30/35
src/components/headless		85.71%	60/70	62.79%	27/43	90.91%	10/11	85.51%	59/69
src/components/input/algorithm		78%	78/100	90.91%	70/77	88.57%	31/35	77.08%	74/96
src/components/input/dfa		91.89%	102/111	93.75%	60/64	95.24%	20/21	91.89%	102/111
src/components/visualization		81%	179/221	83.84%	166/198	88.46%	46/52	80.47%	173/215
src/components/visualization/dfa		73.33%	11/15	100%	2/2	66.67%	4/6	71.43%	10/14
src/types		0%	0/0	0%	0/0	0%	0/0	0%	0/0
src/util		100%	85/85	100%	32/32	100%	26/26	100%	73/73

Figure 28. Screenshot of Istanbul test coverage report.

Besides automated regression testing, manual testing played a large role in this project. To implement the algorithms step-by-step, it was useful to quickly run them using a single command-line script. Because the project was written in TypeScript, normally every file would need to be compiled before execution, taking up to tens of seconds. To improve iteration speed, ts-node, a TypeScript-based execution engine was used. Using the command "npm run dev", a single TS file was run instead and provided near-instant feedback on the algorithms.

Arguably the most important part of the project, the algorithm implementations, were extensively tested using both black-box and white-box testing methodologies. For black-box testing, DFAs were generated using each of the dataset templates described in chapter 8 and were manually checked for equivalence and state minimization (using pen and paper), after which automated tests were written to check that the algorithms get the expected result. The same datasets were used for all algorithms to ensure that all of them had the same result for both state minimization and equivalence testing. These datasets are available in the project in the `src/algorithm/data/datasets.ts` file of the project. For white-box testing, the algorithms were run step-by-step, asserting at each state that the algorithm is in an appropriate state with properties (e.g. Table-Filling table) being consistent with constraints.

5.5. Optimization

The production build of the application is generated by Webpack, a JS build tool. The build process starts by compiling all TypeScript into JavaScript. The JS files are then optimized in terms of size using tree shaking (discarding unused dependency modules) and speed using traditional optimization methods (unrolling loops, inlining variables, *etc.*). Finally, Webpack transpiles the modern code into previous versions of JavaScript to ensure compatibility with browser that do not support modern standards, for example Internet Explorer.

Additionally, in the production build React is built and run in production mode. This mode of React is smaller and runs faster than the development build, although it provides less specific error messages.

The DFAs used for algorithms and algorithms themselves were optimized to use sets and maps where possible, to optimize look-up time. This proved crucial to implementing the algorithms as otherwise lookups would increase their time complexity.

Chapter 6

Professional and Ethical Issues

6.1. Ethical Concerns

The main ethical concern for this project is its educational purpose. Since the software is intended to be used for teaching, it is critically important that the algorithms within must be implemented correctly. Otherwise the software would mislead students and would even be counterproductive to its original purpose. To ensure correctness, the software is extensively tested. Additionally, it is open source, meaning anyone can review the implementations and highlight or fix any possible problems.

Another concern arising from the educational nature of the software is that it should be possible for students to experiment with it and modify it as they wish. Since the project is licensed under the permissive MIT license, there are no legal restrictions on modification or redistribution of the software, rather it is encouraged.

6.2. British Computing Society Code of Conduct

The British Computing Society Code of Conduct (BCS CoC) sets out professional standards to be followed by IT professionals (BCS Code of Conduct 2021). This project was implemented with the CoC in mind and its guidance was followed in all appropriate situations.

The Code of Conduct consists of four distinct chapters:

1. Public Interest. Professionals must respect others and their health, privacy, security *etc.* and must discriminate based on race, sex, age, religion, or any other condition.

Though the project was created by a single person, its software might be used by many. Extra precaution was taken to create the application in a manner where it is usable and understandable by everyone, despite their cultural, ethnic, or any other kind of other background. For example, the software uses generally accepted standards wherever possible (*e.g.* navigation with colored links and buttons) and is built to be compatible with all modern browsers, to not limit usability to only the author's system.

2. Professional Competence and Integrity. All work done most be within competence levels. Professionals must continuously develop themselves and reject unethical behaviour.

This project was implemented solely by the author, working to the best of their abilities. No intentional shortcuts were taken and all opportunities were taken to develop abilities, for example by using libraries that might have been unfamiliar to the author, yet useful for the project.

3. Duty to Relevant Authority. All work must follow local laws and regulations.

The implementation of this project followed all relevant laws and regulations, including UK laws and King's College London regulations. All software/libraries used were checked to be licensed under conditions compatible with the MIT license, under which the project's software is licensed.

4. Duty to the Profession. Professionals must uphold the reputation of the profession and work to improve it.

This project was implemented to the best of the author's abilities and highlights the kind of interactive software professional developers are able to create. Additionally, to improve the profession it is open source, and therefore contributes to the ever-growing software archives others can use for examples and build upon. Finally, the profession's collective competence was improved by the author's contribution to the official TypeScript types repository described in chapter 5, from which others will benefit.

Chapter 7

Experimentation

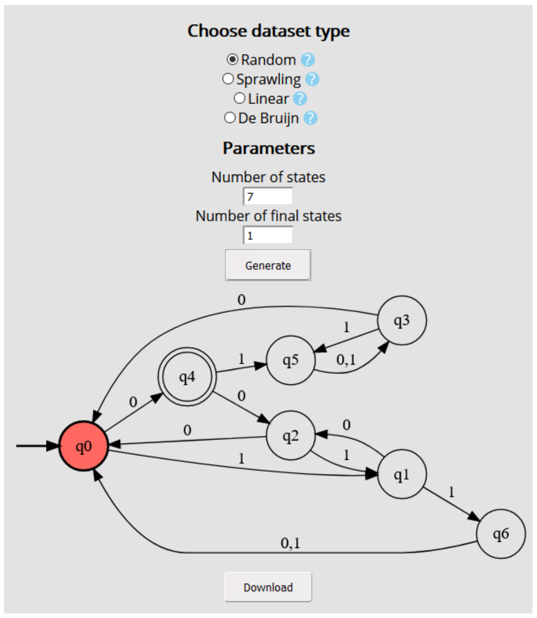
7.1. Datasets

This section details the dataset templates that were created to induce worst-case time complexity in the implemented algorithms. It aims to demonstrate that the templates do indeed cause worst-case performance by showing the process of creating the datasets using knowledge of the algorithms and comparing them with other datasets that do not cause worst-case performance.

Random dataset

The random dataset template was created to generate unstructured and unbiased DFAs to be used as reference cases. These DFAs were used test that all algorithms return the same results for equivalence testing and state minimization for complex and large DFAs. Additionally, random datasets were used to generate reference values to compare running times with datasets that cause worst-case performance.

Random datasets are guaranteed to form a connected DFA, *i.e.* all states are reachable from the starting state. This is achieved by maintaining two sets, one of all states connected to the starting state that have open (uninitialized) transitions and another set of states that are not yet connected to the starting state. A loop iterates over the unconnected states, choosing a random uninitialized transition and setting it to the unconnected state. After connecting all states, the remaining unitialized transitions are set to random states.


Choose dataset type

[✓ Random] Sprawling Linear De Bruijn

Parameters

Number of states: 7

Number of final states: 1

Figure 29. Visualization of Random dataset.

Table-Filling Algorithm (Linear dataset)

As discussed in chapter 2, to have worst-case running time the Table-Filling Algorithm must mark the lowest number of pairs possible in each iteration, while marking the maximum number of pairs over all iterations.

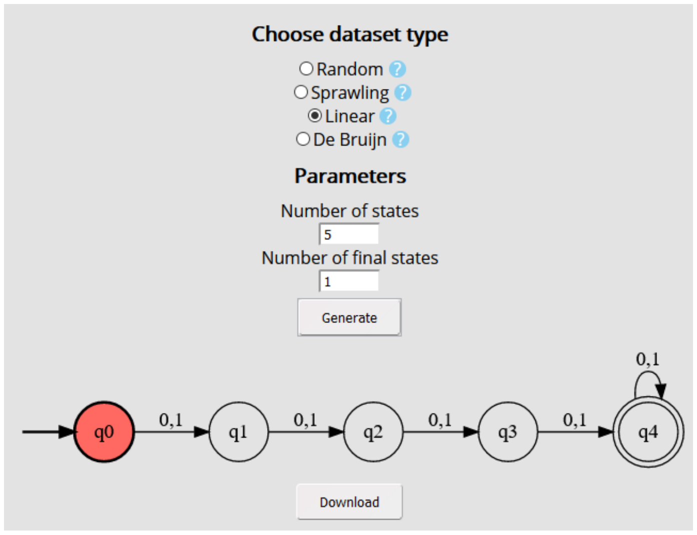
The Table-Filling Algorithm first marks pairs in step 4 of the its pseudocode (available in Figure 9), where all pairs where one state is accepting and the other is non-accepting are marked. The number of such pairs is $c = (n * m)$, where n is the number of non-accepting states and m is the number of accepting states. Since all states are must be either accepting or non-accepting, $n + m$ is the total number of states in the input.

To induce worst-case behavior, states must be divided into accepting and non-accepting ones in a way that that c is minimized. For example, if half the states were assigned to be accepting and the other half were non-accepting then $c = 0.5(n + m) * 0.5(n + m) = 0.25(n + m)^2$ pairs would be marked. Since this number grows quadratically with the number of states, many pairs are marked initially, and the algorithm runs relatively fast. The quadratic scaling of the number of pairs marked is always present whenever the number of accepting/non-accepting states is assigned to be some proportion of total states.

Instead, if the number of accepting states is assigned to be a constant $m = x$, then only $m * n = x * n$ pairs are marked, a number that grows linearly with the number of states. To mark the smallest possible number, x should be as low as possible. If $x = 0$, then no states would be marked and the algorithm would terminate, therefore it is best to have a single final state in the dataset, $x = 1$, in which case $n - 1$ pairs are marked initially.

The next time pairs are marked is in the first iteration of pseudocode step 5, where all pairs where one state leads to the accepting state and one does not are marked. This number of such pairs is minimal if there is only one state that leads to the final state (analogous logic to having a single final state). From that point, all pairs are marked where one state leads to either the final state or the only state leading to the final state and the other state transitions to any other state. This is again minimal if there is only one such state. Continuing with this logic, states must be arranged to form a chain, where each state only leads to a single state, in which case each iteration marks the minimal number of states possible.

Since this arrangement forms a straight chain of states in the graph representation of the DFA, it is called the linear dataset template.


Choose dataset type

Random [✓ Linear] Sprawling De Bruijn

Parameters

Number of states: 5

Number of final states: 1

Figure 30. Visualization of Linear dataset.

Experimentation confirms this dataset indeed causes worst-case performance in the Table-Filling algorithm, having a running time that is an order of magnitude greater than for other datasets.

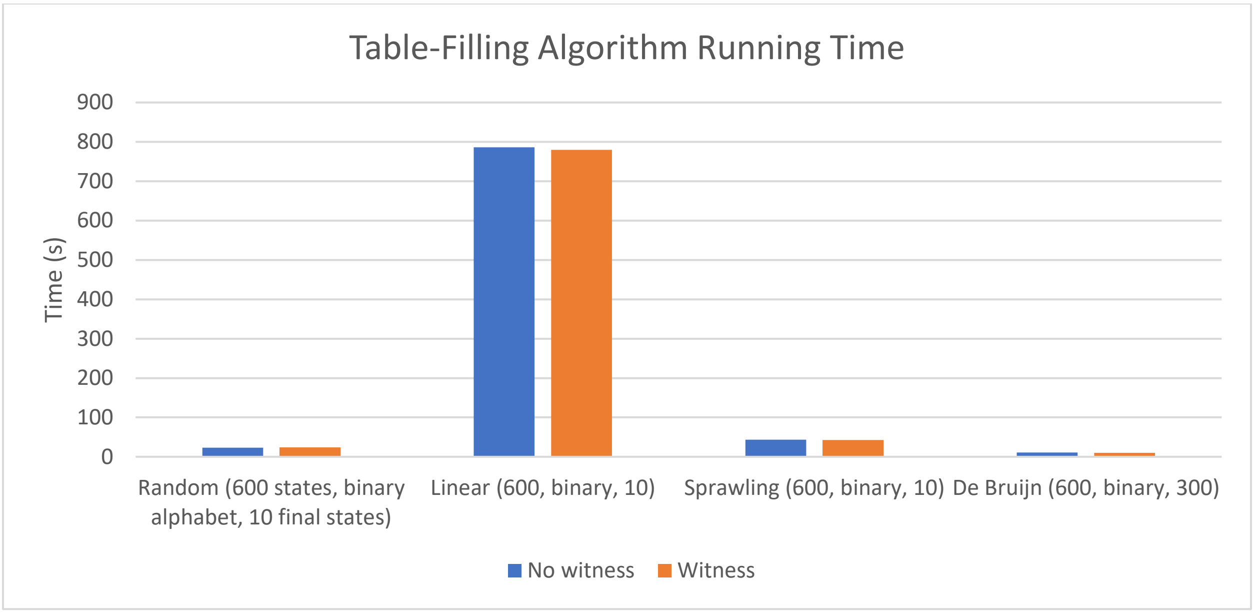


Figure 31. Running time of Table-Filling Algorithm.

Hopcroft Algorithm (De Bruijn dataset)

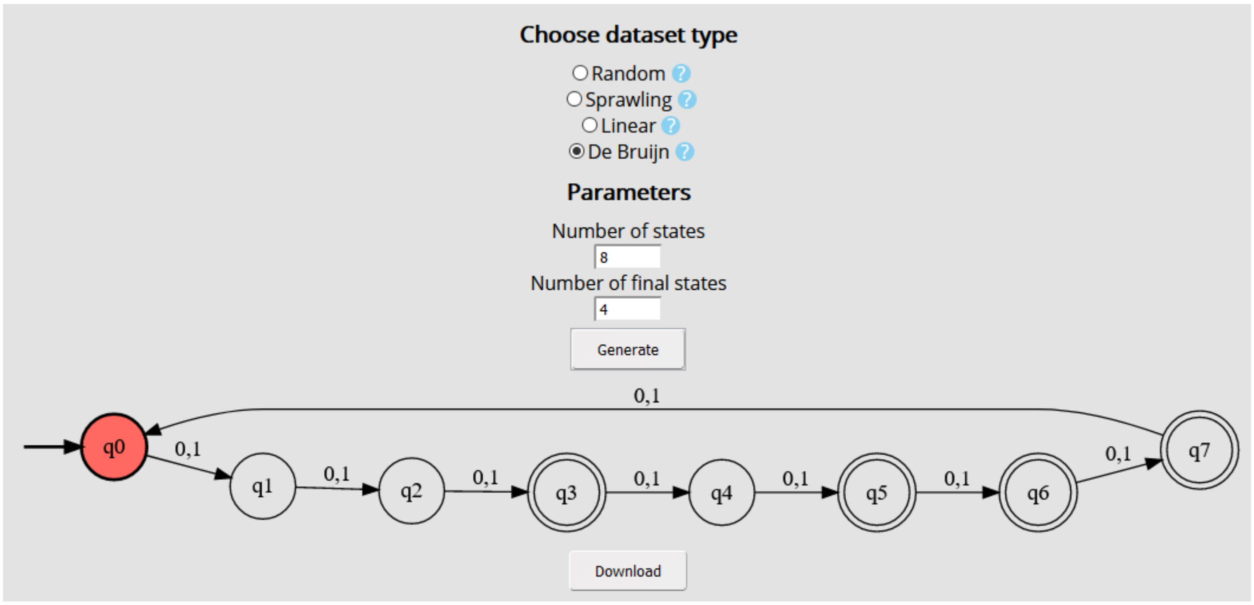
As discussed in chapter 2, a worst-case dataset for the Hopcroft algorithm is one in which half the states are final and half are non-final, and each state is distinguishable from all others. The transition function and allocation of final states must be constructed so that in each iteration of the algorithm's pseudocode (available in Figure 12) step 7 the blocks are halved.

These criteria are fulfilled by a DFA constructed from a binary de Bruijn word. A de Bruijn word w of order n over an alphabet Σ is a word in which each word of length n in Σ has exactly one circular occurrence in w. A word having a circular occurrence in w means it is present in a cycle formed from w in a way that the end of w is connected to its start. For example, 11101000 is a De Bruijn word of order 3 over the binary alphabet $[0,1]$ because a cycle formed by it contains all possible three-number binary sequences exactly once. (Berstel, Carton 2004)

A worst-case Hopcroft Algorithm DFA can be constructed by generating a de Bruijn word w over the binary alphabet and creating a DFA with as many states as there are characters in w. The transition function of that DFA must form a cycle, meaning states are connected as in linear datasets but the final state in the chain transitions to the first state. Accepting states are those corresponding to 1s in the de Bruijn word, *i.e.* if the n-th character of w is 1 then the n-th state in the transition cycle is an accepting state. (*ibid.*)

A de Bruijn word over the binary alphabet always has a length of 2^n, where n is some natural number, and half of its characters are zeroes and half are ones (*ibid.*). Therefore, to induce worst-case performance in the Hopcroft Algorithm, DFAs generated from de Bruijn words should also have 2^n states and have half of the states be final states.

However, the DatasetGenerator interface takes the number of states n and the number of final states m as parameters from the users generating datasets. The user parameters are respected by generating DFAs from de Bruijn words with a length $\geq n$ and final states are assigned to correspond to the first m instances of ones in the word. If $m > n/2$ then the first non-final states in the cycle are assigned to be final until there are m final states.


Choose dataset type

( ) Random
( ) Sprawling
( ) Linear
[✓ De Bruijn]

Parameters

Number of states: 8
Number of final states: 4

Figure 32. Visualization of de Bruijn dataset representing the word 00010111.

Experimentation confirms the de Bruijn dataset indeed causes worst-case performance in the Hopcroft algorithm, albeit barely. The complexity of the algorithm is $O(n \log n)$, where the first part is determined by the number of distinguishable states, and the second part is determined by the transition function and arrangement of final states. Because $n \gg \log n$, random datasets which have all states as distinguishable and de Bruijn datasets still have very similar running times.

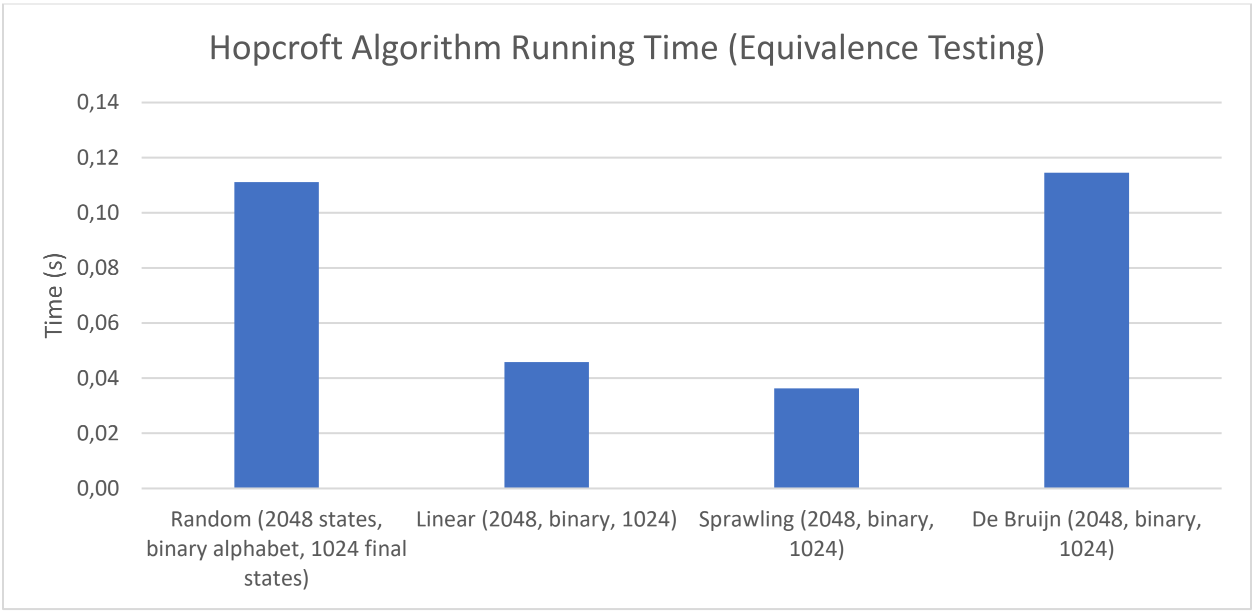


Figure 33. Running time of Hopcroft Algorithm.

The witness variation of the Hopcroft Algorithm works similar to the regular variation, except whenever a block is split, the symbol distinguishing each pair of states in the two created blocks is stored in a table. This causes the algorithm to run in $O(n^2)$ time, which increases running time by several orders of magnitude, though relative performances of datasets remain the same.

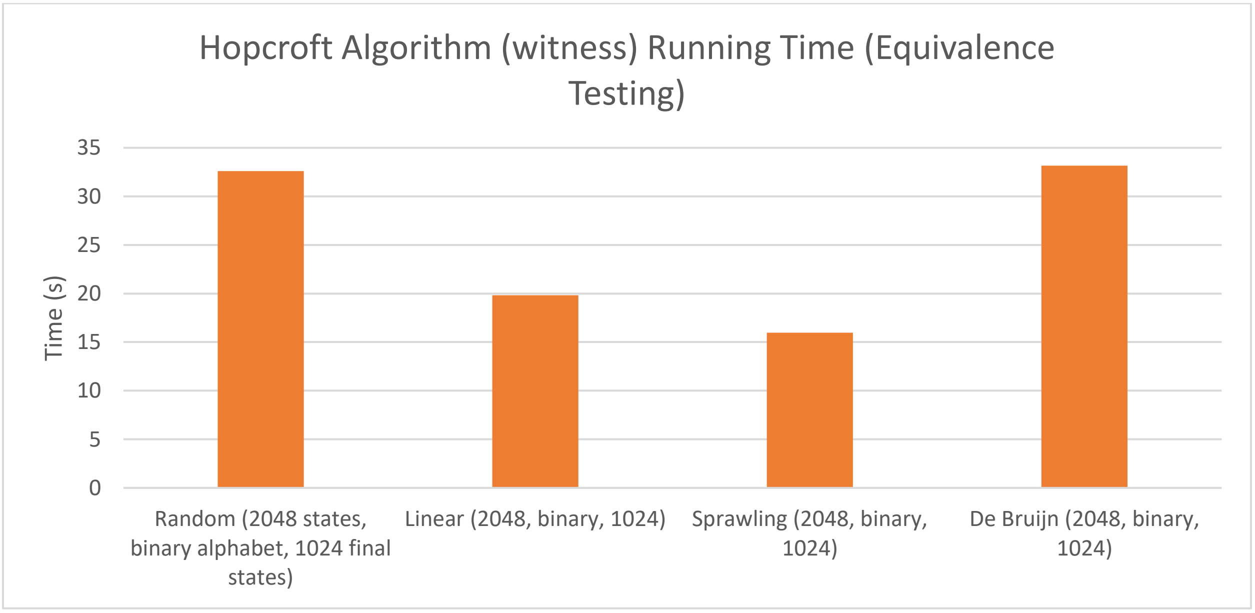
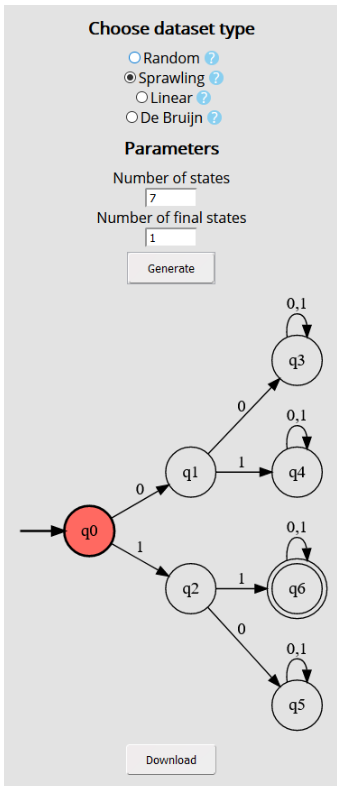


Figure 34. Running time of Hopcroft Algorithm (witness).

(Nearly) Linear Algorithm (Sprawling dataset)

As explained in chapter 2, the worst-case performance for the (Nearly) Linear Algorithm does not depend on the transition function as all reachable states are visited. For the optimized witness variation of the algorithm, where the algorithm exits early, a n-ary tree induces worst case performance, where n is the length of the input's alphabet. This was implemented by setting the transitions of each state in the generated DFA to states at index ($i * alphabet.length + j$), where i is the index of the state and j is a loop variable iterating over the DFA's alphabet. Because the graph representation of such a DFA forms a full tree, it is called the sprawling dataset.



Choose dataset type

Random [✓ Sprawling] Linear De Bruijn

Parameters

Number of states: 7

Number of final states: 1

Figure 35. Visualization of Sprawling dataset.

Experimentation revealed that for the unoptimized variation of (Nearly) Linear Algorithm, the sprawling, linear, and de Bruijn datasets had nearly identical performance, as all states were visited. However, the random dataset had worse performance than other datasets. This was due to the way the V8 JavaScript engine handles memory allocation and garbage collection. In the random dataset, all states were grouped into a single set and intermediate sets were larger, whereas for other datasets states were grouped into sets of two. Because the random dataset used larger sets in the algorithm, their allocation and garbage collection took longer and happened more often. Still, for examples of >32000 states, the time difference between random and other datasets was less than 0.1 seconds.

In the optimized witness variation, the algorithm execution was fastest using the random dataset, while it was of similar magnitude for other datasets. This is because the algorithm exits early whenever a contradiction is found, which happens relatively early for the random dataset, while for other dataset the DFAs compared are equivalent and the algorithm runs to completion. The reason the algorithm runs slower than the non-witness version is that maintaining the witness map takes extra computation time.

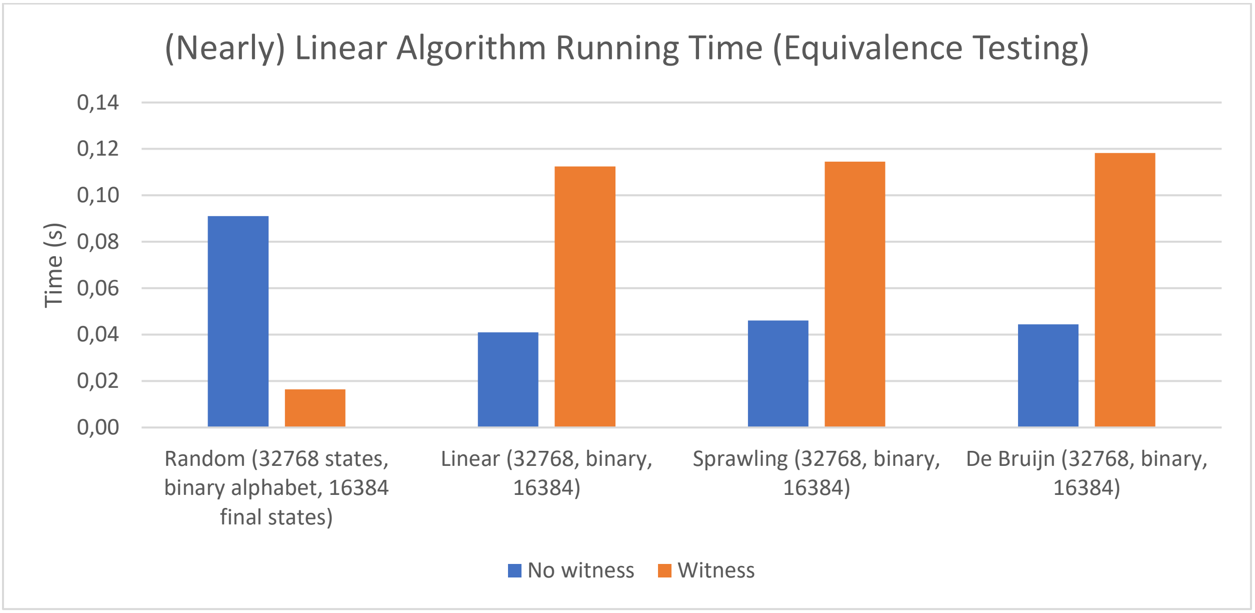


Figure 36. Running time of (Nearly) Linear Algorithm.

7.2. Comparison of Algorithms

Two DFAs generated using the random dataset template with 600 states, the binary alphabet, and 10 final states were used to test the performance of the algorithms relative to each other. The random template was used to avoid bias and get an average case for running every algorithm. The DFAs' size was chosen to be large enough to get a proper sample from the (Nearly) Linear Algorithm but small enough to run the Table-Filling Algorithm in manageable time.

Each algorithm was run 50 times on an air-cooled Intel i7-8560U CPU on a fully charged, plugged in laptop with no background tasks or GUI apps running. The first 5 runs of each algorithm was excluded from final results to account for the JS interpreter warming up.

The results were unsurprising with the Table-Filling Algorithm running the slowest and the (Nearly) Linear Algorithm being the fastest. However, what was surprising was the orders of magnitude of difference in their running times. The Table-Filling Algorithm had an average running time of 22.7 seconds while the (Nearly) Linear Algorithm ran in 0.6 milliseconds, a difference of almost 40000 times. The witness variations of the two algorithms had similar running times, with the Table-Filling (witness) having an average running time 23.7 seconds and the (Nearly) Linear (witness) running in 0.4 milliseconds.

The Hopcroft Algorithm had an average running between the other two algorithms at 32 milliseconds, however its witness variation was much slower at 2.1 seconds.

A chart of the average running times plotted on a logarithmic scale can be seen in Figure 37.

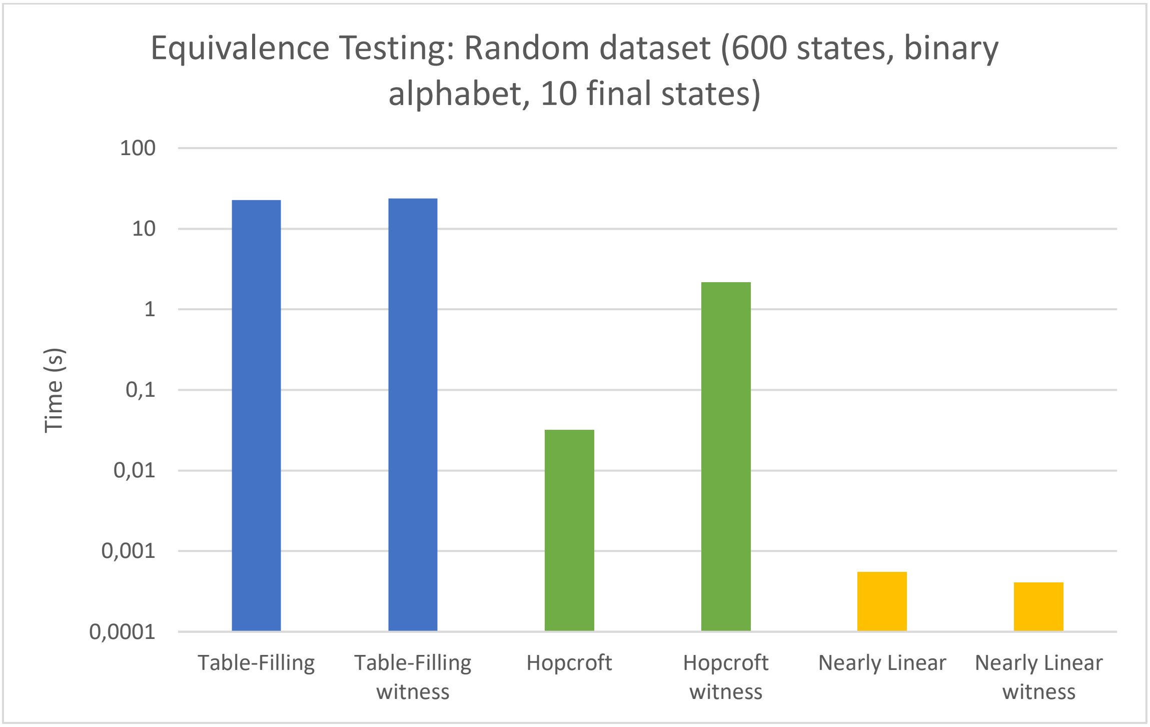


Figure 37. Average running time of algorithms.

Chapter 8

Evaluation

This chapter aims to establish the quality of the project and verify and validate that it fulfills the original brief and requirements laid out in chapter 3.

8.1. Software

8.1.1. Brief

The brief in section 3.1 stated "*The primary goal of this project is to create a piece of educational software that end-users can use to learn about and run DFA equivalence testing and state minimization algorithms and also demonstrate those algorithms' worst-case time complexities using concrete examples.*"

The objectives outlined in the brief were achieved, with the algorithms described in section 2.4 fully implemented and visualized for equivalence testing, state minimization (where applicable) and witness generation. Every algorithm has a corresponding worst-case dataset template and concrete examples were created using dataset generators. Headless mode was implemented to compare algorithms.

8.1.2. Requirements

Requirements for the software were divided into three categories, Front-End (F), Environment (E), and Algorithms and Datasets (A).

All Front-End requirements were met, including optional ones. The application has pages for visualizing algorithms, headless mode, dataset generation, and help. There is a dynamic DFA input system that adapts to user input and renders visualizations of DFAs in real time. DFAs can be saved to and loaded from files. All algorithms have visualizations and users can step through their execution, skip to the end, or reset execution. Users can navigate between algorithms and headless mode without having to re-input their data. Users can generate new DFAs from templates and the results are visualized. Generated DFAs can be used directly in algorithm visualization and headless mode. The application's design is minimalistic, responsive, and accessible.

All Environment requirements were met. The final minimized production build has a compressed (gzipped) bundle size of <500 kB, uses around 9 MB of memory when idle, and uses features supported by >99% of current browser usage. The app dynamically scales content for mobile devices and tablets.

All Algorithms and Datasets requirements were met. Every algorithm described in section 2.4 was implemented and testing confirms that all algorithms return the same answers for equivalence testing. Every algorithm has a bespoke and interactive visualization. Every algorithm has a dataset template corresponding to its worst-case time complexity. Users are provided with pre-generated datasets generated from these templates as well as access to the generators.

In terms of overall software quality, the application was built to a high standard. The code was structured into reusable components and automated tools like prettier and eslint were used to ensure the code is properly formatted and follows best practices. An extensive test suite was written with a test coverage of over 92% of statements which verifies that the app works as intended.

8.2 Experimentation

Experimentation was conducted with a variety of datasets and a sufficiently large sample size to make conclusions. The results of experimentation were validating and matched expectations, both in terms of algorithm implementations and datasets.

Algorithms' running times matched their time complexities – the (Nearly) Linear algorithm was orders of magnitude faster than the Hopcroft Algorithm, which was faster its witness variation, which itself was far faster than the Table-Filling Algorithm.

The algorithms performed as expected with custom datasets. Each dataset caused worst-case performance in the algorithm they were designed for, sometimes being orders of magnitude slower than others. The benchmarking results were easily explainable and the random dataset consistently ran faster than worst-case datasets.

Chapter 9

Conclusion and Future Work

9.1. Conclusion

Finite automata remain prevalent in the modern world and find use in a broad array of fields, including text processing, hardware design, biology, and genetics. The topic of minimizing automata to ensure they consume the least amount of resources, both virtually and physically, is therefore also relevant, as is equivalence checking to ensure that minimization was done correctly.

Though algorithms and tools for these procedures have existed for decades, there is a gap of availability for such applications, especially on the web. This project addresses the gap in opportunities by providing a web-based method of DFA state minimization and equivalence testing. Educational by nature, the application not only implements algorithms but also visualizes them. Overall, the application was made with modern design principles in mind, resulting in a more interactive, sleek, and lightweight experience than similar existing systems.

Three popular DFA equivalence testing and state minimization algorithms were successfully implemented in this project, namely the Table-Filling Algorithm, The *n lg n* Hopcroft Algorithm, and the (Nearly) Linear Algorithm. Additionally, each algorithm had an optional witness mode implemented which clarifies why DFAs being tested are not equivalent.

The algorithms were implemented using a state pattern which enables step-by-step execution. This is used for interactive visualization which users can step through at their own pace. A second way of running the algorithms is provided through headless mode, in which users can run multiple algorithms on the same input without visualization and compare their results and running times.

Custom DFAs were created for each algorithm to demonstrate their worst-case time complexity. These DFAs were made by reusable dataset generator templates from which users can generate arbitrarily large DFAs and use them to compare algorithms with each other or datasets within specific algorithms.

Overall, the project was a success. All requirements were achieved, including optional ones. Measurements from experimentation also confirmed that all algorithms and datasets work as intended and extensive testing and tooling verify the software is of high quality.

9.2. Future Work

Although the project achieved all requirements, there are a variety of extensions that can be done to enhance the project further, for example:

- There are many other DFA state minimization and equivalence testing algorithms, each with their own strengths and weaknesses, that were not implemented in this project. Instead, the project provides a framework into which other algorithms could be easily added in the future.
- This project is only concerned with deterministic finite automata, but the world of finite automate is much larger. For example, algorithms working on non-deterministic or push-down automata could also be supported.
- There are other kinds of DFA datasets that could cause interesting behavior in the algorithms, but further research and analysis is needed to identify and implement them.
- Algorithm visualizations could be more dynamic and gradual, for example by using animations. In the case of the Table-Filling Algorithm, cells could pop out and be highlighted as they were filled in and for the Hopcroft algorithm table rows could split into two as blocks are partitioned.
- Adding graphs to the result of headless mode. Currently users only get feedback on the time taken to run the algorithms and have to visualize results themselves.

References

Agile Alliance. Agile 101, 2021. Accessed 6 February 2021. https://www.agilealliance.org/agile101/

Alfred V. Aho, John E. Hopcroft, Jeffery D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

British Computing Society. BCS Code of Conduct. Accessed 12 March 2021. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/

Daphne A. Norton. Algorithms for Testing Equivalence of Finite Automata, with a Grading Tool for JFLAP. M. S. Project Report, Rochester Institute of Technology, Department of Computer Science, 2009.

David Doty. CV, 2020. Accessed 26 November 2020. https://web.cs.ucdavis.edu/~doty/david-doty-cv.pdf.

David Gries. Describing an algorithm by Hopcroft. Technical Report TR 72-151, Cornell University, December 1972.

Edward F. Moore. Gedanken-experiments on sequential machines. Automata studies, Annals of mathematics studies 34:129-153. Princeton University Press, Princeton, New Jersey, 1956.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Filippo Bonchi, Damien Pous. Checking NFA equivalence with bisimulations up to congruence. ACM SIGPLAN Notices 48(1), 2013.

Gertjan van Noord. FSA6.2XX: Finite State Automata Utilities. Accessed 24 November 2020. https://www.let.rug.nl/~vannoord/Fsa/.

Ivan Zuzak, Ivan Budiselic. Noam. GitHub repository, 2017. Accessed 26 November 2020. https://github.com/izuzak/noam.

Ivan Zuzak, Vedrana Jankovic. FSM simulator, 2017. Accessed 3. December 2020. http://ivanzuzak.info/noam/webapps/fsm_simulator/.

Jean Berstel, Olivier Carton. On the Complexity of Hopcroft's State Minimization Algorithm. Implementation and Application of Automata, 9th International Conference, Kingston, Canada, 2004.

JFLAP version 7.1. JFLAP Web site. Accessed 24 November 2020. http://jflap.org/.

John E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71-114, Cornell University, December 1971.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston, third edition, 2006.

John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical Report STAN-CS-71-190, Stanford University, January 1971.

Kyle Dickerson. automatonSimulator. GitHub repository, 2018. Accessed 26 November 2020. https://github.com/kdickerson/automatonSimulator.

MDN Web Docs. Understanding client-side JavaScript frameworks, 2020. Accessed 15 December 2020. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks.

Michael S. Mahoney. The Structures of Computation and the Mathematical Structure of Nature. Rutherford Journal, volume 3, 2010.

Noam Chomsky. On certain formal properties of grammars. Information and Control 2(2): 137–167, 1959.

Oracle. Java Client Roadmap Update. March 2018.

React. React – A JavaScript library for building user interfaces, 2020. Accessed 15 December 2020. https://reactjs.org/.

Red Hat. What is CI/CD?. Accessed 28 February 2021. https://www.redhat.com/en/topics/devops/what-is-ci-cd

Robert Constable. The Role of Finite Automata in the Development of Modern Computing Theory*. *The Kleene Symposium*, pages 61-83. North-Holland Publishing Company, 1980.

S. L. Kryvyi. Finite-State Automata in Information Technologies. *Cybernetics and Systems Analysis*, 47(5):669-683. Taylor & Francis, 2011.

Sten Laane. hashmap: add undefined as return type to get function. GitHub Pull Request, 2021. Accessed 28 February 2021. https://github.com/DefinitelyTyped/DefinitelyTyped/pull/51282

Timothy M. White, Thomas P. Way. jFAST: A Java Finite Automata Simulator. ACM SIGCSE Bulletin(1):384-388. Association for Computing Machinery, 2006.

Typescript. Typed JavaScript at Any Scale. Accessed 6 February 2021. https://www.typescriptlang.org/