Determinization and Minimization of Non-Deterministic Finite State Automatas - A Distributed Approach

A. Guerville

March 24, 2023

Contents

1 Introduction

Finite State Automata are one of the most fundamental concepts in Computer Science. Their uses range everywhere from parsing to mathematical theory. Finite State Automata exist in two kinds: Deterministic Finite Automata (DFAs) and Non-Deterministic Finite Automata (NFAs).

It is well-known that both structures describe the same set of languages (regular languages), but it is in general significantly easier to work with DFAs than NFAs. Some problems are first transcribed into NFAs, therefore the determinization of a DFA into a NFA, and it's minimization, are critical steps into the understanding of those problems. However, both those problems are computationally complex and memory intensive, hence the need to do it efficiently, and to distribute the computation towards multiple systems.

In this paper, is achieved a program that implements a set of determinization and minimization algorithms, ranging from single-threaded to distributed, in the goal of achieving fast, efficient, and scalable determinization and minimization of NFAs. Further algorithms and solutions are described, and some implemented, that try to achieve that task. The algorithms are then tested and compared on a problem class about the determinization of NFAs: Transportation Graphs AKA Token Passing Networks.

Finally, the topic of determinization and minimization is explored for higher classes of machines, such as Pushdown Automatas.

2 Related Works

Single-threaded NFA determinization and minization algorithms have existed since the 1950s. DFA determinization's *Rabin-Scott superset construction* algorithm is a well-known determinization algorithm which has existed for a long time. However, DFA minimization is younger, and the most well-known minimization algorithm today is Hopcroft's minimization algorithm [**?**]. However, this algorithm becomes less useful as automatas get larger, and the need for parallelization arises.

Parallel NFA determinization algorithms have begun being researched round the 1990s. For example, [**?**] ran a parallel NFA determinization and minimization algorithm on a supercomputer, using a message passing model instead of shared memory. In 2007, [**?**] implements a disk-based distributed algorithm for large NFAs. A disk-based approach avoids the RAM memory space issues from previous implementations. Later, [**?**] proposes a general programming model to migrate RAM-based legacy algorithms into parallel disks - and applies the model to NFA minimization.

Finally, the technologies and distribution models of the 2010s have affected the way distributed computing is done in the case of NFA determinization and minimization. In 2020, [**?**] uses Bulk Synchronous Parallel abstract computer model to implement a more performant distributed NFA determinization and minization algorithm. Later, in 2022, [**?**] compares both the MapReduce and a Bulk Synchonous Parallel-based NFA determinization and minimization algorithm, finding that the BSP/Pregel based solution outperforms the MapReduce solution.

3 Definitions

To understand the presentation of the algorithms in this paper, and most notably the definitions of the minimization algorithms, it is useful to present the definitions that will be used thereafter. Those are as such:

Finite State Machine - Let $M = (S, \Sigma, \delta, S_0, T)$ be a deterministic finite state machine (DFA) where:

- S is the set of all states in M
- Σ: Alphabet,
- $\delta \in S \times \Sigma \times S$: Transition function,
- S_0: Starting states,
- T: Accepting states.

Nondeterministic Finite State Machine - let $N = (S, \Sigma, \delta, S_0, T)$ be a non-deterministic finite state machine (NFA) where:

- S: Set of states,

- Σ: Alphabet,
- $\delta \in S \cup \epsilon \times \Sigma \times P(S)$: Non-deterministic transition function - where $P(S)$ is the powerset of S,
- S_0: Starting states,
- T: Accepting states.

FSM Language [**?**]

- Let $L(M)$ be the language that the finite state machine M describes. For each state $s \in S$, it is

possible to define some subautomaton of M rooted at Let $L_s(M)$ be the set of words recognized by the subautomaton of M rooted at state s, or the *future* of state s. In a finite state machine, 2 states p and q can be considered as equivalent $p \equiv q$ if and only if the *futures* of p and q are the same, that is:

$$p \equiv q \Leftrightarrow L_p(M) = L_q(M)$$

This relationship is called the *Nerode congruence*.

Hence, a **minimal automaton** is definable as some DFA $M = (S, \Sigma, \delta, S_0, T$, such that:

$$\forall p \in S. \forall q \in S. L_p(M) \neq L_q(M)$$

That is, no two state in a minimal automata may share the same future. All DFA minimization comes down to is the task of computing Nerode equivalences between sets of states, until no 2 states from 2 different sets of states are Nerode equivalent, upon which a new automata is defined with each of the given sets of states as states.

Reverse Transitions - For $s \in S, a \in \Sigma$, Let $\delta^{-1}(s, a) = s^{'} \in S | \delta(s^{'}, a) = s$ be the reverse transitions of s for character a. Let $\mathcal{P}$ be a partition - a collection of sets of states, such that $\forall s \in S. \exists! P \in \mathcal{P}. s \in P$, that is all states of M must be unique in the partition. Furthermore, let $\delta(P, a), a \in \Sigma$ be the set of states for which a transition from a state in P lead to. Likewise, let $\delta^{-1}(P, a)$ be the set of states for which a transition to a set in P leads from.

We can say between 2 sets $P, R \in \mathcal{P}$, that the set R can be *split* by $\delta^{-1}(P, a)$ into 2 sets:

$$R \cap \delta^{-1}(P, a) = s \in R | \delta(s, a) \in P \text{ and } R \backslash \delta^{-1}(P, a) = s \in R | \delta(s, a) \notin P$$

It is important to remark that R can be split to either 1 or 2, depending on whether all the states in R are in $\delta^{-1}(P, a)$, or if none of them are in $\delta^{-1}(P, a)$.

With splitting, the Nerode equivalence in the partition $\mathcal{P}$ can be defined as the smallest partition for which no combination of a set in the partition and a letter in Σ can split another set in the partition in 2.

With those definitions in mind, it will be easier to understand and reason about determinization and minimization throughout the paper.

4 The `nfdeterminize` System

The solution that is implemented for this research is a program named `nfdeterminize`. `Nfdeterminize` is a command-line application that takes in a finite state machine, either as a file, or by generating a token passing network. Then, it can either determinize the given NFA, minimize a DFA, or run both functionalities to convert an NFA into a minimal DFA.

4.1 Features

As an NFA determinization and minimization solution that specialises in dealing with token passing network automatas, `nfdeterminizes` supports the following features:

- Loading automatas from file under GAP's printing style for finite state automatas,
- Generating and loading automatas out of token passing network patterns for direct use by the program,
- Determinizing and minimizing any given finite state automaton,
- Choosing between algorithms, sequential, multi-threaded and multi-processing implementations for determinization and minimization.
- Outputting the result either on standard output, or to a file that is under GAP's automata format, such that the automata may be loaded onto GAP later on.

4.2 Language

The software for this project is written in Rust. Rust[1] is a performance-focused and memory-safe programming language with a modern approach to memory management by enforcing memory safety. It is a relatively new language, having only appeared in 2015, and it's distributed computing support is at the current moment relatively limited. However, it's ecosystem is maturing fairly quickly over time, and has a wide community.

The main reason why Rust is chosen here over other languages is for it's commitment to performance and memory safety - it performs as well if not better over certain cases than C or C++, but guarantees memory safety by checking for inconsistencies and memory lifetime at compile time. Furthermore, Rust also deals with concurrent access issues at compile time, which takes away a lot of strain during development of multi-threaded or multi-process algorithms.

Some other potential languages for implementation are C++, Java or Go.

C++ is a high-performance language as well, with a mature ecosystem, but does not deal with memory as well as Rust does. Furthermore, Rust, being a more modern language, has in general more efficient built-in implementations of data structures, such as Rust's `HashMap`, implemented using the swiss table [2].

[1] `https://rust-lang.org`
[2] `https://abseil.io/about/design/swisstables`

Go is another programming language with performance and memory safety in mind. The main advantage of Go is how it easily handles concurrency by using goroutines. Furthermore, Go has very solid support for the Docker API and Kubernetes, making it a good choice for multi-threaded or distributed workloads. However, Go uses a garbage collector during runtime, and in general performs considerably slower for same programs than Rust or C++ would.

Finally, Java is a high-level, object oriented programming language. Java is the only language in this list that is not strictly a compiled language, compiling to bytecode that is run on the JVM, instead of running directly on the machine. Java is not a good choice for this application, as using Java and the JVM comes at severe performance costs.

Overall, Rust was chosen for `nfdeterminize` for the following factors:

- No sacrifice between memory safety and performance,
- Good multithreaded suppport thanks to compile time concurrent access checking, shared memory and message-based concurrency support,
- Modern and easy dependency management with `cargo`[3],

4.3 Distributed Computing in Rust

Rust being a relatively new language, it has at the current moment limited support for Docker. Some small projects exist however, that wrap the docker API for Rust.

Aside from Docker, Rust processes can still communicate between each other using either `ipc-channel`[4], or using ZeroMQ[5]. Therefore, it is possible to make distributed systems running Rust, but managing the network cluster or processes may be more easily done using another language such as Python. Nevertheless, Rust is a nice solution to implement the performance-critical parts of a distributed system.

In the context of `nfdeterminize`, a distributed system is simulated by spawning processes on the same machine, and having them communicate using ZeroMQ. While `ipc-channel` passes file descriptors over sockets under Unix for inter-process messaging, ZeroMQ is more complete, as it supports a lot more kinds of transports such as TCP, which `nfdeterminize` uses.

5 Sequential Approach

5.1 Approach to Determinization

First of all, NFA determinization is a well-known process, and efficient algorithms for it have existed for a long time. The most widely-used algorithm for determinization is the

[3] `https://doc.rust-lang.org/cargo`
[4] `https://github.com/servo/ipc-channel`
[5] `https://zeromq.org`

superset construction algorithm, which explores the NFA from node to node, keeping track of the sets of states visited in a map, until we've explored all reachable nodes.

The major advantage of this algorithm over any other is that it only explores reachable states in the NFA, and produces only reachable states in the resulting DFA. The consequences are two-fold:

- 1. The amount of exploration involved is severely decreased, depending on the NFA that is determinized,
- 2. There is no need to remove unreachable states from the resulting DFA after determinization and before minimization.

The algorithm possesses shared memory in form of M, the structure that maps a kept set of states to the number that it is assigned on the final DFA, because the algorithm needs to check if a state has already been found after producing it.

Algorithm 1 Rabin Scott's Superset Construction Algorithm

```
1:  procedure SupersetConstruction(M = (S, Σ, δ, S_0, T))
2:      M ←[(S_0, 0)]
3:      T' ←[]
4:      if ∃s ∈ S_0.s ∈ T then
5:          T' ←[S_0]
6:      end if
7:      F ←[S_0]
8:      while F ≠ ∅ do
9:          S_next ←pop from F
10:         for all a ∈ Σ do
11:             S' ←
12:             for all s ∈ S_next do
13:                 Add s and all ϵtransitions from sto S'
14:             end for
15:             if S_next ∉ M then
16:                 M ←[M, (S', |M|)]
17:                 if ∃s ∈ T.s ∈ S' then
18:                     T' ←[T', S']
19:                 end if
20:                 F ←[F, S']
21:             end if
22:             δ' ←[δ', (S_next, a, S')]
23:         end for
24:     end while
25: end procedure
```

Complexity-wise, the worst-case time complexity of the superset construction is $O(2^n)$,

where n is the number of states in the original NFA. Such worst-case is unavoidable as the size of the superset of states in the NFA $|P(S)| = 2^{|S|}$, where S is the set of states in the original NFA. However, this treshold is generally never reached, hence the purpose of the superset construction algorithm.

In terms of implementing the sequential version of the superset construction algorithm, most of the design decision comes in how to store sets of states, as a state should be able to describe one of 2^n possible states.

5.1.1 Storing Sets of States

The main challenge of superset construction implementation is not the implementation of the exploration algorithm, but rather how to represent states of superset construction. The issues stems from how in superset construction, there are about 2^n possibly reachable states, so it is required to find a fast and memory-efficient way to store such a state in a hash map. Furthermore, [**?**] states that in a 2 billion-state DFA, each DFA state may consist of upto 20 of the NFA states. Therefore, it is definitely required shorten the size of a state.

The solution implemented in the program is as such -

- During superset construction, when a new state is being searched, represent the set of states as an array of bits. This representation is useful as bitwise operations can be done upon it, for a low cost.
- Then, before hashing the set and inserting it to a hash map, compress the array. Here, the lz4 algorithm is used. The lz4 algorithm is a modern and fast byte array compression method that may simply return a byte array. It's main advantage is its speed compared to that of other compression algorithms, although it is not as size efficient.
- The compressed array is inserted into the hash map. State storage size has been decreased for a moderate speed cost.

In `nfdeterminize`, this data structure is defined as a `Ubig` struct, which stands for `unsigned integer`. It is defined in the `ubig.rs` source file.

5.1.2 ϵ transitions

It is important to note during determinization, that the majority of a token passing network's automaton's transitions are ϵ transitions. However, ϵ transitions in general contribute lightly to the final DFA, and take up a considerable amount of time when constructing a set of states during superset construction. There are 2 solutions to this problem:

- Caching the final set of states for all states in the NFA, and lookup the cache when the state is checked again later on,
- Removing ϵ transitions from the NFA before feeding it to the determinization algorithm.

However, these approaches present some severe downsides.

Firstly, caching takes uses extra memory for the problem, which may be better spent for supporting bigger automatas. Counterintuitively, this approach may also become slower for some NFAs as the time taken adding a set of states to the cache has to be taken into account. The `--cache` argument on the `run` and `determinize` commands add caching to the determinization process.

On the other hand, building a new NFA without ϵ transitions is a sort of transformation that can actually increase the size of the NFA that will be determinized. The disadvantages here are two-fold - first, extra time is taken making an new automaton, which may not be necessary for some NFAs. Second, increasing the size of the NFA may impact the time taken by determinization. Therefore, in many cases, ϵ transition free NFA construction may be slower than running NFA determinization with ϵ transitions taken into account. Nevertheless, the feature is still supported using the `--no-epsilon` argument on the `run` and `determinize` commands.

Overall, the performance of caching and ϵ transition free automata construction depend heavily on the kind of automata that is loaded to the program. In the context of token passing networks, it is preferable to generate automatas from specific patterns like two-stack or buffer-and-stack, for which running determinization directly is preferred.

5.2 Approach to Minimization

While NFA determinization has been a well-known subject for a long time, DFA minimization on the other hand has less well-known algorithms. Out of all the minimization algorithms nowadays, 2 stand out as better algorithms than the rest. Those are Hopcroft's algorithm and Brzozowski's minimization algorithm.

5.2.1 Hopcroft's Algorithm

Hopcroft's algorithm, made by J. Hopcroft in 1971[**?**], is the first, and probably the most well-known non-$O(n^2)$ time complexity DFA minimization algorithm. It is one of the first partition refinement algorithms.

Hopcrof't algorithm separates the states of the DFA into a partition of 2 sets - accept states and non-accept states. Those will be the states of the minimal automata by the end of the algorithm's execution. Then, until the frontier is empty, it searches for states in the partition for which the transitions lead to distinguishable states.

If it is the case, then it means the partition has to be divided further. The algorithm is repeated until all states in each partition contain states that are indistinguishable by their transitions, which means that the resulting DFA holds the same language than the original one, but at it's minimal size.

For definitions, let:

- $\mathcal{P}$: the partition to refine,
- $P \in \mathcal{P}$: a set of states in the partition.

Hopcroft's algorithm relies on the following lemma -

Lemma 1. *Let some finite state machine* $M = (S, \Sigma, \delta, S_p, T)$.
$\forall p \in S. \forall q \in S. \forall a \in \Sigma$, *let* $\delta(p, a) = p^{'}$, $\delta(q, a) = q^{'}$.
$p^{'}$ *and* $q^{'}$ *are distinguishable* $\Rightarrow$ p *and* q *are distinguishable.*

Therefore, Hopcroft's algorithm uses the reverse transitions of the next set in the frontier to establish distinguishability between states in a set of the partition. Distinguishibility is therefore defined as such, for some sets $V, P \in \mathcal{P}$, and $\delta^{-1}(P, a)$ the set of states $s \in Ss.t\delta(s, a) \in P$:

$V \cap \delta^{-1}(P, a) \neq \emptyset \wedge V \backslash \delta^{-1}(P, a) \neq \emptyset \Rightarrow V$ is distinguishible into $V \cap \delta^{-1}(P, a)$ and $V \backslash \delta^{-1}(P, a)$

Algorithm 2 Hopcroft's Algorithm

```
1:  procedure HopcroftAlgo(M = (S, Σ, δ, s_0, T))
2:      P ←[T, S\T]
3:      Q ←[T, S\T]
4:      while |Q| ≠ 0 do
5:          P_next ←pop Q
6:          for all a ∈Σ, V ∈P do
7:              if δ^-1(P_next, a) ∩ V ≠ ∅ ∩ V\δ^-1(P_next, a) ≠ ∅ then
8:                  remove V from P
9:                  push δ^-1(P_next, a) ∩ V into P
10:                 push V\δ^-1(P_next, a) into P
11:                 if V ∈ Q then
12:                     replace [V] in Q with [V\δ^-1(P_next, a), δ^-1(P_next, a) ∩ V]
13:                 else if |V\δ^-1(P_next, a)| ≤ |δ^-1(P_next, a) ∩ V| then
14:                     add V\δ^-1(P_next, a) to Q
15:                 else
16:                     add δ^-1(P_next, a) ∩ V to Q
17:                 end if
18:             end if
19:         end for
20:     end while
21: end procedure
```

Hopcroft's Algorithm, as shown on figure 5.2.1, has asymptotic time complexity $O(knlog(n))$[**?**], where:

- k: the number of input letters in the alphabet Σ,
- n: the number of states in the initial DFA.

This time complexity makes it the minimization algorithm that achieves the best possible time complexity.

Implementation-wise, the approach here is closer to the implementation described in [**?**], with some performance improvements. On line 6 of 5.2.1, instead of looking for all V in $\mathcal{P}$, it is possible to iterate through all partitions linked to a state in $\delta^{-1}(P_{next}, a)$, by keeping a map of what state is linked to which set in $\mathcal{P}$. Doing so avoids the lengthy process of iterating through $\mathcal{P}$ for every set P_{next} in the frontier.

On the rust implementation, sets are represented as ordered vectors. With ordered vectors, difference and intersection construction can be done in $O(n)$ time complexity, and ordered vector construction from inverse transformation is done in $O(nlog(n))$ time complexity, for n the size of the set. Using a vector instead of a set avoids the overhead gotten from consistently hashing values into a hash set.

Finally, the queue Q is done in a circular ring buffer as using contiguous memory, instead of a linked list, for faster memory access, while the partition is done as a simple contiguous memory array, as it is never needed to pop anything from it. Instead, adding to the partition is done by replacing V by $V \cap \delta^{-1}(P_{next}, a)$ and appending $V \backslash \delta^{-1}(P_{next}, a)$ to the end of $/\mathcal{P}$.

5.2.2 Moore's Algorithm

Hopcroft's Algorithm, described and implemented in 5.2.1, is considered as the most efficient partition refinement based minimization algorithm. However, the first partition refinement implemented is Moore's Algorithm, of which Hopcroft's Algorithm is based on.

Moore's algorithm [**?**] is an $O(n^2)$ time complexity algorithm for DFA determinization. It is the classical algorithm for partition refinement, which works as such:

- Get an initial partition of 2 sets - final states and non-final states,
- Until the partition doesn't change, repeat:
  - Produce a new partition $\mathcal{P}'$ s.t. $\forall p, q \in S.\mathcal{P}'[p] = \mathcal{P}'[q] \Leftrightarrow \mathcal{P}[p] = \mathcal{P}[q] \wedge (\forall a \in \Sigma.\mathcal{P}(\delta(p, a)) = \mathcal{P}(\delta(q, a)))$

The reasoning of this algorithm, is that the partition is iteratively split until $\nexists P \in \mathcal{P}.\exists a \in \Sigma.\exists Q \in \mathcal{P}$ splittable by $\delta(P, a)$, upon which it can be guaranteed it is the coarsest partition, and therefore that the minimal DFA has been found.

Moore's algorithm is not implemented here, as it's time complexity $O(n^2)$ makes it less appealing than Hopcroft's algorithm, although it is not as limitedly parallelizable as the latter.

5.2.3 Brzozowski's Algorithm

Brzozowski's algorithm is an exception to the general landscape of DFA minimization algorithms. Most minimization algorithms work by doing partition refinement, like Hopcroft's, and some work by fusion like Revuz's [**?**]. However, Brzozowski's algorithm works, for some

finite state machine $M = (S, \Sigma, \delta, S_0, T)$, by determinizing $M^R = (S, \Sigma, \delta^{-1}, T, S_0)$, where δ^{-1} is the table of inverse transitions from M. Then, perform

determinization of $(M^R)^R$. The result of the determinized $(M^R)^R$ is the minimal DFA representation of M.

This algorithm is very easy to implement as determinization has already been implemented beforehand. However, as with determinization, it has an exponential time complexity.

Performance-wise however, Brzozowski's is known to outperform other minimization algorithm in particular cases, so it is interesting to support. Here, it is supported via arguments to the `run` and `determinize` commands of `nfdeterminize`.

6 Multithreaded Approach

6.1 Towards a Multithreaded Approach

Sequential implementations of NFA determinization and minimization provide excellent results in general for decently-sized automatas. However, as the size of the given automatas increases, single threaded performance does not suffice, hence the need to extend the determinization and minimization process to multiple threads, or multiple machines.

The main challenge of rewriting determinization and minimization comes from the amount of shared memory used during superset construction and Hopcroft's algorithm. While superset construction can be relatively well separated between threads and processes, it is harder to do so using Hopcroft's algorithm, as it is optimised for sequential use.

6.2 New Algorithms

6.2.1 Determinization

To extend the superset construction algorithm for multithreaded usage and avoid as much use of shared memory, the algorithm is modified as such, for some number k of threads:

- Divide the frontier of research into k frontiers, such that each thread has access to its own queue,
- Divide the state to number map to k parts, accessed by each thread. Each thread hashes the set of states it constructed to determine which hash map the set of states should go to,
- The main thread keeps track of which thread's frontier is empty, and sends a signal to each thread to kill itself once all frontiers for each thread is empty, which means all the graph has been explored.

This approach mainly uses mutexes for access to shared memory instead of inter-thread messaging. Messaging-based approach will instead be used later.

The reason the superset construction algorithm works well for a multithreaded approach is because most of the computation going on during superset construction happens when constructing a new state from the frontier, which is a step that does not require shared memory. Furthermore, shared memory issues are inhibited by the sharding of the map and the frontier. Thus, superset construction greatly profits from multithreading and multiprocessing.

Algorithm 3 Multithreaded Superset Construction Algorithm

```
1: procedure SupersetConstruction(M = (S, Σ, δ, S_0, T))
2:     M ←[(S_0, 0)]
3:     T' ←[]
4:     if ∃s ∈ S_0.s ∈ T then
5:         T' ←[S_0]
6:     end if
7:     F ←[S_0]
8:     while F ≠ ∅ do
9:         S_next ←pop from F
10:        for all a ∈ Σ do
11:            S' ←
12:            for all s ∈ S_next do
13:                Add s and all ϵtransitions from sto S'
14:            end for
15:            if S_next ∉ M then
16:                M ←[M, (S', |M|)]
17:                if ∃s ∈ T.s ∈ S' then
18:                    T' ←[T', S']
19:                end if
20:                F ←[F, S']
21:            end if
22:            δ' ←[δ', (S_next, a, S')]
23:        end for
24:    end while
25: end procedure
```

6.2.2 Minimization

While NFA determinization is relatively easy to extend to multithreaded variants, NFA minimization is much harder to parallelize. Hopcroft's minimization algorithm is still the currently best minimization algorithm in terms of time complexity, as explained in 5.2.1. However, Hopcroft's algorithm is not efficiently parallelizable. Therefore an alternative approach is needed to implement multithreaded minimization.

The DFA minimization parallelization started being approached around the 1990s.

Here, a parrallelizable minimization algorithm called *partition refinement* is used, first implemented by [**?**], with the exception that a RAM-based implementation is used in `nfdeterminize`'s case.

7 Multiprocess/Distributed Approach

7.1 Shared Memory Determinization

7.2 Shared Memory Parallel Minimization

8 Benchmarking

By context of the research, it is natural that most of the test cases used to gauge performance of the system are token passing networks. To do so, a set of benchmarks is used to compare the performance of the multiple system configurations on a set of sce

On all cases, benchmarks are run on a 12-threads Intel i5-11400 machine, with 32GB of RAM. All iterations take 25 run samples, with 3 seconds of warmup between iterations. Such methods avoids skewing the results down because of continuous CPU operation.

Results are written to a CSV file, which is then used to generate comparison tables based on performance.

8.1 GAP-generated NFAs

First of all, automatons generated by GAP are used to test determinization and minimization. GAP [**?**] is a system for computational discrete algebra, which provides a programming language and a couple of libraries, two of which being `Automata` and `PatternClass`. The `PatternClass` library provides methods to generate multiple kinds of token passing networks, such as the buffer and stack TPN, and some functions to generally convert graphs into NFAs.

The main property of GAP-generated NFAs is that they generate states in the NFA out of the nodes of the graph and not out of data structures, which ends up building a lot of ϵ transitions which, in the end, will get removed during determinization. The main advantage of using GAP generated NFAs is to stress test how well determinization handles ϵ transitions when finding new states.

In regards to benchmarking, buffer-and-stack NFAs are generated using GAP - from buffer size 2 to 3, and stack size 2 to 7.

8.2 Self-generated NFAs

On top of the NFAs generated by GAP, the program is also able to generate its own NFAs out of token passing network patterns. While GAP has a general algorithm for converting TPNs into NFAs, which leads to NFAs with lots of extra information in form of ϵ transitions, self-generated NFAs are optimised for the patterns they're built for. This means that the leading

NFA has less ϵ transitions but still describes the same language. Therefore it is preferred to generate NFAs this way when researching the language of permutations described by a TPN.

In regards to benchmarking, and to keep benchmark speeds fast enough, buffer-and-stack NFAs and two-stack NFAs are used. In research, buffer-and-stack TPNs are generally studied as simplifications of two-stack TPNs. In practice, both kinds of TPNs are used to stress test different parts of the system.

- 3-buffer-and-k-stack TPNs tend to stress test the determinization process more. For quick benchmarking, buffer-and-stack TPNs of buffer size 2 to 3, and stack sizes 2 to 7 are used to compare the speeds of different implementations.
- 3-stack-and-k-stack TPNs tend to stress minimization more as, by observation, they are usually poorly minimizable. two-stack TPNS of first stack size 2 to 3, and second stack size 3 to 5 are used.

Finally, to measure the speed of each implementation, a measure of k for the biggest 3-buffer-k-stack TPN that can be determinized and determinized in under a minute.

9 Results

9.1 Comparison of Test Cases

Figure 9.1 shows the evolution of automaton sizes for self-generated buffer and stack TPNs against GAP-generated TPNs. Meanwhile, figure 9.1 A general observation that can be derived, is that self-generated TPNs are always initially smaller, and take considerably less time to determinize than GAP-generated TPNs. It can also be observed that the GAP-generated automatas grow in size a lot quicker than self-generated ones. Furthermore, while the time taken to determinize and minimize self-generated automatas barely evolves, with an evolution that is within error margin, the time taken for GAP-generated automatas increases significantly over time.

It is possible to establish now that there are only advantages to using the self-generated NFAs instead of the GAP-generated ones when it comes to researching about token passing networks. However, research of allowed permutations in sequential stack TPNs is often further simplified to searching about stack and buffer TPNs. The reasoning behind it is that the produced NFAs are usually less complex than two-stack ones.

`Nfdeterminize` supports the generation of both TPN types. The question is therefore - are buffer and stack TPNs less complex than two-stack TPNs, and if so, what are the difference between both that lead to such results.

The first difference between two-stack and buffer-and-stack lies in how they are generated. Let's assume that `nfdeterminize`'s NFA generator is used here. The main difference between buffer-and-stack and two-stack is how during automata generation, on each iteration, at most 1 ϵ transition from the first stack to the 2nd stack is made, while at most k ϵ transitions are made from the buffer to the stack for the buffer and stack automaton.

Table 1: Comparison between self-generated TPNs and GAP-generated TPNs

b	s	Initial self-gen	Initial GAP-gen	Inter-mediary self-gen	Inter-mediary GAP-gen	Final self-gen	Final GAP-gen
2	2	18	82	3	3	3	3
2	3	38	244	8	8	8	8
2	4	78	730	16	16	16	16
2	5	158	2188	32	32	32	32
3	2	39	460	4	4	4	4
3	3	117	1826	13	13	13	13
3	4	351	7288	33	33	33	33
3	5	1053	29134	85	85	85	85

Table 2: Comparison of time taken on self-generated automats vs GAP-generated ones

b	s	self-generated deter-minize	GAP-generated deter-minize	self-generated minimize	GAP-generated minimize
2	2	5.23	21.5	2.28	2.31
2	3	23.67	116.45	8.14	8.55
2	4	78.21	644.7	20.27	20.07
2	5	284.96	3278.8	47.04	47.5
3	2	16.0	158.57	3.91	3.83
3	3	107.15	1527.34	16.66	17.0
3	4	648.39	13325.41	54.03	54.54
3	5	3320.03	108088.1	184.0	188.63

Table 3: Automaton size comparison between buffer-and-stack NFAs and Two-stack NFAs

s1	s2	Initial bns	Initial two-stack	Inter-mediary bns	Inter-mediary two-stack	Final bns	Final two-stack
2	2	18	18	3	3	3	3
2	3	38	38	8	8	8	8
2	4	78	78	16	16	16	16
2	5	158	158	32	32	32	32
2	6	318	318	64	64	64	64
2	7	638	638	128	128	128	128
3	2	39	36	4	10	4	10
3	3	117	96	13	35	13	35
3	4	351	253	33	177	33	177
3	5	1053	664	85	1070	85	1070
3	6	3159	1740	221	6682	221	6682
3	7	9477	4557	577	41888	577	41888

Two major remarks can be done from figure 9.1. First, the size of both initial automatas is very similar, and evolves similarly. On buffer/first stack size = 2, they're even equal. This similarity is an example of how closely related those 2 TPN patterns are.

Then, from observations of the intermediary DFA sizes for bns and two-stack TPNs on $s1 = 3$, it can be seen how much quicker the size of the two-stack TPN increases compared to the size of the buffer-and-stack TPN, despite how the initial size of the buffer-and-stack TPN was bigger than that of the two-stack TPN. This evolution may be caused due to how more states in the buffer-and-stack NFA may share the same state in the intermediary DFA, due to the larger number of ϵ transitions.

9.2 Multithreading Against Sequential

9.3 Multiprocessing Approach and Scaling

10 Conclusion

11 Appendix

11.1 Software Tests

The main purpose of NFA determinization and minimization in the context of this paper is the description of the language accepted by transportation graphs, as described in 11.2. Therefore, testing here is crucial for experimental correctness. Here, a presentation of the various unit tests done for the software is shown to prove the correctness of NFA

Table 4: Comparison time taken on sequential against multithreaded mode

b	s	Sequential determinize	Multithreaded determinize	Sequential minimize	Multithreaded minimize
2	2	5.23	3725.09	2.28	2.54
2	3	23.67	4086.26	8.14	8.78
2	4	78.21	3506.48	20.27	20.35
2	5	284.96	3704.92	47.04	47.63
2	6	882.84	3848.89	113.11	114.83
2	7	2704.02	5228.54	271.61	279.62
3	2	16.0	3296.99	3.91	3.9
3	3	107.15	3465.07	16.66	17.42
3	4	648.39	4018.34	54.03	53.54
3	5	3320.03	6020.85	184.0	191.4
3	6	18929.55	12607.03	590.02	614.89
3	7	117353.27	43373.57	1787.97	1849.16

determinization and minimization implementations.

11.1.1 Approach

During development, the approach to development of the multiple determinization and minimization algorithms is done in a Test Driven Approach:

- The unit tests for determinization and minimization were written first, as a way to test for correct algorithm behaviour,
- Each iteration of the minimization and determinization algorithms are tested upon those sets of tests, and considered sound if they pass the test suite.

This development approach allowed to both guarantee that implementations were correct, and allow some level of incremental development by continuously writing new algorithms and testing them against the test suite.

11.1.2 Determinization

Determinizations of NFAs are DFAs that often show certain patterns. For example, a determinization of an NFA often possesses a "sinkhole" state for which all transitions coming from it come back to the state. Other behaviours should be clearly defined, such as how determinization deals with ϵ transitions. Therefore, unit tests check that the behaviours that define determinization are strictly followed, hence proving the correctness of the algorithm.

The figures below list the multiple patterns that were tested during determinization testing.

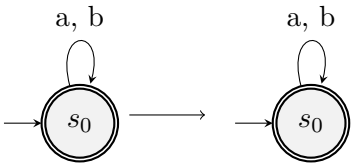

Figure 1: Redundant NFA to redundant DFA

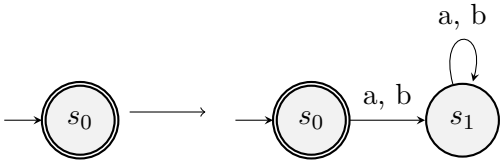

Figure 2: Empty language NFA to empty language DFA

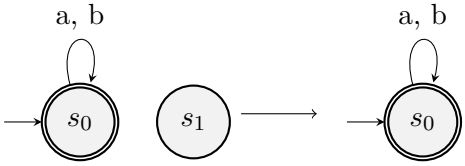

Figure 3: Unreachable state in NFA removed in the DFA

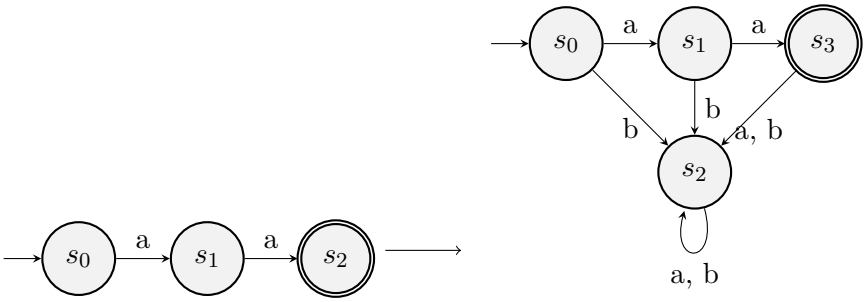

Figure 4: NFA to DFA with sinkhole

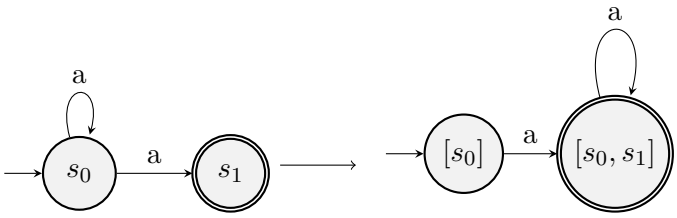

Figure 5: NFA to DFA with sets of NFA states for states

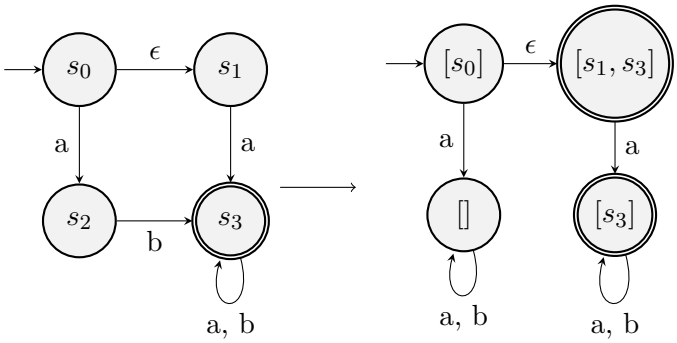

Figure 6: ϵ automaton to DFA

11.1.3 Minimization

Minimization is tested similarly to the way determinization is tested, by testing on DFAs that are minimially bipartite, some with a single separation of sets within a partition, and some unminimizable DFA.

Those tests are not as detailed behaviour-wise as the unit tests for determinization, and are more specialized towards Hopcroft's algorithm. However, they do demonstrate some level of correctness in the algorithm.

The figures below demonstrate the patterns and cases that were tested to show the implementation correctness of the written minimization algorithms.

11.2 Token Passing Networks

A token passing network is a directed graph $G = (V, E)$ such that:

- V: Vertices/nodes,
- $E \in (V \times label \times V)$: edges - an edge connects a vertex to another, and may contain a label.
- There exists a single input node I in V such that there is not ingoing edges to it -

$$!\exists I \in V.\nexists v_2.\exists v_1.\exists e = (v_1, v_2) \in E.v_2 = I$$

- There exists a single output node O in V such that there is no ougoing edges from it -

$$!\exists O \in V.\nexists v_1.\exists v_2.\exists e = (v_1, v_2) \in E.v_1 = O$$

Token Passing Networks, originally called Transportation Graphs by [**?**], were originally studied by [**?**] in order to think about what kind of packet permutations might arise from packet delay in networks.

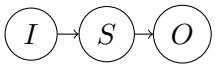

Figure 7: Example of a stack TPN

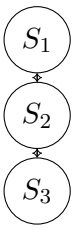

Figure 8: Inner Workings of a size 3 TPN stack

Design patterns in transportation graphs can introduce properties for a transition graph, as well. For example, figure 11.2 shows the design of an infinite stack data structure, where S represents an infinite number of nodes connected as shows figure 11.2.

Transportation graphs are used as such:

- Each node can store one "token",
- "Tokens" can be fetched from the input node I to the next node,
- "Tokens" must be transported to the output node O,
- After all "tokens" from the input stream are consumed, there should be no tokens remaining in the graph.

Tokens are kept in track by keeping the order at which the tokens arrived in. Therefore, it is possible to study the possible order at which the tokens arrive at with a given transportation graph.

11.2.1 Permutation Classes and 3-1-2 avoidance

As described previously, it is possible to describe what possible orders the tokens may arrive at from a token passing network. Such area of study comes from a property of stacks, which is what kinds of permutations are Stack Sortable.

For example, [**?**] describes properties of stacks in regards to what permutations they accept.

Figure 11.2.1 shows a graphical representation of a stack, with an input stream on the right, containing a stream of tokens, and an output stream on the left, which accepts tokens. On one hand, figure 11.2.1 describes a permutation which is accepted by a stack.

On the other hand, 11.2.1 presents a permutation which is is not accepted by a stack. As the figure shows, it is possible to pass the 3rd token to the output first, but then it is

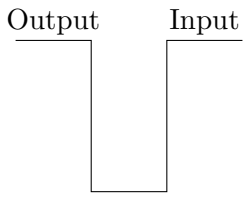

Figure 9: Graphical Example of a stack

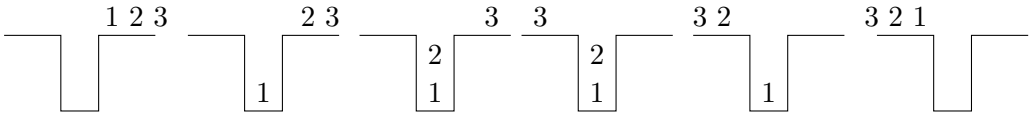

Figure 10: Successful Stack Permutation

impossible to pass the first token, as token 2 is at the front. This class of pattern is called 3-1-2 avoidance/ 2-3-1 avoidance.

The 3-1-2 pattern or 2-3-1 pattern depends on whether the stack tries reorder a sequence of tokens (2-3-1 exclusion), or it tries to permute an ordered sequence. It is therefore said that 2-3-1 is the inverse pattern of 3-1-2.

Thus, the stack model can be modelled with transportation graphs using a stack of nodes, hence the study of accepted permutations for a transportation graph.

11.2.2 Conversion into NFA

A property of token passing networks, is that they can be converted into NFAs, in which the alphabet represents the rank encoding of a token, and a state is represented by the order of a token on the initial ordered input stream.

The following definition from [**?**] defines a *rank encoding* -

> The *rank encoding* of a permutation is generated by replacing each element by its value relative to those elements which come after it.

From the rank encoding it is easy to describe the language of all accepted permutations of a transportation graph, hence the wish to convert transportation graphs into NFAs, and to determinize and minimize them.

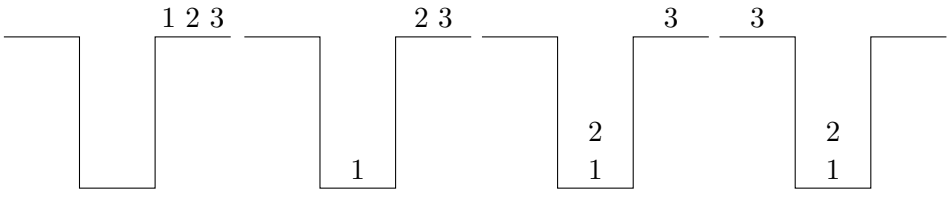

Figure 11: 3-1-2 Avoidance in a Stack

11.3 User Manual

References

[1] J. E. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," 1971.

[2] B. Ravikumar and X. Xiong, "A parallel algorithm for minimization of finite automata," in *Proceedings of International Conference on Parallel Processing*, pp. 187–191, 1996.

[3] V. Slavici, D. Kunkle, G. Cooperman, and S. A. Linton, "Finding the minimal DFA of very large finite state automata with an application to token passing networks," *CoRR*, vol. abs/1103.5736, 2011.

[4] V. Slavici, D. Kunkle, G. Cooperman, and S. Linton, "An efficient programming model for memory-intensive recursive algorithms using parallel disks," in *International Symposium on Symbolic and Algebraic Computation*, 2012.

[5] C. Ba and A. Gueye, "On the distributed determinization of large nfas," *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, pp. 1–6, 2020.

[6] C. A., "A comparative study of large automata distributed processing," *2022 IEEE 16th International Conference on Application of Information and Communication Technologies (AICT)*, pp. 1–6, 2022.

[7] J. Berstel, L. Boasson, O. Carton, and I. Fagnot, "Minimization of automata," 2010.

[8] X. Yingjie, "Describing an n log n algorithm for minimizing states in deterministic finite automaton," 2009.

[9] E. F. Moore, *Gedanken-Experiments on Sequential Machines*, pp. 129–154. Princeton: Princeton University Press, 1956.

[10] D. Revuz, "Minimisation of acyclic deterministic automata in linear time," *Theoretical Computer Science*, vol. 92, no. 1, pp. 181–189, 1992.

[11] S. Linton, "Gap: Groups, algorithms, programming," *ACM Commun. Comput. Algebra*, vol. 41, p. 108–109, sep 2007.

[12] M. Atkinson, M. Livesey, and D. Tulley, "Permutations generated by token passing in graphs," *Theoretical Computer Science*, vol. 178, no. 1, pp. 103–118, 1997.

[13] S. Waton, "On permutation classes defined by token passing networks, gridding matrices and pictures: Three flavours of involvement," 2007.