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

A. Guerville

March 21, 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.

Here, a set of determinization and minimization algorithms, ranging from single-threaded to distributed, is described and implemented to solve that task, and tested on a problem class about the determinization of NFAs: Transportation Graphs AKA Token Passing Networks.

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.

Parallel NFA determinization algorithms have begun being researched round the 1990s. For example, [1] ran a parallel NFA determinization and minimization algorithm on a supercomputer, using a message passing model instead of shared memory.

In 2007, [2] implements a disk-based distributed algorithm for large NFAs. A disk-based approach avoids the RAM memory space issues from previous implementations.

Later, [3] proposes a general programming model to migrate RAM-based legacy algorithms into parallel disks - and applies the model to NFA determinization and minimization.

In 2020, [4] uses Bulk Synchronous Parallel abstract computer model to implement a more performant distributed NFA determinization and minization algorithm.

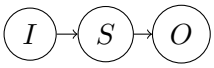

Figure 1: Example of a stack TPN

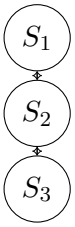

Figure 2: Inner Workings of a size 3 TPN stack

Finally, [5] compares both the MapReduce and BSP-based NFA determinization and minimization algorithm, finding that the BSP/Pregel based solution outperforms the MapReduce solution.

3 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 [6], were originally studied by [6] in order to think about what kind of packet permutations might arise from packet delay in networks.

Design patterns in transportation graphs can introduce properties for a transition graph, as well. For example, figure 3 shows the design of an infinite stack data structure, where S represents an infinite number of nodes connected as shows figure 3.

Transportation graphs are used as such:

- Each node can store one “token”,

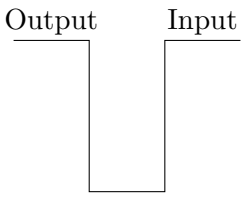

Figure 3: Graphical Example of a stack

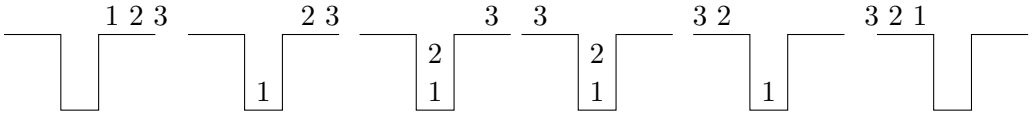

Figure 4: Successful Stack Permutation

- "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.

3.1 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, [7] describes properties of stacks in regards to what permutations they accept.

Figure 3.1 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 3.1 describes a permutation which is accepted by a stack.

On the other hand, 3.1 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 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.

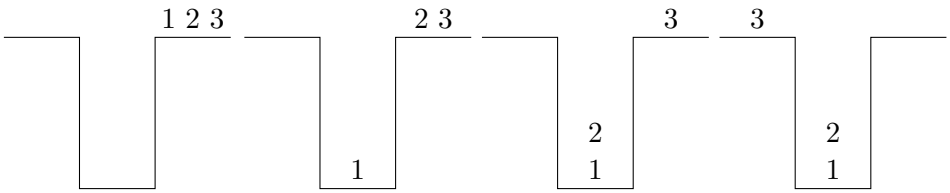

Figure 5: 3-1-2 Avoidance in a Stack

3.2 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 [7] 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.

4 Testing & Benchmarking

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 3. Therefore, testing and benchmarking here involves two main components: testing the correctness of the program's behaviour, and being able to determinize and minimize token passing networks fast.

4.1 Behaviour Testing

The first step in the testing and benchmarking process is to test for correctness of the programs' behaviours. Here, it is done by unit testing the determinization and minimization algorithm by using NFA and DFA examples that rely on particular behaviours of each of the determinization and minimization algorithms.

4.1.1 Testing 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.

Part 7.1 of the appendix lists the multiple patterns that were tested during determinization testing.

4.1.2 Testing 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.

4.2 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.

4.2.1 GAP-generated NFAs

First of all, automatons generated by GAP are used to test determinization and minimization. GAP [8] is as 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.

4.2.2 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.

On all cases, benchmarks are run on a 8-core 16-thread Intel machine.

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 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.

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 $|S(S)| = 2^{|S|}$, 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.

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
```

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, [2] 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.

5.2 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[9], 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 $V \backslash \delta^{-1}(P, a)$

Hopcroft's Algorithm, as shown on figure 5.2.1, has asymptotic time complexity $O(knlog(n))$[9], where:

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

Implementation-wise, the approach here is closer to the implementation described in [10], 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 $\mathcal{P}$.

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
```

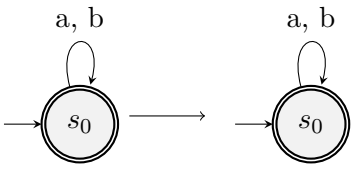

Figure 6: Redundant NFA to redundant DFA

5.2.2 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 [11]. 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`.

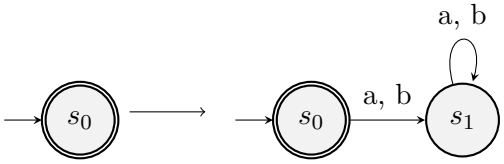

Figure 7: Empty language NFA to empty language DFA

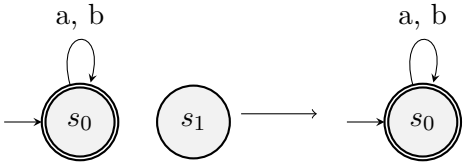

Figure 8: Unreachable state in NFA removed in the DFA

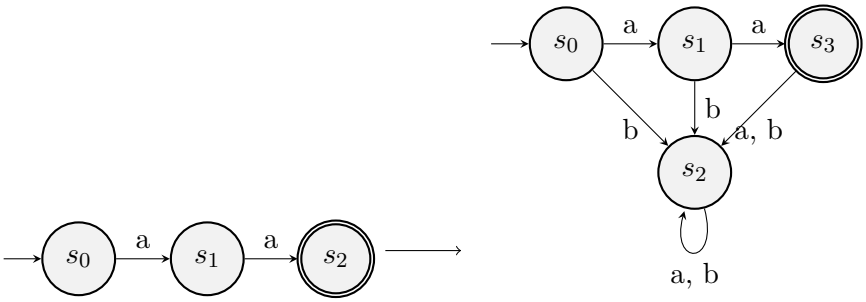

Figure 9: NFA to DFA with sinkhole

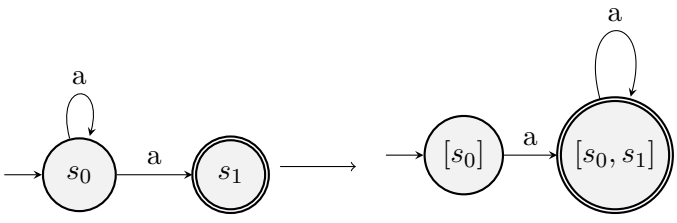

Figure 10: NFA to DFA with sets of NFA states for states

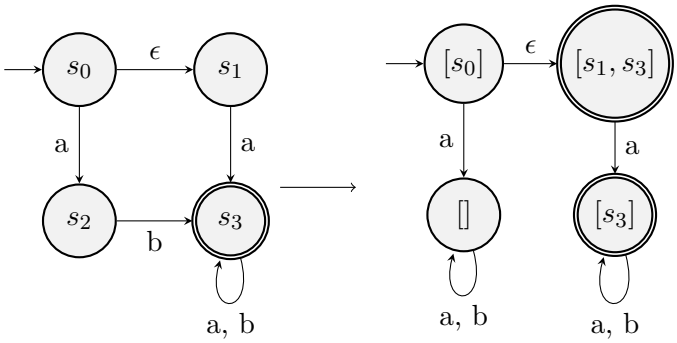

Figure 11: ϵ automaton to DFA

5.3 Benchmarking

5.3.1 GAP-generation vs self-generation

5.3.2 Pitfalls

6 Multithreaded Approach

6.1 Towards a Multithreaded Approach

6.2 New Algorithms

6.2.1 Determinization

6.2.2 Minimization

6.3 Benchmarking

7 Appendix

7.1 NFA to DFA patterns in Unit Tests

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

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