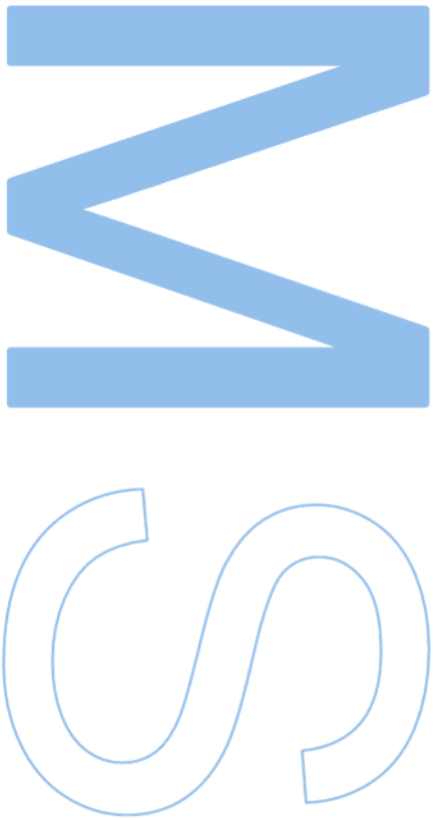
Metaheuristics for NFA Size Reduction



Hugo Matos de Rangel Tavares
Master in Networks and Information Systems Engineering
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Metaheuristics for NFA Size Reduction

Hugo Matos de Rangel Tavares
Master in Networks and Information Systems Engineering
Department of Computer Science
2025

Supervisor
João Pedro Pedroso, Associate Professor
Faculty of Sciences of the University of Porto

Co-supervisor
Rogério Reis, Assistant Professor
Faculty of Sciences of the University of Porto

Universidade do Porto

Masters Thesis

Metaheuristics for NFA Size Reduction

Author:
Hugo Matos de Rangel Tavares

Supervisor:
João Pedro Pedroso

Co-supervisor:
Rogério Reis

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Networks and Computer Systems Engineering

at the

Faculty of Sciences of the University of Porto
Computer Science

September 30, 2025

Acknowledgements

Em primeiro lugar, quero expressar a minha mais profunda gratidão aos meus pais, Maria do Carmo e Paulo Tavares. Foram eles que me acompanharam em todas as etapas da minha vida, transmitindo-me uma educação sólida e, sobretudo, valores como o respeito pelo próximo, a honestidade, a humildade, a compaixão e a perseverança. Esses valores moldaram o meu caráter e guiaram as minhas escolhas pessoais e académicas.

Agradeço também pelo amor incondicional e pelo apoio constante, presentes tanto nos momentos de alegria como nas fases mais difíceis. Sem a sua dedicação, paciência e confiança, não teria conseguido chegar até aqui.

Quero ainda agradecer a toda a minha família que, tal como os meus pais, sempre me apoiou em qualquer situação e me aceitou plenamente, mesmo nos momentos em que me encontrava mais em baixo.

Desejo também deixar uma dedicatória especial ao meu avô Lucindo Rangel Tavares, que foi uma das pessoas que mais impactou a minha vida e ajudou a formar o meu caráter. Tanto ele como o meu pai foram os responsáveis por me ensinar a ser ambicioso na vida, mostrando-me que não preciso de subir por cima de ninguém, mas que tenho a capacidade de criar as minhas próprias oportunidades.

Quero igualmente agradecer ao Nuno Fernandes, por sempre me ter motivado e estimulado desde jovem a cultivar a curiosidade, e por ter dedicado o seu tempo a ajudar-me a escolher o curso que iria seguir na faculdade. A sua amizade sensata e o papel de mentor que desempenhou foram fundamentais durante este período académico.

Quero agradecer à minha namorada, Flávia, por todo o apoio que me deu na nossa nova vida em Barcelona, por todas as refeições que preparou para que eu pudesse dedicar-me ao estudo, pelos sorrisos que me ofereceu nos momentos de tristeza, pelos abraços que me confortaram quando mais precisei e pela amizade e felicidade que me transmite todos os dias, esteja ela perto ou longe.

Quero também agradecer aos meus amigos, com quem partilhei inúmeras ocasiões marcantes da vida académica no Porto — desde as longas noites de estudo até às conversas intermináveis que tornaram os dias mais leves, passando pelas celebrações das vitórias alcançadas e pelo apoio nas fases mais exigentes. Um especial agradecimento vai para os meus amigos dos LáLáPito e da Praxe, que para sempre levarei no meu coração.

Por fim, quero agradecer aos meus orientadores de tese, ao professor Rogério e ao professor João, que ao longo de vários anos me ajudaram, apoiaram e ensinaram, sem nunca desistirem de mim. Apesar de estar a viver noutro país, mostraram-se sempre disponíveis, e a sua orientação foi essencial para a concretização deste trabalho.

Abstract

The minimisation of non-deterministic finite automata (NFA) is a computationally intractable problem: finding the smallest equivalent NFA is PSPACE-complete. Classical approaches, such as state bisimulation, achieve limited reductions. This thesis investigates the use of metaheuristic algorithms to achieve higher state reductions than state bisimulation achieves, while preserving the recognised language.

We propose a framework that combines state merging, state removal, and state expansion rules with higher-level search strategies. Three metaheuristics are explored: a greedy First Improve Descent, an Iterative Merge–Expand Search that alternates between reduction and state splitting, and a Tabu Search that leverages short-term memory to escape local minima. These methods are implemented using the FAdo library and evaluated on randomly generated NFAs derived from regular expressions of varying sizes.

Experimental results show that our metaheuristics consistently outperform classical reduction methods in terms of achieved state reduction, especially on NFAs that recognise large and complex languages. While the greedy search offers efficiency, the metaheuristic approaches facilitate deeper exploration of the solution space, enabling smaller automata at the cost of increased runtime. A practical case study—reducing the automaton of a Wi-Fi QR code validator—demonstrates the applicability of the approach in real-world scenarios.

This work highlights the potential of metaheuristic search for tackling otherwise intractable NFA minimisation. It lays the groundwork for future research into adaptive, hybrid, and domain-specific heuristics for automata size reduction.

Keywords: Automata theory; Metaheuristics; Non-deterministic finite automata size reduction

Resumo

A minimização de autómatos finitos não determinísticos (NFA) é um problema computacionalmente intratável: encontrar o menor NFA equivalente é um problema PSPACE-completo. As abordagens clássicas, como a bisimulação de estados, alcançam normalmente reduções limitadas. Esta dissertação investiga o uso de algoritmos metaheurísticos para obter reduções de estados superiores às alcançáveis através da bisimulação, preservando sempre a linguagem reconhecida

Propomos uma framework que combina regras de colapso de estados, remoção de estados e expansão de estados com estratégias de pesquisa de nível superior. São exploradas três metaheurísticas: First Improve Descent (uma procura ávida), Iterative Merge–Expand Search (que alterna entre redução e divisão de estados) e Tabu Search (que recorre à manutenção de uma memória de curto prazo para escapar de mínimos locais). Estes métodos foram implementados com recurso à biblioteca FAdo e avaliados em NFAs gerados aleatoriamente a partir de expressões regulares de diferentes dimensões.

Os resultados experimentais demonstram que as metaheurísticas propostas superam consistentemente os métodos clássicos de redução em termos de número de estados obtidos. Enquanto a pesquisa ávida apresenta maior eficiência, as metaheuristicas apresentadas permitem uma exploração mais profunda do espaço de soluções, permitindo alcançar melhores soluções, ainda que com maior custo computacional. Um estudo de caso prático—um validador de códigos QR para Wi-Fi—ilustra a aplicabilidade da abordagem em cenários reais.

Este trabalho evidencia o potencial da utilização de metaheurísticas para enfrentar o problema da minimização de NFAs e estabelece bases para investigação futura em heurísticas adaptativas, híbridas e específicas ao domínio da redução do tamanho de autómatos.

Palavras-chave: Teoria de Automatos; Metaheurísticas; Redução do tamanho de autómatos finitos não-determinísticos

Table of Contents

List of Figures

1. Introduction

Finite automata are one of the fundamental models of computation in computer science. They provide a rigorous and compact way to represent and reason about regular languages, which are found in diverse application domains, including lexical analysis, formal verification, communication protocols, and text processing. Despite their theoretical simplicity, automata often become very large in practice, especially when constructed from regular expressions or used as intermediate representations in tools such as compilers and model checkers.

The size of an automaton has a direct impact on its efficiency: the number of states influences memory requirements, runtime performance, and the feasibility of applying further algorithms that operate on the automaton. Therefore, **state reduction** is a critical task. While minimisation of deterministic finite automata (DFA) is well studied and solvable in polynomial time (e.g., Hopcroft's algorithm), the minimisation of non-deterministic finite automata (NFA) is a far more challenging problem. Indeed, finding the smallest equivalent NFA is **PSPACE-complete**, which makes exact minimisation impractical for large instances.

To overcome these limitations, this work investigates the application of **metaheuristic algorithms** for NFA state reduction. Metaheuristics are high-level strategies, such as local search, tabu search, and iterative improvement, that guide underlying heuristics in exploring large and complex solution spaces. By combining reduction rules with exploration techniques such as state splitting, metaheuristics offer a promising approach to reducing the number of states of NFAs while avoiding getting stuck in local minima.

The main contributions of this work are:

1. The development of state reduction strategies that combine *state merging*, *state removal* and *state expansion* to enlarge the search solution space of possible equivalent NFAs.
2. The design and implementation of metaheuristic algorithms—such as First Improve Descent, Iterative Merge-Expand Search, and Tabu Search—for systematic exploration of the reduction problem.

3. An experimental evaluation of the proposed methods, comparing them with existing approaches and demonstrating their advantages in reducing large automata.
4. A practical case study using a Wi-Fi QR code validator to illustrate the applicability and efficiency of the proposed techniques.

The remainder of this thesis is organised as follows:

- Chapter 2 introduces the theoretical background on regular languages, finite automata, and classical state reduction techniques.
- Chapter 3 discusses local search and tabu search as heuristic frameworks.
- Chapter 4 presents state reduction techniques tailored to large NFAs.
- Chapter 5 details the proposed metaheuristics.
- Chapter 6 reports on the experimental results.
- Chapter 7 illustrates a real-world application and impact of the proposed metaheuristics.
- Chapter 8 concludes the thesis and outlines directions for future work.

2. Regular Languages

In formal language theory, an alphabet is a finite set of symbols and a word is a finite sequence of symbols from that alphabet. A language over an alphabet Σ is a subset of $\Sigma^\star$, where $\Sigma^\star$ denotes the set of all finite words over Σ, including the empty word ε [1]. Regular languages are closed under three fundamental operations:

- Union: the union of two languages, $L_1 \cup L_2$, results in a new language that contains all the words in L_1 or L_2.
- Concatenation: the concatenation of two languages, $L_1 \cdot L_2$, results in a language that contains all the words formed by concatenating a word in L_1 with a word in L_2.
- Kleene star closure: the Kleene star closure of a language L, denoted by $L^\star$, is the language formed by concatenating zero or more words from L.

Regular languages are the subset of formal languages that finite automata can recognise.

A finite automaton is a mathematical model used to recognise languages; therefore, it is widely used in computer science in areas such as data processing (regular languages are used in pattern-matching tools), compilers (lexical analysis relies on finite automata for tokenisation), networks (automata models are used to verify communication protocols), among others.

A finite automaton is defined as a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$ [1], where:

- Q is the finite set of states of the automaton.
- Σ is a finite set of symbols recognised by the automaton, called the alphabet.
- δ is the transition function that defines the behaviour of the automaton.
- $Q_0 \subseteq Q$ is the set of initial states of the automaton.
- $F \subseteq Q$ is the set of accepting (or final) states of the automaton.

A finite automaton is deterministic if the set of initial states is a singleton and each state has exactly one transition for each symbol of the alphabet; otherwise, it is non-deterministic.

- For a deterministic finite automaton (DFA), δ is defined as $\delta : Q \times \Sigma \to Q$
- For a non-deterministic finite automaton (NFA), δ is defined as $\delta : Q \times \Sigma \to 2^Q$

Each state in a non-deterministic finite automaton can have multiple outgoing transitions for the same input symbol. Therefore, we can extend the transition function of the NFA to $\delta : 2^Q \times \Sigma \to 2^Q$, meaning that a subset of the NFA's states can transition to another subset of the NFA's states over a symbol of the alphabet.

Furthermore, this transition function can be extended to handle a whole word by defining $\delta : 2^Q \times \Sigma^\star \to 2^Q$. This extended function computes the set of all states that the NFA can reach after processing a word in $\Sigma^\star$, starting from a set of states. This allows us to compute the behaviour of the NFA over longer inputs, not just individual symbols.

The extended transition function is essential to determine whether an automaton recognises a given language. A word $w \in \Sigma^\star$ is recognised by an automaton if, starting from the set of initial states $Q_0 \subseteq Q$, the extended transition function $\delta(Q_0, w)$ leads to a subset of states that includes at least one accepting state in F.

Hence, w is recognised by the automaton if:

$$\delta(Q_0, w) \cap F \neq \emptyset.$$

There exists at least one path in the automaton that terminates in an accepting state after processing the word w. The extended transition function systematically determines all states reachable by the automaton after processing a given input word. This function is essential for determining whether the word is accepted as part of the language recognised by the non-deterministic finite automaton (NFA).

Now let L be a language over an alphabet Σ. The Myhill-Nerode equivalence relation $\sim_L$ over $\Sigma^\star$ relates two words x and y when every continuation z has the same effect on membership in L:

$$x \sim_L y \iff \forall z \in \Sigma^\star,\ xz \in L \Leftrightarrow yz \in L.$$

The Myhill-Nerode theorem [1, 2] states that L is regular if and only if $\sim_L$ has a finite number of equivalence classes. These equivalence classes correspond to the states of the minimal DFA recognising L.

2.1. Languages of a state

Each state of an automaton can be associated with a set of languages:

- Left language of a state: the set of words that can take the automaton from one of its initial states to state q.

$$L(q) = \{w \in \Sigma^\star \mid q \in \delta^\star(Q_0, w)\}$$

- Right language of a state: the set of words that can take the automaton from state q to at least one accepting state.

$$R(q) = \{w \in \Sigma^\star \mid \delta^\star(\{q\}, w) \cap F \neq \emptyset\}$$

- Inner language of a state: the set of words that can take the automaton from state q back to q.

$$I(q) = \{w \in \Sigma^\star \mid q \in \delta^\star(\{q\}, w)\}$$

Figure 2.1 provides a visual representation of these three languages for a generic state q.

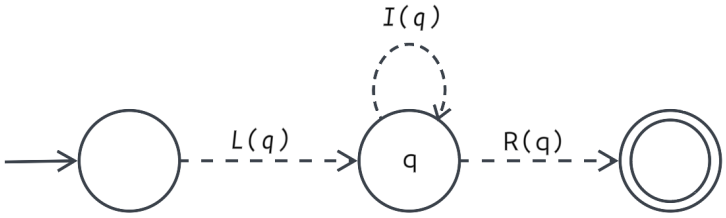


Figure 2.1: Representation of the languages of a state

Based on these definitions of state languages, any word formed by concatenating a word from the left language of a state with a word from its right language belongs to the language of the automaton, L_A.

$$\forall q \in Q,\ \forall w \in L(q),\ \forall u \in R(q),\ w{\cdot}u \in L_A$$

Similarly, if the automaton reaches q, performs a loop labelled by a word in $I(q)$, and then reaches an accepting state, the resulting word is also accepted:

$$\forall q \in Q,\ \forall w \in L(q),\ \forall v \in I(q),\ \forall u \in R(q),\ w{\cdot}v{\cdot}u \in L_A$$

The words in $I(q)$ can be repeated, so the inner language can be closed under the Kleene star operation while preserving acceptance:

$$\forall q \in Q,\ \forall w \in L(q),\ \forall v \in I(q),\ \forall u \in R(q),\ w{\cdot}v^{\star}{\cdot}u \in L_A$$

Therefore, the language associated with paths that pass through state q is included in the language accepted by the automaton:

$$L(q) \cdot I(q)^{\star} \cdot R(q) \subseteq L_A$$

More generally, the accepted language can be decomposed as the union of the left and right languages of all states:

$$\bigcup_{q \in Q} L(q) \cdot R(q) = L_A.$$

Since the inner language captures loops on q, the left and right languages are closed under concatenation with such loops:

$$\forall q \in Q,\ L(q) \cdot I(q) \subseteq L(q)$$

$$\forall q \in Q,\ I(q) \cdot R(q) \subseteq R(q)$$

By closing the inner language under Kleene's closure:

$$\forall q \in Q,\ \ L(q) \cdot I(q)^{\star} \subseteq L(q)\ ,\ I(q)^{\star} \cdot R(q) \subseteq R(q)$$

Thus, for each state q, the contribution of the inner language can be absorbed into the left or right language:

$$L(q) \cdot I(q)^{\star} \cdot R(q) \subseteq L(q) \cdot R(q) \subseteq L_A.$$

2.2. Automata state reduction

2.2.1 Hopcroft's algorithm

Hopcroft's algorithm [1, 3] is a *DFA* minimisation algorithm that produces a language-equivalent DFA with the minimum number of states. The idea is to group states into equivalence classes, where two states are considered equivalent if they cannot be distinguished by any string (i.e. starting from either state leads to the same state for the same inputs). Instead of comparing states in pairs, the algorithm works by partitioning the set of all states and refining those partitions until no further splits are possible.

The algorithm begins by dividing the set of states into two groups: accepting states and non-accepting states. Then, it repeatedly checks whether any group can be split into smaller groups based on how states transition under input symbols. If some states in a group behave differently from others (e.g., they transition to different partitions on a given symbol), that group is split. This refinement continues until no more splitting can occur, at which point each group represents one state in the minimised DFA. Hopcroft's algorithm can be implemented with a time complexity of $\mathcal{O}(n \cdot log(n))$ [3].

2.2.2 Moore's algorithm

Another approach for DFA state minimisation is Moore's algorithm [1, 4].

Moore's algorithm is another method for minimising DFAs, but it works in a different way from Hopcroft's algorithm. Instead of splitting partitions like Hopcroft's, Moore's method is based on successive refinement of equivalence classes by distinguishing states step by step.

The process starts by partitioning states into two sets: accepting and non-accepting. Then, in each iteration, states within the same partition are further distinguished based on the behaviour of their transitions. For example, if two states belong to the same group but, on some input symbol, one is placed into a different group than the other, then they are separated into new groups. This process repeats until the partition no longer changes between iterations (a fixed point). At that point, each group of states represents one state in the minimised DFA.

In terms of efficiency, Moore's algorithm is simpler conceptually, but is less efficient than Hopcroft's; it can be implemented in $\mathcal{O}(n^2)$ [5] in the worst case.

According to Moore's algorithm, two states are equivalent if and only if they exhibit identical transitional behaviour. If, for any symbol, a pair of states in the same partition transitions to different states, these states are distinguishable and must fall in different equivalence classes.

2.2.3 Bisimulation

Bisimulation [6] is a concept used to compare the behaviour of two state systems. In automata theory, this concept is used to compare pairs of states by checking whether their transitions can mutually match each other. Unlike Hopcroft's and Moore's algorithms, which apply to *DFAs* and guarantee the unique minimal DFA, bisimulation-based reduction can be applied to both DFAs and NFAs [7]; however, for NFAs it is a reduction technique and does not guarantee a globally minimal equivalent NFA.

We say that two states are bisimilar when they are related by a bisimulation relation. Intuitively, for every transition from state s under some input a to a state s', there must be a corresponding transition from state t under the same input a to some state t', with s' and t' again related. The same condition must also hold in the opposite direction.

Definition (bisimulation): Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. A binary relation $R \subseteq Q \times Q$ is a bisimulation if, for every pair $(s,t) \in R$, the following conditions hold:

$$s \in F \Leftrightarrow t \in F$$

$$\forall a \in \Sigma,\ \forall s' \in \delta(s,a),\ \exists t' \in \delta(t,a) \text{ such that } (s',t') \in R$$

$$\forall a \in \Sigma,\ \forall t' \in \delta(t,a),\ \exists s' \in \delta(s,a) \text{ such that } (s',t') \in R.$$

From this, we can conclude that if two states are in a relation of bisimilarity on their right language, then they share the same capacity to reach an accepting state, and thus they are mergeable. This conclusion is reinforced by the Myhill-Nerode theorem: if two states have the same right language, they are equivalent and consequently mergeable.

2.3. Complexity of NFA state reduction

We know that the NFA minimisation problem is highly challenging. The problem of finding the smallest NFA that recognises the same language as a given NFA is known to be **PSPACE-complete** [8], meaning that the problem is very complex and computationally challenging, thus, particularly unusable for large NFAs.

PSPACE is a class of computational complexity problems that can be solved by a deterministic Turing machine using a polynomial amount of memory, regardless of the time taken. The term 'PSPACE-complete' refers to the set of the most complex problems within PSPACE.

Because exact NFA minimisation is PSPACE-complete, practical tools often use polynomial-time reduction techniques such as simulation or bisimulation. These techniques compare pairs of states and merge states that cannot be distinguished by the chosen behavioural relation. They are effective in practice, but they should be understood as sufficient reduction criteria rather than complete algorithms for finding the smallest equivalent NFA.

Given the current assumptions on the complexity of the minimisation problem, the problem is intractable [9]; the best we can hope for—and will attempt—is to reduce the number of states in the automaton.

3. Local search and tabu search

Given the computational complexity of finding the minimal NFA equivalent to a given one, we use heuristic-based methods in our study. This chapter discusses local search techniques and advances to more sophisticated metaheuristics, particularly tabu search.

Local search is a direct yet effective approach to reduce NFA states, but its inherent limitations often require more advanced strategies, such as tabu search. By incorporating memory and carefully managed non-greedy exploration, tabu search enhances both the robustness and efficiency of the reduction process, helping to find smaller NFAs that are still language-equivalent to the initial one.

3.1. Heuristics and metaheuristics

Heuristics are problem-solving techniques designed to find good, if not optimal, solutions within a reasonable time. Unlike exhaustive methods that guarantee an optimal solution, which are often computationally expensive, heuristic approaches provide a balance between solution quality and efficiency. Their performance relies heavily on domain knowledge, which helps guide the search toward promising areas of the solution space.

In the context of NFA minimisation, heuristics often involve merging states with equivalent behaviours or removing states that contribute redundantly to language recognition. However, these methods are inherently greedy, as they perform immediate best moves without considering future consequences. As such, they are prone to getting trapped in local optima, from which single transformations can make no further progress.

Metaheuristics are higher-level procedures that orchestrate the use of heuristics to enhance performance further. These strategies enable the search to have a more global exploration of the search space, incorporating memory, randomness, and adaptive behaviour. Metaheuristics aim to avoid premature convergence to poor solutions and are especially effective in complex optimisation domains like NFA state reduction.

3.2. Local search

Local search is one of the most fundamental metaheuristics. It iteratively explores the neighbourhood of the current solution, accepting transitions that improve the objective function. In our case, the objective is to minimise the number of states in an NFA while maintaining its language equivalence.

Formally, given a solution $s \in S$ in the solution space S, the neighbourhood of s, denoted $N(s)$, is defined as the set of solutions that can be obtained by applying a predefined set of transformations to s. The choice of neighbourhood structure N determines both the efficiency of the search and the quality of the solutions found, since it determines which alternative solutions are available at each step of the search.

Definition (local optimum): A solution x^* is a local optimum if there is not a better solution (i.e., with a smaller objective) in its neighbourhood $N(x^*)$:

$$\forall x \in N(x^*),\ f(x) \geq f(x^*)$$

Here, f is the objective function, which for our problem is the total number of states. The neighbourhood $N(x^*)$ consists of all automata that can be reached from x through a valid transformation.

An effective local search relies on defining meaningful neighbourhoods. For the problem of NFA state reduction, we define the neighbourhoods of a state by applying each of the following operations:

- State merging: States with equivalent left or right languages can be merged.
- State removal: Redundant states identified via language inclusion or union can be safely removed.
- State splitting: A state is divided into two or more states with fewer transitions, increasing short-term complexity but potentially enabling future merges.

Splitting is particularly valuable when stuck in a local optimum. By introducing a new structure into the automaton, we open up new paths for reduction.

The local search algorithm proceeds as follows:

↪ Begin with an initial reduced NFA using basic reduction rules.

↪ While there exists a possible reduction:

  ↪ Search for a neighbour in $x' \in N(x)$ such that $f(x') < f(x)$.

  ↪ Update $x \leftarrow x'$ with the first improving neighbour.

This greedy approach is straightforward but limited because it terminates at the first local minimum.

3.3. Tabu search

To overcome the limitations of greedy local search, we employ tabu search [10], a metaheuristic designed to explore a solution space by temporarily allowing non-improving moves and maintaining memory structures.

Tabu search addresses the main limitation of local search, getting stuck in local optima. It allows transitions to solutions that may initially worsen the objective function, in the hope that these moves help escape local valleys.

After a split, we mark the new sibling states as a short-tenure tabu block, preventing immediate merges/removals that would undo the change. This path-awareness is crucial to avoid cycles where the search revisits the exact solutions repeatedly. It also encourages exploring less obvious moves that may initially seem worse but might lead to better minima after further transformations, thereby diversifying and exploring new regions of the solution space.

In our implementation, tabu search maintains a list of recently applied transformations to disable immediate reversals. This approach enables the algorithm to escape local minima by allowing non-improving moves (with certain restrictions), effectively extending the exploration of the solution space.

We implement tabu search as follows:

- ↪ We begin by reducing an NFA as much as possible by applying sequentially merging and removal rules.
- ↪ When no further reductions are possible, apply state splitting transformations to ”open” states, randomly in our implementation.
- ↪ Record sibling states in a tabu list to avoid undoing beneficial splits prematurely.
- ↪ Continue alternating between reduction (merging/removal) and exploration (splitting) phases until no further improvement is found or a stopping criterion is met.

By using tabu search in our search, we aim to systematically explore the solution space, overcoming the limitations of purely greedy methods and finding smaller NFAs that recognise the same language.

4. State reduction in large NFAs

In this chapter, we propose functions to identify mergeable or removable states in non-deterministic finite automata (NFA).

Also, we will explain the reasoning behind our implementations and correlate them with the study in the previous chapters. The functions mentioned can be found in the *Appendix*.

4.1. State Merging by Language Equivalence

To reduce the number of states in an automaton, we employ two processes: state merging and state removal. It is essential to note that to remove or merge a state, the affected states must meet certain conditions to ensure that the language is preserved.

To find mergeable states in an automaton, we evaluate the languages of the states and look for equalities in the transitions of the states, as determined by the transition function of the NFA. Here, we define that a group of states can be merged into a single state if and only if any of the following reduction rules apply.

Reduction rule 1: A group of states can be merged if their left language is equivalent.

A group of states $\{q_1, q_2, \ldots, q_n\}$ can be merged into a single state q' if:

$$L(q_1) \equiv L(q_2) \equiv \cdots \equiv L(q_n)$$

As illustrated in Figure 4.1, merging states based on the equivalence of their left languages preserves the recognised language of the automaton.

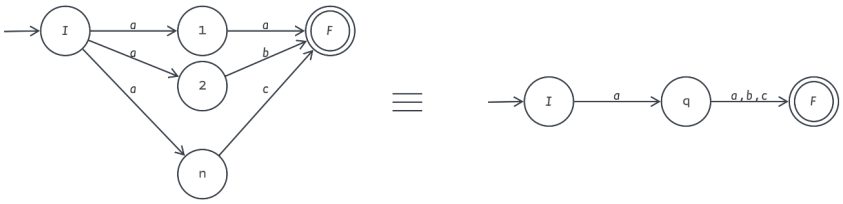


Figure 4.1: Example of merging two states by equivalence on their left language.

Reduction rule 2: A group of states can be merged if their right language are equivalent.

A group of states $\{q_1, q_2, \ldots, q_n\}$ can be merged into a single state q' if:

$$R(q_1) \equiv R(q_2) \equiv \cdots \equiv R(q_n)$$

Figure 4.2 shows an example of merging states based on the equivalence of their right languages.

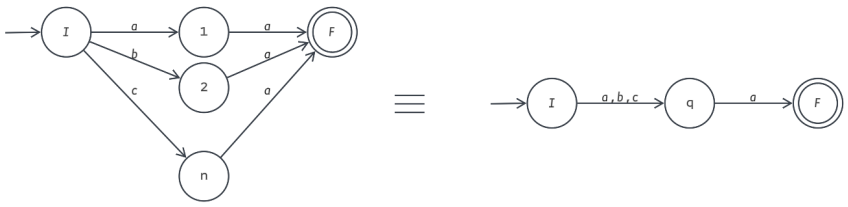


Figure 4.2: Example of merging two states by equivalence on their right language.

In Function 1 we present the function $\mathrm{GetMergibleStatesByPartitions}$. This function identifies mergeable states in an NFA by analysing the states' transitions and categorising them into partition sets based on their left and right languages.

Initially, the method initialises the LeftPartition dictionary (map), which will contain the sets of states (values of the map) partitioned by their left language (keys of the map), and the RightPartition set, which includes the sets of states (values of the map) partitioned by their right language (keys of the map).

Then, the algorithm enters a loop that iterates over each state of the NFA. For each state, the algorithm searches its set of transitions (both incoming and outgoing) to form the tags of the state's languages.

As shown in Section 2.1, $L(q) \cdot I(q)^\star \subseteq L(q)$ and $I(q)^\star \cdot R(q) \subseteq R(q)$. With this, when there is a loop in a state, we create an ambiguous, extra, tag that will enable us to identify mergeable states with loops, for example, if two states $s1$ and $s2$ have only two transitions in an automaton A, like, $[(s1, a) = s1; (s2, a) = s2]$, or $[(s1, a) = s1; (s2, a) = s1]$, in both cases, the states $s1$ and $s2$ have the same left language, in both cases they should be mergeable if they are both in the same condition, both final or neither final.

Once the tag is generated for the state's left language, the state is placed in a set in the $LeftPartition$ where the key to this set is the tag of its left language, and, if it applies, also in a set with its loop tag as key, then the same process is done for the state in the $RightPartition$.

Finally, once the loop ends and all the states have been assigned a partition, we extract from the partitions dictionaries a list of sets of states that is the union of the values of both partitions that contain at least two elements. This algorithm returns two values, the

first is the $LeftPartition$ and the second is the $RightPartition$, both reduced to sets with mergeable states, either by equivalence on their left or right languages.

We refer to the resulting automaton obtained by iteratively applying Function 1, until no further merging is possible, as a *Local Minimum* within the space of possible equivalent NFAs.

4.2. State Removal by Language Inclusion

As a way to further reduce the number of states of the NFA, we also evaluate yet another reduction rule.

Reduction rule 3: A group of states can be removed if there is another state that both its left and right languages contain, respectively, the left and right languages of the states to be removed.

A group of states $\{q_1, q_2, \ldots, q_n\}$ can be removed if there exists another state q' such that:

$$\bigcup_{i=1}^{n} L(q_i) \subseteq L(q') \wedge \bigcup_{i=1}^{n} R(q_i) \subseteq R(q')$$

As shown in Figure 4.3, states can be removed when both their left and right languages are included in those of another state.

Figure 4.3: Example of removing two states due to language inclusion. The languages of 1 contain the languages of 2 and n.

The reasoning behind the **reduction rule 3** is based on the concept of state language inclusion in an NFA. If two states $s1$ and $s2$ in the automaton have identical transition behaviours and can be considered functionally equivalent, then they can be merged. Specifically, the rule checks whether the inset and the outset of these states are subsets of each other. If the outset of $s1$ is a subset of the outset of $s2$, and the inset of $s1$ is a subset of the inset of $s2$, then $s1$ can be considered redundant, hence removed.

With this third reduction rule, when applied to all combinations of the set of states of an automaton, we could reduce the automaton even further. However, just like the subset

construction algorithm, the algorithm to evaluate the third reduction rule on all combinations of the set of states of an automaton would be exponential, $\mathcal{O}(2^n)$.

So, in our implementation, Function 2 $\mathrm{RemoveStateByLanguageInclusion}$, we only apply the reduction rule 3 for each possible pair of states, evaluating, at most, $n^2 - n$ pairs of states each iteration, hence setting the temporal complexity of this search at $\mathcal{O}(n^2)$.

The functions 3 and 4, $\mathrm{CompareOutset\_isS1SubsetOfS2}$ and $\mathrm{CompareInset\_isS1SubsetOfS2}$, are used to validate the transitions of the states. Function 3 is used to compare the right language of the states by validating that the transitions that start from state $s1$ are a subset of the transitions starting from state $s2$. And Function 4 is used for the other way around, to compare the left language of the states by validating that the transitions that end in state $s1$ are a subset of the transitions ending in state $s2$.

With these verifications, we ensure that the left language of $s2$ contains the left language of $s1$ and the right language of $s2$ includes the right language of $s1$, so the language of the automaton is preserved if we remove $s1$.

4.3. State Removal by Language Union

With **reduction rule 3**, we may remove a state if its left and right languages can be described as sublanguages of another state's left and right languages, respectively. As stated before, the reasoning supporting **reduction rule 3** is that a state is redundant if, for every path in the NFA from an initial state to this state, there is also a path in the automaton for the same words from an initial state to another state; and if for every word that constitutes a path in the automaton from the state to remove to an accepting state also constitutes a path in the automaton from another state to an accepting state.

Now, for the following reduction rule, we define that a state can be removed (while the automaton's language is preserved) if its left or right language can be represented as the union of the respective languages of two or more states. However, to preserve the language accepted by the automaton, the remaining states must inherit the opposite language of the removed state (i.e. if we can remove a state s_1 by the left-language union of a group of states S, then every state in S must inherit the right-language of s_1, and vice-versa).

If the state to remove is final and the removal was due to the inclusion of the right language, then at least one of the remaining states must also be final. If the removal was due to the inclusion of the left language, then all of the remaining states must also be final. If the removed state is not final, none of the remaining states can be final. This is because they either inherit the left or right language of the removed state, and in either case, the finality of the remaining states is irrelevant.

Reduction rule 4: A state can be removed if either its left or right languages are equivalent to the union of, respectively, the left or right languages of a group of states.

A state q' can be removed if there exists a group of states $\{q_1, q_2, \ldots, q_n\}$ such that:

$$\bigcup_{i=1}^{n} L(q_i) = L(q') \vee \bigcup_{i=1}^{n} R(q_i) = R(q')$$

Figure 4.4 illustrates the removal of a state whose language can be expressed as the union of the languages of other states.

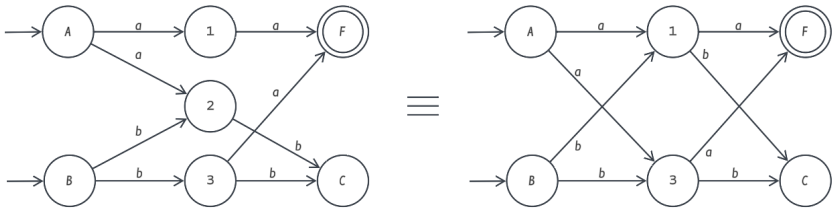


Figure 4.4: Example of removing a state due to language union. The union of left languages of 1 and 3 is equal to the left language of 2, therefore 2 can be removed if both states 1 and 3 inherit the right language of 2.

Again, for the **reduction rule 4** to have a valid result (i.e. preserve the automaton's language), the states that enabled the removal of the state (in our example in figure 4.4, the states 1 and 3) have to inherit the languages of the removed state (state 2 in our example). Since, in our example, we were able to remove the state 2 based on the equality on the left language with the union of the left languages of the states 1 and 3, these states must inherit the right language of the state 2, adding the following transitions to the NFA's transition function: $\delta(1,'b') = C$ and $\delta(3,'a') = F$.

Just like the implementation of **reduction rule 3**, we could consider larger sets of states to evaluate the union of their languages and compare it to other states' languages. Still, here we faced the same challenge as in **reduction rule 3**, checking all combinations of states for such a language-union inclusion is computationally infeasible, growing on the order of 2^n for an automaton with n states.

To implement the **reduction rule 4**, here, we present the Function 5, *RemoveByState-LanguageUnion*, which can remove a state by right language union if we pass the nfa in the arguments, or remove a state by left language union if we pass the reverse nfa (*nfaR*). The function, calls *find_union_transitions* with the NFA's $delta$ and gets a list of sets where each set will represent a language (meaning that all elements of the set will have the same language, either the right language or the left language, depending on the argument passed to the function), like $\{s_0, (s_1, s_2)\}$. Then, we filter the sets that contain both singletons and tuples. After filtering the list, it selects a random element to remove, copying the opposite language (it copies the left language if we trigger the search for right language inclusion, or copies the right language if we trigger the search for left language inclusion) of the state s_0 into s_1 and s_2, then removing the state s_0 from the NFA.

The Function 6, *FindUnionTransitions*, retrieves a list of elements like $(s_0, (s_1, s_2))$, where s_0 is a state of the NFA that can be removed due to the language union of the states

s_1 and s_2. Depending on the transition function passed as an argument to the function *find_union_transitions*, if we pass the $delta$ of the NFA as argument, we get a list of elements for state removal by right language union; if we pass the $deltaReverse$ of the NFA, we get a list for state removal by left language union.

The *find_union_transitions* function first produces a transition map, which we denote as $TransitionMap$, that is built by evaluating the transition function of the NFA. For each state of the NFA, we first calculate the language of the state (the language depends on the argument passed to the function) and map it into an ordered list of transitions, then we add a new entry in the $TransitionMap$ with the state index as key and the transitions list as the value. After this first evaluation, for each state index lower than the state under assessment, we join the second state's language (retrieved from the $TransitionMap$) with the state under evaluation, we sort the transition list and add a new entry in $TransitionMap$ where the key is a tuple of both states evaluated and the value is the joint transitions list. After iterating over all states to collect the removable states by language union from the $TransitionMap$, we revert the $TransitionMap$, leaving the transition lists in the keys and a list of states and tuples in the values. Then we can collect a list of elements to remove by state language inclusion by getting all the values where there is, at least, a state and a tuple in the list, like $[s_0, (s_1, s_2)]$, if we end up with no values like $[s_0, (s_1, s_2)]$, then it's because there are no removable states by language inclusion.

The complexity of implementing Function 5, *RemoveByStateLanguageUnion*, is limited by the complexity of Function 6, *FindUnionTransitions*, because after calling this function, Function 5 will randomly select one of the candidates retrieved by it, if any. In our implementation, building the map of $TransitionMap$, we only go through the set of states in an NFA once; we limit the complexity of this operation at $\mathcal{O}(n)$. Since we need to invert the keys and values of the $TransitionMap$ (to group states and tuples by their language), this process will have, at most, $n + n^2$ steps (one per state and one per each pair of states). Therefore, we can assert that the overall complexity of Function 5 is bounded by $\mathcal{O}(n^2)$.

4.4. Automata state expansion

With the reduction rules presented earlier, through state bisimulation on their languages, we can reach some local minima of the solution space, which is all the possible automata that can recognise the same language.

These results represent automata with no mergeable states; however, we hypothesise that, by applying some transformations to these automata, we will be able to leave a local valley and reach different local minima, possibly even better than the ones already discovered.

To escape local minima like the ones we reached in the previous section, here, we introduce some transformations to the automaton to change the transition function of the automaton and, hopefully, find more pairs of states that satisfy any of the **Reduction Rules**, hence, reaching a different local minimum. We introduce a process for automata state splitting, which applies transformations to the automata while preserving the language it recognises.

State splitting is a technique widely used to transform NFAs into equivalent DFAs by splitting nondeterministic states into multiple deterministic states, each state representing a specific path the NFA could take from the original state on a given input. By systematically applying this technique, we can gradually eliminate the nondeterminism in an NFA.

While state splitting increases the number of states, it can also be a valuable tool for subsequent state reduction. We apply state splitting as a strategy to create new states that may be mergeable with existing states.

Splitting rule 1: We can split a state if its right language can be expressed as a union between two or more sublanguages.

A state q can be split into a group of states S if:

$$\forall s \in S,\ L(q) \equiv L(s) \wedge R(q) \equiv \cup R(s)$$

Figure 4.5: Example 1 of expanding a state by right language union

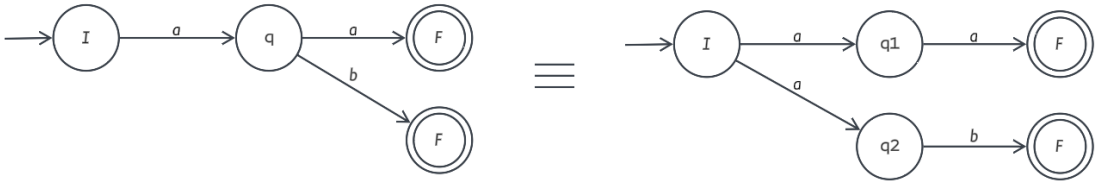


Figure 4.6: Example 2 of expanding a state by right language union

Splitting rule 2: We can split a state if its left language can be expressed as the union between two or more sublanguages.

A state q can be split into a group of states S if:

$$\forall s \in S,\ R(q) \equiv R(s) \wedge L(q) \equiv \cup L(s)$$

Figure 4.7: Expanding a state by states left language union

The process of state expansion through language union is illustrated in Figures 4.5, 4.6, and 4.7.

Through these mutations, we replace a state by a group of states that recognise simpler languages (as the original state languages contain the union of the descendant states' languages). So, we propose that it is possible that, after these operations, the new states are mergeable with other states that were not before the splitting.

The Function 7 ($\mathrm{OpenRandomCandidateState}$) is our implementation for randomly selecting and expanding a state in an NFA by splitting its language, exclusively on either its left or right language.

First, the Function 7 gets the best candidates to expand on their right language from Function 8 and the best candidates to expand on their left language from Function 9. In our implementation, we define the ”best candidates” as states with multiple *in-transitions* and multiple *out-transitions*, and preferably, non-final and non-initial states. However, this last premise is ignored if we don’t have other candidates.

The Functions 8 and 9, that filter the states of the NFA that have potential for beneficial expansion of their right language. By removing states without outgoing transitions and those that are isolated in their partitions, the algorithm returns a refined set of states that, when expanded, could lead to further state merging or other optimisations in the NFA.

Then, one candidate from the list of states is randomly selected to be expanded. To expand a selected state s_1 that was a candidate to open its right language, we replace the s_1 by as many states as right transitions the selected state had; if s_1 had three out-going transitions, then, s_1 will be replaced by three states, each state will have a copy of all the in-transitions and one out-transition of s_1. If the selected state was a candidate to open its left language, then the same process is done, but in the opposite direction.

The algorithms to open a state by its left or right language are, respectively, Function 11 and Function 10.

Suppose the state-splitting didn’t enable subsequent state reduction. In that case, we will be able to apply one of the reduction rules presented in Section 2.2 and merge them back to the original state. Since the process of selection and splitting can be done at worst, once per each iteration, the temporal complexity of this mutation is set to $\mathcal{O}(1)$.

5. Metaheuristics for NFA state reduction

In this chapter, we extend the work presented in Chapter 4 by introducing some metaheuristic algorithms to reduce the size of NFAs through state reduction. While Chapter 4 detailed individual functions for merging, splitting, and evaluating state equivalence, this chapter leverages these functions at a higher level.

The proposed metaheuristic approach combines both state merging and selective state expansion strategies. By alternating between merging and controlled state ”opening” operations that explore alternative automata configurations, the algorithms aim to minimise the number of states without changing the language recognised by the NFA. This dynamic balance allows the search to escape local minima and explore a broader range of candidate solutions.

These programs were written in Python 3, chosen for its simplicity, expressiveness, and extensive library support.

A library crucial to the development of this work is *FAdo* (version 2.2.0). During the development, we extensively used *FAdo* to create, manipulate, and analyse finite automata.

Some other libraries were used during the development of this application, including *matplotlib* for data presentation, *random* to introduce aleatory elements in the experimental analysis, *graphviz* to visualise the generated automata, and *datetime* for performance analysis.

The full implementation, including scripts to reproduce the experiments, is available at the following GitHub repository: https://github.com/hmrt/Meta-heuristics-for-NFA-minimization.

5.1. First Improve Descent

The Function 12, FirstImproveDescent, is designed to aggressively minimise an NFA by applying two merging techniques in succession until no further reduction is possible.

In each iteration, the function first calls MergeByPartitionRefinement. Once there are no more states mergeable by partition refinement, the function resorts to language inclusion verification by invoking RemoveStateByLanguageInclusion. The program then returns to MergeByPartitionRefinement and repeats the process until neither function reduces the automaton.

The search terminates once both functions $\mathrm{MergeByPartitionRefinement}$ and $\mathrm{RemoveStateByLanguageInclusion}$ results in no changes in the NFA, meaning that it isn't possible to reduce the NFA further.

5.2. Iterative Merge-Expand Search

The Function 13, $\mathrm{IterativeMergeExpandSearch}$, optimises an NFA by reducing its number of states through state merging and state removal and then expands states when it reaches local minima, when there are no more mergeable states. This exchange between merging and splitting states enables us to search through different automaton configurations, which we explore to find a smaller equivalent NFA.

For this, Function 13 takes as argument the NFA subject to the search (nfa), the number of opening iterations (n_opens) and the number of openings that will be executed in each opening iteration (n_per_iter).

With the input arguments and some initial setup, we enter a loop where we first try to merge states (with $\mathrm{MergeByPartitionRefinement}$) and remove states (with $\mathrm{RemoveStateByLanguageInclusion}$), just like in $\mathrm{FirstImproveDescent}$. Only when we have no more mergeable or removable states (having reached a local minimum) do we initiate the state expansion phase. We then execute n_per_iter openings with $\mathrm{OpenRandomCandidateState}$ and proceed to another round of searching for mergeable and removable states, followed by another iteration of state splitting. This process is done n_opens times. After n_opens iterations, the search goes once again through the merge or removal processes, and it ends when there are no more possible reductions.

5.3. Tabu Search

We employed the tabu technique on top of the searches from Sections 5.1–5.2 with a short–term memory that prevents immediate reversals of state splitting. The method repeatedly applies the same reductions, but passes the tabu list as an argument to avoid reducing the states currently in the tabu list. When no reduction is possible, it opens (splits) a candidate state to create new merging opportunities, then marks the newly created siblings as *tabu* for a finite amount of time, to which we will refer as $tabu\_k$, or simply k. We retain the best solution found so far and fall back to it when the search stalls.

Input An NFA N, a stopping criterion (which for us will be a finite amount of time, denoted with $\mathrm{run\_time}$), and the tabu tenure of a block/group of sibling states, $\mathrm{tabu\_k}$ (number of transformations that a newly created state will stay in the tabu list).

Transformations In each iteration, we first try to reduce the number of states of the nfa, in order: $\mathrm{mergeByPartitionRefinement\_withTabu}$, $\mathrm{removeStateByLanguageInclusion}$, $\mathrm{removeByStateRightLanguageUnion\_withTabu}$, $\mathrm{removeByStateLanguageUnion\_withTabu}$.

If none applies, we *open* a random candidate state ($\mathrm{openRandomCandidateState}$), and push the set of newly created sibling states into the tabu list as a tabu block. The tabu list operates as a FIFO queue that keeps the tabu blocks for $\mathrm{tabu\_k}$ iterations. If the search times out, we reset to the best solution found, clear the tabu list, and retrigger the search. When there is no other move, instead of opening a state, the search stops.

Correctness. All transformations preserve the language recognised by the automaton, as argued in Chapter 4. The state splitting replaces one state by a set of siblings whose languages form a disjoint union equal to the original state language (Section 4.4); thus, the resulting automaton will recognise the same language as the initial one, differing only in its structure, which can enable further merges.

Complexity. Let I be the number of iterations and n_{open} the number of openings actually performed ($n_{\mathsf{open}} \leq \mathrm{run\_time}$-bounded). If C_{merge}, C_{incl}, C_{union}, C_{open} denote the costs of one pass of the corresponding operators and $C_{\mathsf{open}} = O(|Q| + |\delta|)$ is the rewiring cost of a split, then:

$$\mathcal{O}\big(I \cdot (C_{\mathsf{merge}} \cdot C_{\mathsf{incl}} \cdot C_{\mathsf{union}}) \ \cdot \ n_{\mathsf{opens}} \cdot C_{\mathsf{open}}\big)$$

Taking that I and n_{opens} are finite numbers, we may simplify the complexity of these to the order of $\mathcal{O}(1)$. So we can simplify the formula to:

$$\mathcal{O}\big(C_{\mathsf{merge}} \cdot C_{\mathsf{incl}} \cdot C_{\mathsf{union}} \cdot C_{\mathsf{open}}\big)$$

From this, since C_{open} is almost direct and doesn't iterate over a list (an element from a list is randomly chosen), we may also consider it as $\mathcal{O}(1)$. Moreover, the complexity of the C_{merge} is $\mathcal{O}(n)$, so we can simplify even further the formula:

$$\mathcal{O}(n \cdot C_{\mathsf{incl}} \cdot C_{\mathsf{union}})$$

The function $\mathrm{removeStateByLanguageInclusion}$ is based on comparing the languages of all pairs of states, meaning that it compares each transition of each pair of states (although it never compares the same pair of states twice), so this function can be implemented with a time complexity of $\mathcal{O}(n \cdot log(n))$.

$$\mathcal{O}\big(n \cdot (n \cdot log(n)) \cdot C_{\mathsf{union}}\big)$$

In the Section 4.3 we showed that C_{union} is bounded by $\mathcal{O}(n^2)$.

$$\mathcal{O}\Big(n \cdot (n \cdot log(n)) \cdot n^2\Big) \equiv \mathcal{O}\Big(n^4 \cdot log(n)\Big)$$

Which leads us to a final and generic complexity of $\mathcal{O}(n^4)$.

6. Results

In this chapter, we present the results obtained from testing the implementation delivered in this study for the minimisation of nondeterministic finite automata. We explore the effectiveness of our methods in reducing the number of states while ensuring that the language of the automaton is preserved. By applying splitting techniques to local minima, our approach searches for better local minima compared to traditional state reduction algorithms.

We first analyse the performance of the proposed methods on a range of benchmark NFAs, evaluating the reduction in state complexity and computational efficiency. The results are compared against classical NFA minimisation techniques, highlighting the advantages and limitations of our approach. Additionally, we discuss how different parameter choices and heuristics influence the final minimised automaton.

The remainder of this chapter is structured as follows: Section 6.1 provides a quantitative evaluation of state reduction using bisimulation and the proposed reduction functions. Section 6.2 presents the greedy First Improve Descent search. Sections 6.3 and 6.4 discuss the merge-expand and tabu-search variants, and Section 6.5 summarises the overall results.

To test and benchmark the performance of the metaheuristics presented in Chapter 5, we generated random regular expressions with $FAdo$ using different sizes and then generated the NFAs in Table 6.1 for each regex with Antimirov's partial derivative algorithm for automata generation:

NFA	Number of states	Size of the alphabet
A	503	1
B	509	2
C	1036	2
D	1093	10
E	2466	10
F	2619	26
G	4292	26

Table 6.1: Benchmark NFAs used in the experimental evaluation.

All experiments were conducted on a MacBook Pro equipped with an Intel Core i9-9880H CPU @ 2.30 GHz (8 physical cores, 16 logical threads), 32 GB of RAM, running macOS 15.6.1. The implementation was developed in Python 3.10 with the FAdo 2.2.0 library and supporting packages.

6.1. Initial results

6.1.1 Right language auto-bisimulation

To compare our approach to the current state of the art in NFA state reduction, we first present the results of applying the FAdo library's method for state bisimulation on the right language of states, named autobisimulation2().

The *FAdo.FA.autobisimulation2* is a method that identifies pairs of states in an NFA that have an equivalent right language, meaning that they have the same ability to reach final states. By resorting to bisimulation, we infer that two states simulate each other for all inputs.

The function works by initially marking state pairs as distinguishable if only one of them is a final state, since such pairs clearly differ in language acceptance. Then, iteratively marks additional state pairs as distinguishable based on their transitions: if two states have different sets of input symbols, or if following the same input leads to transitions into already distinguishable states, they too are marked. This marking stage continues until no new pairs are marked in an iteration.

In the end, the function returns all unmarked pairs — these represent state pairs that have not been proven to behave differently and are therefore considered mergeable.

For this, we created the search $test\_FAdo\_autobisimulation2$ which iteratively runs the method *FAdo.FA.autobisimulation2* and merges a random set of mergeable states returned by the method until no more mergeable sets of states are found.

The reduction performance and execution time obtained using this approach are presented in Figure 6.1 and Table 6.2.

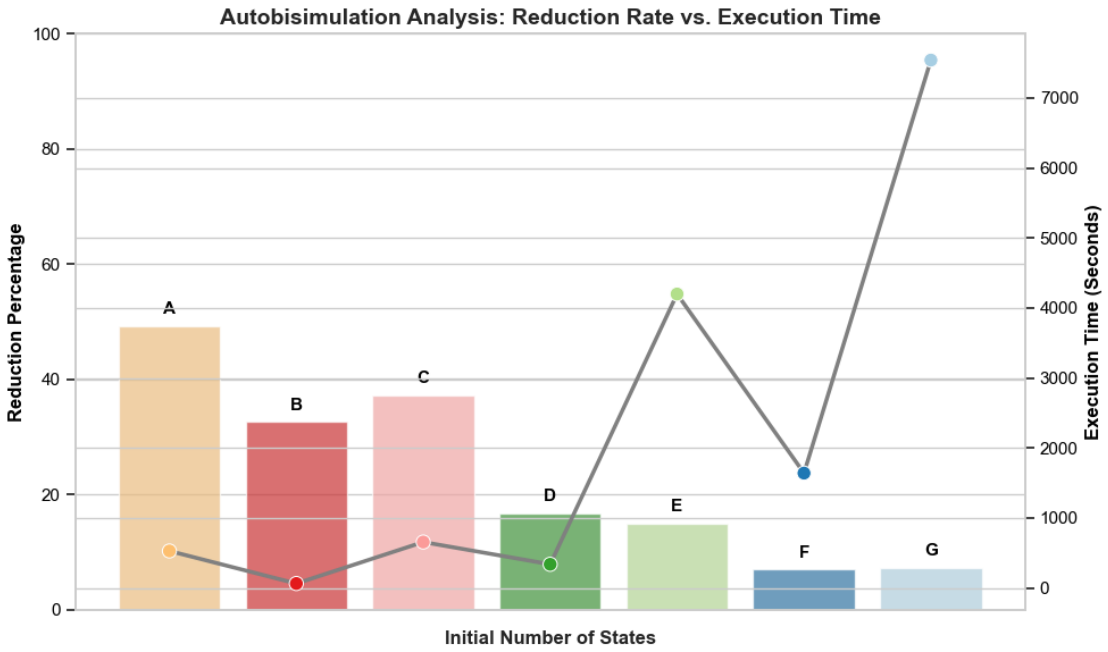


Figure 6.1: Performance Analysis on Autobisimulation search.

NFA	$\|Q\|_i$	$\|Q\|$	R(%)	Time
A	503	255	49.304	0:08:49.190
B	509	343	32.613	0:01:04.189
C	1036	650	37.259	0:10:56.728
D	1093	910	16.743	0:05:39.056
E	2466	2097	14.963	1:09:59.650
F	2619	2437	6.949	0:27:21.374
G	4292	3979	7.293	2:05:41.477

$|Q|_i$: Size of the initial automaton

$|Q|$: Final size of the automaton

$R(\%)$: Percentage of the search reduction

Table 6.2: Results of *test_FAdo_autoBisimulation2*.

6.1.2 Comparing our reduction functions

In this subsection, we present the results obtained by systematically applying the functions proposed in Chapter 4. By comparing them, we can justify the approach taken in the algorithms presented in chapter 5.

For this, we ran each test of the reduction functions 20 times, where, on each test, we systematically ran the function under testing until no further reductions were possible. Then, we collected the results of the best iteration out of the 20 and summarised them in Figure 6.2 and Table 6.3.

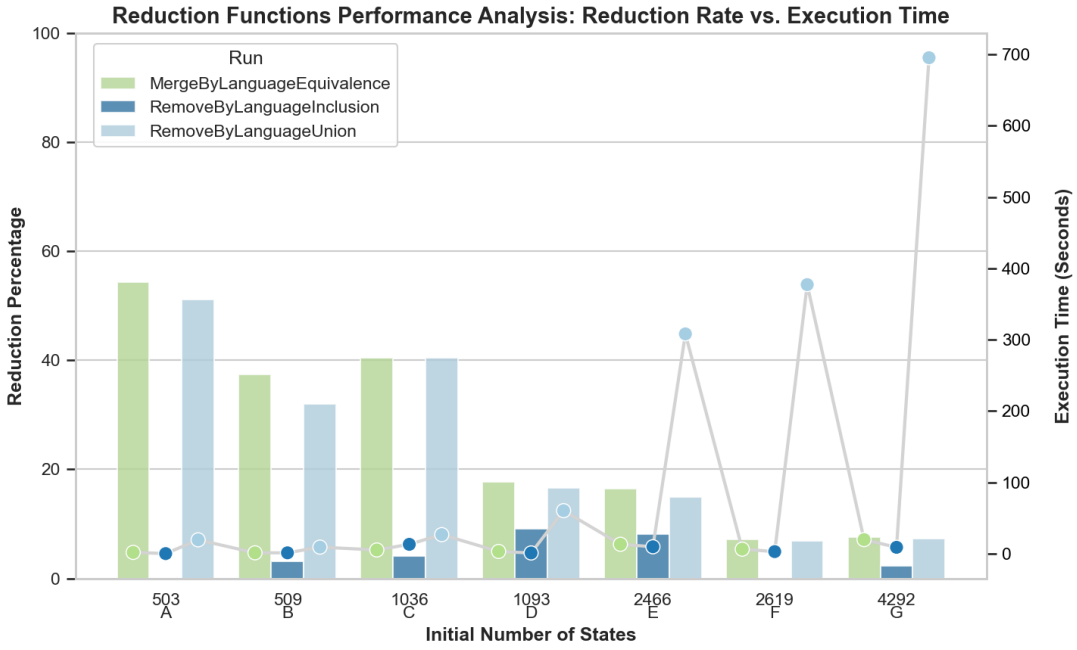


Figure 6.2: Performance comparison between our reduction functions.

		(1)		*(2)*		*(3)*	
NFA	$\|Q\|_i$	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)
A	503	54.27	1.8	0	<0.1	51.09	19.4
B	509	37.33	1.2	3.14	1.2	32.02	9.2
C	1036	40.45	5.2	4.15	13.1	40.44	26.7
D	1093	17.66	3.1	9.15	1.0	16.56	60.3
E	2466	16.46	13.2	8.11	9.7	14.88	308.2
F	2619	7.14	6.3	0.0	2.9	6.87	377.3
G	4292	7.57	19.7	2.33	8.9	7.32	695.5

R(%): Percentage of the search reduction

T(s): Execution time in seconds

(1): Results of $mergeByPartitionRefinement$

(2): Results of $rule\_of\_eq\_1$

(3): Results of $removeByStateRightLanguageUnion$

Table 6.3: Results of testing the search functions.

From these initial results we can easily tell that the method *(1)* can achieve better results than the other methods, not only that, but it can achieve a better result within a fraction of the time taken by *(3)*, hence we chose to always minimize first the NFA subject to study by the method *(1)*

From these initial results, it is evident that method *(1)*, corresponding to *mergeByPartitionRefinement*, consistently achieves higher reductions across all tested NFAs when compared to methods *(2)* and *(3)*. Additionally, it achieves this while maintaining a low execution time, comparable to or better than method *(2)*, and significantly faster than method *(3)*. For instance, in the case of NFA A, method *(1)* achieves a *54.27%* reduction in just 1.8 seconds, while method *(3)*, although close in effectiveness (*51.09%*), requires 19.4 seconds, corresponding to a slowdown factor of approximately 10.8.

This trend holds across all NFAs tested, indicating that method *(1)* not only offers a more efficient reduction but also scales better in terms of performance. Method *(2)*, while extremely fast, yields negligible reductions in most cases, suggesting that its application alone is insufficient for meaningful reduction. However, we can still use it to complement method *(1)*.

Consequently, given its superiority in both effectiveness and efficiency, we adopt method *(1)* as the primary reduction strategy in our searches. This prioritization aims to reduce computational overhead, and keep tractable runtimes in more extensive experiments.

6.2. First Improve Descent

The *FirstImproveDescent* function is a greedy search that iteratively looks for sets of mergeable states, merges a random set of mergeable states, and returns to the same search until no more mergeable sets of states are found.

We ran this test 20 times, from which we collected the results of each execution. Figure 6.3 and Table 6.4 report the worst, best and average outcomes.

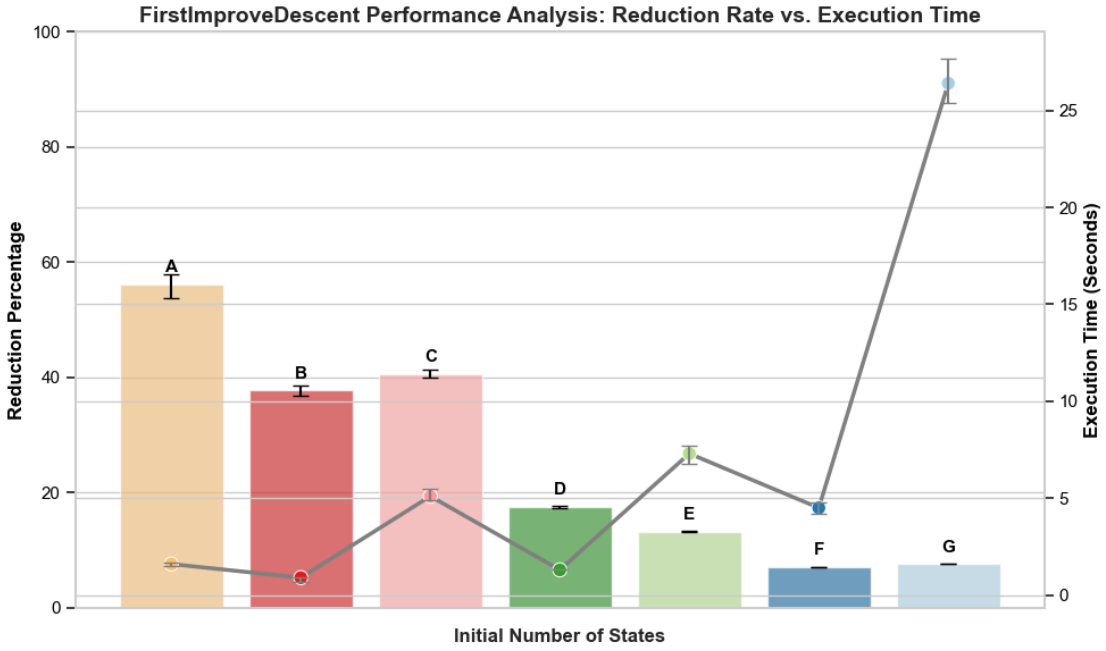


Figure 6.3: Performance Analysis on FirstImproveDescent search.

		Worst			Average			Best			
NFA	$\|Q\|_i$	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$Var(\|Q\|)$
A	503	233	53.68	1.5	220.55	56.15	1.6	212	57.85	1.7	16.3475
B	509	322	36.73	0.9	316.89	37.74	0.9	313	38.50	1.1	3.2579
C	1036	623	39.87	4.7	615.08	40.63	5.1	607	41.41	4.9	12.0136
D	1093	903	17.38	1.3	900.96	17.57	1.3	900	17.66	1.3	0.7589
E	2466	2061	13.22	6.9	2060.29	13.25	7.3	2060	13.26	7.8	0.2059
F	2619	2432	7.14	4.2	2432	7.14	4.5	2432	7.14	4.2	0
G	4292	3968	7.55	27.7	3967.25	7.57	26.4	3967	7.57	27.4	0.1875

$|Q|_i$: Size of the initial automaton

$|Q|$: Final size of the automaton

$R(\%)$: Percentage of the search reduction

$T(s)$: Execution time in seconds

Var : Variance of the final sizes

Table 6.4: Results of FirstImproveDescent.

6.3. Merge-expand search

This search runs the same search as *first_improve_descent* and, when no more mergeable state sets are found, it performs a round of openings, returning to *first_improve_descent* after the opening round. The process repeats until the search reaches the defined number of opening rounds and there are no more mergeable state sets.

In our third test, we ran the search *merge_expand_search* with fifty (50) rounds of openings with ten (10) opens in each iteration. We ran this test 10 times; Figure 6.4 and Table 6.5 report the worst, best and average outcomes.

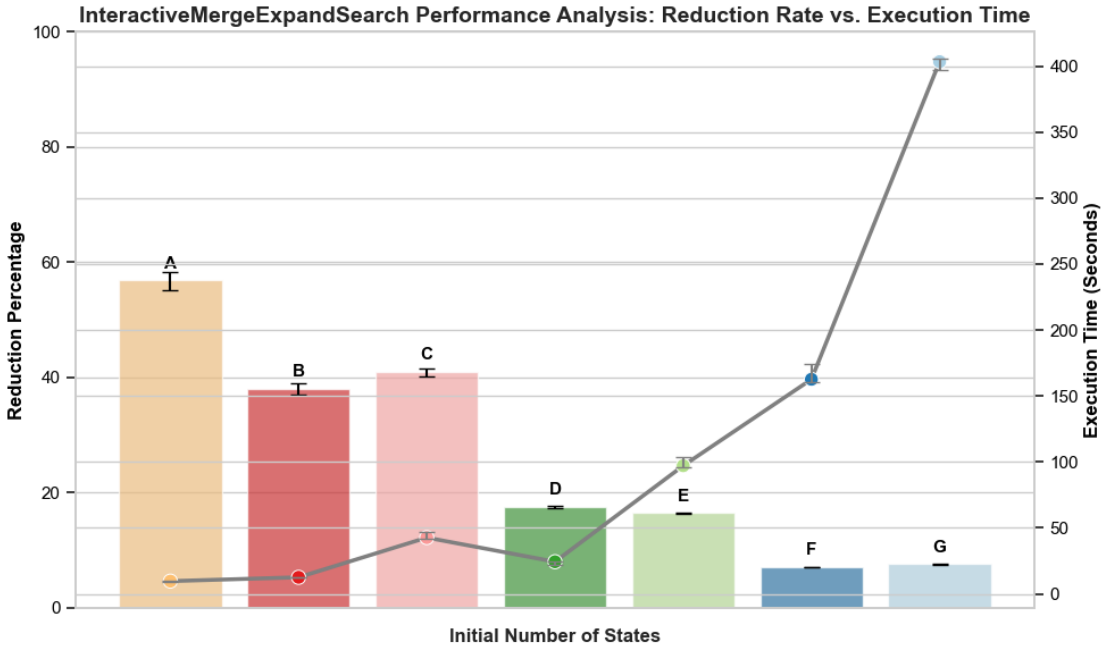


Figure 6.4: Performance analysis on IterativeMergeExpand search 1.

		Worst			Average			Best			
NFA	$\|Q\|_i$	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$Var(\|Q\|)$
A	503	226	55.07	9.8	217.1	56.84	9.6	210	58.25	9.3	22.29
B	509	321	36.94	12.5	315.7	37.98	12.5	311	38.90	12.6	2.45
C	1036	619	40.25	46.9	611.8	40.95	42.6	605	41.60	43.3	9.54
D	1093	903	17.38	24.7	901.24	17.54	24.3	900	17.66	21.9	0.76
E	2466	2062	16.38	103.5	2060.43	16.45	97.1	2060	16.46	98.5	0.27
F	2619	2432	7.14	174.2	2432	7.14	162.6	2432	7.14	160.8	0
G	4292	3969	7.53	400.7	3967.6	7.56	403.4	3967	7.57	409.7	0.44

$|Q|_i$: Size of the initial automaton

$|Q|$: Final size of the automaton

$R(\%)$: Percentage of the search reduction

$T(s)$: Execution time in seconds

Var : Variance of the final sizes

Table 6.5: Results of Iterative Merge-Expand Search 1.

Then, we ran the same search with fifty (50) rounds of openings with twenty five (25) opens in each iteration. We ran this test 10 times; Figure 6.5 and Table 6.6 report the worst, best and average outcomes.

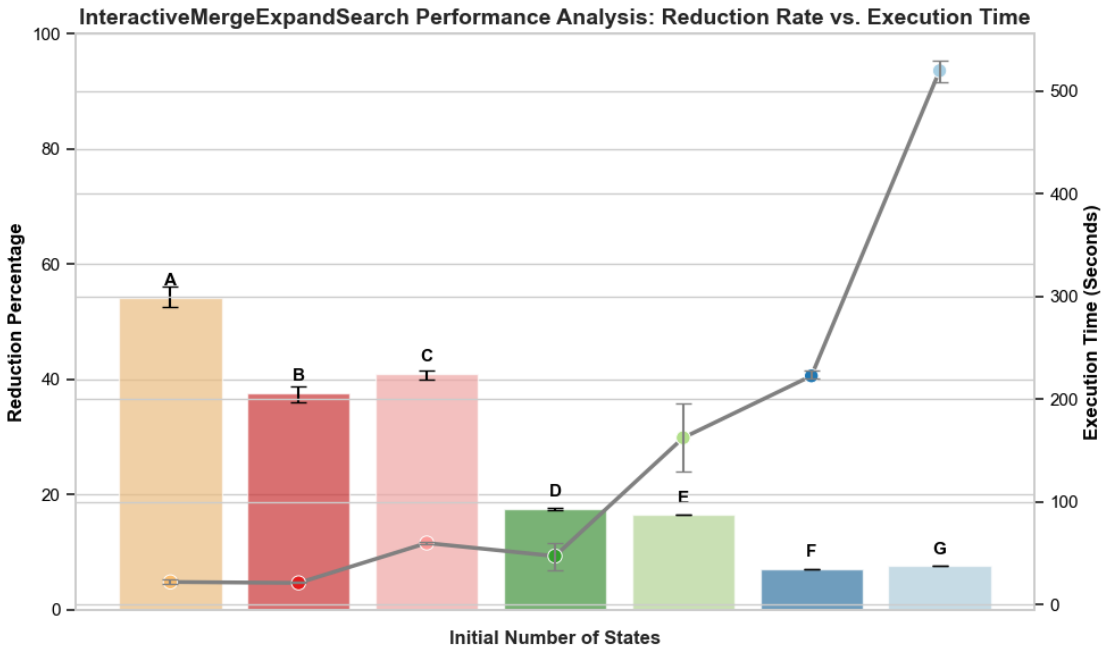


Figure 6.5: Performance analysis on IterativeMergeExpand search 2.

		Worst			Average			Best			
NFA	$\|Q\|_i$	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$Var(\|Q\|)$
A	503	238	52.68	25.1	230.3	54.21	22.1	221	56.06	20.6	29.41
B	509	326	35.95	22.5	317.49	37.62	21.2	312	38.70	21.3	7.8899
C	1036	623	39.87	61.1	611.84	40.94	60.0	605	41.60	59.1	14.1944
D	1093	903	17.38	34.7	901.18	17.55	47.3	900	17.66	33.7	0.8476
E	2466	2061	16.42	196.2	2060.39	16.45	162.4	2060	16.46	195.3	0.2379
F	2619	2432	7.14	227.8	2432.0	7.14	222.6	2432	7.14	220.3	0.0
G	4292	3968	7.55	509.2	3967.4	7.56	519.5	3967	7.57	507.8	0.24

$|Q|_i$: Size of the initial automaton

$|Q|$: Final size of the automaton

$R(\%)$: Percentage of the search reduction

$T(s)$: Execution time in seconds

Var : Variance of the final sizes

Table 6.6: Results of Iterative Merge-Expand Search 2.

Still, with the same search, we ran the search with five (5) rounds of openings with one hundred (100) opens in each iteration. We ran this test 10 times; Figure 6.6 and Table 6.7 report the worst, best, and average outcomes.

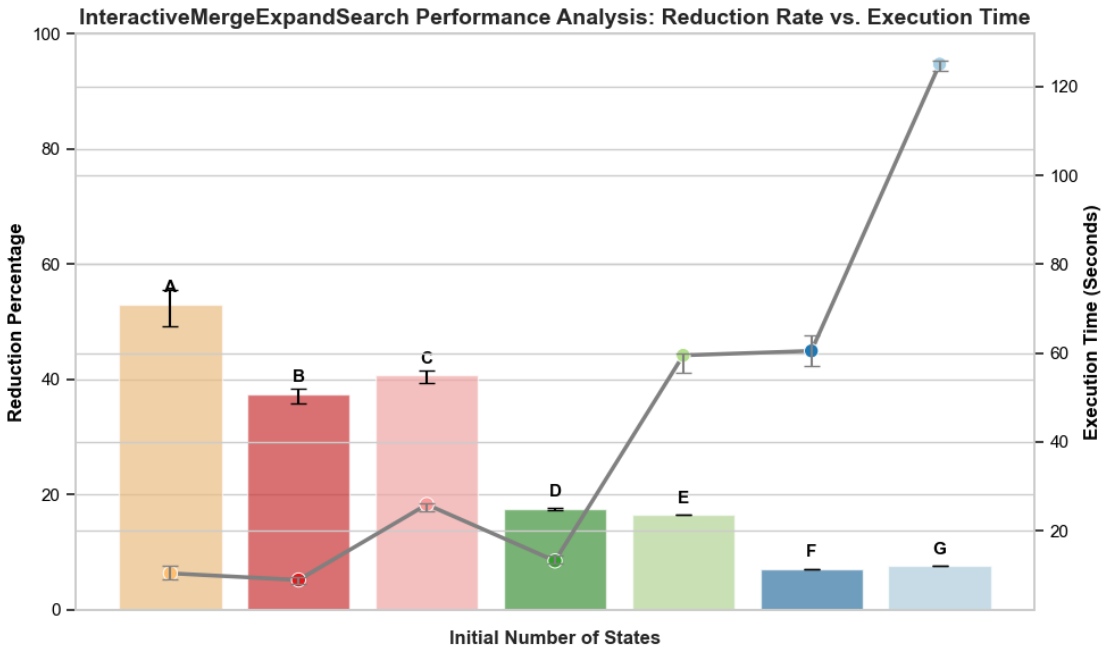


Figure 6.6: Performance analysis on IterativeMergeExpand search 3.

		Worst			Average			Best			
NFA	$\|Q\|_i$	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$Var(\|Q\|)$
A	503	255	49.30	12.2	236.5	52.98	10.4	224	55.47	9.1	72.05
B	509	327	35.76	9.5	318.64	37.40	8.9	314	38.31	8.1	7.65
C	1036	629	39.29	26.1	614.69	40.67	25.9	606	41.51	24.5	17.31
D	1093	904	17.29	12.1	901.25	17.54	13.2	900	17.66	12.6	0.79
E	2466	2061	16.42	59.1	2060.3	16.45	59.5	2060	16.46	63.3	0.21
F	2619	2432	7.14	63.9	2432.0	7.14	60.5	2432	7.14	63.9	0.0
G	4292	3968	7.55	126.0	3967.4	7.56	125.1	3967	7.57	126.5	0.24

$|Q|_i$: Size of the initial automaton

$|Q|$: Final size of the automaton

$R(\%)$: Percentage of the search reduction

$T(s)$: Execution time in seconds

Var : Variance of the final sizes

Table 6.7: Results of Iterative Merge-Expand Search 3.

6.4. Tabu search results

The TabuSearch results are presented in Figure 6.7 and Table 6.8.

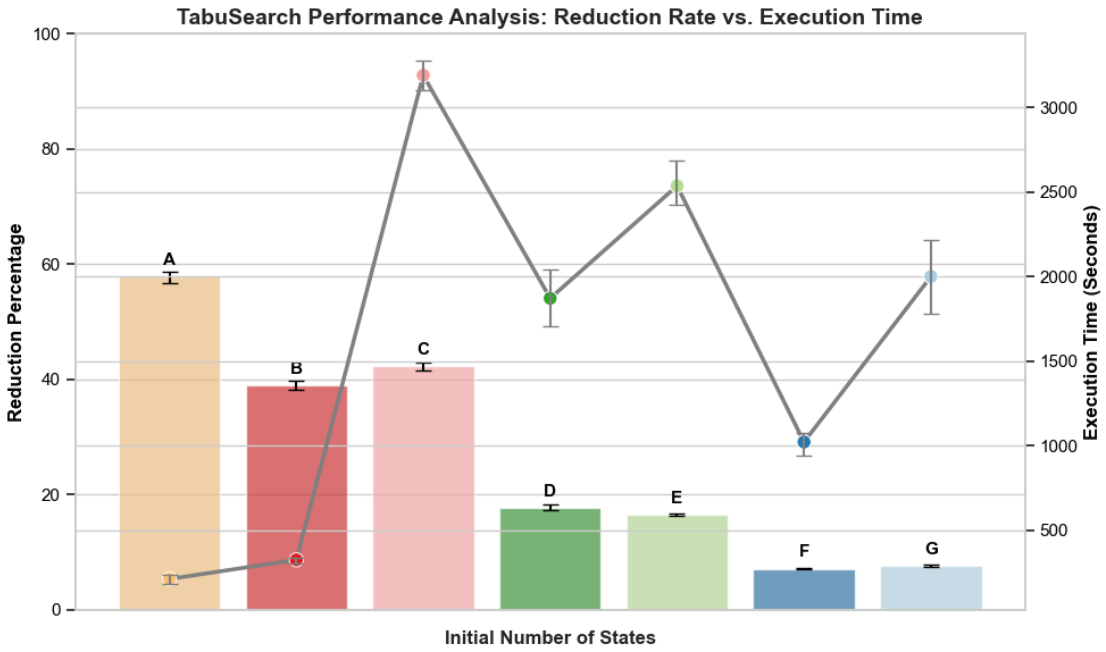


Figure 6.7: Performance analysis on TabuSearch.

		Worst			Average			Best			
NFA	$\|Q\|_i$	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$\|Q\|$	R(%)	T(s)	$Var(\|Q\|)$
A	503	218	56.66	190.4	212	57.85	212.7	208	58.65	185.8	14.8
B	509	315	38.11	310.2	311	38.90	325.7	307	39.69	303.6	6.12
C	1036	606	41.51	3102.3	598	42.28	3191.8	592	42.86	3279.5	10.34
D	1093	905	17.20	1699.4	900	17.66	1872.6	894	18.21	2038.1	0.92
E	2466	2065	16.26	2388.6	2060	16.46	2537.3	2052	16.79	2645.0	0.47
F	2619	2435	7.03	971.4	2432	7.14	1022.7	2430	7.22	1105.0	0.29
G	4292	3972	7.46	1784.8	3966	7.60	2001.2	3960	7.74	2219.0	0.41

$|Q|_i$: Size of the initial automaton

$|Q|$: Final size of the automaton

$R(\%)$: Percentage of the search reduction

$T(s)$: Execution time in seconds

Var : Variance of the final sizes

Table 6.8: Results of TabuSearch.

6.5. Results summary

In summary, the experiments confirm that metaheuristic search strategies consistently outperform classical reduction methods such as state bisimulation, particularly for NFAs that recognise large and more complex languages. The First Improve Descent heuristic proved efficient but prone to early stagnation, while the Iterative Merge–Expand Search

and Tabu Search achieved substantially greater reductions at the cost of additional runtime. These findings highlight a clear trade-off between computational effort and the quality of the solution, as shown in Figure 6.8 and Table 6.9, and demonstrate that the benefits of advanced metaheuristics increase with the size and complexity of the automaton.

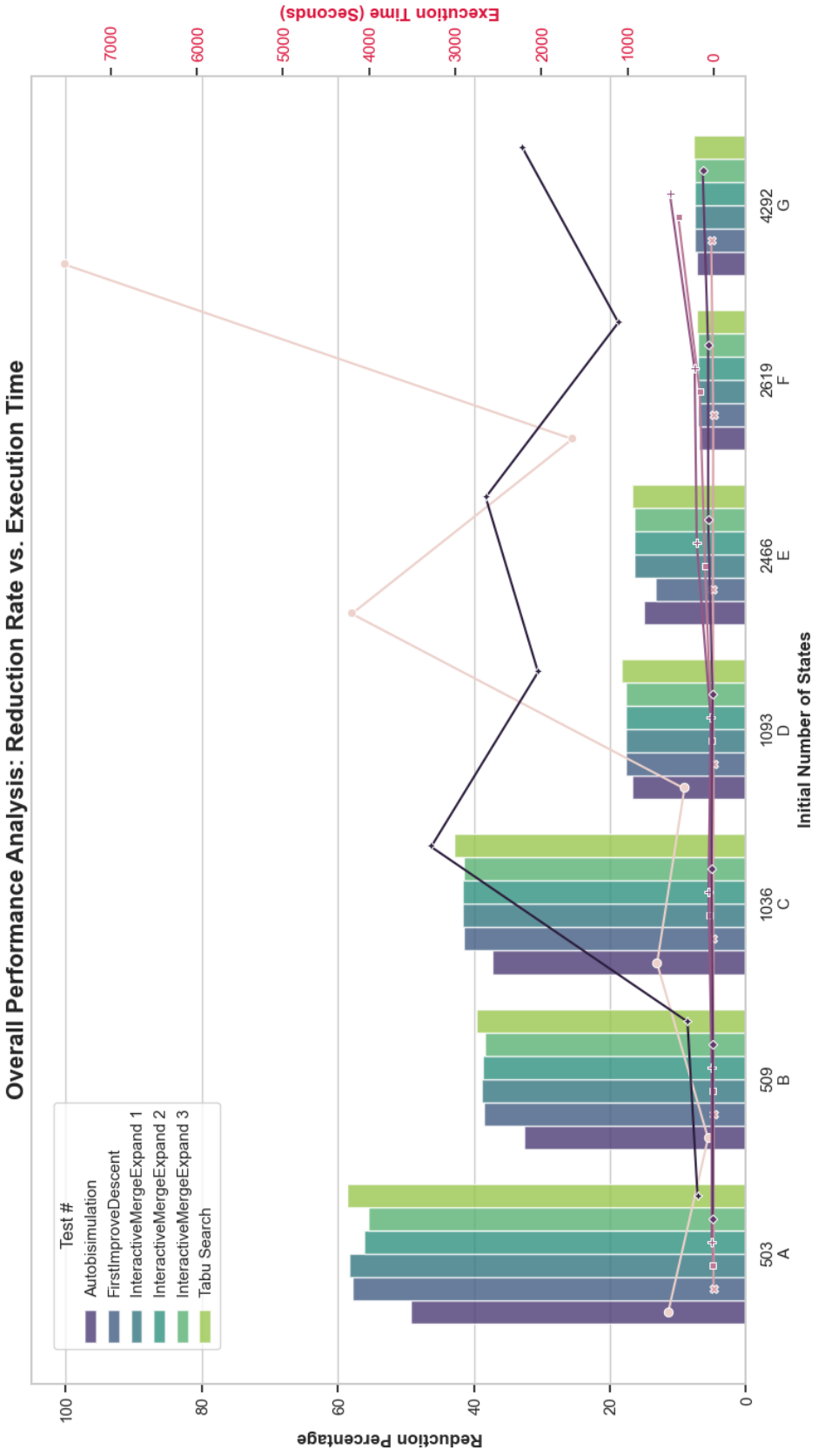


Figure 6.8: Overall performance comparison.

		(1)		(2)		(3)		(4)		(5)		(6)	
NFA	$\|Q\|_i$	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)	R(%)	T(s)
A	503	49.30	529.2	57.85	1.7	58.25	9.3	56.06	20.6	55.47	9.1	58.65	185.8
B	509	32.61	64.2	38.50	1.0	38.90	12.6	38.70	21.4	38.31	8.1	39.69	303.6
C	1036	37.26	656.7	41.41	4.9	41.60	43.4	41.60	59.0	41.51	24.5	42.86	3279.5
D	1093	16.74	339.1	17.66	1.3	17.66	21.9	17.66	33.7	17.66	12.6	18.21	2038.1
E	2466	14.96	4199.7	13.26	7.8	16.46	98.5	16.46	195.3	16.46	63.3	16.79	2645.9
F	2619	6.95	1641	7.14	4.2	7.14	160.8	7.14	220.3	7.14	63.9	7.22	1105.6
G	4292	7.29	7541.5	7.57	27.4	7.57	409.7	7.57	507.8	7.57	126.5	7.74	2219.3

$R(\%)$: Percentage of the search reduction

$T(s)$: Execution time in seconds

(1): Results of best execution of *test_FAdo_autobisimulation2*

(2): Results of the best execution of *first_improve_descent*

(3): Results of the best execution of *merge_expand_search* 1

(4): Results of the best execution of *merge_expand_search* 2

(5): Results of the best execution of *merge_expand_search* 3

(6): Results of the best execution of *TabuSearch*

Table 6.9: Comparing the executions of the presented searches.

7. Practical example: Wi-Fi QR code validator

As part of the MSc in Networks and Computer Systems Engineering, the practical relevance of formal verification methods in network systems is central to the degree. A concrete application arises in validating structured text payloads transmitted over network interfaces—such as Wi-Fi configuration QR codes—where correctness must be guaranteed efficiently and at scale. This section illustrates how the NFA reduction techniques developed in this thesis can be applied to such a real-world network validation task.

Many QR scanners decode a QR code into a plain text payload, which then must be checked against an expected schema for validation. A typical example is the Wi-Fi configuration payload, where the content should encode either a protected network with a password or an open network without one.

To prevent issues with special characters in regex parsers and to keep the automaton over a small fixed alphabet, in our validator, we can first map the $ASCII$ payload to binary using an 8-bit encoding. This allows us to create an NFA with a binary alphabet, i.e., $\Sigma = \{0,1\}$.

Formally, let $enc_8 : ASCII \to \{0,1\}^8$ be the standard 8-bit encoding and extend it to strings by concatenation:

$$ENC(a_0a_1...a_n) = enc_8(a_0) \cdot enc_8(a_1) \cdot ... \cdot enc_8(a_n)$$

Now, let $A \subseteq ASCII$ be the set of allowed characters for tokens (e.g., letters, digits, and special characters). We define the one-character encoding language $STRING_A$ and the token language $TOKEN_A$ as:

$$STRING_A = \bigcup_{a \in A} ENC(a) \quad \text{and} \quad TOKEN_A = STRING_A \cdot STRING_A^\star$$

Now, for the patterns expected of a protected Wi-Fi connection, we can define the following language:

$$\begin{aligned} L_{protected}(A) = {} & ENC(\text{'WIFI:T:'}) \cdot \Big(ENC(\text{'WPA'}) + ENC(\text{'WEP'})\Big) \cdot \\ & ENC(\text{';S:'}) \cdot TOKEN_A \cdot ENC(\text{';P:'}) \cdot TOKEN_A \cdot ENC(\text{';;'}) \end{aligned}$$

And for Wi-Fi connections that don't require a password, we define the following language:

$$L_{open}(A) = ENC(\text{'WIFI:T:'}) \cdot ENC(\text{'nopass'}) \cdot ENC(\text{';S:'}) \cdot TOKEN_A \cdot ENC(\text{';;'}) .$$

From these languages, we can extract the language of the automaton that can validate a QR code connection description as the union of both languages above.

$$L = L_{protected} \cup L_{open}$$

This language is regular because it has finite unions and concatenations; therefore, it can be recognised by finite automata.

Now, using the $FAdo$ library, we will generate two NFAs: one using Glushkov's position construction algorithm (NFA A) and another using Antimirov's partial derivative algorithm (NFA B), respectively, $nfaPosition$ and $nfaPD$, in the $FAdo$ library. Table 7.1 presents the size of the generated automata before reduction.

NFA	$\|Q\|$	$\|F\|$	$\|\delta\|$	$\|\Sigma\|$
A	3577	2	31522	2
B	653	1	924	2

Table 7.1: Initial NFAs generated for the Wi-Fi QR validator.

By applying our $TabuSearch$ with different values of k (the tabu tenure, i.e. the number of search iterations for which a newly created state remains in the tabu list, as defined in Section 5.3), we got the results shown in Table 7.2. The value $k = 10$ was included to test whether a longer tabu memory improves the reduction; in this example, $k = 7$ produced the smallest automata for both constructions.

	k = 5		k = 7		k = 10	
NFA	$\|Q\|_F$	$\|\delta\|_F$	$\|Q\|_F$	$\|\delta\|_F$	$\|Q\|_F$	$\|\delta\|_F$
A	358	651	314	464	342	560
B	264	337	262	320	264	328

Table 7.2: Wi-Fi QR validator reduction results using TabuSearch.

Now, by applying the reduction algorithm of auto-bisimulation from $FAdo$ on the right language equivalence, we get the results shown in Table 7.3.

From the resulting automata, we compared the decidability time required to determine whether a given word from a set of words is part of the language of the automaton or not.

NFA	$\|Q\|_F$	$\|\delta\|_F$	$Runtime$
A	3098	27880	1:32:44
B	566	828	0:07:25

Table 7.3: Wi-Fi QR validator reduction results using FAdo auto-bisimulation.

The timings are presented in Table 7.4.

Note: For this test, we used the resulting automata from the $TabuSearch$ with $k = 7$, as it was the best-performing configuration in our tests.

	(a)		(b)	
	NFA_A	NFA_B	NFA_A	NFA_B
w_1	0.46	0.36	1.80	0.86
w_2	0.44	0.38	1.72	0.81
w_3	0.49	0.35	1.77	0.83
w_4	0.43	0.39	1.85	0.85
w_5	0.47	0.37	1.74	0.82
w_6	0.45	0.34	1.69	0.80
w_7	0.41	0.36	1.83	0.84
w_8	0.50	0.38	1.76	0.79
w_9	0.42	0.35	1.81	0.86
w_{10}	0.46	0.37	1.73	0.81
***Avg**	**0.45**	**0.37**	**1.77**	**0.83**

(a) Testing results of $TabuSearch$
(b) Testing results of $autobisimulation$
* Average of the column

Table 7.4: Testing results for the Wi-Fi QR validator decidability in seconds.

Here, we showcased the application of our approach in a real-world validation task: verifying that Wi-Fi QR codes adhere to the expected schema. By modelling the QR code payload as a regular language and encoding it into an automaton, we could thoroughly check whether a given string represents a valid protected or open Wi-Fi configuration. The experimental results with different NFA constructions and reduction strategies demonstrate that our framework not only captures the required patterns but can also be optimised to achieve significant reductions in states and transitions, without compromising the language accepted by the automaton.

This use case is just one example of the wide potential of our method. Many QR-based standards, such as payment payloads, event tickets, or vCards, rely on structured text formats that can also be expressed as regular languages. By automatically generating,

reducing, and efficiently using automata for validation tasks, our work highlights its practical relevance and applicability to various domains where robustness and performance are key.

8. Conclusions

In this thesis, we addressed the problem of reducing the number of states in non-deterministic finite automata (NFAs), a problem known to be PSPACE-complete. Traditional approaches, such as state bisimulation, are helpful, but the reduction is not optimal for NFAs. To overcome these limitations, we developed and implemented a suite of metaheuristic strategies tailored for NFA state reduction.

We began by exploring theoretical foundations, particularly the properties of regular languages and automata. We defined concepts for the languages of automata states, such as left, right, and inner languages, and used these definitions to formalise a set of reduction rules. These rules allowed us to identify and remove (or merge) states without altering the language recognised by the automaton.

Building upon these foundations, we introduced some heuristic-based reduction functions: merging by language equivalence, removal by language inclusion, and removal by language union. To escape local minima in the space of automata configurations for a given language, we proposed an approach based on state opening (splitting the states' languages) that creates potential for further reductions by introducing new mergeable states through controlled splitting.

We then integrated these techniques into two metaheuristic algorithms: *FirstImproveDescent*, a greedy strategy focusing on systematic reductions, and *IterativeMergeExpandSearch*, which alternates between state merging and expansion to explore a broader solution space. Both algorithms were implemented in Python, leveraging the FAdo library for automata manipulation.

Our experimental results demonstrate that these approaches consistently outperform traditional methods, such as FAdo's autobisimulation, in terms of reduction rate and runtime, particularly when dealing with large NFAs.

Beyond these two methods, we subsequently introduced a third metaheuristic, *TabuSearch*, which is built on top of *IterativeMergeExpandSearch* with short-term, path-aware memory. After each opening, the newborn sibling states are temporarily marked as tabu, preventing immediate reversals (i.e., merges/removals that would collapse them back). By employing the tabu technique, we were able to prevent cycling in a local optimum and

take the search toward unexplored regions of the solution space. As shown in Chapter 6, *TabuSearch* complements *IterativeMergeExpandSearch*. The *TabuSearch* often achieves the greatest reductions, though it requires more time to find better solutions.

To assess practical relevance, we also included a case study on validating Wi-Fi QR code payloads. We defined the schema of the validator as a regular language (by 8-bit encoding over a binary alphabet) and built NFAs using Glushkov and Antimirov construction algorithms. Applying our *TabuSearch*, we produced substantially smaller automata while preserving the initial automata language. From the reduced NFAs, we can verify the improvement in membership (decidability) time on a set of test strings, illustrating how our metaheuristic can directly translate into faster validations.

In summary, this work contributes both theoretical insights and practical tools for NFA state reduction. The proposed metaheuristics provide language-preserving methods that are both more scalable and more efficient than current alternatives for the NFA state reduction problem. Future work could focus on adaptive transformations and defining opening policies (potentially guided learning), dynamic tabu tenure/aspiration driven by progress signals, and broader validation on domain-specific automata where structure can be leveraged for further gains.

References

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Addison-Wesley, 2006. [Cited on pages 3, 4, and 7.]

[2] J. Myhill, "Finite-state machines," *Annals of Mathematics Studies*, vol. 34, pp. 127–141, 1957. [Cited on page 4.]

[3] J. E. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," *Theory of Machines and Computations*, pp. 189–196, 1971. [Cited on page 7.]

[4] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies*, ser. Annals of Mathematics Studies, C. E. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956, no. 34, pp. 129–153. [Cited on page 7.]

[5] J. Berstel, L. Boasson, and O. Carton, "Minimization of automata," Slide deck, Jun. 2009, talk slides (Liège, 8 June 2009). Includes complexity discussion of Moore's algorithm. [Online]. Available: https://www-igm.univ-mlv.fr/~berstel/Exposes/2009-06-08MinimisationLiege.pdf [Cited on page 7.]

[6] D. M. R. Park, "Concurrency and automata on infinite sequences," in *Proceedings of the 5th GI Conference on Theoretical Computer Science*, ser. Lecture Notes in Computer Science, P. Deussen, Ed., vol. 104. Springer, 1981, pp. 167–183. [Cited on page 8.]

[7] M. Almeida, N. Moreira, and R. Reis, "FAdo and GUItar: Tools for automata and regular expressions," in *Implementation and Application of Automata*, ser. Lecture Notes in Computer Science, vol. 5642. Springer, 2009, pp. 65–74. [Cited on page 8.]

[8] V. Halava, T. Harju, and J. Karhumäki, "Minimization of nondeterministic finite automata is PSPACE-complete," *Theoretical Computer Science*, vol. 327, no. 3, pp. 311–324, 2004. [Cited on page 9.]

[9] M. Sipser, *Introduction to the Theory of Computation*, third edition, international edition ed. Boston, Massachusetts: Cengage Learning, 2013. [Cited on page 9.]

[10] F. Glover, "Tabu search—part i," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989. [Cited on page 13.]

Appendix A

Function 1 GetMergibleStatesByPartitions

```
def getMergibleStatesByPartitions(self, nfaR:FAdo.fa.NFAr):
    leftPartitions = dict()
    rightPartitions = dict()
    for state in range(0, len(nfaR.States)):
        state_name_prefix = ""
        if state in nfaR.Initial:
            state_name_prefix += "I"
        if state in nfaR.deltaReverse.keys():
            inDelta = self.getListOfTuplesFromDict(nfaR.deltaReverse[state])
            str_inDelta = state_name_prefix + str(inDelta)
            if str_inDelta not in leftPartitions:
                leftPartitions[str_inDelta] = set()
            leftPartitions[str_inDelta].add(state)
            str_inDelta_loop = self.getLoopString(str_inDelta, state)
            if str_inDelta_loop is not None:
                if str_inDelta_loop not in leftPartitions:
                    leftPartitions[str_inDelta_loop] = set()
                leftPartitions[str_inDelta_loop].add(state)
        if state in nfaR.delta.keys():
            outDelta = self.utils.getListOfTuplesFromDict(nfaR.delta[state])
            str_outDelta = state_name_prefix + str(outDelta)
            if str_outDelta not in rightPartitions:
                rightPartitions[str_outDelta] = set()
            rightPartitions[str_outDelta].add(state)
            str_outDelta_loop = self.getLoopString(str_outDelta, state)
            if str_outDelta_loop is not None:
                if str_outDelta_loop not in rightPartitions:
                    rightPartitions[str_outDelta_loop] = set()
                rightPartitions[str_outDelta_loop].add(state)
    leftPartitions = [v for k, v in leftPartitions.items() if len(v) > 1]
    rightPartitions = [v for k, v in rightPartitions.items() if len(v) > 1]
    return leftPartitions, rightPartitions
```

Function 2 RemoveStateByLanguageInclusion

```
def removeStatesByLanguageInclusion(self, nfa:FAdo.fa.NFA):
    for s1 in range(len(nfa.States)):
        if s1 in nfa.Initial:
            continue
        for s2 in range(len(nfa.States)):
            if s1 == s2 or s2 in nfa.Initial:
                continue
            if s1 in nfa.Final and s2 not in nfa.Final:
                continue
            is_subset = self.compareOutset_isS1SubsetOfS2(nfa, s1, s2)
            if is_subset:
                nfaR = nfa.toNFAr()
                is_subset_r = self.compareInset_isS1SubsetOfS2(nfaR, s1, s2)
                if is_subset_r:
                    nfaR.deleteStates({s1})
                    return nfaR.toNFA()
    return nfa
```

Function 3 CompareOutset_isS1SubsetOfS2

```
def compareOutset_isS1SubsetOfS2(self, nfa:FAdo.fa.NFA, s1, s2):
    if s1 not in nfa.delta:
        return True
    if s2 not in nfa.delta:
        return False
    for sy in nfa.delta[s1]:
        if sy not in nfa.delta[s2]:
            return False
        for rightState in nfa.delta[s1][sy]:
            if rightState == s1:
                if s2 not in nfa.delta[s2][sy]:
                    return False
            if rightState not in nfa.delta[s2][sy]:
                return False
    return True
```

Function 4 CompareInset_isS1SubsetOfS2

```
def compareInset_isS1SubsetOfS2(self, nfaR:FAdo.fa.NFAr, s1, s2):
    if s1 not in nfaR.deltaReverse:
        return True
    if s2 not in nfaR.deltaReverse:
        return False
    for sy in nfaR.deltaReverse[s1]:
        if sy not in nfaR.deltaReverse[s2]:
            return False
        for rightState in nfaR.deltaReverse[s1][sy]:
            if rightState == s1:
                if s2 not in nfaR.deltaReverse[s2][sy]:
                    return False
            if rightState not in nfaR.deltaReverse[s2][sy]:
                return False
    return True
```

Function 5 RemoveByStateLanguageUnion

```
def removeByStateLanguageUnion(self, nfa:FAdo.fa.NFA):
    mergibleStatesSet = self.find_union_transitions(nfa.delta, nfa.Initial, nfa.Final)
    if len(mergibleStatesSet) == 0:
        return nfa
    for toMerge in mergibleStatesSet:
        to_remove = {x for x in toMerge if isinstance(x, int)}
        tuples = copy(toMerge)
        for stateToRemove in to_remove:
            for stateUnion in tuples:
                if isinstance(stateUnion, int) and stateUnion != stateToRemove \
                        and ((stateToRemove in nfa.Final and stateUnion in nfa.Final) or (
    stateToRemove not in nfa.Final and stateUnion not in nfa.Final)):
                    nfa = self.copyLeftTransitions_withLoops(nfa, stateToRemove, stateUnion)
                    nfaR = nfa.toNFAr()
                    nfaR.deleteStates(to_delete)
                    return nfaR.toNFA()
                if isinstance(stateUnion, tuple):
                    if stateToRemove not in nfa.Final or self.compareCondition_containsFinal(nfa,
    stateUnion):
                        if stateToRemove not in stateUnion:
                            for state in stateUnion:
                                nfa = self.copyLeftTransitions(nfa, stateToRemove, state)
                                nfaR = nfa.toNFAr()
                                nfaR.deleteStates(to_delete)
                                return nfaR.toNFA()
    return nfa
```

Function 6 FindUnionTransitions

```
def find_union_transitions(self, delta: dict):
    results = list()
    states = list(delta.keys())
    tabu = set()
    langMap = dict()
    langMap_extended = dict()
    for state in states:
        stateLang = self.getListOfTuplesFromDict(delta[state])
        if stateLang == set():
            continue
        langMap[state] = stateLang
        deltaWithoutLoop = self.getDeltaWithoutLoop(delta, state)
        stateLang = self.getListOfTuplesFromDict(deltaWithoutLoop)
        if stateLang == set():
            continue
        langMap_extended[state] = stateLang
        for s in states:
            if s >= state:
                break
            if s in langMap.keys():
                langMap[(s, state)] = self.getUnionTransitionSet(langMap[state], langMap[s])
    # Group keys by their values
    value_to_keys = {}
    for key, value in langMap.items():
        value_to_keys.setdefault(str(value), []).append(key)
    for key, value in langMap_extended.items():
        value_to_keys.setdefault(str(value), []).append(key)
    # Filter elements that contain both a singleton and a tuple
    return [keys for keys in value_to_keys.values() if len(keys) > 1 and any(isinstance(x, int) for
      x in keys)]
```

Function 7 OpenRandomCandidateState

```
def openRandomCandidateState(self, nfa:FAdo.fa.NFA):
    rightCandidates = self.getBestCandidatesToOpenRight(nfa)
    leftCandidates = self.getBestCandidatesToOpenLeft(nfa)
    if len(rightCandidates) + len(leftCandidates) == 0:
        return nfa
    state = random.choice(list(leftCandidates.union(rightCandidates)))
    if state in rightCandidates:
        new_nfa = self.openStateRightLanguage(nfa, state)
    else:
        new_nfa = self.openStateLeftLanguage(nfa, state)
    return new_nfa
```

Function 8 GetBestCandidatesToOpenRight

```
def getBestCandidatesToOpenRight(self, nfa:FAdo.fa.NFA):
    candidates = set(range(0, len(nfa.States)))
    leftPartitions, rightPartitions = self.getLanguagesPartitions(nfa.toNFAr())
    for state_index in range(len(nfa.States)):
        if state_index not in nfa.delta:
            candidates.remove(state_index)
    for partition in rightPartitions:
        if len(partition) == 1:
            candidates.remove(partition.pop())
    return candidates
```

Function 9 GetBestCandidatesToOpenLeft

```
def getBestCandidatesToOpenLeft(self, nfa:FAdo.fa.NFA):
    candidates = set(range(0, len(nfa.States)))
    leftPartitions, rightPartitions = self.getLanguagesPartitions(nfa.toNFAr())
    for state_index in range(len(nfa.States)):
        if state_index not in nfa.delta:
            candidates.remove(state_index)
    for partition in leftPartitions:
        if len(partition) == 1:
            candidates.remove(partition.pop())
    return candidates
```

Function 10 OpenStateRightLanguage

```
def openStateRightLanguage(self, nfa:FAdo.fa.NFA, state_index:int):
    if state_index not in nfa.delta:
        return nfa
    if state_index in nfa.Initial:
        return nfa
    new_nfa = copy.deepcopy(nfa)
    state_loop_symbols = self.get_state_loop_symbols(nfa, nfa.States[state_index])
    right_transitions = self.getListOfTuplesFromDict(nfa.delta[state_index])
    left_transitions = self.getListOfTuplesFromDict(nfa.toNFAr().deltaReverse[state_index])
    if sum(len(t[1]) for t in right_transitions) - len(state_loop_symbols) < 2:
        return nfa
    if sum(len(t[1]) for t in left_transitions) < 2:
        return nfa
    for transition in right_transitions:
        for right_state in transition[1]:
            if right_state == state_index:
                continue
            new_state_index = nfa.addState()
            new_nfa = self.copyLeftTransitions(nfa, state_index, new_state_index)
            new_nfa.addTransition(new_state_index, transition[0], right_state)
            for loop_symbol in state_loop_symbols:
                new_nfa.addTransition(new_state_index, loop_symbol, new_state_index)
            if state_index in nfa.Final:
                new_nfa.addFinal(new_state_index)
            if state_index in nfa.Initial:
                new_nfa.addInitial(new_state_index)
    new_nfa.deleteStates({state_index})
    return new_nfa
```

Function 11 OpenStateLeftLanguage

```
def openStateLeftLanguage(self, nfa:FAdo.fa.NFA, state_index):
    nfaR = nfa.toNFAr()
    if state_index not in nfaR.deltaReverse or state_index in nfa.Initial:
        return nfa
    new_nfa = copy.deepcopy(nfa)
    state_loop_symbols = self.get_state_loop_symbols(nfa, nfa.States[state_index])
    right_transitions = self.getListOfTuplesFromDict(nfa.delta[state_index])
    left_transitions = self.getListOfTuplesFromDict(nfaR.deltaReverse[state_index])
    if sum(len(t[1]) for t in right_transitions) < 2:
        return nfa
    if sum(len(t[1]) for t in left_transitions) - len(state_loop_symbols) < 2:
        return nfa
    for transition in left_transitions:
        for left_state in transition[1]:
            if left_state == state_index:
                continue
            new_state_index = nfa.addState()
            new_nfa = self.copyRightTransitions(nfa, state_index, new_state_index)
            new_nfa.addTransition(left_state, transition[0], new_state_index)
            for loop_symbol in state_loop_symbols:
                new_nfa.addTransition(new_state_index, loop_symbol, new_state_index)
            if state_index in nfa.Final:
                new_nfa.addFinal(new_state_index)
            if state_index in nfa.Initial:
                new_nfa.addInitial(new_state_index)
    new_nfa.deleteStates({state_index})
    return new_nfa
```

Function 12 FirstImproveDescent

```
def firstImproveDescent(self, nfa:FAdo.fa.NFA):
    while True:
        new_nfa = self.mergeByPartitionRefinement(nfa)
        if len(new_nfa.States) == len(nfa.States):
            new_nfa = self.removeStateByLanguageInclusion(new_nfa)
            if len(new_nfa.States) == len(nfa.States):
                break
        nfa = new_nfa
    return new_nfa
```

Function 13 IterativeMergeExpandSearch

```
def iterativeMergeExpandSearch(self, nfa:FAdo.fa.NFA, n_opens:int, n_per_iter:int):
    while True:
        new_nfa = self.mergeByPartitionRefinement(nfa)
        if len(new_nfa.States) == len(nfa.States):
            new_nfa = self.removeStateByLanguageInclusion(new_nfa)
            if len(new_nfa.States) == len(nfa.States):
                if n_opens > 0:
                    for i in range(n_per_iter):
                        new_nfa = self.openRandomCandidateState(new_nfa)
                    n_opens -= 1
                else:
                    break
        nfa = new_nfa
    return new_nfa
```

Function 14 TabuSearch

```
def tabuSearch(self, initial_nfa:FAdo.fa.NFA, run_time:datetime, tabu_k:int):
    best_nfa = copy(nfa)
    tabu_list = list()
    start_time = datetime.now()
    while True:
        new_nfa = nfa
        if len(new_nfa.States) < len(best_nfa.States):
            best_nfa = copy(new_nfa)
        tabu = set().union(*tabu_list)
        new_nfa = self.mergeByPartitionRefinement(new_nfa, tabu)
        if len(new_nfa.States) == len(nfa.States):
            new_nfa, removed_state = self.rule_of_eq_1(new_nfa)
            if len(new_nfa.States) == len(nfa.States):
                new_nfa = self.removeStateByLanguageInclusion(new_nfa, tabu)
                if len(new_nfa.States) == len(nfa.States):
                    if run_time > datetime.now() - start_time:
                        if len(tabu_list) >= tabu_k:
                            tabu_list = tabu_list[1:]
                        new_nfa, new_states = self.openRandomCandidateState(new_nfa)
                        tabu_list.append(new_states)
                    else:
                        if tabu_list != set():
                            new_nfa = copy(best_nfa)
                            tabu_list = set()
                        else:
                            break
        nfa = copy(new_nfa)
    return best_nfa
```