Implementation and analysis of NFAs with adjustable deterministic blow-up

Andrei Lissovoi

June 28, 2010
Summary

Non-deterministic and deterministic automata both have the expressive power to recognize regular languages; thus, for any non-deterministic automaton there exists an equivalent deterministic automaton that recognizes the same regular language (and vice versa). Depending on the regular language recognized, however, the number of states in minimal NFAs and DFAs recognizing the language may differ significantly; an \( n \)-state NFA can be equivalent to an DFA with as few as \( n \) and as many as \( 2^n \) states.

This project explores the question of whether there always exists an \( n \)-state minimal NFA such that the equivalent minimal DFA has \( n \leq d \leq 2^n \) states. Past research has shown several methods to construct an \( n \)-state minimal NFA for a given \( (n, d) \) pair over varying alphabet sizes; this report presents several of those constructions and proves their correctness.

In addition, some of the constructions are implemented using Java, allowing the NFAs for a particular \( (n, d) \) pair to be quickly constructed, visualized, and converted to DFA or regular expressions representations.
Contents

1 Introduction 4

2 Implementation 6
  2.1 Representing Automata ........................................... 6
  2.2 Converting Automata .............................................. 7
    2.2.1 Subset construction ........................................ 7
    2.2.2 DFA minimization .......................................... 7
    2.2.3 Regular expressions ...................................... 9
  2.3 Applications ................................................... 10

3 Constructions 12
  3.1 Proving that a non-deterministic finite automaton is minimal . 12
  3.2 The two extreme cases ........................................ 13
  3.3 Exponential alphabet size ...................................... 14
  3.4 Linear alphabet size .......................................... 15
  3.5 4-letter alphabet ............................................. 17
  3.6 On smaller alphabets .......................................... 24

4 Conclusion 25

5 References 26
1 Introduction

Regular languages are widely used in programming – both as regular expressions used to search strings, and finite automata allowing modeling and verification of behavior of complex software. There are two types of automata recognizing the class of regular languages: deterministic and non-deterministic finite automata; abbreviated DFA and NFA respectively. Both of those types can be thought of as a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where \(Q\) is a finite set of states, where \(q_0 \in Q\) is the initial state, and states in \(F \subseteq Q\) are considered accepting, \(\Sigma\) is the alphabet over which the automaton is defined, \(\delta\) is a transition function specifying the automaton’s behavior as it encounters a particular input symbol. If, after starting in \(q_0\), reading an input string and transitioning as specified by \(\delta\), the automaton ends in a state in \(F\), the input string is considered to be part of the language recognized by the automaton.

The difference between deterministic and non-deterministic automata lies solely in the transition function. For deterministic automata, \(\delta\) maps a \(q \in Q, \sigma \in \Sigma\) pair to a state \(q' \in Q\). For non-deterministic automata, \(\delta\) maps the \((q, \sigma)\) pair to a subset of states \(Q' \subseteq Q\); the angelic non-determinism interpretation states that at each point when the NFA could transition to multiple states, it should transition to the state that will eventually allow it to accept the input string.

Those two types of automata are equally expressive (as they accept the same class of languages), and for any non-deterministic automaton, there also exists an equivalent deterministic finite automaton (and vice versa). However, the number of states in the minimal deterministic finite automaton corresponding to an \(n\)-state non-deterministic finite automaton may vary – it may be as low as \(n\), or as many as \(2^n\), depending on the NFA, with the two extreme cases demonstrated in Figure 1.

Figure 1: Top: an \(n\)-state NFA equivalent to an \(n\)-state minimal DFA over a unary alphabet; bottom: an \(n\)-state NFA equivalent to a \(2^n\)-state minimal DFA over a binary alphabet.

It is this flexibility in the number of states of a DFA equivalent to a particular NFA that raises the question of whether, for every pair of integers \((n, d)\) such that \(1 \leq n \leq d \leq 2^n\), there exists a minimal \(n\)-state NFA such that the minimal equivalent DFA has exactly \(d\) states. In this report, unless otherwise stated, the term “\((n, d)\) pair”, or simply \((n, d)\), is used to mean a pair of integers subject
to the above inequality.

The answer depends on the size of the alphabet the NFA is to be defined over; this report demonstrates and proves the correctness of several ways to construct NFAs for a particular \((n, d)\) pair over alphabets of varying size. The implementation component also implements some of the proved constructions using Java, and contains an application that allows NFAs and equivalent DFAs to be constructed for a user-specified \((n, d)\) pair.
2 Implementation

In addition to proving that non-deterministic automata exist for all \((n, d)\) pairs, some of the constructions described in the next chapter are also implemented using Java, allowing the automata constructed for a particular \((n, d)\) pair to be examined easily. This chapter describes the implemented functionality, and serves as a summary of finite automata concepts used in subsequent proofs.

Implementation source code and JAR files are available for download at http://www.student.dtu.dk/~s072442/bach/. In general, the implementation consists of three elements: classes implementing regular automata representation and conversion (the reglang package), classes implementing particular NFA constructions (the generators package), and classes implementing Swing user interfaces for particular applications (the app package).

2.1 Representing Automata

The reglang package contains three Java classes allowing representation of regular languages: NFA, DFA, and RegEx, corresponding to non-deterministic finite automata, deterministic finite automata, and regular expression representations of regular languages. Of the three, only NFA allows creation of mutable objects, which can subsequently be customized and then converted to deterministic finite automaton and regular expression representations. This fits with the general purpose of the implementation: NFA objects will be constructed based on an \((n, d)\) pair, and then potentially converted to DFA or regular expression representations to verify that the minimal equivalent DFA does indeed have the required number of states.

In the automata implementations, each state is assigned an integer index. A BitSet\(^1\) object is used to keep track of which states are accepting. The transition function is stored as a two-dimensional array of either state indices or BitSet objects, allowing each (state, symbol) pair to be associated with either a state or a set of states for deterministic and non-deterministic automata respectively. In addition, NFA objects store the index of their start state, while DFA objects start in state 0 by convention (afforded by the fact they are always constructed from an NFA using the subset construction algorithm).

Regular expressions are represented by RegEx objects, which are used only to provide a textual representation of a regular expression recognizing the language recognized by an NFA. Within the RegEx object, regular expressions are represented as a tree of objects representing operations – concatenation, union, and closure – on other regular expressions, eventually leading to leaf nodes representing a particular input symbol (or an empty string).

\(^1\)BitSet is provided by Java as a way to represent a vector of bits without resorting to an array of booleans.
2.2 Converting Automata

This subsection describes the algorithms used in the implementation to convert a given NFA object to a DFA or RegEx representation.

2.2.1 Subset construction

The introduction presented the angelic non-determinism interpretation of the transition function in non-deterministic automata – the NFA should, at each input symbol, select the transition to a state that will eventually lead it to accept the input string.

One way to implement this would be to simply keep track of all states the NFA could be in after processing a symbol – i.e. allow the NFA to be in multiple states at once. Then, if the NFA is could reach a set of states $Q' \subseteq Q$, and then processes an input symbol $\sigma$, it could reach any state reachable by a $\sigma$ transition from any of the states in $Q'$, i.e. it would transition to a set of states $Q''$:

$$Q'' = \bigcup_{q \in Q'} \delta(q, \sigma)$$

where $Q' \subseteq Q, \sigma \in \Sigma$

If, after processing the input string, the NFA reaches a set of states $Q' \subseteq Q$ that contains at least one accepting state (i.e. $Q' \cap F \neq \emptyset$), the input string is accepted by the NFA, as there exists a series of choices that would lead to that accepting state.

Keeping track of which states the NFA could be in leads to a natural way to construct an equivalent DFA: if $|Q| = n$, there are $2^n$ different subsets of $Q$; we can define a DFA with $2^n$ states such that each state is equivalent to one subset, the initial state is the state equivalent to $\{q_0\}$, and accepting states are those that are equivalent to subsets of NFA states that contain at least one of the NFA’s accepting states. The transition function would be defined as above: each DFA state, equivalent to some subset $Q'$ of NFA states, would transition to the DFA state equivalent to $Q''$. Depending on the NFA’s transition function, however, this method could produce a DFA with several connected components – i.e. states that could never be reached from the DFA’s start state.

To avoid generating unnecessary and unreachable DFA states, the implementation is based on lazy subset construction, which creates DFA states by starting with only a state equivalent to $\{q_0\}$ (the NFA’s initial state), and ensuring that for all states in the DFA, the transitions on all symbols lead to a DFA state equivalent to the correct subset of NFA states. The algorithm used is shown in Figure 2.

2.2.2 DFA minimization

A deterministic finite automaton is minimal if there exist no other deterministic finite automata with fewer states that accept the same language. While lazy subset construction ensures that all DFA states are reachable from the start
procedure LazySubsetConstruction\((Q, \Sigma, \delta, q_0, F)\)

\(\text{equivalent} \leftarrow \text{an empty bidirectional map.}\)

\(\text{equivalent}(\{q_0\}) \leftarrow q'_0, i \leftarrow 0, m \leftarrow 0, \delta' \leftarrow \text{new transition function}\)

\(\textbf{while } i \leq m \textbf{ do}\)

\(Q' \leftarrow \text{equivalent}[q'_i]\)

\(\textbf{for each } \sigma \in \Sigma \textbf{ do}\)

\(Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, \sigma)\)

\(\text{if } \text{equivalent contains no mapping for } Q'' \text{ then}\)

\(m \leftarrow m + 1\)

\(\text{equivalent}[Q''] \leftarrow q'_m\)

\(\delta'(q'_i, \sigma) \leftarrow \text{equivalent}[Q'']\)

\(i \leftarrow i + 1\)

\(F' \leftarrow \{q'_j \mid \text{equivalent}[q'_j] \cap F \neq \emptyset\}\)

\(\text{return } \{q'_0, \ldots, q'_m\}, \Sigma, \delta', q'_0, F'\)

Figure 2: The lazy subset construction algorithm used to generate a DFA that recognizes the same language as a given NFA.

Figure 3: A non-deterministic finite automaton over the alphabet \(\{a\}\) is converted to a 3-state DFA by subset construction; however, a 2-state DFA can recognize the same language. The state labelled 12 in the constructed DFA is equivalent to the NFA being in the set of states \(\{1, 2\}\).

state, it does not necessarily produce a minimal DFA, as the example in Figure 3 demonstrates. As we’re interested in verifying whether the NFA constructed for an \((n, d)\) pair is actually equivalent to a \(d\)-state minimal DFA, we need to be able to produce a minimal DFA from any other DFA.

The key observation is that the DFA accepts the same set of strings from states 1 and 12, allowing those states to be merged into a single state, and yielding the minimal equivalent DFA. We define two DFA states to be distinguishable if there is some string \(w\) that is only accepted from one of those states — i.e. the DFA would transition to an accepting state from one of the two states, and would transition to a non-accepting state from the other; if no such string exists, the two states are indistinguishable and may be replaced by a single state.

The implementation uses the table filling algorithm described in [3, pp. 159-163] to identify which states are distinguishable — essentially, first marking all accepting states to be distinguishable from all non-accepting states (\(w\) being the empty string in this case), and then proceeding to mark any part of states
\((q_1, q_2)\) as distinguishable if on some symbol \(\sigma \in \Sigma \) \(\delta(q_1, \sigma)\) and \(\delta(q_2, \sigma)\) are distinguishable, stopping once no additional states can be marked. This allows the original DFA to be divided into blocks of indistinguishable states (some consisting of a single state, some of several), which can then be replaced by a single state in the minimized DFA. The DFA produced using this method is minimal, as proven in [3].

2.2.3 Regular expressions

The implementation also allows a non-deterministic finite automaton to be converted to a regular expression using the state elimination method described in [3, pp. 98-102]; the method can be summarized as treating the NFA as a transition system, where transitions are annotated with the regular expressions that allow the system to transition from one state to another, and iteratively removing states until only the start state and accepting state remain.

The conversion procedure in our implementation proceeds as follows:

1. A transition system is created with all of the NFA’s states and transitions. The transitions are annotated with the regular expression recognizing the symbol the transition is valid for. If some transition does not exist in this transition system, just consider its annotation to be equal to the regular expression accepting no strings.

2. A new state is added to the transition system, along with transitions to this new state from all of NFA’s accepting states, annotated with the regular expression recognizing only an empty string. This allows the new state to be treated as the only accepting state in the transition system.

3. Then, for each state \(q\) that is not the starting or the new accepting state:
   - Prepend the closure of the \(q \rightarrow q\) transition annotation to every transition from \(q\). Then remove the \(q \rightarrow q\) transition.
   - For each pair of transitions \(q_s \rightarrow q\) and \(q \rightarrow q_t\), replace the annotation on the \(q_s \rightarrow q_t\) transition with the union of the annotation and the concatenation of the annotations on the \(q_s \rightarrow q\) and \(q \rightarrow q_t\) transitions.
   - Remove state \(q\) from the transition system.

4. This leaves us with a transition system that contains only an initial state and the new accepting state; as there were no transitions from the accepting state in the transition system, and none could have been added, the accepting state has no outgoing transitions or self-loops, while the initial state may have a self-loop and a transition to the accepting state.

5. The equivalent regular expression is then the closure of the self-loop annotation on the start state concatenated with the annotation on the transition from the start state to the accepting state.
The regular expression produced for the constructions discussed in this paper are often relatively unwieldy; and, while they could likely be compacted by applying additional analysis, it is unlikely that they’ll be a meaningful expression of the regular language recognized by the automaton. The conversion method described above also works on DFAs; however, as all automata produced by the implementation initially exist in NFA form, it was not deemed necessary to implement a similar conversion from a DFA to a regular expression, partly because the increase in the number of states would likely make the expression even less comprehensible.

2.3 Applications

The representations and conversion algorithms are used to implement the various NFA constructions described in the next chapter. Classes implementing the Generator interface may be asked to generate an NFA for a particular \((n, d)\) pair, which would then be converted to a minimal DFA to verify its validity. The generators are implemented by classes in the generators package, and generally follow the structure of the proofs they are based on.

Additionally, the implementation contains two executable applications: one to visualize the various constructions, and one to count the number of NFAs that satisfy a particular \((n, d)\) pair over the binary alphabet.

The former, implemented in the LangView class, allows the user to specify an \((n, d)\) pair, select a construction type, and view the transition function or a visualization of the NFA and the equivalent DFA. Graphviz\(^2\) is an open source graph visualization application; the implementation runs its executable in a sub-process to generate a PNG image based on the description of the finite automaton being visualized. To view a particular representation of the regular language, the user can select the desired representation and constructions from drop-down boxes, and type in a comma-separated \((n, d)\) pair into the text box, pressing Enter to view the requested representation.

The latter, implemented in the LangCount class, enumerates a particular class of NFAs whose minimality is easy to prove, and tallies how many different NFAs require an equivalent DFA with a particular number of states; exploiting symmetry to slightly reduce the size of the search space.

JAR files for running those applications are included in the submission along with source code.

\(^2\)The Graphviz homepage, containing downloads and descriptions of the program is located at http://www.graphviz.org/.
Figure 4: The LangView application allows the constructions to be visualized; the top image displays the non-deterministic finite automaton visualized using Graphviz, the bottom displays the transition function of the equivalent deterministic finite automaton in a table. In a GraphViz view, a yellow background indicates the automaton’s initial state.
3 Constructions

Past research has shown that it is possible to construct $n$-state non-deterministic automata such that the minimal deterministic automaton equivalent has $1 \leq n \leq d \leq 2^n$ states for a wide variety of alphabet sizes for any pair of integers numbers $(n, d)$ satisfying the above inequality. This chapter demonstrates several of those constructions, providing both a way to construct an appropriate NFA for a given $(n, d)$ pair, and presenting proofs that those constructions do produce minimal NFAs such that the minimal DFAs have the required number of states.

Before discussing the constructions, it is worth discussing how we could prove that the NFAs produced in them are minimal, and point out solutions for a few specific $(n, d)$ pairs that could be used with almost any alphabet.

3.1 Proving that a non-deterministic finite automaton is minimal

Since the question is whether there always exist minimal NFA such that the minimal equivalent DFA has a particular number of states, we need a way to verify whether the NFA specified by a particular construction is minimal. Unfortunately, state equivalence cannot be used to prove that a non-deterministic automaton is minimal (as shown in [3, p. 164]), and the problem of finding a proof that an arbitrary NFA is minimal has been shown to be PSPACE complete in [4].

It is, however, possible to show that the specific NFAs constructed in the proofs are minimal without solving the general problem. For instance, the method below is useful in showing minimality of the first few of the constructions:

**Lemma 1.** A non-deterministic finite automaton with at least $n$ states is required to recognize a non-empty regular language where the shortest string is of length $n - 1$. Equivalently, if the shortest string accepted by an $n$-state NFA is $n - 1$ symbols long, the NFA is minimal.

This can be proven by considering the shortest string $w$ accepted by an NFA. To accept $w$, the NFA must read at least $|w|$ symbols, and, upon reading each symbol, reach a state that has not been reached before (otherwise, it would be possible to remove symbols that do not cause the NFA to enter new states, and still have an accepted string, which contradicts the assumption that $w$ is the shortest such string). As a consequence of this, the NFA must have at least $|w| + 1$ states: a starting state, and $|w|$ additional states to provide a previously unreached state for each of the $|w|$ transitions required to recognize $w$. So to recognize a string of length $n - 1$, an $n$-state NFA is necessary.

While Lemma 1 provides a simple way to prove that a NFA is minimal, it imposes requirements on the structure of the NFA. A more flexible method is
to use the fooling set theorem:

**Lemma 2.** To prove that an $n$-state NFA $M$ is minimal, it is sufficient to show that there exists a “fooling set” of string pairs $A = \{(a_i, b_i) \mid 1 \leq i \leq n\}$ such that such all strings $a_ib_i$ are accepted by $M$, but at least one of the strings $a_ib_j$ or $a_jb_i$, for all pairs of $i, j$ where $i \neq j$ is not accepted.

Let $P_i$ be the set of states $M$ is in after reading $a_i$; as the string $a_ib_i$ is accepted by $M$, there must exist a state $p_i \in P_i$ such that reading $b_i$ while in this state would lead to an accepting state. It remains to show that the states $p_i \neq p_j$ for all $i \neq j$; suppose $p_i = p_j$ and $i \neq j$, then both $a_ib_j$ and $a_jb_i$ are accepted by the automaton, which contradicts the definition of the fooling set.

Therefore, if $A$ is a fooling set for a particular language, an NFA recognizing the language must have at least $|A|$ states. \(\square\)

### 3.2 The two extreme cases

There are relatively straightforward constructions for $(n,n)$ and $(n,2^n)$ cases over alphabets of at least 1 and 2 symbols respectively.

For $d = n$, it is sufficient to consider the language of all strings at least $n - 1$ characters long. Here, both the NFA and the DFA have exactly $n$ states, and are shown in Figure 5. It is worth pointing out that while the automaton in the figure is defined over a 1-letter alphabet, the same construction can be used over any alphabet size simply by defining $\delta(q, \sigma) = \delta(q, a)$ for every additional $\sigma$ symbol introduced. The NFA is minimal per Lemma 1; minimality of the DFA is trivial to verify as all of the states are distinguishable by the number of $a$ symbols they have to process before reaching an accepting state.

![Figure 5: This $n$-state automaton can be interpreted as both an NFA and a DFA.](image)

For $d = 2^n$, consider the language of all strings at least $n - 1$ characters long that do not contain a character $b$ more than $n - 1$ times in a row, and, if a string is $n$ characters or longer, the $n$'th character from the end is an $a$. It is recognized by the NFA shown in Figure 6 below. The NFA is minimal per Lemma 1.

To prove that the minimal equivalent DFA has $2^n$ states, observe that each state $q_i$ of the NFA has a unique string, $b^{n-i}$, that is only accepted if the automaton is in that state – because of this, any subset of NFA states is distinguishable from any other subset. This means it is sufficient to show that any subset of the NFA states is reachable.

The empty set of states is reachable by reading $b^n$; for any non-empty subset of states $Q' \subseteq \{q_1, \ldots, q_n\}$, there exists a string $s = \sigma_1 \ldots \sigma_{m-1}$ where $m$ is the greatest index of the state in $Q$ (i.e. $q_m \in Q' \land \forall i : i > m \Rightarrow q_i \notin Q'$), and
Figure 6: This $n$-state non-deterministic automaton has no equivalent deterministic automaton with fewer than $2^n$ states.

$$\sigma_i = a \text{ if } q_{m-i} \in Q' \text{ or } b \text{ otherwise.}$$

Intuitively, the string $s$ can be arrived at by a simple observation: each input symbol shifts all states the automaton was in by one to the right, while $a$ also returns it to $q_1$; thus, if the automaton must be in $q_i$ after processing $s$, then it must’ve been in $q_1 \ i − 1$ input symbols ago, so either $|s| = i − 1$ (the automaton was in $q_1$ because it is the initial state), or the $i$‘th character from the end of $s$ was an $a$ (and returned the automaton to $q_1$).

The construction shown in Figure 6 can thus be used to produce $n$-state NFAs that are equivalent to $2^n$-state DFAs over any alphabet with two or more symbols without adding additional transitions – it is perfectly okay to have the automaton transition to $\emptyset$ when encountering “new” symbols, as $\emptyset$ is already reachable by reading $b^n$.

### 3.3 Exponential alphabet size

It is relatively easy to construct an $n$-state NFA such that the corresponding minimal DFA would have $n \leq \alpha \leq 2^n$ states by adjusting the size of the alphabet $\Sigma$ depending on $\alpha$. We’ve previously demonstrated an $n$-state NFA that is equivalent to an $n$-state minimal DFA, so the following proof concerns itself only with $n < \alpha \leq 2^n$.

To begin with, construct a skeleton for the $n$-state NFA; labeling the states $Q = \{q_1, \ldots, q_n\}$, where $q_1$ is the initial state, $q_n$ is the only accepting state, and a partial definition of the transition function is as below. The skeleton is also shown in Figure 7.

$$\delta(q_i, \sigma_i) = \{q_{i+1}\} \text{ for } i < n, q_i \in \Sigma$$

$$\delta(q_i, \sigma_j) = \emptyset \text{ for } i < n, j \neq i, \sigma_j \in \Sigma$$

$$\delta(q_j, \sigma_i) = \emptyset \text{ for } i < n, j \neq i$$

The minimality of any NFA constructed using this skeleton can be proven using Lemma 1: the minimal accepted string, $\sigma_1 \sigma_2 \ldots \sigma_{n-1}$, is of length $n − 1$, so any NFA recognizing this language must have at least $n$ states, and therefore any NFA based on this skeleton is minimal. Another interesting property is that each state $q_i$ has a unique string, $\sigma_i \sigma_{i+1} \ldots \sigma_{n-1}$, that is accepted only from that state. This leads to the observation that for any two subsets $Q_i, Q_j \subseteq Q, Q_i \neq Q_j$.
Figure 7: This NFA skeleton can be used as base to construct an NFA such that the equivalent DFA would require \( n < d \leq 2^n \) states, simply by adding transitions on additional symbols from the final state \( q_n \).

If we were to use subset construction to construct a DFA based on the skeleton NFA (and thus have a DFA where each state is equivalent to some subset of states of the NFA), it would have \( n + 1 \) states: all of the states equivalent to single-state sets containing the NFA states \( q_i \in Q \), and a state equivalent to \( \emptyset \). When \( \alpha > n + 1 \), it is possible to add additional transitions to the NFA to add additional states to the DFA. Suppose that \( Q' \subseteq Q \) is a subset of NFA states that is currently not equivalent to any state in the DFA, and \( \sigma' \in \Sigma \) is a symbol that occurs nowhere else in the NFA; we can then add transitions so that \( \delta(q_n, \sigma') = Q' \). This adds a state equivalent to \( Q' \) to the DFA constructed by subset construction, as \( Q' \) is now reachable in the NFA. No additional states are added to the DFA as a result of making \( Q' \) reachable – as each symbol is only used in a transition from a single state, any states reachable by reading the symbol while in \( Q' \) are already in the DFA.

There’s a total of \( 2^n \) subsets of the \( n \)-state set \( Q \); therefore, we have enough subsets of states to create an \( \alpha \)-state DFA for any \( n < \alpha \leq 2^n \).

### 3.4 Linear alphabet size

While the previous section shows a construction that scales with the number of desired DFA states (and therefore scales exponentially with the number of NFA states), it is possible to show an equivalent result using smaller alphabets. The construction described in this section uses an alphabet with at most \( 2^n \) symbols: a linear increase in alphabet size compared to the number of NFA states rather than an exponential one.

The construction and proof presented here is based on the operations used to prove that such NFAs must exist in [1]; the structure of the proof and that of the constructed automata has been altered for the purposes of providing a simpler and more concise presentation.

**Induction hypothesis.** For all pairs of integers \((n, d)\) such that \( 1 \leq n < d \leq 2^n \), there exists a minimal \( n \)-state non-deterministic finite automaton over a \( 2^n \)-letter alphabet \( \{a_1, \ldots, a_n, b_1, \ldots, b_n\} \), such that the minimal equivalent deterministic finite automaton requires \( d \) states, and such that the shortest
string accepted by either automaton is of length \( n - 1 \).

**Base case.** The only integer pair allowed for \( n = 1 \) is \((1, 2)\); we can show that the induction hypothesis holds by considering automata that recognize a language consisting solely of the empty string: a single accepting NFA state \( q_1 \) (with no transitions), or a two-state DFA (starting state is accepting, transitions on \( a_1, b_1 \) from both states lead to the non-accepting state) are both minimal and recognize this language. We note that both automatons accept the empty string so the shortest accepted string is of length 0 = \( n - 1 \).

**Induction step.** Assuming the induction hypothesis is satisfied for all integer pairs \((n, d)\), \( n < d \leq 2^n \), it remains to show that automata pairs exist for all integer pairs \((n + 1, \alpha)\) satisfying \( n + 1 < \alpha \leq 2^{n+1} \).

We apply the following modifications to any of the NFA constructed for \((n, d)\): add a new starting state \( q_{n+1} \), add a transition \( \delta(q_{n+1}, a_{n+1}) = \{q_n\} \), and do one of the following:

1. Add no additional transitions.
2. Add transitions \((q_i, b_{n+1}) \rightarrow \{q_i, q_{n+1}\}\) for all \( i = 1, \ldots, n \).
3. Add transition \((q_n, b_{n+1}) \rightarrow \{q_n\}\), and \((q_i, b_{n+1}) \rightarrow \{q_i, q_{n+1}\}\) for all \( i = 1, \ldots, n - 1 \).

The shortest string accepted by the \((n + 1, \alpha)\) NFA is \( a_{n+1} \) followed by the \( n - 1 \) character shortest string accepted by the \((n, d)\) NFA, yielding a shortest accepted string of length \( n - 1 + 1 = (n + 1) - 1 \), satisfying a part of the induction hypothesis. This also proves that the \((n + 1, \alpha)\) NFA is minimal, as \( n + 1 \) NFA states are required to recognize a language with a minimum string length of \( n \) per Lemma 1.

Observe the effect of the three options on the number of reachable subsets of states of the new NFA; noting that all of the reachable subsets of the original \( n \)-state NFA remain reachable and distinguishable. Denote the set of reachable subsets of the \( n \)-state NFA \( R \), \( |R| = d \).

1. Adds a single reachable subset \( \{q_{n+1}\} \). \( \alpha = d + 1 \) if we choose this modification.
2. Adds reachable subsets \( \{q_{n+1}\} \cup r, r \in R \); so \( \alpha = 2d \).
3. Adds reachable subsets \( \{q_{n+1}\} \cup r, r \in R \land r \neq \{q_n\} \); so \( \alpha = 2d - 1 \).

Subsets containing \( q_{n+1} \) are distinguishable from all the previously-existing subsets, as the presence of \( q_{n+1} \) allows them to accept strings starting with the \( a_{n+1} \) symbol. Additionally, those subsets are also distinguishable from each other: if \( r' \) is a subset of reachable states of the \( n+1 \)-state NFA, then \( r' \setminus \{q_{n+1}\} = r \in R \), and all of the reachable subsets of the \( n \)-state NFA are distinguishable from each other.

The modifications above allow construction of NFAs satisfying \((n + 1, \alpha)\), where \( \alpha \in \{d + 1, 2d, 2d - 1\}, n < d \leq 2^n \). It remains to show that by choosing proper combinations of \((n, d)\) NFAs and modifications, we can construct a NFA for every \((n + 1, \alpha)\) pair satisfying \( n + 1 < \alpha \leq 2^{n+1} \).
Using the first modification on all possible choices of \((n,d)\) NFAs allows construction of \((n+1,\alpha)\) NFAs where \(n+1 < \alpha \leq 2^n + 1\). The second modification constructs all NFAs such that \(2n < \alpha \leq 2^{n+1}\) where \(\alpha\) is even. The third modification constructs all NFAs such that \(2n - 1 < \alpha \leq 2^{n+1} - 1\), where \(\alpha\) is odd. The last two modifications cover all values of \(\alpha\) such that \(2n < \alpha \leq 2^{n+1}\); which, combined with the first modification, yields \(n+1 < \alpha \leq 2^{n+1}\), confirming the induction hypothesis.

**Conclusion.** This proves that minimal \(n\)-state NFAs can be constructed over a \(2^n\)-letter alphabet such that the minimal equivalent DFA requires \(n < d \leq 2^n\) states. As a way to construct the \((n,n)\) NFA over a unary alphabet has been demonstrated previously, the statement can be expended to \(n \leq d \leq 2^n\).

The construction method for any \((n,\alpha)\) pair can be summarized as follows:

1. If \(\alpha = n\), the NFA should accept all strings of at least \(n-1\) symbols.
2. If \(\alpha \leq 2^{n-1} + 1\), use the first modification on the \((n-1,\alpha-1)\) NFA.
3. If \(\alpha\) is even, use the second modification on the \((n-1,\alpha/2)\) NFA.
4. If \(\alpha\) is odd, use the third modification on the \((n-1, (\alpha+1)/2)\) NFA.

### 3.5 4-letter alphabet

The previous sections have proven that there exist ways to construct an NFA for any \((n,d)\) pair over an alphabet that is allowed to scale at least linearly with the number of NFA states. This section proves that the same result holds even with a fixed-size alphabet, by focusing on the four-letter alphabet construction presented in [2]; the presentation of the proof below differs from the original paper.

The proof can be outlined as follows: we first construct a \(k\)-state NFA that is equivalent to a \(2^k\)-state minimal DFA, then add one additional NFA state to the \(k\)-state NFA to construct \(k+1\)-state NFAs equivalent to \(d\)-state minimal DFAs where \(2^k < d < 2^{k+1}\), and finally use prefixes to construct NFAs such that \(n < d < 2^{n-1} + 1\).

**Constructing a \((k,2^k)\) NFA**

To begin with, we want to construct a \(k\)-state NFA that would be equivalent to a \(2^k\)-state DFA. The structure of the \(k\)-state NFA is shown in Figure 8; it consists of the set of states \(Q = \{q_1, \ldots, q_k\}\) where \(q_1\) is the start state, \(q_k\) is the only accepting state, and the transition function \(\delta\) is described in the figure.

The structure of this NFA is similar to that used for the \((n,2^n)\) case over the binary alphabet: here, the \(c\) transitions can be used to demonstrate that all states have a unique string that is only accepted from that state; and \(a\) and \(b\) transitions serve the same purpose as before – \(a\) returns the automaton to \(q_1\), \(b\) does not, while both advance it one state to the right. The major modification is the addition of additional transitions from the \(q_k\) state; those do not, however, affect the proofs of minimality of the NFA (still minimal per Lemma 1) or the
DFA (all subsets are still reachable and distinguishable: $\emptyset$ by $c^k$, any other subset by the same $ab$-string as before).

Thus, as for the $(n, 2^n)$ binary-alphabet NFA, this NFA generates $2^k$ states, and all subsets of states are reachable in the equivalent DFA, with $\emptyset$ reached by $c^k$ and any other subset by a specific $ab$-string.

One additional state

The next step of the construction is to use one additional state, $q_0$, to allow construction of NFAs equivalent to $d$-state minimal DFAs where $2^k < d < 2^{k+1}$. This state is added as shown in Figure 9 below, with additional transitions to be added to make it reachable depending on the value of $d$.

Figure 9: Additional DFA states can be added to the DFA equivalent to this automaton by adding transitions on $d$ from the states $q_1, \ldots, q_k$ leading to some subsets of states containing $q_0$.

To find out which additional transitions should be added, let’s consider a related automaton where $q_0$ is the starting state instead of $q_1$, and the alphabet has been restricted to $\{a, b\}$: as all subsets of $\{q_1, \ldots, q_k\}$ are already reachable from $q_1$, we want to consider what other subsets – containing $q_0$ can be made reachable, and, as $c$-transitions lead away from $q_0$, we can safely ignore those, except in their function to ensure that all subsets of states are ultimately
distinguishable from each other.

\[ \text{Figure 10: Moving the initial state and removing } c\text{-transitions yields this automaton.} \]

The equivalent DFA is shown in Figure 11 for \( k = 4 \). Recalling that all subsets of \( \{q_1, \ldots, q_k\} \) are reachable from \( \{q_1\} \) using strings only containing \( a \) and \( b \) symbols, and noting that both \( \{q_0\} \) and \( \{q_0, q_1\} \) subsets are reachable (\( q_0 \) is the initial state, the latter subset is the result of an \( a \) transition from the initial state), we can conclude that the equivalent DFA has \( 2^k \) states, all of which are equivalent to a different subset of NFA states containing \( q_0 \).

\[ \text{Figure 11: DFA equivalent to the NFA in Figure 10, restricted to the alphabet } \{a, b\} \text{ for } k = 4. \text{ The state labels represent the set of NFA states equivalent to that DFA state; thus state 012 is equivalent to the set of NFA states } \{q_0, q_1, q_2\}. \]

**Lemma 3.** If we consider states containing \( q_k \) to be “leaf” states (ignoring transitions to \( \{q_0, \ldots, q_k\} \) from those states), the equivalent DFA’s structure is a complete binary tree of height \( k - 1 \) rooted at \( \{q_0, q_1\} \), with \( 2^k - 1 \) states in total.
Observe that the state \( q_k \) will be reached from the state \( q_1 \) after \( k - 1 \) input symbols are processed, regardless of whether those symbols are \( a \) or \( b \). Once the automaton reaches a state containing \( q_k \), all transitions on \( a \) and \( b \) lead to \( \{q_0, \ldots, q_k\} \), which is itself reachable by \( a^{k-1} \). As the DFA has \( 2^k \) states in total, that means that \( 2^k - 1 \) states (excluding \( \{q_0\} \), which is the initial state) must be reachable by a string of length at most \( k - 1 \) from \( \{q_0, q_1\} \): there are \( 2^{k-1} + 2^{k-2} + \ldots + 2^0 = 2^k - 1 \) such strings, so each string must lead to a unique state if all states are to be reached. As there are only two transitions leading away from any particular state, this means that from \( \{q_0, q_1\} \), the equivalent DFA is structured like a complete binary tree of height \( k - 1 \).

**Lemma 4.** It is possible to select at most \( h + 1 \) disjoint subtrees from a complete binary tree of height \( h \) such that the selection contains \( 1 \leq m < 2^{h+1} \) nodes.

For \( h = 0, 1 \leq m < 2 \), which is fulfilled by selecting the whole tree as a subtree of itself. Then, treating the Lemma as an induction hypothesis, and assuming that it holds for trees of height \( h \), let’s consider trees of height \( h + 1 \), where \( 1 \leq m < 2^{h+1} \):

- If \( m = 2^{h+2} - 1 \), we select the root of the height \( h + 1 \) tree.
- If \( m < 2^{h+1} \), use the selection the induction hypothesis states exists for trees of height \( h \) from the subtree rooted at one of the root node’s children.
- If \( 2^{h+1} \leq m < 2^{h+2} - 1 \), we select one of the children of the root node, gaining \( 2^{h+1} - 1 \) states, and get the rest from the other child: the height \( h \) tree rooted at the child has to provide \( 2^{h+1} - (2^{h+1} - 1) \leq m < 2^{h+2} - 1 - (2^{h+1} - 1) \iff 1 \leq m < 2^{h+1} \) nodes by selecting at most \( h + 1 + 1 - 1 = h + 1 \) subtrees, and *that* selection exists per the induction hypothesis.

Therefore, the Lemma holds for trees of all heights \( h \geq 0 \).

Because the complete binary tree-like structure in the equivalent DFA is not a proper complete binary tree – all the leaf nodes transition to \( \{q_0, \ldots, q_k\} \) – Lemma 4 needs to be refined slightly before it can be applied to the equivalent DFA. The refinements apply to the last two cases of the induction proof: if \( m < 2^{h+1} \), we want the selection of sub-trees to continue with the subtree rooted at the state led to by an \( a \)-transition and if \( 2^{h+1} \leq m < 2^{h+2} - 1 \), we want to select the sub-tree rooted at the state led to by a \( b \)-transition, and continue with the subtree rooted at the state led to by an \( a \)-transition. Observe that the last subtree selected will be reachable by a \( a^i \) string (as we begin at a state reachable by an \( a^0 \) string, and always follow the \( a \) transitions down the tree), and all preceding subtrees will be reachable by \( a^i b \) strings. This ensures that exactly one subtree rooted at a \( \{q_0, q_1, q_2, \ldots, q_r\} \) state is selected, and all the other subtrees rooted at a \( \{q_0, q_2, q_3, \ldots, q_s\} \)-type state, which allows us to count the \( \{q_0, q_1, \ldots, q_k\} \) node as part of the former subtree, and not part of the latter subtrees as far the number of states added to the DFA is concerned.

As a consequence of the Lemma 4, any integer \( 1 \leq m < 2^k \) can be written
as
\[ m = (2^{k_1} - 1) + (2^{k_2} - 1) + \ldots + (2^{k_{l-1}} - 1) + (2^{k_l} - 1) \]
where \( 1 \leq l \leq k \) and \( k \geq k_1 > k_2 > \ldots > k_{l-1} \geq k_l \geq 1 \); simply consider the individual parenthesis terms as the number of nodes contributed by selected sub-trees.

Thus, for a given \((k+1, \alpha)\) pair, where \(2^k < \alpha < 2^{k+1}\), we compute an \( m = \alpha - 2^k \), express it as
\[ m = (2^{k_1} - 1) + (2^{k_2} - 1) + \ldots + (2^{k_l} - 1) \]
where \( 1 \leq l \leq k \) and \( k \geq k_1 > k_2 > \ldots > k_{l-1} \geq k_l \geq 1 \), and add \( d \) transitions to the NFA of Figure 9 as follows:
\[
\delta(q_i, d) = \begin{cases} 
\{q_0, q_2, q_3, \ldots, q_{k-k_i+1}\} & \text{for } 1 \leq i < l \\
\{q_0, q_1, q_2, \ldots, q_{k-k_i+1}\} & \text{for } i = l \\
\emptyset & \text{otherwise}
\end{cases}
\]
Note that the sets of states transitioned to on \( d \) symbols are subsets of each other – for any subset of states \( Q' \) containing multiple states with non-empty \( \delta(q, d) \) sets, the union of those sets will always be equal to the value of \( \delta(q_j, d) \) where \( j \) is the maximal index of a state in \( Q' \) such that \( j \leq l \). Thus, no additional states are introduced by the possibility of the NFA taking \( d \)-transitions from multiple states at the same time. This modification thus adds \( m \) additional states to the equivalent DFA, which allows construction of \((n, d)\) NFAs such that \(2^{n-1} < d < 2^n\).

Adding a prefix
The preceding sections describe how to construct automata for \((n, d)\) pairs where \(2^{n-1} < d < 2^n\), by selecting \( k = n-1 \). As we’re also interested in \( n < d \leq 2^{n-1} \), the question of what to do with the \( n - k - 1 \) “unused” states when \( k < n - 1 \).

Those states can be used to form a prefix to the \( k + 1 \)-state automaton, as shown in Figure 12. Each state in the prefix will contribute a single additional state to the DFA (namely, the state equivalent to the NFA being only in that prefix state) – and, as \( \{q_1\} \) is still reachable from the start state by reading \( b^{n-k-1} \), the \( 2^k < d < 2^{k+1} \) DFA states generated by the \( k + 1 \)-state NFA are still reachable.

Thus, the prefix method can be used to construct an \( n \)-state NFA such that the equivalent DFA has \( 2^k + (n - k - 1) < d < 2^{k+1} + (n - k - 1) \) states. Note what happens to this inequality as \( k \) is ranges from \( n - 1 \) to 1:
\[
\begin{align*}
k = n - 1 & \quad \Rightarrow \quad 2^{n-1} + 1 \leq d < 2^n \\
k = n - 2 & \quad \Rightarrow \quad 2^{n-2} + 2 \leq d < 2^{n-1} + 1 \\
k = n - 3 & \quad \Rightarrow \quad 2^{n-3} + 3 \leq d < 2^{n-2} + 2 \\
\ldots & \\
k = 1 & \quad \Rightarrow \quad n + 1 \leq d < n + 2
\end{align*}
\]
Figure 12: The $n-k-1$ states can be used to form a prefix to the $k+1$ state NFA.

The ranges are adjacent, which allows us to conclude that by selecting an appropriate $k$ value, we can construct an $n$-state NFA such that the equivalent DFA has $n < d < 2^n$ states.

Recap and minimality

We have thus arrived at a method to construct an $n$-state NFA for an $(n,d)$ pair where $n < d < 2^n$, which can be summarized as follows:

1. The NFA consists of states $q_0, q_1, \ldots, q_{n-1}$.
2. Select a $k$ such that $2^k + (n - k - 1) < d < 2^{k+1} + (n - k - 1)$.
3. If $k = n - 1$, $q_1$ is the starting state; otherwise, $q_{n-1}$ is the starting state.
4. The only accepting state is $q_k$.
5. Let $m = d - 2^k - (n - k - 1)$. Then, $m$ can be written as
   
   $m = (2^{k_1} - 1) + (2^{k_2} - 1) + \ldots + (2^{k_{l-1}} - 1) + (2^{k_l} - 1)$
   
   where $1 \leq l \leq k$ and $k \geq k_1 > k_2 > \ldots > k_{l-1} \geq k_l \geq 1$
6. The transitions are defined as follows:

\[
\begin{align*}
\delta(q_i, a) & = \\
& \begin{cases} 
\{q_0, q_1\} & \text{for } i = 0 \\
\{q_1, q_{i+1}\} & \text{for } 0 < i < k \\
\{q_1, q_2, \ldots, q_k\} & \text{for } i = k \\
\emptyset & \text{otherwise}
\end{cases} \\
\delta(q_i, b) & = \\
& \begin{cases} 
\{q_0\} & \text{for } i = 0 \\
\{q_{i+1}\} & \text{for } 0 < i < k \\
\{q_1, q_2, \ldots, q_k\} & \text{for } i = k \\
\{q_1\} & \text{for } i = k + 1 \\
\{q_{i-1}\} & \text{for } i > k + 1
\end{cases} \\
\delta(q_i, c) & = \\
& \begin{cases} 
\{q_{i+1}\} & \text{for } i < k \\
\emptyset & \text{otherwise}
\end{cases} \\
\delta(q_i, d) & = \\
& \begin{cases} 
\{q_0, q_2, q_3, \ldots, q_{k-k_i+1}\} & \text{for } 1 \leq i < l \\
\{q_0, q_1, q_2, \ldots, q_{k-k_i+1}\} & \text{for } i = l \\
\emptyset & \text{otherwise}
\end{cases}
\end{align*}
\]

The equivalent DFA would then have \((n - k - 1)\) states from the \(b\)-transition prefix, \(2^k\) states equivalent to all the subsets of \(\{q_1, \ldots, q_k\}\), and between 1 and \(2^k - 1\) states that are subsets of \(\{q_0, \ldots, q_k\}\) that always contain \(q_0\) made reachable by the \(d\)-transitions.

It remains to show several things: that all of the DFA states mentioned above are distinguishable from each other, and that the NFA specified by this construction is minimal itself.

To prove the former, recall that all of the states \(q_i, 0 \leq i \leq k\) have a unique string \(c^{k-i}\) that is accepted only from that state, thus any subset of only those states is distinguishable from any other subset of only those states. The prefix states are distinguishable from non-prefix states (they do not accepting any \(c^x\) strings), and from each other, as each does not accept strings \(b^x c^{k-1}\) where \(x < k - i\), so for any pair of prefix states, there exists a string that is accepted by one but not the other. Thus, all of the states made reachable in the equivalent DFA are distinguishable, the the equivalent minimal DFA does indeed have \(n < d < 2^n\) states.

To prove that the NFA itself is minimal, we use the fooling set method from Lemma 2, with the following pairs of strings:

1. \((b^{n-k-1}c^i, c^{k-1-i})\) for \(0 \leq i < k\), and \((b^{n-k-1}d, c^k)\); call this the first group of string pairs
2. \((b^{-i}, b^{n-k-i}c^{k-1})\) for \(1 \leq i < n - k\); call this the second group of string pairs

Those string pairs form an \(n\)-pair fooling set for the language recognized by the automaton, as:

23
1. For any string pair \( (a_i, b_j) \), the string \( a_i b_j \) is accepted by the automaton. In our case, those strings are \( b^{n-k-1} c_k \) and \( b^{n-k-1} d c_k \), which are accepted by the automaton.

2. For any two string pairs \( (a_i, b_i) \) and \( (a_j, b_j) \), \( i \neq j \), at least one of the strings \( a_i b_j \) and \( a_j b_i \) is not accepted by the automaton.

The second point is proved by considering the various ways to select a pair of different string pairs \( (a_i, b_i) \) and \( (a_j, b_j) \) from the fooling set:

- If the pairs come from different groups, combining the prefix from the second group with the suffix from the first will yield a string \( b^x c^y \) where \( x < n - k - 1 \), causing the automaton to reject the string.
- If both pairs come from the first group, combining the longer \( c^x \) suffix with the other pair’s prefix will yield a string \( b^{n-k} c^i c^x \), where \( i + x > k - 1 \), causing the automaton to reject the string.
- If both pairs come from the second group, combining the shorter \( b^x \) prefix with the other pair’s suffix will yield a string \( b^y c^{k-1} \), where \( y < n - k - 1 \), causing the automaton to reject the string.

Thus, for all combinations of different string pairs in the set, at least one of the strings \( a_i b_j \) and \( a_j b_i \) is not accepted by the automaton; so the set is an \( n \)-element fooling set for the automaton, which proves that the NFA is minimal per Lemma 2.

Combining the minimality of the NFA with the minimality of the equivalent DFA, we can conclude that this construction is usable for any \( (n, d) \) pair such that \( n < d < 2^n \). Combined with the previous constructions for \( d = n \) and \( d = 2^n \), we can conclude that it is possible to construct an \( n \)-state NFA over a 4-letter alphabet such that the equivalent DFA has \( n \leq d \leq 2^n \) states.

### 3.6 On smaller alphabets

For the three-letter alphabet, [5] has proven that NFAs exist for any \( (n, d) \) pair, and demonstrated a construction similar to that used for the 4-letter alphabet.

The problem for the binary alphabet, the question of whether NFAs exist for any \( (n, d) \) pair is still unanswered. One of the programs included in the implementation part of this project counts the number of \( n \)-state NFAs (limited to those able to be proved minimal by Lemma 1) that are equivalent to minimal \( d \)-state DFAs; because this requires construction of an exponential number of NFAs, the approach quickly becomes computationally infeasible, but can be used to demonstrate that for all \( (n, d) \) pairs for \( n < 5 \), the desired NFAs exist, and generally seem to increase in number with \( n \).

In the case of a single-letter alphabet, [6] has shown that that \( O(e^{\sqrt{n \log n}}) \) DFA states are sufficient to simulate an \( n \)-state NFA, and thus no NFAs can be constructed for some \( (n, d) \) pairs.
4 Conclusion

Through the course of this report, we have shown how to construct an \( n \)-state minimal non-deterministic finite automaton that is equivalent to a \( n \leq d \leq 2^n \) non-deterministic automaton for three different classes alphabet sizes: an alphabet scaling exponentially with \( n \), an alphabet scaling linearly with \( n \), and a fixed, 4-symbol alphabet. Those constructions have also been proven to have the desired properties.

The implementation component of this project implements the linear and fixed-alphabet constructions using Java, allowing the non-deterministic automaton for a \((n,d)\) pair to be constructed, converted to a deterministic automaton using subset construction and elimination of equivalent states, or converted to a regular expression. An application allowing the user to select the \((n,d)\) pair, construction method, and view the non-deterministic and deterministic finite automata (presented graphically if Graphviz is available to provide visualization; or as a transition table if it is not), or regular expression recognizing the same language.

Additionally, the report provided an brief overview of the results for smaller alphabets than those considered in its proofs: a construction is known for the ternary alphabet, the question remains open for the binary alphabet, and it has been shown that no NFAs can exist for some \((n,d)\) pairs over the unary alphabet. A smaller component of the implementation enumerates NFAs over the binary alphabet the minimality of which can be proven by considering the length of the shortest accepting string, and has shown that NFAs exist for all pairs \((n,d)\) \( n < 5 \) before the computational power required becomes prohibitive.
5 References


