Follow Automata Paper Analysis and Implementation

Bucknell Computer Science Technical Report #12-1

Elaina Miller	Alex Ororbia	Bonnie Reiff
B.S. Computer Science	Computer Science & Engineering	B.S. Computer Science
Class of 2013	Class of 2013	Class of 2013
ekm008@bucknell.edu	ago004@bucknell.edu	bpr005@bucknell.edu

December 20, 2012

1 Presentation of the Team and Project

The members of this team are Elaina Miller, Alex Ororbia, and Bonnie Reiff. Our project will be based on the paper, *Follow Automata*, by Lucian Ilie and Sheng Yu.

1.1 Programming Language Chosen

The team has chosen to use Python for the implementation of the result of the paper. No team member has extensive experience in this language, but we are all willing to make an effort to learn.

2 First Reading of the Paper

2.1 What the Team Understood

There are a number of things that the team understood well after an initial reading of the paper:

- All algorithms
- Most introductory definitions including quotient set
- Concept of reduced regular expressions
- Diagrams
- ε-elimination and ε-transitions
- Position automaton
- Partial derivative automaton
- remove( ) and avoid( )

2.2 What the Team Did Not Understand

After going through what we could of the paper, there are still some aspects we need to clarify:

- Definition of right invariant
- Definition of quotient automaton
- Complexity analysis

We also struggled with the logic of the proofs of many of the theorems and lemmas. Our hope is that with more understanding of the topics noted above and with a second read of the paper, the reasoning will become clearer.

2.3 What the Team Liked about the Paper

The flow of the paper makes it fairly easy to follow. For example, the authors made sure to define important terms and complete the proofs necessary to understand information in subsequent sections. They also made the purpose of their paper very clear. The examples that they provide serve two beneficial purposes: Many of the examples are clear and provide for a better and easier understanding of the material. The examples that are not clear point out to us what we need to examine further in the future. The inclusion of the graphs and pseuocode to better illustrate the presented algorithms were also extremely helpful.

3 Detailed Analysis of the Paper

3.1 General Summary of the Article

In this article, the authors present two new algorithms that contribute to the results. The first algorithm constructs εNFAs which are smaller than all other εNFAs constructed by similar algorithms. The paper then demonstrates that the size is extremely close to the provable optimum. The second algorithm builds off of the first by creating NFAs using ε-elimination on the εNFAs produced by the first algorithm, ultimately producing a *follow* automaton. This follow automaton is the simplest of all of the possible automata produced by similar algorithms and is faster to construct than the others.

In addition presenting the algorithm for reduced regular expressions and their two new algorithms, the authors also included proofs of theorems, propositions, and lemmas fundamental to the paper content. Throughout the paper, there were also many examples and diagrams to help the readers understand difficult concepts.

3.2 Summary of the Content by Section

3.2.1 Introduction

The introduction of this paper expands on the information given in the abstract and explains the structure of the paper. Some terminology is introduced, but these terms are not defined until the second section. It gives the reader an idea of what is to come and what terms are essential to understanding the paper. It also mentions that their εNFA construct will specifically by compared against those of Thompson and Sippu and Soisalon-Soininen.

3.2.2 Regular expressions and automata

This section defines and gives the notations for two terms that are essential to the understanding of the rest of the paper: regular expressions and automata. For regular expressions, the size and the set of operations that are allowed to be performed on regular expressions are given. For the automata, each element in the quintuple is defined and explained.

This section also defines an equivalence relation and what it means for the relation to be right invariant.

3.2.3 Reduced regular expressions

In this section, the authors define an algorithm to form reduced regular expressions via $\emptyset$-reduction, ε-reduction, and '$*$'-reduction. The goal of a reduced regular expression is to decrease the total size of the expression without changing the language generated by it. After the algorithm, the authors also present propositions regarding the size of a reduced regular expression, α, with respect to the number of occurrences in α of letters $a \in A - \{\emptyset, \varepsilon\}$.

3.2.4 Small εNFAs from regular expressions

Here, the authors introduce their algorithm for generation of εNFAs from regular expressions, called the follow εNFA and denoted $\mathbf{A}_{\mathrm{f}}^{\varepsilon}$. The construction of the εNFA is presented via Figure 1, with further improvements at each step listed as well. An example is shown in Figure 2 using a regular expression generated by the authors of the paper. The regular expression is used in all of the subsequent examples so that the reader can see how the same regular expression can appear differently based on the rules used to generate an automaton and can see how the algorithm handles a regular expression from start to finish. It is shown that the language generated by the follow εNFA on regular expression α is equivalent to the language generated by α itself. We note that the majority of the proofs concerning this algorithm only address the core construction (i.e. they ignore the further improvements for simplification). The authors then present a comparison of this new construction to Thompson's and Sippu and Soisalon-Soininen's algorithms and prove that this construction is the smallest.

The second half of this section concerns a proof of the upper bound on the size of the εNFA. For this, the authors define the terms $*$-*avoidable* and $*$-*unavoidable*, which allow for the construction of avoid(α) and

remove(β) (where α and β are regular expressions), two functions created with the goal of removing stars from the expression such that the language of α remains the same, but the size of the expression decreases. The end result is that they are able to show that for any reduced regular expression α, $|\mathbf{A}_{\mathrm{f}}^{\varepsilon}(\alpha)| \leqslant \frac{3}{2}|\alpha| + \frac{5}{2}$.

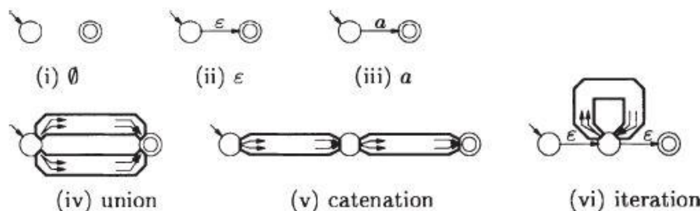

Figure 1: The general construction of $\mathbf{A}_{\mathrm{f}}^{\varepsilon}$.

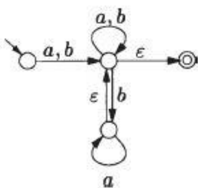

Figure 2: $\mathbf{A}_{\mathrm{f}}^{\varepsilon}(\alpha)$ for $\alpha = (a + b)(a^* + ba^* + b^*)^*$.

3.2.5 Positions and partial derivatives

This section reviews two well-known constructions of NFAs from regular expressions: the *position automaton* and the *partial derivative automaton.*

For the position automaton, pos(α), first(α), last(α), follow(α, i), $\overline{\alpha}$, and $\overline{A}$ are defined. Two other important definitions given are that of the position automaton itself – $\mathbf{A}_{\mathrm{pos}}(\alpha)$, and the transition function – $\delta_{\mathrm{pos}} = \{(i, a, j) \mid j \in \text{follow}(\alpha,\ i),\ a = \overline{a_j}\}$. These definitions are then used throughout the remainder of the paper and are crucial to the understanding of it. Note that to relate this paper to what we have learned in class, the position automaton is derived from Thompson's algorithm combined with ε-elimination. See Figure 3 for an example.

We are also given a definition of the partial derivative automaton, as well as its transition function and inductive definitions of the partial derivatives of the empty string, a single character, and the union, concatenation, and Kleene star of regular expressions. An example of this type of automaton is presented in Figure 4.

3.2.6 $\mathbf{A}_{\mathrm{pd}}$ revisited

This section is dedicated to giving a simpler version of a proof by Champarnaud and Ziadi showing that the partial derivative automaton $\mathbf{A}_{\mathrm{pd}}$ is a quotient of $\mathbf{A}_{\mathrm{pos}}$ – this implies that $|\mathbf{A}_{\mathrm{pd}}| \le |\mathbf{A}_{\mathrm{pos}}|$. It also gives the Brzozowski definition of derivatives, which is a slightly different definition than that given in the previous section. The authors explain that one way in which they simplify the proof is by applying the rules for ε and $\emptyset$ wherever possible to remove the existence of similar regular expressions. A definition of a *continuation automaton* is then given in order to prove that the partial derivative automaton is indeed a quotient of the position automaton. This continuation automaton is important to understand as it is used in the propositions and examples in this section to assist in the proof.

$\mathsf{first}(\tau) = \{1, 2\}$

$\mathsf{last}(\tau) = \{1, 2, 3, 4, 5, 6\}$

i	$\mathsf{follow}(\tau, i)$
1	$\{3, 4, 6\}$
2	$\{3, 4, 6\}$
3	$\{3, 4, 6\}$
4	$\{3, 4, 5, 6\}$
5	$\{3, 4, 5, 6\}$
6	$\{3, 4, 6\}$

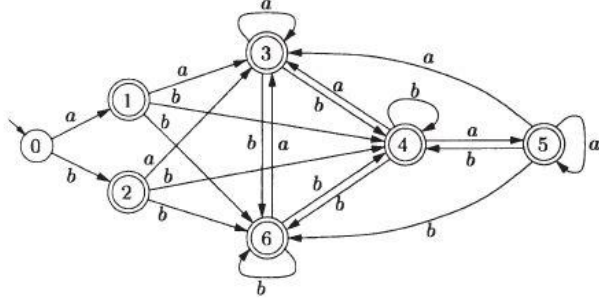

Figure 3: An example to help understand position automata: $\mathbf{A}_{\text{pos}}(\alpha)$ for $\alpha = (a + b)(a^* + ba^* + b^*)^*$ along with the corresponding values of the mappings `first`, `last`, and `follow`.

$\partial_a(\tau) = \{\tau_1\}$ $\qquad \tau_1 = (a^* + ba^* + b^*)^*$

$\partial_b(\tau) = \{\tau_1\}$

$\partial_a(\tau_1) = \{\tau_2\}$ $\qquad \tau_2 = a^* \tau_1$

$\partial_b(\tau_1) = \{\tau_2, \tau_3\}$ $\qquad \tau_3 = b^* \tau_1$

$\partial_a(\tau_2) = \{\tau_2\}$

$\partial_b(\tau_2) = \{\tau_2, \tau_3\}$

$\partial_a(\tau_3) = \{\tau_2\}$

$\partial_b(\tau_3) = \{\tau_2, \tau_3\}$

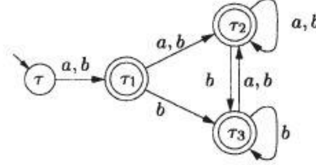

Figure 4: An example to help understand partial derivative automata: $\mathbf{A}_{\text{pd}}(\alpha)$ for $\alpha = (a+b)(a^* + ba^* + b^*)^*$ along with the corresponding partial derivatives of all states in the automaton.

3.2.7 Follow automata

The second algorithm describing the construction of NFAs from regular expressions is presented in this section. Before the algorithm is given, the *follow automaton* is defined as the quintuple, $\mathbf{A}_{\text{f}}(\alpha) = (Q_f,\ A,\ \delta_f,\ 0_f,\ F_f)$. Then, the algorithm for the removal of the ε-transitions to convert from $\mathbf{A}^{\varepsilon}_{\text{f}}(\alpha)$ to $\mathbf{A}_{\text{f}}(\alpha)$ is given as pseudocode. Figure 5 shows the follow automaton for the example regular expression that the paper follows through the entire algorithm.

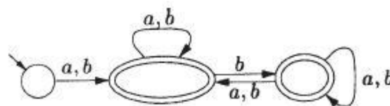

Figure 5: $\mathbf{A}_{\text{f}}(\alpha)$ for $\alpha = (a + b)(a^* + ba^* + b^*)^*$.

After providing the algorithm, some important notes concerning the complexity provided. The authors show that the complexity of their algorithm is $\mathcal{O}(|\alpha|^2)$, which is faster than the previous best worst-case given in a paper by Hopcroft and Ullman.

3.2.8 $\mathbf{A_f}$ is a quotient of $\mathbf{A_{pos}}$

Now that the follow automaton has been defined and derived, a proof is given that it is the quotient of $\mathbf{A}_{\text{pos}}$, meaning $|\mathbf{A}_{\text{f}}| \leq |\mathbf{A}_{\text{pos}}|$. The authors point out that this was unexpected but very important. They then give the definition of a follow equivalence to be used throughout the section: $i \equiv_f j$ iff i, j, or none belong to last(α) and follow(α, i) = follow(α, j). This equivalence is similar in structure to the $\equiv_k$ we learned in class for Hopcraft's minimization. The rest of this section is a series of proofs and lemmas that build upon eachother and eventually prove that the follow automaton is a quotient of the position automaton. Therefore, based on previous information, both the follow and partial derivative automata are quotients of the position automaton, but they are not comparable, which is seen in a later section.

3.2.9 $\mathbf{A_f}$ uses optimally the positions

The point of this section is to prove that the authors' construction of the follow automaton is optimal. They do so by showing that the minimal automaton, $\overline{\min(\mathbf{A}_{\text{f}}(\overline{\alpha}))}$, is equal to the follow automaton and pointing out that computing the follow automaton via the presented ε-elimination algorithm is faster than using Hopcroft's algorithm.

3.2.10 Comparing $\mathbf{A_f}$ with other constructions

Rather than providing any proofs, this section serves to show comparisons between the complexities of the different automata discussed throughout the paper. Using the four examples provided, it is shown that $\mathbf{A}_{\text{f}}$ can be smaller than $\mathbf{A}_{\text{pos}}$ and $\mathbf{A}_{\text{pd}}$, and that $\mathbf{A}_{\text{f}}$ is incomparable with either $\mathbf{A}_{\text{pd}}$ or $\mathbf{A}_{\text{cfs}}$.

3.2.11 Conclusions and further research

This section provides a nice summary of the main points of the paper:

- Two new algorithms were provided to construct NFAs.
- The constructed εNFAs are smaller than any other construction known (at the time of the publishing of this paper).
- The follow automata are the simplest out of all of the mentioned automata, are always a quotient of the position automaton, are easy to compute, and are at least as small as all the other automata.
- The time required to build the follow automata seems to be linear in terms of its size, although this is not always true.
- A more rigorous comparison between the automata should be done but is probably very complicated and most easily done using real applications.

3.3 Analysis of the Scientific Contribution of the Article

There have been many papers written on the topic of automata, with each providing algorithms to construct different types with the smallest time complexity possible, and also with the smallest size possible. This particular paper provides a definition for εNFAs that are smaller than all other known constructions and proves that its size is very close to optimal, thus setting the bar for further research in the same area. By using this almost optimal εNFA construction, the authors were able to obtain a follow automaton in the simplest way known to date. They also claim that it has a linear size most of the time, which would make it the smallest of the compared automata, but this was not proven. However, this provides opportunities for further research on this follow automaton to gain a better perspective of where exactly it fits in with the others and better gauge how optimal and small it truly is.

3.4 Main Result of the Article to be Implemented

Our team will implement the construction of follow automata from reduced regular expressions. We will do so by creating regular expressions, and then use the algorithms provided in the paper to reduce them, create and optimize εNFAs, and construct NFAs from those εNFAs. The algorithms described and given will be followed as closely as possible to ensure that the sizes and time complexities are comparable to the paper.

4 Software Design and Testing

For the project, we created software that implemented the optimized creation of a follow εNFA. We did so by first reducing the given regular expression using Algorithm 1. We could then construct an εNFA from the reduced regular expression using Algorithm 4 in the paper. Then, using Algorithm 20, we constructed the final follow automata using ε-elimination on the εNFA.

In this section, we present the data structures and algorithms developed for this software as well as the testing methodologies for both software soundness and completeness and for complexity testing. For a full description of the classes involved in the implementation, refer to Appendix A for the full API.

4.1 Data Structures

Set Definition:
The Set class is essential to our software as four of the five attributes of an automaton are represented as sets. The underlying structure of a Set object is a Dictionary, meaning that the Set object achieves the property of an unordered grouping of elements. Each set consists of key, value pairs where the keys are integers or strings and the values can be of any type, although the value type should be consistent over the entire set. The Set class contains functions to discover information about and modify the Set, as well as functions that run only on sets of Transitions.

Regular Expression Node Definition:
A regular expression node (a node in the tree representing a regular expression) is defined as an object with a value that is either an element of the alphabet of the automaton or an operation, in addition to two children and a parent, representing the other nodes in the tree to which the object is connected. The definition of this class also contains functions to test whether the language recognized by a tree rooted at a specified node is equal to null, is equal to the empty word, or contains the empty word.

Regular Expression Tree Definition:
In order to represent any given input regular expression, we chose to design a general tree data structure, since a regular expression is essentially a set of characters related by operations. Each internal node of the tree will be an operator (a two-operand union or '+', a two-operand concatenation or '.', or a one-operand Kleene star '$*$') and each leaf node will be a character of alphabet A (including `E` to represent ε, or the empty word, and `N` to represent $\emptyset$, or null). Note that due to the chosen symbols for $\emptyset$ and ε, the alphabet A will not be allowed to contain uppercase letters.

The tree is recursively defined in the following manner:

```
Expr ::=  N
        | E
        | a, a ∈ alphabet A
        | Expr + Expr
        | Expr . Expr
        | Expr*
        | ( Expr )
```

This class also contains functions to reduce the size of the regular expression via an algorithm specified in the paper, as well as a function to help build an automaton.

State Definition:
The State class is a very simple class to aid in the representation and building of an automaton. A State object has three attributes: identification, acceptance, and merge. The identification is an integer representation of the State. Note that states in an automaton are not in numerical order due to the way in which the automaton is constructed. The acceptance property is a boolean property defining whether the the state is accepting. Lastly, the merge property has a value of 0, 1, or 2 that is set during the initial automaton construction process. These values identify whether the state is part of a concatentation or iteration operation, which the program needs to know to order to perform Algorithm 4 modifications.

Transition Definition:
Transition is another straightforward class. A Transition object is represented as a triple containing the source state, the transition label (represented as a character), and a destination state.

Finite Automaton Defintion:
In order to represent any finite automaton, we chose to design a four-tuple (or 4 element tuple or collection), $M = (Q, \delta, q_0, F)$. The elements within the representation for an Automaton object are defined as follows:

Q is a set containing all of the various states of the finite automaton M.

q_0 is the starting state (identification '0').

F is a set containing all of the final or accepting states (where $F \subseteq Q$).

δ is a set containing all of the transitions in the finite automaton M.

Note that we do not include the Alphabet A in our Automaton construction; this is instead hardcoded and set up upon initial Automaton creation. The majority of the functions within the Automaton class are functions that operate on an Automaton object to modify it in some way or discover information about it. These include functions that execute the modificiations defined in Algorithm 4, that peform epsilon elimination following Algorithm 20, and that test whether a word will be accepted by an automaton.

4.1.1 Algorithms

Based on the paper, we identified three main algorithms that were needed to convert any given regular expression to a follow NFA. The general procedure is as follows: We begin with a regular expression. The algorithms for $\emptyset$-reduction, ε-reduction, and '$*$'-reduction (as defined in the paper) are all applied to this regular expression to obtain a reduced regular expression. Then, the reduced regular expression is converted into an εNFA. This conversion process is defined by the diagram presented by the paper representing the newly proposed follow εNFA construction (see Figure 1), as well as the futher modifications to remove unnecessary states and transitions. Lastly, the εNFA is converted to an NFA via an ε-elimination algorithm that makes use of topological sorting (a graph theory concept). The functions used to simulate these algorithms are presented below.

`reductions`: This function is called on a regular expression. It applies three iterations of reductions to the regular expression – $\emptyset$-reduction, ε-reduction, and '$*$'-reduction. These subroutines are all recrusive functions that are initially called on the root of the regular expression and traverse through the tree. They perform reductions within the tree by removing children of a node and overwriting the value where appropriate.

`algorithm4`: This function must be called on a reduced regular expression. It's primary purpose is to call the automaton constructor and then to call the `createENFA` function on the newly instantiated automaton – `createENFA` will update each element of the automaton quintuple appropriately and then `algorithm4` returns this automaton. The `createENFA` function simulates the creation of the follow εNFA based soley on Figure 1. This function must be paired with the `optimize` function to obtain the full functionality of Algorithm 4.

`optimize`: The `optimize` function corresponds to the second half of Algorithm 4 – it implements the further improvements defined to eliminate ε-cycles and unnecessary ε-transitions. It follows that this function must also be able to merge states in the automaton. Note that while there are subroutines for optimizatios (a)-(c), there is no subroutine for optimization (d) because our method of set construction eliminates the possibility of the occurrence of multiple transitions with the same source state, transition label, and destination state.

`algorithm20`: This function is called on the εNFA created by the `algorithm4` and `optimize` functions and returns the final follow NFA. The general idea of the ε-elimination is similar to the algorithm that was learned in class – if there exists a path between states p and q that involves all ε-transitions (an ε-path), and for state r, there exists a transition $q \xrightarrow{a} r$, then we remove all of the transitions mentioned and add a transition $p \xrightarrow{a} r$. However, this ε-elimination algorithm uses topological sorting to order the states based on the ε-transitions within the εNFA. It is important to note that the topological sorting fails if any ε-cycles remain after the `optimize` function completes.

A full list of functions used in this implementation with valid Python syntax can be found in the API section of the Appendix (Appendix A).

4.2 Experimental Protocol for the Implementation

The three fundamental properties of our program that our experimental protocol must check are soundness (the output of our program is ensured to be correct), completeness (each input our program reads results in an output), and program performance (examining both space and time efficiency).

4.2.1 The Design of the Experimental Protocol

GenerateRandRegex.py

The experimental protocol is composed of three Python object classes. The primary algorithm that defines the testing protocol is found within "GenerateRandRegEx.py" is generateExprCore(number_of_operations, regularExpression_tree_object_name), which generates a single regular expression pseudo-randomly. Every other routine, procedure, and object class is built from this core algorithm. This algorithm generates a regular expression, represented in string form using the conventions of the RegExTree.py and RegExNode.py classes, and in effect Python code to be used in TestingProtocol.py, using recursion controlled by the paritioning of a total number of regular expression operations. This way of controlling recursion means that the number of operations designated to be any output regular expression (provided in the first argument of the generateExprCore(...) algorithm), and ultimately this manner of controlling the recursive generation of a regular expression accounts for all possible forms of regular expressions that should be generated.

If we examine any subtree of regular expression tree, we see that any particular component subtree has one of three possible structures, assuming that the root node is either a concatenation or union operator (if it is a Kleene's Star operator, it can only have a left-subtreee): Symmetrical (both children nodes contain roots of subrees with an equal number operators), Left-weighted Asymmetrical (only the left subtree that has more operators than the right subtree), and Right-weighted Asymmetrical (the right subtree contains more operators than the left subtree). A good regular expression generation algorithm should potetnailly be able generate each kind of regular expression structure at any point within the overall output regular expression tree. To ensure that all three possible expression structures are possible, the generateExprCore(...) algorithm, once called, will randomly select a number k between 0 and n-1 (inclusive, note that we cannot have n be a potential value for k because we need to subtract 1 at every node that is an operator to properly reflect the amount of operators left to partition for any given subtree), and will partition the total number of argument operators allowed at any given subtree, allowing for any possible type of symmetrical, left-weighted asymmetrical, and right-weighted asymmetrical structure.. Further, at each node, the algorithm will pick a number between 1 and 3 (inclusive) and will select an operator to put at that node (1 means Concatentation, 2 means Union, and 3 means Kleene's Star) and will then proceed to recurse into that expression's arguments (where each arugment will become an additional node in the overall regular expression tree). Any time that a node is chosen to be a terminal (a character a-z or ε), the algorithm will randomly pick an index number between 0 and 26 (inclusive), and set that node appropriately to the terminal indexed (by that number)

from the alphabet inherently coded into the GenerateRandRegEx class (this random choosing for a terminal is done at the very start of each call to the algorithm, a termainl variable is generated and set aside).

Further, this generation algorithm will simultaneously generate a unique key, or string, that is essentially a "roadmap" of the final output regular expression (this key serves as a sort of "roadmap" because it is constructed at each node of the regular expression and will contain a unique symbol to represnt what kind of operator or terminal is at the current node). This key is crtical to esnure regular expression uniqueness in the upper level classes that are built upon the functioning of the generateExprCore(...) algorithm. While the generateExprCore(...) algorithm could theoretically generate a random regular expression of any possible size (as small as length = 2, for example: z* , and up to size ∞), due to limitations of time and perhaps stack space (though the latter has not posed a problem anymore for this project due to the stack modification code added in TestingProtocol.py), we only tested to see if generatedExprCore(...) could generate an expression with a maximum of 1,000,000 million operators and a potential expression length in between 1,000,000 and 2,000,000, and found that the algorithm was successfully able to. Further, this algorithm will also keep track of a few expression statistics that will be used in the validateExpr(...) routine. The second parameter to generateExprCore(...) is the name of the desired regular expression tree object that generateExprCore(...) is building (in other words, generateExprCore(...) generates a string of Python code that will ultimately have to be parsed to create the actual RegExTree object.

The function generateExpr(number_of_operations, seed, regularExpression_tree_object_name) is simply a wrapper function meant to give the programmer better control of the primary regular expression generation algorithm. The extra parameter "seed" is used to initalize the random number generator as well as to faciliate experimental reproducibility in the results of the generateExprCore(...) algorithm, and thus the seed makes the algorithm pseudo-random. Further the use of this wrapper function is necessary so that the expression generation algorithm can be called as many times as desired, since the wrapper function will do a little behind-the-scenes work to clean up the "GenerateRandRegEx.py" object class's data members, as well as store the critical input parameter that will determine the total number of operators in the output regular epxression (for use in the validity rountine described next).

To guarantee correctness of the regular expression generation algorithm, one can use validateExpr(...), also found within "GenerateRandRegEx.py". This routine will use the statistics collected in generateExprCore(...) to verify that the number of Union, Concatenation, and Star operators is equal to the number of operators input into generateExpr(...). Further, the routine will calculate how many total terminals should be in the output expression based on a simple rule and the number of each operator, and compare this prediction to the collected number of actual terminals. The key of a regular expression that is malformed and does not pass this validity check will be logged in a log file. One current limitation is that this function must be used before calling generateExpr(...) again, as the statistics for the immediately generated regular expression will be cleared.

RegExDataSet.py

The "RegExDataSet.py" object class is a child class of the "GenerateRandRegEx.py" and will ultimately utilize the regular expression generation algorithm (using the wrapper function) and the validity routine to generate a collection of a specified size of random, unique regular expressions. Uniqueness is ensured by comparing the key (as specified in the description of generateExprCore(...)) of a newly generated expression to each key of each regular expression string currently stored in the RegExDataSet collection, and if that key is found, the new expression is simply discared, but if it is not, then the expression is added to the end of the collection.

TestingProtocol.py

The "TestingProtocol.py" object class is a child class of "RegExDataSet.py" and is the actual class that contains the testing procedure to collect performance data. This class is designed for the programmer to use, as there are several lines that are uncommentable to collect different kinds of data and store them to the final "data.plot" file. It is important to note that a critical Python hack, the use of the Python function `eval()`, has been used to get everything to work. The "GenerateRandRegEx.py" and "RegExDataSet.py" classes will ultimately generate a colleciton of lines of Python code that must be parsed by the Python interpretter. Each regular expression within the RegExDataSet specialized collection is a line of Python code following the

API of “RegExTree.py” and “RegExNode.py.” In essence, the `eval()` function is used to parse each line of Python code and ultimately construct a regular expression tree from the code. Another hack is found within the “TestingProtocol.py,” as each character must be encoded into an identically named variable that the `eval()` function will evaluate to the appropriate character. Simply put, in the spirit of metaprogramming, the “GenerateRandRegEx.py,” “RegExDataSet.py,” and “TestingProtocol.py” object classes work together to write and parse further Python code that will generate the set of test regular expressions. Further, using the limits of the experiment that the programmer inputs into “limit” and “seed” the testing protocol will pre-allocate a collection that will serve to organize the data being collected based on regular expression size (i.e. the index of each element of this collection represents the regular expression size). It is important to note that seed number would need to be increased in order to smooth out a graph curve, or reduce, the variance of the data points collected.

Finally, the data, which “TestingProtocol.py” formats exactly for GNUPlot, is then used inside the “makefile.sh” script to generate the three requried plots of data that examine the desired performance characteristics of the team’s automata generation algorithms. The testing protocol was broken into 3 different versions, “TestingProtocol_noCorrectness.py”, “TestingProtocol_min.py”, and “TestingProtocol_max.py”. Each version handled the data differently: “TestingProtocol_noCorrectness.py” took collected data such as execution time or automaton size and kept a running average of that time or size at that index in the pre-allocated collection, “TestingProtocol_min.py” kept a running minimum on the same data (that is, it compared every new time or size to the current one at a given regualr expression size and took the minimum of the two), and “TestingProtocol_max.py” kept a running maximum on the same data. The Linux Computing Cluster was used to compute the massive data sets we required, which was done through a small system of scripts that connected to a primary batch job script (which was sent to the computing cluster).

4.2.2 Testing Program Soundness

For the primary functions of our program (which will determine the overall program completeness), we will test soundness as follows:

`reductions(regex)`: We feed thousands of regular expressions of random sizes into this function and then proceed to feed the resulting output (a reduced regular expression) back into the fucntion. If the resulting output of both calls to the function is identical, we have verified that this function is sound.

εNFA and NFA constructions: To test that our εNFAs and NFAs are being constructed properly, we run `isAccepted(string)` on the εNFA after optimizations and then on the corresponding NFA to ensure that the string is accepted or rejected by both automata. More specifically, for each regular expression, we generate thousands random strings with a random length between 0 and 20. The characters that are used in the string generation use only the characters in the alphabet of the current automaton that is being checked, to give us a better chance of getting accepted strings. The “True” or “False” outputs for each string are stored in arrays for each automata. The arrays are then checked for equality. This makes it much easier to check soundness rather than doing the checking for equality for each of the automata ourselves.

4.2.3 Testing Program Completeness

Our experimental protocol should test for crashes. To do this, we will use a method similar to above, but instead ensure that for each randomly generated regular expression and string, we have no crashes while running through the implementation of the paper. More specifically:

`reductions(regex)`: We pass thousands of regular expressions into this function and ensure that for every input there is some sort of output, whether it be a reduced regular expression or an error message that we generated ourselves.

εNFA and NFA constructions: To test that our program doesn’t crash on the construction of εNFAs and NFAs, we run a few thousand reduced regular expressions through `createENFA(reducedRegex)` and then pass the produced εNFA into `algorithm20()`. For each reduced regular expression used, both constructions complete without resulting in a crash.

This testing was done in parallel with our tests for soundness and obtaining our data for the graphs. Those required the generations of thousands of regular expressions, which were then run through all of our algorithms (reducing, generating εNFAs, optimizing, generating NFAs). Because no crashes occurred during all of this, we trust that our code has the property of completeness.

4.2.4 Testing Program Performance

To ensure that our program runs with a complexity that agrees with the complexity given in Theorem 11 (page 148) and Theorem 21 of the paper (page 154), we run the entire program using about 1,600,000 regular expressions (400 expressions of length 2 to 800, 4,000 seeds) and create nine graphs: three for the execution time versus the size of the regular expression, three of space used for the construction of a εNFA versus the size of the regular expression, and three of space used for the construction of the final follow-automaton versus the size of the regular expression. For each comparison, one graph is made for each of the average values, maximum values, and minimum values. We have verified that for the time data plots and the three space data plots for construction of a final automata, the relationship is $\mathcal{O}(|\alpha|^2)$ for both time and space, which agree with the paper. Further, for the data plots for the construction of an εNFA, the relationship is $\mathcal{O}(|\alpha\|)$, as asserted in the paper. All data was plotted on a log scale, so a data curve, in order to exhibit the appropriate desired behavior as stated in the paper, would need to be a straight line (and a slope that would need to as close to possible to the base linear or polynomial function desired). These graphs can be seen in Appendix B. It is interesting to note that for all graphs, raising the seed parameter did not smooth out the curve evenly, as variance appears to get worse towards the end of a given data curve (or towards the maximum tested expression length). For the graphs that either plotted minimum or maximum values, one can see that having a steep seed size (as used in the plots that contained averaged data) in relation to the expression length range did not yield as great a smoothing effect as it did in the averaged data plots.

4.2.5 Subroutine testing

Within our major algorithms (`reductions`, `algorithm4`, `optimize`, and `algorithm20`), there are numerous subroutines that needed to be created and tested for correctness as well. The tests and corresponding output for some of these subroutines can be seen using any classes that end with “Tester”.

4.2.6 Potential Protocol Improvement

Though the primary regular expression generation algorithm was validated, and is a sufficient check given the scope of this experiment, a more formal and rigorous algorithmic proof of its output should be conducted using a routine that accepts as input the regular expression outputted by the generation algorithm, then checks to make sure all of the parentheses are balanced and all operators have appropriate arguments (i.e. verify that no blank arguments popped up, which would result in a malformed expression). Further, while the current protocol does sufficiently create a working set of unique regular expressions of wide range of lengths, a great deal of space inefficiency resutls from this process. The middle man class, “RegExDataSet.py”, could very simply be removed, and the flow of regular expressions could be generated and this process could be integrated right within “TestingProtocol.py” and the uniqueness could occur right on the spot within the primary program loop of that class (however, this would not cut out the linear search time needed to compare the key of a new expression to those in the current RegExDataSet collection object, all it would simply do is move this linear search time to occur later but within the “TestingProtocol.py”, as one would still need to check the key within any collection one creates to store the expressions at that stage). Additionally, no study was conducted as to the time efficiency of the built-in Python evaluation function used to parse the regular expression Python code strings. It would be possible to rewrite the expression generation algorithm to recursively construct a regular expression tree using RegEx object nodes, although this could take more space, as the object data tied to a RegEx node is larger than of string. Whenever a RegEx tree is created in our current version of the testing protocol, it is used on the spot, then discarded after all of the paper algorithms have been applied to it. It was decided that this was more space efficient, rather than storing a collection of RegEx node objects trees, however a study of this should be conducted to verify this assumption. Finally, an algorithm that range generates expressions solely based on a given expression output length would be better and allow for no range in possible expression sizes in the output (currently, our implementation

yields an expression that is guaranteed to have the same number of operators entered at time of input, but the expression size of the output could fall within a range of sizes, which requires more seeds in the final program to create a better distribution of regular expression sizes) and ultimately more control over the final expression output.

4.3 Project Comments

4.3.1 Reflection on our Program

After running our algorithms using thousands of regular expressions and their corresponding εNFAs and NFAs, we were able to collect runtime and size data. Plotting those data and comparing the time and space complexities to those claimed in the paper revealed that our algorithms support the claims of the authors.

4.3.2 The Use of Python for this Project

There were pros and cons to using Python for this project. The major benefit was that each team member already knew Java, and Python is very easy to learn with this background. However, Python is certainly not the best choice for a project like this that deals with a lot of recursion, list comprehension, and tree traversal. There were many instances where we wished we had chosen OCaml or Haskell due to the amount of code that certain algorithms require in Python versus a functional language. However, there were also times when it was great to be using Python because we can could local variables, whereas functional languages cannot.

Another point to be made about Python that had benefits, but also caused some issues, is the fact that it is untyped. At first it was great to not worry about variable, parameter, or return types because it was less coding and made progress a bit faster, but as our code base grew, it became difficult to remember exactly what types we had certain functions taking in (i.e. a string representation of an integer versus an actual integer). This led to the frequent need to use the Python debugger (PDB) to discover exactly what was happening, as ever though the logic was wrong, the program would still run.

4.3.3 Teamwork Experience

The distribution of work ended up being fairly even at the end at the end of the project. The algorithms were distributed among two team members, most of the testing and graph generation was given to another team member, and all members worked on the paper and final presentation. The version control repository (SVN) was extremely useful for this project and frequent team meetings allowed us to keep track of the progress of all team members.

Appendices

A API

A.1 Set

This class defines a Set type, not using the python-provided sets. It contains the operations necessary to interact with sets such as adding, removing, and getting elements, and it also provides for the ability to union, intersect and take the difference of two sets. This project does not allow the same key to be associated with two different values

Class Constructor:

```
def __init__(self):                  Constructs an empty Set object. For the Set, keys can be
                                     numbers or strings.
```

Class Methods:

```
def addElement(self, key, value):    Adds an element to the set
                                     Params: key - a key to associate with the value
                                             value - the value of the element to be added

def removeElement(self, key):        Removes an element from the set using the specified key
                                     Param: key - they key associated to the value to be removed

def getElement(self, key):           Returns the value corresponding to the key if
                                     the key is present in the Set, None otherwise
                                     (serves to test whether the element is present in the set)
                                     Param: key - the key associated with the element we want to
                                                  access
                                     Returns the element or None if it is not an element

def setValue(self, key, value):      Sets the value of the key value pair for the pair
                                     corresponding to the given key
                                     Params: key - the key for the pair the user
                                                   would like to update
                                             value - the new value for the pair

def size(self):                      Returns the size of the set (number of elements)

def setUnion(self, other):           Unions two sets
                                     Param: other - the set to union the current set with
                                     Returns a set that is the union of the two sets

def setIntersect(self, other):       Intersects two sets
                                     Param: other - the set to intersect the current set with
                                     Returns a set that is the intersection of the two sets

def setDifference(self, other):      Takes the difference of the two sets
                                     Param: other - the set to take the difference of the current
                                                    set with
                                     Returns a set that is the difference of the two

def printFullSet(self):              Prints all elements of the set in the form (key, value)
                                     with elements seprated by commas
```

```
def printKeys(self):                    Prints the key for each pair in the set with the values
                                        separated by commas

def printValues(self):                  Prints the value for each pair in the set with the values
                                        separated by commas

def copySet(self, oldSet):              Copies all elements from the parameter to the Set on which
                                        the function is being called
                                        NOTE: the Set on which the function is being called must
                                        already be initialized
                                        Param: oldSet - the set to be copied into the current set

def eTransitionFrom(self, state):       Finds all transitions that exist from the given state to
                                        another state via an epsilon transition. Function should be
                                        called on a set of Transitions.
                                        Param: state - the state to use in the transition search
                                        Returns a list of epsilon transitions

def eTransitionTo(self, state):         Finds all transitions that exist from a state to the given state
                                        via an epsilon transition. Function should be called on a set of
                                        Transitions.
                                        Param: state - the state to use in the transition search
                                        Returns a list of epsilon transitions

def transitionFrom(self, state):        Finds all transitions that exist from the given state to another
                                        one via a transition that is not epsilon. Function should be
                                        called on a set of Transitions.
                                        Param: state - the state to use in the transition search
                                        Returns a list of transitions

def transitionTo(self, state):          Finds all transitions that exist from a state to the given state
                                        via a transition that is not epsilon. Function should be called
                                        on a set of Transitions.
                                        Param: state - the state to use in the transition search
                                        Returns a list of transitions

def transitionExists(self, f, v, s):    Determines if a transition exists with the specified
                                        states and transition
                                        Params: f - the source state of the transition
                                                v - the transition label
                                                s - the destination state of the transition
                                        Returns true if the specified transition exists and
                                        false otherwise

def findTransitions(self, string, state):
                                        Finds transitions that have either an epsilon transition or a
                                        transition with the specified transition label from a given
                                        state
                                        Params: string - the transitionLabel to look for
                                                state - the state to look from
                                        Returns an array of transitions from the given state with an
                                        epsilon transition or a transition with the specified label
```

```
def removeTransitions(self, state):   Removes all transitions in the set that have a sourceState
                                      equal to the state passed in
                                      Param: state - the state to remove completely from the eNFA

def sort(self, transitionSet):        Sorts a set of states topologically with respect to the
                                      order p is less than or equal to q if and only if
                                      there exists an epsilon transition from p to q.
                                      Should be run on a set of states.
                                      Param:transitionSet - the full set of transitions over
                                      which the function should be run
                                      Returns a list of states ordered from least to greatest
```

A.2 RegexNode

This class defines a Regular Expression Node type. It contains the operations necessary to interact with a node such as creating one, and setting its parent and children. It also has accessor methods and special methods pertaining to the paperto determine the language from a node.

Class Constructor:

```
def __init__(self, value, leftChild, rightChild):  Constructs a basic regular expression node
                                                   element which is the fundamental component
                                                   of the regular expression tree. It has a
                                                   value, a leftChild, and a rightChild.
```

Class Methods:

```
def getCurrentNodeVal(self):     Gets the value of the current node
                                 Returns the value of the node

def getLeftChild(self):          Gets the left child of the current node
                                 Returns the left child node of the current node

def getRightChild(self):         Gets the right child of the current node
                                 Returns the right child node of the current node

def getParent(self):             Gets the current node's parent
                                 Returns the current node's parent if it has one

def setParent(self, newParent):  Sets the current node's parent. Takes care of setting the new
                                 parent as the grandparent's child, if there is a grandparent
                                 Param: newParent - a new node that will be used as the new parent

def isLeftChild(self):           Determines if the current node is the left or right child
                                 Returns true if it is the left child and false if it is the
                                 right child

def otherChild(self):            Determines the "sibling" of the other node
                                 Returns the sibling of the current node if there is one

def eIs(self):                   Tests whether the language recognized by tree rooted at the
                                 given node is equal to E
                                 Returns true if the language recognized is E and false otherwise
```

`def eIn(self):`	`Tests whether E is an element of the language recognized by the tree rooted at the given node` `Returns true is E is an element of the language and false otherwise`
`def nullIs(self):`	`Tests whether the language recognized by the tree rooted at the given null is equal to N` `Returns true if the language is equal to N and false otherwise`

A.3 RegexTree

This class defines a Regular Expression Tree type. It contains the operations necessary to interact with a tree such as traversing, constructing, and printing (postorder). It also contains methods that are necessary for implementation of the paper such as reducing a given regular expression tree to contain as few epsilons, Kleene Stars and nulls as possible. *The nulls are either completely removed or the entire tree reduces to null

Class Constructor:

`def __init__(self):`	`Initializes a basic regular expression tree, initally set to be the emtpy set N.`

Class Methods:

`def symbol(self, char):`	`Creates a terminal node with a specified value`
	`Param: char - the character to set as the terminal value`
	`Returns the created node`

`def eps(self):`	`Creates a terminal node with the specific value of E for empty string` `Returns the created node`
`def null(self):`	`Creates a terminal node with the specific value of N for null` `Returns the created node`

`def union(self, node1, node2):`	`Creates a node with value +, which is the union of two other nodes`
	`Params: node1 and node 2 - the two children nodes of the union`
	`Returns the created node with any subnodes`

`ef concat(self, node1, node2):`	`Creates a node with value . for concatentation`
	`Params: node1 and node2 - the two children nodes of the concatenation`
	`Returns the node and any subnodes`

`def star(self, node):`	`Creates a node with value * for Kleene Star`
	`Param: node - the root of the subtree that will be starred`
	`Returns the star node and its subnodes`

`def construct(self, rootNode):`	`Sets the passed in node as the root and returns a now completed tree`
	`Param: rootNode - the root of the tree`
	`Returns the root node (entire tree)`

```
def printRegex(self):            Prints the regular expression that the tree represents

def printTree(self):             Prints a tree in postorder traversal (calls traverse)


def inTraverse(self, node):      Traverses a tree in-order (for printRegex())
                                 Params: node - the node to traverse from (initial call uses root)

def postTraverse(self, node):    Traverses a tree in post order and prints it out as it traverses
                                 Param: node - the node to start traversing from
                                 (usually the root)

def starReduction(self, node):   Implementation of the *-reduction portion of Algorithm 1
                                 'for any vertex labelled by *, if its child is also labelled
                                 by *, then replace it by its child'
                                 Param: node - the node that is currently being examined
                                 for reduction

def eReduction(self, node):      Implementation of the E-reduction portion of Algorithm 1
                                 'for each vertex with the language from that node(B) equalling
                                 E, then if the parent of B is labelled by ., then replace the
                                 parent by the other child if the parent is labelled by *, then
                                 replace the parent by the child if the parent of B is labelled
                                 by + and E is in the language of the other child, then replace
                                 it by its child'
                                 Param: node - the node that is currently being examined
                                 for reduction

def emptyReduction(self, node):  Implementation of the null reduction portion of Algorithm 1
                                 'compute, for each vertex B,whether ot not L(B) = N and then
                                 modify the tree such that at the end, either it evaluates to
                                 N, or contains no N'
                                 Param: node - the node that is currently being examined
                                 for reduction

def reductions(self):            Calls each of the reductions in succession. This is equal to
                                 running Algorithm 1

def algorithm4(self):            Implements Algorithm4 from the paper, which translates a
                                 regular expression into an eNFA
                                 Returns the automaton

def getSize(self, node):         Returns the size of the regular expression as defined by the
                                 paper
```

A.4 State

This class defines a State type. Each has a unique identifier and a flag as to whether it is an accepting state or not.

Class Constructor:

```
def __init__(self, identifcation, acceptance, merge):  Constructs a state with the specified
                                                        identification value, whether it is
                                                        accepting or not, and a flag to be used
                                                        by the merge function as part of
                                                        Algorithm 4
```

Class Methods:

```
def getID(self):           Gets the unique identifier of the current state
                           Returns the value of the identifier (integer)

def getAcceptance(self):   Gets whether the current state is accepting or not
                           Returns true is the state is accepting and false otherwise

def __str__(self):         Prints a state
```

A.5 Transition

This class defines a transition. A transition is a tuple that consists of a state, a transition character, and another state.

Class Constructor:

```
def __init__(self, sourceState, transitionLabel, destinationState):
                                Constructs a basic transition triple. A transition triple
                                is composed of a state, transition label, and a
                                destination state (in that order)
```

Class Methods:

```
def getSourceState(self):       Gets the first value in the tuple (source state)
                                Returns the first value of the tuple, which will be a
                                State type

def getTransitionLabel(self):   Gets the second value of the tuple (transition label)
                                Returns the transition label (a character in our case)

def getDestinationState(self):  Gets the third value in the tuple (ending state)
                                Returns the last value of the tuple, which will be a
                                State type

def __str__(self):              Prints a transition
```

A.6 Automaton

This class defines an Automaton with the standard properties of an Automaton (set of states, an alphabet, set of transitions, starting state, set of final states) It is a tuple, but does not use tha Python-provided tuple

Class Constructor:

```
def __init__(self, Q, delta, q0, F):  This constructs an automaton, which is a tuple
                                      (not using the python provided tuple). The tuple
                                      consists of:
                                        Q - a set of states
                                        A - the alphabet used (hardcoded in)
                                        delta - a set of transitions
                                        q0 - the starting state
                                        F - a set of accepting states
```

Class Methods:

```
def algorithm20(self):                This implements algorithm 20 from the paper which takes
                                      an eNFA and turns it into an NFA and then returns the NFA

def createENFA(self, currentNode, nextStateID, fromState, toState):
                                      Traverses through a regular expression tree, creating
                                      states and transitions according to the algorithm in the
                                      paper for creating eNFAs.
                                      Params: currentNode - the current node of the RegexTree
                                              being traversed
                                              nextStateID - the id that the next created state
                                              will be assigned (incremented by one each time a
                                              state is created)
                                              fromState - equivalent to sourceState in a
                                              transition
                                              toState - equivalent to destinationState in a
                                              transition
                                      Returns nextStateID so that it can be passed up through the
                                      recursive calls

def optimize(self):                   Removes unnecessary transitions in eNFA according the the
                                      paper

def printAutomaton(self):             This will print an automaton. It will print the value
                                      of each member of the "tuple" and if it is a set, a special
                                      print function will be defined for how to print the set

def getSize(self):                    Returns the size of the automaton as defined by the
                                      paper

def isAcceptedByE(self,state):        Recursively checks to see if when we are at the end of
                                      a word, there are e-transitions to an accepting state
                                      Param: state - that state to check for e-transitions
                                      from
                                      Returns true if there are e-transitions to a final state
                                      and false otherwise
```

```
def isAccepted(self, string, state):  Checks to see if a string is accepted by the automaton
                                      Params: string - the string to check
                                              state - the state to check from (original call
                                              needs to make sure state=0
                                      Returns true if the word is accepted and false otherwise

def findCycle(self, currState, origState, visitedSet):
                                      Builds lists of states that form epsilon cycles within
                                      the automaton. Used as a helper function for optimizationB.
                                      Params: currState - the state currently being visited
                                              origState - the state in the middle of an iteration
                                              visitedSet - the set of states and boolean values
                                              indicating what states have already been visited
                                      Returns a list of lists that contain possible paths for an
                                      epsilon cycle

def concatMerge(self, mergeFrom, mergeTo):
                                      Effectively merges two states of the automaton
                                      Params: mergeFrom - the state that will be merged into
                                      another state
                                              mergeTo - the state into which the two states
                                              will merge

def iterMerge(self, cycleList):       Merges the states of the automaton and removes
                                      the transitions, following the regrouping defined
                                      by optimizationB
                                      Param: cycleList -  a list of lists that gives the epsilon
                                              cycles in the automaton

def optimizationA(self):              Merges two states that are connected by an epsilon
                                      transition after concatenation

def optimizationB(self):              Identifies the states in the middle of iterations
                                      contained in epsilon cycles and defines how the states
                                      of the automaton should be merged, then calls out to
                                      iterMerge function for actual merging

def removeAllECycles(self):           (Similar to the operation of optimizationB) Identifies
                                      epsilon cycles formed by merges from previous optimization
                                      iteration and defines how the states of the automaton
                                      should be merged.
                                      Returns the number of cycles found (length of the
                                      cycles list)

def optimizationC(self):              If there is one initial transition from the starting
                                      state and it is labelled by  an epsilon, the starting
                                      state is merged with the destination state of that
                                      epsilon transition

def optimize(self):                   Removes unnecessary transitions in the generated eNFA by
                                      calling out to functions that execute the three
                                      modifications
```

B Graphs

In this section, beginning on the next page, you will find the nine graphs referenced in the section of the paper that explained how we tested program performance. Each figure is accompanied by a caption explaining what the graph describes as well as a brief comment on data point variance.

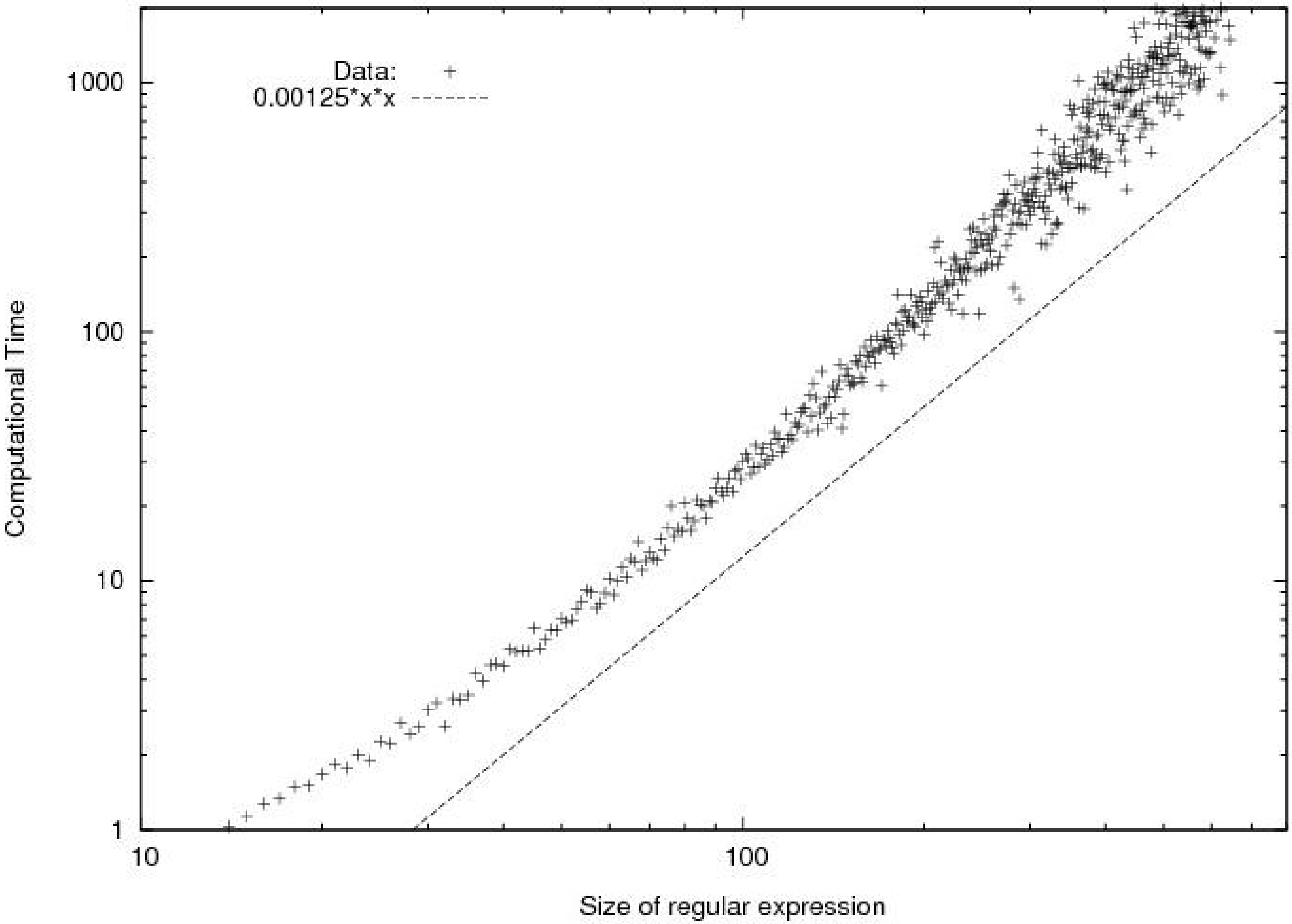

Figure 6: Time it takes to run `reductions`, `algorithm4`, `optimize`, and `algorithm20`. These data points have been averaged. Note that variance of the data points increases as expression size gets larger.

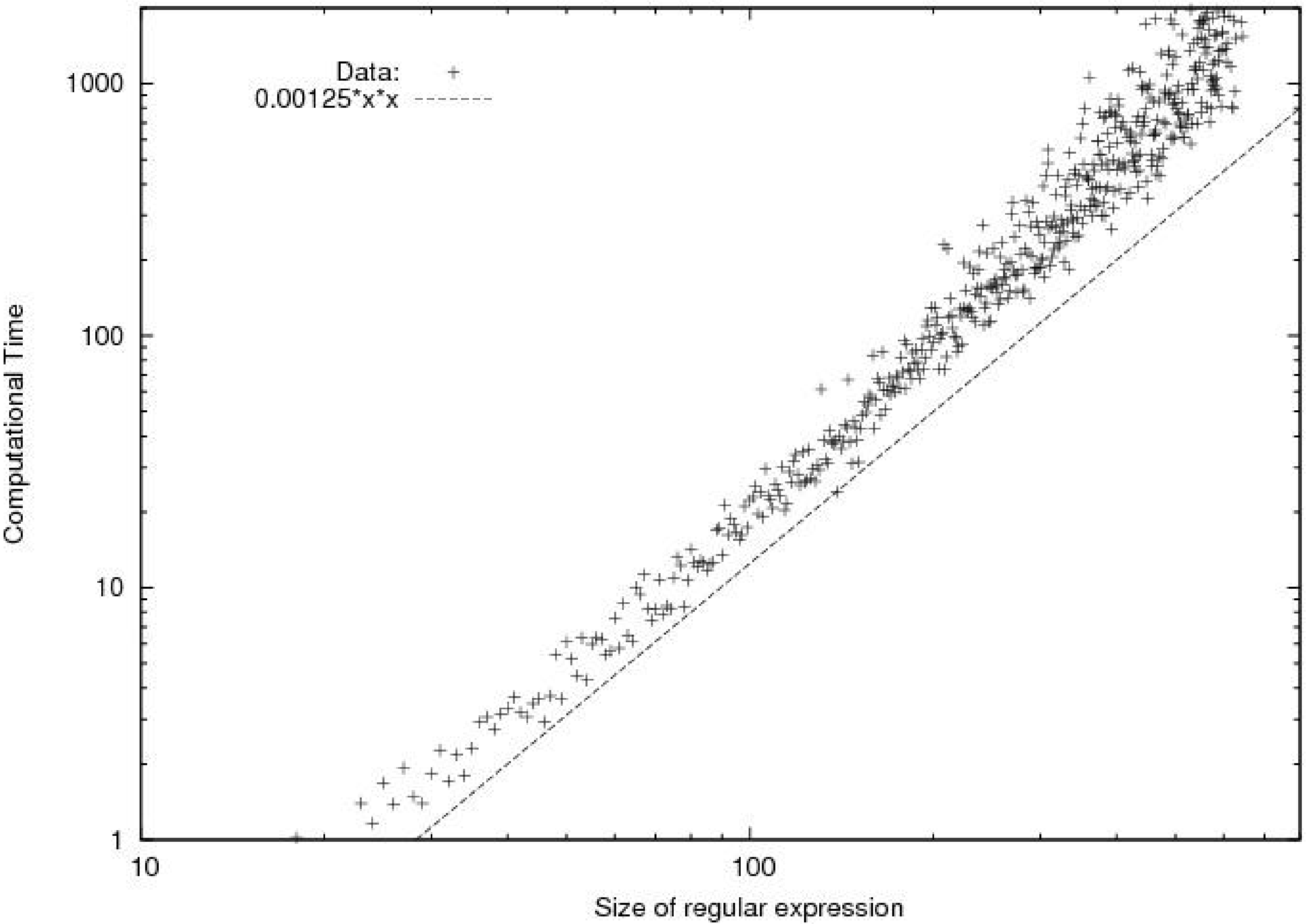

Figure 7: Time it takes to run `reductions`, `algorithm4`, `optimize`, and `algorithm20`. These data points have been minimized. Note that variance of the data points increases as expression size gets larger, and is worse in general across the whole curve in comparison to the averaged data.

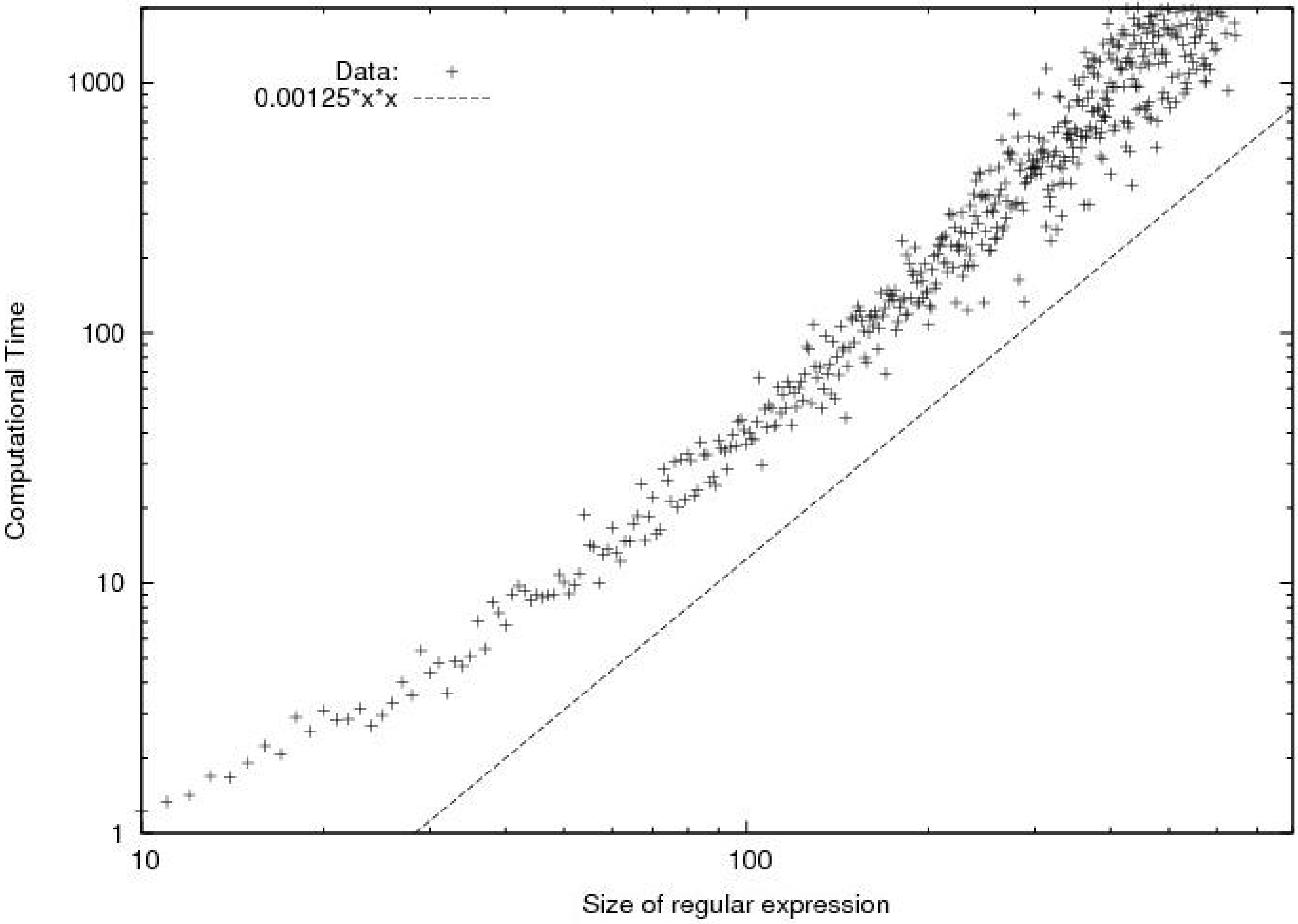

Figure 8: Time it takes to run `reductions`, `algorithm4`, `optimize`, and `algorithm20`. These data points have been maximized. Note that variance of the data points increases as expression size gets larger, and is worse in general acorss the whole curve in comparison to the averaged data.

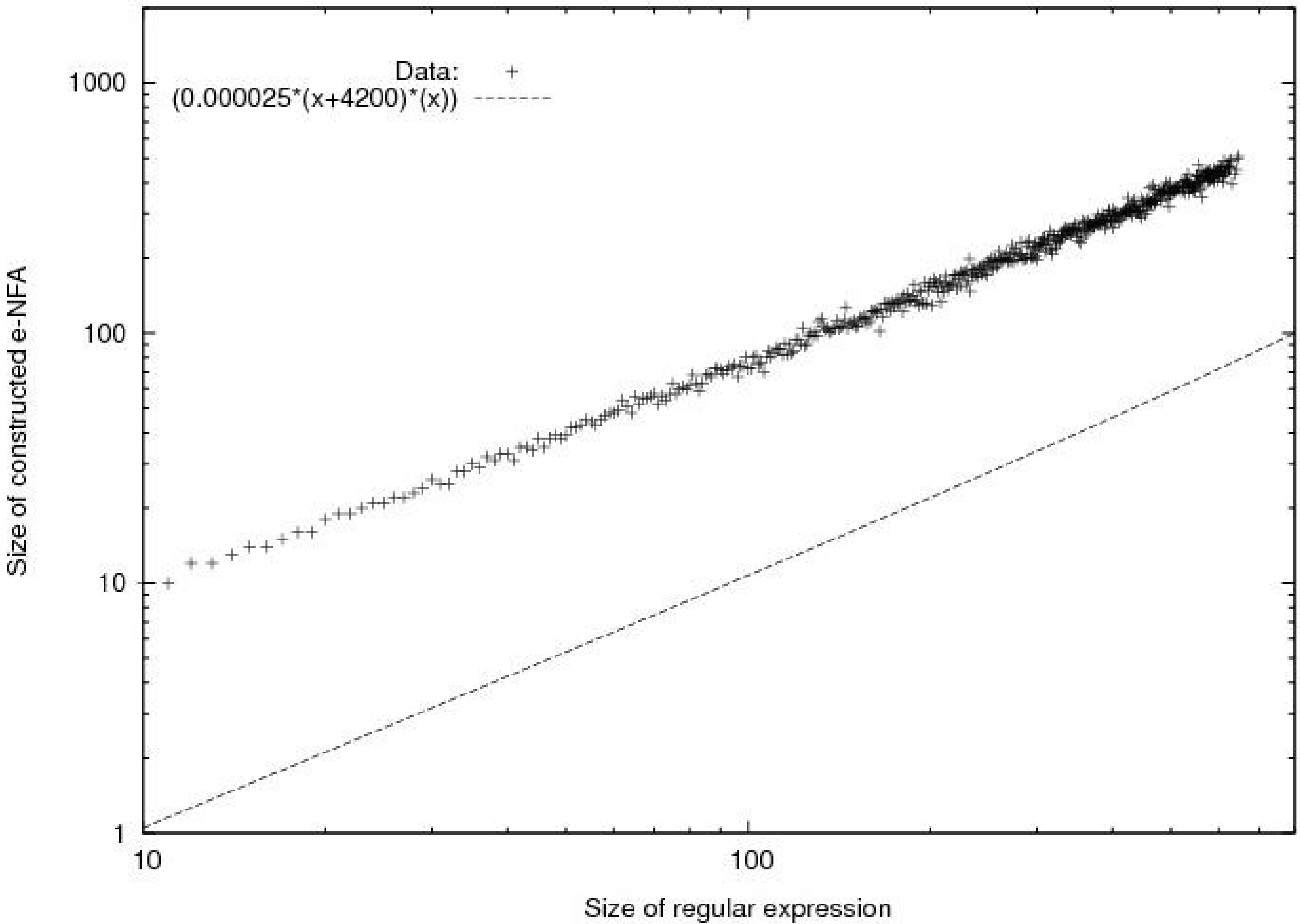

Figure 9: Space it takes to construct an εNFA using `algorithm4`. These data points have been averaged. Note that the variance of the data points increases as the expression size increases.

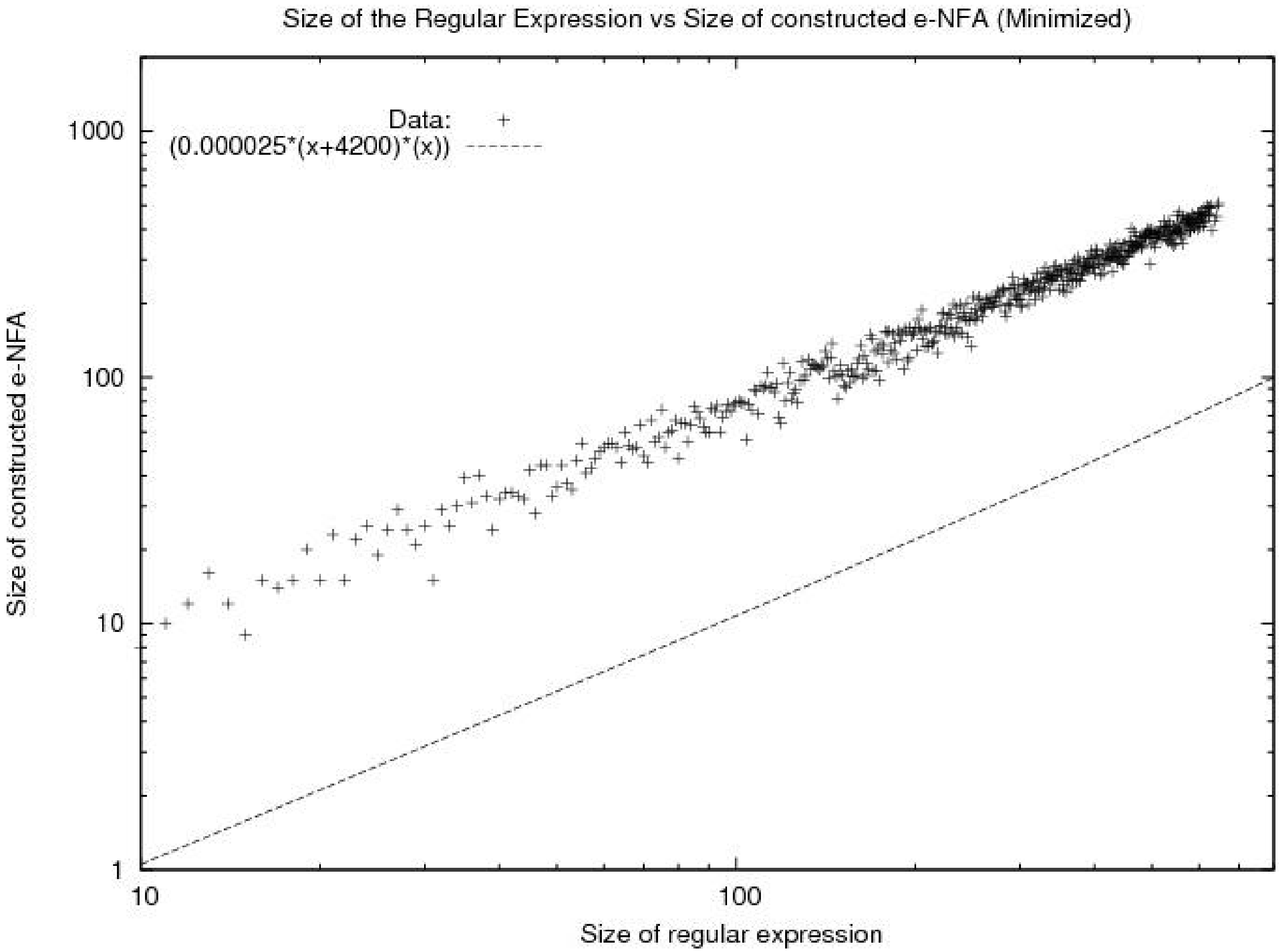

Figure 10: Space it takes to construct an εNFA using `algorithm4`. These data points have been minimized. Note that the variance of the data points is worse in general across the whole curve in comparison to the averaged data.

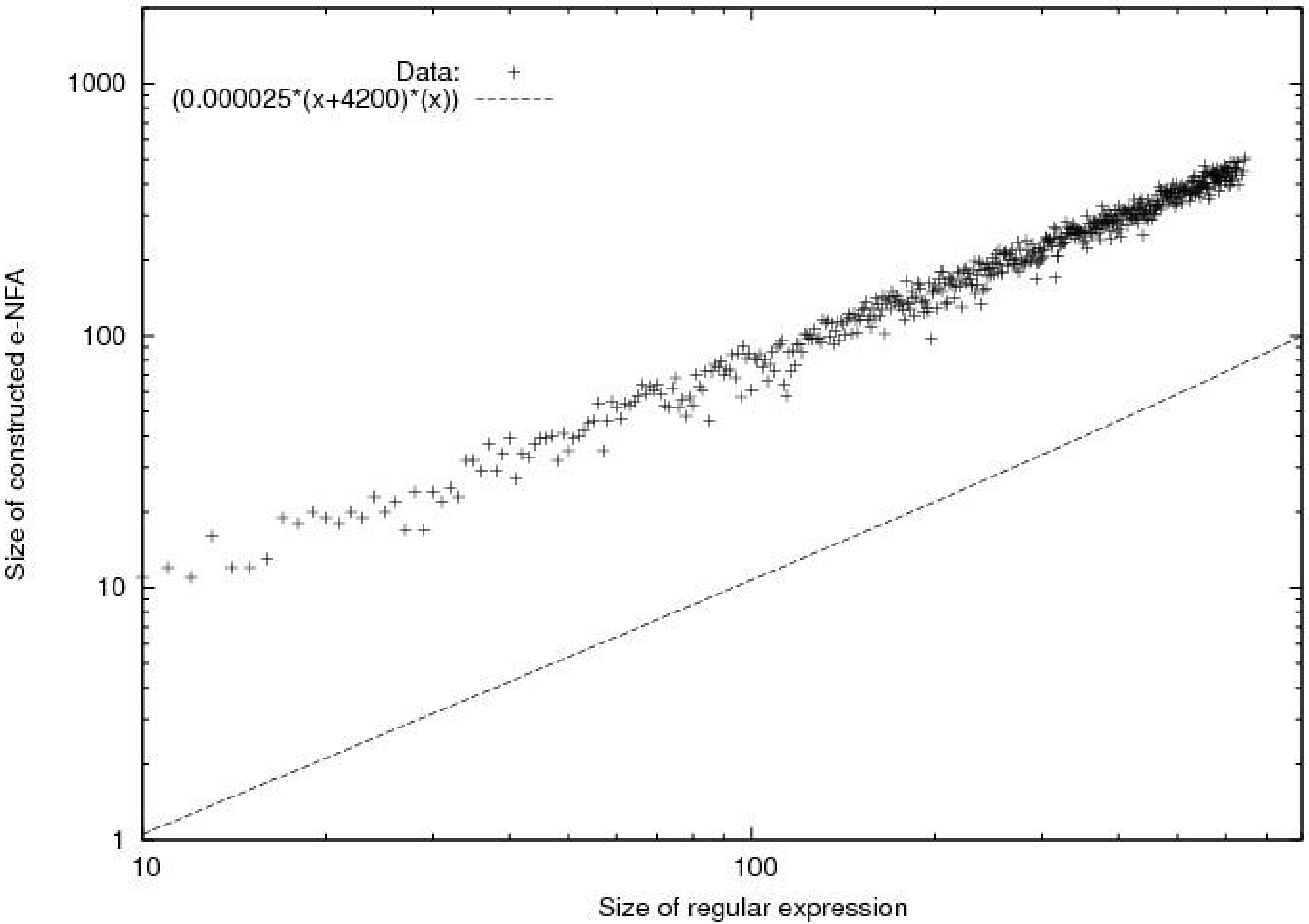

Figure 11: Space it takes to construct an εNFA using `algorithm4`. These data points have been maximized. Note that the variance of the data points is worse in general across the whole curve in comparison to the averaged data.

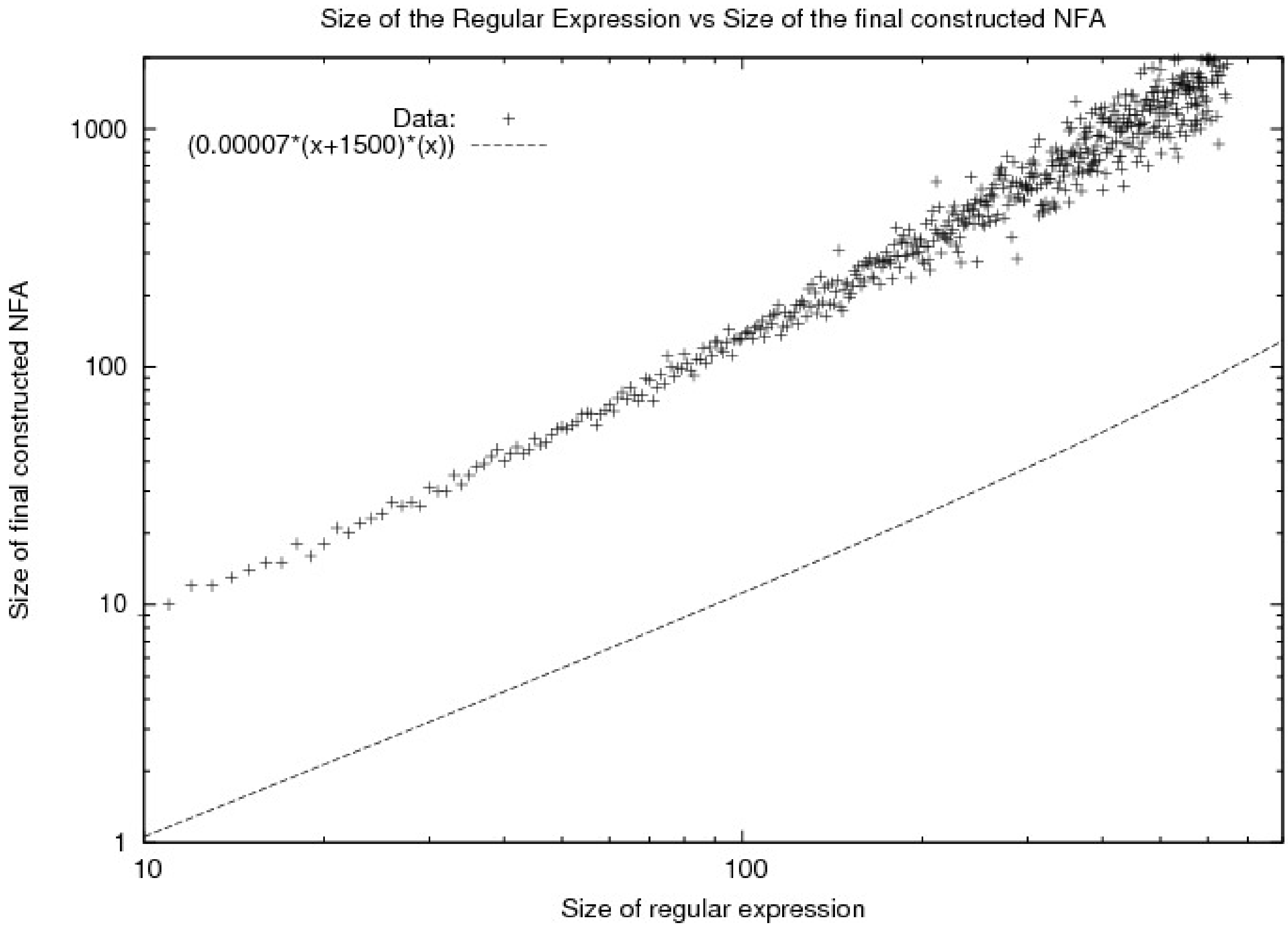

Figure 12: Space it takes to construct the final automaton using `reductions`, `algorithm4`, `optimize`, and `algorithm20`. These data points have been averaged. Note that the variance of the data points increases as the expression size increases.

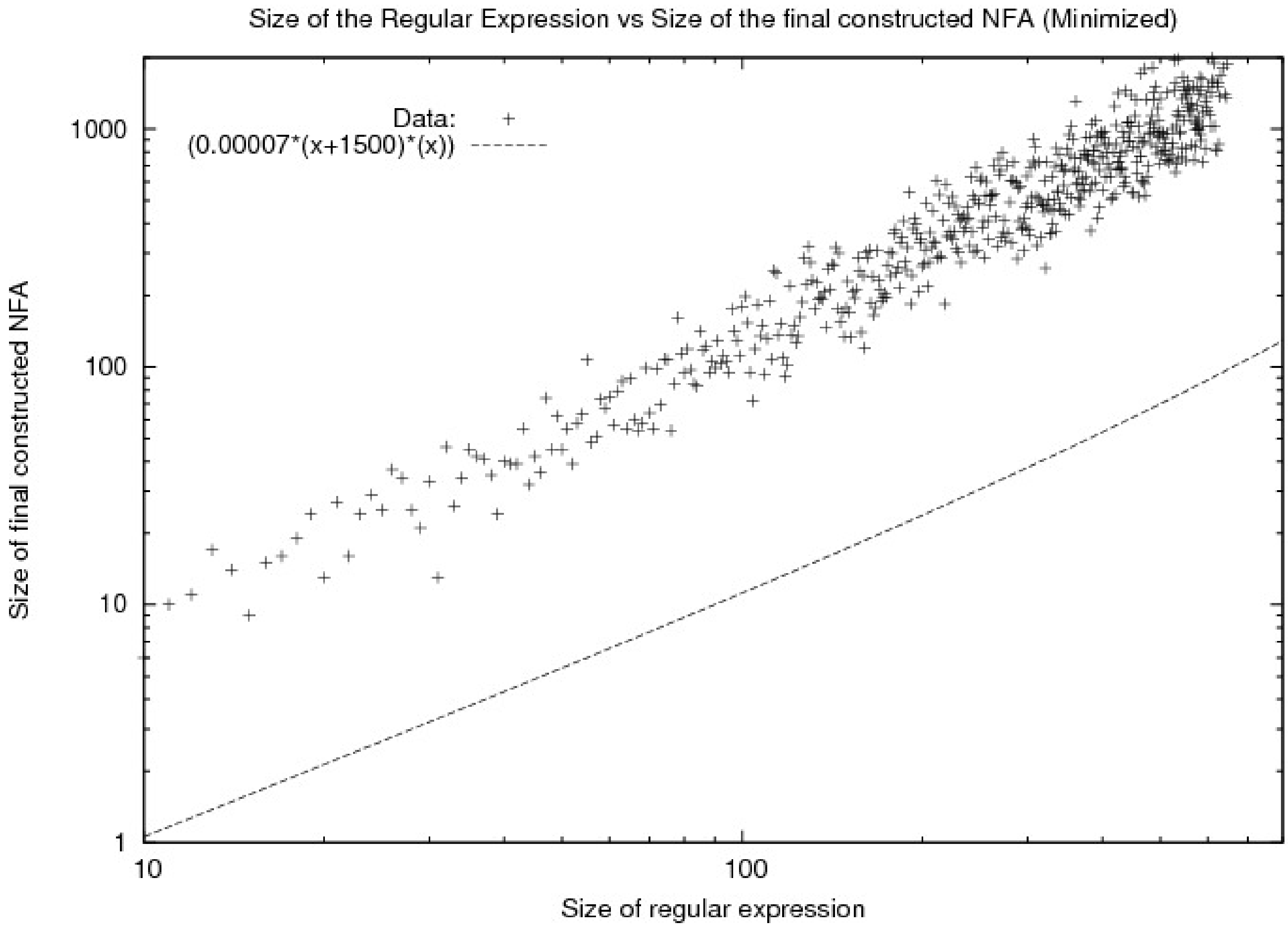

Figure 13: Space it takes to construct the final automaton using `reductions`, `algorithm4`, `optimize`, and `algorithm20`. These data points have been minimized. Note that the variance of the data points is worse in general acoess the whole curve in comparison to the averaged data.

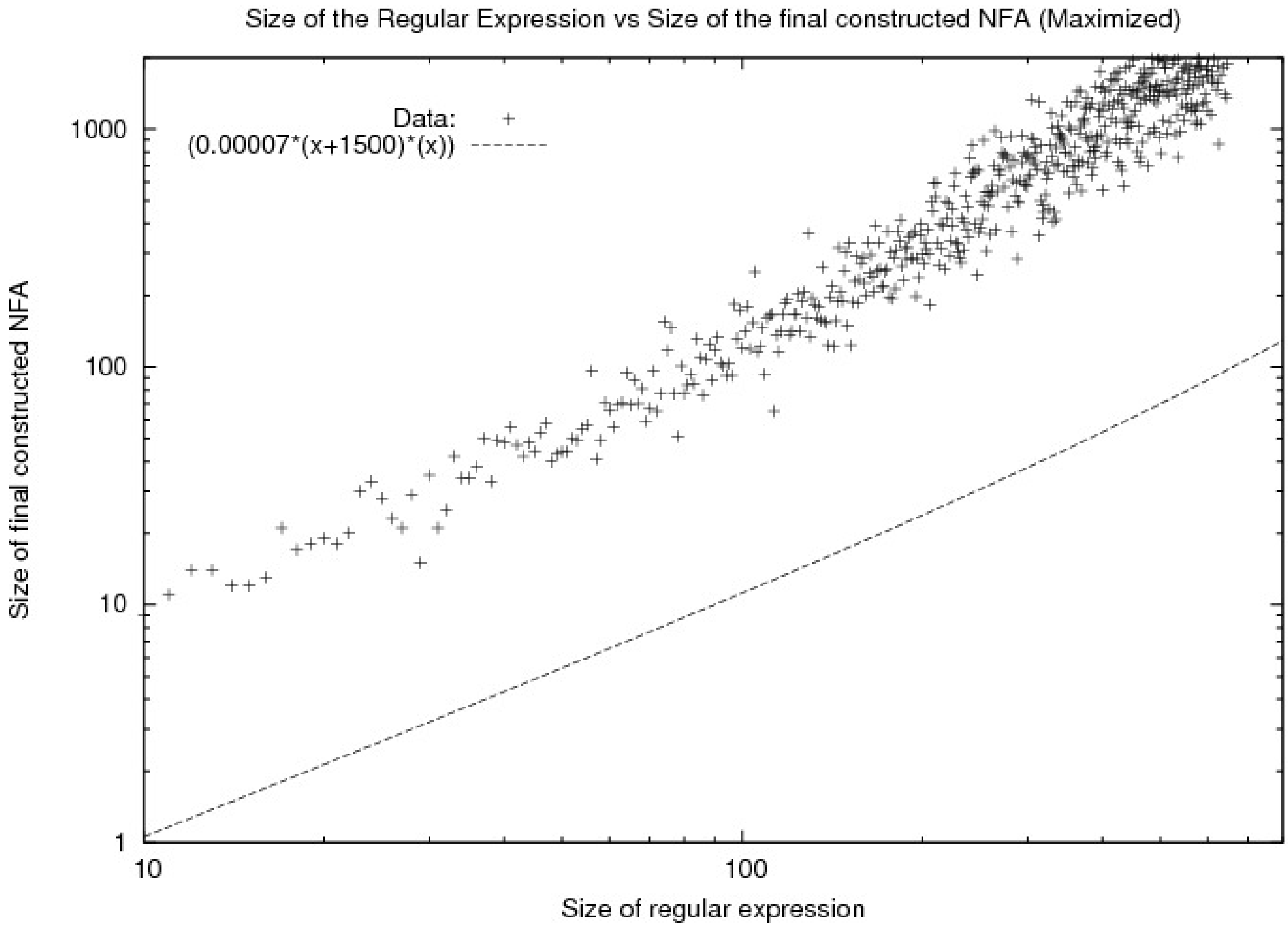

Figure 14: Space it takes to construct the final automaton using `reductions`, `algorithm4`, `optimize`, and `algorithm20`. These data points have been maximized. Note that the variance of the data points is worse in general acoess the whole curve in comparison to the averaged data.