COMPUTER SCIENCE THEORY AND PROGRAMMING

Why the One Needs the Other, and We All Need to Know a Bit of Both

Prof. Brink van der Merwe

April 2016

COMPUTER SCIENCE THEORY AND PROGRAMMING: WHY THE ONE NEEDS THE OTHER AND WE ALL NEED TO KNOW A BIT OF BOTH

Inaugural lecture delivered on 25 April 2016

Prof. Brink van der Merwe
Department of Computer Science
Faculty of Science
Stellenbosch University

Editor: SU Language Centre
Printing: SUN MeDIA
ISBN: 978-0-7972-1595-5

Biographical information

Brink van der Merwe is a full Professor in Computer Science at the University of Stellenbosch and part of an established research group working at the intersection of algebra, automata theory, and machine learning. After studying actuarial science at the University of Stellenbosch (1985–1987) and working for the insurance company Sanlam, he moved to the USA for five years and received his PhD in algebra at Texas A&M University in 1995 under the supervision of Carl Maxson. From 1996 to 2001, he was a lecturer and later a senior lecturer at the Mathematics Department at the University of Stellenbosch, before moving to Computer Science. He served as head of Computer Science from 2005 to 2007. Since 2008, six students received their MSc degrees in Computer Science under his supervision, five of them cum laude. He has produced twenty-two peer-reviewed journal articles and fifteen peer-reviewed publications in conference proceedings. He is on the editorial board of the *Journal of Universal Computer Science*, is on the programme committee for the Conference on Implementation and Application of Automata, and regularly acts as a reviewer for international conferences and journals. He is married to Antoinette van der Merwe and they have one son, Alexander.

Abstract

It is not often that I get the opportunity to write about a wide range of topics, some that I know only a little about, and picking only the parts that support the story I want to tell. My aim is to write in such a way that some of the content (at least Section 2 and the conclusion) will be accessible to almost everyone. I will focus on what computer science is, including a bit of history, and the interplay between theory and practice. I will also point out some of the common misconceptions of what the theory of computability and complexity tell us about practical problems, especially in relation to regular expressions.

1 Introduction

I will follow the unconventional approach of simply giving an outline of what I will discuss in this introduction, and having the real introduction in the next section.

In the next section I will start with a discussion on what computer science is since this is a widely misunderstood topic and of general interest. From this discussion, it will become clear that algorithms – what they can do and what not – and algorithmic efficiency issues, form a large part of computer science. After this, I will focus on the concept of ambiguity in both context-free grammars (a formalism used to describe the syntax of computer programs) and in regular expressions. The aim is to explain the interplay between computer science theory and programming, especially for regular expressions. Regular expressions (or more precisely, regular grammars) and context-free grammars are the simplest, and most useful, grammars in the Chomsky hierarchy [2, 5]. The Chomsky hierarchy is a containment hierarchy of classes of formal grammars studied within the fields of computer science and linguistics. Since some useful properties that context-free grammars might or might not have cannot be checked algorithmically, we will focus mainly on the more well-behaved regular expressions.

No general audience-focused discussion dealing with algorithms, and specifically algorithmic complexity, is complete without mentioning the famous open P versus NP problem. The P versus NP problem is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute, and which carries a million dollar prize [9]. It asks if problems solvable in non-deterministic polynomial time, i.e., NP problems (or equivalently, problems whose solutions are verifiable in polynomial time), are also solvable in polynomial time. I will thus also show the relevance of this open problem to regular expressions.

2 What is Computer Science?

In 2004, former US Secretary of Education Richard Riley predicted [14]: "The top 10 in-demand jobs in the future don't exist today. We are currently preparing students for jobs that don't yet exist, using technologies that haven't been invented, in order to solve problems we don't even know are problems yet."

Without mentioning computer science, this quotation might be enough to convince most people that computer science should from part of almost every degree programme at universities and other educational institutions. But it also adds to the overall confusion about ICT (informations and communications technology) and computer science. So what is the difference between these

two disciplines? ICT has a focus on spreadsheets, databases, Powerpoint, safety on the Internet, etc. It is currently the dominant of the two disciplines in terms of what is being taught at school and tertiary level.

Computer science, which has a focus on computation, algorithms, data structures, programs and skills in programming, communication and coordination, computational thinking, abstraction, modeling, design, etc., is barely taught. Every child learns, for example, science from primary school level onwards, not because all of them will become physicists or chemists, but because they are empowered by understanding how the world works around them. But the same is also true of computation and computer science in general (beware, many other disciplines might claim this falsely!) – it is a generic skill required in almost every work environment. We need it to understand both the digital and natural world.

To enforce the point, I quote from Sedgewick and Wayne [15]: "The basis for education in the last millennium was reading, writing, and arithmetic; now it is reading, writing, and computing." US President Barack Obama has also promoted the importance of programming with the statement [11]: "If we want America to stay on the cutting edge, we need young Americans to master the tools and technology that will change the way we do just about everything."

Given the focus on the importance of programming and computer science in general, it is interesting to note that the UK is currently (since September 2014) the only country having computer science as part of its school curriculum [12]. It begins at primary school, with a focus on fundamental principles and concepts of computer science, including abstraction, logic, algorithms, and data representation.

No discussion on what constitutes computer science is complete without a brief mention of Ada Lovelace, the only legitimate child of the poet Lord Byron, and commonly regarded as the first computer programmer. Ada realized that programming applies to (almost) any process based on logical symbols, and with Charles Babbage, we regard her as the first computer scientist. In February 1843, Charles Wheatstone, a friend of Babbage, suggested that Ada should produce an English translation of an article by Luigi Menabrea on Babbage's Analytical Engine, for Taylor's *Scientific Memoirs*. The translated article, with notes added by Ada Lovelace, was published in September 1843.

Lovelace's notes were labeled from A to G, and Note G (see Figure 2) gave an algorithm to compute Bernoulli numbers – describing step by step how the algorithm is fed into the Analytical Engine, including two recursive loops. The published program was a numbered list of coding instructions, which included destination registers, operations, and comments. The only help came from her spouse (William King-Noel, first Earl of Lovelace), who did not understand the mathematics or programming, but was willing to trace in ink what Ada wrote in pencil.

Now we move ahead 140 years and discuss how a cover page story published in *Time* magazine in 1984 links to algorithms and the theory of computability.

3 The Algorithm Shaving Those Algorithms Not Shaving Themselves

In 1984, *Time* magazine ran a cover story on computer software. They quoted the editor of a certain software magazine as saying: "Put the right kind of

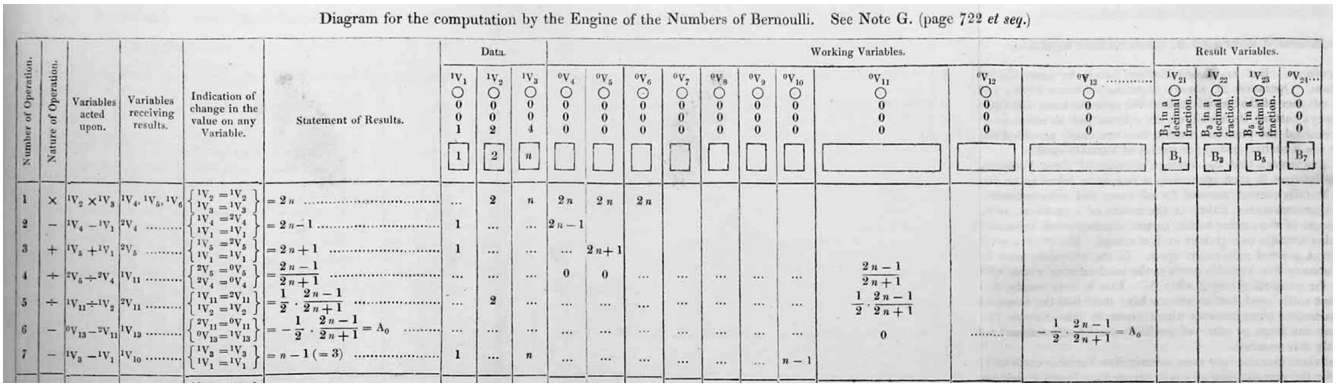

Figure 1: Ada Lovelace's diagram from Note G, describing an algorithm to compute Bernoulli numbers on Charles Babbage's Analytical Engine.

Algorithm 1 InverseHalt

```
1: function InverseHalt(program)
2:     if Halt(program, program) then
3:         go into infinite loop
4:     else
5:         return
```

software into a computer, and it will do whatever you want it to. There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software." That there are no limits to what software can do is of course not true, which follows more or less from the barber's paradox [8].

Although most people will be familiar with the barber's paradox, let us briefly recall it. Suppose a given town has just one barber, who is male. In this town, every man shaves and does so by either shaving himself or being shaved by the barber. Also, the barber shaves all those and only those men in town who do not shave themselves. So, who shaves the barber?

Let us rephrase this in terms of an algorithm which we will call InverseHalt, listed as Algorithm 1.

To define InverseHalt, we assume that we have an algorithm Halt(P, I) that returns TRUE if we run program P with input I, and P halts, and FALSE otherwise. So, what happens when we call the algorithm InverseHalt with itself as input? Will it halt or not?

It turns out that InverseHalt(InverseHalt) halts if Halt(InverseHalt, InverseHalt) returns FALSE, that is, if InverseHalt does *not* halt when given itself as input. Similarly, InverseHalt(InverseHalt) does not halt if Halt(InverseHalt, InverseHalt) returns TRUE, that is, if InverseHalt does halt when given itself as input.

To see the similarity with the barber's paradox, simply regard halting as shaving. Also, InverseHalt(*program*) halting is equivalent to the barber shaving *program*, and Halt(*program*, *program*) returns TRUE if *program* shaves itself, and FALSE otherwise. To avoid a contradiction, we have to assume that we cannot have a subroutine Halt(P, I) with the above-specified properties. This conclusion is summarized by saying that the halting problem is undecidable.

The self-referential nature of the halting prob-

lem is self-evident. The halting problem is also similar to Russell's paradox from set theory, which asks if the set of all sets that are not members of themselves, is a member of itself. Although the undecidability of the halting problem, or at least the presentation given above, seems rather esoteric in nature, similar ideas can be used to show that more practical problems are not solvable algorithmically. In fact, Rice's Theorem [13] states that there exists no automatic method that decides, with generality, non-trivial questions on the behavior of computer programs.

One such example, obtained from Rice's Theorem, by being imprecise and blurring the lines between programs and context-free grammars, deals with knowing if a particular context-free grammar (the formalism typically used to describe syntax in computer science), assigns at most a single syntactic description to a given input program. This property of context-free grammars is formulated by saying that it is undecidable if a given context-free grammar is ambiguous.

4 Ambiguity of Context-Free Grammars

In this section, we explain what it means for a context-free grammar to be ambiguous. Instead of using the well-known example of prepositional phrase attachment ambiguity from the Groucho Marx movie, *Animal Crackers* (1930) – "While hunting in Africa, I shot an elephant in my pajamas. How he got into my pajamas, I don't know" [18] – to explain the concept of ambiguity of context-free grammars, I will rather consider the sentence: "Alex ate sushi with chopsticks."

Consider the toy context-free grammar in Figure 4. Here $\langle NP \rangle$, $\langle VP \rangle$, $\langle DT \rangle$, $\langle PP \rangle$, $\langle V \rangle$, $\langle N \rangle$, and $\langle NNP \rangle$ denote a noun phrase, verb phrase, determiner, prepositional phrase, verb, noun, and proper noun, respectively. The grammar rules generate sentences by starting with $\langle S \rangle$, and by replacing the symbol appearing in the left-hand side of a rule by the right-hand side. Thus, the reason for calling this sentence generating mechanism context-free becomes clear – rules are applied without taking context into account.

$$
\begin{aligned}
\langle S \rangle &\rightarrow \langle NP \rangle \, \langle VP \rangle \\
\langle NP \rangle &\rightarrow \langle DT \rangle \, \langle NP \rangle \mid \langle NP \rangle \, \langle PP \rangle \mid \langle N \rangle \mid \langle NNP \rangle \\
\langle VP \rangle &\rightarrow \langle V \rangle \, \langle NP \rangle \, \langle PP \rangle \mid \langle V \rangle \, \langle NP \rangle \\
\langle PP \rangle &\rightarrow \langle P \rangle \, \langle NP \rangle \\
\langle N \rangle &\rightarrow \text{“}\texttt{boy}\text{”} \mid \text{“}\texttt{chopsticks}\text{”} \mid \text{“}\texttt{sushi}\text{”} \\
\langle NNP \rangle &\rightarrow \text{“}\texttt{Alex}\text{”} \\
\langle V \rangle &\rightarrow \text{“}\texttt{ate}\text{”} \\
\langle P \rangle &\rightarrow \text{“}\texttt{with}\text{”} \\
\langle DT \rangle &\rightarrow \text{“}\texttt{the}\text{”}
\end{aligned}
$$

Figure 2: A toy context-free grammar.

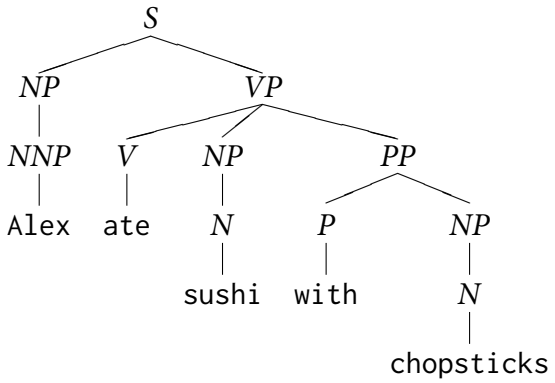

Figure 3: The intended parse tree for the sentence: "Alex ate sushi with chopsticks."

In our particular case, the given grammar is ambiguous, since the parse trees given in Figures 3 and 4 provide two ways of assigning syntax to the

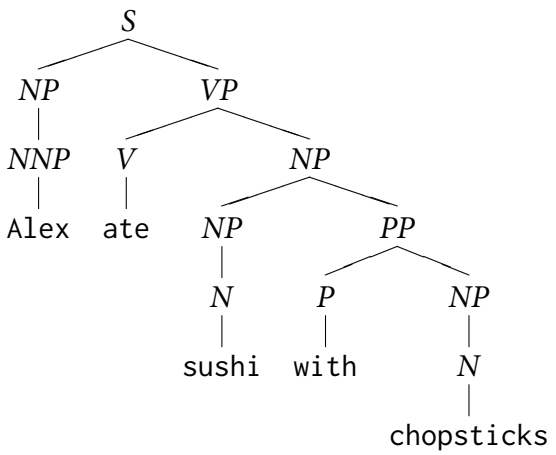

Figure 4: An unlikely parse tree for the sentence: "Alex ate sushi with chopsticks."

sentence: "Alex ate sushi with chopsticks." Since no algorithm can tell us in general whether a context-free grammar is ambiguous or not, and since some context-free grammars are inherently ambiguous – that is, they cannot be replaced by equivalent unambiguous grammars – the context-free grammar formalism is not as useful as theoreticians might want us to believe. In practice, there are two general approaches to deal with the ambiguity issue of context-free grammars. In many cases, such as in the case of parsing a computer program, fairly ad hoc mechanisms are used to select a given parse from the potentially multiple ways of assigning syntax to given input. Alternatively, a formalism that is never ambiguous is used, such as the fairly recent notion of parsing expression grammars, developed by Bryan Ford around 2004 [6].

In the next section, we discuss ambiguity in the case of regular expressions. In this setting, ambiguity is not an undecidable property, but it has led to software implementations that are vulnerable to algorithmic complexity attacks. This vulnerability occurs because it could in practice take prohibitively long to try and match a string with a highly ambiguous regular expression (also known as an evil regular expression).

5 Examples of the Mismatch between Theory and Practice in Regular Expressions

In this section I give a flavor of some of my recent work on regular expressions, a formalism that one may describe as wildcards on steroids. In general, a regular expression is a special text string for describing a search pattern, used for searching through (usually textual) data.

For example, the regular expression `[0-9]+`, where `+` indicates "one or more times", makes it possible to search through a text document for any integer number. Other familiar examples include patterns to ensure that a user-supplied e-mail address (on a web form) is of a valid format, or that a chosen password is complicated enough. It is also extensively used in web scraping, i.e. getting computer programs to look for and capture text from online resources mentioning say a particular person or company.

I will describe on a high level the discrepancy between regular expressions in theory and practice, and how they nowadays seem to work most of the time in practice, and it is now simply a matter of also making them work in theory all the time. Jamie Zawinski, a well-known software developer, summarizes the use of regular expressions as follows [19]: "Some people, when confronted with a problem, think 'I know, I'll use regular expressions. Now they have two problems.' "

In the following three subsections, I give examples of situations where there is a big discrepancy between how regular expression are usually described, in theory, versus how they work in programming. In the first subsection, I briefly discuss

exponentially ambiguous regular expressions, also known as evil regular expressions. Evil regular expressions might lead to software that is vulnerable to denial of service attacks. In the next subsection, I discuss the link between back references in regular expressions and the P versus NP problem. Back references [7] are used to force a regular expression to repeat a previous submatch later in the same regular expressions. Finally, I discuss how regular expressions parse input during matching, in contrast to how theory regards regular expressions only as a pattern matching formalism.

5.1 Evil Regular Expressions

First, I give an example to explain ambiguity in regular expressions. Assume that the symbol ? indicates that a sub-pattern should be matched at most once. Then `(subpattern? subpattern?)` indicates that we want to match `subpattern` between zero and two times. But this regular expression is ambiguous, since if we want to match `subpattern` only once, we can do it either with the first or second `subpattern`. In this case, we can remove the ambiguity by rewriting the regular expression as `(subpattern subpattern?)?`.

Evil regular expressions are those where the worst-case ambiguity grows exponentially in the length of the input string we are trying to match. For example, consider `(ab)*|(a|b)*`, where `|` denotes "or", and `*` denotes "zero or more times". This regular expression has exponential ambiguity, since on an input string of the form ababab . . . abx, all possible ways of matching each of the input a's with each of the subexpressions `(ab)` or `(a|b)` are typically attempted by regular expression matching software. Note that `(ab)*|(a|b)*` and `(a|b)*` are not equivalent. The regular expression `(ab)*|(a|b)*` indicates if the input string is a sequence of pairs ab, and if not, it checks if it consists only of a's and b's.

The example of an evil regular expression, given above, was selected to keep the exposition short and easy. Readers that are hardcore programmers should convince themselves that the following regular expression for e-mail address validation, split over three lines for formatting reasons, is also evil:

```
email=([a-zA-Z0-9_\.\-])+\@
      (([a-zA-Z0-9\-])+\.)+
      ([a-zA-Z0-9]{2,4})+
```

5.2 P versus NP – Sometimes NP-Hard Problems Might Not Be That Hard

No general audience article on theoretical computer science will be complete without saying something about P versus NP. The P versus NP problem is a major unsolved problem in computer science. It asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.

Stephen Cook gave the precise statement of the P versus NP problem in 1971 [3]. It is worth having a look at [20], which collects links to papers that try to settle the P versus NP question (in either way). They list 107 fairly reputable papers that try to contribute to the P versus NP question. Among these papers, there is only one paper, by Mihalis Yannakakis, that has appeared in a peer-reviewed journal, and that shows that a certain approach to settling the P versus NP question will never work out.

In computability and computational complexity theory, a reduction is an algorithm for transforming one problem into another. It is used to show

that the second problem is at least as difficult as the first.

Below, I will give a slightly modified version of a reduction from SAT (the Boolean Satisfiability Problem) to the problem of deciding whether a regular expression (with back references) match a given input string. I obtained this reduction from Perl lover's webpage of Perl paraphernalia [10]. This reduction shows that matching with Perl-like regular expressions is computationally at least as hard as the deciding whether a boolean logic formula is satisfiable (which is NP-complete).

To make the discussion concrete, let us consider the following boolean logic formula:

$$\begin{aligned} F := &(x_1 \vee x_2 \vee \neg x_3) \\ &\wedge (x_1 \vee \neg x_2 \vee x_3) \\ &\wedge (\neg x_1 \vee \neg x_2 \vee x_3) \\ &\wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3). \end{aligned}$$

Note that $\vee$, $\wedge$ and $\neg$ denotes "or", "and", and "not", respectively. By setting x_1 to TRUE, and x_2 and x_3 to FALSE, we note that F is satisfiable.

We translate F to the string s and regular expression R (spread over five lines for typesetting reasons) given below. The relationship between F and the last four lines of R is fairly direct, and the reasons for picking s and the first line of R as below will become clear soon.

```
s = xxxxxx;xx,xx,xx,xx
R = (x|xx)(x|xx)(x|xx).*;
    (\1|\2|\3x),
    (\1|\2x|\3),
    (\1x|\2x|\3),
    (\1x|\2x|\3x)
```

Note that `.` denotes any character, and thus, `.*` denotes any sub(string).

Now we need to convince ourselves that the boolean logic formula F has a satisfying assignment precisely when the regular expression R can match the string s. Note that, in the regular expression R, `\1`, `\2`, and `\3` refer to what gets matched by each of the first three subexpressions `(x|xx)`, respectively. Also, given the position of the symbol `;`, and each of the three symbols `,` in s and R, respectively, we have that each of the subexpressions `(\1|\2|\3x)`, `(\1|\2x|\3)`, `(\1x|\2x|\3)` and `(\1x|\2x|\3x)` have to match the string `xx`. Furthermore, `xx` and `x` correspond to TRUE and FALSE, respectively.

By letting the first of the three consecutive subexpressions `(x|xx)` of R match `xx`, and letting the other two match `x` – i.e., `\1` becomes `xx`, and `\2` and `\3` both become `x`) – we note that each of

```
(\1|\2|\3x)
(\1|\2x|\3)
(\1x|\2x|\3)
(\1x|\2x|\3x)
```

can match `xx` (recall that `|` denotes "or"), which corresponds to being able to make each of four clauses in F TRUE.

Although regular expression matching (with back references) is thus at least as hard as an NP-complete problem, could it be implemented efficiently, even though the current state of the art matcher RE2 [4], developed at Google, has various shortcomings? Since regular expressions used in practice have only a small number of back references (i.e., subexpressions of the form `\1`, `\2`, etc.), it should be noted that the NP-hardness of regular expression matching in general does not exclude the possibility of efficient software implementations of regular expression matchers.

We encounter NP-hard problems often in real life and we should not believe the common misconception that these problems cannot be solved in practice. Although space limitations do not allow the discussion of these topics, parametrized complexity, approximation, or randomised algorithms should be used when we encounter NP-hard problems; see [16] for a gentle introduction.

5.3 Regular Expression Matching as a Form of Parsing

Few formal language research topics have greater practical reach than regular expressions. As a result, the practical implementations have in many ways surged ahead of research, with new features that require underpinnings different from the original theory. Most practical implementations of regular expressions matchers perform regular expression matching as a form of parsing, using *capturing groups*, outputting what subexpression matched which substring.

In Perl, we can for example match an email address and place the username and hostname in `$1` and `$2` as follows:

```
if ($email =~/([^@]+)@(.+)/){
    print "Username is $1\n";
    print "Hostname is $2\n";
}
```

A popular implementation strategy is a worst-case exponential-time depth-first search for matching and finding a correct way of parsing the input string. Complications are introduced by the matching semantics dictating a single output string for each input string, using rules to determine a "highest priority" match among the potentially exponentially many possible ones. In practice, regular expressions are used to match and parse input, but this is far more sophisticated than their theoretical counterparts. Some ideas on how to close this gap between theory and practice are given in [1].

In Figure 5, a Java-based state machine for the regular expression `(a*)*` is given. The regular expressions `(a*)*` will, of course, never be used, but it is simply chosen to keep the exposition as simple as possible. Also, note that the state machine in Figure 5 is an abstraction of how the approximately 10 000 lines of code in the Java regular expression matching library handles this particular regular expression. Analyzing the abstraction, and understanding the discrepancies between the abstraction and the real implementation, forms the crux of developing a proper understanding of regular expression matching libraries.

Next, we discuss the state machine in Figure 5 in more detail. State q_0 is the initial and q_7 the final state. Transitions not marked with any symbols, are taken without reading or outputting any symbols. Transition marked with "a" indicates that `a` is read from the input (and produced as output), and "[" or "]" on a transition indicates that the corresponding bracket is produced as output. The dashed lines indicate lower priority transitions – they are only taken if the higher priority transitions do not lead to acceptance.

The accepting path for the string a^n – i.e., a string of length n having only `a`'s – in the state machine in Figure 5, is $q_0q_1(q_2aq_1)^nq_0q_3$; note that the exponent "n" simply indicates that the particular substring should be repeated n times. Since there are, for the input strings a^n, exponentially many paths in this state machine, a regular expression matcher using an input-directed depth-first search (without memoization as in Perl), such as the Java implementation, will take exponential time in attempting to

match the strings $a^n x$, for $n \geq 0$.

On an abstract level, a state machine associated with a regular expression should be regarded as a function. The domain of this function is a subset of all possible strings and is equal to the classical interpretation of a regular expression. The function values are defined in such a way to indicate which subexpression captured which substring(s). For the given regular expression `(a*)*`, the domain is a^*, and each string a^n in the domain is mapped to $[a^n]$. This indicates that the outer `*` in `(a*)*` is used only once, and it corresponds to the parse that is chosen by all implementations in such ambiguous cases.

Next we provide more small and slightly artificial examples of regular expressions to point out additional differences between regular expressions in theory and practice. The regular expressions R = `(a)(a*)` and R' = `(a*)(a)` are equivalent in the traditional sense, but in practice, the capturing happens in different ways. An input string a^n from a^+ (i.e., one or more a's) is captured by R as $[_1 a]_1 [_2 a^{n-1}]_2$, where as R' captures it as $[_1 a^{n-1}]_1 [_2 a]_2$. Also note that the same subexpression in a regular expression may capture more than one substring, for example, for $(a^* \mid b)^*$ the input string $a^p b^q a^r$ is captured as $[a^p][b] \ldots [b][a^r]$, for $p, q, r > 0$.

We conclude this section on regular expressions by discussing what happens when software implementations do not drive theory and standards. Regular expressions are standardised in the IEEE POSIX[1] standard for regular expressions. But the real standard has become the implementation of the regular expression matching library in the Perl programming language. Larry Wall, author of the Perl programming language, describes this situation as follows [17]: "Perl regular expressions are only marginally related to other regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here. I will, however, generally call them [Perl regular expressions] regexes, or regexen, when I'm in an Anglo-Saxon mood".

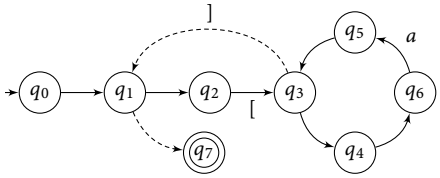

Figure 5: Java based state machine for the regular expression `(a*)*`; dashed edges indicate lower priority transitions.

6 Conclusions

My main aim was to show that the disconnect between theory and practice provides ample opportunity for interesting and useful research. In the case of regular expression matching, theory in isolation loses out on key features and relevance, whereas in the case of software, practical implementations lose consistent semantics and often have unpredictable matching time. A similar relationship between theory and practice is, of course, also relevant to many other disciplines. Finding the right mix between theory and practice is, likewise, essential when designing academic programs at universities (and other educational institutions), and when coming up with strategies to build successful and relevant research teams.

[1] Institute of Electrical and Electronics Engineers Portable Operating System Interface, ISO/IEC/IEEE 9945:2009.

References

[1] Martin Berglund and Brink van der Merwe. On the semantics of regular expression parsing in the wild. In Frank Drewes, editor, *Implementation and Application of Automata – 20th International Conference, CIAA 2015, Umeå, Sweden, August 18–21, 2015, Proceedings*, volume 9223 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2015.

[2] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[4] Russ Cox. Implementing regular expressions. `http://swtch.com/~rsc/regexp/`, 2007. Accessed March 3, 2015.

[5] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages (2nd Ed.): Fundamentals of Theoretical Computer Science*. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[6] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14–16, 2004*, pages 111–122. ACM, 2004.

[7] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[8] David Harel. *Computers Ltd.: What They Really Can't Do*. Oxford University Press, Inc., New York, NY, USA, 2000.

[9] Clay Mathematics Institute. Millennium problems. `http://www.claymath.org/millennium-problems`. Accessed September 15, 2015.

[10] Perl Lover. Perl regular expression matching is NP-hard. `http://perl.plover.com/NPC/`. Accessed September 15, 2015.

[11] Wired Magazine. Obama says everyone should learn how to hack. `http://www.wired.com/2013/12/obama-code/`. Accessed September 15, 2015.

[12] UK Department of Education. England national curriculum: computing programmes of study. `https://www.gov.uk/government/publications`. Accessed September 15, 2015.

[13] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[14] Kathryn Scanland, Steve Gu, and Roberts Jones. *The Jobs Revolution: Changing How America Works*. Copywriters Inc., 2004.

[15] Robert Sedgewick and Kevin Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison-Wesley Publishing Company, USA, 1st edition, 2007.

[16] Udacity. Intro to theoretical computer science: Dealing with challenging problems. `https://www.udacity.com`. Accessed September 15, 2015.

[17] Larry Wall. Apocalypse 5: Pattern matching. `http://www.perl6.org/archive/doc/design/apo/A05.html`. Accessed September 15, 2015.

[18] Wikipedia. Animal crackers (1930 film). `https://en.wikipedia.org/wiki/Animal_Crackers_(1930_film)`. Accessed September 15, 2015.

[19] Wikiquote. Zawinski quotes. `https://en.wikiquote.org/wiki/Jamie_Zawinski`. Accessed September 15, 2015.

[20] GJ Woeginger. The P-versus-NP page. `http://www.win.tue.nl/~gwoegi/P-versus-NP.htm`. Accessed September 15, 2015.