The Computational Complexity Column

BY

Michal Koucký

Computer Science Institute, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
koucky@iuuk.mff.cuni.cz
https://iuuk.mff.cuni.cz/~koucky/

Automata and Formal Languages: Shall we let them go?

Michal Koucký
Computer Science Institute
Charles University, Prague
koucky@iuuk.mff.cuni.cz

Abstract

In this article I give my thoughts on the role of automata and formal languages in our computer science curriculum.

1 Introduction

This article is a reflection of my thoughts on the content of the course on automata and formal languages at Charles university in Prague. Similar courses are a mandatory part of computer science curricula at many other universities around the globe. Four years ago I was asked by our math department to redesign their graduate level course *Automata and computational complexity*. As the name suggests traditionally that course contains a large portion of automata and formal language theory. After discussions with the head of the math department I realized that they do not necessarily care for any particular topic what they care for is that the course covers theoretical foundations of computer science. So I redesigned their course to match my view of foundations of computer science.

As you might expect from the title of this article the role of automata diminished substantially in the new course. I will come back to that new course later. However, this prompted me to take a fresh look at our own computer science course *Automata and grammars*. I believe the time has come to let the automata go and replace the content of the course by what it perhaps always meant to be: theoretical foundations of computer science.

Next I will briefly review the origins of the current course and I will try to put it into historical context of development of computer science over past 90 years. Then I will propose what should be covered in a modern course on theoretical foundations of computer science, and I will go over some hurdles which one might encounter when trying to modify the course.

1.1 Automata and grammars

Our course on automata and formal languages was designed more than 40 years ago and it is largely based on the classical book by Hopcroft and Ullman [8], and its Czech cousin by Chytil [2]. Over the years the course underwent various updates and modifications as the allotted time for the course varied. Today the course mostly follows Sipser's book [10] or the current version of Hopcroft-Ullman [6]. However, the core focus of the course remains the same: automata, grammars and languages recognized by those models.

The course used to be accompanied by a course on recursion theory and later also by a course on computational complexity. The two latter courses moved to graduate level courses during the past 20 years and were substantially overhauled.[1] The course *Automata and grammars* remains mostly unchanged with its focus on automata, grammars and classification of problems according to the Chomsky hierarchy.

My view is that the course was originally designed to reflect then-current knowledge of theoretical foundations of computer science and complexity theory. Let us briefly review the historical context of its origins and development in theory of computing over the years.

1.2 Brief history of modern theory of computing

Here, I am going to present my personal take on history of theoretical computer science. It might not be perfectly accurate but it should be approximately correct. With few exceptions I will ignore the names of many great computer scientists who contributed to this development so I will focus only on the main ideas. A thorough historical account of development of computability is given in the book by Soare [11]. Development of complexity theory is covered in the article by Fortnow and Homer [4].

The development of modern computer science was instigated by logicians. Their program from the turn of the 20th century summarized by Hilbert asked whether mathematical truth can be established by mechanical means. Those questions led to development of models for mechanical procedures: In 1931-34, Gödel proposed definition of recursive functions and Church proposed his λ-calculus, then in 1936 Turing defined what we call *Turing machines* [12]. All those models were quickly established to be equivalent. Arguably, a Turing machine is the right model to capture computation, and it allowed for the development of theory of computation as we know it today. Considerations about what problems are algorithmically solvable lead to development of *computability theory* (*recursion theory*).

[1] In line with European-wide changes to university systems, after the year 2000 our originally five year program was divided into a three year bachelor program and a two year master program.

The recursion theory is concerned with what can be computed by an algorithm and what cannot be computed by an algorithm. A prototypical uncomputable problem is the *Halting problem*.

Soon after development of the concept of computability many people realized that not all algorithmically solvable problems are born equal: Some problems are harder to solve than others, they are more *complex*, their computation might require more steps to complete. This aspect was famously referred to by Gödel in *Gödel's lost letter*.

A simple tool to gauge the complexity of a computational problem is provided by a finite automaton. Finite automata were proposed in 1940's. Motivated by parsing human languages and programming languages, over the next three decades finite automata flourished into a rich theory for formal language classification: from finite automata, to automata with multiple heads, marking automata, push-down automata, etc. This development is captured till large extent by the book of Hopcroft and Ullman [8].

One of the central concepts in this area, the *Chomsky hierarchy*, was formulated in the late 50's [1, 3]. Chomsky hierarchy provides a tool to classify computational problems into easy to solve: *regular languages*, moderate: *context-free languages*, harder: *context-sensitive* and hardest: *recursively-enumerable*. This is a crude classification of their *computational complexity*.

This is the development which is covered by the typical courses on automata and formal languages.

The 1960's saw the origins of a different approach to classification of problems into easy and hard to solve: Hartmanis and Stearns [7] defined space and time complexity, and proved basic hierarchy theorems. This was in the context of nascent *computational complexity theory*. This theory took a central stage with the introduction of NP-completeness in the 1970's. During that decade, the notions of *efficient algorithms*, *complexity measures* and *complexity classes* took a firm hold in theoretical computer science. In its generality those concepts are the focus of *structural complexity theory*. For concrete algorithmic problems this is the focus of *algorithm and data structure design*.

The new point of view stimulated rapid development of new algorithmic techniques and data structures. The complexity approach also laid foundations for modern cryptography. The introduction of public key cryptography in the 1970's in connection with complexity theory led to modern day theoretical cryptography. The 1980's saw development of circuit complexity (studied in the Soviet Union already in 60's and 70's) The late 1980's and early 90's gave us interactive proof-systems, zero-knowledge proofs and the PCP Theorem. This stimulated the development of non-approximability and approximation algorithms in the 1990's and after 2000.

The class P (polynomial time) was established as the equivalent of efficient computation already in the 1970's. Despite many of its desirable properties (e.g.

closeness under poly-time reductions) not everyone was happy with that definition. The late 1990's saw first incursions into sub-linear time algorithms which morfed into the area of property-testing. Concurrently, massive amounts of data that needed to be processed led to the notion of streaming algorithms which blossomed in the first two decades of the 21st century. Streaming algorithms are on some level similar to finite automata: they perform one pass over the data and they allow limited but non-constant memory. However, they go hand in hand with relaxing the requirement for correctness as they allow the answer to be only approximately correct and typically they are randomized.

Late in the first decade of the 21st century new area emerged: fine-grained complexity. The fine-grained complexity pushes the realm of *efficiently computable* closer to usual algorithm design. It establishes very efficient reductions between various concrete problems of interest. It also links the difficulty of algorithm design for NP-complete problems such as Satisfiability with the difficulty of improving algorithms for ordinary problems in P such as All-Pairs Shortest Paths.

All those areas including automata theory are active to this day although their focus has shifted in new directions, and the mainstream of theoretical computer science has changed since the 1970's. The main take-away message from the past 90 years of development of theory of computation is the quest to capture what is and what is not efficiently solvable. The notion of *efficiency* is not static. It progressed from being computable (recursion theory), to being in P (complexity theory), to being linear or quadratic (streaming algorithms and fine-grained complexity).

Arguably the course on automata and formal languages should reflect the progress of our understanding.

2 Foundations of computer science - new syllabus

Here I will present my take on what a redesigned course should focus on. On a high level the focus should stay the same as before, the course should introduce the following ideas: There are problems that can be solved by a computer, and there are problems that cannot be solved by a computer (*computability*). Some problems that can be solved by a computer are easier to solve than others (*complexity*). Those two points should be the primary focus. Here is the syllabus of my *Automata and computational complexity* course I designed for the math department [9]:

1. Computational models: computer, RAM, Turing machine, Boolean circuit.

2. Undecidable problems, Halting problem, reductions.

3. Time complexity, class P.

4. Class NP, NP-hardness, NP-completeness, Cook-Levin Theorem.

5. Space complexity, class PSpace, PSpace-complete problem QBF, Polynomial Hierarchy.

6. Class Log, *s*-*t*-Connectivity, Savitch's Theorem.

7. Finite automata, regular languages.

8. Hierarchy Theorems, fine-grained complexity.

To start, any argument about computability or complexity needs a rigorous definition of an algorithm, so we need models of computation: *Turing machines*, *RAM*, *their equivalence*. Once we prove that the *Halting Problem* is uncomputable it is useful to extend the result to other problems via *reductions*. Then *time complexity* should come in the picture together with the *class* P representing efficiently computable problems. Afterwards we can move to the *class* NP defined as a class of problems for which we can efficiently verify solutions but we may not know how to find their solutions efficiently. NP-*hardness* via *polynomial time reductions* is a natural next step. This sets the stage for NP-*completeness* of *Satisfiability (*SAT*)* and the *Cook-Levin Theorem*.

Space is another resource which one cares about so we should move on to space complexity: *the class* PSpace with the *complete problem Quantified Boolean Formulas (*QBF*)* as a natural generalization of SAT. By restricting the number of quantifier alternations in QBF formulas we get the Σ_k-SAT, and the *levels of the Polynomial Hierarchy* as the classes of problems reducible to the Σ_k-SAT by polynomial time reductions.

PSpace contains the whole Polynomial Hierarchy and especially NP, so it contains problems that we do not know how to solve efficiently. So we should turn our focus to small space algorithms, the *class* Log of problems solvable in logarithmic space. Arguing about problems in Log requires *log-space reductions* which can be *composed*. A complete problem for Log is the *s*-*t*-Connectivity on undirected graphs (without giving Reingold's algorithm). One can venture into the *non-deterministic log-space (*NLog*)* as the class of problems log-space reducible to *s*-*t*-Connectivity on *directed* graphs. Once there it makes sense to show *Savitch's Theorem* that *s*-*t*-Connectivity can be solved in space $O(\log^2 n)$.

At this point we can go even further with restricting space: we get the class of problems solvable in *constant space*. Constant space on Turing machines is equivalent to *no-space* as we can push the content of the tapes into the state of the Turing machine. Problems decidable by a Turing machine with no-space can be decided by a no-space Turing machine which moves its input head only in one direction: *finite automaton*. Now we reached the well known class of *regular*

languages and our journey downwards stops here. We can show that the language 0^n1^n is *not regular* (*without* using the Pumping Lemma.)

Since we defined all the complexity classes we can compare them to each other by means of *Time Hierarchy* and *Space Hierarchy Theorems*. An excursion into *fine-grained complexity* is a natural next step: a connection between the hardness of improving algorithms for NP-complete problems and problems in P is one of the most enlightening discoveries of the past few decades. My favorite is the *reduction from* SAT *to the Orthogonal Vector Problem*. It is easy and it gets the point across.

Optionally, one can throw in *Boolean circuits* as the model of non-uniform algorithms. This is a model which many students are already familiar with. They know Boolean circuits because they model real hardware, and because of the special case: neural networks.

Nondeterminism. One can completely avoid talking about non-deterministic computation per-se. Personally I am not sure whether we should teach *all* students about the abstract construct which is the non-deterministic computation. If I need to present this concept to my students in my graduate level classes I like to start with randomized computation which is more natural and realistic. Students are usually familiar with randomized computation because they know some randomized algorithms for particular problems. For beginners I prefer the presentation of NP purely using the efficient verification paradigm (see e.g. the textbook by [5]). Other non-deterministic classes (Polynomial Hierarchy, NLOG) can be presented as the classes of problems efficiently reducible to their respective complete problems.

Comparison with Automata and formal languages. Arguably, the high-level structure of current courses on automata and formal languages is the same as that of the proposed syllabus. The traditional course focuses on *recognition of languages* by various models of computation with limits on their computational resources. With respect to the new syllabus the only difference is the choice of restrictions on those resources, the choice of the models, and going top-down instead of bottom-up.

The focus on the current computational complexity will align the course with other course such as on algorithm and data structure design. Those courses revolve around design of efficient algorithms and data structures with respect to time and space complexity where we are concerned with the asymptotic behaviour of the two measures. This is at odds with the Chomsky hierarchy, the classification of problems according to what type of automata recognizes them (finite automata/push-down automata/Turing machines).

3 Adopting a new syllabus

One of the technical hurdles to adopting a new syllabus is the availability of an appropriate textbook. The best current textbook to cover the new syllabus could still be the book by Sipser [10] if one skips Chapters 1 and 2. However, the book in its current form is not ideal as there are many topics and exercises in the later chapters that are concerned with a bit artificial problems on automata from earlier chapters.

Another hurdle is the momentum of the education system. For better or worse, university environment is a rather conservative place with respect to modification of its curricula. There are competing interests of various parties, and different courses are intertwined. Some of the concepts covered by the classical automata and formal languages courses are relied upon in particular branches of computer science.

Natural language processing historically relied upon grammars although most of the current system rely on deep-neural networks. Similarly, theory of programming languages relies on grammars although compilers and interpreters for many current languages do not use them for parsing directly. The *notion* of finite automaton is useful for software engineering to model systems which can be in restricted number of distinct states. The same is true for description of cryptographic primitives and network protocols. This has an impact on the area of software and system verification. Niche applications of regular expressions are in some text editors for searching and in system programming for rule specification. However, none of those latter applications relies on the ability of finite automata to recognize precisely regular languages which is the primary focus of the classical courses on automata and formal languages.

So there are many areas which rely to some degree on the notions covered by the course but perhaps they do not care so much about the closure of regular languages under concatenation, union, intersection, etc. So they could perhaps be satisfied with much more modest coverage of the topics.

Acknowledgements

I love finite automata and the beautiful theory surrounding them. I was introduced to it by Michal Chytil in the very course I am discussing in this article. My master thesis done under the guidance of late Vašek Koubek deals with the hierarchy of marking one-way multi-head finite automata. I also love recursion theory. This was sparked by passionate and thoroughly enjoyable lectures of Antonín Kučera. I owe him for a discussion on the history of computability and a pointer to the book by Soare [11]. Many years ago, Antonín Kučera in his intellectual honesty dissuaded

me from pursuing recursion theory for my doctoral studies and suggested to focus my attention elsewhere. I know I wasn't the only one receiving that advice from him. I leave the reader with a question: *How well do we serve our students if we teach them a theory that they cannot develop substantially further?*

References

[1] Noam Chomsky. Systems of syntactic analysis. *J. Symb. Log.*, 18(3):242–256, 1953.

[2] Michal Chytil. *Automaty a gramatiky*. SNTL, 1984.

[3] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. Elsevier, 1963.

[4] Lance Fortnow and Steven Homer. A short history of computational complexity. *Bull. EATCS*, 80:95–133, 2003.

[5] Oded Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition, 2nd Edition*. Addison-Wesley, 2003.

[7] Juris Hartmanis and Richard Edwin Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[8] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1969.

[9] Michal Koucký. NMMB415: Automata and computational complexity. `https://iuuk.mff.cuni.cz/~koucky/vyuka/AVS-ZS2021/index.html`, 2021.

[10] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[11] Robert I. Soare. *Turing Computability - Theory and Applications*. Theory and Applications of Computability. Springer, 2016.

[12] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1936.