Automating Problem Creation

Benjamin Darnell
Faculty Advisor: Dr. Ben Hardekopf
Department of Computer Science, University of California, Santa Barbara

Background

At UCSB, computer science undergraduates are required to take CMPSC 138, which explores various classes of decision problems. Decision problems are essentially questions that can be answered either 'yes' or 'no', such as "is the number n prime?" or "does the string s contain the substring 'ab' at least twice?". Students are asked to represent these decision problems using mathematical models, or automata, which can be thought of as a simple program.

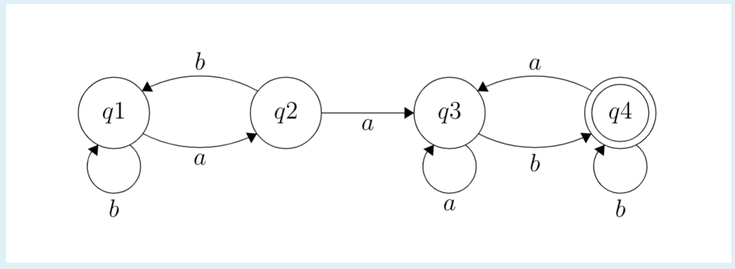


Fig. 1: An automaton for the language "strings that contain 'aa' and end with 'b'".

Challenges

The aim is to create a method for automatically generating construction problems for DFA's, a subset of regular automata, to be used for homework and exam problems in classrooms. The system for problem generation should be robust, producing many reasonable and unique problems, and configurable with respect to verbosity and difficulty level of the problems. A successful tool would have to consistently generate output that meet these five criteria:

- **Regularity**: Any generated problems must be guaranteed to be regular languages. Otherwise, a DFA that recognizes the language cannot exist, so there wouldn't be a valid solution.
- **Variability**: The problems must be both unique and highly variable so the instructor would have a large set of options to choose from. Problems with small differences are not useful since they commonly share similar solutions.
- **Difficulty**: The metric of 'difficulty' is how challenging the problem would be for a student to solve, irrespective of syntax. Hence, the difficulty score is mostly drawn from the features of the DFA itself.
- **Reasonability**: The metric of 'reasonability' is loosely defined as how much the instructor would like to use the problem. I believe that this score would be higher when the textual description is concise and easily comprehensible, and when the difficulty of the problem falls closer to the desired value.
- **Natural Language Description**: There needs to be a suitable English description to the problem, one that is both unambiguous and easy for students to comprehend.

Methodology

While regular languages can be represented using DFA's, this does not nicely translate to a concise problem description. Instead, I used MOSEL, a syntax for monadic second-order logic, which is a much more faithful representation of the problem's structure. Crucially, any valid MOSEL formula is guaranteed to be regular. A MOSEL syntax can be programmatically generated using known techniques, which would then be converted into a DFA using a pre-existing tool. The structure of the MOSEL syntax also lends itself well to creating a natural language description, which can be done inductively.

Natural Language Description

The number of occurrences of the substring "0" is even and it is not true that the string begins with "10".

Fig. 2: Description of an example DFA construction problem.

MOSEL Syntax Tree

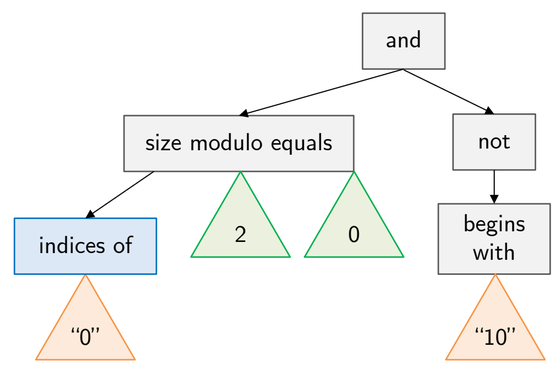
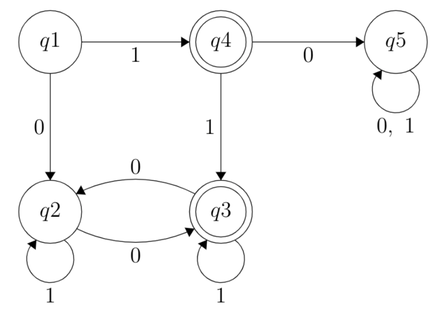


Fig. 3: MOSEL syntax tree for fig. 2, color coded by node type.

Equivalent DFA



Syntactic Complexity: 1.4
Difficulty: 8

Fig. 4: DFA for the problem in fig. 2, converted with MONA.

1) First, a MOSEL abstract syntax tree is generated using fuzzing techniques. Each node is randomly selected from a weighted list of options, obeying contextual constraints. The generated AST is refined by adding generated constants, removing duplicates, and simplifying redundant syntaxes. Each formula is procedurally converted into its textual representation.

2) Next, the MOSEL syntax is converted into pure monadic second-order logic, to serve as input for the MONA tool. Lower-level formulae are not changed, but higher-level types, like 'contains aba' or 'size mod 3 = 1' are recursively translated into their MSO representations. The tree is converted into a MONA program and run, outputting a DFA.

3) The resulting DFA is analyzed and scores are assigned for "difficulty" and "reasonability", based on the complexity of the syntax and the resultant DFA, respectively. The problem is kept if it falls within the desired difficulty threshold. The set of remaining problems are presented to the user in order of their reasonability, giving their description and DFA solution.

Why is this important?

In most iterations of this course, the instructor must either use problems found in a limited set of textbooks or spend time to create their own. Using textbook problems for assignments or exams is not ideal, since answer keys are easily available online. This dilemma can be overcome by automatically generating the problems. Potential upsides of programmatic generation include:

- Efficiency, the professor being able to create many problems in a short amount of time,
- Uniqueness, preventing students from sharing answers or using a published key,
- Customizability, as individual students can receive problems more directly tailored to their needs.

For this project, I chose to focus on a certain type of problem, DFA construction problems, which give a textual description of a language and ask students to create a DFA (deterministic finite-state automaton) that accepts it. The goal is to create a tool that can automatically generate DFA construction problems that instructors can give to students. The instructor will use this tool by specifying the number of problems to generate and a level of difficulty, and the tool would return a list of problems from which they can select their favorites.

Results

Using the methods outlined above, I created a command-line tool that can generate a specified number of problems in a range of target difficulties, addressing each of the challenges. Regularity is enforced by the MOSEL syntax, and duplicates are removed at multiple stages to ensure differences between each problem. Reasonability and difficulty scores are calculated for each problem, which is rejected if it falls outside a limited range. During testing, the tool generated hundreds of problems per minute, many times faster than a human.

The quality of generated problems appears to be on-par with handwritten ones, but further studies should be done to verify this claim.

```
>./dfagen -n 2 -d beginner -v -a
1: the string has an odd length
(length mod 2 = 1)
alphabet: {0, 1}
start: 0
accepting: {1}
1 (0 -> 0) (1 -> 0)
0 (0 -> 1) (1 -> 1)

2: every '1' is immediately preceded by a '0'
('0' precedes '1')
alphabet: {0, 1}
start: 0
accepting: {1, 0}
0 (1 -> 2) (0 -> 1)
1 (1 -> 0) (0 -> 1)
2 (1 -> 2) (0 -> 2)
```

Fig. 5: Sample output from the command-line tool. Here, the user requests 2 problems at a beginner difficulty level.

Conclusion

The process of generating MOSEL formulae using fuzzing techniques appears to yield results comparable in quality to problems written by humans and requires minimal effort from instructors. It may be possible to generalize this process to other types of construction problems, like NFA's, PDA's or Turing machines. In subsequent research, I would like to employ these methods in CMPSC 138, and conduct user studies to refine and verify the generation process.

Literature Cited

- Alur, R., D'Antoni, L., Gulwani, S., Kini, D. and Viswanathan, M., Automated Grading of DFA Constructions.
- Shenoy, V., Aparanji, U., K., S. and Kumar, V., 2016. Generating DFA Construction Problems Automatically. International Conference on Learning and Teaching in Computing and Engineering,.

Acknowledgements

My CCS Summer Research Project was possible due to the generosity of donors to The College of Creative Studies Computer Science Endowment.

I would also like to thank my research advisor, Ben Hardekopf, for his guidance and support.