Combinatorial Problem Solving (CPS)

Project: Minimum Consistent Finite Automata.

Updated: March 12, 2025

1 Description of the Problem

A *(deterministic) finite automaton* is a directed graph in which vertices are called *states* (in the example shown in Figure 1 there are 3 states, namely q_0, q_1 and q_2). From each state there are exactly two outgoing arcs, called *transitions* in this context, one labelled with a 0, and the other labelled with a 1. Among the states, there is one which is called the *initial state* (in the example, the state q_0, which is indicated with the only unlabelled arrow). Finally, states have an additional attribute: they can be *accepting* or not. In particular, the initial state can be accepting or not (in the example, it is the only accepting state, which is indicated with a double circle).

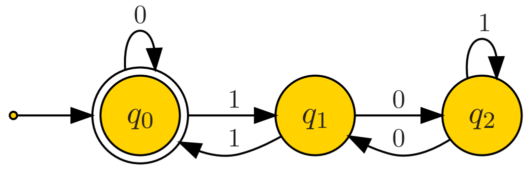

Figure 1: Example of a finite automaton.

A (binary) sequence (that is, a sequence of bits, either 0 or 1) is *accepted* by the finite automaton if, starting at the initial state and moving along transitions according to the bits in the sequence, at the end of the sequence we are at an accepting state.

For instance, the sequence 011 is accepted by the automaton in the example: We start at state q_0. Then we move to state q_0 (since we read the first bit of the sequence, 0, and the transition from state q_0 labelled with 0 goes back to q_0). Then we move to state q_1 (since we read the second bit of the sequence, 1, and the transition from state q_0 labelled with 1 goes to state q_1). Finally we move to state q_0 (since we read the third and last bit of the sequence, 1, and the transition from state q_1 labelled with 1 goes to state q_0). Since state q_0 is accepting, the sequence 011 is accepted.

However, the sequence 10 is not accepted by the automaton. We start at state q_0. Then we move to state q_1 (since we read the first bit of the sequence, 1, and the transition from state q_0 labelled with 1 goes to q_1). Then we move to state q_2 (since we read the second and last bit of the sequence, 0, and the transition from state q_1 labelled with 0 goes to state q_2). Since state q_2 is not accepting, the sequence 10 is rejected.

The *minimum consistent finite automaton problem* is that of finding a finite automaton with as few states as possible that is consistent with a given sample (a finite collection of sequences, each labeled as to whether the automaton should accept or reject). That is, given sets $A, R \subseteq \{0,1\}^*$ of sequences such that $A \cap R = \emptyset$, the goal is to find an automaton with the minimum number of states that accepts all sequences in A, and rejects all sequences in R. The behaviour of the automaton on sequences not in A and not in R is unspecified, and therefore free to decide.

2 Input and Output Formats

This section describes the format in which instances are written (Section 2.1), as well as the expected format for the corresponding solutions (Section 2.2).

2.1 Input Format

An instance is a text file consisting of several lines of integer values separated by blank spaces. It has two blocks. In the first one, the heading line contains $|A|$, the number of sequences to be accepted. Then $|A|$ lines follow, each corresponding to a sequence $w \in A$, with the following format: first, $m = |w| \geq 0$, the length of w; and then the m bits w_0, w_1, …, w_{m-1} of $w = (w_0, w_1, \ldots, w_{m-1})$. In the second block the structure is the same, but applied to the set R of sequences to be rejected instead of A.

It is ensured that the sequences from each of the lines are different. In particular, $A \cap R = \emptyset$. Note that sequences may have different lengths, and that the empty sequence is a valid sequence.

For example, Figure 2 shows the data of an instance.

```
4
 0
 1     0
 2     0 0
 2     1 0

3
 1     1
 2     0 1
 2     1 1
```

Figure 2: Input file of an instance.

Notice that, in this case, a sequence of length at most 2 should be accepted if and only if, when viewed as the binary representation of a natural number, that number is a multiple of 2.

2.2 Output Format

Output files only contain integer values. The output starts with a copy of the input data (this is convenient to make a sanity check of the solutions, see below). The next line is the minimum number of states n. Then n lines follow with the description of the optimal automaton. If we represent the states by q_0, q_1, …, q_{n-1}, then the i-th line (where $0 \leq i < n$) corresponds to q_i, and contains three numbers j, k and a, meaning that:

- when reading 0 from q_i the automaton moves to state q_j;
- when reading 1 from q_i the automaton moves to state q_k; and
- $a = 1$ if and only if state q_i is accepting.

It will be assumed that the initial state of the automaton is state q_0.

For example, Figure 3 shows an optimal finite automaton for the instance in Figure 2, and the corresponding output file.

Notice that, in this solution, intuitively state q_0 means "*the number read so far is even*", and state q_1 means "*the number read so far is odd*".

```
4
0
1    0
2    0 0
2    1 0

3
1    1
2    0 1
2    1 1

2
0 1 1
0 1 0
```

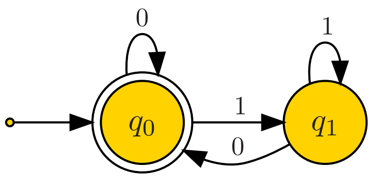

Figure 3: Output file of the solution and its graphical representation.

3 Project

The purpose of this project is to model and solve the proposed problem with the three technologies considered in the course: constraint programming (CP), linear programming (LP) and propositional satisfiability (SAT).

For the development of the project, in addition to this document, students will be provided with the following materials:

- a suite of *problem instances* (in the format specified in Section 2.1).
- a *checker* that reads the output of solving a problem instance (following the format given in Section 2.2) from `stdin`, makes a sanity check and plots the solution in PNG format. Take into account that **the checker does not guarantee that the solution is optimal**. It just checks that the solution satisfies the required constraints. So there may be other solutions with better value of the objective function.

  Note: GraphViz tools are required for plotting. Although this software is already installed in the lab machines, it can also be easily installed in personal laptops. E.g., in Ubuntu, one just needs to type in a terminal:

  ```
  sudo apt-get install graphviz
  ```

- a *results table* with the optimal value of the objective function for some of the problem instances, to make debugging easier.

As a reference, solving processes that exceed a time limit of 60 seconds of wall clock time should be aborted (`Linux` command `timeout` may be useful). Take into account that, depending on the technology, on the machine, etc., some of the instances may be too difficult to solve within this time limit. For this reason, it is not strictly necessary to succeed in solving all instances to pass the project. However, of course you are encouraged to solve as many instances as possible, and the more instances you can solve, the higher the grade.

The project has three deadlines, one for each technology:

- **CP:** 17 Apr.
- **LP:** 18 May.
- **SAT:** 17 Jun.

For each technology, a `zip` compressed archive should be delivered via Racó with the following contents:

- a directory `out` with the output files of those instances that could be solved. **Please ONLY provide the outputs of those instances for which an optimal solution could be found.** The output file corresponding to an instance `filename.inp` should be named `filename.out`.
- a directory `src` with all the source code (`C++`, scripts, `Makefile`, etc.) used to solve the problem.

  Your program will be tested on a Linux platform. Independently of the system on which the development was carried out (e.g., Windows, MacOS), you should also provide a `README` file with basic instructions for compiling and executing **in Linux**. You can use the lab machines to check that your code compiles and executes as expected on a Linux system.

  Your program is expected to solve one instance at a time. Moreover, it is required that your program **reads the instance from `stdin` and writes the solution to `stdout`.** If say your executable is named `program`, then in the evaluation of the project your results will be recomputed by calling `program < filename.inp > filename.out`. Any other output (debugging information, etc.) should be written to `stderr`.

In particular, **suboptimal solutions should not be written to** `stdout`, only the optimal solution (if eventually found).

Do not worry about setting the time limit in your code. When your program is rerun over all instances, it will be forced from the outside with the command `timeout`. Therefore, de facto you can implement your program as if there was no time limit.

Any script/program that you used to generate the outputs in the `out` directory, should be independent from the stand-alone program that solves one instance at a time.

- a report in PDF describing the variables and constraints that were used in your model, as well as any remarks or comments that you consider appropriate.

Please follow these indications when submitting your deliveries.

Finally, some points of advice:

- Run the checker that is provided with the materials to get confidence that your programs are correct. Wrong answers are much more penalized than being unable to solve some of the instances.
- Use the results table that is provided with the materials to check that your solutions are consistent with the expected optimal values.
- For the LP and SAT parts, compare your results with those obtained in previous deliveries, and make sure that they are consistent. Note however that optimal solutions may not be unique.
- Projects are **individual**. Any fraudulent delivery will be severely penalized.