Inferring Automata using State-merging

In this project you would have to build a program that given an automaton, infers a smaller automaton which is an approximation for the input automaton with respect to a given set of positive and negative examples which confirms to a set of constraints.

The entire project is split into sequential stages for ease of comprehension and execution. The stages DO **NOT** indicate an equal distribution of grade, time to be spent, or effort required. The grade for each task is mentioned as a percentage of the total project grade. Stage 5 is a bonus section which you may attempt if time permits after everything else is completed.

You will have to write clean and readable code which compiles. Any code that does not compile will not be graded. You will also provide a short report detailing how each task is handled. This report would also contain a brief but detailed answers for every non-coding task outlined.

Stage 0: Introduction and setup

This project involves use of automata. Unless otherwise specified automata (automaton) refer to deterministic finite-state automata (DFAs).

You can use the dk.brics.automaton Java package for handling and manipulating automata. It is well-maintained, has a good documentation and tutorials, and has most features you would need. You can use any other programming language and any other automata package you see fit.

You should be able to input and output automata in a standard human-readable format. The dk.brics.automaton package doesn't have a common read-write to a readable format, so you might have to write a wrapper for the Automaton class.

For simplicity we will consider only binary strings and automata over binary strings.

The modules/functions described are not rigid specifications, and you may slightly vary from function design as long as they have the same functionality and the do the task in the same or in a better way.

Stage 1: Automaton induction, and state merging

Deterministic Finite-state Automaton (DFA) induction is a popular technique to infer a regular language from positive and negative sample strings defined over a finite alphabet. Several state-merging algorithms have been proposed to tackle this task, including RPNI, EDSM etc. These algorithms start from a prefix tree acceptor (PTA), that accepts all the positive examples only, and successively merge states to generalize the induced language.

- Learn what the RPNI algorithm is and how merging two states works. (What happens if outgoing transitions from the two states to be merged go to different states? What happens if one of the states is accepting, and the other is rejecting?) [**5**%]

- Create a class/module which implements the function **merge** that takes as input an automaton, and two states and outputs the automaton which is the result of merging of the two states. [**15**%]

Stage 2: Best possible merge

For two automata A_1 and A_2, we define the measure that the two automata are consistent with respect to a finite non-empty set E of examples as the fraction of examples from the set E which give the same result on both the automata A_1 and A_2 (i.e. examples which are either accepted by both or rejected by both).

The best possible merge would the one which gives the automaton which is the most consistent (has the highest measure, as described above) with the original automaton.

- Of all possible automata obtained by a single merge operation, you have to find the one which is the most consistent to the original automaton. Write a function **shrink** which does this. It's input will be a single automaton, and a set of examples and it's output will be a single automaton. [Hint: For an automaton with n states there will be at most $n(n-1)$ ways to merge two states; and an initial way to find the best merge would be to simply try out all possible merges and pick the best one.] [**15**%]

- Is there a better way to find the best merge than simply checking *all* possible merges? Try to create a better method or come up with interesting optimizations over simple brute force. [**5**%]

Stage 3: Approximation using state-merging

The next step is to come up with an approximate automaton which has at most a given number of states using recursive shrink operations.

- Given as input an automaton A, a set of examples E, and a limit k, you should output an automaton whose number of states is at most k. Obtain by recursively applying the shrink operation. Implement a module/class with the function **r-shrink** to this effect. [**10**%]

Stage 4: Compilation of results

Here we wish to find a k-state automaton which is the most consistent with the original automaton with respect to the given set of examples and how its measure of consistency compares with the automaton obtained from the previous recursive shrink function.

- Implement a function **most-cons** which takes as input an automaton A, a set of examples E, and a limit k, and finds a k-state automaton A_k which is the most consistent with A. [Hint: You could try to model this in a theory and use a solver.] [**20**%]

- Compare the measures of consistency (with respect to E) between **most-cons(**A,E,k**)** and A, and that between **r-shrink(**A,E,k**)** and A. [**5**%]

- Generate various test cases of varying sizes. Again, while the maximum maximum size of cases your implementation can handle is not specified try to improve performance as much as possible. (For instance, at least for the the first 3 stages, you should be able to easily handle a 15-state automata, with a couple of thousand examples. This is not a set goal but a suggestion, and you should be able to do better.) Try to create unit tests to test each module/function/unit as you go along. [**15**%]

- You should also have flags/arguments to output the results of the recursive shrink procedure at each step of the recursion. Create this results for the test cases as well and try to represent them in a good manner. [**5**%]

Your code must be sufficiently documented (Javadoc or similar documentation is not necessary but can be included), a build file must be provided, and source, libraries, tests, and results must be cleanly arranged into directories.

Stage 5: Improving results

This stage is about trying to improve or build upon previous stages. This is not essential to the project. This is a **bonus** section, and may count towards extra credit in the course.

- Try to self-generate the set of examples from the given automaton instead of having one provided as input. The examples should try to capture the essence of the original automaton and must cover all important examples pertinent to it. [Hint: Look into characteristic set of automata.]

- Like the recursive shrink, try to implement a composite shrink function **c-shrink** which has the same inputs, and output but instead of applying the shrink function recursively try to obtain a k-state automaton which is the most consistent with the inout automaton by trying something smarter than **r-shrink** but faster than **most-cons**.

Try to add any other features or methods which you might think are interesting or related.

*** *** ***