```
Task documentation MKA: FA minimization in python3 for IPP 2013/2014
Name and surname: Ivan Ševčík
Login: xsevci50
```

Task specification

The task was to create a script in python3 language that would normalize and optionally minimize or analyze finite automata (referenced further in text as FA). The input consists of five components delimited by comma. First component contains all states of FA, second contains transition symbols, third is set of allowed rules, fourth represents starting state and last component is set of final states. The FA provided as input should be a well-specified FA (or deterministic FA in case of implementing extension). The normalized form of output is similar to input with some additional spacing and sorting requirements. Available switches with description can be printed with `--help` switch when running the script.

Proposed solution

The input needs to be processed into some form in which it will be possible to access every state and rule as standalone entity. After doing so, it is possible to run various searches and queries that can be used to check validity of input and prepare the output.

For finding the non-finishing state, the solution can be as simple as running search from final states using rules in opposite direction and marking visited states. All states left unmarked are non-finishing, and in case of well-specified FA this should result only in one state at maximum.

The minimization algorithm was constructed according to IFJ course procedure. First of all the states are grouped into two groups - the final states and all the others. Then all possible rules for one transition symbol are applied to states in group, resulting in new set of outcome states. If all states from this set belong to one group, the checked group is left as it was. Otherwise the states are separated into new groups according to division made by the set of outcome states. This happens for all available symbols and algorithm is repeated on all groups until there is no division needed. The resulting groups each represent a state in final minimized FA.

Implementation details

First of all, there was a need to support all command line arguments to be able to properly test the behavior later. For this purpose a finite state machine (FSM) was created that can parse through arguments and detect error. Another FSM was used to analyze FA provided as input. Processing one character at a time, it needed several state variables that would modify how the next character will be interpreted such as symbol recording flag that notified that anything read is part of symbol or state name. The behavior of FSM changes as the different components are processed.

Each entity is represented as a class, therefore while parsing input, every state, symbol and rule has its own instance of corresponding class. These instances are stored in lists inside instance of class representing finite automata. An additional class was used to represent a group of states used while minimizing the FA, which is later converted back to state class.

The states can be viewed as places and rules as edges in terminology of search algorithms. It is therefore easy to modify existing search algorithms for this particular case. For example a depth first search was used for finding non-finishing states as proposed earlier.

Interesting about minimization procedure is the fact that it produces new minimized FA without changing the previous one. When the regrouping of states is finished, each group is converted into single state by concatenating the names of all states inside group with character _. Symbols are copied as they were and rules are modified to match new states. New starting and final states are identified and these data are fed into FA constructor which produces new instance with minimized FA.

Python's magic methods, as they are referred to, were also used extensively to simplify implementation. For sorting into desired format, basic sort method from python was used and classes simply got their comparators overridden. Also `__str__` method was overridden in most of classes, providing specific formatting when output had to be put together.

The implementation was straightforward and no complications to what was proposed occurred. The focus was on data simplicity and elimination of redundancy, therefore the time complexity is a little higher than ideal. For this particular usage this doesn't present a problem, but optimizations would have to be implemented if the algorithm was to be run on large data sets.

Extensions

A bonus extension that allows for input to be a deterministic finite automata instead of well-specified one was implemented. It is available through `--wsfa` switch. The conversion into well-specified FA happens immediately after input parsing, creating unified input for the rest of script. The conversion consists of eliminating non-finishing states and providing only single `qFALSE` state if needed. All symbols not used in existing rules going out of a state connects this state with `qFALSE` through a newly created rule. Being deterministic finite automata, it is also needed to check if all states are reachable and if not, produce an error message.

Conclusion

A script providing required functionality was created along with this documentation. It was relatively easy to implement as python language provides elegant few-lines solution to most of what was required and working algorithm was already available from IFJ course materials. Difficulty of 3 seems therefore a little too high and could be lowered to 2 in future. The task itself was interesting and a nice way to verify minimization algorithm on large FA.