FiKnightomata Project Report

By: Alexander Darino, Jorge Rivas, Daniel Vivas

# Overview

# FiKnightomata (FKA) File Format Specification

An FKA file is a Unicode text file that represents an NFA, which may also be a DFA if the transitions are deterministic. The file is organized as follows:

|  |  |
| --- | --- |
| State Definitions | |
| INTEGER STATES\_TOTAL | Total quantity of states. States are indexed from 1 to STATES\_TOTAL |
| STRING state\_name\_i | STATES\_TOTAL quantity of lines: Each line state\_name\_i (0 < i ≤ STATES\_TOTAL) corresponds to the indexed states; each line stores the name of its corresponding state |
| Alphabet Symbol Definitions | |
| INTEGER ALPHABET\_SYMBOLS\_TOTAL | Total quantity of alphabet symbols. Symbols are indexed from 1 to ALPHABET\_SYMBOLS\_TOTAL |
| STRING alphabet\_symbol\_i | ALPHABET\_SYMBOLS\_TOTAL quantity of lines: Each line alphabet\_symbol\_i (0 < i ≤ ALPHABET\_SYMBOLS\_TOTAL) corresponds to the indexed alphabet symbols; each line stores the symbol encoding for its corresponding alphabet symbol |
| Transitions Function | |
| String state\_transition\_function\_i | STATES\_TOTAL quantity of lines: Each line state\_transition\_function\_i (0 < i ≤ STATES\_TOTAL) corresponds to the indexed states. Each line defines a state transition function formatted as follows:   * (ALPHABET\_SYMBOLS\_TOTAL + 1) space-delimited lists * Each list l\_i (0 ≤ i ≤ ALPHABET\_SYMBOLS\_TOTAL) stores the indices of the transition states on a given input, separated by commas.   + l\_0 represents transitions on epsilon   + l\_i (i>0) represents transitions on alphabet\_symbol\_i   + A “0” may be used in place of a list to denote a transition to the built-in state 0, which may represent “Null”, “Don’t Care”, “Error”, “Terminate”, etc. All transitions from state 0 go to state 0. |
| Start State Definition | |
| INTEGER START\_STATE\_INDEX | The index of starting state |
| Accepting State Definitions | |
| INTEGER ACCEPTING\_STATES\_TOTAL | Total number of accepting states |
| INTEGER accepting\_state\_i | ACCEPTING\_STATES\_TOTAL quantity of lines: each line accepting\_state\_i (0 < i < ACCEPTING\_STATES\_TOTAL) corresponds to the index of an accepting states. |

## Sample FKA Files

|  |  |
| --- | --- |
| mod3.fka  3  Zero  One  Two  3  0  1  2  0 1 2 3  0 2 3 1  0 3 1 2  1  1  1 |  |
| chain\_of\_useless\_nodes.fka  6  1  2  3  4  5  6  1  0  2 1  3 0  4 0  5 0  6 0  0 0  1  1  3 | Gets minimized to: |

|  |  |
| --- | --- |
| dead\_paths.fka  14  1  2  3  4  5  6  7  8  9  10  11  12  13  14  1  0  0 2,3,6,8,12  0 0  0 4  0 5  0 3  0 7  0 0  0 9  0 10  0 11  0 9  0 13  0 14  0 12  1  3  6  8  12 | Gets minimized to: |

# High-Level Architecture and Detailed Design

Please refer to FiKnightomata API specifications for more information. The FiKnightomata API has been supplied with the software and also provided in Appendix A for your convenience.

# Deterministic Finite Automaton Simulator

The DFASimulator class is an online, interactive DFA simulation program. It accepts the filename of a valid FKA file as a required command-line argument. The program operates as follows:

* The program reads the FKA file as an NFA
* The NFA is converted into a minimized DFA
* The transition function of the DFA is printed. This is done by printing the state-level transition functions for each state in no particular order.
* The start state is printed. The user is informed of whether or not the start state is accepting.
* The user is continuously prompted for input. With each input, the corresponding state transition appears on-screen as well as whether or not the terminal state reached is accepting. If the user supplies an invalid input, the automaton automatically enters the default “null” state, which is “inescapable”.
* There is no means to terminate the program aside from issuing a TERM or KILL signal (eg. Ctrl+C at the command-line).

# NFA-to-DFA Conversion

The NFA.toDFA() method returns a DFA representation of the NFA. The conversion occurs in two phases:

1. An epsilon-free NFA representation of the NFA is produced.
2. A DFA representation of the epsilon-free NFA is produced.

## NFA.toEpsilonFreeNFA()

The toEpsilonFreeNFA() method lazily constructs an epsilon-free NFA representation of an NFA. It’s algorithmic implementation is relatively textbook and no further optimizations are performed at this stage.

## NFA.toDFA()

The toDFA() method lazily constructs an epsilon-free DFA representation of an epsilon-free NFA. It’s algorithm implementation, a derivative of a breadth-first traversal, is also relatively textbook, although the constructed DFA is optimized offline after it has been constructed and prior to it being returned. A SetLabel class has been implemented to facilitate the construction of the DFA. Whenever states are consolidated into a single new state in the DFA, the SetLabel class provides both the name of the new state (as a single set representation of the older states) as well as the set of the derived states from the original NFA. This allows the DFA construction algorithm to easily map transitions from new states onto new states based on the transitions from the original states.

## DFA.clean()

The clean() method is internally used to minimize a DFA prior to being used. It operates using a proprietary “Double-DFS” algorithm, which operates as follows:

1. DFS #1: Mark as “keep” all states along all simple paths from the start state to all accepting states
   1. Using a DFS traversal, whenever an accepting state is reached, mark all of the states along the current path from the start state to that state as “keep”
2. DFS #2: Mark as “keep” all states along all cyclical paths from the start state to all accepting states
   1. Using a DFS traversal, whenever the traversal forms a cycle, mark all the states on the current path as “keep” whenever the start/end node of the cycle has been marked as “keep” from DFA #1.