June 14, 2018

Matching a graph with a non-deterministic finite automaton

Alexandre Boulgakov

Matching a graph with a non-deterministic finite automaton

ABSTRACT

The problem of matching a graph with a non-deterministic finite automaton (NFA) is of importance in various domains of computer science. An example is regular-expression matching, which can be formulated as a graph-matching problem. Current techniques of matching graphs against NFAs have relatively high computational complexity. This disclosure presents matching techniques with complexity that is linear in the size of the graph. The graph to be matched against the NFA is itself considered as an NFA. A synchronized product of the two NFAs is defined, and the matching problem is shown equivalent to a reachability problem solvable in linear (time and space) complexity.

KEYWORDS

- Graph matching
- Non-deterministic finite automaton (NFA)
- Deterministic finite automaton (DFA)
- Regex matching

BACKGROUND

Certain problems in computer science (e.g., regular-expression matching and similar problems) are based on matching a rooted, edge-labeled graph with a non-deterministic finite automaton (NFA). Specifically, given a rooted, edge-labeled graph, a determination is made if any sequence of edge labels on any possible rooted path through the graph is matched by a given NFA. Current techniques for determining match/no-match between the given graph and the given NFA have high computational complexity.

DESCRIPTION

This disclosure describes techniques to determine a match between a given rooted, edge-labeled graph and a given NFA. The techniques have a worst-case computational complexity proportional to the product of the sizes of the graph and the NFA. Computational complexities in both time (e.g., speed of execution) and space (e.g., memory usage) are linear in the size of the graph and the NFA. Moreover, localized non-determinisms have only a local effect. Thus, in practical applications, worst-case complexity is not reached. Rather, space/time complexity is generally better than worst-case, e.g., it is proportional to the size of the graph, with run-times similar to run-times achieved when the techniques are applied to a deterministic finite automaton (DFA).

To determine match between a graph and NFA, the graph is itself considered as an NFA with all states accepting. A synchronized product of the two NFAs is defined such that the problem of determining match of graph to NFA is equivalent to determining whether the product-NFA has a non-empty acceptance set. In this manner, the problem of matching graph to NFA is reduced to a simple reachability problem solvable in linear time and space, e.g., using a breadth-first or depth-first search.

To formally delineate the techniques, notation is presently established. The NFA is represented as a five-tuple $(Q_1, \Sigma, \Delta_1, q_1, F_1)$ illustrated in Fig. 1.

$(Q_1,\ \Sigma,\ \Delta_1,\ q_1,\ F_1)$, where

Q_1 is the set of states of the NFA;

Σ is the set of input symbols to the NFA (the alphabet);

Δ_1 is a transition function between states of the NFA, e.g.,
$\Delta_1 : Q_1 \times (\Sigma \cup \{\epsilon\}) \longrightarrow 2^{Q_1}$, ϵ being the empty string;

$q_1 \in Q_1$ is an initial start state of the NFA; and

$F_1 \subseteq Q_1$ is a set of accepting states of the NFA.

Fig. 1: Notation for the NFA

In a similar manner, the rooted, edge-labeled graph is represented by another NFA, denoted (Q_2, Σ, Δ_2, q_2, F_2).

The *synchronized product* of the two NFAs is defined as an NFA which shares the alphabet of the constituent NFAs, whose set of states is the Cartesian product of the two constituent NFAs, and which has a transition function as defined in Fig. 2.

$$\begin{aligned}
&\text{Synchronized product of } (Q_1,\ \Sigma,\ \Delta_1,\ q_1,\ F_1) \text{ and } (Q_2,\ \Sigma,\ \Delta_2,\ q_2,\ F_2) \\
&\quad = \ (Q_1 \times Q_2,\ \Sigma,\ \Delta',\ (q_1, q_2),\ F_1 \times F_2), \text{ where} \\
&\Delta' \ = \ \{\ (\ (s_1, s_2), \epsilon, (t_1, s_2)\ ) \mid s_2 \in Q_2 \wedge (s_1, \epsilon, t_1) \in \Delta_1\ \} \ \cup \\
&\qquad \{\ (\ (s_1, s_2), \epsilon, (s_1, t_2)\ ) \mid s_1 \in Q_1 \wedge (s_2, \epsilon, t_2) \in \Delta_2\ \} \ \cup \\
&\qquad \{\ (\ (s_1, s_2), a, (t_1, t_2)\ ) \mid a \neq \epsilon \wedge (s_1, a, t_1) \in \Delta_1 \wedge (s_2, a, t_2) \in \Delta_2\ \}.
\end{aligned}$$

Fig. 2: Definition of synchronized product of two NFAs

The synchronized product of two NFAs is thus a restriction of their Cartesian product such that only transitions where both automata recognize the same input symbol are allowed. The first two components of Δ' allow each of the two NFAs to make an epsilon-move independently while the third component allows the NFAs to recognize the same symbol in lockstep.

Explicit computation of Δ' requires $|Q_1| \times |Q_2|$ states and up to $|\Delta_1| \times |\Delta_2|$ transitions. However, in practice, many of these states and transitions are unreachable. The synchronized product is therefore computed on the fly, ensuring that only necessary states and transitions are computed. To efficiently compute the transitions outgoing from a state pair, one automaton is used to support efficiently finding the outgoing transitions from its states. Thus $\Delta_1(s)$ is efficiently computed on the fly as $\Delta_1(s) = \{ (a_1, t_1) \mid (s, a_1, t_1) \in \Delta_1 \}$. The other automaton similarly supports finding the outgoing transitions from a state with a given label. Thus $\Delta_2(s, a)$ is efficiently computed on the fly as $\Delta_2(s, a) = \{ t_2 \mid (s, a, t_2) \in \Delta_2 \}$. In this manner, transitions from a state (s_1, s_2) are computed as shown in Fig. 3 below.

for each $(a_1, t_1) \in \Delta_1(s_1)$:

if $a_1 = \epsilon$, then $((s_1, s_2), \epsilon, (t_1, s_2))$ is a transition.

if $a_1 \neq \epsilon$, then

for each $t_2 \in \Delta_2(s_2, a_1)$:

$((s_1, s_2), a_1, (t_1, t_2))$ is a transition.

for each $t_2 \in \Delta_2(s_2, \epsilon)$: $((s_1, s_2), \epsilon, (s_1, t_2))$ is a transition.

Fig. 3: Computation of state transitions

On-the-fly computation of the synchronized product, per techniques of this disclosure, enables not only finite-state but also some infinite-state systems to be processed in finite time. For example, non-deterministic automata with finite branching, e.g., where $\Delta_1(s)$ is finite for all states reachable from q_1, make progress, and a breadth-first search enables accepting states to be found in finite time if a finite string is accepted. Another implication is that the alphabet Σ need not be finite, e.g., Σ is a set of UTF-8 strings (not characters).

Per the techniques of this disclosure, compute-intensive and explicit representation of $\Delta_2(s, a)$, e.g., with an adjacency matrix, is obviated. The ability of one of the automata to efficiently and *on-the-fly* compute $\Delta_2(s, a)$ using $\Delta_2(s, a) = \{ t_2 \mid (s, a, t_2) \in \Delta_2 \}$ enables the techniques to efficiently operate on and represent other abstractions such as counter-extended NFAs. This enables efficient matching of bounded repetitions such as the regular expression (regex) operator `{m,n}`, which matches a regular expression at least `m`, but no more than `n`, times. This is traditionally matched by creating an automaton with `n` subunits corresponding to the repeated expression. With on-the-fly computation, the operator is represented symbolically, and the states are generated only if needed. For example, traditional regex engines take a long time to compile `a{2,1000000000}`, even for matching short strings such as `b`, `a`, and `aaa`. With the present techniques, checking is performed in time and space proportional to the length of the string.

The described techniques apply to problems involving matching a subset of a graph against another graph, or generally for matching problems in any domain where data is expressed as graphs. For example, the techniques find use in matching specifications of a particular behavior of interest against a superset of behaviors extracted from an executable file of interest using static analysis. Another application of the techniques, e.g., within the domain of computer security, is to match the same behavior specification against the observed behavior of an application when run on a test system.

CONCLUSION

This disclosure presents techniques to match a graph (or subset thereof) with another graph or non-deterministic finite automaton (NFA). The techniques are of relatively low

complexity, e.g., linear in the size of the graph and NFA. The techniques have various applications, e.g., matching of particular behaviors of binary executables against a superset of behaviors, regular-expression (regex) matching, etc. The techniques define a product of the given NFA and the graph-to-be-matched, itself considered as an NFA, such that the matching problem reduces to a reachability problem. The reachability problem is solved in linear time and space using, e.g., depth-first or breadth-first search.

REFERENCES

[1] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, "FDR3: A parallel refinement checker for CSP," *Int. J. Softw. Tools Technol. Transfer (2016) 18: 149* (https://doi.org/10.1007/s10009-015-0377-y)

[2] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe A. W. Roscoe, "Modelling and analysis of security protocols: the CSP approach," Addison-Wesley (2001)

[3] A. Boulgakov, T. Gibson-Robinson, A. W. Roscoe, "Computing maximal weak and other bisimulations", *Formal Aspects of Computing 28(2)* pp. 381-407

[4] A. Boulgakov "Improving Scalability of Exploratory Model Checking." *PhD thesis, University of Oxford (2016)* (https://ora.ox.ac.uk/objects/uuid:76acb8bf-52e7-4078-ab4f-65f3ea07ba3d)

[5] A. W. Roscoe, "The theory and practice of concurrency," Prentice Hall NJ (1997)