UNIWERSYTET
IM. ADAMA MICKIEWICZA
W POZNANIU

Wydział Matematyki i Informatyki

Bohdan Bondar, Marcin Jałoński, Aleksander Mendoza-Drosik
Numer albumu: sS432778,s434701,s434749

Solomonoff - kompilator transduktorów skończenie stanowych

Solomonoff - Finite state tranducer compiler

Praca inżynierska na kierunku **informatyka**
napisana pod opieką
dr Bartłomieja Przybylskiego

Poznań, luty 2021

Poznań, 22 listopada 2020 r.

Oświadczenie

Ja, niżej podpisana **Bohdan Bondar, Marcin Jałoński, Aleksander Mendoza-Drosik**, studentka Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Solomonoff - kompilator transduktorów skończenie stanowych* napisałam samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałam z pomocy innych osób, a w szczególności nie zlecałam opracowania rozprawy lub jej części innym osobom, ani nie odpisywałam tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[TAK/TAK/TAK] wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[TAK/TAK/TAK] wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

Streszczenie

Słowa kluczowe: klasa

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Keywords: klasa

Tu możesz umieścić swoją dedykację.

Spis treści

Rozdział 1

Introduction

This project focuses on research in the field of automata theory and inductive inference. The main product of our work is the "Solomonoff" regular expression compiler for finite state transducers. Plenty of research has gone into development of the theory behind this system. As a result the transducers contain several features not know before.

The most innovative achievements is the lexicographic arctic semiring of weights, specialized adaptation of Glushkov's construction for subsequential transducers and the most significant flagship feature - built-in support for inductive inference and machine learning of transducers. Thanks to the cooperation with LearnLib and Dortmund University, Solomonoff supports learning algorithms such as RPNI and several of it's derivatives. Solomonoff contributes its own more specialized for transducers inference. We implemented OSTIA for efficient learning of deterministic transducers. For nondeterministic ones we developed our own OFTIA algorithm, that was not known before.

All those features together make Solomonoff a unique library that stands out from all the alternatives. We support most of the features of UNIX regexes, including look-aheads and look-behinds, which is unusual for automata-based regex engine. The key that allows Solomonoff for doing that is the possibility of emulating look-aheads with careful placement of transducer outputs. As a result Solomonoff can compete and do much more than existing projects such as RE2 developed by Google, BRICKS automata developed at Aarhus University or even to certain extent with Pearl/Java based regular expression engines. Another, much stronger competitor for Solomonoff is the OpenFST project developed by Google. Their Thrax grammars are capable of doing most of the things that Solomonoff can and they also support probabilistic automata. OpenFST is much older and more established in the scientific community. They support a lot more features that were developed by scientists, by the course of many years. Solomonoff cannot compete with this level of sophistication, but perhaps, what might seem like a limitation, is in fact our strongest advantage. Solomonoff

focuses on functional transducers and enforces this property at compilation time. Any arising nondeterministic output is automatically rejected as an error. This allows Solomonoff to perform a lot more optimisations, the automata are smaller and their behaviour is more predictable. Moreover, lexicographic weights allow for precise disambiguation of nondeterministic paths whenever necessary and their most important advantage is that lexicographic semiring is not commutative and hence it does not "propagate" throughout entire automaton. This only increases Solomonoff's robustness and allows for (exponentially) smaller automata, without sacrificing predictability of the regular expression. On th contrary, probabilistic weights in Thrax, make the whole system, more heavyweight, unpredictable and difficult to maintain. The results are especially palpable when comparing our benchmarks. Solomonoff was written in Java and Thrax uses C++, but despite this our compiler is several magnitudes more efficient. We performed efficiency tests on a large corpus of linguistic data (dictionary with 6000 records). Solomonoff compilation times were around 2 seconds, whereas Thrax took 19 minutes. Solomonoffs automata were also much smaller, as thanks to Glushkov's construction, there is a 1:1 relationship between size of regular expression and size of transducer. As a result Thrax's transducer takes of 6336K of RAM, whereas Solomonoff only takes 738K. Execution time for such large corpora was about 10 miliseconds in Solomonoff (which is roughly the same performance as using Java's HashMap), while Thrax took about 250 miliseconds. One might argue that, such great differences are achievable, only because Thrax supports a lot more features. OpenFST uses epsilon transitions, while Solomonoff does not implement them and instead all automata are always and directly produced in epsilon-free form. OpenFST performs operations such as sorting of edges, determinization, epsilon-removal and minimization. Solomonoff always has all of its edges sorted and it doesn't need a special routine for it (which makes for additional performance gains), it has no epsilons to remove and it does not need determinization procedure, because it can pseudo-minimise nondeterministic transducers, using heuristics inspired by Kameda-Weiner's NFA minimization. One could, half-jokingly, summarize that the difference between Thrax and Solomonoff is like that between "Android and Apple". Thrax wants to support "all of the features at all cost", whereas Solomonoff carefully chooses the right features to support. We believe this approach will be our strongest asset, that will make our compiler a serious alternative to the older and more established OpenFST. An additional strength that favours Solomonoff over OpenFST is that we support inductive inference out-of-the-box, require no programming (regular expressions are the primary user interface and Java API is minimal and optional) and we provide automatic

conversion from Thrax to Solomonoff, so that existing codebases can be easily migrated.

The characteristic feature of Solomonoff, is that its development focuses on the compiler and regular expressions instead of library API. Everything can be done without writing any Java code. It also allows the developers for much greater flexibility, because the internal implementation can be drastically changed at any time, without breaking existing regular expressions. There is very little public API that needs to be maintained. As a result backwards-compatibility is rarely an issue.

The primary philosophy used in implementing this library is the top-down approach. Features are added conservatively in a well thought-through manner. No features will be added ad-hoc. Everything is meant to fit well together and follow some greater design strategy. For comparison, consider the difference between OpenFst and Solomonoff.

— OpenFst has Matcher that was meant to compactify ranges. In Solomonoff all transitions are ranged and follow the theory of (S,k)-automata. They are well integrated with regular expressions and Glushkov's construction. They allow for more efficient squaring and subset construction. Instead of being an ad-hoc feature, they are well integrated everywhere.
— OpenFst had no built-in support for regular expression and it was added only later in form of Thrax grammars, that aren't much more than another API for calling library functions. In Solomonoff the regular expressions are the primary and only interface. Instead of having separate procedures for union, concatenation and Kleene closure, there is only one procedure that takes arbitrary regular expression and compiles it in batches. This way everything works much faster, doesn't lead to introduction of any ϵ-transitions and allows for more optimisation strategies by AST manipulations. This leads to significant differences in performance.

While, most of other automata libraries (RE2/BRICKS) were not designed for large codebases and have no build system, Solomonoff comes with it's very own build system out-of-the-box. Thrax is the only alternative that does have a "build system" but it's very primitive and relies on generating Makefiles. Solomonoff has a well integrated tool that assembles large projects, detects cyclic dependencies, allows for parallel compilation and performs additional code optimisations. It also ships with syntax highlighting and tools that assist code refactoring. Our project strives to make automata as easy to use and accessible to mass audiences as possible. For this reason we developed a website with online playground where visitors could test Solomonoff and experiment without

any setup required. The backend technology we used is Spring Boot, because it allowed for convenient integration with existing API in Java.

While at the beginning our attempts focused on implementing the library is C, switching to Java turned out to be a major advantage. While, it's true that C allows for more manual optimisations, having a garbage collector proved to be a strong asset. The compilation relies on building automata in form of directed graphs. Each vertex itself is a separate Java object. The automaton is always kept trim because all the unreachable vertices are lost and free to be garbage collected at any time. If we tried to implement such data structure in C we would need to reinvent a simple (and most likely, less efficient) garbage collector ourselves. Manipulation of linked lists, linked graphs and other recursive data structures is not as convenient in C as it is in Java. Another advantage that Java gave us over C is that the plenty of existing infrastructure was already implemented in Java. Hence we became more compatible with Samsung's systems where Solomonoff could be deployed. We could also easily integrate our compiler with LearnLib.

For environments where Java is not an option, Solomonoff will allow for generating automata in form of C header files that can be easily included and deployed in production code. It will not allow for manipulation of automata from C level, but our compiler was never intended to be used programmatically. There is, in fact, not much benefit from doing so. If user needs to create their custom automata manually, it's much easier to generate them in AT&T format and then have them read by the compiler. Later they can by manipulated with regular expressions like any other automaton. Generating AT&T has also additional benefits, as such custom script for generating AT&T can be attached to Solomonoff build system as an external routine and then compilation can be optimised more efficiently. An example scenario would be having two independent routines that the build system could decide to run in parallel and then it could cache the results for subsequent rebuilds. For comparison, if anybody decided to use OpenFST manually from from C++, they would need to first learn the API (which is less user friendly than learning AT&T format) and then they would also need to implement caching and parallelization themselves. Moreover, the users are also tied to C++, whereas, AT&T format can be generated in any language. This shows, why API for programmatic manipulation offers little to no benefit, while imposes much more limitations. It should also be mentioned that Solomonoff allows user for implementing their own Java functions that could then be incorporated into the regular expressions as "native" calls.

Rozdział 2

Transducers

Rozdział 3

Build system

Rozdział 4

Web technologies