Faster Reachability Analysis for LR(1) Parsers

Frédéric Bour
Inria & Tarides
Paris, France
frederic.bour@inria.fr

François Pottier
Inria
Paris, France
francois.pottier@inria.fr

Abstract

We present a novel algorithm for reachability in an LR(1) automaton. For each transition in the automaton, the problem is to determine under what conditions this transition can be taken, that is, which (minimal) input fragment and which lookahead symbol allow taking this transition. Our algorithm outperforms Pottier's algorithm (2016) by up to three orders of magnitude on real-world grammars. Among other applications, this vastly improves the scalability of Jeffery's error reporting technique (2003), where a mapping of (reachable) error states to messages must be created and maintained.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Keywords: Compilers, parsing, error diagnosis, reachability

ACM Reference Format:
Frédéric Bour and François Pottier. 2021. Faster Reachability Analysis for LR(1) Parsers. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486608.3486903

1 Introduction

The Reachability Problem. An LR(1) automaton is a state machine that is equipped with a stack, a sequence of past states. The state found at the top of the stack is the current state. At each step, the automaton's action is determined by its current state and by the lookahead symbol, that is, by the first unconsumed input symbol. The automaton's palette of actions includes consuming the first input symbol, following a transition to a new state (which is then pushed onto the stack), and popping a number of states off the stack (thus changing the current state).

SLE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486903

A pair of a state s and a symbol z is *reachable* if there exists an input sentence that leads the automaton from its initial configuration to a configuration where the current state is s and the current lookahead symbol is z. In its simplest form, the *reachability problem* can be formulated as follows:

Given: a state s; a terminal symbol z.
Find: whether (s, z) is reachable.

The problem can also be stated under a more elaborate form, as follows:

Given: a state s; a transition tr that leaves the state s; two terminal symbols a and z.
Find: whether there exists a sentence w such that:
- the first symbol of wz is a, and
- whenever the automaton is in state s and its input begins with wz, the automaton consumes w, leaves z unconsumed, and takes the transition tr.

In short, the second problem asks under what conditions each transition can be taken. A solution of this problem yields a solution of the first problem. Indeed, a solution of the second problem allows constructing a directed graph G, whose vertices are pairs (s, z), such that the first problem boils down to ordinary reachability in the graph G. The role played by the terminal symbol a may not clearly appear to the reader at this point; we come back to it in §3.

By an "LR(1) automaton", we mean a state machine that is equipped with a stack and can peek at the first input symbol. There are many methods for constructing such an automaton, including Knuth's canonical method [15], SLR [8], LALR [6, 7], Pager's method [24], and IELR(1) [5]. Furthermore, this construction process can be influenced by user-provided precedence declarations, which indicate how shift/reduce and reduce/reduce conflicts must be resolved. Because we state the reachability problem in terms of the automaton, not in terms of a grammar, we do not care how the automaton is constructed. Our reachability algorithm is compatible with all of the construction methods cited above.

Applications. Solving the reachability problem can be useful in a range of situations. One simple application is error diagnosis, that is, the task of producing a diagnostic message when a syntax error is encountered. Jeffery [14] suggests producing a message based solely on the current state of the automaton, while disregarding its stack. This is done by setting up in advance a mapping of states to messages. Because state numbers are opaque and brittle, the user is in fact expected to set up a mapping of erroneous

input sentences to messages. Pottier [27] notes that this table must be *correct* (every sentence is erroneous), *irredundant* (no two sentences lead to the same state), and *complete* (every state where an error can be triggered corresponds to some sentence). While correctness and irredundancy are easy to enforce, completeness is a more challenging property. Constructing a complete collection of erroneous input sentences, or just checking that a collection of erroneous input sentences is complete, requires the ability to compute the *reachable error states*, that is, to determine which states s are such that some input sentence triggers an error in state s. This requires solving the reachability problem.

Pottier [27] proposes a reachability algorithm and implements it in the Menhir parser generator [28]. Menhir's support for Jeffery's error diagnosis methodology is exploited in production in several compilers, such as CompCert C [20] Catala [22], and Stan 3 [31].

A reachability algorithm has applications beyond error diagnosis. It could be used, for instance, to test an LR(1) parser by generating a set of input sentences that reaches every state, or one that triggers every reduction in every state. It could also be used as a component in syntax error recovery algorithms and in syntactic completion algorithms.

Challenges and Contributions. The reachability problem seems inherently costly. The question that appears in the second problem statement above is parameterized with a transition tr and with two terminal symbols a and z. Thus, the number of questions that may be asked is $|Tr| \times |T|^2$, where Tr is the set of all transitions and T is the set of all terminal symbols. An analysis of the problem reveals that these questions seem interdependent, so answering one of them potentially requires answering many of them. Without even analyzing the cost of answering one question, this suggests that the complexity of a reachability algorithm can be high. We analyze the complexity of a naïve reachability algorithm later on (§3.3).

Pottier [27] proposes a reachability algorithm whose implementation in Menhir [28] has been in use for five years. We find that, in practice, this algorithm does not scale as graciously as one might wish. For the OCaml grammar and parser, which involve 121 terminal symbols, 230 nonterminal symbols, and 1672 LR(1) states, we find that this algorithm can require over 5 gigabytes of memory and 5 minutes of run time. Although the space requirement is tolerable, such a time requirement is much too large to achieve a smooth edit-compile-debug cycle.

Contribution. We present a new reachability algorithm which we find is much more efficient in practice than Pottier's algorithm. Our algorithm processes the OCaml parser in 50 megabytes of memory and half a second, improving over Pottier by a factor of 10^2 in space and 6×10^2 in time (§7).

We derive this new algorithm in two main steps. First, we propose a matrix-based formulation of the reachability problem, which requires computing a *cost matrix* for every transition. Then, we remark that cost matrices are redundant in two ways: (a) they contain many identical columns; (b) they contain many rows whose content is not relevant, insofar as only the *combination* of these rows via the function "pointwise minimum" is of interest. This suggests that it is possible to gain space (and save time) by merging certain rows and columns. We propose a method for determining ahead of time (before the cost matrices are computed) which rows and columns can be merged, as well as a direct method for constructing *compact cost matrices* where these rows and columns are merged.

These ideas, combined with a number of other optimizations, allow us to report excellent performance in practice. The largest grammar in our test suite, a grammar for the C++ programming language [12], is processed in 2.6 gigabytes of memory and under one minute, whereas it could not be processed by Pottier's algorithm.[1]

Outline. After introducing some notation (§2), we present our formulation of the reachability problem in terms of cost matrices (§3). Then (§4), we explain several ways in which this formulation can be improved, including merging certain rows and columns. This leads to a reformulation of the problem in terms of compact cost matrices (§5). The paper ends with a discussion of implementation details (§6), an experimental evaluation of the algorithm's performance (§7), a review of the related work (§8), and concluding words (§9).

2 Notation

We assume that a context-free grammar $\mathcal{G}$ has been fixed and that an LR(1) automaton has been constructed for this grammar. We assume that the reader is familiar with these concepts [1, 11]. In the following, we recall some standard notation and propose specific notation that is useful in this paper.

LR(1) Automata. We write:

- $a, b, c, z \in T$ for terminal symbols,
- $A \in N$ for nonterminal symbols,
- $x \in X$, where X is $T \cup N$, for arbitrary symbols,
- $Z \in \mathcal{P}(T)$ for sets of terminal symbols,
- $w \in \vec{T}$ for sentences (sequences of terminal symbols),
- $\alpha, \beta \in \vec{X}$ for sentential forms (sequences of symbols),
- $s \in S$ for automaton states.

A transition in the automaton is a directed edge from a source state s to a target state s'. Every transition is labeled with a symbol x. The "shift" transitions are the transitions whose label is a terminal symbol; the "goto" transitions are

[1] Pottier's algorithm, as implemented by him in Menhir, is limited to 256 terminal symbols. This is caused by the choice of a low-level representation that involves packing several pieces of information in a single machine word. Even if this limitation was removed, our performance evaluation (§7) shows that Pottier's algorithm would not be able to handle this grammar.

those whose label is a nonterminal symbol. We write Tr for the set of all transitions.

Because the automaton is deterministic, the target state s' of a transition is determined by s and x. Thus, a transition can be identified by the pair (s, x), and Tr can be viewed as a subset of $S \times X$.

The target state of a transition is given by the function $target : Tr \to S$, which can also be viewed as a partial function $target : S \times X \rightharpoonup S$. We generalize this function so as to accept a state s and a sequence of symbols α as arguments. Thus, $target(s, \alpha)$ is the state that is reached by following the path labeled α out of the state s, if such a path exists. We write $incoming(s')$ for the set of the transitions that enter the state s', that is, for the set $\{tr \in Tr \mid target(tr) = s'\}$.

We assume that the reduction actions of the automaton are given by a function *reduce*, which maps a pair of a state s and a production $A \to \alpha$ to a set of terminal symbols. When the automaton is in state s, if the next input symbol (the "lookahead symbol") is a member of $reduce(s, A \to \alpha)$, then the automaton will reduce the production $A \to \alpha$.

Costs. A cost is either a nonnegative natural number or ∞. We write $\bar{\mathbb{N}}$ for $\mathbb{N} \cup \{\infty\}$, and equip it with semiring structure. We use the min-plus semiring $(\bar{\mathbb{N}}, \oplus, \otimes)$, where:

- $x \oplus y = min\{x, y\}$,
- the unit of $\oplus$ is $+\infty$,
- $x \otimes y = x + y$,
- the unit of $\otimes$ is 0.

Matrices. We use "cost matrices", that is, matrices whose elements are costs. We write "$\cdot$" for the matrix product over the cost semiring. The rows and columns of cost matrices are indexed with terminal symbols (§3). Later on, we introduce "compact cost matrices" whose rows and columns are indexed with sets of terminal symbols (§5).

Partitions. A *partition* of the terminal symbols is a set P of nonempty subsets of T such that every terminal symbol is a member of exactly one element of P. We write $\mathrm{Part}(T)$ for the set of all partitions of T. We let P, Q range over partitions, and p, q range over *classes*, that is, elements of a partition.

A partition P *refines* a partition Q if every element of P is a subset of some element of Q. We write $P \preceq Q$ when this is the case. Equipped with this partial order, $\mathrm{Part}(T)$ forms a lattice. Its bottom element $\bot$ is the finest partition, $\{\{a\} \mid a \in T\}$. Its top element $\top$ is the coarsest partition, $\{T\}$. The meet of two partitions P and Q, written $P \wedge Q$, is the coarsest partition that refines both P and Q. The join of two partitions P and Q, written $P \vee Q$, is the finest partition that is refined both by P and by Q.

If Z is a set of terminal symbols, we write $Z?$ for the partition $\{Z, T \setminus Z\} \setminus \{\emptyset\}$. This partition has at most two classes, namely Z and $T \setminus Z$. If one of these sets is empty, then it has only one class. This partition distinguishes members versus nonmembers of Z.

We let $Z!$ stand for $(\{T \setminus Z\} \cup \bigcup\{\{a\} \mid a \in Z\}) \setminus \{\emptyset\}$. The classes of this partition are $T \setminus Z$ (if it is nonempty) and the singletons $\{a\}$, where a ranges over Z. This partition distinguishes all members of Z, and places all nonmembers of Z in a single class. We have $Z! \preceq Z?$.

Finally, if P is a partition, we define the partition $P \downarrow Z$, pronounced "P inside Z", as $(P \wedge Z?) \vee Z!$. Taking the meet of P and $Z?$ refines P by distinguishing members versus nonmembers of Z. Then, taking a join with $Z!$ coarsens the result by conflating all nonmembers of Z into a single class. We have $Z! \preceq P \downarrow Z \preceq Z?$.

3 Problem Specification

3.1 Transition Costs

Let us briefly explain again what problem we wish to solve. In short, for each transition in the automaton, we would like to determine under what conditions this transition can be taken. So, we might wish to ask:

Question 3.1. Let tr be a transition. What input sentences w allow tr to be taken, and are consumed when it is taken?

In the case of a shift transition (s, a), naturally, the answer is immediate: this transition can be taken when the first input symbol is a, and this symbol is consumed in the process.

In the case of a goto transition (s, A), answering is more difficult. To take such a transition, the automaton must consume a sequence w of input symbols so as to reach a state where one of the productions associated with the symbol A, say $A \to \alpha$, can be reduced. There, the automaton must reduce α to A, which is permitted only if the input symbol that follows w, say z, allows such a reduction. The symbol z is known as the *lookahead symbol*: it is consulted before the reduction, but not consumed.

Because the answer to the question depends on z, the first input symbol that follows w, and because we seek a pleasant decomposition of the main problem into subproblems, the need naturally arises to refine our question by imposing a constraint on the *first symbol* a of the sentence wz.

Question 3.2. Let tr, a, z be a transition and two terminal symbols. What sentences w allow tr to be taken, under the assumption that the first input symbol following w is z, and under the constraint that the first symbol of wz must be a?

For greater simplicity, instead of answering Question 3.2 with a set of sentences, we focus on computing the *minimum length* of a sentence in this set, a number in $\bar{\mathbb{N}}$. Once the cost of every transition is known, it is a routine exercise to reconstruct a minimum-cost sentence that allows each transition to be taken.

The answer to Question 3.2 is the *cost* of transition tr with respect to the initial symbol a and the lookahead symbol z. We write $cost(tr)_{a,z}$ for this cost; it is a number in $\bar{\mathbb{N}}$.

It is useful to think of transition costs as matrices. For each transition tr, $cost(tr)$ can be viewed as a *cost matrix* whose

rows are indexed by a and whose columns are indexed by z. This matrix can be viewed as the answer to Question 3.1. Whereas Question 3.2 is a family of questions, indexed by a and z, each of which is answered by a single cost, Question 3.1 is a single question that is answered by a cost matrix.

3.2 A Characterization of Transition Costs

We now characterize the cost matrices $cost(tr)$ by providing a family of equations that these matrices satisfy.

The cost of a shift transition (s, a) is easy to express. It is 1 if the constraint imposed on the initial input symbol allows taking this transition; it is ∞ otherwise. Thus:

$$\begin{array}{ll} cost(s,a)_{b,z} = 1 & \text{if } a = b \\ cost(s,a)_{b,z} = \infty & \text{otherwise} \end{array}$$

In other words, the matrix $cost(s, a)$ contains a row of 1's at index a and contains ∞ everywhere else:

$$cost(s,a) = \begin{array}{c} \vdots \\ a \\ \vdots \end{array} \begin{bmatrix} \infty & \ldots & \infty \\ 1 & \ldots & 1 \\ \infty & \ldots & \infty \end{bmatrix}$$

The cost of a goto transition $cost(s, A)$ is characterized by the following equations:

$$cost(s,A) = \bigoplus_{A \to \alpha} \begin{cases} let\ s' = target(s,\alpha)\ in \\ cost(s, A \to \epsilon \bullet \alpha) \cdot \\ \Delta reduce(s', A \to \alpha) \end{cases} \quad (1)$$

$$cost(s, A \to \alpha \bullet x\beta) = \begin{cases} cost(s,x) \cdot \\ cost(target(s,x), A \to \alpha x \bullet \beta) \end{cases} \quad (2)$$

$$cost(s, A \to \alpha \bullet \epsilon) = I_T \quad (3)$$

Figure 1. A Characterization of Transition Costs

Equation 1. Equation 1 expresses the fact that the cost of the transition (s, A) is the minimum, over all productions $A \to \alpha$, of the costs of the paths that begin in state s, follow a sequence of transitions labeled α, and reach a state where $A \to \alpha$ can be reduced.

Equation 1 involves the auxiliary matrix $cost(s, A \to \alpha \bullet \beta)$, defined by Equations 2 and 3, which denotes the cost of a path that begins in state s, is labeled β, and reaches a state where the production $A \to \alpha\beta$ can be reduced.

Equation 1 involves a multiplication with the matrix ΔZ, where Z is the set of lookahead symbols that allow reducing $A \to \alpha$ in state s'. The matrix ΔZ, pronounced "filter Z", is defined as follows:

$$\begin{array}{ll} (\Delta Z)_{a,z} = 0 & \text{if } a = z \text{ and } a \in Z \\ (\Delta Z)_{a,z} = \infty & \text{otherwise} \end{array}$$

We note that ΔT is the identity matrix. In general, ΔZ is a variant of the identity matrix where the diagonal entries whose index lies outside of Z are set to ∞ instead of 0.

A multiplication $M \cdot \Delta Z$ produces a copy of the matrix M where all columns whose index lies outside Z are set to ∞. The multiplication with ΔZ in Equation 1 reflects the idea that if reduction is permitted, then it has zero cost, since no input symbol is consumed; if it is not permitted, then it has infinite cost; and whether it is permitted depends on the lookahead symbol.

Equations 2 and 3. Equation 2 concerns the case of a nonempty path, labeled with the sentential form $x\beta$. The cost matrix for the first transition, which leaves the state s and is labeled x, and the cost matrix for the remaining transitions, which begin in the state $target(s, x)$[2] and follow the path labeled β, are combined via matrix multiplication. Indeed, multiplication expresses the idea that the "lookahead symbol" that is chosen while examining the first transition must also be the "first symbol" that is chosen while examining the remaining transitions.

Equation 3 concerns the case of an empty path, labeled with the sentential form ϵ. The cost matrix for this path is the identity matrix I_T.

Recursion. Equations 1–2 are mutually recursive. They are monotone with respect to the ordering $(\bar{\mathbb{N}}, \leq)$, lifted pointwise to matrices. That is, when the cost matrices that appear in the right-hand side of an equation grow, the value of this right-hand side grows as well. This ensures that Equations 1–3 have least and greatest solutions. The desired cost matrices form the least solution.

3.3 Computing Transition Costs

Equations 1–3 form a system of monotone equations whose variables are matrices of costs and whose right-hand sides involve operations on matrices, such as multiplication $\cdot$ and minimum $\oplus$.

To solve these equations, one approach is to use a generic least fixed point computation algorithm: see, e.g., Pottier [26] and references therein. However, such an approach is very naïve. Indeed, working at the level of matrices is too coarse-grained: when a matrix is updated, every right-hand side where this matrix appears must be re-evaluated. Thus, even a tiny update (one that affects only a few matrix entries) requires a large amount of computation.

A better approach is to work at the level of matrix entries. Equations 1–3 can be reformulated as a system of equations whose variables are costs and whose right-hand sides involve operations on costs, such as minimum and addition. The benefit of such a reformulation is two-fold. First, this reduces

[2] It may be the case that there is no transition labeled x out of the state s, in which case $target(s, x)$ is undefined. Then, we consider that the right-hand side of Equation 2 yields a matrix where every entry is ∞.

the amount of computation that a generic fixed point computation algorithm must perform. Second, because addition is a superior function,[3] the reformulated problem is of a specific form that can be efficiently solved by Knuth's generalization of Dijkstra's algorithm [16]. This algorithm is potentially more efficient than a generic fixed point algorithm, and its worst-case time complexity is easier to analyze.

Unfortunately, this improved approach remains naïve: its space and time complexity are quite bad.

The space complexity of this approach can be assessed as follows. The number of matrices of the form $cost(tr)$ is $|Tr|$, the number of transitions in the automaton. The number of matrices of the form $cost(s, A \to \alpha\bullet\beta)$ is the total star size $\mathcal{S}$ [27]. We believe that $|Tr| \leq \mathcal{S}$ holds. Thus, the number V of cost variables is $O(\mathcal{S} \times |T|^2)$. This is also the space required by the cost matrices alone.

Provided a suitable "priority queue" data structure is used, the time complexity of Knuth's algorithm is $O(V \log V + E)$ where E is the number of dependencies between variables that the equations exhibit. Here, some variables participate in B dependency relations, where B is the maximum number of productions associated with each nonterminal symbol; some participate in $|T|$ relations. Assuming that B is smaller than $|T|$, which is usually true in practice, we find that E is $O(V \times |T|)$. Assuming that $\log \mathcal{S}$ is smaller than $|T|$, which is usually true in practice, we find that $V \log V$ is dominated by E, so the time complexity of this approach is $O(\mathcal{S} \times |T|^3)$. This analysis seems supported by our experiments (§7).

In practice, what does this mean? The largest grammar in our test suite, a grammar for C++ [12], has 422 terminal symbols and 755 nonterminal symbols. The LR(1) automaton produced by the Menhir parser generator has over 10^4 states and 5×10^5 transitions. Its total star size $\mathcal{S}$ is over 2×10^6. This is the number of cost matrices that we wish to compute. Since the size of a matrix is $422^2 = 1.78 \times 10^3$, the total size of the matrices is over 3.63×10^{11}. Assuming that a matrix entry occupies a 64-bit word, this requires at least 2.6 terabytes of memory. Considering today's economic constraints on the price and availability of memory, this is not realistic: the naïve algorithm does not scale well. The optimizations that we describe reduce the space consumption of the algorithm, in this case, by a factor of more than 10^4.

4 Overview of Optimizations

Four main ideas allow us to reduce the space and time requirements of the naïve algorithm, namely: *merging of identical columns*, *merging of peer rows*, *matrix chain multiplication optimization*, and *maximal sharing of matrix multiplications*. These optimizations reduce the size of the cost matrices that we wish to compute; therefore, the computation time is reduced as well. The computation itself is performed in the manner that was suggested (§3.3), that is, by formulating the problem as a system of equations bearing on cost variables, and by using Knuth's algorithm to solve these equations.

Merging Identical Columns. In general, at each step, the lookahead symbol determines the behavior of the LR(1) automaton: to shift a terminal symbol, to reduce a production, or to report a syntax error. However, it is often the case that two distinct lookahead symbols dictate the same behavior. For instance, if the terminal symbols z_1 and z_2 are both members of the set $reduce(s', A \to \alpha)$, then, in state s', the distinction between these symbols is irrelevant: both allow reducing the production $A \to \alpha$. This implies that some cost matrices have repeated columns. Our first optimization is to *precompute which columns must be identical* and *merge* them ahead of time so that they are computed only once.

Merging Peer Rows. Suppose that the cost computation involves a matrix multiplication $M \cdot M'$, which arises as an instance of Equation 2. Thanks to the previous optimization, we may happen to know ahead of time that the columns indexed by b_1 and b_2 in the matrix M must be equal. This implies that the rows indexed by b_1 and b_2 in the matrix M' participate in the multiplication as *peers*: they always appear in an expression of the form $(M_{a,b_1} \otimes M'_{b_1,c}) \oplus (M_{a,b_2} \otimes M'_{b_2,c})$, where the equality $M_{a,b_1} = M_{a,b_2}$ holds, so this expression is equal to $M_{a,b_1} \otimes (M'_{b_1,c} \oplus M'_{b_2,c})$. Thus, to compute the result of the multiplication, it is not necessary to have separate access to the rows $M'_{b_1,\_}$ and $M'_{b_2,\_}$. Instead, it suffices to have access to the combination of these rows by the function "minimum", that is, $M'_{b_1,\_} \oplus M'_{b_2,\_}$. This indicates that, even though these rows are not identical, they can be merged. Instead of allocating space for two rows, one can allocate space for a single row and use it to store $M'_{b_1,\_} \oplus M'_{b_2,\_}$.

This reasoning assumes that the matrix M' participates in only one multiplication, namely $M \cdot M'$. If M' participates in several multiplications (a situation that typically arises when a state has several incoming transitions) then the rows $M'_{b_1,\_}$ and $M'_{b_2,\_}$ can be merged only if every matrix that appears on the left-hand side of a multiplication $\_ \cdot M'$ is known to have identical columns at indices b_1 and b_2.

Our second optimization, therefore, is to *determine ahead of time which rows are peers* and *merge* them.

Matrix Chain Multiplication Optimization. Once certain columns and rows have been merged, as described above, we no longer work with cost matrices whose dimensions are $|T|$ by $|T|$. Instead, we compute compact cost matrices, which are rectangular and can have various dimensions. Therefore, our third optimization is to *identify chains of multiplications* and *optimize each such chain* by exploiting the associativity of multiplication. A standard dynamic programming algorithm is used to solve the Matrix Chain Ordering Problem [4, §15.2] [34].

As a consequence of this optimization, we do not necessarily compute the auxiliary matrices $cost(s, A \to \alpha\bullet\beta)$

[3] A function f of type $\bar{\mathbb{N}} \times \bar{\mathbb{N}} \to \bar{\mathbb{N}}$ is superior if $f(x, y) \geq \max(x, y)$ holds.

described in the previous section. Because we optimize the chain of multiplications expressed by Equation 2, we may go through a different family of auxiliary matrices, while preserving the end result of the chain, namely the matrix $cost(s, A \to \epsilon\bullet\alpha)$, which appears in Equation 1.

Maximal Sharing of Matrix Multiplications. Often, two paths of interest in the automaton share a suffix. This in turn means that two matrix multiplication chains share a suffix.

For instance, suppose $target(s_1, \alpha) = target(s_2, \alpha) = s'$, that is, the path labeled α out of s_1 and the path labeled α out of s_2 lead to a common state s'. Suppose that there exists a production $A \to \alpha\beta$ and that the states s_1 and s_2 have outgoing transitions labeled A. Then, we would like to compute the cost matrices $cost(s_1, A \to \epsilon\bullet\alpha\beta)$ and $cost(s_2, A \to \epsilon\bullet\alpha\beta)$. Each of these matrices is the result of a multiplication chain, and these two chains share a common suffix: indeed, both of them involve the matrix $cost(s', A \to \alpha\bullet\beta)$.

This raises a question: when we optimize two multiplication chains that share a suffix, should we optimize each chain separately, ignoring the presence of a shared suffix, or should we decompose the two chains into three subchains (two prefixes and a shared suffix) that do not overlap and can be optimized independently? The first option yields greater freedom in optimizing long multiplication chains, but destroys sharing that is present in the initial formulation of the problem and thereby potentially creates redundant computation. The second option preserves sharing but gives up opportunities of optimizing long chains of multiplications.

After experimenting with both options, we choose the first one, which seems most beneficial. To make up for the loss of sharing that it causes, *we explicitly impose maximal sharing:* that is, after each multiplication chain has been separately optimized, we use hash-consing [10, 32] to discover matrix multiplication trees that appear several times, and share them again. This is our fourth and last major optimization.

In the next section (§5), we describe the first two key optimizations, namely the merging of rows and columns, which allows us to compute compact cost matrices. Then (§6), we describe some implementation details that lead to increased efficiency in practice.

5 Merging Rows and Columns

The cost matrices have dimension $|T| \times |T|$: their rows and columns are indexed with terminal symbols. In this section, we would like to merge certain rows and certain columns. We wish to do so a priori: that is, before any matrices are constructed, we wish to compute which rows and which columns can be merged. This allows us to directly construct *compact cost matrices* whose rows and columns are indexed by *equivalence classes* of terminal symbols.

We proceed as follows:

1. For each transition (s, x), we compute two partitions, named $first(s)$ and $follow(s, x)$. The partition $first(s)$ tells which rows in the cost matrix associated with this transition can be merged; the partition $follow(s, x)$ tells which columns can be merged.
2. For each transition (s, x), we compute a compact cost matrix $ccost(s, x)$ whose row indices range over $first(s)$ and whose column indices range over $follow(s, x)$. An entry in this matrix is identified by two equivalence classes $p \in first(s)$ and $q \in follow(s, x)$.

These two phases are described in the next two subsections.

5.1 Which Rows and Columns Can Be Merged?

To begin, we define the functions $first : S \to \mathrm{Part}(T)$ and $follow : Tr \to \mathrm{Part}(T)$. These functions determine which columns and rows we can safely merge.

As explained earlier (§4), we merge two columns when we know ahead of time that they would be identical. Thus, we need *follow* to satisfy the following soundness property:

$$\forall q \in follow(tr) \quad \forall b,\ b' \in q \quad \forall a \quad cost(tr)_{a,b} = cost(tr)_{a,b'}$$

That is, if the partition $follow(tr)$ indicates that the terminal symbols b and b' are equivalent (members of the same class), then the columns indexed by b and b' in the matrix $cost(tr)$ must be identical. The reverse implication is not true: the columns indexed by b and b' may happen to be identical even when this is not predicted by the partition $follow(tr)$, that is, even when b and b' are not members of the same class. Thus, the partition $follow(tr)$ is a conservative prediction of which columns will be identical.

Another way to think about $follow(tr)$ is that this partition indicates which distinctions between lookahead symbols definitely do not influence how the transition tr can be taken.

It does not seem easy to state a soundness property that *first* must satisfy. As explained earlier (§4), two rows are merged not when they are identical, but when we know in advance that only the combination of these rows by the function "minimum" is relevant. So, we state no such property.

The partitions $first(s)$ and $follow(s, x)$ are characterized by a set of recursive monotone equations: they form the greatest fixed point of these equations. Any fixed point is sound; the greatest fixed point allows the greatest amount of merging. The equations are as follows:

$$follow(s, a) = \top \tag{4}$$

$$follow(s, A) = \bigwedge_{A \to \alpha} \begin{cases} let\ s' = target(s, \alpha)\ in \\ let\ Z = reduce(s', A \to \alpha)\ in \\ first(s') \downarrow Z \end{cases} \tag{5}$$

$$first(s) = \bigwedge_{tr \in incoming(s)} follow(tr) \tag{6}$$

Figure 2. Which Rows and Columns Can Be Merged

Equation 4 indicates that the partition $follow(s, a)$ is $\top$. This is sound because all columns in the matrix $cost(s, a)$ are identical (§3.2). An equivalent statement is that $follow(tr)$ is $\top$ when tr is a shift transition.

Equation 5. Equation 5 defines $follow(s, A)$, where we assume that the state s has an outgoing transition labeled A. In doing so, it aims to answer the question: "which lookahead symbols influence the manner in which this transition can be taken?". If two symbols b and b' lie in the same class, then the input sentences that allow taking this transition are the same, regardless of whether the lookahead symbol is b or b'.

This transition can be taken if there is a production $A \to \alpha$ such that (1) the sentential form α can be recognized, taking the automaton from state s to state $s' = target(s, \alpha)$, and (2) in state s', the production $A \to \alpha$ can be reduced. Any such production can be chosen, hence Equation 5 involves a meet over all such productions. Let us now focus on one such production and examine the right-hand side of Equation 5.

In the state s', reducing the production $A \to \alpha$ is permitted if and only if the lookahead symbol is a member of the set $reduce(s', A \to \alpha)$. Let us refer to this set as Z. Then, there definitely is *no need to distinguish between nonmembers of* Z: indeed, all lookahead symbols outside Z forbid reducing the production $A \to \alpha$. This explains why a partition of the form "$\ldots \downarrow Z$" appears in Equation 5.

There remains to explain what partition to choose in place of "$\ldots$". This determines which terminal symbols inside Z must be distinguished.

Choosing $\bot$ means that all symbols inside Z must be distinguished. That would be sound but pessimistic, and would largely prevent the merging of rows and columns.

Choosing $\top$ means that no distinction inside Z is necessary. That would be unsound. Indeed, before reducing $A \to \alpha$, the right-hand side α must be recognized, and this process may be influenced by the lookahead symbol.

What partition describes this influence in a sound way, and is not as pessimistic as $\bot$? We remark that, in order to go from the state s along the path labeled α to the state s', the automaton must (as its final step) enter the state s', and to do so, it must follow some transition $tr \in incoming(s')$. Therefore, two lookahead symbols must be distinguished only if this distinction influences some transition $tr \in incoming(s')$. In other words, a sound choice is the meet, over all such transitions, of the partitions $follow(tr)$. This happens to be precisely $first(s')$.

Equation 6. Equation 6 defines the partition $first(s)$ as the meet of the partitions $follow(tr)$, where tr ranges over the incoming transitions of the state s. The reason for this is as follows. The partition $first(s)$ is supposed to determine which rows in a cost matrix $cost(s, x)$ can be merged. As explained earlier (§4), two rows in such a matrix can be merged if this matrix is always multiplied (on its left) with matrices where the corresponding columns are identical. This is the case if every incoming transition tr of state s has a cost matrix $cost(tr)$ where these columns are identical. Thus, for every $tr \in incoming(s)$, we need $first(s) \preceq follow(tr)$ to hold. This explains Equation 6.

Because all of the incoming transitions of a state s carry the same symbol, either all of them are shift transitions, labeled with a terminal symbol a, or all of them are goto transitions, labeled with a nonterminal symbol A. In the former case, Equation 6 can be simplified: by Equation 4, every incoming transition tr satisfies $follow(tr) = \top$, so $first(s)$ is $\top$.

5.2 A Characterization of Compact Cost Matrices

Compact Cost Matrices. For each transition (s, x), we wish to compute a compact cost matrix $ccost(s, x)$, whose row indices are members of $first(s)$ and whose column indices are members of $follow(s, x)$.

We now provide several equations that characterize these matrices, and allow them to be computed. These equations involve a family of auxiliary matrices $ccost(s, A \to \alpha\bullet\beta)$ of dimensions $first(s)$ by $first(target(s, \beta))$.

The compact cost matrix $ccost(s, a)$ associated with a shift transition is defined as follows:

$$\begin{array}{ll} ccost(s, a)_{p,q} = 1 & \text{if } a \in p \\ ccost(s, a)_{p,q} = \infty & \text{otherwise} \end{array}$$

Since $follow(s, a)$ is $\top$, a one-class partition, this matrix has width one: it is a column matrix. It has exactly one "1" entry, found at the unique index p such that $a \in p$ holds, and "∞" entries everywhere else.

The compact cost matrix $ccost(s, A)$ associated with a goto transition and the auxiliary matrices $ccost(s, A \to \alpha\bullet\beta)$ are defined as follows:

$$ccost(s, A) = \bigoplus_{A \to \alpha} \begin{cases} ccost(s, A \to \epsilon\bullet\alpha) \cdot \\ creduce(s, A \to \alpha) \end{cases} \tag{7}$$

$$ccost(s, A \to \alpha\bullet x\beta) = \begin{cases} let\ s' = target(s, x)\ in \\ ccost(s, x) \cdot \\ coerce(follow(s, x), first(s')) \cdot \\ ccost(s', A \to \alpha x\bullet\beta) \end{cases} \tag{8}$$

$$ccost(s, A \to \alpha\bullet\epsilon) = I_{first(s)} \tag{9}$$

Figure 3. A Characterization of Compact Cost Matrices

Equation 7. Equation 7 resembles Equation 1 in Figure 1, but the square matrix $\Delta reduce(target(s, \alpha), A \to \alpha)$ that is visible in Equation 1 is replaced with the rectangular matrix $creduce(s, A \to \alpha)$, whose dimensions are $first(target(s, \alpha))$

by $follow(s, A)$. This matrix is defined as follows:

$$\begin{array}{ll} creduce(s, A \to \alpha)_{p,q} = 0 & \text{if } q \subseteq Z \wedge q \subseteq p \\ & \text{where } Z = reduce(s', A \to \alpha) \\ & \text{and } s' = target(s, \alpha) \\ creduce(s, A \to \alpha)_{p,q} = \infty & \text{otherwise} \end{array}$$

This generalizes the definition of the matrix ΔZ to a setting where, instead of two terminal symbols a and z, we consider two classes $p \in first(target(s, \alpha))$ and $q \in follow(s, A)$.

It is worth noting that, by virtue of Equation 5, we have $follow(s, A) \preceq Z?$. This implies that q must lie either entirely within Z or entirely outside of Z. Thus, the test $q \subseteq Z$ is cheap: it suffices to pick an arbitrary element of q and to test whether it is a member of Z.

Also by virtue of Equation 5, we find that $follow(s, A)$ refines $first(target(s, \alpha)) \downarrow Z$, from which we can deduce that $follow(s, A) \downarrow Z$ refines $first(target(s, \alpha)) \downarrow Z$. This implies that the condition $q \cap Z \subseteq p \cap Z$ can be decided by picking an arbitrary element of q and testing whether it is a member of $p \cap Z$. Assuming that $q \subseteq Z$ holds, this can be rephrased as follows: $q \subseteq p$ can be decided by picking an arbitrary element of q and testing whether it is a member of p.

Equation 8. Equation 8 is analogous to Equation 2, but a *coercion matrix* must be inserted in the middle of the matrix multiplication expression, because the matrices $ccost(s, x)$ and $ccost(target(s, x), A \to \alpha x \bullet \beta)$ do not have matching numbers of columns and rows. The columns of the former matrix are indexed with classes in $follow(s, x)$, while the rows of the latter matrix are indexed with classes in $first(target(s, x))$. There is a refinement relation between these partitions: as a consequence of Equation 6, we have $first(target(s, x)) \preceq follow(s, x)$. In such a situation, it is possible to construct a coercion matrix, whose definition and meaning are explained next.

Coercion Matrices. Let $P, Q \in \mathrm{Part}(T)$ be two partitions such that $Q \preceq P$ holds, that is, Q refines P. The matrix $coerce(P, Q)$, whose rows are indexed with classes in P and whose columns are indexed with classes in Q, is defined as follows:

$$coerce(P, Q)_{p,q} = \begin{cases} 0 & \text{if } q \subseteq p \\ \infty & \text{otherwise} \end{cases}$$

Because Q refines P, every class $q \in Q$ is a subset of some class in P. Thus, for every $p \in P$ and $q \in Q$, the class q is a subset of either p or $T \setminus p$. Thus, the condition $q \subseteq p$ can be decided by testing an arbitrary element of q.

In the special case where P and Q are the same partition, the coercion matrix $coerce(P, Q)$ is square: it is an identity matrix. Otherwise, the matrix $coerce(P, Q)$ is rectangular: it has more columns than rows. The row identified by the class p has a "0" entry at each column q such that the class q is a piece of the class p, and "∞" entries everywhere else. Since every class $q \in Q$ is a subset of exactly one class in P, on every column, there is exactly one "0" entry.

The matrix $coerce(P, Q)$ has the remarkable property that, when used in a matrix multiplication, it serves as an adapter: it can expand a matrix from dimension P to dimension Q and shrink a matrix from dimension Q to dimension P.

Indeed, the product $M \cdot coerce(P, Q)$ expands the matrix M from width P to width Q by duplicating columns: the column found at index p in the matrix M is copied in the matrix $M \cdot coerce(P, Q)$ at every index q such that $q \subseteq p$ holds.

In the opposite direction, the product $coerce(P, Q) \cdot M$ shrinks the matrix M from height Q to height P by merging rows: the row found at index p in the product $coerce(P, Q) \cdot M$ is the pointwise minimum of the rows found in the matrix M at some index q such that $q \subseteq p$ holds.

Soundness of Compact Cost Matrices. The cost matrix $cost(s, x)$ and the compact cost matrix $ccost(s, x)$ are related as follows:

Claim 5.1. *Let (s, x) be a transition. Then, we have:*

$$\begin{array}{r} coerce(first(s), \bot) \cdot cost(s, x) = \\ ccost(s, x) \cdot coerce(follow(s, x), \bot) \end{array}$$

The multiplication by $coerce(follow(s, x), \bot)$ on the second line duplicates some columns of the compact cost matrix, so as to undo the merging of identical columns.

The multiplication by $coerce(first(s), \bot)$ on the first line compresses some rows of the cost matrix by computing their pointwise minimum. This reflects the fact that the compact matrix contains less information than the cost matrix: some information is lost when peer rows are merged.

6 Implementation Details

We now propose an overview of several implementation details and optimizations which, together, yield significant additional performance improvements.

6.1 Representing Partitions

The computation of *first* and *follow* partitions (§5.1) requires an efficient representation of partitions of the set T of the terminal symbols. Several options come to mind. Following Lee [18], one could represent a partition as an array of size $|T|$, where each terminal symbol is mapped to a representative element of its block. Or, following Hopcroft [13] and Paige and Tarjan [25], one could adopt the mutable data structures used in partition refinement algorithms, which involve doubly-linked lists of elements and doubly-linked lists of blocks. In our setting, a persistent and compact representation is desirable. We represent a partition as an (unordered) linked list of blocks, and represent a block as a sparse bit set.[4] In

[4] Whereas an ordinary bit set would be represented as an array of machine words and would have size $O(|T|)$, regardless of its cardinality, a sparse bit set is represented as linked list of pairs of an integer offset and a machine word, and has size $O(n)$, where n is the cardinality of the bit set.

the worst case, a partition occupies $O(|T|)$ space in memory; yet, in several common cases, it can require significantly less space. Indeed, we allow one block to be omitted from the linked list of all blocks. This causes no loss of information: the omitted block is the complement of the blocks that do appear in the list. For example, in the representation of the partitions $Z?$ and $Z!$, the block $T \setminus Z$ can be omitted. Thus, these partitions occupy only $O(|Z|)$ space.

Our implementation of sparse bit sets is able to efficiently compute a "unique-or-shared prefix" of two nonempty bit sets Z_1 and Z_2. Let us write $Z < Z'$ when every element of Z is less than every element of Z', according to an arbitrary, fixed total order on T. Then, the unique-or-shared-prefix operation produces either (a) a nonempty subset Z_1' of Z_1 such that $Z_1' < Z_1 \setminus Z_1'$ and $Z_1' < Z_2$ are satisfied; or (b) symmetrically, a nonempty subset Z_2' of Z_2 such that $Z_2' < Z_1$ and $Z_2' < Z_2 \setminus Z_2'$ are satisfied; or (c) a nonempty subset of $Z_1 \cap Z_2$ such that $Z_1' < Z_1 \setminus Z_1'$ and $Z_2' < Z_2 \setminus Z_2'$.

This operation is used in the construction of a carefully optimized implementation of $\bigwedge$, the meet of a family of k partitions. This algorithm is inspired by the standard k-way merge algorithm [33], whose main loop involves a priority queue.

In our setting, the priority queue is populated with pairs of a block and its provenance. The priority of such a pair is the minimum element of the block. A provenance is a set of identifiers, where initially each block in each of the k partitions receives a unique identifier. Pairing a block with a provenance allows us to record which initial blocks this block is a subset of. When the priority queue is first populated, the provenance of each block is a singleton set, which contains just the identifier of this block.

The algorithm's main loop proceeds as follows. As long as the queue has size greater than one, the two blocks with minimum priority, say Z_1 and Z_2, are extracted, and their unique-or-shared prefix, say Z, is computed. If it is unique, then it is set aside, together with its provenance; if it is shared, then it is inserted into the queue again with an updated provenance, the union of the provenances of Z_1 and Z_2. In either case, the remainders of the two blocks, namely $Z_1 \setminus Z$ and $Z_2 \setminus Z$, are inserted into the queue (if they are nonempty).

Once this loop terminates, the blocks that have been set aside are sorted according to their provenance, and those that have the same provenance are merged. The list of blocks thus obtained is the meet of the k input partitions.

6.2 Approximating *first* and *follow* Partitions

We have presented the computation of *first* and *follow* as the computation of a greatest fixed point (§5.1). A standard fixed point computation algorithm [26], based on a chaotic iteration scheme, can be used to compute this fixed point in an exact way. In our experience, however, chaotic iteration can be slow, so this computation can become a bottleneck. To address this issue, we use a faster algorithm, which visits each cost variable at most twice and computes a post fixed point, that is, a solution of the inequations obtained by replacing equality "=" with the refinement relation "$\preceq$" in Equations 7–9. Because every post fixed point refines the greatest fixed point, the partitions that we obtain are an under-approximation of *first* and *follow*. Such an approximation does not endanger the correctness of our approach; it just means that we do not merge rows and columns as aggressively as we could with the greatest fixed point.

Our algorithm examines the dependency graph formed by all instances of Equations 7–9, computes its strongly connected components, and processes one component at a time, in topological order. Inside each component, it proceeds as follows. By inspection of Equations 7–9, one can see that the partition variables that belong in this component form two families $X_1, \ldots, X_m$ and $Y_1, \ldots, Y_n$[5] and that the inequations that these variables must satisfy are of four kinds:

$$\begin{array}{lll} X_i \preceq P_i & \text{where } P_i \text{ is a constant partition} & (A) \\ X_i \preceq Y_j \downarrow Z_{ij} & \text{where } Z_{ij} \text{ is a constant set} & (B) \\ Y_j \preceq Q_j & \text{where } Q_j \text{ is a constant partition} & (C) \\ Y_j \preceq X_i & & (D) \end{array}$$

Our algorithm solves these constraints as follows:

1. Compute $P = \bigwedge_i P_i \wedge \bigwedge_{ij} Z_{ij}? \wedge \bigwedge_j Q_j$.
2. Assign to each X_i the value $P_i \wedge \bigwedge_j (P \downarrow Z_{ij})$.
3. Assign to each Y_j the value $Q_j \wedge \bigwedge_i X_i$.

It is not difficult to check that this assignment satisfies all of the inequations A–D. The second step is where each X_i is assigned an under-approximation of its ideal value; in the third step, each Y_j receives a value as accurate as possible, considering the approximation made in the previous step.

In our experience, this approximation leads to an increase of less than 2% in the size of the compact cost matrices, due to the fact that the merging of rows and columns is no longer as aggressive as it could be. This does not lead to an observable slowdown in the computation of compact cost matrices. On the other hand, the approximate computation of *first* and *follow* can be 10 times faster than the exact computation. As a result, it is no longer a bottleneck.

6.3 Replacing Knuth's Priority Queue with a FIFO

We have indicated earlier (§3.3) that the computation of the compact cost matrices (§5.2) can exploit Knuth's algorithm [16], which has known asymptotic worst-case complexity. This is attractive, because it makes it easy to assess the time complexity of our algorithm (§3.3). However, we find that a FIFO queue offers similar performances in practice while being simpler to implement. Knuth's algorithm guarantees that every matrix entry is visited at most once, so that, when an entry is visited, it is assigned its final value. By using a

[5] The X_i's correspond to *follow* partitions, while the Y_j's correspond to *first* partitions.

FIFO queue instead of a priority queue, we lose this property: an entry may be visited several times. In practice, we find that entries are visited 1.1 times on average. This is a small price to pay. In return, a FIFO queue is easier to implement and can be (marginally) more compact and faster.

6.4 Representing Dependencies

During the computation of compact cost matrices (§5.2), an efficient representation of the dependencies between matrix entries is needed, so that, when an entry is updated, the entries that depend on it can be enqueued for re-examination. There is a space-time trade-off between an explicit representation, where the dependency edges form a graph that is explicitly stored in memory, and an implicit representation, where the dependency edges are not stored in memory, but can be computed on demand.

The number of matrix entries can be huge and and is the dominating factor that determines the memory requirement of this computation. Thus, we strive to maintain as little information per matrix entry as possible, and cannot use a fully explicit representation of dependency edges.

To every entry in every compact cost matrix, we assign a unique number. Then, with each entry, we associate three pieces of information, which occupy three words in memory. One piece is the content of this matrix entry, that is, a cost, represented as a machine integer. The second piece is a reference to the matrix to which this entry belongs. (Because entries are numbered sequentially and because the dimensions of every matrix are known, this information suffices to also recover the coordinates of this entry within its matrix.) The third and last piece is either *null* or the number of another matrix entry: this way, the FIFO queue of all "dirty" matrix entries is represented as a linked list.

To encode the dependencies between matrix entries, we combine a coarse-grained explicit scheme and a fine-grained implicit scheme. At the outer level, dependencies between matrices are explicitly stored in memory. At the inner level, dependencies between matrix entries are computed on demand. Given the coordinates of an entry and the (outer-level) dependencies of the matrix in which it appears, we are able to enumerate the matrix entries that depend on this entry.

6.5 Representing Constant Matrices

In general, a cost matrix is represented as an array of entries. This can be quite expensive in terms of space. We identify several cases where a specialized representation is beneficial.

The matrices $ccost(s, a)$ are column matrices where a single row contains a "1" entry. We store just the index of this row.

The auxiliary matrices $ccost(s, A \longrightarrow \alpha\bullet\epsilon)$ need not be represented at all. Indeed, they are identity matrices. If they appear in a multiplication, then this multiplication can be simplified. If they appear in a minimum $\oplus$, then the result matrix is eagerly initialized with appropriate "0" entries.

The coercion matrices that appear in products of the form $M_1 \cdot coerce(P, Q) \cdot M_2$ have two (redundant) representations. One representation allows an efficient propagation of an update in M_1 to the result of the multiplication; the other representation allows an efficient propagation of an update in M_2. Both are sparse representations of the coercion matrix; one representation offers efficient access to the "0" entries along every row; the other offers efficient access to the "0" entries along every column.

Because the matrices $creduce(s, A \longrightarrow \alpha)$ always appear on the right-hand side of a matrix multiplication, a single sparse representation, which allows efficient access to the "0" entries along every row, suffices.

6.6 Testing Block Inclusion

The construction of the *creduce* and *coerce* matrices (§5.2) involves many tests of the form $q \subseteq p$, where it is known that either $q \subseteq p$ or $q \cap p = \emptyset$ must hold. Indeed, q and p are blocks in two partitions Q and P such that $Q \preceq P$. Therefore, the condition $q \subseteq p$ can be decided by picking an arbitrary member of q and testing whether it is a member of p.

7 Experimental Evaluation

7.1 Comparison of Three Algorithms

We compare the performance of three algorithms, namely Pottier's algorithm [27], dubbed "P", the naïve algorithm based on cost matrices (§3), dubbed "CM", and our optimized algorithm based on compact cost matrices (§5), dubbed "CCM". We run each algorithm on the 394 grammars of Menhir's test suite. This suite contains only a handful of small artificial grammars; the majority of the grammars in it are real-world grammars, ranging from small to large grammars. The tests are run sequentially and are repeated 5 times; only the best time is kept. We use an Intel Xeon E7-4870 machine running at 2.4Ghz, with 1TB of RAM.

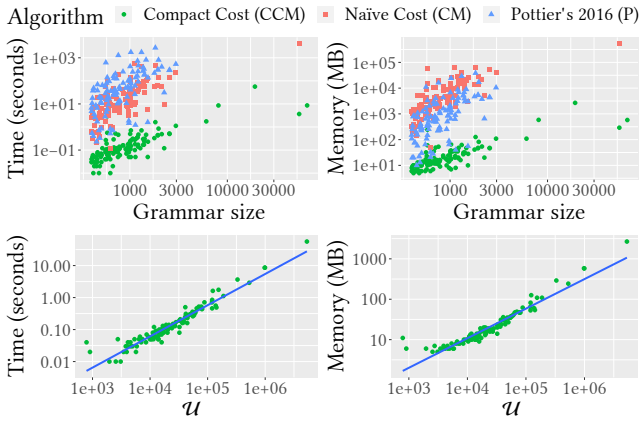

Figure 4. Performance Aspects of Reachability Algorithms

Our performance results are summarized in Figure 4 in four graphs. Each graph uses a logarithmic scale along each axis.

The top-left and top-right graphs show the time and memory consumption of the three algorithms. To reduce the noise, only grammars whose size[6] is larger than 400 are taken into account when an average[7] is computed. A first finding is that CM can be competitive with P: on average, it is 3 times faster, but consumes 5 times more memory. A second finding is that CCM is the clear winner. On average, it is 95 times faster and uses 215 times less memory than CM. Compared with P, it is on average 300 times faster and uses 42 times less memory. In fact, the larger the grammar, the greater the gain: our algorithm can be up to 2700 times faster than Pottier's, and can use as little as 470 times less memory.

It is worth mentioning that, as a result of its much smaller time and space requirements, CCM is able to handle much larger grammars than P and CM.

The bottom-left and bottom-right graphs in Figure 4 are an attempt to predict CCM's time and memory requirements. We find that both are correlated[8] with the parameter $\mathcal{U}$, which we define as follows:

$$\mathcal{U} = \sum_A edges(A) * prodsize(A)$$

where $edges(A)$ is the number of transitions labeled A and $prodsize(A)$ is the sum of the lengths of the productions associated with the nonterminal symbol A.

7.2 Impact of Successive Optimizations

Table 1 shows the impact of several optimizations (§4) that form a path from Algorithm CM to Algorithm CCM. The three optimizations that we consider are: (1) merging rows and columns, (2) reordering matrix products, and (3) sharing common subexpressions in matrix products. Each optimization requires the previous one: reordering applies after merging; sharing applies after reordering.

As explained earlier (§4), before applying optimizations (1), (2) and (3), we take a first step, namely: (0) abandon the sharing of cost matrices that is inherent in Equations 1–3. This has a negative impact in terms of space and time, but opens the way to the steps that follow, which have positive impact. The four lines in Table 1 show the impact of steps (0), (1), (2), (3), respectively.

We measure the impact of each optimization on the number of matrix entries, V, and on the number of dependencies between matrix entries, E. These quantities correspond to the parameters V and E that appear in the complexity analysis of Knuth's algorithm (§3.3). The parameter V is directly related to the memory requirement of Knuth's algorithm, while the parameter E is directly related to its time requirement. Algorithm CM is an instance of Knuth's algorithm, so its space and time complexity is directly related with V and E. Algorithm CCM consists of several pre-computation phases (namely merging, reordering, sharing), followed with a run of Knuth's algorithm. The complexity of this last phase is still directly related with V and E. We find that, in practice, the pre-computation phases and the last phase have roughly comparable execution time.

Table 1. Impact of Successive Optimizations

	Impact on V	Impact on E
Unsharing	on avg. $\times 1.53$ up to $\times 2.59$	on avg. $\times 2.08$ up to $\times 3.72$
Merging	on avg. $/ 5.59 \times 10^3$ up to $/ 9.04 \times 10^4$	on avg. $/ 2.10 \times 10^5$ up to $/ 1.55 \times 10^7$
Reordering	on avg. $/ 0.89$ up to $/ 1.73$	on avg. $/ 2.17$ up to $/ 7.25$
Sharing	on avg. $/ 1.18$ up to $/ 1.92$	on avg. $/ 1.32$ up to $/ 2.52$

Each cell of Table 1 shows the impact of an optimization, that is, the ratio between a quantity before applying this optimization and this quantity after the optimization has been applied. For instance, the top left cell shows "on avg. $\times 1.53$". This cell appears in the column entitled "Impact on V". This means that, on average, the number of matrix entries is multiplied by 1.53 when sharing is abandoned in step (0). The cell below this one shows "on avg. $/ 5.59 \times 10^3$". This means that, on average, the number of matrix entries is divided by 5.59×10^3 when merging is applied in step (1). This cell also contains "up to $/ 9.04 \times 10^4$", which indicates that, in some cases, the number of matrix entries is divided by 9.04×10^4. Naturally, the impact of an optimization grows as grammars grow larger.

The second line of Table 1 shows that merging alone has a dramatic impact. The last two lines show that reordering and sharing both allow reaping an additional (albeit much smaller) performance improvement. The cumulative effect of all four steps can be computed by multiplying the factors shown in each column.

Interestingly, reordering slightly increases the number of matrix entries on average, as indicated by the number "$/ 0.89$" in the third line of the first column. Indeed, what is optimized is not the total size of the matrices that are being multiplied, but the number of elementary operations required by the multiplications. One can see, on the third line in the second column, that reordering has a positive impact on the parameter E.

8 Related Work

The reachability problem that we consider seems closely related to a shortest paths problem, which, as is well-known,

[6] The size of a grammar is the sum of the lengths of its productions.
[7] When an "average" is mentioned, a geometric mean is meant.
[8] Both have a Pearson correlation coefficient of 0.998 with $\mathcal{U}$.

can be expressed in terms of cost matrices [19]. It may be possible to reduce our reachability problem to a shortest paths problem in a suitably constructed graph. We have not attempted to do so; instead, we have expressed the reachability problem directly in terms of cost matrices, and we have proved that the equations that characterize these cost matrices (§3) can be solved by using Knuth's generalization [16] of Dijkstra's algorithm.

We speed up this naïve algorithm by remarking that, in the specific setting of the LR(1) reachability problem, many rows and columns of the cost matrices can be merged, letting us compute much smaller "compact cost matrices" (§4). Several additional optimizations help improve the algorithm's performance in practice (§6).

Pottier's reachability algorithm [27] seems related to the naïve algorithm that we have described: indeed, it uses a priority queue, very much in the same way as Knuth's algorithm [16], to compute and process a set of reachability facts. Its presentation, however, is not matrix-based, and it does not have the ability of handling several terminal symbols together, as an equivalence class: each reachability fact concerns just one terminal symbol.

The LR(1) reachability problem can be considered a special case of the reachability problem for pushdown systems, which has been thoroughly studied, as it has many applications in model-checking and in program analysis [3, 9, 17, 29, 30]. Although our merging technique is probably specific to LR(1), it would be interesting to draw a detailed comparison with this line of work.

Minamide and Mori [23] formalize an HTML5 parser as a conditional pushdown system [21], that is, a pushdown system extended with the ability to test whether the stack matches a regular expression. They propose a new reachability algorithm, based on earlier algorithms by Bouajjani et al. [3] and Esparza et al. [9]. They use this algorithm to prove that certain branches in the parser's code are dead, to prove that the parser never attempts to pop an item off an empty stack, and to generate a set of input sentences that achieves good coverage of the parser's code and can be used for testing the HTML5 parsers found in industrial browsers. Although their work is similar to ours in its motivation, the HTML5 parser that they consider is different from (and more complex than) an LR(1) parser. As a result, their algorithm is more costly: a 438-line fragment of the HTML5 grammar is analyzed in 82 minutes.

By speeding up the computation of the reachable states in an LR(1) parser, our algorithm makes Jeffery's approach to error diagnosis [14] more scalable. Indeed, as argued by Pottier [27], this approach requires computing which error states are reachable and which (minimal) erroneous input sentences lead to these states. As a grammar evolves over time, the maintainer of the grammar must re-compute this information over and over. In order to achieve a smooth edit-compile-debug cycle, a fast reachability algorithm is required. Although Pottier [27] successfully applies this technique to the CompCert C parser, his algorithm requires several minutes to process somewhat larger grammars, such as the OCaml grammar, which our algorithm handles in half a second, and is unable to process much larger grammars, such as the C++ grammar [12] that our algorithm handles in a little less than a minute.

A broad survey of error diagnosis techniques, and of the quality of the diagnostic messages produced by various techniques and tools, is offered by Becker et al. [2].

9 Conclusion

We present a new algorithm for the reachability problem in LR(1) parsers. This algorithm vastly outperforms the state of the art, which as far we know, is represented by Pottier's algorithm [27]. We have not proved the correctness of our algorithm, but have implemented it in the Menhir parser generator [28] and have experimentally verified, using a test suite of 390 grammars, that it produces the same results as Pottier's earlier algorithm.

We expect our new algorithm to make Jeffery's approach to error diagnosis [14] more scalable. Furthermore, we believe that a fast reachability algorithm has many potential applications beyond computing the reachable error states. In the future, we wish to investigate its application to more powerful error diagnosis schemes, to error recovery, and to syntactic completion.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. 1972. *The theory of parsing, translation, and compiling*. Prentice Hall.

[2] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 177–210.

[3] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 1243)*. Springer, 135–150.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (Third Edition)*. MIT Press.

[5] Joel E. Denny and Brian A. Malloy. 2010. The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution. *Science of Computer Programming* 75, 11 (2010), 943–979.

[6] Frank DeRemer and Thomas Pennello. 1982. Efficient Computation of *LALR*(1) Look-Ahead Sets. *ACM Transactions on Programming Languages and Systems* 4, 4 (1982), 615–649.

[7] Franklin Lewis DeRemer. 1969. *Practical Translators for LR(k) Languages*. Technical Report MIT-LCS-TR-065. Massachusetts Institute of Technology.

[8] Franklin L. DeRemer. 1971. Simple *LR*(*k*) grammars. *Commun. ACM* 14, 7 (1971), 453–460.

[9] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In

Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 1855). Springer, 232–247.

[10] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *ACM Workshop on ML*. 12–19.

[11] Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing techniques: a practical guide, second edition*. Springer.

[12] Masatomo Hashimoto. 2021. The Code Continuity Analysis Framework. (2021). https://github.com/codinuum/cca.

[13] John E. Hopcroft. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, Zvi Kohavi and Azaria Paz (Eds.). Academic Press, 189–196.

[14] Clinton L. Jeffery. 2003. Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems* 25, 5 (2003), 631–640.

[15] Donald E. Knuth. 1965. On the translation of languages from left to right. *Information & Control* 8, 6 (Dec. 1965), 607–639.

[16] Donald E. Knuth. 1977. A Generalization of Dijkstra's Algorithm. *Inform. Process. Lett.* 6, 1 (Feb. 1977), 1–5.

[17] Akash Lal and Thomas W. Reps. 2006. Improving Pushdown System Model Checking. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 4144)*. Springer, 343–357.

[18] Tony T. Lee. 1981. Order-Preserving Representations of the Partitions on the Finite Set. *Journal of Combinatorial Theory, Series A* 31, 2 (1981), 136–145.

[19] Daniel J. Lehmann. 1977. Algebraic Structures for Transitive Closure. *Theoretical Computer Science* 4, 1 (1977), 59–76.

[20] Xavier Leroy. 2021. The CompCert C compiler. http://compcert.org/.

[21] Xin Li and Mizuhito Ogawa. 2010. Conditional weighted pushdown systems and applications. In *ACM Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM)*. 141–150.

[22] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. 2021. Catala: a programming language for the law. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

[23] Yasuhiko Minamide and Shunsuke Mori. 2012. Reachability Analysis of the HTML5 Parser Specification and Its Application to Compatibility Testing. In *Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 7436)*. Springer, 293–307.

[24] David Pager. 1977. A Practical General Method for Constructing $LR(k)$ Parsers. *Acta Informatica* 7 (1977), 249–268.

[25] Robert Paige and Robert E. Tarjan. 1987. Three partition refinement algorithms. *SIAM J. Comput.* 16, 6 (Dec. 1987), 973–989.

[26] François Pottier. 2009. Lazy Least Fixed Points in ML. (Dec. 2009). Unpublished.

[27] François Pottier. 2016. Reachability and error diagnosis in LR(1) parsers. In *Compiler Construction (CC)*. 88–98.

[28] François Pottier and Yann Régis-Gianas. 2005–2021. The Menhir parser generator. https://gitlab.inria.fr/fpottier/menhir/.

[29] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1-2 (2005), 206–263.

[30] Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. 2006. Efficient Algorithms for Alternating Pushdown Systems with an Application to the Computation of Certificate Chains. In *Automated Technology for Verification and Analysis (ATVA) (Lecture Notes in Computer Science, Vol. 4218)*. Springer, 141–153.

[31] Sean Talts. 2019. stanc3: rewriting the Stan compiler. https://statmodeling.stat.columbia.edu/2019/03/13/stanc3-rewriting-the-stan-compiler/.

[32] Wikipedia. 2021. Hash-consing. https://en.wikipedia.org/wiki/Hash_consing.

[33] Wikipedia. 2021. k-way merge algorithm. https://en.wikipedia.org/wiki/K-way_merge_algorithm.

[34] Wikipedia. 2021. Matrix chain multiplication. https://en.wikipedia.org/wiki/Matrix_chain_multiplication.