Putting Newton into Practice: A Solver for Polynomial Equations over Semirings[⋆]

Maximilian Schlund[1], Michał Terepeta[2], and Michael Luttenberger[1]

[1] Technische Universität München, {schlund,luttenbe}@model.in.tum.de
[2] Technical University of Denmark, mtte@dtu.dk

Abstract. We present the first implementation of Newton's method for solving systems of equations over ω-continuous semirings (based on [5,11]). For instance, such equation systems arise naturally in the analysis of interprocedural programs or the provenance computation for Datalog. Our implementation provides an attractive alternative for computing their exact least solution in some cases where the ascending chain condition is not met and hence, standard fixed-point iteration needs to be combined with some over-approximation (e.g., widening techniques) to terminate. We present a generic C++ library along with the main algorithms and analyze their complexity. Furthermore, we describe our implementation of the counting semiring based on semilinear sets. Finally, we discuss motivating examples as well as performance benchmarks.

1 Introduction

Given a system composed of several components (e.g. the procedures of a recursive program), the interaction of the components can be naturally described by a system of equations where for every component we have a variable X_i and an equation $X_i = F_i(X)$ which is formulated over some algebraic structure. The behavior of the complete system, or some particular aspect of it, can then be obtained as a solution of this system of equations. Especially the problem of finding the least or the greatest solution arises often in applications like program analysis, formal languages, or database theory [5,7,11]

When the algebraic structure exhibits a complete partial order (with least element 0), and F is continuous, then fixed-point iteration yields a monotonically increasing sequence (ω-chain) $0, F(0), F(F(0)), \dots$ which converges to the least solution. However, in order to guarantee termination in general, one either needs to require that *every* ω-chain is eventually constant (ascending-chain condition) or resort to over-approximation e.g. by using a widening operator.

Recently, Newton's method – the standard method to approximate the roots of nonlinear functions over the reals – was generalized to systems of equations over so called ω-continuous semirings (see e.g. [5]). In this particular setting it was shown that (1) Newton's method starting from 0 always converges to the

[⋆] This work was partially funded by the DFG project "Polynomial Systems on Semirings: Foundations, Algorithms, Applications" and MT-LAB (http://www.mt-lab.dk/).

least solution (in contrast to the reals where it is usually non-trivial to find a suitable initial approximation), (2) it converges at least as fast as the standard fixed-point iteration, and (3) it converges within a finite number of iterations for several interesting instances of ω-continuous semirings, e.g., commutative and idempotent semirings, for which fixed-point iteration does not reach the fixed-point in a finite number of steps. Thus, Newton's method allows to compute precise solutions of equation systems over many domains where the standard fixed-point iteration does not terminate.

Contributions, Features We present here the first implementation of Newton's method for ω-continuous semirings; it is freely available from `https://github.com/mschlund/newton`. Our library is implemented in C++ and leverages templates to offer a very flexible interface to instantiate Newton's method for a concrete semiring. To this end, all algorithms and data structures (e.g. the generic Newton solver, polynomials, matrices) are parametrized, for instance by the semiring (in case of polynomials) or the method to solve linear equations (for the generic Newton solver). Hence, the library can be easily extended (without changing the main algorithms) by user-defined semirings, linear solvers, etc.—of course, it also features a set of predefined ones and some generic constructions like product and matrix semirings to build complex semirings from simpler ones. To handle systems efficiently that are very large but sparse, our implementation offers the option to preprocess systems by decomposing them into strongly connected components (cf. [6,5]). The library can be accessed by its API, but also includes a stand-alone solver together with a parser for equations over a number of predefined semirings (e.g. the counting semiring, non-negative reals, commutative regular expressions).

2 Preliminaries

We briefly recall some facts on *semirings*, for details see e.g. [3]. A *semiring* $\langle S, +, \cdot, 0, 1\rangle$ consists of a commutative, additively written monoid $\langle S, +, 0\rangle$ and a (not necessarily commutative) monoid $\langle S, \cdot, 1\rangle$ written multiplicatively where multiplication distributes over addition from both sides, and for all $a \in S$ we have $0 \cdot a = a \cdot 0 = 0$. The semiring is *commutative* resp. *idempotent* if multiplication is commutative (i.e. $a \cdot b = b \cdot a$) resp. addition is idempotent (i.e. $a + a = a$). In the following we will only consider *ω-continuous* semirings: these come equipped with a complete partial order $\sqsubseteq$ with 0 the least element, and both multiplication and addition are continuous in both arguments. Further, the sum of any countable sequence is well-defined and behaves as absolutely convergent series do over the reals. In particular, the *Kleene star* is defined by $a^* := \sum_{i \in \mathbb{N}} a^i$.

An *algebraic system* $X = F(X)$ over a semiring $\langle S, +, \cdot, 0, 1\rangle$ is a system of equations where the right-hand sides F_i are *polynomials*, i.e. finite terms constructed from $+$, $\cdot$, the semiring elements and the variables. Let n denote the number of variables occurring in a given algebraic system. Then F induces a continuous map over S^n, and the least solution of $X = F(X)$ is the least fixed-point μF of this map which is the limit of the sequence obtained by standard fixed-point iteration. As shown in [5], μF is also the limit of the sequence $\nu^{(k)}$ defined by

$$\nu^{(k+1)} = \nu^{(k)} + \Delta^{(k)} \text{ with } \Delta^{(k)} := J_F|^{*}_{\nu^{(k)}} \cdot \delta^{(k)} \text{ and } \nu^{(0)} := 0 \tag{1}$$

where J_F denotes the Jacobian of F (suitably generalized to the setting of semirings) and $\delta^{(k)}$ denotes *any* element satisfying $\nu^{(k)} + \delta^{(k)} = F(\nu^{(k)})$. This iteration scheme is the generalization [3] of *Newton's method* to algebraic systems over ω-continuous semirings, and it usually converges much faster to μF then the standard fixed-point iteration. In the next section, we present the implementation of this definition, i.e. how to compute $\delta^{(k)}$ and $\Delta^{(k)}$.

3 Algorithms and Data Structures

Once the semiring is fixed the central computational problems for implementing Newton's method (Eq. 1) are (1) the computation of $\delta^{(k)}$, (2) the efficient computation of the Kleene star of the Jacobian $J_F|_{\nu^{(k)}}$ based on the Kleene star provided by the underlying semiring, and (3) the efficient representation of the semiring and its elements. We will discuss (1) and (2) in general in the following. As (3) depends on the actual semiring, we will discuss these topics for the special case of the counting semiring; we deem this semiring particularly interesting as Newton's method reaches μF within a finite number of steps.

3.1 Computing $\delta^{(k)}$

Recently, it was shown that $\delta^{(k)}$ is computable for general (also non-commutative) semirings since it corresponds to one part of an unfolding of the equation system [11]. In the special case of idempotent semirings, one can set $\delta^{(k)} := F(0)$ in every iteration (and even simplify the whole definition to $\nu^{(k+1)} = J_F|^{*}_{\nu^{(k)}} F(\nu^{(0)})$) as shown in [4]. If the semiring is commutative we can collect common terms and express the j-th component of $\delta^{(k)}$ succinctly using higher-order derivatives:

$$\delta_j^{(k)} = \sum_{\|i\|_1 \geq 2} \frac{1}{i!} \left(\frac{\partial}{\partial X^i} F_j \right) \Big|_{\nu^{(k-1)}} \cdot X^i \big|_{\Delta^{(k-1)}}.$$

Note that $i \in \mathbb{N}^n$ is a multi-index, so we sum over all derivatives of at least second order evaluated at the previous Newton approximation. The crucial point when implementing this equation is to avoid generating unnecessary multi-indices i (those for which the derivative will be zero anyways) like a naive implementation which generates $(\deg(F_j))^n$ many vectors. Note that the derivative is a linear operator, so we only need to focus on the case where $F_j = aX_1^{d_1} \cdots X_n^{d_n}$ is a monomial of degree $D = \sum_k d_k$. Any element from the set $\{(i_1, \dots, i_n) \in \mathbb{N}^n : \forall_k i_k \leq d_k \wedge 2 \leq \sum_k i_k \leq D\}$ constitutes a valid multi-index. This set contains less than $\prod_k (d_k + 1) \leq \left(1 + \frac{D}{n}\right)^n \leq e^D$ elements and can be enumerated without repetition leading to an implementation in $\leq |\mathcal{M}_j| \cdot e^D$ many steps where $\mathcal{M}_j$ is the set of monomials of F_j.

[3] Over $\mathbb{R}_{\geq 0}$ it coincides with the standard definition of Newton's method.

3.2 Solving Linear Equation Systems

We have implemented two main variants of the Kleene star computation: one is the well-known Floyd-Warshall algorithm [2] and another one is a recursive divide-and-conquer algorithm [10,1]. This algorithm can be seen as an implementation of a star identity from [3]. We take a subdivision of our input matrix $\mathbf{M}$, and compute $\mathbf{M}^*$ recursively:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \qquad \mathbf{M}^* = \begin{bmatrix} \mathbf{F} & \boldsymbol{\alpha}\mathbf{G}^* \\ \mathbf{G}^*\boldsymbol{\beta} & \mathbf{G}^* \end{bmatrix} \qquad \text{with} \quad \begin{aligned} \boldsymbol{\alpha} &= \mathbf{A}^*\mathbf{B} \\ \boldsymbol{\beta} &= \mathbf{C}\mathbf{A}^* \\ \mathbf{G} &= \mathbf{D} + \mathbf{C}\alpha \\ \mathbf{F} &= \boldsymbol{\alpha}\mathbf{G}^*\boldsymbol{\beta} + \mathbf{A}^* \end{aligned} \quad .$$

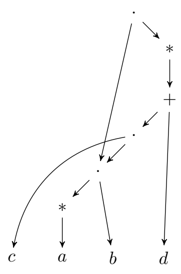

Fig. 1: Succinctly representing $a^*b(ca^*b+d)^*$by sharing subexpressions.

Both algorithms need $\Theta(n^3)$ semiring operations (which is optimal for general semirings if only $+$ and $\cdot$ are allowed [8]), but create slightly different semiring expressions during computation and thus the optimal choice between them depends on the semiring in question.

We also included the option to solve the system only once symbolically and then in each iteration substitute the values $\nu^{(k-1)}$ into this symbolic solution. Symbolic solving can be understood as interpreting the linear system over the free semiring and computing the Kleene star there. Of course, this does not change the asymptotic complexity of the procedure, but allows us to detect common subexpressions (see Fig. 1 for an illustration) and thus greatly reduces the number of semiring operations required to compute the solution. Sharing can reduce this number by 70–90% which is significant for semirings where each operation is expensive, e.g., for the counting semiring presented in Sec. 3.3.

3.3 Implementation of the Counting Semiring

The counting semiring $\mathcal{C} = \langle 2^{|\Sigma|}, \cup, \cdot, \emptyset, \{\mathbf{0}\}\rangle$, consisting of the Parikh images of the formal languages over Σ, is a prime example of an ω-continuous semiring which admits infinite ascending chains. It is known that Newton's method reaches μF on this semiring in at most n steps and that all $\nu^{(i)}$ are rational [4]. Thus, it suffices to give *effective* definitions of the operations on the rational subsets $\mathcal{C}^{\text{rat}}$. Our implementation follows these definitions closely.

Operations A subset $L \subseteq \mathbb{N}^k$ is called *linear*, if $L = \boldsymbol{v_0} + \mathbb{N}\boldsymbol{v_1} + \cdots + \mathbb{N}\boldsymbol{v_n}$ for $\boldsymbol{v_i} \in \mathbb{N}^k$. A set S is called *semilinear* if it is a finite union of linear sets (i.e. a finite sets of linear sets in our implementation). Let us denote the semilinear sets of $\mathbb{N}^k$ by $\mathcal{S}$. We represent linear sets L as pairs $(\boldsymbol{v_0}, G)$ with $\boldsymbol{v_0}$ the *offset* and $G := \{\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}\}$ the *generators* of L.

$\mathcal{C}^{\text{rat}}$ is the (commutative, idempotent) semiring $\mathcal{C}^{\text{rat}} := \langle \mathcal{S}, \cup, \cdot, \emptyset, \{\mathbf{0}\}\rangle$. Multiplication is defined by $S_1 \cdot S_2 := \{L_1 \cdot L_2 \mid L_1 \in S_1,\ L_2 \in S_2\}$ where $(v, G) \cdot (w, H) := (v + w, G \cup H)$ for two linear sets. The Kleene star over $\mathcal{C}^{\text{rat}}$ can be computed inductively by: $S^* :=$ **if** $S = \emptyset$ **then** $\{\mathbf{0}\}$ **else** $L^* \cdot (S \setminus \{L\})^*$ (**where** $L \in S$) having the star of a linear set $L = (\boldsymbol{v}, G)$ defined by $(\boldsymbol{v}, G)^* = \{\mathbf{0}\} \cup (\boldsymbol{v}, \{\boldsymbol{v}\} \cup G)$. It should be clear that the space complexity of the Kleene star for $\mathcal{C}^{\text{rat}}$ is exponential. All these definitions can be regarded as implementations of well-known identities that hold over any commutative, idempotent semiring (cf. [3])

Optimizations Due to the complexity of $\cdot$ and $(-)^*$ a practical implementation of semilinear sets is challenging and usually requires exponential space (e.g. in the number of Newton steps). Since explicit representations of the Parikh image of a CFG can be exponential in the size of the grammar, some exponential blowup is essentially unavoidable (cf. [9] for a detailed analysis). However, the representation of the Newton approximations exhibits a lot of redundancy, e.g., often linear sets subsume each other and generators can be linearly combined (with coefficients in $\mathbb{N}$) from others. Therefore, we implemented several optimizations: We use extensive sharing and store only one copy of each vector and linear set in memory. Furthermore, we try to determine whether a generator can be combined from other generators, and similarly try to simplify the linear sets. Despite the fact that the latter two "simplification" steps require to solve an NP-complete problem (essentially subset-sum [2]), our implementation based on memoization performs very well since the vectors usually contain small numbers.

These simplifications are necessary to get concise solutions for most equation systems and their impact is illustrated in Table 1 in Sec. 4.

Approximations Finally, we have developed two approaches to over-approximate semilinear sets. These significantly improve the performance of the semilinear sets and still yield valuable information in many cases. For a simple example, both preserve finiteness and emptiness.

One of the ideas is to to "collapse" a semilinear set into a pair two sets — one of offsets and the other one of generators. We call this structure a *multilinear set*. The intuition behind it is that we can choose any of the offsets and then use the generators as in the case of linear sets. This approximation is precise if the generator sets attached at different offsets are the same. Otherwise this approximation still keeps "asymptotic upper/lower" bounds on the relationship of different components (i.e. when the offsets are negligible). Consider a semilinear set consisting of two linear ones: $(v_1, \{v_2\})$ and $(v_1', \{v_2'\})$, the corresponding abstraction would be $(\{v_1, v_1'\}, \{v_2, v_2'\})$. Clearly (unless $v_2 = v_2'$) we add some "spurious" points by additionally admitting, e.g., $v_1 + \mathbb{N}v_2'$.

Another idea is to divide every generator $\boldsymbol{v}$ by the greatest common divisor of its elements to obtain a (shorter) vector $\tilde{\boldsymbol{v}}$. For a generator $\boldsymbol{v}$ the set $\mathbb{N}\boldsymbol{v} \subseteq \mathbb{N}^k$ describes a one dimensional discrete "line with gaps". Our approximation corresponds to filling these gaps with more integer points but does not change the direction of the generators, i.e. $\mathbb{N}\boldsymbol{v} \subseteq \mathbb{N}\tilde{\boldsymbol{v}} \subseteq \mathbb{Q}\boldsymbol{v} \cap \mathbb{N}^k$.

4 Experiments

One of the potential applications for counting analyses is to analyze the use of certain resources in a program. For instance, a reentrant lock should be released the same number of times that it has been acquired. Below is a simple example of a recursive program that will obey these rules.

```
proc AcquireRelease
  Lock!();
  if
  :: true => AcquireRelease()
  :: true => skip
  fi;
  Release()
end
```

```
proc Release
  if
  :: true => Unlock!()
  :: true => Release()
  fi
end

proc main
  AcquireRelease()
end
```

However, it is using the stack to ensure that it acquires and releases the lock the same number of times. Even though the stack is unbounded, our solver can verify that — the result of counting the `Lock` and `Unlock` actions is: $\{(\langle 1,1\rangle, \{\langle 1,1\rangle\})\}$. In other words, the behavior is characterized by a linear set with offset $\langle 1,1\rangle$ (there is at least one `Lock` and one `Unlock` action) and generator $\langle 1,1\rangle$ (the number of those actions can be arbitrarily large, but equal in number).

Next we show the behavior of our implementation on two sets of examples over different semirings. We compiled the tool using gcc 4.7 with optimizations (-O2) and ran it on a machine with an Intel 2.7 GHz CPU and 8 GB RAM.

For the first benchmark we computed the Parikh images of all 1,932 grammars provided with the tool cfg-analyzer from `http://www2.tcs.ifi.lmu.de/~mlange/cfganalyzer/`. We simply interpret the grammars as equation systems over the counting semiring and solve them. The grammars are quite simple—at most three terminal, and less than ten nonterminal symbols. We used a timeout

		$> 15s$ (timeout)	$(0.01s, 5s)$	$\leq 0.01s$
Exact	sl-sets, w/o simp	55	35	1842
	sl-sets, simp	0	30	1902
Approx.	ml-sets, w/o simp	2	0	1930
	ml-sets, simp	0	0	1932

Table 1: Parikh image computation for the cfg-analyzer benchmarks; Number of instances solved in the respective times for semi- resp. multilinear sets with and without the optimizations from Sec.3.3.

of 15 seconds, but for most examples computation took only a few milliseconds (see Tab. 1). In all but the timeout-cases, memory usage was negligible (less than 1MB). Since the Parikh image can be viewed as an overapproximation, this could be used as an (incomplete) method to check for non-equivalence of context-free grammars in some cases.

The second benchmark studies a problem that is important in natural language processing and the study of branching processes. The task is to compute the extinction probabilities for stochastic context-free grammars, i.e. the probability for each non-terminal to derive the empty word [6]. To solve this problem we just have to change the semiring in our implementation. This setting also

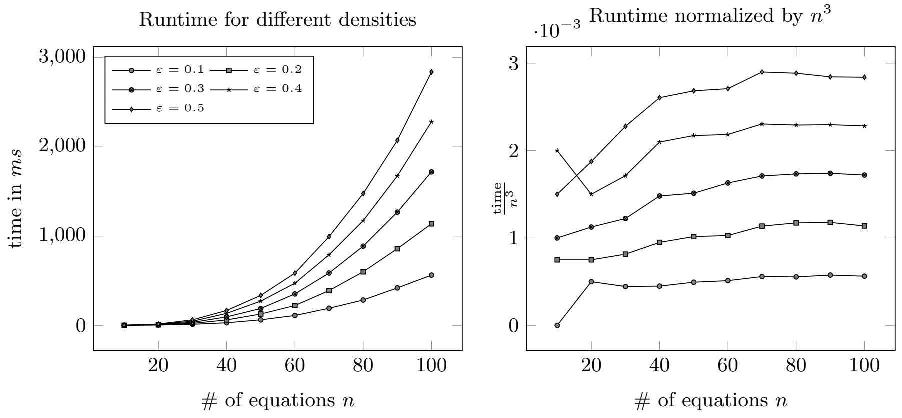

Fig. 2: Approximating the solution (doing 10 Newton steps) of n quadratic equations over $\mathbb{R}_{\geq 0}$ with $\varepsilon\binom{n}{2}$ monomials in each equation. Left: Average solving time (taken over 5 runs) in milliseconds. Right: Numbers from the left divided by n^3.

allows us to demonstrate the scalability of our generic algorithms and to show that our implementation faithfully implements all algorithms with a running time that matches the theoretical analysis. To this end, we randomly generated quadratic equations over $[0, 1]$ and record the running time needed to solve the equations. [4] As we are only interested how the performance varies with the size of the system we fixed the number of Newton iterations to 10. Each equation has $\varepsilon\binom{n}{2}$ monomials and we vary the "density" ε from 0.1 to 0.5—note that these systems are rather dense and large (e.g. the textual description of the system with 100 variables and density 0.5 needs 7.6 MB). For these systems we expect a cubic runtime which is well supported by the data (cf. Fig. 2).

5 Conclusions and Future Work

In this paper we have presented the first implementation of the Newton's method generalized to ω-continuous semirings [5]. We have briefly described the main

[4] These benchmarks are available at `https://github.com/mschlund/newton/tree/master/c/test/grammars/float-random`.

algorithms behind our library as well as the implementation of the counting semiring based on semilinear sets. One of our goals was to make the library generic and flexible—new semirings can be defined and used without changing the main algorithms. Furthermore, we have implemented and discussed various optimizations such as common subexpression elimination during Kleene star computation or simplification of semilinear sets. We have provided motivating applications and discussed initial benchmarks of our library.

Concerning future work, computing the Kleene star for matrices is a problem well suited for parallelization [1] and a generic parallel implementation for general semirings would be useful but does not exist yet to the best of our knowledge. Furthermore, there are well-known symbolic representations of semilinear sets described in the literature, e.g., NDDs or Presburger formulae which we plan to integrate into our library. The main challenge there is to compute the Kleene star efficiently which has not yet been addressed for these representations. Finally, we plan on using our library to solve more involved program analysis problems like pointer may-alias analysis.

Acknowledgments We would like to thank Michael Kerscher for his help with the implementation and Javier Esparza for helpful comments and suggestions.

References

1. Buluç, A., Gilbert, J.R., Budak, C.: Solving Path Problems on the GPU. Parallel Comput. 36(5-6), 241–253 (2010)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)
3. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer (2009)
4. Esparza, J., Kiefer, S., Luttenberger, M.: On Fixed Point Equations over Commutative Semirings. In: STACS. pp. 296–307 (2007)
5. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian Program Analysis. J. ACM 57(6), 33 (2010)
6. Etessami, K., Yannakakis, M.: Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations. J. ACM 56(1) (2009)
7. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS. pp. 31–40 (2007)
8. Kerr, L.R.: The Effect of Algebraic Structure on the Computational Complexity of Matrix Multiplication. Ph.D. thesis, Cornell University, Ithaca, NY, USA (1970)
9. Kopczynski, E., To, A.: Parikh Images of Grammars: Complexity and Applications. In: LICS 2010. pp. 80–89 (2010)
10. Kot, L., Kozen, D.: Kleene Algebra and Bytecode Verification. Electr. Notes Theor. Comput. Sci. 141(1), 221–236 (2005)
11. Luttenberger, M., Schlund, M.: Convergence of Newton's Method over Commutative Semirings. In: LATA. LNCS, vol. 7810, pp. 407–418 (2013)