JFP **31**, e10, 26 pages, 2021.
doi:10.1017/S0956796821000083

Explainable dynamic programming

MARTIN ERWIG AND PRASHANT KUMAR

School of EECS, Kelley Engineering Center 3047, Oregon State University, Corvallis, Oregon, 97331, USA
(*e-mails:* erwig@oregonstate.edu, kumarpra@oregonstate.edu)

Abstract

In this paper, we present a method for explaining the results produced by dynamic programming (DP) algorithms. Our approach is based on retaining a granular representation of values that are aggregated during program execution. The explanations that are created from the granular representations can answer questions of why one result was obtained instead of another and therefore can increase the confidence in the correctness of program results.

Our focus on dynamic programming is motivated by the fact that dynamic programming offers a systematic approach to implementing a large class of optimization algorithms which produce decisions based on aggregated value comparisons. It is those decisions that the granular representation can help explain. Moreover, the fact that dynamic programming can be formalized using semirings supports the creation of a Haskell library for dynamic programming that has two important features. First, it allows programmers to specify programs by recurrence relationships from which efficient implementations are derived automatically. Second, the dynamic programs can be formulated generically (as type classes), which supports the smooth transition from programs that only produce result to programs that can run with granular representation and also produce explanations. Finally, we also demonstrate how to anticipate user questions about program results and how to produce corresponding explanations automatically in advance.

1 Introduction

Which properties should a program have? The answers given by most computer scientist would be *correctness* and *efficiency*. Many functional programmers may want to add *elegance*, *generality*, and maybe *succinctness*. As a software engineer you might want to point out *maintainability*. Ackley (2013) and more recently Vardi (2020) added *robustness* and *resilience*. In this paper, we argue that *explainability* is another desirable property of programs. Especially, the dynamic behavior of a program might be surprising and in need of an explanation even if its static description looks clear and correct.

A program can be in need of explanations in different situations. For example, when a program is to be edited, any changes should be made based on a good understanding of how the existing code works. Here we have explainability supporting maintainability. Whenever programs are used in a teaching context, they need to be explained as well. Finally, whenever the result of a program execution is surprising, an explanation of why the result is correct is required to ensure confidence in the correctness of the program. More generally, if it is unclear whether and how a program works, a user may not feel confident in using the results produced by the program.

In situations when a program behaves in unexpected ways, the mismatch between program behavior and user expectation may be due to a bug in a program or due to wrong

expectations on the part of the user. To find out what is the case, a programmer may employ a debugger to gain an understanding of the program's behavior (Murphy *et al.*, 2006; Roehm *et al.*, 2012). However, by focusing on fault localization debuggers are often not effective tools for gaining program understanding, since they force the user to think in terms of implementation details. In fact, as has been observed by Parnin & Orso (2011), debuggers typically already assume an understanding of the program by the programmer. In addition, it is well known that debugging is very costly and time-consuming (Vessey, 1986). The work on customizable debugging operations is additional testimony to the limitations of generic debugging approaches, see Marceau *et al.* (2007) and Khoo *et al.* (2013). Finally, debugging is not an option for most users of software, simply because they are not programmers. Therefore, debuggers are not the right tool for explaining program behavior.

Our approach to explaining program behavior is based on the idea that a user's confidence and belief in the correctness of a specific result can be supported by providing reasons for why it is to be preferred over potential alternative results. To this end, we track data that is aggregated during a computation and keep the unaggregated representation alongside so that it can be queried later to explain the effects of the performed computation. Specifically, we employ so-called *value decompositions* to maintain unaggregated representations of data that are the basis for decisions in computations that might require explanations. We illustrate this strategy in Section 2 through a simple example before we introduce the concept formally in Section 3.

To make our explanation approach work effectively for a large class of optimization algorithms, we illustrate in Section 4 how dynamic programming (DP) algorithms can be expressed as recurrence equations over semirings, and we present a Haskell implementation to demonstrate that the idea is feasible in practice. Our approach builds on previous work by Goodman (1999) and Rush (2009). In Section 5, we demonstrate how to use this implementation to operate with value decompositions and produce explanations. In Section 6, we present two additional examples to illustrate how to apply our approach and the library in other situations.

Value decompositions produce explanations for decisions. Specifically, they are used to answer questions such as "Why was A chosen over alternative B?". Such explanations are an instance of so-called *contrastive explanations* in the philosophy literature (Lipton, 1990, 2004), which specifically compare two phenomena and justify "Why this rather than that?" (Garfinkel, 1981). Alternatives against which decisions are to be explained are typically provided by users, but as we demonstrate in Section 7, sometimes they can be anticipated, which means that comparative explanations can be generated automatically. Finally, we compare our approach with related work in Section 8 and present some conclusions in Section 9. The main contributions of this paper are as follows.

- A framework based on semirings for expressing *dynamic programming algorithms* that supports *computation with value decompositions*.
- An extension of this framework to enable the *automatic generation of explanations*.
- A method for the *automatic creation of counterexamples* and *preemtive explanations*.
- An implementation of the approach as a *Haskell library*.

```
type Profile = [Int]
type Distance = [Int]
type Score = Int

dist :: Profile -> Profile -> Distance
dist p = map abs . zipWith (-) p

score :: Profile -> Profile -> Score
score p = sum . dist p

matches :: Profile -> [Profile] -> [(Score,Profile)]
matches p ps = sort [(score p q,q) | q <- ps]

best :: Profile -> [Profile] -> (Score,Profile)
best p = head . matches p

buddies :: [Profile]
buddies = [alice,bob,carol]
          where alice = [2,2,7]
                bob   = [3,1,2]
                carol = [3,4,2]
```

Fig. 1. Finding closest matches by aggregating profile distances.

2 Motivating example

In this section, we illustrate the basic idea of generating explanations by computing with value decompositions. Consider a friend-finding application in which users can express their preferences within specific categories of interest using integers between 0 and 10. Assuming for simplicity that the categories are numbered starting from 0, a user's interest profile can be represented by a list `p` of integers, so that interest in category k is represented by the list element `p!!k`. We can then compute a "proximity score" for a pair of profiles as the sum of the absolute differences between the scores for each category. For a given query profile `p`, the best match from a pool of profiles `ps` is the profile with the smallest proximity score to `p`. An implementation of this idea is shown in Figure 1.[1]

Suppose now that we try to find the best match for `dan = [2,2,3]` among the profiles `buddies`.

```
> best dan buddies
[3,1,2]
```

The program correctly computes Bob as the closest match. However, Dan might wonder why it isn't Alice, since she is a perfect match for the first two categories whereas Bob has a nonzero distance to Dan in all categories. The obvious answer is that the proximity score for Bob is less than that for Alice, which we can easily check in this case by computing all matches.

```
> matches dan buddies
[(3,[3,1,2]),(4,[2,2,7]),(4,[3,4,2])]
```

[1] Formally, a profile list of length n represents a point in an n-dimensional space, and the function `score` computes the Manhattan distance between points in that space.

```
matches :: Profile -> [Profile] -> [(Distance,Profile)]
matches p ps = sortBy (compare `on` sum.fst) [(dist p q,q) | q <- ps]

best :: Profile -> [Profile] -> (Distance,Profile)
best p = head . matches p
```

Fig. 2. Finding closest matches while keeping profile distances.

While this explanation is technically correct, it is also not satisfying, since it doesn't directly address the question. In this case, we would like to say that the large distance in the third category countervails the close proximity for the first two categories. In this small example, the counteracting effect of the third category is easy enough to identify, but in applications with dozens of categories the reasons can be more intricate, and they can become more difficult to pinpoint. Moreover, the effects generally have also a more complicated structure. For example, an unexpected counterbalance might be caused not by a single category but rather by a group of categories.

To prepare for providing better explanations, we can extend our program such that it preserves as part of the generated output the contributions of the individual components of the input. The granular output can then later be exploited to generate a more targeted explanation.

A minor change to the function `matches` is sufficient to track the contribution of individual preferences to the computed result. Specifically, instead of aggregating the differences of the profile components with `score`, we use the `dist` function to return the difference between profiles as a vector of differences between individual categories. We then have to adapt the sorting function, so that it compares the sums of the first component of each distance pair, see Figure 2.[2] For completeness, we also repeat the code for the `best` function, since its type has changed.

Computing the matches for Dan with this changed implementation yields the following result:

```
> matches dan buddies
[([1,1,1],[3,1,2]),([0,0,4],[2,2,7]),([1,2,1],[3,4,2])]
```

This output is not meant to be an explanation by itself, nor should it even be presented to users, but it contains enough information to explain why Alice is not the best match for Dan. In this case, it is the large distance contained in the third component. Interestingly, the decomposition of the individual numbers in the decomposed values not only explains why Alice is not the best match for Dan, but also why Dan might consider Alice a best match in the first place. This fact can be exploited to automatically generate seemingly plausible alternatives to be used as counterfactual explanations, a topic we will discuss in Section in Section 7.

To systematically analyze and compare the proximity scores based on individual categories, we can compute their differences, which shows their relative effect on the aggregated values on which decisions, such as finding a closest match, are based. For

[2] The sorting criterion passed to `sortBy` uses the predefined function `on` to apply the binary function `compare` to the values obtained from applying `sum.fst` to the list elements being compared.

example, the difference between Dan's proximity scores for Bob and Alice shows the following:

```
delta :: Distance -> Distance -> Distance
delta = zipWith (-)

> (delta `on` (dist dan)) alice bob
[1,1,-3]
```

The result reveals that while Dan has a greater distance to Bob than to Alice with regard to the two first categories, this is more than made up by his greater proximity in the third category.

If the number of factors gets large, the advantage of one solution over another might not be easy to see. In a practical application, a user's interest profile may consist of dozens of components. To illustrate the potential problems, let's consider the following profiles with just 10 components:

```
alice = [6,2,7,8,9,9,5,4,6,2]
bob   = [2,3,6,7,2,2,4,8,7,4]
dan   = [6,5,6,8,8,1,9,9,5,7]
```

Again, Bob is a better match for Dan than Alice, since the sum of the distance for him is 25, compared with Alice's, which is 28. But why exactly is Bob a better match than Alice? If we compare their distances, we can observe two perfect matches (i.e., the distance is 0) and three close matches (distance 1) for Alice, whereas we can see only one perfect and three close matches for Bob.

```
> dist dan alice
[0,3,1,0,1,8,4,5,1,5]

> dist dan bob
[4,2,0,1,6,1,5,1,2,3]
```

How can we explain that Bob is indeed a better match without resorting simply to the total score? The difference between Bob's and Alice's distances shows the categories in which Dan is closer to Bob as negative values. Correspondingly, the categories closer to Alice are represented by positive numbers.

```
> (delta `on` (dist dan)) alice bob
[4,-1,-1,1,5,-7,1,-4,1,-2]
```

To conclude that Bob is a better match, all we need to show is that the negative numbers outweigh the positive numbers, but, crucially, we don't necessarily need *all* the negative numbers to make that point. It is sufficient to find the smallest subset of negative numbers whose sum (taken as an absolute value) is greater than the sum of all positive numbers. In this example, categories 6, 8, and 10 with a sum of -13 constitute such a set, since the absolute value of $-7 + -4 + -2 = -13$ exceeds the total sum of all positive differences $4 + 1 + 5 + 1 + 1 = 12$. We call a subset, such as $\{-7, -4, -2\}$, a *minimal dominating set*, or *MDS* for short. An MDS can explain decisions based on decomposed values but uses as few as possible data and thus makes explanations easier to process and comprehend.

We could take this idea one step further and ask which of the positive components are required to force the MDS to be as large as it is—providing a kind of *meta-explanation* or *explanation justification*. In our scenario, we could ask, for example, why do we need these three components to justify Bob's closeness, and why aren't just two sufficient? Again, instead of just reporting all categories with a positive difference, we want to find the smallest subset whose sum exceeds the absolute value of any proper MDS subset. We call this set the *minimal forcing set* or *MFS* for short. In our example, the subset $\{-7, -4\}$ with a total of -11 is the one to "beat," which means that no proper subset of the positive numbers will suffice to force the inclusion of -2 in the MDS. In other words, the MFS contains all categories with a positive difference to justify the MDS in this case. Since the examples we employ don't have very many categories, the additional explanatory use provided by the notion of an MFS doesn't apply. We will therefore not pursue this idea any further in this paper.

Obviously, the presented application was quite small and was chosen to easily transform a program so that it can work with value decompositions and produce explanations. In general, however, the situation is not so simple, and to keep the burden of adding value decompositions to programs low, we will integrate our explanation approach into a framework for specifying dynamic programming algorithms based on semirings. To this end, we first formalize value decompositions and minimal dominating sets in Section 3 and then explain the semiring framework in Section 4.

3 Formalizing value decompositions and minimal dominating sets

Many decision and optimization algorithms select one or more alternatives from a set based on data gathered about different aspects for each alternative. In the following, we formalize this view through the concepts of *value decomposition* and *valuation*.

Given a set of categories C, a mapping $v : C \to \mathbb{R}$ is called a *value decomposition* (with respect to C). The (total) *value* $\hat{v}$ of a value decomposition is defined as the sum of its components, that is, $\hat{v} = \sum_{(c,x)\in v} x$.[3] A *valuation* for a set S (with respect to the set of categories C) is a function φ that maps elements of S to corresponding value decompositions, that is, $\varphi : S \to (C \to \mathbb{R})$. We write $\hat{\varphi}(A)$ to denote the total value of A's value decomposition.[4] The elements of S can be ordered based on the valuation totals in the obvious way:

$$\forall A, B \in S.\ A > B :\Leftrightarrow \hat{\varphi}(A) > \hat{\varphi}(B)$$

In the following, we consider an arbitrary but fixed finite set of categories C. When a user asks about a program execution why A was selected over B, the obvious explanation is $\hat{\varphi}(A) > \hat{\varphi}(B)$, reporting the valuation totals. However, such an answer might not be useful, since it ignores the categories that link the raw numbers to the application domain and thus lacks a context for the user to interpret the numbers. If the value decomposition is maintained during the computation, we can generate a more detailed explanation. First, we

[3] This definition can be extended to include different weights for the different components. While this would slightly complicate all the following definitions, it wouldn't change the overall structure of explanations and wouldn't contribute to the insights gained from working with dominating sets. We therefore stick to the simpler definition.

[4] $\hat{\varphi}(A) = \hat{v}$ where $v = \varphi(A)$.

can rewrite $\hat{\varphi}(A) > \hat{\varphi}(B)$ as $\hat{\varphi}(A) - \hat{\varphi}(B) > 0$, which suggests the definition of the *valuation difference* between two elements A and B as follows:

$$\delta(A, B) = \{(c, x - y) \mid (c, x) \in \varphi(A) \wedge (c, y) \in \varphi(B)\}$$

The total of the value difference $\hat{\delta}(A, B)$ is given by the sum of all components, just like the total of a value decomposition.

It is clear that the value difference generally contains positive and negative entries and that for $\delta(A, B) > 0$ to be true the sum of the positive entries must exceed the absolute value of the sum of the negative entries. We call the negative components of a value difference its *barrier*. It is defined as follows:

$$\delta^{-}(A, B) = \{(c, x) \mid (c, x) \in \delta(A, B) \wedge x < 0\}$$

The total value $\hat{\delta^{-}}(A, B)$ is again the sum of all the components. The decision to select A over B needs as support some, but not necessarily all, of the positive components of $\delta(A, B)$, which are called the *dominator components* and which are defined as follows:

$$\delta^{+}(A, B) = \{(c, x) \mid (c, x) \in \delta(A, B) \wedge x > 0\}$$

Any subset of $\delta^{+}(A, B)$ whose total is larger than $|\hat{\delta^{-}}(A, B)|$ will suffice as an explanation. We call such a subset a *dominator*. The set of all dominators is defined as follows:

$$\Delta(A, B) = \{D \mid D \subseteq \delta^{+}(A, B) \wedge \hat{D} > |\hat{\delta^{-}}(A, B)|\}$$

The fewer elements a dominator has, the better it is suited as an explanation, since it requires fewer details to explain how the barrier is overcome. We therefore define the *minimal dominating set* (MDS) as follows:

$$\underline{\Delta}(A, B) = \{D \mid D \in \Delta(A, B) \wedge \forall D' \subset D.D' \notin \Delta(A, B)\}$$

Note that $\underline{\Delta}$ may contain multiple elements, which means that minimal dominators are not unique. In other words, a decision may have different minimally sized explanations.

4 Dynamic programming with semirings

Dynamic programming (DP) is a powerful optimization technique that can be used for the efficient implementation of many problems. The (Bellman–Ford) shortest path algorithm (Ford & Fulkerson, 1956; Bellman, 1958), an algorithm for solving the knapsack problem (Dantzig, 1957; Bellman, 1957b; Bartholdi, 2008), and the value iteration algorithm for solving Markov decision processes (MDPs) (Bellman, 1957a) are all examples of dynamic programming algorithms. One approach to developing a dynamic programming algorithm for a problem is to formulate a recurrence relation, which can then be solved efficiently through the dynamic programming approach.

We first motivate the use of a dynamic programming library in Section 4.1 by illustrating how turning inefficient Haskell function definitions into more efficient dynamic programming algorithms can be tedious. Then we demonstrate how to represent DP algorithms by semirings (Section 4.2) and how such a representation can automatically generate efficient DP implementations from recursive specifications in Haskell (Section 4.3). We illustrate the use of the library with an extended example in Sections 4.4 and 4.5. The semiring basis

```
type Table = [(Integer,Integer)]

fibS :: Integer -> State Table Integer
fibS 0 = return 0
fibS 1 = return 1
fibS n = do t <- get
            case lookup n t of
              Just f -> return f
              Nothing  -> do f1 <- fibS (n-1)
                             f2 <- fibS (n-2)
                             let f = f1+f2
                             modify $ \t -> (n,f):t
                             return f

fib :: Integer -> Integer
fib n = fst (runState (fibS n) [])
```

Fig. 3. Dynamic programming implementation of `fib` using the `State` monad.

for DP does not only provide a uniform, high-level formulation of DP problems, it also allows the systematic integration of explanations. We will discuss this aspect in Section 5.

4.1 Ad hoc dynamic programming

To illustrate the benefits of using a semiring-based DP library for implementing DP algorithms, we consider the computation of Fibonacci numbers as a simple example where a dynamic programming algorithm can lead to a significantly more efficient solution. Here is the well-known recurrence that defines the Fibonacci numbers:

$$\begin{aligned} F_1 &= 0 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n > 2 \end{aligned}$$

The naive Haskell implementation of the Fibonacci function captures the recursive equations quite nicely, but it is very inefficient because it calculates the intermediate Fibonacci values multiple times:

```
fib :: Int -> Int
fib 1 = 0
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

This implementation runs in $O(2^n)$ time. However, one could save intermediate results in a table and reuse them later: When a subproblem arises that has been solved before, its solution is retrieved from the table rather than solving it again (Michie, 1968).

This idea can be directly realized in Haskell by defining a function that uses a table maintained within a state monad, see Figure 3. The helper function `fibS` first tries to find the Fibonacci number of `n` in the table that is the state of the monadic computation.[5] If found, it simply returns that value. Otherwise, `fibS (n-1)` and `fibS (n-2)` are recursively

[5] The runtime can, of course, be improved by representing tables with a balanced binary search trees instead of lists.

$$
\begin{aligned}
a \oplus (b \oplus c) &= (a \oplus b) \oplus c \\
a \oplus b &= b \oplus a \\
a \oplus \mathbf{0} &= \mathbf{0} \oplus a = a \\
a \otimes (b \otimes c) &= (a \otimes b) \otimes c \\
a \otimes \mathbf{1} &= \mathbf{1} \otimes a = a \\
a \otimes (b \oplus c) &= a \otimes b \oplus a \otimes c \\
(a \oplus b) \otimes c &= a \otimes c \oplus b \otimes c \\
a \otimes \mathbf{0} &= \mathbf{0} \otimes a = \mathbf{0}
\end{aligned}
$$

Semiring	Set	$\oplus$	$\otimes$	$\mathbf{0}$	$\mathbf{1}$
Boolean	$\{true, false\}$	$\vee$	$\wedge$	*false*	*true*
Counting	$\mathbb{N}$	$+$	$\times$	0	1
Tropical (Min-Plus)	$\mathbb{R}^+ \cup \{\infty\}$	min	$+$	∞	0
Arctic (Max-Plus)	$\mathbb{R}^+ \cup \{-\infty\}$	max	$+$	$-\infty$	0
Viterbi	$[0, 1]$	max	$\times$	0	1

Fig. 4. Semiring axioms and examples.

computed using, if available, already stored values. Then the sum of both results defines the result of `fibS n`, which is stored in the table before it is returned.

Although conceptually simple, implementing this optimization requires some work on behalf of the programmer and can be quite tedious. This implementation sacrifices the simplicity of the original program structure and its similarity to the recurrence relation. In the following, we describe a mechanism for efficiently implementing dynamic programming algorithms without sacrificing the simplicity that was offered by the naive implementations of their corresponding recurrence relation.

4.2 Semirings and dynamic programming

A semiring is an algebraic structure $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, which consists of a set S with binary operations for addition ($\oplus$) and multiplication ($\otimes$) plus neutral elements zero ($\mathbf{0}$) and one ($\mathbf{1}$) (Golan, 1999). Figure 4 lists the axioms that a semiring structure has to satisfy and several semiring examples.

The semiring framework allows us to convert a naive implementation to an optimized one while maintaining the structural simplicity of the naive implementation. The following equations, called *semiring recurrence representation*, look very similar to the original ones, except for the use of the constants and binary operation from the semiring:

$$
\begin{aligned}
F_1 &= \mathbf{0} \\
F_2 &= \mathbf{1} \\
F_n &= F_{n-1} \oplus F_{n-2} \qquad \text{for } n > 2
\end{aligned}
$$

Along with the semiring recurrence representation, one also needs to know the semiring over which this representation is defined. In the case of Fibonacci, it is the Counting semiring. The semiring recurrence representation and the semiring over which it is defined determine the computation they represent. Describing recurrence relations more abstractly provides a uniform way to talk about recurrence relations, and consequently about the various dynamic programming algorithms whose computation they represent. These ideas originate from Joshua Goodman's work (1999) on semiring parsing and the Semirings library for Haskell by Sasha Rush (2009).

Why are semirings appropriate to model dynamic programming? Mohri (2002) describes the correspondence between the *optimal substructure property* of dynamic programming algorithms and the *monotonicity* property of semirings. The optimal substructure property says that the optimal solution of a dynamic programming problem

contains the optimal solutions of the subproblems into which the original problem was divided. Monotonicity is defined as follows.

Definition 4.1 (Monotonic Semiring). *A semiring* $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *with a partial order* $\leq$ *over* S *is* monotonic *if* $\forall s, t, u \in S. s \leq t \Rightarrow s \oplus u \leq t \oplus u \wedge s \otimes u \leq t \otimes u \wedge u \otimes s \leq u \otimes t$.

A monotonic semiring ensures the optimal substructure property as follows. Suppose the values s and t in Definition 4.1 correspond to the two solutions of a subproblem such that s is a better solution than t (i.e., $s \leq t$). Suppose further that u is the optimal solution of a set of subproblems that does not include the subproblems producing the values s and t. The monotonicity property ensures that s combined with u (and not t combined with u) always results in the optimal solution when the aforementioned subproblem is combined with the set of subproblems.

Since DP computations generally produce correct solutions only for monotonic semirings, we need to ensure that the semiring employed by a DP library is monotonic. Lemma 4.2 is a standard result that delegates the task of checking for monotonicity to the task of checking for idempotence via an induced natural ordering (Mohri, 2002).

Definition 4.2 (Idempotent Semiring). *A semiring* $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *is called an* idempotent semiring *if* $\forall s \in S,\ s \oplus s = s$.

Idempotency gives rise to a partial-order definition, which makes a semiring monotonic.

Lemma 4.1 (Natural Order). *For any idempotent semiring* $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$*, the relation* $\leq$ *defined by* $a \leq b \Leftrightarrow a \oplus b = a$ *is a partial order over* S*, called the* natural order *over* S.

The shown definition of natural order was chosen to fit into the context of Min-Plus semiring; it might be different in other text about semirings.

Lemma 4.2 (Monotonicity). *An idempotent semiring* $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *with the natural order* $\leq$ *over* S *is monotonic.*

Idempotency is typically easier to check than monotonicity. For example, it is obvious that the Min-Plus and Max-Plus semirings are idempotent, since $\min(s, s) = s$ and $\max(s, s) = s$.

4.3 A Haskell library for dynamic programming

We have implemented a library for dynamic programming and semirings that is based on the DP library by Sasha Rush (2009).[6] The first component is a representation of semirings. The semiring structure can be nicely captured in a Haskell type class. Of course, the required laws cannot be expressed in Haskell; it's the programmer's obligation to ensure that the laws hold for their instance definitions.

[6] See `http://hackage.haskell.org/package/DP`. The code has not been maintained in some time and doesn't seem to work currently. Our implementation is available at `https://github.com/prashant007/XDP`.

```
class Semiring a where
  zero, one :: a
  (<+>), (<.>) :: a -> a -> a

ssum :: Semiring a => [a] -> a
ssum = foldr (<+>) zero
```

A `Semiring` type class has already been defined as part of several Haskell packages (some of which are defunct). In general, previous approaches have the advantage that they integrate the `Semiring` class more tightly into the existing Haskell class hierarchy. For example, `zero` and `<+>` are essentially `mempty` and `mappend` of the class `Monoid`. Mainly for presentation reasons, we decided to define the `Semiring` type class independently, since it allows the definition of instances through a single definition instead of being forced to split it into several ones.

The Counting semiring that is needed for implementing the function for computing Fibonacci numbers is obtained by defining a number type as an instance of the `Semiring` class in the obvious way.

```
instance Semiring Integer where
  {zero = 0; one = 1; (<+>) = (+); (<.>) = (*)}
```

The semiring recurrence representation shown in Section 4.2 can be represented in Haskell using the DP library as follows:

```
fibT :: DP Integer Integer
fibT 0 = zero
fibT 1 = one
fibT n = memo (n-1) <+> memo (n-2)

fib :: Integer -> Integer
fib n = runDP fibT n
```

The definition of `fibT` looks very similar to the naive implementation. A noticeable difference is that the recursive function calls are made using `memo` to indicate when intermediate results of recursive calls should be stored in a table. The implementation is similar to the one shown in Figure 3 that is based on the state monad and consists of two parts, (a) the definition of the recurrence relation that denotes a table-based, efficient implementation, and (b) an interface to execute the code.

The definition makes use of the following building blocks provided by the dynamic programming library. (We will see the use of the function `inj` in later examples.)

- The functions `<+>` and `<.>` correspond to semiring addition ($\oplus$) and multiplication ($\otimes$), respectively.
- The type `DP t r` represents a dynamic programming computation. Parameter `t` represents the argument, corresponding to the table index, on which recursion occurs, and `r` is the result type of the computation.
- The function `memo` takes an index as input. The index can be thought of as the input to the smaller subproblems that need to be solved while solving a dynamic programming problem; it is the quantity on which the algorithm is recursively invoked. With

`memo` a subproblem for a given input value is solved only once, and the result is stored in a table for later reuse.

- The function `inj` turns any semiring value (different from **0** and **1**) into a `DP` value.
- The function `runDP` executes a dynamic programming specification `DP t r` that works on tables indexed by a type `t`. The function `runDP` yields a function that computes results of type `r` from an initial value of type `t`.

Note that the Counting semiring is *not* idempotent but still monotonic. Thus, the implementation of Fibonacci numbers is still correct.

4.4 Computing the lengths of shortest paths

Next, we consider the shortest-path problem as a more realistic DP example for a computation that can actually benefit from explanations based on value decompositions. In its simplest form, a shortest-path algorithm takes a graph together with a source and destination node as inputs and computes the length of the shortest path between the two nodes.

In the following, we show how to implement an algorithm for computing shortest paths using the semiring library. Specifically, we employ the *Bellman–Ford* algorithm (Ford & Fulkerson, 1956; Bellman, 1958), which can be concisely described by the following recurrence relation, in which $SP_s(v, i)$ denotes the length of the shortest path with at most i number of edges between the start node s and any other node v. This algorithm works only for graphs with nonnegative edge weights.

$$SP_s(v,i) = \begin{cases} 0 & i = 0 \text{ and } v = s \\ \infty & i = 0 \text{ and } v \neq s \\ \min(SP_s(v,i-1), \min_{(u,v)\in E}(SP_s(u,i-1) + w(u,v))) & \text{otherwise} \end{cases}$$

Here E is the set of edges in the graph, and $w(u, v)$ denotes the weight of edge (u, v). This algorithm incrementally updates connection information between nodes. When all edge labels in a graph with n nodes are positive, the shortest path contains at most $n - 1$ edges. Therefore, the shortest path to a node t can be obtained by the expression $SP_s(t, n)$. The algorithm in each step considers nodes that are one more edge away from the target node and updates the distance of the currently known shortest path.

The computation described by the recurrence relation is captured by the Min-Plus semiring (see Figure 4): The min function of the recurrence is represented by the semiring addition $\oplus$, and the numeric addition is represented by semiring multiplication $\otimes$. The constants **0** and **1** represent the additive and the multiplicative identity and stand for ∞ and 0, respectively. Written in terms of the semiring operations the recurrence relation looks as follows:

$$SP_s(v,i) = \begin{cases} \mathbf{1} & i = 0 \text{ and } v = s \\ \mathbf{0} & i = 0 \text{ and } v \neq s \\ SP_s(v,i-1) \oplus \bigoplus_{(u,v)\in E}(SP_s(u,i-1) \otimes w(u,v)) & \text{otherwise} \end{cases}$$

Note that this formulation of the algorithm is actually more general than the original, since the operations can be instantiated with operations of a different semiring to express different computations. We will later take advantage of this generality by generating, in addition

```
type Node = Int
type Edge = (Node,Node)
type Graph l = [(Edge,l)]

noNodes :: Graph l -> Int
noNodes = length . nub . concatMap (\((p,q),_) -> [p,q])

class Semiring r => SP l r where
  result :: (Edge,l) -> r

  sp :: Graph l -> Node -> DP (Node,Int) r
  sp g s (v,0) = if s==v then one else zero
  sp g s (v,i) = memo (v,i-1) <+>
                 ssum [memo (u,i-1) <.> (inj.result) e | e@((u,v'),_)<-g, v'==v]

  shortestPath :: Graph l -> Node -> Node -> r
  shortestPath g s t = runDP (sp g s) (t,noNodes g-1)
```

Fig. 5. Generic shortest path implementation.

to the shortest path value, decomposed values, the path itself, and explanations. To make use of the semiring framework, we show how the shortest-path algorithm can be expressed as a dynamic programming algorithm in our library.

The Min-Plus semiring is implemented in Haskell through a class instance definition for the type constructor `Large` that adds ∞ to a number type. We need ∞ to represent the case when there isn't a path between two nodes.

```
data Large a = Finite a | Infinity deriving (Eq,Ord)

instance (Num a,Ord a) => Semiring (Large a) where
  {zero = Infinity; one = Finite 0; (<+>) = min; (<.>) = (+)}
```

The instance definitions for `Functor` and `Applicative` are straightforward (they are basically the same as for `Maybe`), and we omit them here for brevity. The instance definition for `Num` is also standard; it uses `liftA2` for binary functions and `fmap` for unary functions. One subtle, but important, difference between `Large` and `Maybe` is that `Infinity` is defined as the second constructor in the data definition, which makes it the largest element of the `Large` data type when an `Ord` instance is automatically derived.

For the Haskell implementation of the algorithm, we represent edges as pairs of nodes and a graph as a list of edges paired with their weights, see Figure 5. We use a multi-parameter type class `SP` to facilitate a generic implementation of the shortest-path function that works for different edge label types (type parameter `l`) and types of results (type parameter `r`). As in the Fibonacci example, the implementation consists of two parts: (a) the recurrence specification of the DP algorithm (the function `sp`) and the function for running the described computation (the function `shortestPath`). Both functions have a default implementation that doesn't change for different class instances. The class consists of an additional member `result` that turns labeled edges into values of the DP result type `r`. The definition of the `sp` function is directly derived from the semiring representation of the Bellman–Ford recurrence relation. Note that the `memo` function in the definition of `sp` takes pairs as input and effectively denotes a recursion of the `sp` function, memoizing the output of each recursive call for later reuse. The second argument of the `<+>` function in

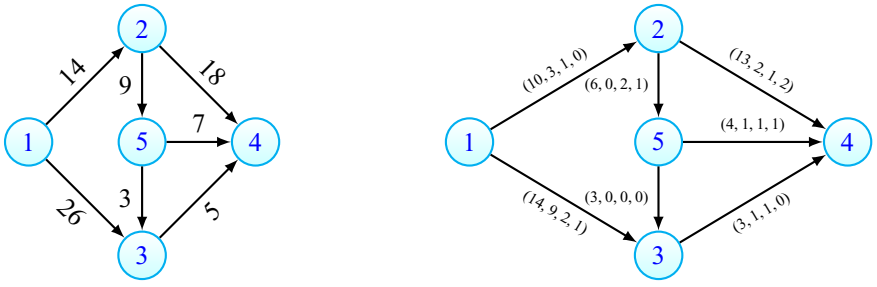

Fig. 6. Left: An edge-labeled graph (`g`). Right: A version of `g` with decomposed edge-labels (`gd`).

the recursive case of the `sp` function implements the part $\bigoplus_{(u,v)\in E}(SP_s(u, i-1) \otimes w(u, v))$ of the recurrence relation. The function `ssum` takes a list of values, namely all incoming edges at node `v`, and combines these using the semiring addition function `<+>`. Finally, the actual computation of a shortest path between two nodes is initiated by the function `shortestPath` through calling `sp` and passing the number of nodes of the graph as an additional parameter (computed by the helper function `noNodes`).

To execute the `shortestPath` function for producing path lengths for graphs with non-decomposed edge labels, we need to create an instance of the `SP` type class with the corresponding type parameters. Since the functions `sp` and `shortestPath` are already defined, we only need to provide a definition of the function `result`.

```
instance SP Double (Large Double) where
  result (_,l) = Finite l
```

The result of running the shortest-path algorithm on the non-decomposed graph shown on the left of Figure 6 produces the following output:

```
> shortestPath g 1 4 :: Large Double
30.0
```

Specifying the result type (`r`) to be `Large Double` selects the implementation in which the `result` function maps a labeled edge to the DP result type as shown. In addition to the length of the shortest path, we may also want to know the path itself. We develop a solution based on semirings next.

4.5 Computing shortest paths

To compute shortest paths in addition to their lengths using our DP library, we need an instance of `Semiring` for the type `(Large Double,[Edge])`. A first attempt could be to define pairs of semirings as semirings. This would require both components to be semirings themselves, but since there is not a straightforward instance of lists as semirings, we have to adopt a different strategy.

If we look at this example more closely, we can observe that the DP computation of a shortest path is solely driven by the first component of the pair type and that the paths are computed alongside. This means that the path type doesn't really need to support the

```
data View a b = View a b

class Selector a where
  first :: a -> a -> Bool

instance Ord a => Selector (Large a) where
  first = (<=)

instance (Selector a,Semiring a,Monoid b) => Semiring (View a b) where
  zero = View zero mempty
  one  = View one mempty
  l@(View x _) <+> r@(View y _) = if first x y then l else r
  l@(View x a) <.> r@(View y b) | l == zero || r == zero = zero
                                | otherwise = View (x <.> y) (mappend a b)
```

Fig. 7. The `View` semiring.

`Semiring` structure. We can exploit this fact by defining a semiring instance for pairs that relies for its semiring semantics only on the semiring instance of its first component. To handle the combination of values in its second component, we require that the type be a monoid and use the binary operation of the monoid in the instance definition for the `<.>` function. The `<+>` function acts as a selector of the two values, and the selection is controlled by a selection function that the first type parameter has to support through a corresponding instance definition for the class `Selector`. The function `first` implements a selection decision between two values; it returns `True` if the first argument is selected and `False` otherwise. The complete code is shown in Figure 7. Note that we can't simply replace `Selector` by `Ord`, since we might also want to be able to use `<` as comparison function for making `Large` (i.e., a Max-Plus semiring) a `View` instance.

The conditional used in the definition of `<+>` ensures that the absorption rule ($a \otimes \mathbf{0} = \mathbf{0} \otimes a = \mathbf{0}$) holds and that `View` is thus a semiring.

With the help of the `View` semiring, we can obtain a DP algorithm that computes the paths alongside the lengths. To this end, we represent a path `p` and its length `l` as a value `View l p` so that the length provides a view on the path on which the DP computation operates.

```
type Path l = View l [Edge]
```

The shortest-path algorithm results from an instance of the `SP` type class for the result type `Path (Large Double)`, which again only requires the definition of the `result` function to map labeled edges to the DP result type.

```
instance SP Double (Path (Large Double)) where
  result (e,l) = View (Finite l) [e]
```

The result of running the shortest-path algorithm on the non-decomposed graph produces the following output.[7] Again, we specify the result type of the DP computation to select the appropriate implementation of `result` and thus `sp`.

```
> shortestPath g 1 4 :: Path (Large Double)
30.0 <~ [(1,2),(2,5),(5,4)]
```

[7] We pretty-print values `View x y` of the `View` data type as `x <~ y` to indicate that the value `y` is viewed as, and represented in computations by, `x`.

This is the correct result, but we don't get an explanation why it is faster than, say, the path `[(1,3),(3,4)]`. In the next section, we develop a version of the shortest-path program that can answer questions like this.

5 Explaining dynamic programming with value decomposition

In our introductory example from Section 2, we have represented decomposed values as lists and then could simply aggregate these lists into values where needed to make decisions (see function `match` in Figure 2). For programs implemented using the DP library, however, this simple strategy does not work in general, since the programs employ the constants `one` and `zero`, which need to be instantiated to lists of different lengths depending on the context. For example, in the Min-Plus semiring setting, we would expect $\mathbf{1}$ to denote $[0, 0]$ in the context of $[1, 2] \otimes \mathbf{1}$, while it should denote $[0, 0, 0]$ in the context of $[1, 2, 3] \otimes \mathbf{1}$. We can achieve this behavior by defining a data type for decomposed values and providing a `Num` instance definition in which by default decomposed values consist of singleton lists but will be padded to match the length of potentially longer arguments. The code for this is not complicated but a bit lengthy and can be found in the supplemental material.

```
newtype Decomposed a = Values {values :: [a]}
```

We also need an `Ord` and `Eq` instance, which are used in the code later. Both instances use the sum of the elements of the lists contained in the `Values` constructors to perform the comparison.

```
instance (Eq a,Num a) => Eq (Decomposed a) where
  (==) = (==) `on` sum.values

instance (Ord a,Num a) => Ord (Decomposed a) where
  (<=) = (<=) `on` sum.values
```

These definitions ensure that decomposed values are treated in comparisons as the sums they represent.

With a `Num` instance the `Decomposed` data type can be used in implementing versions of the shortest-path algorithm that can handle graphs with decomposed edge values. Specifically, we can obtain two more versions of the shortest-path algorithm as an instance of the `SP` type class, one for computing lengths only, and another one for computing paths alongside lengths. The type of the edge labels is `[Double]` to reflect the decomposed edge labels in the input graphs. The result types for the DP computations are either the path lengths represented as decomposed edge labels or the view of paths as decomposed values. Here are the corresponding instance definitions.

```
instance SP [Double] (Large (Decomposed Double)) where
  result (_,l) = Finite (Values l)

instance SP [Double] (Path (Large (Decomposed Double))) where
  result (e,l) = View (Finite (Values l)) [e]
```

Running either of the shortest-path algorithms has to use the graph `gd` with decomposed edge labels (shown on the right of Figure 6) and needs to specify the desired output type.

```
> shortestPath gd 1 4 :: Large (Decomposed Double)
[20.0,4.0,4.0,2.0]

> shortestPath gd 1 4 :: Path (Large (Decomposed Double))
[20.0,4.0,4.0,2.0] <~ [(1,2),(2,5),(5,4)]
```

With value decompositions available, we can now compute valuation differences and minimal dominating sets to compare results with alternative solutions. For example, the decomposed length of the alternative path `[(1,3),(3,4)]` between nodes 1 and 4 is `[17,10,3,1]` (as can be easily verified by adding the edge label components of the graph `gd` in Figure 6). Since `Decomposed` and `Large` are `Num` instances, we can immediately compute the valuation difference with respect to the shortest path to be `[3.0,-6.0,1.0,1.0]`. Before we can implement a function for computing minimal dominating sets, we need two more things: first, we have to extract the decomposed values (of type `Decomposed Double`) from the semiring values (of type `Large (Decomposed Double)`) produced by the shortest-path function. Second, when sorting the components of the valuation difference into positive and negative parts, we need to decide which parts constitute the barrier and which parts are supporting components of the computed optimal value. This decision actually depends on the semiring on which the computation to be explained is based. In the shortest path example, we have used the Min-Plus semiring for which positive value differences constitute barriers and negative values overcome the barrier. In general, a value s is a *supporting value* (for overcoming a barrier) if:

$$s \oplus \mathbf{1} = s$$

We can realize both requirements through a (multi-parameter) type class `Decompose` that relates semiring types with the types of values used in decompositions, see Figure 8.[8]

With this type class, we can directly implement the definition of $\underline{\Delta}$ from Section 3 as a function for computing the smallest sublist of supporting values whose (absolute) sum exceeds the sum of the barrier, in this case it's the singleton list `[-6.0]`. However, the number itself doesn't tell us what category provides this dominating advantage. To assign a meaning to the bare numbers, we can employ a data type `Labeled` that pairs values with strings.

```
data Labeled a = Label String a

unlabel :: Labeled a -> a
unlabel (Label _ x) = x
```

Creating a `Num` instance for `Labeled` allows us to assign labels to the individual numbers of a `Decomposed` value and apply the computation of dominating sets to work with labeled

[8] The additional argument of type `a` for `supportive` is required for technical reasons to keep the types in the multi-parameter type class unambiguous. As can be seen in the instance definition, it is not really used. Unfortunately also, we can't give the generic definition for `supportive` indicated by the equation, since that would as well lead to an ambiguous type.

```
class Semiring a => Decompose a b | a -> b where
  dec :: a -> Decomposed b
  supportive :: a -> b -> Bool

instance Decompose (Large (Decomposed Double)) Double where
  dec Infinity = Values []
  dec (Finite vs) = vs
  supportive _ x = x<0

withCategories :: Decomposed a -> [String] -> Decomposed (Labeled a)
withCategories d cs = Values (map Label cs) <*> d

explainWith :: (Decompose a b,Ord b,Num b) => [String] -> a -> a
-> Decomposed (Labeled b)
explainWith cs d d' = Values $ head $ sortBy (compare `on` length) doms
    where (support,barrier) = partition sup $ values delta
          doms = [d | d <- sublists support, abs (sum d) > abs (sum barrier)]
          delta = dec d `withCategories` cs - dec d' `withCategories` cs
          sup = supportive d . unlabel
```

Fig. 8. Minimal dominators and explanations.

numbers, resulting in the function `explainWith`. If `p` is the result of the shortest-path function shown above and `p'` is the corresponding value for the alternative path considered, we can explain why `p` is better than `p'` by invoking the function.

```
> explainWith ["Distance","Traffic","Weather","Construction"] p p'
[Traffic:-6.0]
```

The result says that, considering traffic alone, `p` has an advantage over `p'`, since the traffic makes the path of `p` faster by `6`.

6 Additional examples

The generation of explanation works in much the same way for other DP algorithms. The general workflow is as follows.

- Identify the appropriate semiring for the optimization problem. This might require the definition of a new Haskell data type and its `Semiring` instance. The `View` semiring offers opportunities to realize a variety of computations that produce results on different levels of detail.
- Implement the DP algorithm as a type class that contains the main recurrence, a wrapper to run the described computation, plus the function `result` that ties the DP computation to different result types.
- Define a value decomposition for the result type. The categorization of the type of values to be optimized allows the function `explainWith` to compare optimal results with alternatives and produce explanations based on value categories.

We briefly illustrate these steps with two more examples. In the following, we only show the programs. Our library additionally contains encoded actual examples.

```
type Index   = Int
type Table v = [(Capacity,v)]

class Semiring r => KS v r where
  result :: (Index,v) -> r

  ks :: Table v -> DP (Int,Capacity) r
  ks t (0,_) = one
  ks t (_,0) = one
  ks t (k,c)
    | ck > c    = memo(k-1,c)
    | otherwise = ((inj.result) (k-1,vk) <.> memo (k-1,c-ck)) <+> memo(k-1,c)
    where (ck,vk) = t !! (k-1)

  knapSack :: Table v -> Capacity -> r
  knapSack t b = runDP (ks t) (length t,b)
```

Fig. 9. Dynamic programming solution to the Knapsack problem.

Solving Knapsack problems

The first example is the DP algorithm for solving knapsack problems, that is, optimization problems for finding a selection of items that maximizes their combined value while not exceeding a certain capacity with their total cost. Assume that C is the total capacity of the knapsack, and let c_i and v_i denote the cost and the value, respectively, of the i^{th} element. If $KS(k, C)$ denotes the value of the highest-valued solution that uses items from the set $\{1, 2, \ldots, k\}$, then KS can be defined through the following recurrence:

$$KS(k, C) = \begin{cases} 0 & \text{if } k = 0 \vee C = 0 \\ KS(k-1, C) & \text{if } c_k > C \\ \max(v_k + KS(k-1, C-c_k), KS(k-1, C)) & \text{otherwise} \end{cases}$$

Since the optimization aims at maximizing the total value of the items placed in knapsacks, we observe in the first step that we need to work with the Max-Plus semiring. Therefore, we need a data type `Small` (dual to `Large` with a constructor for representing negative infinity) together with its straightforward `Eq`, `Ord`, and `Semiring` instances.

In the second step, we can describe the class of knapsack DP computations using a type class `KS` shown in Figure 9, which is quite similar to the `SP` class from in Figure 5.

Finally, we need to model the decomposition of the values for different items to be placed in the knapsack. This step depends, of course, on the application domain in which the optimization is used. For example, in the case of planning investments with limited funds, the value of individual investment options could be decomposed into their current value, their stability, and growth expectation.

The fact that we can define the value decomposition after deciding on the application illustrates nicely how we can specialize the explanations for one DP algorithm to different application domains independently of the implementation of the algorithm itself.

Solving decoding problems in hidden Markov models

Hidden Markov models (HMMs) relate hidden states and observations. HMMs have many applications in reinforcement learning and temporal pattern recognition, including speech,

```
type TTable a c = [((a,a),c)]
type ETable a b c = [((a,b),c)]

class (Ord a,Ord b,Num c,Ord c,Semiring r) => Viterbi a b c r where
  result :: (a,b,Small c) -> r

  viterbi :: (TTable a c,ETable a b c) -> [b] -> DP (Int,a) r
  viterbi (ts,es) xs (i,k)
     | length xs==i = res (k,xs!!0,one)
     | otherwise    = ssum [memo(i+1,j) <.> res (k,xs!!0,Finite' p) <.>
                            (res.lookupE (j,xs!!i) k) es | ((s,j),p) <- ts, s==k]
     where res = inj . result

  viterbiPath :: (TTable a c ,ETable a b c) -> [b] -> a -> r
  viterbiPath tes xs s = runDP (viterbi tes xs) (0,s)

lookupE :: (Ord a,Eq b) => (a,b) -> a -> ETable a b c -> (a,b,Small c)
lookupE x@(p,q) s = maybe (s,q,NegInfinity) (\v->(s,q,Finite' v)) . lookup x
```

Fig. 10. Dynamic programming implementation of the Viterbi recurrence.

handwriting, and gesture recognition, bioinformatics, transportation, etc. One HMM application is the so-called *decoding problem*, which is to infer a sequence of hidden states from a sequence of observations. The *Viterbi algorithm* is a dynamic programming algorithm that can be used for this purpose. An HMM is represented by a table t_{kj} of so-called *transition probabilities* between hidden states k and j. In addition, $e_k(x_i)$ is a table of so-called *emission probabilities* of making observation x_i in state k. The recurrence relation for the Viterbi algorithm for a list of observations $x_1, \ldots, x_n$ is shown below. $V(i,k)$ represents the maximum probability value of observing x_i in state k:

$$V(i,k) = \begin{cases} 1 & i = n \\ \max_{ij}\{V(i+1,j) \times t_{kj} \times e_k(x_i)\} & \text{otherwise} \end{cases}$$

Again, in the first step, we have to identify the proper semiring for performing the computations. With maximization as the selection function and multiplication (of probabilities) as aggregation, the Viterbi semiring shown in Figure 4 seems like the obvious choice. However, while we could decompose probabilities into contributing factors, we cannot compute value differences and minimal dominating sets. Therefore, we apply a log-transformation, that is, map all probabilities to their log-values, which allows us to operate with the Max-Plus semiring. We then have to use the `Small` data type in the implementation, as we did in the knapsack example.

In the second step, we describe the class of Viterbi DP computations using a type class `Viterbi` shown in Figure 10. Apart from the more complicated input (two probability tables), storing more information in the DP result, and some additional auxiliary definitions, the encoding of the recurrence in Haskell is again mostly straightforward, with the one notable exception that we have to log-transform the probabilities from a multiplicative structure into an additive structure so that we can deal with them within the Max-Plus semiring. (This happens when examples are fed into computations and is not shown here; it can be seen in the supplemental material.)

The third step again selects an application domain and defines a value decomposition that can support the explanation of solutions.

7 Proactive generation of explanations

In the previous section, we have demonstrated how value decompositions can help provide succinct explanations for why solutions obtained by dynamic programming algorithms are better than any given alternative. In those scenarios, it is the user who has to supply such alternatives as an argument for the function `explainWith`. It turns out that in many cases such examples can be automatically generated, which means that the questions users might have about computed solutions can be anticipated and preemptively answered.

In the case of finding shortest paths, a result may be surprising—and therefore might prompt the user to question it—if the suggested path is not the shortest one in terms of traveled distance. This is because the travel distance retains a special status among all cost categories in that it is always a determining factor in the solution and can never be ignored. This is different for other categories, such as traffic or weather, which may be 0 and in that case play no role in deciding between different path alternatives.

In general, we can distinguish between those categories that always influence the outcome of the computation and those that only *may* do so. We call the former *principal categories* and the latter *minor categories*. We can try to exploit knowledge about principal and minor categories to anticipate user questions by executing the program with decomposed values but keeping only the values for the principal categories. If the result is different from the one produced when using the complete value decomposition, it is an alternative result worthy of an explanation, and we can compute the minimal dominating set accordingly.

Unfortunately, however, this strategy doesn't work as expected, because if we remove minor categories to compute an alternative solution, the values of those categories aren't aggregated alongside the computation of the alternative and thus are not available for the MDS computation. Alternatively, instead of changing the underlying decomposition data, we can change the way their aggregation controls the DP algorithm. Specifically, instead of ordering decomposed values based on their sum (as in the `Ord` instance definition shown at the beginning of Section 5), we can order them based on a primary category (or a sum of several primary categories). In Haskell, we can achieve this by defining a new data type `Principal`, which is basically identical to `Decomposed` but has a correspondingly altered `Ord` instance definition, see Figure 11. We also need a function that can map `Principal` data into `Decomposed` data within the type of the semiring to get two `Decomposed` values that can be compared and explained by the function `explainWith`. With these preparations, we can define the function `explain` that takes *two* instances of the function to be explained, one for producing `Principal` values and one for producing `Decomposed` values. It applies both functions to the input and if the results differ, that is when the result considering only the principal categories yields a different result, then the difference is explained as before using the function `explainWith`.

To use `explain` in our example, we have to create another instance for the `SP` class for working with `Principal` data. We also have to create an instance for the function `fromPrincipal`, so that we can turn `Principal` data into `Decomposed` data inside the `Large` type.

```
newtype Principal a = PValues {pvalues :: [a]}

class FromPrincipal f where
  fromPrincipal :: f (Principal a) -> f (Decomposed a)

explain :: (FromPrincipal f,Decompose (f (Decomposed a)) b,Eq (f (Decomposed a)),
            Ord b,Num b) =>
           (i -> f (Decomposed a),i -> f (Principal a)) -> [String] -> i ->
           (f (Decomposed a),Maybe (f (Decomposed a),Decomposed (Labeled b)))
explain (f,g) cs i | o==o'     = (o,Nothing)
                   | otherwise = (o,Just (o',explainWith cs o o'))
                   where (o,o') = (f i,fromPrincipal (g i))
```

Fig. 11. Automatic explanations.

```
instance SP [Double] (Large (Principal Double)) where
  result (_,l) = Finite (PValues l)

instance FromPrincipal Large where
  fromPrincipal = fmap asDecomposed where asDecomposed (PValues xs) = Values xs
```

Finally, to be able to apply `explain` we have to normalize the argument type of the shortest-path function into a tuple.

```
type SPInput = (Graph [Double],Node,Node)

spD :: SPInput -> Large (Decomposed Double)
spD (g,v,w) = shortestPath g v w

spP :: SPInput -> Large (Principal Double)
spP (g,v,w) = shortestPath g v w
```

When we apply `explain`, it will in addition to computing the shortest path also automatically find an alternative path and explain why it is not a better alternative.

```
> explain (spD,spP) categories (gd,1,4)
([20.0,4.0,4.0,2.0],Just ([17.0,10.0,3.0,1.0],[Traffic:-6.0]))
```

Of course, the output could be printed more prettily. Moreover, the need to pass two functions as arguments to `explain` seems annoying, but unfortunately Haskell's type class system doesn't let us derive, in general, the second function for computing with `Principal` data from the function computing with `Decomposed` data.

8 Related work

In an earlier work, we proposed the idea of preserving the structure of aggregated data and using it to generate explanations for reinforcement learning algorithms based on' so-called *minimum sufficient explanations* (Erwig *et al.*, 2018; Juozapaitis *et al.*, 2019). That approach is less general than what we describe here and strictly situated in a machine learning context that is tied to the framework of *adaptation-based programming* by Bauer *et al.* (2011). The concepts of value decomposition and minimal dominating sets presented in this paper constitute a general approach to the generation of explanations for a wide

range of algorithms. The concept of minimal sufficient explanations was also used in related work on explanations for optimal MDP policies (Khan *et al.*, 2009). That work is focused on automated planning and on explaining the optimal decision of an optimal policy. Those explanations tend to be significantly larger than explanations for decisions to select between two alternatives. Also, that work is not based on value or reward decompositions. The idea of explaining the choice of one alternative over another is similar to the notion of *contrastive explanations* (Garfinkel, 1981; Lipton, 2004, 1990), which work by comparing a phenomenon to potential alternatives.

The idea behind *provenance semirings* (Green *et al.*, 2007; Cheney *et al.*, 2009) is to describe the origins and history of data over their life cycle. To this end, each tuple in the original database is annotated with a unique identifier. Queries then generate provenance expressions for each tuple in the resulting tables. The algebraic operations involved in this computation form a semiring. We can think of provenance expressions as a kind of trace, and they can be used to answer questions such as "Which input tuples were used to produce an output tuple?" and "How are input tuples used to produce a given output tuple?". This approach is similar to our technique in that additional information (in the form of provenance expressions) is generated by semiring computations which are later used for explanations. However, the two techniques differ in how this additional information is processed and presented as explanations to the end users.

Debugging

Debugging can be regarded as a process to find explanations for incorrect program behaviors. Debugging presumes that a program is incorrect, and its primary purpose is to locate and eliminate a program fault. In contrast, explanations of program behavior generally could (and should) work for correct programs as well. Nevertheless, debugging is a widely used method for understanding program behavior.

Zeller (2002) describes the use of *delta debugging* to reveal the cause–effect chain of program failures, that is, the variables and values that caused the failure. Delta debugging needs two runs of a program, a successful one and an unsuccessful one. It systematically narrows down failure-inducing circumstances until a minimal set remains, that is, if there is a test case which produces a bug, then delta debugging will try to trim the code until the minimal code component which reproduces the bug is found. Delta debugging and the idea of MDSs are similar in the sense that both try to isolate minimal components responsible for a certain output. An important difference is that delta debugging produces program fragments as explanations, whereas an explanation based on value decompositions is a structured representation of program inputs.

The process of debugging is complicated by the low-level representation of data processed by programs. *Declarative debugging* aims to provide a more high-level approach, which abstracts away the evaluation order of the program and focuses on its high-level logical meaning. This style of debugging is discussed by Pope (2005) and is at the heart of, for example, the Haskell debugger Buddha. The so-called *Evaluation Dependence Tree* (EDT) (Nilsson & Sparud, 1997) hides the operational details of programs and is used as the basis for Freja (Nilsson & Fritzson, 1994), another declarative debugging tool for lazy functional languages. Though this debugging strategy is more abstract than most

standard debugging strategies, the generated explanations are still too low level to be used as explanations for program users.

Still another method is *observational debugging*, employed by the Haskell debugger Hood (Gill, 2001), which allows the observation of intermediate values within the computation. The programmer has to annotate expressions of interest inside the source code. When the source code is recompiled and rerun, the values generated for the annotated expressions are recorded. Like value decomposition, observational debugging expects the programmers to identify and annotate parts of the programs that are relevant to generate explanations. A potential problem with the approach is that the number of intermediate values can become large and not all the intermediate values have explanatory significance. A large number of intermediate values can impact comprehension as the programmer has to spend time identifying interesting intermediate values.

The *Whyline* system (Ko & Myers, 2004, 2009) inverts the debugging process, allowing users to ask questions about program values and responding by pointing to parts of the code responsible for the outcomes. Although this system improves the debugging process significantly, it can still only point to places in the program, which limits its explanatory power. In the realm of spreadsheets, the *goal-directed debugging* approach (Abraham & Erwig, 2007, 2005) goes one step further and also produces change suggestions that would fix errors. Change suggestions are a kind of counterfactual explanations.

Traces

Traces of program executions are a major source for explaining how outputs were produced from inputs. While traces are often used as a basis for debugging, they can support more general forms of explanations as well.

Constructing traces is by itself not difficult. But since traces can get quite large for even small programs, finding those parts in a trace that are crucial for an explanation poses a challenge. To address this problem, program slicing mechanisms have been employed to filter out irrelevant parts. Specifically, *dynamic slicing* is a technique for isolating segments of a program that potentially contribute to the value computed at a point of interest. For example, Biswas (1997) describes the generation of a dynamic slice for a higher-order programming language, and Ochoa *et al.* (2004) describe techniques for computing slices for a first-order lazy programming language.

Perera *et al.* (2012) describe the use of dynamic slicing on traces to specifically generate explanations for the executions of functional programs. Their approach supports slicing criteria that that make it possible to eliminate or select arbitrary portions of the output. This approach has been extended to imperative functional programs in Ricciotti *et al.* (2017). Our approach does not produce traces as explanations. Instead of collecting intermediate values over the execution of a program, value decomposition maintain a more granular representation of values that are still aggregated. Our approach requires some additional work on the part of the programmers in decomposing the inputs (even though in our library we have tried to minimize the required effort). An advantage of our approach is that we only record the information relevant to an explanation in contrast to generic tracing mechanisms, which generally have to record every computation that occurs in a program, and require aggressive filtering of traces afterwards.

9 Conclusions

We have introduced a general approach to explain the results of dynamic programming algorithms through value decompositions and minimal dominating sets: value decompositions offer more details about how decisions were made, and minimal dominating sets minimize the amount of information a user has to absorb to understand an explanation. We have demonstrated the wide applicability of the technique by integrating it into a library for dynamic programming, which requires only little effort from a programmer to get from a traditional, value-producing program to one that can also produce explanations of its results. This explanation component is modular and allows the explanations for one DP algorithm to be specialized to different application domains independently of its implementation. Moreover, the general explanation approach is not limited to dynamic programming; it can in principle be applied in any application domain. In addition to producing explanations in response to user requests, we have also shown how to anticipate questions about results and how to produce corresponding explanations automatically.

Conflict of Interest

None.

Acknowledgements

We are grateful to Sasha Rush for his DP library (Rush, 2009). Our implementation depends heavily on his code. This work is partially supported by DARPA under the grant N66001-17-2-4030 and by the National Science Foundation under the grant CCF-1717300.

References

Abraham, R. & Erwig, M. (2005) Goal-directed debugging of spreadsheets. In *IEEE International Symposium on Visual Languages and Human-Centric Computing*, pp. 37–44.

Abraham, R. & Erwig, M. (2007) GoalDebug: A spreadsheet debugger for end users. In *29th IEEE International Conference on Software Engineering*, pp. 251–260.

Ackley, D. H. (2013) Beyond efficiency. *Commun. ACM* **56**(10), 38–40.

Bartholdi, J. J. (2008) *The Knapsack Problem*. Boston, MA: Springer US, pp. 19–31.

Bauer, T., Erwig, M., Fern, A. & Pinto, J. (2011) Adaptation-based programming in Haskell. In *IFIP Working Conference on Domain-Specific Languages*, pp. 1–23.

Bellman, R. (1957a) A Markovian decision process. *J. Math. Mech.* **6**(5), 679–684.

Bellman, R. (1958) On a routing problem. *Q. Appl. Math.* **16**(1), 87–90.

Bellman, R. (1957b) *Dynamic Programming*, 1st edn. Princeton, NJ, USA: Princeton University Press.

Biswas, S. K. (1997) *Dynamic Slicing in Higher-Order Programming Languages*. Ph.D. thesis.

Cheney, J., Chiticariu, L. & Tan, W. (2009) Provenance in databases: Why, how, and where. *Found. Trends Databases* **1**(4), 379–474.

Dantzig, G. (1957) Discrete-variable extremum problems. *Oper. Res.* **5**, 266–288.

Erwig, M., Fern, A., Murali, M. & Koul, A. (2018). Explaining deep adaptive programs via reward decomposition. In *IJCAI/ECAI Workshop on Explainable Artificial Intelligence*, pp. 40–44.

Ford, L. R. & Fulkerson, D. R. (1956) Maximal flow through a network. *Canad. J. Math.*, **8**, 399–404.
Garfinkel, P. (1981) *Forms of Explanation*. New Haven, CT, USA: Yale University Press.
Gill, A. (2001) Debugging Haskell by observing intermediate data structures. *Electron. Notes Theoretical Comput. Sci.* **41**(1), 1.
Golan, J. S. (1999) *Semirings and Their Applications*. Dordrecht, Netherlands: Springer.
Goodman, J. (1999) Semiring parsing. *Comput. Linguist.* **25**(4), 573–605.
Green, T., Karvounarakis, G. & Tannen, V. (2007, June). Provenance semirings.
Juozapaitis, Z., Fern, A., Koul, A., Erwig, M. & Doshi-Velez, F. (2019). Explainable reinforcement learning via reward decomposition. In *IJCAI/ECAI Workshop on Explainable Artificial Intelligence*, pp. 47–53.
Khan, O. Z., Poupart, P. & Black, J. P. (2009) Minimal sufficient explanations for factored Markov decision processes. In *19th International Conference on Automated Planning and Scheduling*, pp. 194–200.
Khoo, Y. P., Foster, J. S. & Hicks, M. (2013) Expositor: Scriptable time-travel debugging with first-class traces. In *ACM/IEEE International Conference on Software Engineering*, pp. 352–361.
Ko, A. J. & Myers, B. A. (2004) Designing the Whyline: A debugging interface for asking questions about program behavior. In *SIGCHI Conference on Human Factors in Computing Systems*, pp. 151–158.
Ko, A. J. & Myers, B. A. (2009) Finding causes of program output with the Java Whyline. In *SIGCHI Conference on Human Factors in Computing Systems*, pp. 1569–1578.
Lipton, P. (1990) Contrastive explanation. *Royal Inst. Phil. Suppl.* **27**, 247–266.
Lipton, P. (2004) *Inference to the Best Explanation*. New York, NY, USA: Routledge.
Marceau, G., Cooper, G. H., Spiro, J. P., Krishnamurthi, S. & Reiss, S. P. (2007). The design and implementation of a dataflow language for scriptable debugging. *Autom. Softw. Eng.* **14**(1), 59–86.
Michie, D. (1968) "Memo" functions and machine learning. *Nature* **218**, 19–22.
Mohri, M. (2002) Semiring frameworks and algorithms for shortest-distance problems. *J. Autom. Lang. Comb.* **7**(3), 321–350.
Murphy, G. C., Kersten, M. & Findlater, L. (2006) How are Java software developers using the eclipse IDE? *IEEE Softw.* **23**(4), 76–83.
Nilsson, H. & Fritzson, P. (1994) Algorithmic debugging for lazy functional languages. *J. Funct. Program.* **4**(3), 337–369.
Nilsson, H. & Sparud, J. (1997) The evaluation dependence tree as a basis for lazy functional debugging. *Autom. Softw. Eng.* **4**(2), 121–150.
Ochoa, C., Silva, J. & Vidal, G. (2004) Dynamic slicing based on redex trails. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 123–134.
Parnin, C. & Orso, A. (2011) Are automated debugging techniques actually helping programmers? *International Symposium on Software Testing and Analysis*, pp. 199–209.
Perera, R., Acar, U. A., Cheney, J. & Levy, P. B. (2012) Functional programs that explain their work. In *17th ACM SIGPLAN International Confernce on Functional Programming*, pp. 365–376.
Pope, B. (2005) Declarative debugging with Buddha. In *5th International Conference on Advanced Functional Programming*, pp. 273–308.
Ricciotti, W., Stolarek, J., Perera, R., & Cheney, J. (2017) Imperative functional programs that explain their work. *Proc. ACM Program. Lang.* **1**, 14:1–14:28.
Roehm, T., Tiarks, R., Koschke, R. & Maalej, W. (2012) How do professional developers comprehend software? In *34th International Conference on Software Engineering*, pp. 255–265.
Rush, S. (2009) *Semirings Library*. https://github.com/srush/SemiRings/tree/master
Vardi, M. Y. (2020) Efficiency vs. resilience: What covid-19 teaches computing. *Commun. ACM* **63**(5), 9.
Vessey, I. (1986) Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybernet.* **16**(5), 621–637.
Zeller, A. (2002) Isolating cause-effect chains from computer programs. In *10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 1–10.