Extending the CLP engine for Reasoning under Uncertainty

Nicos Angelopoulos

Department of Computer Science, York University, York
nicos@cs.york.ac.uk

Abstract. We show how the amalgamation of Logic Programming with probabilistic reasoning enhances its capabilities for intelligent reasoning. Unlike current approaches we use concepts from Constraint Logic Programming in order to achieve this. In particular, we use the constraint store for storing probabilistic information and inference, and finite domains as sets of basic elements over which distributions can be defined. We describe a new language, Probabilistic finite domains and show how it can be used to code code two examples. First the Monty Hall problem is coded and the extensional means of simulating intelligence within our system are described. Second, we illustrate the benefits of the probabilistic information over the crisp finite domains in solving a simple encoding scheme. Aspects of a prototype implementation, a Prolog meta-interpreter, are discussed.

1 Introduction

Logical and probabilistic reasoning are integral parts of intelligent behaviour. In a systemic context, Logic Programming (LP) has been successful as a platform for implementing Logic based intelligent systems. Although there have been a number of proposals for enhancing LP with probabilistic reasoning capabilities, there has not emerged a platform for providing these capabilities in a general problem solving setting.

Current probabilistic extensions to Logic Programming can be thought of as either Bayesian [9, 8] or Markovian [4, 6]. These attach the probabilities either to the clause definition, or to clauses of restricted form. The Bayesian approaches have difficulties with interpreting the probability of predicates that include variables and in general the form of the clause is altered substantially. In effect, the logical (crisp) reasoning is subordinate to the probabilistic inference. On the other hand, Markovian approaches subordinate the latter to the former; as a result they are more successful for parameter estimation and machine learning tasks rather than as a tool for modelling intelligent behaviour.

Our approach, Probabilistic finite domains (*Pfd*) is in effect a Bayesian approach, however, probabilistic information is not presented in the form of clauses but as constraints. By using Constraint Logic Programming (CLP) ideas on how to extend the reasoning capabilities of the Logic Programming engine, *Pfd* alleviates some of the Bayesian limitations. First we use the store to hold probabilistic information, thus extricating the deduction process from probabilistic inference tasks. Second, probability distributions are ascribed over finite domains. Third, probabilistic events are mapped to goals containing probabilistic variables.

We assume some familiarity with LP and grasp of the basic intuitions in Constraint Logic Programming. The rest of the paper is structured as follows. The following section briefly introduces the *Pfd* language and an implementation of the language. Section 3 details a *Pfd* program for the Monty Hall problem. Section 4 illustrates how probabilistic information can enhance crisp computation in a Caesar encoding scenario. Final conclusions are in Section 5.

2 The language

Probabilistic Variables. A finite domain variable is a logical variable with a restricted set of possible values (its finite domain). Here we will we use lists of distinct objects (e.g. $[a, b, c, \ldots]$) for finite domains. Based on this, a probabilistic variable has two parts; (a) a *finite domain*, which at each stage holds the collection of possible values that can be assigned to the variable, and (b) a *probability assigning function* (or simply function) which is used to assign probabilities to the elements of the domain. The probability function declares the basic statistical behaviour of the variable. In *Pfd*, the two constituents of a probabilistic variable are kept separate. There are two benefits in such an approach. Firstly, the variable is still capable of participating in finite domain constraints. Secondly, probabilistic functions capture the probabilistic behaviour in a manner which is, to a large extent, independent of specific domain values.

Construct 1 Let Fd be a list of distinct objects representing a finite domain, $\phi_{\mathtt{V}}$ be a probability function defined over all sublists of Fd and $Arguments$ a list of arguments that customise ϕ_V. Probabilistic variable V is introduced to the store with $V \in_{\mathbf{p}} \phi_{\mathtt{V}}(Fd, Arguments)$

Two examples of how the introduced variable definition can be applied in practice are: $Die \in_{\mathbf{p}} \mathtt{uniform}([i, ii, iii, iv, v, vi])$

and $Coin \in_{\mathbf{p}} \mathtt{finite\_geometric}([head, tail], 2)$.

Function `uniform` takes no other arguments. The behaviour it describes is that the remaining elements of its domain (a sublist of $[i, ii, iii, iv, v, vi]$ in this case) are, in the absence of other evidence, equiprobable. Function `finite_geometric` defines a geometric probability distribution over the domain elements. In this instance, the deterioration factor of the progression is 2. Table 1, presents some variations of how the two examples, can be used. Rows three and four show the formulae for these two functions, while the lower rows give concrete examples with associated distributions.

Events. In Probability Theory events collate subsets of random variables' values into semantic units. Predicates containing probabilistic variables can be viewed as such events over the Cartesian space of these probabilistic variables. The main intuition is that the probability assigned to an event is proportional to the space covered by combinations leading to successful derivations. Let $pvars(E)$ be the

$\phi_{\mathtt{V}}$	Fd	$Arguments$	$Distribution$
Descriptions			
`uniform`	$[El_1, \ldots, El_n]$		$[1/n, \cdots, 1/n]$
`finite_geometric`	$[El_1, \ldots, El_n]$	F	$[_{j=1}^{n} F^{n-j} / \sum_{i=0}^{n-1} F^i]$
Examples			
`uniform`	$[i, ii, iii, iv, v, vi]$		$[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$
`uniform`	$[i, ii, iv, v, vi]$		$[1/5, 1/5, 1/5, 1/5, 1/5]$
`finite_geometric`	$[head, tail]$	2	$[2/3, 1/3]$
`finite_geometric`	$[low, med, high]$	2	$[4/7, 2/7, 1/7]$

Table 1. Examples of probabilistic variable definitions

lucky(iv, head). lucky(v, head). lucky(vi, head).	?- $Die \in_{\mathbf{p}}$ `uniform`$([i, ii, iii, iv, v, vi])$, $LCoin \in_{\mathbf{p}}$ `finite_geometric`$([head, tail], 2)$, $PLucky$ *is* $\mathbf{p}(\mathtt{lucky(Die, LCoin)})$. PLucky $= 1/3$

Table 2. Left: Definition for lucky/2. Right: Probability of event example query.

probabilistic variables in E, e be a vector collecting one element from the finite domain of each variable in $pvars(E)$, $p(e)$ be the corresponding vector of probability values, $\mathcal{P}$ be the program defining E, and $\mathcal{S}$ be a particular constraint store (a set of constraints). E/e denotes predicate E with its probabilistic variables replaced by their respective elements in e. Also, in what follows we use $\mathcal{S} \vdash$, store-derives, to mean $\mathcal{S} \cup \mathcal{P} \vdash$, store-and-program-derive. The probability of predicate E with respect to store $\mathcal{S}$ is

$$P_{\mathcal{S}}(E) = \sum_{\substack{\forall e \\ \mathcal{S} \vdash E/e}} P_{\mathcal{S}}(e) = \sum_{\substack{\forall e \\ \mathcal{S} \vdash E/e}} \prod_i p(e_i)$$

Construct 2. The probability of predicate E with respect to current store S is assigned to unbound logical variable *Probability* by : *Probability is* $\mathbf{p}(\mathtt{E})$

An example of how to query the probability of a predicate is shown in Table 2.

Conditional Constraint. The store may contain different kinds of probabilistic information involving probabilistic variables. Here we use a restricted form of the conditional constraints presented in [1].

Construct 3. For probabilistic variables $Dependent$ and $Qualifier$, and probability $\pi, 0 \leq \pi \leq 1$, the conditional states: $Dependent \mid\!\neq_{\pi} Qualifier$

If no π is given, it is assumed to be equal to one. The conditional should be read as $Dependent \neq X : \pi \oplus Dependent = X : (1 - \pi) \mid Qualifier = X$. For a free variable X, this reads as the value of $Dependent$ being different from that of $Qualifier$ with probability π and equal to it with probability $1 - \pi$. We say that $Dependent$ depends on $Qualifier$.

For the probability of *Dependent* in an arbitrary store we need to consider the following: (a) each variable, upon which it depends must be assigned a value from its finite domain (b) the weight of each such assignment is equal to the product of the probability for the assigned elements (c) all possible assignments must be considered (d) $\neq$ and $=$ parts, across multiple constraints conditioning the variable, are considered in a multiplicative way ($\otimes$).

Finite domain solvers provide labelling predicates. These, instantiate a variable to one element of each domain at a time, backtracking into the predicate in order to instantiate the variable to all possible values. Similarly, *Pfd* provides for the labelling of variables the $\mathbf{label}(Vars, Select, Probs, Total)$ predicate. *Vars* are labelled by the method described by *Select*. Associated probabilities are in $Probs$. $Total$ holds the sum of products of all $Probs$ seen so far. $Select$ directs the way in which the probability associated with the finite domain elements influences labelling.

A meta-interpreter. We have built a prototype system implementing *Pfd*. This has the form of a Prolog meta-interpreter. The full featured system runs on SICStus 3.8.5 or later, while restricted versions run on the Yap 2.3.19 and Swi 5.0.1 Prolog systems. The main restriction, for the two latter systems, is the lack of FD interaction since they do not provide a clp(fd) library. In what follows, we refer to the system behaviour under the SICStus engine. Features of the meta-interpreter include:

1. Rational numbers are represented as quotients of integers, with the underlying prolog engine providing fast, arbitrarily long integer arithmetic.
2. An approximate algorithm, labelling many variables at once. This is selected with `mua` labelling method.
3. Graph algorithms maintain the consistency of variable dependencies.

The labelling method used here, `mua` (maximum unique alternative), implements an approximate search on the tree of all possible valuations. Unlike other approaches this approximation does not compromise the semantics of the reasoning. What is approximated is the order in which the different combinations of values for the labelled variables appear. Some form of approximation is in general needed in this setting, since finding the most promising combination in most cases collapses to a breadth first search. Our algorithm traverses the space of possible valuations according to an approximating heuristic. Each branch upon introduction to the tree, is assigned a value which approximates the branch's final probability. At each stage the children of the most promising branch are added to the tree (in the case of a leaf, its total probability is returned to the top level). Thus, at a random time the tree is expected to have branches at various depths. Children of the selected branch are found by instantiating one of the still free variables to a number of its possible values (starting from the most probable ones). The metric used in `mua` is $M = ParentM + f(MarginalP - MaxAlternativePrb)$ where M and $ParentM$ are the obvious variables, $MarginalP$ is the probability of the instantiated curtains(gamma, Swt, Prb) :-

$Gift \in_{\mathbf{p}}$ `uniform`$([a, b, c])$,	$First \in_{\mathbf{p}}$ `uniform`$([a, b, c])$,
$Reveal \in_{\mathbf{p}}$ `uniform`$([a, b, c])$,	$Second \in_{\mathbf{p}}$ `uniform`$([a, b, c])$,
$Reveal \mid\!\neq Gift$,	$Reveal \mid\!\neq First$,
$Second \mid\!\neq Reveal$,	$Second \mid\!\neq_{Swt} First$,
$Prb\ is\ \mathbf{p}(Second{=}Gift)$.	

Fig. 1. *Pfd* clause for probability of win under strategy γ.

$Variable$ to assume the particular value (which differentiates this child from its siblings) and $MaxAlternativePrb$ is the maximum probability assigned to this value in any of the non selected Variables. Function f ensures that the resulting number is an integer value within certain limits. These limits are evaluated when setting-up the labelling procedure and depend on the number of variables and the number of alternative values for each one of them.

3 Monty Hall

We first examine a problem which is often used to demonstrates that reasoning with probability is hard and often leads to wrong answers. In the Monty hall tv show, [5, p.137] there are three curtains. Behind these, a car and two goats are concealed. A contestant chooses an initial curtain. One of the remaining curtains is then drawn revealing one of the goats. The contestant is hence asked to make a final choice. Whatever is behind this final curtain is the contestant's gift. We will say the contestant won if his final choice is the curtain concealing the car.

Can the contestant develop a strategy that will increase their winning chances ? By far, most people believe that this is not possible, and that both options at the second choice (stay-with-same, against switch-to-other curtain) have the same probability (1/3). However, a closer look reveals that the latter choice is twice as likely to lead to a win. (For a justification see [5].) Furthermore, both choices can be seen as part of a single strategy; that of switching with probability Swt. The first, stay-with-same is equivalent to $Swt = 0$ and switch-to-other to $Swt = 1$. Lets call this unifying strategy γ . Then probability of win is $P(\gamma) = \frac{1+Swt}{3}$.

An intelligent system would ideally be able to construct the analytical formula from a basic description of the problem. However, this is a very ambitious target. Instead, we require that the system, given a description, should perform computations that provide the same answers to those computed by the formula. In *Pfd* the program modelling the game, is used to extensionally compute the answers given by the analytical formula, from the constituents parts of the problem. The *Pfd* clause for strategy γ is shown in Fig. 2. The reading for the clause is that Prb is the probability of a win given that the player switched with probability Swt. Probabilistic variables *First*, *Gift*, *Reveal* and *Second* have no prior information that distinguishes the three values in their domains. Constraint $\mid\!\neq$ is used to declare pairs of variables that must be different (for example the

dictionary words	amicable	cohered	euphoria	mutant	shame	verb
Caesar encoding	twcbtspo	bfhoyoa	onrhfyct	wnvtdv	mhtwo	loys

dict. letter	a	b	c	d	e	h	i	l	m	o	p	r	s	t	u	v
encoded by	t	s	b	a	o	h	c	p	w	f	r	y	m	v	n	l

Table 3. Example encoding. Letters not in dictionary words, are not encoded.

revealed curtain cannot be the player's first choice *Reveal* $\neq$ *First*). Variable *Second* is defined to be different than *First* with probability *Swt* and to be the same as *First* with probability $1 - Swt$. $P(\gamma)$ is assigned to *Prb* with the call to `is/2`. Querying this program with `curtains(gamma, 1/2, Prb)` provides the answer $Prb = 1/2$. The meta-interpreter, when trying to prove `is/2` starts giving values to *Gift*, *First* and *Reveal*. This is because variables in $Second = Gift$, the goal for which we wish to find the probability, depend on them. The distribution of *Second* can then be established for each set of values from the final conditional. Note that each such set has an associated probability. The total probability of *Second* is equal to the sum of all sub-probabilities.

4 Caesar encoding

Caesar's scheme is one of the earliest encoding schemes [7], where each letter of the alphabet is encoded by another (unique) letter. Also, each letter in the encoding correspond to a single decoded letter (see Table 3 for an example of a valid encoding). We consider codings of words formed from lower case letters drawn from a dictionary (the Unix[1] words dictionary was used in our experiments). We let each input, encoded, letter correspond to a unique variable. The domain of this variable is all the possible decodings of the encoded letter it stands for (i.e. all letters in the alphabet). For ease of reference we will use X_i for the variable corresponding to the ith lower case letter. For example, the encoded word 'bfhoyoa', will be represented by $< X_2, X_6, X_8, X_{15}, X_{25}, X_{15}, X_1 >$ with $X_i \in \{a, \ldots, z\}$. Furthermore the desired solution for this example is : $< X_2 = c, X_6 = o, X_8 = h, X_{15} = e, X_{25} = r, X_1 = d >$. So, in order to get the original words (decode) we replace in encoded words c's for b's, o's for h's , etc.

The model, so far, is applicable to both finite domain variables and to probabilistic ones. The full size of the search tree for a word with four free variables, which is our starting point, is $26!/22! = 358800$. The classical approach in solving the Caesar encodings, is to order the combinations in accordance to some likelihood measure. Normally this is done with specialised algorithms in imperative languages and the order is dictated by the proximity of two frequencies. The first frequency is the relative number of occurrences of a single letter in the dictionary and the second is the letter occurrences in the encoded words. The

[1] in our system this contains 25143 words

D_i	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
freq(D_i)	24	8	14	12	31	5	8	6	23	0	1	14	6	20	16	4	0	18	18	16	7	2	1	1	4	1

X_i	E_i	$f(E_i)$	$px(X_i, a)$	$px(X_i, b)$	$px(X_i, c)$	$px(X_i, d)$	*out of*	...
X_1	a	5/38	5883	5731	5788	5769	/149500	...
X_2	b	2/38	1898	1942	1993	1980	/49900	...
...	...			...				...

Table 4. Top: dictionary frequencies in units of 260 occurances. Bottom: Proximity based probabilities.

```
probabilistic_method( proximity(Fd,Marker,Frqs,Prbs) ) :-
    rationals_subtract_list_from( Frqs, Marker, Subtr ),
    maplist( rationals_abs, Subtr, Abstr ),
    rationals_add_list( Abstr, Sum ),
    rationals_subtract_list_from( Abstr, Sum, Diffs ),
    rationals_to_probabilities( Diffs, Prbs ).
```

Fig. 2. Clause for Proximity Method Definition

intuition behind this approach is that letters will appear equally often in both, the dictionary and in the encoded words.

For example the frequency of encoded letters can be used as a marker. Then, the probabilistic distribution for the corresponding variable is a function of the differences the dictionary frequencies have to this marker. In Table 4 we give an example of some calculated values for our **proximity** function ($px(X, D)$, which calculates the proximity probability between encoded and dictionary letters). These calculations are based on the frequencies of encoded letters X_i and dictionary letters D_i, $freq(X_i)$ and $freq(D_i)$ respectively. In full we have:

$$px(X_i, D_j) = \frac{\mid freq(X_i) - freq(D_j) \mid}{\sum_k \mid freq(X_i) - freq(D_k) \mid} \tag{1}$$

Programs. We have programmed two solutions to the Caesar encoding. One based on clp(fd) [2] and a *Pfd* one, based on the proximity probabilities given above. Both run on the SICStus prolog engine. However, clp(fd) is a system library, while the *Pfd* solution runs over an extra meta-interpretation layer. We have tried to make the two programs as similar as possible, while taking care not to compromise the efficiency of either. The two programs (fd.pl and pfd.pl) can be downloaded from `http://www.doc.ic.ac.uk/~nicos/sware/pbs/caesar/`. The *Pfd* program differs to the clp(fd) one, in two aspects: (a) in the variable definitions and (b) in labelling. Probabilistic variables are introduced with $X_i \in_{\mathbf{p}}$ `proximity`($Alphabet, [Marker, Freqs]$) with X_i, the variable declared as probabilistic. *Alphabet* and $Freqs$ are invariants that refer to the dictionary alphabet ('a'-'z' here) and frequencies. $Marker$ is the frequency of the encoded letter that corresponds to the variable X_i. The probability construction method `proximity`, an implementation of Eq. 1, is shown in Figure 2. Predicates with names prefixed by *rationals*, refer to a library implementing rational arithmetics.

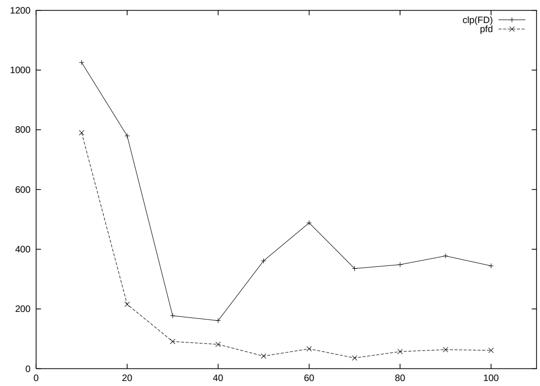

Fig. 3. *Pfd* versus clp(fd) timings comparison.

The decoding of words proceeds by selecting the word which currently has the least degrees of freedom (equal to the number of distinct variables times the elements in each variable's domain). All variables in the word are then labelled and their instantiated form checked against the decoded words. The predicate succeeds on backtracking, until a word is found in the dictionary. For word variables, *WordVs*, the call $\textbf{label}(WordVs, \texttt{mua}, Probs, AccProb)$ was used for labelling in the *Pfd* program (Section 2).

Results. For each N in $\{10, 20, \ldots, 100\}$ we encoded 10 sets of N random words from our dictionary. Only words of four distinct letters or more were considered. The comparative mean execution times for the *Pfd* and clp(fd) programs, in solving these sets of encodings are shown in Fig. 3. In Caesar encoding problems the hardest word to decode is very often by far the first one. For the solution presented here, as well as most of the solutions in the literature, the easiness with which the solution is found, is determined by the closeness of the two letter frequencies (dictionary versus encoded). This in turn is dependent on the number of words to decode. Intuitively, the more words to decode, the more likely it is to have a frequency similar to the dictionary one. Thus fewer combinations will have to be considered before finding the correct decoding. (Note there is no such concept as a hard or easy encoding for a given set of words.)

Our experiments found a boundary in the region of thirty words. Thirty or more random words, as can been seen in Fig. 3 provide enough probabilistic information that help *Pfd* solve the problem in time significantly reduced in comparison to the clp(fd) program. Because clp(fd) is implemented natively whereas *Pfd* relies on an extra layer of meta-interpretation, it is likely that the improvement in performance effected by *Pfd* is greater than Fig. 3 suggests. The increase in time required by clp(fd) occurring after the thirty word mark is mainly due to the amount of wrong decodings the program has to try before it finds the correct one.

Another encouraging factor, in the analysis of performance, is that approximately half the of the execution time, is consumed in garbage collection. The main data-structure that attributes to this is the `mua` labelling procedure. This will be substantially reduced in an Object Oriented or imperative implementation of *Pfd* or even in Prolog systems that implement *Pfd* labelling natively.

5 Conclusions

We presented a language integrating logical and probabilistic reasoning seamlessly. This integration is crucial in intelligent systems platforms. Our approach is based on constraints ideas on how to extend the LP paradigm. The integration is facilitated by the use of (a) the store for probabilistic knowledge representation and reasoning (b) finite domains as units over which probabilistic distributions are defined and (c) reading of goals as events in the probability space.

A prototype implementing *Pfd* has been built to provide initial feedback. This can be downloaded and tested on-line at `http://www.doc.ic.ac.uk/~nicos/sware/pfds/`. The prototype system has been used to demonstrate both the soundness of probabilistic inference from intuitive first principles and the exploitation of probabilistic information for improvements in performance within a declarative framework. However, the meta-interpreter presented here is a research tool, using very simple techniques. Past trends in Constraint Programming have shown that constraint networks are best implemented using Object Oriented or imperative languages. For example, [3] describes such an approach within CLP. We feel that similar techniques will also lead to substantial performance improvements for *Pfd* constraints.

Acknowledgements. Thanks to Tarhata Orlanes for corrections in the prose, to James Cussens for forcing me to think more about the maths and to the anonymous referees for their corrections and suggestions.

References

1. N. Angelopoulos. *Probabilistic Finite Domains.* PhD thesis, City U., London, 2001.
2. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Progr. Languages: Implem., Logics, and Programs*, 1997.
3. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
4. James Cussens. Stochastic logic programs: Sampling, inference and applications. In *16th Conference on Uncertainty in AI (UAI-2000)*, pages 115–122, 2000.
5. C. M. Grinstead and J. L. Snell. *Introduction to Probability.* AMS, 1997.
6. Yoshitaka Kameya and Taisuke Sato. Efficient learning with tabulation for parameterized logic programs. In *1st Int. Conf. on Comput. Logic*, pages 269–294, 2000.
7. Alan G. Konheim. *Cryptography, a primer.* John Willey & Sons, 1981.
8. Thomas Lukasiewicz. Probabilistic logic programming. In *13th biennial European Conference on Artificial Intelligence*, pages 388–392, Brighton, Uk, August 1999.
9. R. Ng and V. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101:150–201, 1992.