Belief Propagation with Strings

Anonymous Author(s)
Affiliation
Address
email

Abstract

Strings and string operations are very widely used, particularly in applications that involve text, speech or sequences. Yet the vast majority of probabilistic models contain only numerical random variables, not strings. In this paper, we show how belief propagation can be applied to do inference in models with string random variables which use common string operations like concatenation, find/replace and formatting. Our approach is to use weighted finite state automata to represent messages and transducers to perform message computations. Using belief propagation mean that string variables can be mixed with numerical variables to create rich hybrid models. We illustrate this approach by showing inference results for hybrid models with string and numerical variables in the domains of information extraction and computational biology.

1 Introduction

Strings and string operations are very widely used, typically for displaying information in a human-readable form or for parsing text from an external source. However, the vast majority of probabilistic models in use today include only numerical random variables, such as real, integer or boolean variables. Where text is modelled (for example using LDA [1]) it is usually broken into words or n-grams and then represented using integers. There has been relatively little investigation into models and inference algorithms where the variables in the model are themselves strings.

The ability to do inference directly over string variables would be of great help in applications such as information extraction and machine translation due to the need to reason about uncertain interpretations of text. Strings can also be generalised to any sequence: for example, they can be used to represent sequences of biological molecules such as DNA nucleotides or amino acids, suggesting that probabilistic models with strings would find many applications in computational biology. When probabilistic models are being defined using a probabilistic programming language (such as Infer.NET [2] or Church [3]), it is particularly helpful to allow string random variables since it is very natural for the programmer to want to write out or read in text, just as they do in a conventional programming language. Existing probabilistic programming languages provide rich functionality for running programs with numerical random variables, but provide little support for efficient execution of programs with string variables. Historically, unstructured text has been an anathema to programmers – but a probabilistic programming language with good string support would make getting useful, computable information out of unstructured text a much easier task.

In this paper, we use the Infer.NET inference framework and exploit its extensible factor API to add support for string random variables and string operations. Infer.NET primarily performs inference using message passing algorithms such as expectation propagation [4] and variational message passing [5]. To add string support, we chose to support expectation propagation, although given our choice of distribution type, this simplifies to belief propagation. Handling strings using a standard algorithm means that string variables can be used in combination with all other kinds of variables that are already supported by Infer.NET, to allow rich hybrid modelling.

We model distributions over strings using weighted finite state automata and compute string operations using weighted finite state transducers. Both of these have been widely used in previous machine learning applications such as speech recognition [6], computational biology [7, 8] and machine translation [9], amongst many others. Our contribution is to provide a general belief propagation algorithm for handling strings and their common operations (concatenation, substring, formatting etc.) that applies automata and transducers automatically, given an appropriate model. Our aim is to broaden the applicability of automatic inference systems to the many domains where string- or sequence-based reasoning is critical.

2 Belief propagation in models with string variables

In this section we explain how belief propagation (BP) [10] can be used to perform inference in models with string random variables. In BP, inference is performed by passing messages, or beliefs, between factors and variables in a model according to some schedule. There are two types of messages: from a factor to a variable, and from a variable to a factor, and both are functions of the involved variable. The message from a factor f to a variable x is

$$\mu_{f\to x}(x) = \sum_{X_f\setminus\{x\}} f(X_f) \prod_{x'\in X_f\setminus\{x\}} \mu_{x'\to f}(x'), \tag{1}$$

where X_f is the set of variables connected to f. The message from a variable x to a factor f is

$$\mu_{x\to f}(x) = \prod_{f'\in F_x\setminus\{f\}} \mu_{f'\to x}(x), \tag{2}$$

where F_x is the set of factors connected to the variable x. The marginal of a variable x can then be computed using $p(x) \propto \prod_{f\in F_x} \mu_{f\to x}(x)$.

We want to run belief propagation in models that use string random variables and string operations. To be precise, we wish to support the following set of commonly-used string operations:

- concatenating two strings (*Concat*),
- extracting a substring of a string (*Substring*),
- finding whether a string contains another string and, if so, where (*IndexOf*),
- trimming whitespace (*TrimLeft*, *TrimRight*) and changing case (*ToLower*, *ToUpper*),
- finding and replacing a string (*Replace*) or a character (*ReplaceChar*).

These common operations can be used to construct more complex ones. For example, consider string formatting where a template like "$\{0\}$ was born on $\{1\}$" is filled in by providing strings for the first and second placeholders. This operation can be constructed using multiple consecutive *Replace* operations.

We will next illustrate how, in principle, belief propagation can be used to perform inference in such models. In section 3, we go on to explain how we implemented the described scheme in practice.

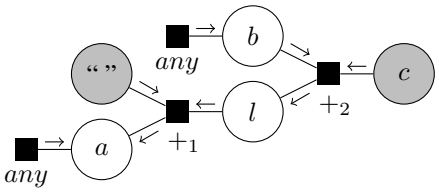
2.1 Applying BP to models with strings

We will use the "Hello world" probabilistic program of Fig. 1a to illustrate the application of BP to models with strings. In this program there are two string random variables, a and b, which are sampled from a uniform distribution over all possible strings. These variables are then concatenated together with a space in between to obtain a third variable, c, which is observed to be equal to the string "Hello uncertain world". We are now interested in the marginal distributions over a and b conditioned on this value of c.

Since the number of possible strings is infinite, the priors over a and b are improper distributions. However, the inference task is still well-defined. In fact, improper priors can be extremely useful when dealing with strings, since it is often convenient to express the infinite, but well-defined, subset of strings which are allowed to have non-zero probability (for example, email addresses of unbounded length).

```
var a = StringUniform();
var b = StringUniform();
var c = a + " " + b;
c.ObservedValue = "Hello uncertain world";
```

(a)



(b)

Figure 1: (a) An example Infer.NET program defining a probability distribution over three string variables, one of which is observed. (b) The corresponding factor graph. Subscripts have been added to distinguish between different instances of the concatenation factor. Arrows show the message-passing schedule for computing the marginal of a.

To help understand how to apply belief propagation to the probabilistic program of Fig. 1a, it can be useful to draw the corresponding factor graph to visualize the message passing process. A factor graph for our probabilistic program is shown in Fig. 1b (for later examples we will only show the program, since factor graphs become large even for small programs). Since our graph is a tree, to compute the marginal of a, we use the message passing schedule shown in Fig. 1b. We will now describe, at a high level, how to perform inference according to this schedule:

1. The prior sends b the message $\mu_{any\to b}(b) = 1$, which is then sent on to $+_2$ as $\mu_{b\to+_2}(b)$.
2. Then c sends $+_2$ a point mass message $\mu_{c\to+_2}(c) = \mathbb{1}[c = \text{“Hello uncertain world”}]$.
3. The factor $+_2$ sends
$$\mu_{+_2\to l}(l) = \sum_{b,c} \mathbb{1}[l + b = c]\mu_{b\to+_2}(b)\mu_{c\to+_2}(c)$$
$$= \sum_{b} \mathbb{1}[l + b = \text{“Hello uncertain world”}] = \mathbb{1}[l \text{ is a prefix of “Hello uncertain world”}]$$
to the intermediate concatenation result l. The message is then sent on to $+_1$ as $\mu_{l\to+_1}(l)$.
4. The factor $+_1$ sends a message to the variable a,
$$\mu_{+_1\to a}(a) = \sum_{l} \mathbb{1}[a + \text{“ ”} = l]\mu_{l\to+_1}(l)$$
$$= \mathbb{1}[a = \text{“Hello”}] + \mathbb{1}[a = \text{“Hello uncertain”}].$$
5. The variable a receives a uniform message $\mu_{any\to a}(a) = 1$ from its prior and computes its marginal $p(a) \propto \mu_{any\to a}(a)\mu_{+_1\to a}(a)$, which is a uniform distribution over the strings “Hello” and “Hello uncertain”.

Similarly, the marginal of b can be computed by propagating the belief *any string followed by a space* from $+_1$ to $+_2$. The factor $+_2$ can then use this belief to compute and send b a uniform message over “uncertain world” and “world”. This message is also the marginal of b since the message from its prior is uniform.

This toy example illustrates that in order to implement this hypothetical scheme in practice, we need to fulfill the following requirements:

- Beliefs like *any string*, *any prefix of a certain string*, *any string followed by a space* etc. need to be represented in a unified, compact and efficient manner;
- Products of messages in our chosen representation must be efficiently computable, in order to work out marginals and messages from variables to factors;
- For each factor (string operation) that we want to support, it must be possible to compute the outgoing messages from that factor using (1);
- Normalizers must be efficiently computable for computing marginals (unless the distribution is improper);
- Additionally, if one wants to define mixture models via gates [11], the representation must allow for computing weighted sums of messages.

3 Message passing using weighted finite state transducers

To meet these requirements, we need to use a message that itself has structure. Various types of structured message have previously been used to good effect in inference algorithms, such as Algebraic Decision Diagrams [12] or confactors [13]. To model distributions over strings, however, we need to use a different kind of structured message: a weighted finite state automaton. Indeed, the central idea of this paper is that we use:

- weighted finite state automata (WFSA) to represent messages and marginals,
- N-way weighted finite state transducers (WFST) to represent factors.

If this representation is used, we will show that all BP computations can be performed solely by applying automaton and transducer operations.

3.1 Weighted finite state transducers

Weighted finite state transducers [6] constitute a class of N-way functions mapping tuples of sequences into elements of a semiring. 1-way transducers are also known as weighted finite state automata. In this paper we limit our attention to transducers that map tuples of strings to real values.

An N-way transducer can be defined by a directed graph with annotations on its vertices, or states, and its edges, which we call transitions here. Each transducer state can be labeled as a start state, an end state, or both. Each transition is given a weight and a tuple of N characters from the alphabet of interest, which may include the special value ε indicating no character. The value of an N-way transducer on a string tuple $(x_1, x_2, \ldots, x_N)$ can be defined by the following procedure:

1. Find all paths between start and end states such that if you concatenate first characters of every character tuple along the path (omitting ε), you get x_1, if you concatenate second characters, you get x_2 and so on up to x_N.
2. Compute the weight of each path by multiplying the weights of all transitions in the path.
3. Compute the sum of path weights to obtain the value of the transducer on the string tuple.

Dynamic programming can be used in practice to perform this computation efficiently [14].

For example, the transducer shown in Fig. 2 defines a function that is 1 if and only if its second argument equals to its first with all leading spaces removed. Paths through the transducer first loop through s_1, effectively consuming all leading spaces in the first argument. They then either stop there, if the second string is empty, or proceed to s_2. The loop at s_2 ensures that the arguments are equal after the leading spaces have been consumed. Here we use a shorthand notation $Copy_{ij}$ to denote a set of all possible tuples (which, in turns, induces a set of transitions) of the form

$$(\varepsilon : \ldots : \varepsilon : \underset{i}{c} : \varepsilon : \ldots : \varepsilon : \underset{j}{c} : \varepsilon : \ldots : \varepsilon),$$

where c takes any character value and is present in tuples in positions i and j. We will also denote $Copy_{11}$ by $(* : \varepsilon)$ for 2-way transducers, and by $(*)$ for automata. In our actual implementation of a weighted finite state transducer library such transition sets are not represented explicitly, but rather use a special type of edge.

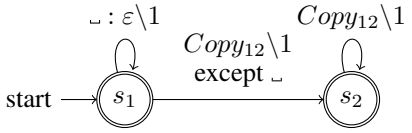


Figure 2: An example of a 2-way transducer for the *TrimLeft* operator. End states are indicated by double circles. Characters in tuples are separated by a colon. Transition weights are separated from character tuples by a backslash – from now on these will be omitted if they are equal to 1.

An important property of transducers is that they are closed under summation, multiplication, scaling and also projection of an automaton onto a transducer

$$\text{proj}[T, A, x_k](x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_N) = \sum_{x_k} T(x_1, \ldots, x_k, \ldots, x_N) A(x_k), \tag{3}$$

where T is an N-way transducer, A is an automaton and the result is an $(N-1)$-way transducer with a specified variable summed out. Moreover, there exist efficient algorithms for performing these operations on transducers represented as graphs [15]. Computing the sum of all transducer values is also tractable.

3.2 Message computations

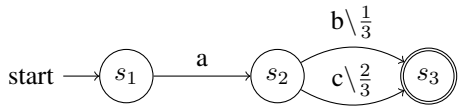
If we represent messages and marginals as automata, we can capture the kinds of string distribution that arose in section 2.1. For example, Fig. 3 shows several automata representing these kinds of string distribution. When messages are represented using automata, marginal computation and variable-to-factor message computation (2) reduce to multiplying the automata representing those messages.

Computing a factor-to-variable message (1) can be done by using (3) to project incoming messages from all variables, except the target one, onto the transducer representing the factor. It follows from the fact that transducers are closed under operations needed for message passing that all beliefs which can arise from transducer-representable operations are representable as automata. Fig. 4 shows how transducers can represent the *Concat*, *ToUpper*, *ReplaceChar* and *Replace* operations. It should be noted that the behavior of the *Replace* operation shown in Fig. 4d is different from what is usually available in modern programming languages: it replaces one *random* occurence of a substring within a string. Representing a more conventional version of *Replace* via a transducer is also possible, if required, but requires a significantly more complex (and so slower) transducer.

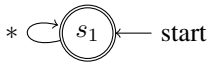
The *Substring* and *IndexOf* operations differ in that not all of their arguments are strings. In this case, messages can still be computed if we can create a transducer that represents the factor with non-string variables summed out. As an example, consider the *Substring* operation

$$f(x, y, p, l) = \mathbb{1}\big[y = x[p \ldots (p+l)]\big]. \tag{4}$$

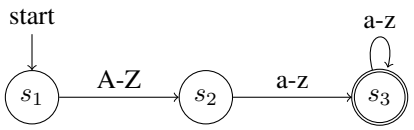
Suppose that the substring position p is known to be 1, while the substring length l is either 1 with probability 0.7, or 2 with probability 0.3. Fig. 5 shows an example of a transducer $\tilde{f}(x, y) = 0.7 f(x, y, 1, 1) + 0.3 f(x, y, 1, 2)$, that can be used for computing messages to x and y. Such a transducer can be built either from $f(x, y, p, l)$ by using transducer summation and scaling operations, or manually, if an efficient representation of it is known. Messages for the *IndexOf* operation can be computed using a similar transducer – the only difference is that we do not impose any constraints on the length of the substring.



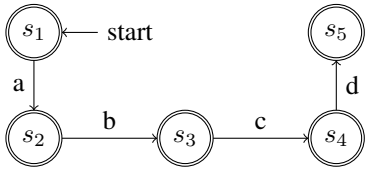
(a) $f(x) = \frac{1}{3}\mathbb{1}[x = \text{“ab”}] + \frac{2}{3}\mathbb{1}[x = \text{“ac”}]$



(b) $f(x) = 1$

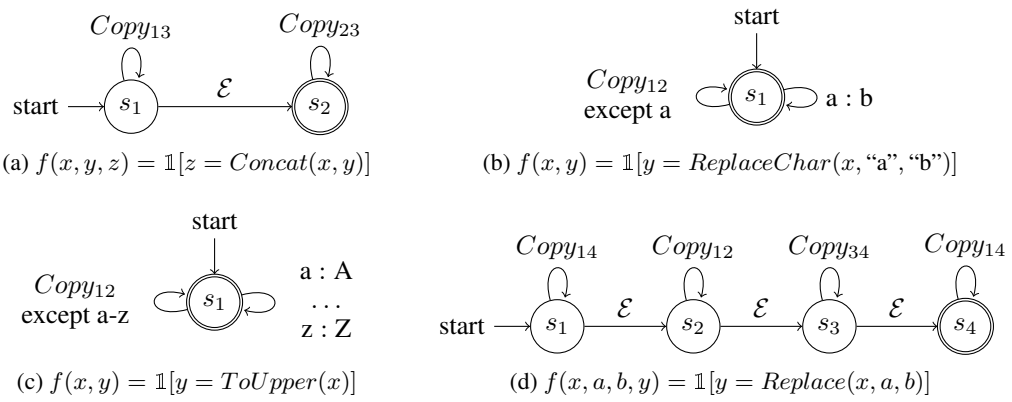


(c) $f(x) = \mathbb{1}[x \text{ is a capitalized word}, |x| \geq 2]$



(d) $f(x) = \mathbb{1}[x \text{ is a prefix of “abcd”}]$

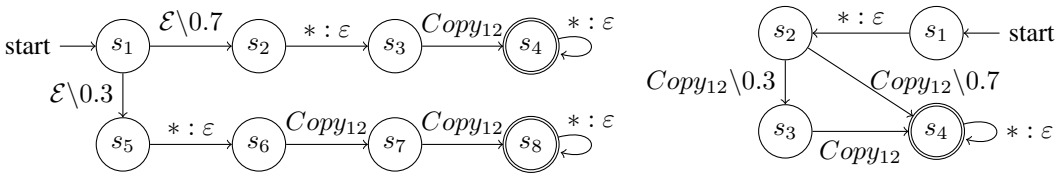
Figure 3: Examples of string beliefs represented as automata. A-Z stands for all transitions with characters from A to Z. Automata containing loops e.g. (b,c) define improper distributions.



(a) $f(x, y, z) = \mathbb{1}[z = Concat(x, y)]$ (b) $f(x, y) = \mathbb{1}[y = ReplaceChar(x, \text{“a”}, \text{“b”})]$

(c) $f(x, y) = \mathbb{1}[y = ToUpper(x)]$ (d) $f(x, a, b, y) = \mathbb{1}[y = Replace(x, a, b)]$

Figure 4: Examples of string operations represented as transducers. $\mathcal{E}$ stands for a tuple of an appropriate size that consists of ε-characters only.



(a) Built using summation and scaling operations. (b) Built manually.

Figure 5: A substring transducer with substring length summed out.

We have now shown that all of our required string operations can be represented as transducers. Of course, some models may require other string operations, such as parsing arbitrarily nested brackets, which are not representable as a transducer. If the messages for such operations can be computed by some plug-in algorithm than can consume and return automata, it would still be possible to apply BP to the model. Transducers just provide an easy way to do this for a broad class of operations.

4 Examples

In this section we show two examples of probabilistic models with mixed string and non-string variables that can be handled by the proposed method. To run inference we employ the Infer.NET inference engine [2], with our custom extensions installed (these may be downloaded from *link removed for blind review*). Fig. 6 shows the Infer.NET code for each model.

Person name clustering: As a first example, we address a simplified form of a problem that arises in information extraction (see, for instance, [16]): given a set of texts that refer to a number of unknown persons in different ways, infer which text refers to which person and attributes (the first, middle and last names) of each person mentioned. The model (Fig. 6a) assumes that each text refers to exactly one person and that each person can be referred to in three different ways: by first and last name only, or by first, middle and last name or with the middle name represented as an initial. It additionally assumes that names in texts are preceded and succeeded by either non-word characters like space, comma or period, or string boundaries. In this example model the number of persons is assumed to be known, but it is also possible to use a more complex model to learn this number.

In this program `StrCapitalized` is an Infer.NET extension that we have added which returns a random variable having a uniform distribution over all strings that consist of an uppercase letter followed by one or more lowercase letters (as in Fig. 3c). Other extensions used are `Sub` which implements the *Substring* operation and `WordPrefix`/`WordSuffix` that define a variable having a uniform distribution over either an empty string or any string ending/starting with a non-word character. We also add an overload of the plus operator that implements the *Concat* operation. The program specifies that the distribution over a text is defined by the following procedure: pick

```
var p = new Range(personCount);
var first = Array<string>(p);
var middle = Array<string>(p);
var last = Array<string>(p);
using (ForEach(p)) {
  first[p] = StrCapitalized();
  middle[p] = StrCapitalized();
  last[p] = StrCapitalized();
}
var t = new Range(textCount);
var texts = Array<string>(t);
using (ForEach(t)) {
  var pt = DiscreteUniform(p);
  using (Switch(pt)) {
    var n = New<string>();
    var f = DiscreteUniform(3);
    using (Case(f, 0))
     { n.SetTo(first[pt]+" "+last[pt]); }
    using (Case(f, 1))
     { n.SetTo(first[pt]+" "+
       Sub(middle[pt],0,1)+". "+last[pt]); }
    using (Case(f, 2))
     { n.SetTo(first[pt]+" "+middle[pt]
       +" "+last[pt]); }
    texts[t] = WordPrefix()+n+WordSuffix();
  }
}
```

(a) Person name clustering.

```
var c = new Range(motifLen);
var motifProbs = Array<Vector>(c);
motifProbs[c] = Dirichlet(motifProbPrior);
var s = new Range(sequenceCount);
var motifPos = Array<int>(s);
var hasMotif = Array<bool>(s);
var sequences = Array<string>(s);
using (ForEach(s)) {
  motifPos[s] = DiscreteUniform(
    seqLen - motifLen + 1);
  hasMotif[s] = Bernoulli(hasMotifProb);
  using (If(hasMotif[s])) {
    var motifChars = Array<char>(c);
    motifChars[c] = Char(motifProbs[c]);
    var motif = StrFromArray(motifChars);
    var bgL = StrOfLen(motifPos[s],bgDist);
    var bgR = StrOfLen(
      seqLen - motifLen - motifPos[s],
      bgDist);
    sequences[s] = bgL + motif + bgR;
  }
  using (IfNot(hasMotif[s])) {
    sequences[s] = StrOfLen(
      seqLength, bgDist);
  }
}
```

(b) A motif finder.

Figure 6: Infer.NET programs for two example hybrid models that contain both string and non-string variables. For details of syntax see [2] – note that the `Variable` class has been imported.

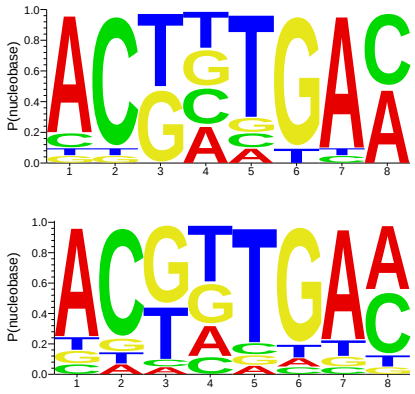
a random person, pick a random name format, format person's name according to the format and generate the text by pre-pending and appending random strings to the name.

Results of inference on an example text dataset are shown in Table 1. In many texts, names cannot be uniquely distinguished from surrounding capitalized words – it is the simultaneous parsing of multiple texts that resolves this ambiguity. Note that the results reflect the fact that we have observed only the first letter of one of the middle names. It should be noted that the model we use is symmetric over person index permutations, so true name marginals would be an average across all people – we used symmetry breaking to avoid this and obtain the results shown.

Motif finder: Our second example model (Fig. 6b) implements a basic motif finder. In genetics, a sequence motif is a commonly occurring stochastic pattern in nucleotide or amino-acid sequences that is likely to have some biological significance. The problem of motif finding is to discover the pattern given a set of sequences. The model makes the following assumptions: the motif length and the probability that a sequence contains the motif are known in advance, all sequences are of the same length, non-motif sequence parts are drawn from the supplied `bgDist` distribution independently at every position. We additionally assume that nucleobase occurences at different positions of the motif are independent. In the program `StrOfLen` is our extension that defines a string random variable of given length and having a given character distribution at every position, and `StrFromArray` is a utility extension that makes a string random variable from an array of

Table 1: Inference results for the name clustering example

Data	Variable	Inferred marginal
"Peter James Price was mentioned in a newspaper."	$first_1$	$\mathbb{1}[x =$ "Peter"$]$
	$middle_1$	$\mathbb{1}[x =$ "James"$]$
"Charlotte N. Pope: The Story of My Life"	$last_1$	$\mathbb{1}[x =$ "Price"$]$
"Charlotte Pope was born on..."	$first_2$	$\mathbb{1}[x =$ "Charlotte"$]$
"The Mysterious Peter Price: a Short Bio.'	$middle_2$	$\frac{1}{Z}\mathbb{1}[x$ starts with "N"$]$
"Peter James Price — who is he?"	$last_2$	$\mathbb{1}[x =$ "Pope"$]$



(a) True (top) and inferred (bottom) motif probabilities shown as sequence logos.

	$Phas$	$Pmode$	$Ptruth$
AGTTTGCGTTGAACCGCGTGATATA	0.95	0.95	
CCTGGGGGCCGCAATACGGCTGAAC	0.32	0.35	
CTACTTTGACCATGCAACACTCAGG	0.99	0.99	
GTTGGTCATAATGGAACTTGGTCGG	0.62	0.64	
GATTAGAAATTATCAACCCTCTGTT	0.22		
AGCCGTTGTTGTATTTCGAGGTCGC	0.29		
GCTGTTCGGTTACGGTGAAATCACA	0.99	0.99	
TTAGGACAGTCGTTTGTTAGTCGCG	0.07		
GGAAGTTTGATGTAGCTACGGTGAA	0.98	0.97	
AGCGAACTCGACGGTGACGTGTTAC	0.99	0.99	
GCAAGTACTCCGGCGCATATAAGCA	0.05		
CTGGGTACTGCTTTCTTCGTTGACG	0.96	0.96	
TCGGAATTCCTGTGATGCGGTTGAA	0.63	0.47	0.20
TTCCGATCACAATAATACTTTGAAG	0.99	0.99	

(b) Sample results on individual sequences. Ground truth is red, predictions are orange, overlap is green. $Phas$ is the posterior probability of the motif being present on a sequence. $Pmode$ is the probability of the predicted position, and $Ptruth$ is the probability of the ground truth position under the position posterior, if different from the prediction.

Figure 7: Inference results for the motif finder example. (a) Inferred motif probabilities are very close to the truth (b) as are inferred motif locations.

random characters. Notice that the model makes heavy use of existing Infer.NET functions including non-string variables, in combination with the new string operations.

To test inference in the model, we sampled a set of 50 synthetic sequences of length 25 from the model, 80% of which contain a motif of length 8. We then inferred motif character probabilities (Fig. 7a) and motif positions (Fig. 7b). It can be seen that the inferred motif nucleobase distributions are very close to the ground truth. In most sequences, the inferred most-probable location is correct – where it is not the true location still has reasonably high probability indicating genuine ambiguity (likely caused by an unusual ground truth motif).

5 Discussion and future work

We have shown that weighted finite state transducers can be used to perform belief propagation in models with string variables. Whilst we've shown relatively small examples in this paper, this approach has been used as part of a larger system processing hundreds of thousands of lines of text. In general, we have found the approach can scale well to very large models. However, some queries/models can lead to messages growing very large, even for small models, which substantially slows down inference. This happens either because the distribution being represented becomes very complex or because it is being represented inefficiently. In the latter case, automaton simplification algorithms can be applied to reduce the size of the message without changing the function that it represents. We found that a straightforward simplification heuristic, which finds a part of an automaton that is a tree and replaces it with an equivalent trie, was sufficient for our purposes. We are also keen to investigate more sophisticated techniques, like bisimulation [17]. When the message is inherently complex, however, automaton simplification will not help. In this case, it may be helpful to instead use expectation propagation (EP) [4]. In EP, messages outgoing from a factor are projected onto a particular distribution family, which in our case could be automata of bounded size (say, with no more than 500 states). Unfortunately, finding a way to do this remains an open research problem. However, we have found one way to improve the speed of our system, inspired by EP, even without projection. In EP, when sending a message from a factor to a variable, the reverse message on the same edge is multiplied into the factor function before projection and then divided out. If we do the same *without* projection, this can substantially reduce the complexity of the computed message (and the computation time) because any strings which have zero value in the reverse message can be pruned away.

This work suggests that it would be valuable to explore message passing algorithms for other types of variables. To this end, we are keen to explore inference algorithms for types such as dates, collections or compound objects with properties.

References

[1] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[2] Thomas Minka, John Winn, John Guiver, and David Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[3] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[4] Thomas Minka. *Expectation Propagation for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, 2001.

[5] J. Winn and C. M. Bishop. Variational Message Passing. *JMLR*, 6:661–694, 2005.

[6] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

[7] I. Holmes. Using guide trees to construct multiple-sequence evolutionary hmms. *Bioinformatics*, 19(suppl 1):i147–i157, 2003.

[8] Robert K Bradley and Ian Holmes. Transducers: an emerging probabilistic framework for modeling indels on trees. *Bioinformatics*, 23(23), 2007.

[9] Shankar Kumar and William Byrne. A weighted finite state transducer implementation of the alignment template model for statistical machine translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 63–70. Association for Computational Linguistics, 2003.

[10] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[11] Tom Minka and John Winn. Gates. In *Advances in Neural Information Processing Systems*, pages 1073–1080, 2008.

[12] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

[13] David Poole and Nevin Lianwen Zhang. Exploiting contextual independence in probabilistic inference. *J. Artif. Intell. Res.(JAIR)*, 18:263–313, 2003.

[14] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[15] Mehryar Mohri. Weighted automata algorithms. In *Handbook of weighted automata*, pages 213–254. Springer, 2009.

[16] Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart Russell, and Ilya Shpitser. Identity uncertainty and citation matching. *Advances in neural information processing systems*, pages 1425–1432, 2003.

[17] Peter Buchholz. Bisimulation relations for weighted automata. *Theoretical Computer Science*, 393(1):109–123, 2008.