Reanalyzing the Most Probable Sentence Problem: A Case Study in Explicating the Role of Entropy in Algorithmic Complexity

Eric Corlett **Gerald Penn**
Department of Computer Science
University of Toronto
{ecorlett,gpenn}@cs.toronto.edu

Abstract

When working with problems in natural language processing, we can find ourselves in situations where the traditional measurements of descriptive complexity are ineffective at describing the behaviour of our algorithms. It is easy to see why — the models we use are often general frameworks into which difficult-to-define tasks can be embedded. These frameworks can have more power than we typically use, and so complexity measures such as worst-case running time can drastically overestimate the cost of running our algorithms. In particular, they can make an apparently tractable problem seem NP-complete. Using empirical studies to evaluate performance is a necessary but incomplete method of dealing with this mismatch, since these studies no longer act as a guarantee of good performance. In this paper we use statistical measures such as entropy to give an updated analysis of the complexity of the NP-complete Most Probable Sentence problem for pCFGs, which can then be applied to word sense disambiguation and inference tasks. We can bound both the running time and the error in a simple search algorithm, allowing for a much faster search than the NP-completeness of this problem would suggest.

1 Introduction

Natural Language Processing uses many algorithms that are theoretically intractable, but work well in practice. The k-means clustering algorithm, for example, has an exponential worst-case time, but is generally polynomial in practical applications (Arthur et al., 2009). If we look at problems, as opposed to algorithms, the task of training neural networks (Blum and Rivest, 1993), sentence disambiguation in pCFGs and HMMs (Sima'an, 2002), and solving cryptograms (Nuhn and Ney, 2013) have even been shown to be NP-complete. These results would seem to be at odds with the observation that we are able to actually perform these tasks. Clearly, while NP-completeness results are a necessary part of our understanding, they do not tell the whole story about our algorithms' behaviour.

Faced with the inadequacy of existing descriptive complexity measures to represent program operation, some researchers have resorted to empirical measures of performance (e.g. Carroll, 1994). Unfortunately, empirical studies have their own pitfalls, and these are also difficult to address. Specifically, empirical studies of program performance have an implicit dependence on the distribution of inputs used in the study. Changing this distribution of inputs can change the overall performance drastically. This, after all, is why we separate training and evaluation test sets.

Ultimately our analytic tools should work together with our empirical studies, in the sense that we should be able to give quantify how much program performance changes as we change our input distributions. This would allow us to circumscribe our guarantees of program performance; if those bounds are not met (e.g., if a program takes much longer than expected to run), we have evidence that the set of inputs did not comply with the distribution in the first place.

The situation of NLP is a little more complicated, however. Many of our algorithms include, implicitly or explicitly, probability distributions as part of their input. Take for example a natural language parser. Thirty years ago, these relied upon lexicalized pCFGs that explicitly assigned probabilities to different rules and word n-grams, whereas today, the distributions used by neural parsers avail themselves of distributions that are rather more implicit. But in both cases, the distributions are acquired by sampling training data.

The aim of our research programme is to incorporate these distributions into our theoretical anal-

Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics, pages 3354–3362
April 19 - 23, 2021. ©2021 Association for Computational Linguistics

yses of program performance. If we were to follow existing analyses of a program's dependence on textual input, we may choose to characterize that dependency as a function of the length of the distribution's description. Uniform distributions, arguably the shortest to write, often ensure the worst program performance. Indeed, there is plenty of evidence from statistics, information theory and other AI applications that a particular derivative characteristic of distributions which is systematically related to the length of their descriptions captures many aspects of this dependency: *entropy*.

In this paper we take steps towards this goal by demonstrating how to link the run time of a specific inference problem, the Most Probable Sentence (MPS) problem for pCFGs (Sima'an, 2002), to the entropy of its input. While this is somewhat of a niche problem, it does provide a good demonstration of our approach. We will describe below in Section 8 some other graphical models that this approach can be readily applied to.

We start our analysis of the MPS problem in Section 3 by outlining the problem definition, the NP-hardness proof of the problem, and the main bound we are going to prove about it. In Sections 4 and 5, we develop an analysis of the MPS problem by showing how entropy, specifically, the conditional entropy of a parse tree given the words of a sentence, changes the running time of a simple search algorithm for the most probable tag sequence over that sentence. In Section 6, we show that there is an inherent trade-off between accuracy and run time whose worst case is dictated by the pCFG sentence-to-tree entropy. Finally, in Section 7, we characterize the run time of our search when we hold our grammar constant and draw many sentences from it.

2 Previous Work

Historically, most complexity results in computer science have dealt with worst-case complexity. A major NLP result in this vein has been the study by Sima'an (2002), which shows that the most probable parse problem (MPP) for pSTSGs and the most probable sentence problem (MPS) for both pSTSGs and pCFGs are all NP-complete. These problems can be thought of as renormalization over a graphical model, and have been studied further in several publications, including De la Higuera and Oncina (2013) and Goodman (1998). Similar NP-completeness results exist for the problem of finding the optimal word order for phrases in machine translation (Germann et al., 2001) and for solving letter-substitution problems such as cryptograms (Nuhn and Ney, 2013).

An argument could be made for average-case complexity (Levin, 1986; Impagliazzo, 1995) as an alternative to what we are attempting here. Average-case complexity is a step towards the type of results that we want, but it still has its shortcomings. In particular, an average case complexity analysis relies on the underlying distribution of inputs being known (and, usually, easy to work with) in advance. The distribution of choice is often naïvely uniform. In situations where the input distribution is defined by the problem task, or where it is actually defined as part of the input, the problem of how program behaviour changes with the distribution is not completely addressed. Average-case complexity, moreover, does not give any sort of guarantee on program performance.

Within artificial intelligence, approximation results are perhaps a more natural approach to analysis. Nearly all of these endeavour to establish a worst-case result, however, and in some cases, such as with Markov Logic Networks and Bayesian networks (Roth, 1996; Cooper, 1990), it is hard even to approximate for probabilistic inference. What we want is the worst case, but not the worst distribution. We want the worst case for a typical distribution.

A general framework for "probabilistic complexity" has been explored in Ackerman et al. (2011), in which it is found that the calculation of values such as conditional probability can be intractable, even on an otherwise tractable distribution. On the other hand, attempts to give decompositions for large families of distributions into tractable bases, as described in Erdélyi et al. (2009), can be used to characterize input probabilities for which algorithms will often be efficient. What these papers do not do is explore how the task-specific aspects of a problem in a powerful model can shift an algorithm's complexity into a hard or easy part of the input space.

A promising alternative approach to analyzing program complexity is to look at the smoothed complexity of an algorithm. In this paradigm, the complexity of an algorithm is not based on the worst input that an arbitrary adversary could choose for the problem. Instead, the adversary is constrained by having a small error added to their inputs. The

expected complexity of the resulting problem is then reported, and can often be much better than the usual worst case. This approach to complexity analysis is very effective for describing the behaviour of many otherwise exponential algorithms, such as the simplex algorithm (Spielman and Teng, 2004) or the k-means clustering algorithm (Arthur et al., 2009). The major limitation of this approach for us, however, is that the improvement in performance is assumed to be due to measurement error in the input. Such an assumption is very appropriate when talking about, for example, measurements taken of continuous real-world quantities like length. It is not likely to be appropriate for situations where the program improvement is due to observable regularities in the input data.

What we really want to do is to exploit the fact that we see only a fragment of the theoretically possible input space in the real world. Unfortunately, the task of identifying good problem fragments for our models is itself very hard, and so we will instead attempt to use statistical measures to build tools for narrowing down the space in which our inputs will lie. A start towards this goal is found for the letter-substitution problem in (Corlett and Penn, 2013), but this approach does not give a tight enough bound to describe when we should expect better running times, and it does not immediately generalize to other tasks. We would like to strengthen the approach used in that paper to make it more applicable to different areas. With this in mind, we turn to another NP-complete problem, the MPS problem for pCFGs.

3 Tagging and NP-Completeness

To start, we recall the definition of the problem:

Most Probable Sentence Problem (MPS)

Instance: A pCFG model G and a string s.
Question: Find a sequence of part-of-speech tags, σ, that maximizes the sum of probabilities of trees with yield s and pre-terminals σ.

As an example, suppose we were looking at the sentence "Time flies like an arrow". There are several assignable tag sequences, such as

$$N\ V\ Adv\ Det\ N$$

or

$$Adj\ N\ V\ Det\ N.$$

Each tag sequence might be obtainable from more than one parse tree. The probability of any such tag sequence is the sum of the probabilities of the trees yielding that sequence.

The difficulty of this problem lies in the fact that we are looking for the POS tags that maximize the sum of the probability of all trees using those tags. If we were looking for the single most likely tree, or if we were looking for the sum of all trees with yield s, without fixing the POS tags, the usual $\mathcal{O}(n^3)$ parsers would work.

A reduction, as described in Sima'an (2002) is made from 3-SAT as follows: given a 3-SAT formula ϕ with η clauses and κ variables, we build a pCFG grammar G_ϕ that generates strings that contain 3η copies of a single terminal, such as x. The tags for these terminals are either T or F – that is, they are the truth assignments for the literals of ϕ (e.g., if $\eta = 2$, we would generate six-tag sequences such as *TTFTTF*). The grammar G_ϕ is capable of generating two types of trees for every input string: one that generates true assignments that may or may not be consistent across variable instances, and one that generates consistent assignments for one designated variable. A string of tag assignments that has one tree of the first type and κ trees (one for each variable) of the second type will have a total probability that is higher than trees that lack this many, and so there is a threshold Q that separates input strings with true, consistent assignments from input strings without true, consistent assignments.

In the following sections, however, we will prove the following:

Theorem 3.1 *Suppose that we draw a sentence s from a pCFG grammar G. Then, the MPS problem can be solved in $\mathcal{O}(n^4[p_G(s)/p_G(\tau(s))])$ time, where:*

- *n is the length of s,*
- *$\tau(s)$ is the tag sequence assigned within the most likely tree, $t(s)$, that has yield s, and*
- *$p_G(s)$ is the probability of s in G, and $p_G(\tau(s))$ is its probability given $\tau(s)$.*

This is a major improvement over the bound of Sima'an (2002) because it shows that the exponential run times that we expect from an NP-complete problem are in fact bounded by the ratio $p_G(s)/p_G(\tau(s))$. For the pathological grammars that are used in the NP-completeness proof, this ratio grows exponentially with the size n of the sentence. On the other hand, we conjecture that it is

quite small for the natural language grammars that we see in practice. Empirically computing the average such ratio on a corpus can itself be very trying,[1] but there are good indications that it is relatively constrained. In the Penn Treebank, for example, 46.5% of the word tokens in the corpus have only one POS tag assignable, and these words are fairly well dispersed throughout the corpus: the median intrasentential distance between them (not including the single-tag words themselves) is 1, with an average of 4.4. In any case, a parser that avails itself of this theorem does not need to explicitly calculate $p_G(s)/p_G(\tau(s))$. $p_G(s)/p_G(t(s))$, which directly relates to the probability of the best tree for s, is likely to be a much looser bound. We will attempt to bound it theoretically in Section 7.

Proof of Theorem 3.1: This is an immediate consequence of Lemmata 5.5 and 5.6. The proofs are presented below. Section 4 describes a simple search algorithm that will be the focus of the discussion. Section 5 will analyze the running time of this algorithm in a way that provides the basis for these lemmata. In Section 6, we will extend this result to describe the error incurred if we allow early stopping. ■

4 Analysis of Problem and Exemplary Algorithm

We will use a similar approach to that used in (Corlett and Penn, 2013), in that we will use an A^* search to solve the MPS problem for pCFGs. The idea behind the search is as follows: given a pCFG G and a string s of n words, we can find the overall probability $p_G(s)$ in G through a cubic time algorithm that algebraically mirrors a conventional all-paths parsing algorithm. But for our purposes, we can do it differently: given any instance w of a word in s, we can make a guess as to the POS tag r that w will take, and then run the algorithm, but with the restriction that only the tag r will be counted for w. In our earlier example of "Time flies like an arrow," perhaps this would mean running the parser given the restriction that "Time" is an adjective, while letting the other words take any tag that they can. Clearly, this run of the parser will still take $\mathcal{O}(n^3)$ time, as it would if we simultaneously constrained any subsequence of the words of s rather than just one.

Recall that $p_G(s)$ is the sum of all $p_G(t)$, where t ranges over all trees with the yield of s. Furthermore, if, for a subsequence of word instances $(w_{u_1}, w_{u_2}, \ldots, w_{u_J})$, we restrict the parser so that only the POS tags $(r_1, r_2, \ldots, r_J)$ will be considered, the probability calculated will be the sum of $p_G(t')$, where t' ranges over all trees with both have the yield of s and which have the desired POS tag restrictions. We will use this probability as an admissible heuristic in our A^* priority queue, and consider possible sequences of tag assignments as nodes in the search space. We will refer to sequences of tag assignments with the character a, and we will index them by the letter i. Sequences of tag assignments will not, in general, fix a tag for every word in s.

To run the search, we fix an order for the words in s. We assume a last-to-first ordering here, but any ordering will suffice. We push an empty assignment $\{\}$ with the score $p_G(s)$ onto a probability queue Q. While Q is nonempty, we pop its maximum element a_i and look for the first word w_u in our ordering of s which is not fixed by a_i. For every possible POS tag r_j for w_u, we add the POS tag assignment $\{w_u : r_j\}$ to a_i to get a new sequence of tag assignments $a_i^{[j]}$, compute $p_G(s)$ restricted to $a_i^{[j]}$, and insert $a_i^{[j]}$ into Q with the resulting probability. We return the first tag assignment a popped from Q that fixes every word in s.

To see that this A^* search is correct, we first note that it must terminate: every time a tag assignment A is popped from the queue, either a fixes every word in s and the program terminates, or a series of strictly longer tag assignments a' is added to the queue.

Furthermore, since our heuristic for a tag assignment sequence a is the sum of the probabilities of all suitably restricted trees t in G with yields of s, we can see that when we extend a tag assignment sequence a, we are simply adding new constraints to the contributing trees t, and so the set of trees counted is non-increasing. This means that the heuristic is also non-increasing. The same argument tells us that any extension of a tag assignment sequence a that fixes every word in s must have a probability that is at most that of s given a. These arguments, taken together, indicate that the first tag sequence found that fixes every word in s will be the one that gives the maximum probability, and so this assignment will give the solution to the MPS problem.

[1] The reader may wish to refer to our work on this subject at https://doi.org/10.5683/SP2/CM9QY1.

5 Initial Analysis

In order to find the time complexity of this algorithm, recall the fact that the admissible heuristic is non-increasing. If we apply this to the first and last tag assignment sequences popped from the stack, we can see:

Lemma 5.1 *If $M_G(s)$ is the probability of s given its most likely tag sequence, and if a is any tag sequence expanded in the search, then*

$$p_G(s) \geq p_G(s|a) \geq M_G(s).$$

Similarly, if a is a tag sequence that is seen but not expanded in the search,

$$M_G(s) \geq p_G(s|a).$$

So understanding the algorithm complexity becomes an issue of determining how many a are in this range. Furthermore:

Lemma 5.2 *If we extend a specific tag sequence a_i in our search by one tag assignment into the new sequences $a_i^{[1]}, a_i^{[2]}, \ldots, a_i^{[m]}$, then*

$$p_G(s|a_i) = \sum_j^m p_G(s|a_i^{[j]}).$$

This just means that in our example sentence s =“Time flies like an arrow,” if we were to argue that “flies” can only be an *N* or a *V*, then the probability of s is the sum of the probability of s given that “flies” is a *N* plus the probability of s given that “flies” is a *V*. Any of the parse trees that are compatible with one restriction are incompatible with the other, and so the outcomes in their probabilities are disjoint.

In fact, since Lemma 5.2 applies to every tag sequence we expand, we can go so far as to apply it iteratively to cover the whole search space:

Corollary 5.2.1 *Let QO be the selected optimal solution together with the items left stranded on Q at the end of the search. Then:*

$$\sum_{a \in QO} p_G(s|a) = p_G(s).$$

Let D^* be the set of all a_i such that every expansion of a_i, $a_i^{[1]}, a_i^{[2]}, \ldots, a_i^{[m]}$ is a member of QO. Some of the members of QO will be complete sequences of tag assignments — every word in s has received a tag in these. Others may not be complete, because the search terminated before they could be expanded. Because of this, D^* is not merely the set of QO parent nodes in the search space — some parents will be left out because they were extended both to assignment sequences that were further extended, as well as to assignment sequences that were not further extended.

Let D' be the subset of QO consisting of every node with its parent in D^*. Every element of QO only has one parent node (no joins in the search space) because we always marshal out the tags of s in the order that we defined over its words. By Corollary 5.2.1:

$$\begin{aligned} p_G(s) &= \sum_{a \in QO} p_G(s|a) \\ &= \sum_{a \in D'} p_G(s|a) + \sum_{a \in QO \setminus D'} p_G(s|a) \\ &\geq \sum_{a \in D'} p_G(s|a). \end{aligned}$$

Then, by repeated applications of Lemma 5.2 to the elements of D^*:

Lemma 5.3

$$p_G(s) \geq \sum_{a \in D'} p_G(s|a) = \sum_{a \in D^*} p_G(s|a).$$

Interestingly, the nodes of D^* are the nodes that are the most complete assignment sequences that still satisfy $p_G(s|a) \geq M_G(s)$, from the first clause of Lemma 5.1 — every node expanded in the search is either in D^* or is an ancestor of a node in D^*. It takes at most $n-1$ steps of expansion to arrive at this frontier, because every step assigns a tag to a new word of s, and there are at most $|D^*|$ such paths through the search space. So the total number of nodes expanded must be at most $\mathcal{O}(n|D^*|)$.

Lemma 5.4 *Let $R_G(s) = p_G(s)/M_G(s)$. The total number of nodes expanded must be at most $\mathcal{O}(nR_G(s))$.*

Proof: We have just learned that for every $a \in D^*$, $p_G(s|a) \geq M_G(s)$. Therefore, by Lemma 5.3, $p_G(s) \geq \sum_{a \in D^*} M_G(s)$, and so $R_G(s) \geq \sum_{a \in D^*} 1 = |D^*|$. ■

Finally, note that every one-word expansion in our search procedure comes with a decoding step that is bounded by $\mathcal{O}(n^3)$ time. Thus we have:

Lemma 5.5 *The running time for the entire search is $\mathcal{O}(R_G(s)n^4)$.*

The proof by Sima'an (2002) of overall NP-completeness implies that $R_G(s)$ can be exponentially large in n. His reduction exhibits a grammar which generates a single s with probability one, while the value of $M_G(s)$ is proportional to a threshold value, which is exponentially small in n. So in this degenerate case, the ratio $R_G(s)$ will always be exponentially large in the size of the grammar. In any case, determining $R_G(s)$ is the key to understanding the running time of our search.

Finding $R_G(s)$ is as difficult as finding $M_G(s)$, since we can use one to calculate the other. So we cannot directly use $R_G(s)$ in our calculations — we need a bound for it instead. Clearly, given a positive lower bound for $M_G(s)$, L, then $R_G(s) \leq p_G(s)/L$ also.

A very simple lower bound L is the most probable single parse tree $t(s)$ for s, which can be calculated in $\mathcal{O}(n^3)$ time. Clearly, the probability of $t(s)$ is at most the probability of all trees with the same tag assignment as $t(s)$, and this sum is itself at most the probability of the largest full tag assignment. Thus:

Lemma 5.6 *For any sentence s,*

$$R_G(s) \leq p_G(s)/p_G(\tau(s)) \leq p_G(s)/p_G(t(s)).$$

In the case of pure pCFG parsing, finding $p_G(s)$ also admits an $\mathcal{O}(n^3)$ algorithm, although the implicit sum over all possible trees includes a potentially infinite series over cycles of unary phrase structure rules. When this series converges, finding the probability mass added by these extra trees involves inverting a potentially large matrix that can be prohibitively expensive, although this cost is independent of the input length, and can be precomputed offline.

6 A Trade-off Between Accuracy and Time

Practically, we may not want to run a full A^* search, especially if the overall search takes a long time. The above analysis is a convenient starting point for investigating the consequences of stopping our search early.

If we do decide to stop the A^* search early, we will want to output a tag sequence. Let this tag sequence be $\hat{a}(s)$. We will assume here that it is $\tau(s)$, the one associated with the most likely tree, $t(s)$, which means that it can be calculated easily, and can easily be combined with any other estimate (just check both tag sequence estimates and take the better).

The heuristic score of $\hat{a}(s)$ provides a lower bound on the probability of the most likely tag sequence, which may in fact not be $\hat{a}(s)$. As the search progresses, those scores will decrease, and so the range of values that the maximum probability can take on will also decrease. In particular, the error that we incur by simply choosing the tag sequence $\hat{a}(s)$ as our output will always be at most $p_G(a_i)/p_G(\hat{a}(s))$. Given some $k \leq n$, let us run our A^* search for $n2^k$ iterations. We assert that:

Theorem 6.1 *If we run our search for $n2^k$ iterations, we will get a reduction in error of at least k bits, i.e., we achieve an overall reduction in error of at least 2^{-k}.*

Proof: Suppose otherwise. Then, since $p_G(s|a_i)$ is the score of the latest partial solution on the queue after $n2^k$ iterations, we have that $p_G(a^*) < p_G(s|a_i)$, where a^* is an optimal tag assignment. But, since the error that we incur by choosing $\hat{a}_t$ is not reduced by at least 2^{-k}, then $p_G(s|a_i)/p_G(\hat{a}(s)) > 2^{-k}(p_G(s)/p_G(\hat{a}(s)))$.

Let Q_i be the partial tag sequence a_i together with the items left stranded on Q when we terminate the search. Then, as we saw in Corollary 5.2.1:

$$\sum_{a \in Q_i} p_G(s|a) = p_G(s).$$

Furthermore, let D_i and D'_i be defined analogously to D^* and D' from Section 5. The arguments of Lemma 5.3 still apply, so that

$$\sum_{a \in D_i} p_G(s|a) \leq p_G(s).$$

Since $p_G(s|a_i) \leq p_G(s|a)$ for every $a \in D_i$, it follows that:

$$\begin{aligned} &\sum_{a \in D_i} p_G(a) = p_G(s) \\ \Rightarrow &\sum_{a \in D_i} (p_G(s|a_i)) \leq p_G(s) \\ \Rightarrow &\sum_{a \in D_i} 1 \leq p_G(s)/(p_G(s|a_i)), \end{aligned}$$

in which case:

$$\begin{aligned}|D_i| &\leq p_G(s)/(p_G(s|a_i)) \\ &= \Big(\frac{p_G(s)}{p_G(\hat{a}(s))}\Big)/\Big(\frac{(p_G(s|a_i))}{p_G(\hat{a}(s))}\Big) \\ &< \Big(\frac{p_G(s)}{p_G(\hat{a}(s))}\Big)/\Big(2^{-k}\big(\frac{p_G(s)}{p_G(\hat{a}(s))}\big)\Big) \\ &= 2^k.\end{aligned}$$

At this point, the number of nodes expanded by the algorithm is at most the number of unexpanded nodes in the search space times the number of their ancestors. where the number of ancestors of any node is at most n (since this is the maximum depth in the tree). On analogy to Section 5, the total number of nodes that have been expanded in the search is less than $n2^k$. But we have stated that the algorithm has been run at least $n2^k$ times. Therefore, we have an error reduction of at least 2^{-k}, as desired. ■

If the algorithm halts in this time, we have an exact solution, and so our error rate is zero. So if we allow early stopping in our search we can guarantee that the error times the number of times the search iterates is $\mathcal{O}(n(p_G(s)/p_G(\hat{a}(s))))$. Due to the cubic run time of the parsing algorithm, we find:

Lemma 6.2 *If we allow early stopping as described above, the total run time multiplied by the algorithm error will be* $\mathcal{O}((p_G(s)/p_G(\hat{a}(s))n^4))$.

As in Section 5, we can see that what we need to find is a bound on $p_G(s)/p_G(t(s))$.

7 Bounding $p_G(s)/p_G(t(s))$

Here we will find a bound on $p_G(s)/p_G(t(s))$ given only the grammar G. We can consider G as a model that probabilistically generates trees t, which in turn generate the yield s.

We know $p_G(t) \leq p_G(t(s))$, where $t(s)$ is the most likely tree that has the yield s. So, by the previous arguments, $p_G(t)$ is a lower bound for $M_G(s)$, and so $p_G(s)/p_G(t(s)) \leq p_G(s)/p_G(t)$. We will relate these values to the per-word sentence and tree entropies $H_{G,n}(s)$ and $H_{G,n}(t)$, where n is fixed.

If the the generative process for trees in G is ergodic and stationary, then as the size of the tree t increases, the density of the nonterminals in t approaches some distribution π_t. Furthermore, as the density of the words in s approach some distribution π_s with high probability, then for any $\delta, \varepsilon > 0$, there is an $N > 0$ such that if $n > N$, then, with probability at least $1 - \delta$, $\log p_G(s) < -n(H_{G,n}(s) - \varepsilon/2)$ and $\log p_G(t) > -n(H_{G,n}(t) + \varepsilon/2)$. In this case:

$$\begin{aligned}p_G(s)/p_G(t(s)) &\leq p_G(s)/p_G(t) \\ &= 2^{\log(p_G(s)) - \log(p_G(t))} \\ &< 2^{-n(H_{G,n}(s)-\varepsilon/2)+n(H_{G,n}(t)+\varepsilon/2)} \\ &= 2^{n(H_{G,n}(t)-H_{G,n}(s)+\varepsilon)}.\end{aligned}$$

Since s is completely determined by the tree t, we have that $H_G(s) = H_G(s,t)$, and so:

$$\begin{aligned}2^{n(H_{G,n}(t)-H_{G,n}(s)+\varepsilon)} &= 2^{n(H_{G,n}(t)-H_{G,n}(s,t)+\varepsilon)} \\ &= 2^{n(H_{G,n}(t|s)+\varepsilon)}\end{aligned}$$

Thus:

Theorem 7.1 *If the grammar G, taken as a generative process, is ergodic and stationary, then for any $\varepsilon, \delta > 0$ there is a number n such that, for any tree with at least n leaves:*

$$\log\left(p_G(s)/p_G(t(s))\right) \leq n(H_{G,n}(t|s) + \varepsilon)$$

with probability greater than $1 - \delta$.

Even if the grammar is not stationary and ergodic, we can ask what happens when we hold the grammar fixed and draw sentences from it, as is the case in typical NLP applications. In this case, the expected value of $log(p_G(s)/p_G(t))$ is still the conditional entropy $H_G(t|s)$ – we just cannot fix the sentence size. This value is not guaranteed to be an upper bound for the running time of the algorithm, or even for the logarithm of the expected value of $(p_G(s)/p_G(t))$.

Instead, we can look at the second moment $\nu_G(t|s)$ of $log(p_G(s)/p_G(t))$ in order to bound the overall likelihood of encountering a high-running-time input: For any $\delta > 0$, we can find a c such that the probability of drawing an s such that $log(p_G(s)/p_G(t)) > H_G(t|s) + c\nu_G(t|s)$ is less than δ. Thus:

Theorem 7.2 *Suppose we hold G constant and draw trees t from it, and that s is the yield of t. Then, for any $\delta > 0$ there is a c such that:*

$$p_G(s)/p_G(t(s)) < 2^{H_G(t|s)+c\nu_G(t|s)}$$

with probability greater than $1 - \delta$.

This bound unfortunately does not tell us how the problem difficulty scales with the sentence length n, but it does give us a sense of how difficult the MPS problem will be for a given grammar. While it is true that finding the exact values for either $H_G(t|s)$ or $\nu_G(t|s)$ is undecidable for general pCFGs, these values are can be given rough bounds – for example $H_G(t|s)$ is bounded above by the tree entropy $H_{G,n}(t)$. Finding better bounds is clearly an area for future research. Further discussion of the variance of the tree entropy can be found in (Chi, 1999).

8 Other Applications

In this paper, we have only considered the MPS problem, but the same approach can be applied to other inference problems, such as those found in Bayesian networks or CRFs. In particular, suppose that we are performing inference on a graphical model in which:

- we are trying to assign values to a finite set S of hidden variables in order to maximize the probability p of an event E, based on a finite set of hidden variables,
- we can build partial solutions to our inference problem by assigning values to one or more of these variables,
- we can use the probability of E over the partial solutions, or an approximation thereof, as an A^* heuristic on these partial solutions,
- extending nodes can always proceed according to an order defined as a function of the input (but independently of the node being expanded or its score), and
- extending nodes in the A^* search will partition its score among the child nodes.

If all of these conditions are met, then the A^* search above will still solve the problem, and will admit the same analysis. If the probability of E with no constraints is $p_0(E)$ and the probability of E under the optimal constraints is $p^*(E)$, the ratio $p_0(E)/P^*(E)$ will fill the same role as the ratio $p_G(s)/p_G(t)$ in our above analysis, and can be used to create a similar entropic bound. The main difficulty lies in the time we take to find the probability $p(E)$ given a set of constraints. This time will be application-dependent: in our case it took $\mathcal{O}(n^3)$ time. Our current method will generalize to any task in which the $p(E)$ value can be efficiently found.

9 Conclusions

Our analysis tells us something about how a particular NP-complete problem – the MPS problem for pCFGs – relates to a specific property of grammars. Our worst case running time is $\mathcal{O}(n^4 2^{n(H_{G,n}(t|s)+\varepsilon)})$ with high probability. Slightly more promising is that, when we draw a sentence s, it will take at most $2^{H_G(t|s)+c\nu_G(t|s)}n^4$ steps with probability $1-\delta$, where ν is the second moment of the entropy and c depends on δ. If we perform early stopping on the search, the product of the running time with the error will have the same bound. More practically, if we can determine that the ratio $p_G(s)/p_G(\tau(s))$ is small, our algorithm will have a much quicker run time.

Even in the worst-case scenario, the problem complexity collapses if we can break a sentence into several smaller instances. While we cannot do so in the grammar given by Sima'an (2002), it is possible in many NLP grammars. Roark and Hollingshead (2008) showed, for example, that we can find clause boundaries in sentences generated by the Penn Treebank grammar both quickly and reliably. Future work should focus on formalizing this for the MPS problem.

References

Nathanael L Ackerman, Cameron E Freer, and Daniel M Roy. 2011. Noncomputable conditional distributions. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 107–116. IEEE.

David Arthur, Bodo Manthey, and H Roglin. 2009. k-means has polynomial smoothed complexity. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 405–414. IEEE.

Avrim L Blum and Ronald L Rivest. 1993. Training a 3-node neural network is np-complete. In *Machine learning: From theory to applications*, pages 9–28. Springer.

John Carroll. 1994. Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 287–294. Association for Computational Linguistics.

Zhiyi Chi. 1999. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160.

Gregory F Cooper. 1990. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405.

Eric Corlett and Gerald Penn. 2013. Why letter substitution puzzles are not hard to solve: A case study in entropy and probabilistic search-complexity. In *The 13th Meeting on the Mathematics of Language*, page 83.

Gábor Erdélyi, Lane A Hemaspaandra, Jorg Rothe, and Holger Spakowski. 2009. Generalized juntas and np-hard sets. *Theoretical Computer Science*, 410(38):3995–4000.

Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. 2001. Fast decoding and optimal decoding for machine translation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 228–235. Association for Computational Linguistics.

Joshua Goodman. 1998. Parsing inside-out. *arXiv preprint cmp-lg/9805007*.

Colin De la Higuera and Jose Oncina. 2013. The most probable string: an algorithmic study. *Journal of Logic and Computation*, 24(2):311–330.

Russell Impagliazzo. 1995. A personal view of average-case complexity. In *Proceedings of Tenth Annual IEEE*, pages 134–147.

Leonid A Levin. 1986. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286.

Malte Nuhn and Hermann Ney. 2013. Decipherment complexity in 1: 1 substitution ciphers. In *ACL (1)*, pages 615–621.

Brian Roark and Kristy Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 745–752.

Dan Roth. 1996. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302.

Khalil Sima'an. 2002. Computational complexity of probabilistic disambiguation. *Grammars*, 5(2):125–151.

Daniel A Spielman and Shang-Hua Teng. 2004. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463.