Induction of random fields in NLP

Alexander Clark

August 25, 1998

Contents

List of Figures

Chapter 1

Introduction

It is now generally accepted that NLP systems that deal with unrestricted texts must have some statistical component. Frequently however the statistical component is not integrated into the structure of the model in any theoretically well grounded way, but is an *ad hoc* addition, tacked on to improve the performance when the original model has failed to behave as desired. In addition the statistical models used have often not been well-founded.

In this work I look at a particular class of statistical model, the random field, that has up to now been little studied in NLP but shows great promise. The two key papers on the application of random fields in NLP are 'Inducing features of random fields' by Steven Della Pietra, Vincent Della Pietra and John Lafferty([DPDPL97]), hereinafter DDL, that introduces some of the technical machinery in a comparatively simple application, and 'Stochastic Attribute-Value Grammars' by Steven Abney([Abn95]), that shows how they can be applied in a more linguistically complex area.

I show how random fields can be used to model linguistic data given a minimal, uncontroversial assumption of constituent structure, and how the features and parameters of the field can be induced from a corpus of parsed data. I examine some technical problems with the techniques and show how some of them can be solved. I conclude that random fields are potentially an extremely powerful tool, well-suited to NLP, but there are many technical problems involved, some of which may be insoluble.

In chapter 2 I introduce random fields in the context of statistical models in NLP in general, and compare them to a statistical model that may be familiar to some Sussex-based readers, the APRIL3 model. In chapter 3 I examine the work of DDL and Abney in some detail. In chapter 4 I introduce the domain in which I wish to apply this work. In chapter 5 I look at some of the algorithms I will use at a fairly abstract level. In chapter 6 I look in some detail at the use of Markov Chain Monte Carlo methods to estimate some functions over the field – this is the major technical difficulty with random fields, and forms a comparatively self-contained area. In chapter 7 I discuss the overall design of the program, and in chapter 8 I look at some of the more concrete issues. In chapter 9 I look at a sample run of the program and evaluate the success of it, and then finish with some concluding remarks in chapter 10, along with some speculations as to the future directions in which this work might go.

In the appendices there is first a glossary of some of the more technical notation, then a trace of one run of the program and finally the listings of the source code used.

Chapter 2

Random fields

2.1 Statistical Models in NLP

It is now widely recognised([Gaz96], [Abn96]) that real-world NLP systems will have to incorporate some form of statistical component to deal with various pervasive sources of ambiguity, such as prepositional phrase attachments in sentences like:

> I saw the man on the hill with the telescope.

coordination in sentences like:

> I like ham and cheese or tomato and pickle sandwiches.

and also to cope with the inevitable incompleteness of the lexicon, and of the grammar (if the the grammar is not completely lexicalised).

The range of statistical models in current use is not terribly wide. The most well studied are either various n-gram models (see for example Church's trigram tagger [Chu88]) or stochastic context-free grammars. The relationship between these two models is also well understood ([SS94]) but both of them have problems: they are both incapable of capturing the full range of statistical dependencies that are known to occur in natural language, and are difficult to extend.

Once a good statistical model has been constructed it can be used in a number of different ways – either to rank parses produced by some other method, or to guide a parser that works by some form of stochastic optimisation – an annealing parser[Sam96], for example. Given that we can easily observe contextual dependencies in corpora, it make sense to use a statistical model that is capable of picking up these influences. People have in general been very wary of using more sophisticated statistical models, because of a perception that the number of parameters that need to be estimated tends to explode exponentially with the depth of the dependencies. With random fields though, you can capture these dependencies quite parsimoniously.

These models suffer from a certain degree of arbitrariness: they assume that the relevant statistical properties are of a certain form – an assumption that we know to be false. There is however a principled set of models that make no such assumption; a set of models which can capture statistical regularities of any degree of complexity: *Maximum Entropy* models([BDPDP96]).

2.2 Maximum Entropy

The basic ideas of maximum entropy techniques are quite straightforward. The entropy measures the degree of order or information in a system. It is defined as

$$E = -\sum_x p(x) \log p(x) \tag{2.1}$$

where x is a random variable. It is important to note that the use of the log function here is not an arbitrary choice. The expression can be derived as being the unique expression that satisfies various constraints regarding the combination of entropies. The entropy measures the degree of disorder or information in a distribution. Distributions with higher entropy are more informative, or alternatively have fewer implicit assumptions. The distribution with maximum entropy is the one that makes no assumptions about it at all. In the case of a distribution over a finite set, the maximum entropy distribution is the uniform one, the one that divides the probability mass equally amongst all the options.

The distribution that has maximum entropy is then the one that is completely non-committal about the object under study. There are no hidden assumptions. One can also maximise entropy subject to certain constraints. Often one has a certain rather limited piece of information about a distribution. You may know for example that the expectation of a particular function over a distribution has a certain value[1]. Given this information, and nothing else, the principled way to model the distribution is to use the maximum entropy model subject to this constraint. In some circumstances this may not be possible because the information is insufficiently strong - in this case the maxent distribution diverges.[Jay83]

A good example of the power of maximum entropy techniques is to solve the following question (first discussed by Jaynes):

> Given a normal six-sided die, that when thrown does not give the expected average of 3.5, but rather of some higher figure say 4, what are the probabilities of the various numbers being thrown?

Traditional statistical methods have no answer for this: they say it is an ill-formed question. Maximum entropy gives a simple answer – maximise the entropy subject to the this constraint and you get the answer

$$\{0.103, 0.123, 0.146, 0.174, 0.207, 0.247\}$$

for the six probabilities.[2].

The maximum entropy modelling technique works thus: you identify some properties of the data that seem significant, and then model the data as the maximum entropy model subject to those constraints. These constraints normally take the form of saying that some property or *feature* has a certain expectation. The features can be of almost any sort; they can overlap or include each other, there can be execptions and so on. When you have a maximum entropy model that is defined over a graph[3], it is normally called a random field. Since linguistic structures have often been represented either by trees, or feature structures,

[1] The maximum entropy distribution given the expectation and the variance is the Gaussian (normal) distribution ([Bis95] page 243)

[2] see *http : //www.public.iastate.edu/ millers/Dice.html* for some further discussion

[3] A graph in the mathematical sense is a collection of node connected by arcs.

both of which can be represented as graphs, the appropriate statistical model for this field would appear to be some sort of random field.

2.3 Random fields

The Gibbs models used in this work are defined to be the maximum entropy models with certain features. Given the expectations of these features, these exponential models can be shown to have maximum entropy. For example, given a set of parse trees, we might observe that a particular configuration, say a node labelled `S` immediately dominating nodes labelled `N` and `V`, occurred with a particular frequency. If that is all the information we have about the distribution of the trees, then the correct model is a random field with that single feature and a weight that can be calculated from the expectation, as shown below.

The random fields considered here have the following structure: [4]

$$p(\omega) = \frac{1}{Z} e^{\sum_i \lambda_i f_i(\omega)} \tag{2.2}$$

In this definition ω is a particular configuration – in the application developed here this would be a labelled tree, f_i is a feature that is a function from configurations to real numbers, and λ_i is the corresponding weight, where i ranges over the finite set of features and weights. In the applications under consideration here, the feature functions take the values 1 or 0, corresponding to the presence or absence of a particular configuration. To be completely concrete, a feature might be the presence of a node labeled `S` that immediately dominates two nodes labelled `N` and `V`. Notionally we have a different feature at each point in the graph under consideration, but we *tie* the weights of each feature: that is we say the same features apply at each point in the graph (this property is *homogeneity*) and they all have the same value. Thus the cumulative value of the feature can be a non-negative integer corresponding to the number of times the feature occurs in the structure being considered. In the concrete case the feature would take the value zero if the tree in question contained no occurrences of a a node labeled `S` immediately dominating two nodes labelled `N` and `V`, 1 if there is 1 occurrence, 2 if it occurs twice and so on.

Note also that the λs can take negative values, in which case the presence of the configuration reduces the probability, positive values in which case it increases it, or zero values in which case the presence or absence has no effect – this is equivalent to the features not being present at all.

The normalisation factor Z, whose function is to make the probabilities sum to one, is unknown, and is in practice extremely difficult to calculate: it is the expectation of the field weights over the field. This must of course be finite: we will see later that this analytic requirement is not trivial.

Random fields can be considered to be a generalisation of Markov chains. They were initially investigated in the context of statistical physics. A key article that introduced these distributions to a wider audience was Geman's[GG84] that used them, and the Gibbs sampler in the field of image-processing. DDL prove that the maximum-likelihood distribution of the form of equation 2.2 and the maximum entropy distribution with the correct feature expectations are the

[4] These are technically homogeneous Gibbs exponential random fields.

same distribution. These random fields then are not merely 'a distribution with nice properties' but the unique distribution that satisfies two uncontroversially desirable properties.

This power does not come without a price: as shall be seen below it is often impossible to use it to calculate the exact probability of a structure: one will only know the 'improper' probability, that is the probability multiplied by some fixed, unknown constant. We can thus know the *ratio* of two probabilities precisely, and for many applications that is sufficient.

Relating this to other statistical models used in NLP is interesting. The stochastic context-free grammar, which is often estimated using the inside-outside algorithm is an oft-studied example[Cha93]. How does this relate to random fields? Given a parsed corpus or tree-bank we can set the weights of the rules to the observed expected value from the corpus – i.e. the number of local trees that instantiate that rule in the corpus, divided by the number of local trees dominated by the category on the left hand side of the rule. This has a number of undesirable properties stemming from the fact that this is not in general the best estimate of the expectation in the underlying distribution given small amounts of data. If you smooth the weights of the rules to take account of this, allowing local trees that do not appear in the corpus, then you end up with something not dissimilar to a random field where the features are the local trees that appear in the corpus.

2.4 The APRIL model

It is interesting to see how another statistical model of language compares to this, namely the model in APRIL3 presented in chapter 7 of [Sam96]. This model has been tuned to deal well with the idiosyncracies of real data – theoretical purity has not been a major goal, yet this model can be recast as a random field of a particular type.

The score produced by the language model of the APRIL3 system is a numerical measure related to the probability of the tree. It is quite close to being a probability but there are some alteration factors applied that mean it will not sum to 1. It is defined thus:

> The figure of merit assigned to an entire tree over a word-hypothesis lattice will be the product of the scores worked out as above for the productions associated with the various non-terminal nodes of the tree, the scores worked out as discussed in ch. 6 for the various terminal nodes of the tree and a downgrading factor for each word [that violates certain capitalisation rules].

In addition root nodes are treated separately and are given a score that is

> ...the product of these conditional probabilities for each of the non-punctuation constituents immediately dominated by the root.

. There is a further scoring components that deals with conjunction and multiplies the scores from the productions with additional scores for closing conjuncts.

It is not possible to go into great detail onto the exact equivalence, but given that all the root nodes in the data modelled by the APRIL system, which is based on the SUSANNE parsing scheme have a distinguished label, we can define a separate feature function for each mother label, including the special function for the root label, and separate features for the presence of conjuncts, and for penalising miscapitalised words, and we have a random field. In this case we have one complex feature taking a non-zero value at each node, namely the feature that has the correct mother label.

Getting the penalty factors to conform to the formal definition of the random field requires a bit of work[5] , but the whole model fits nicely within the framework of a random field, despite its complexity, and the fact that it produces scores not probabilities. This is because the score is essentially the product of scores calculated independently at each node, each of which has a restricted domain of locality, or 'support' in random field terminology.

[5]Since the penalty depends in some cases on the word being sentence initial, to be very strict it might require an additional two nodes to form a labelled arc from the sentence node to its first word node.

Chapter 3

Previous Work

3.1 DDL

DDL introduce an application of random fields. Their domain of application was inferring some rules or patterns of English orthography from a large[1] corpus of written English. Figure 3.1 shows some of the first 1000 features produced by their algorithm together with the values of β ie e^{λ}, and of λ. The symbols

feature	β	λ
$s >$	7.25	1.98
$< re$	4.05	1.40
$ght >$	3.71	1.31
$al > 7+$	94.19	4.54
$[a - z][A - Z]$	0.003	-5.8
ae	0.001	-7
$ing >$	16.18	2.78
$< wh$	5.26	1.66
$ment$	10.83	2.38

Figure 3.1: Sample of the first 1000 features

$<$ and $>$ signify the beginning and ends of words, and 7+ means it must form part of a word with 7 or more letters. So the model has for example successfully recognised that 'ing' occurs very frequently at the end of a word, and that 'al' occurs frequently at the end of long words. Note also the presence of *negative* features such as 'ae', whose presence will reduce the score of a string.

Their task is to minimise the Kullback-Liebler divergence[2] defined as:

$$D(\tilde{p} \parallel p) = \sum_{\omega \in \Omega} \tilde{p}(\omega) log \frac{\tilde{p}(\omega)}{p(\omega)} \tag{3.1}$$

where $\tilde{p}$ is the observed distribution and p is the random field. This sum ranges over Ω which is the configuration space – in this application the set of all strings

[1] 350 million words

[2] The choice of the Kullback-Liebler divergence or relative entropy is fairly standard – see for example [PTL93]

drawn from the appropriate terminal alphabet. Note that you only need to sum over those configurations with $\tilde{p} > 0$, ie in the case where you are just looking at the sample frequencies you just need to sum over the configurations in the corpus in question. The outline of their algorithm is as follows:

- Start from a null field with no features.
- Propose a set of new features consisting of atomic features or atomic features combined with field features.
- Calculate which of these will have the highest 'gain'.
- Choose this one and re-estimate the parameters of the field to bring the expectations of the features in the field in line with the observed expectations.
- Repeat from step 2.

I will consider the various points in turn.

3.1.1 Feature proposal

They have a rather simple proposal system. They start with no features. They define a set of atomic features. Candidate features are then either atomic features or combinations of atomic features and features in the field. Thus the algorithm will incrementally build up increasingly complex features that cover longer portions of the input.

3.1.2 Feature selection

Their algorithm is 'greedy': at each step it adds the feature that will give it the largest immediate reduction in the Kullback-Liebler divergence. They prove that for each feature the maximum gain for that feature will be given by a weight α that is the solution to:

$$\tilde{p}[g] = q_{\alpha g}[g] \tag{3.2}$$

where $q_{\alpha g}$ is the field derived by taking the field q and adding the feature g with weight α. The square bracket notation here means the expectation ie

$$q[g] = \sum_{\omega \in \Omega} q(\omega)g(\omega) \tag{3.3}$$

where q is some probability and g is some function ie a feature.

To solve this, first we compute some coefficients g_k which are the expectations of the feature taking the value k (occurring k times in the case of tied binary features):

$$g_k = \sum_{\omega} q(\omega)\delta(k, g(\omega)) \tag{3.4}$$

where $\delta(i, j) = 1$ if $i = j$ and is 0 otherwise. So for each k, g_k is the probability that g will take the value k. This means that

$$q[g] = \sum_{k=1}^{N} k g_k \tag{3.5}$$

where N is the maximum value that g can take and for any function h of the feature

$$q[h(g)] = \sum_{k=1}^{N} h(k) g_k \tag{3.6}$$

in particular

$$q[e^g] = \sum_{k=1}^{N} e^k g_k \tag{3.7}$$

DDL show in this way that the solution of equation 3.2 will be the solution to:

$$\tilde{p}[g] = \frac{\sum_{k=0}^{N} k g_k \beta^k}{\sum_{k=0}^{N} g_k \beta^k} \tag{3.8}$$

where $\beta = e^{\alpha}$.

This equation(3.8) is straightforward to solve using standard numerical techniques.[3]

The gain produced is given by:

$$G_q(\alpha, g) = \alpha \tilde{p}[g] - \log q[e^{\alpha g}] \tag{3.9}$$

3.1.3 Re-estimation

The previous step estimated the gain by leaving all the other parameters unchanged. There will obviously be interactions between the features, and so there is a necessary step of adjusting all the parameters after the feature has been chosen and added to the set of active features of the field. This cannot be done in closed form so has to be done by iterative re-estimation. DDL produce the following formula for one iteration of the algorithm:

$$\sum_{m=0}^{M} a_{m,i}^{(k)} \beta_i^m = 0 \tag{3.10}$$

where k is the iteration number[4] , m ranges from 1 to the maximum value of any feature, i is an index over the features, $\beta_i = e^{\gamma_i^{(k)}}$ and the coefficients $a_{m,i}^{(k)}$ are defined by:

$$a_{m,i}^{(k)} = \begin{cases} \sum_{\omega} q^{(k)}(\omega) f_i(\omega) \delta(m, f_{\#}(\omega)) & m > 0 \\ -\tilde{p}[f_i] & m = 0 \end{cases} \tag{3.11}$$

where

$$f_{\#}(\omega) = \sum_{i=0}^{n} f_i(\omega) \tag{3.12}$$

At each iteration we change the value of each parameter by $\gamma_i^{(k)}$ and when it converges we stop.

DDL say (p. 13)

[3] I used the Newton method, together with a few bits of ad-hocery to produce the correct (non-negative) root when the order of the polynomial, N, is even.

[4] I follow DDL's notation: it is unnecessarily confusing that they change k from ranging over the values of the features to being the iteration number, and replace it with m.

> We assume that $q_0(\omega) = 0$ whenever $\tilde{p}(\omega) = 0$. This condition is commonly written $\tilde{p} \ll q_0$, and it is equivalent to $D(\tilde{p} \parallel q_0) < \infty$.

This is rather problematic. Recall that $\tilde{p}$ is the reference distribution – DDL define it thus:

> Typically the distribution $\tilde{p}$ is obtained as the empirical distribution of a set of training examples.

This appears to mean that, given a set of training examples of size N, the empirical value for a feature f will be the number of times f occurs in the corpus divided by N. This is the *maximum likelihood estimator* and has a number of well-known flaws. As Sampson([Sam96] page 135-136) observes

> In particular, it is important to understand that dividing observed frequency of examples of a category by total observed examples of all categories is *not* the best way to estimate the probability of the category.
>
> This comes as a shock to many people.

In their application where the data points are words, and they are using a corpus of several hundred million examples, the observed distribution is probably quite a good fit to the empirical distribution, as the ratio of tokens in the sample to types in the data set is quite high. In other examples, such as the one I exhibit later on, or in Abney's work, the number of types is roughly the number of sentences in the English language, and the number of tokens will be the number of sentences in the parsed corpus being used – clearly the ratio in this case will be much less favourable. This makes it essential to use some sort of smoothing rather than just using the observed distribution.

This statement commits us to an initial null field that will not generate any configurations not present in the corpus – an impossible task. There are two possible solutions. The first is to have some kind of smoothing of the empirical distribution so that $\tilde{p}$ assigns some non-zero probability to all configurations in the domain of the null field. This is the solution that Sampson recommends – using a Good-Turing smoothing technique([Sam96], p 137-146, [GS96]). Otherwise we will be forced to assign a weight of $\lambda = -\infty$ or $\beta = 0$ which will assign a probability of zero to all trees with this configuration. This is undesirable because the configuration may be absent from the corpus because of sparse data problems, not because it is extremely low probability. This is a variant of the classic problem of over-training. The other possibility is then to limit the ability of the model to exclude configurations by putting a lower bound on the values of the weights.

In fact it turns out not to be critical: the formula for the gain is incorrect though in the case where $\tilde{p}[g] = 0$, but I have derived another formula that gives the correct result in this case.

3.1.4 Sampling

We need to calculate the expectations of these features under the random field. This cannot be done in closed form in general, because the calculation involves looking at each possible configuration – which though it is finite for each subfield (the set of all trees with the same number of nodes) becomes prohibitively large extremely quickly. It is therefore necessary to use Monte Carlo techniques, that is to generate a random sample from the field, and count the number of occurrences of each feature . They use Gibbs sampling, which can be show to be a specific application of the more general Metropolis-Hastings-Green algorithm. They first of all pick a length at random, and then repeatedly go over each character in the string, changing it to one drawn from the marginal distribution, that is the distribution for that character with all the other characters fixed. This distribution is fairly straightforward to calculate given the nature of the features in the field.

3.1.5 Criticisms

The only constraints that they can develop are concatenations of basic features. They cannot create disjunctions of features, and as such they fail to learn for example the vowel/consonant distinction. Nor can they learn facts about inflectional morphology that depend on the syntactic class of the output of rules.

They don't look at how the errors in the Monte Carlo estimation of the feature expectations will affect the solution of the equations. In fact they confuse the true feature expectations with the estimates when they assume that there is some N such that $g_k = 0 \, \forall \, k \geq N$. In fact one would expect the true feature expectations to be greater than 0 for all positive k, unless one allows some arbitrary limit to the length of English sentences. They do not consider the issue of smoothing the Monte Carlo estimates: this will cause problems when the estimates are zero, if the feature in question occurs in the corpus, because equation 3.8 will be insoluble.

It is useful to separate out the various elements of their model. First of all we have the choice of the exponential random field, together with the use of the Kullback-Liebler divergence to measure goodness of fit, which is well-motivated on various grounds. We then have the feature selection algorithm which estimates the gain, by assuming the other feature weights stay the same. This is clearly a useful technique. Next we have the re-estimation algorithm, which is a very useful and general technique, that is quite independent of the feature selection algorithm. Finally we have the feature proposal algorithm that at each step proposes a set of candidate features based on the current state of the field. This is the least theoretically interesting part and depends greatly on the particular application.

3.2 Abney

Abney critiques earlier work on estimating parameters of attribute-value (AV) grammars. He shows that Brew's[Bre95] work is flawed in that the estimation is biased – that is, in the infinite data limit, the values do not converge to the correct ones.

3.2.1 Domain

Abney wishes to construct distributions over directed acyclic graphs (dags) : given a set of dags and an attribute-value grammar, one must define a distribution over the set of admissible dags, and estimate the parameters. One way is to

Tree	Probability	Field Weight	Probability * Weight
$(N((T))$	0.9	e^5	133.57
$(N((N((T))))$	0.09	e^{10}	1982.3
$(N((N((N((T))))))$	0.009	e^{15}	29421
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\underbrace{(N((N \cdots ((T) \cdots)}_{n\,Ns}$	$9 * 10^{-n}$	e^{5n}	$9e^{(5-log(10))n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 3.2: Pathological tree sequence

associate parameters with each AV rules, and then estimate them by assigning the observed weights, and then renormalising. Abney shows that this technique due to Brew and Eisele[Eis94] is incorrect in that the weights do not converge to the correct weights[5] as the sample size goes to infinity.

Abney shows that the random field techniques developed by DDL do have the desired properties.

3.2.2 Field definition

Abney defines his field over a null field defined by a stochastic CFG. This is extremely problematic because this field may in general not be well-defined because the normalisation constants, which are sums over the unnormalised 'probabilities', may diverge. Intuitively this is because the raw SCFG probabilities get smaller exponentially but the field weights get bigger exponentially. A small example will illustrate this.

Consider a stochastic CFG defined as follows:

- One non-terminal symbol N
- One terminal symbol T
- $0.1 : N \rightarrow N, T$
- $0.9 : N \rightarrow T$
- A single atomic feature f, which is merely the presence of N, with weight $\lambda = 5$

The sequence of trees in Figure 3.2 will have increasing values of the product of the field weight and the probability with respect to the context free grammar, and as a result the expectation of the field weight will be infinite with respect to the SCFG and thus the field is unnormalizable.

3.2.3 Sampling algorithm

Abney suggests the Metropolis-Hastings algorithm as the appropriate sampling method. The particular technique he suggests is called the random-walk Metropolis-Hastings algorithm. This is not suitable, as the stochastic CFG will

[5] defined as those that minimise the Kullback-Liebler divergence

not be sufficiently close to the true densities. Thus it will not propose enough in the high density areas and the MH algorithm will have ‘too much to do’ – it will get stuck in the same place for a very long time. There are however other more suitable variants of the MH algorithm that can be used as I shall discuss below.

Chapter 4

Application

I wish to apply this work in a slightly different domain.

4.1 Meta-theoretical context

Abney's work involves estimating parameters over an attribute-value grammar. This defines a probability distribution over the set of well-formed sentences or phrase-markers defined by that grammar. The probability distribution so defined will clearly be useful for a range of purposes notably disambiguation. It will not however deal with the converse problem of under-generation. In addition it seems that there is an undesirable doubling up of linguistic information between the base cfg and the random field. In the absence of some principled way of dividing the effort between these two models there are bound to be areas of confusion and overlap. It thus seems preferable to shift all of the linguistic modelling into the random field, and have the null field being as neutral as possible – ideally being the maximum entropy distribution over the set of trees in question[MOS92].

There are some well-known areas of context-sensitivity in English grammar. By this I do not mean that the string set is non context free, rather that there are statistical dependencies that seem to extend beyond local trees. The tendency for subjects of sentences to be pronouns is a common example - there are more complex ones[CW97].

Abney says:

> As a closing note, it should be pointed out explicitly that the random field techniques described here can also be profitably applied to context-free grammars. As Stanley Peters nicely puts it, there is a distinction between *possibilistic* and *probabilistic* context-sensitivity. Even if the language described by the grammar of interest – that is, the set of possible trees – is context-free, there may well be context-sensitive statistical dependencies. Random fields can be readily applied to capture such statistical dependencies whether or not $L(G)$ is context-sensitive.

4.2 The grammaticality distinction

Only recently has the idea of grammaticality been called into question. Sampson uses evidence from the SUSANNE corpus[Sam95] itself to question this, showing quite persuasively that there is no *prima facie* evidence for this distinction. This assumes however that this is an empirical point: in practice it seems to be more a meta-theoretical point.

If we abandon the notion of grammaticality as a pre-theoretical intuition and merely look at all the data, we can possibly recover 'grammaticality' later as an emergent statistical property. Clearly probability itself is not a measure of acceptability as sentences with the word 'otiose' in are much less frequent than those with 'unnecessary' but not in general less grammatical. However we might find that the ratio of the probability assigned under a simple uni-gram model, to that defined under a more sophisticated model might capture many of the intuitions of syntactic grammaticality.

In any event we can see that the traditional notion of grammaticality is a special case of this approach – a distribution is a function from a large set of phrase markers into the set of real numbers $[0, 1]$, ie $\{x|0 \leq x \leq 1\}$. The traditional generative grammarian would define a language as a set of well-formed phrase markers, the characteristic function of which is a function from a set of *possible* phrase markers into $\{0, 1\}$.

4.3 Application Domain

My goal is thus to produce a richer context sensitive statistical model of a parsed corpus. Given a set of parsed data, the goal is to construct a statistical model that is a random field, using variants of the algorithms in DDL and Abney. There are other models one could construct more easily: one could construct a simple stochastic cfg by choosing as rules all local trees that occur in the corpus and weighting by their observed frequencies. This would be very straightforward but would have several drawbacks:

- It could not pick up any context dependencies in the corpus.
- In the absence of any smoothing, any tree that contains a local tree that doesn't occur in the corpus will be inadmissible.

4.4 Definition of the field

Abney's definition of the field as discussed above has some technical problems. My approach is slightly closer in feel to the DDL definition. We partition the domain of the density function into finite sets that obviously can then be normalised. I partition them according to the total number of nodes in the tree. This is not an ideal solution however. For a start, if one wished to parse using some form of stochastic optimisation, one needs to be able to keep the number of terminal nodes constant but vary the number of non-terminal nodes. This would cause problems because to compare the scores of two trees with different total number of nodes would require knowledge of the normalisation constants for the two values of the 'size' of the trees. This is extremely difficult to do, but not impossible[1].

So formally we define the field as follows. We have a set of features $\{f_1, \ldots, f_m\}$ each with an associated weight of λ_i. In addition we have a distribution p_0 over the size (total number of nodes) of the trees. The normalisation constants are then:

$$Z_n = \sum_{|x|=n} \left\{ \prod_{i=1}^{m} e^{\lambda_i f_i(x)} \right\} \tag{4.1}$$

and the probability density function can be written as:

$$p(x) = p_0(|x|) \frac{1}{Z_n} \prod_{i=1}^{m} e^{\lambda_i f_i(x)} \tag{4.2}$$

We tie the features so that λ_i is the same for all values of n. Note that we are using a uniform (maximum entropy) distribution over trees of the same size. This is slightly problematic because it is difficult to sample directly from that distribution – the combinatorics are rather difficult. Knuth gives a method of calculating the formulae involved[Knu97].

Since the normalisation constants are finite sums there is no risk of them diverging.

4.5 Limitations

The final state of this process will be a random field. The way this is defined there will be upper bounds on the depth of the trees that define the features. This means that this approach cannot directly capture unbounded dependency constructions. This can be overcome by using features in a GPSG-style framework[GKPS85]. By using features to pass the details of UDC's up and down the trees, we can turn non-local dependencies into local ones. Thus in this approach, given a large GPSG-annotated corpus, we could define features, or more likely sets of features, that could capture generalisations expressed in the GPSG framework as the head feature principle, slash percolation meta-rules and so on.

4.6 The Lexicon

Most of the discussion is of syntactic rules: clearly regardless of how much syntax one might want to stick in the lexicon, there is the lexicon to learn. The quantity of information contained in the lexicon seems like it might require rather different techniques, in implementation at least, even if there is some theoretical unity underneath.

[1] There is a technique called reverse logistic regression that might work.

Chapter 5

Algorithms

5.1 Overview

Given a data set consisting of a set of trees, with each node labelled from a finite set of possible labels, our chosen task is to model this with a random field. So we define a feature set that will capture the right sort of generalisations over the data, and an algorithm that will generate successively larger and more complex feature sets. In the abstract we can separate the algorithm into a proposal step and a selection step.

5.2 Feature proposal

One of the difficulties with the algorithms proposed by DDL is that a significant complex feature may not have any parts that are significant. Even in the linear case which DDL examine we can see that if for example 'ing' appears more often than by chance , this does not necessarily imply that 'in' or 'ng' will appear more than by chance – there might be very low expectations of 'ina', 'inb' and so on that would counterbalance the high expectation of 'ing'.

5.3 Feature selection

We will follow DDL in using a greedy algorithm that attempts to maximise the gain at each step of the algorithm. We have an algorithm that at each step proposes a set of new candidate features. We then use the gain formula given by DDL, with a modification discussed below, to choose the feature we add at this step.

5.4 The gain formula

In the case when $\tilde{p}[g] = 0$ (that is the feature does not occur in the corpus at all) the formula DDL give for the gain is incorrect. Obviously we will wish β to be 0, so we do not need to solve any equations for α as we will want it to be $-\infty$. This will have the effect that all trees containing the feature, ie with $g > 0$ will then have a probability of zero. The gain will be given by:

$$G_q(-\infty, g) = -\log(g_0) \tag{5.1}$$

Intuitively this gives the right result too, as of the features that don't occur in the corpus, the one that occurs most often in the field will give the highest gain. This can be derived thus:

The gain of adding feature g with weight α is defined as:

$$G_q(\alpha, g) = D(\tilde{p} \parallel q) - D(\tilde{p} \parallel q_{\alpha g}) \tag{5.2}$$

From the definition of the Kullback-Leibler divergence we have:

$$G_q(\alpha, g) = \sum_{\omega \in \Omega} \tilde{p}(\omega) \log\left(\frac{q_{\alpha g}(\omega)}{q(\omega)}\right) \tag{5.3}$$

By hypothesis, g takes the value 0 whenever $\tilde{p} > 0$, as g does not occur in the corpus, so the unnormalised field weights in the numerator and the denominator are the same. That is, for every tree in the corpus,(every $\omega \in \Omega$ where $\tilde{p}(\omega) > 0$), g does not occur, so the unnormalised field weights with respect to q and $q_{\alpha g}$ are the same. The only difference is caused by the decrease in the normalisation constants caused by the zeroing of the elements where $g > 0$ (which occurs in the region of the configuration space where $\tilde{p}$ is zero), and this decrease is just by the expectation of the new feature with respect to the old field i.e by a factor of $\frac{1}{q[e^{\alpha g}]}$. Thus

$$q_{\alpha g}(\omega) = \begin{cases} 0 & g(\omega) > 0 \\ \frac{q(\omega)}{q[e^{\alpha g}]} & g(\omega) = 0 \end{cases} \tag{5.4}$$

In this case it is easy to see that

$$q[e^{\alpha g}] = \sum_{k=0}^{N} g_k e^{\alpha k} = g_0 \tag{5.5}$$

since $\alpha = -\infty$ and thus the gain is

$$G_q(\alpha, g) = \sum_{\omega \in \Omega} \tilde{p}(\omega) \log\left(\frac{1}{g_0}\right) = -\log(g_0) \tag{5.6}$$

Chapter 6

Monte Carlo methods

The expectations of the features under the random field cannot in general be calculated directly. They must be estimated by Monte Carlo methods. That is to say we generate a sample of them at random, and then count up the expectations. However there are a number of complications. First of all we cannot generate random samples from an arbitrary field. We cannot even calculate the probability of a sample when we have generated it because we do not know the value of the normalisation factors involved. Luckily these are not insuperable problems. There are a number of ways of sampling that do not require either of these. In particular the Metropolis-Hastings-Green algorithm is a very general set of algorithms that can be used to generate arbitrary distributions when we have a sampling distribution that is absolutely continuous with respect to the desired one, and a way of calculating the ratios of probabilities of pairs of samples. In this case both these conditions are satisfied so we have a solution. However we do not have any guarantee that the samplers will converge rapidly, and the naive application of that does in fact converge prohibitively slowly.

These form a class of samplers called Markov Chain Monte Carlo (MCMC) samplers[Gam97], because rather than generating the distributions directly, you generate a Markov Chain, designed so that the stationary or equilibrium distribution of the chain is the distribution you wish to sample from.

6.1 The Gibbs sampler

The Gibbs sampler can be shown to be a special case of the Metropolis-Hastings algorithm but is slightly different in feel. The Gibbs sampler works by going through the variables of the data, one at a time, and sampling from each with respect to the marginal distribution of that variable. This therefore requires that you can calculate and sample from the full marginal distributions for each variable. In DDL's application it is easy to see that this is straightforward – each variable is a letter, and if the rest of the word is fixed, it is fairly easy to calculate for each new letter what the new weight would be, and thus what the marginal distribution should be. This can be shown to be a Metropolis-Hastings algorithm where the acceptance ratio (as defined in equation 6.1) is always 1 – ie each change is always accepted.

There are a few varieties of the Gibbs sampler. The *canonical* Gibbs sampler proceeds in sequence through each variable, keeping a fixed order and at the end starting again at the beginning $1, 2, \ldots, n, 1, 2, \ldots$. Other variants choose each variable at random at each step. In the current application, it might be possible to use the Gibbs sampler for the labels of the trees. In that case there could be several versions of the canonical Gibbs samplers – bottom-up and top-down for instance. A bottom up one would first apply the sampler to the leaves of the tree, then to their parents, and so on and the top-down would do the reverse. I have not attempted to implement this though.

6.2 The Metropolis-Hastings algorithm

This algorithm is commonly defined as follows(p. 7 [CG95]):

Given a distribution that you can sample from that is allowed to depend on the current state of the process, since this is a Markov Chain method, $q(x, y)$ and a way of calculating the probabilities of the distribution you wish to sample from, π, we proceed as follows. You generate an element x_n , and then generate a new element y from q. You then either accept it or reject it, with probability $A(y|x_n)$ defined thus:

$$A(y|x_n) = \min\left\{1, \frac{\pi(y)q(y, x_n)}{q(x_n, y)\pi(x_n)}\right\} \tag{6.1}$$

If it is accepted then you add this to the corpus you are generating. If it is rejected then you add x_n to the corpus again. You carry on in this way. Note that we might reject several in a row – the MH algorithm is characterised by long sequences of duplicate elements - these are what adjust the expectations. These blocks of duplication occur in the areas where q is higher than p, and account for the adjustment from p to q. Note also that as π appears both in the numerator and the denominator of the fraction, we only need to know π up to a constant of proportionality – which is essential in our case because we do not know the normalisation constants of the fields.

In situations where all the probability mass is concentrated in a comparatively small volume (as measured by the sampling distribution), the MH algorithm will perform extremely poorly. When it finds by chance a high field weight tree, it will get stuck for a long time in that tree. This very high auto-correlation means that prohibitively long runs would be needed to get convergence. In addition, you have to discard the initial portion of the run because the distribution will not have converged yet to the equilibrium distribution of the Markov chain.

There are a few additional criteria that must be met in addition for the Markov Chain to be guaranteed to have an equilibrium distribution: irreducibility and aperiodicity. Irreducibility means there must be a non-zero probability of moving from any point in the space to any other after a large enough number of moves, and aperiodicity means that it must not be the case that you only move there at periodic intervals – for example, if the chain could only visit a particular point on even numbered occasions, this would violate aperiodicity.

6.3 Detailed Balance

The acceptance formula used above is derived from the requirement of what is called 'detailed balance[1]'[CG95].

$$\pi(x)q(x,y) = \pi(y)q(y,x) \tag{6.2}$$

where $\pi(x)$ is the distribution that you wish to sample from and $q(x,y)$ is the generating probability. $\int q(x,y)dy = 1$, and at each x a new point y is generated from $q(x,y)$.

In Abney's paper he uses a $q(x,y)$ that does not depend on the current state of the process x that is $q(x,y) = p(y)$.

6.4 Multi-path MH

An attempt to overcome this is to use a multi-path algorithm. Instead of having a single path that we wish to traverse all the space, we do a large number of different runs, starting in different places, and average them. This will not be enough in this case, as the algorithm still won't stumble over the important parts enough.

6.5 Variable at a time MH

The only way the algorithm will get to the high density areas is by annealing – that is by randomly changing small parts of the tree structure, and discarding most of the changes that lead to a decrease in the score. This means that the algorithm has a chance to find the sort of complex structures that have high scores.

6.6 Re-estimation and Importance sampling

I have experimented with the use of importance sampling to accelerate the re-estimation procedure. Because I am using slightly samplers that have not fully converged, the re-estimation process will not converge properly, as even at the theoretically correct point in parameter space, the estimation errors will cause the algorithm to oscillate. I thus have tried using importance sampling, with a fixed set of samples for several iterations. This will only work when the changes in the parameters are quite small, but it is much quicker. I need a way of telling when this is not working very well.

Importance sampling is a rather primitive form of Monte Carlo sampling, where a sampler for p is converted to a sampler for q, by reweighting the samples by the ratio of the weights – that is by their importance. The advantage of this for the re-estimation, is that we can sample once for the whole algorithm, not once per iteration. Unfortunately, one has to make an unjustified (ie false) assumption that the normalisation constants don't change when the weights are changed. This is clearly not always the case, but I was unable to implement

[1]This is also called 'reversibility'

an alternative in the time available. Since the basic samplers are so erratic this seems almost the only way to get the re-estimation algorithm to converge.

Chapter 7

Design

7.1 Data

7.1.1 The SUSANNE corpus

Initially, I decided to use the SUSANNE corpus[Sam95]. This corpus has several advantages:

- It is parsed
- It has rather flat constituent structures
- It is in the public domain

I decided to just use atomic features, and in order to restrict the number of them I performed the rather radical step of mapping them all to the first character of the label. This is clearly extremely sub-optimal, but is the obvious first simplification.

It soon became clear however that the technical problems involved with using real data were too complex to be dealt with in a project of such restricted scope so I determined to use a set of fake data, that I could generate from a simple stochastic grammar that would have some context dependencies that could be picked out by the algorithm.

7.1.2 The stochastic grammar

I decided to generate the fake corpus from a simple generalisation of stochastic context free grammars. Rather than having rules that expand an initial category to a local tree, I allow rules to expand categories to trees of arbitrary depth, where the terminal nodes of the rule-trees can be either terminal or non-terminal nodes of the language. This rather crude formalism has a number of difficulties that make it unsuitable for linguistic modelling – the dislocation of constituent trees and derivation trees, and the difficulty of calculating directly the probabilities of a particular tree – but for the simple task of generating a corpus with some context dependencies it seems adequate.[1]

[1] Bod([Bod93] uses this formalism in his data-oriented parsing, where it is called a *Stochastic Tree-Substitution Grammar*. He uses a very large grammar generated by every fragment of the corpus, and has to resort to Monte Carlo methods to calculate the probabilities.

7.2 Features

Given these characteristics, there are a number of sensible ways to define the basic features. It is quite difficult to get an intuition about how the features interact in a random field. Since features can have weights that are negative[2], features can override each other. Indeed part of the point of the re-estimation is to take account of this effect, and part of the weakness of the gain-estimation in DDL's algorithm is that it will fail to notice this.

The selection of the form of the features depends almost entirely on the structure of the data that is to be modelled. The two data sets I am using are very different in structure and thus should me modelled quite differently. The SUSANNE corpus has a very large number of local trees: it tends to have rather flat constituent structures. The local trees form a fairly open class: the features therefore must be general enough to cover sets of trees. On the other hand the corpus generated by the stochastic grammar has deep structures with low numbers of daughters in each local tree.

It seems necessary to separate out in some way the ordering information implicit in the constituent structure trees, from the dominance relations. One way of doing this is to use the division into immediate dominance rules and linear precedence rules developed in the GPSG formalism[GKPS85]. This however involves some substantive issues – in particular the exhaustive constant partial ordering (ECPO) hypothesis, that may not be observed in the data under study.

I decided not to try to model the data completely, but rather to see whether the algorithm could extract some useful features. I therefore used only features that did not depend on the order of the nodes in the tree at all. In addition, I decided not to try to model the words at all, but to restrict features to the labellings of the tree. The features then are just trees, with each node having a label, and there being no order to the daughters of a particular node. At any stage the algorithm can choose an atomic feature, that is the tree consisting of a single node, or a tree made up of a feature already in the field, combined with an atomic feature. They can be combined in one of two ways. The new feature can consist of the atomic feature at the root, and the feature taken from the field as its only daughter, or alternatively of the feature taken from the field at the root, and the atomic feature added either as a daughter or granddaughter. This is obviously not completely general, but the choice of feature combination algorithm must steer a fine line between combinatorial explosion and failure to propose significant features. This seems to have the balance about right.

7.3 Sampling

I decided to use a variant of a variable at at time sampler. Recall that the distribution is based on the uniform (maximum entropy) distribution. Sampling from this is rather difficult. Knuth gives a formula for the underlying combinatorics as follows:

$$na_{n+1} = a_1 s_{n1} + 2a_2 s_{n2} + \ldots + na_n s_{nn} \tag{7.1}$$

where a_n is the number of ordered trees with n nodes and

[2] $\lambda < 0 \Leftrightarrow \beta < 1$

$$s_{nk} = \sum_{1 \leq j \leq n/k} a_{n+1-jk} \tag{7.2}$$

Given these coefficients one could sample without too much difficulty from the uniform distribution choosing the labels again uniformly and independently. In fact I didn't use this correct algorithm – I sample from a rough approximation to this distribution.

7.3.1 From the null field

We define a null field in a simple way from a corpus. We define the distribution of sizes of trees as the observed distribution of the sizes of trees in the corpus, without smoothing, and we gather the possible labels for each of the nodes.

To sample from the null field we draw independently at each step a size from the size distribution, and then we choose a random tree with that size, and then label each node with a label drawn uniformly and independently from the list of possible labels. To sample from the set of all trees with a certain size, we merely start with the unique tree of size 1, and then for each new node, we add it at as the daughter of one of the nodes in the tree, chosen uniformly. As it stands this has a slight bias to make daughters on the right have more daughters than daughters on the left, as we merely push the new nodes onto the front (the left) of the list of daughters of the nodes. Though this bias is plausible, it is eliminated by shuffling the order of all daughters with a simple algorithm.

7.3.2 From the field with features

Starting from this sampler to the null field we now convert it into a sampler for the random field with a Metropolis-Hastings step. If we merely choose a tree at random it is unlikely it will ever be in a region of high probability, because a moderately complex field will concentrate all of the probability mass into very small areas. So we need to allow the sampler to make a sequence of small changes to the tree, that will tend to make it walk into the high probability regions of the configuration space. This is different from Abney's sampler where each choice does not depend on the current state of the sampler.

The changes to be made to the tree fall into two classes. First the labels on each node can be changed. To do a move of this type, we pick a node of the tree at random, and change its label to another label, chosen uniformly. Secondly we can change the structure of the tree: this is clearly more complex. There are two types of structure-changing move available: one which tends to make the tree deeper, and one which makes it shallower. The deepening move takes a node which has at least two daughters, and makes one of them the daughter of the other rather than the sister. The 'shallowing' move does the converse[3]. For a particular tree there will be different numbers of each of them available: for the maximally shallow tree (one mother and $n-1$ daughters) only deepening moves are available, and for the maximally deep tree (a chain with no siblings at all) only shallowing moves will be available. To make the

[3]Both of these moves keep the total number of nodes in the tree constant, while possibly changing the number of terminal nodes. Given the way we define the field this is essential.

change, we count up the candidate deepening moves[4], (say i) and the number of shallowing moves (say j) and then choose one at random out of the $i + j$ possible sites. Choosing a move then involves first choosing whether to make a label-changing or a structure-changing move with fixed probability (equal probabilities in this case), and the selecting further the details of this move.

To have the correct sampler we must maintain detailed balance: the probability of going from one tree to another must match the probability of going back, for the sampler to give the correct results. This requires not merely adjusting for the changing field weights as Abney does, but also adjusting for the changing values of i and j that will affect the chance of going back.

For it to satisfy detailed balance we must set the move probability to be:

$$A(y|x) = \min\left\{1, \frac{\pi(y)q(y,x)}{q(x,y)\pi(x)}\right\} \tag{7.3}$$

where $\pi(x)$ is the field probability and $q(x, y)$ is the probability of going from x to y – ie of making the move from x to y. We know from the definition of the field that:

$$\pi(x) = p_0(|x|)\frac{1}{Z_{|x|}}\prod_{i=1}^{m} e^{\lambda_i f_i(x)} \tag{7.4}$$

so substituting, and dividing out the constants, noting that $|x| = |y|$ since none of the moves change the number of nodes in the tree, we get

$$A(y|x) = \min\left\{1, \frac{F(y)q(y,x)}{F(x)q(x,y)}\right\} \tag{7.5}$$

where $F(x)$ is the field weight. So in addition to the ratio of the field weight we also need to adjust the acceptance ratio by the ratio in the probability of going from x to y to that of going from y to x which is fairly easy to calculate given the various values of i and j and the number of sisters and so on at the relevant nodes. The way I have designed this algorithm is so this correction factor is close to 1: in any event I am not completely confident that my code calculates the correct answer in all circumstances. There is a certain amount of flexibility in how this algorithm is designed; the issue is to choose a set of moves, and a way of selecting them that makes the correction factor $\frac{q(y,x)}{q(x,y)}$ easy to calculate – this way seems as easy as any other.

7.4 Accuracy

It is obvious that the Monte Carlo estimates will become increasingly inaccurate as the features get more complex and thus more infrequent. This should not be a problem in general. During the feature selection phase, if the estimate of the expectation of a candidate feature is incorrect this will merely distort the predicted gain, and thus make it more or less likely to be chosen *at that step*. The fact that the algorithm is run repeatedly means that a feature, if it is significant, will eventually be chosen. The variations in the Monte Carlo estimates will just effect the order the features are chosen in. Obviously it means that some rather

[4] Actually the number of nodes that can be the site of a deepening move. For each possible site there may be several possible moves to make.

irrelevant features will be chosen but these will only degrade the performance marginally and could be pruned out in later runs of the algorithm.

Similarly in the re-estimation phase, the variations in the estimates will show up as noise in the values of the weights of the features. As long as the weights are accurate when the algorithm terminates, this will merely degrade the algorithm marginally. Thus it should be possible to 'get by' with rather short and thus inaccurate sampling runs during the body of the algorithm, and then to do one long, slow, accurate re-estimation when the feature selection process has finished.

Chapter 8

Implementation

8.1 Language choice

It is implemented in CLOS the object-oriented extension of Common Lisp[Ste90]. Given the rather small size of the program, I did not use the package system at all. I avoided using very Lisp-specific programming techniques to make it easier to port to other more efficient languages: in particular I limited my use of higher order functions (passing functions as parameters) to a few very limited places.

8.2 Modules

The program is divided into the following components:

susanne that loads files of data from the SUSANNE corpus into a corpus object.

fake that generates a corpus of data from a stochastic grammar.

re-est that re-estimates the parameters of the field using DDL's algorithm, using the importance sampling modification.

poly that solves polynomial equations using the Newton method.

proposal that proposes new candidate features.

tree that contains the basic definitions for the tree structures.

feature that contains the definitions for the various types of feature, and the functions that match them against the trees.

expect that computes various functions over corpora.

corpus that defines the objects that have sets of trees: these are both real corpora and corpora generated by the Monte Carlo samplers from the field definitions, or from the stochastic grammar.

field that defines the structures of the fields and the initial null fields.

name	description	value
`*re-estimation-max-iterations*`	maximum number of iterations for the re-estimation algorithm	500
`*re-estimation-error*`	admissible error	10^{-5}
`*re-estimation-burn-in*`	number of trees to discard at the beginning of each path	1000
`*re-estimation-paths*`	number of discrete paths to take	1000
`*re-estimation-sample-size*`	number of samples from each path	1000
`*max-polynomial-iterations*`	maximum number of iterations for the solution of the polynomials	200
`*min-polynomial-error*`	admissible error for the polynomials	10^{-13}
`*pad-value*`	value added to polynomial coefficients to make them soluble	1
`*max-tree-length*`	maximum number of daughters in a local tree	1000
`*max-feature-occurrence*`	maximum number of features that can be 'on' in a single tree	1000

Figure 8.1: Control variables

In the final version of the program, I was working almost exclusively with the fake data. Using the SUSANNE data was very useful though as a 'reality check' because it rapidly showed up errors in my code where I had not anticipated particular problems.

My main aim in the choice of data structures was to make them flexible and easy to modify, since I knew that what I was trying to do would probably change during the course of the project. I thus tried to have fairly clean interfaces between the various parts of it.

8.3 Efficiency

The computational demands of these techniques are extensive. The major problem is the rate of convergence of the samplers which is extremely slow and gets slower with more complex fields.

8.4 Tuning Parameters

Most of the parameters that govern the behaviour of the program are 'special' variables, that is global ones. They govern for example the convergence criteria for the numerical components, the maximum number of iterations of various loops and so on. Figure 8.1 gives a list of most of the important ones, a description, and their initial values.

8.5 Smoothing

A problem that I encountered frequently with more complex features was that the Monte-Carlo estimates returned zero even for configurations that were intuitively reasonable given the rest of the field. This causes serious problems if the feature has non-zero expectation with respect to the empirical distribution $\tilde{p}$. The reason for this is just that with a fairly underspecified field, the range of possible configurations is quite large compared to the number of samples produced by the Monte Carlo algorithm. The obvious solution is to smooth the samples using for example the Simple Good-Turing algorithm[GS96]. It was not possible to implement this in the time available, so I merely aborted the re-estimation process when this occurred.

Chapter 9

Evaluation

The major problems encountered are in the efficiency of the MCMC samplers used to estimate the field feature expectations. MCMC techniques are notoriously expensive computationally, and it is in addition very difficult to judge their accuracy. It is possible for the sample variance to be very small, that is for the sampler to produce consistently similar results, and yet for the estimates to be extremely unreliable if an important mode is not sampled from at all.[Nea98]

9.1 Bugs and Errors

This is by no means a stable and tested program; there are undoubtedly many bugs and errors of varying degrees of seriousness.

9.1.1 Zero probabilities

Rather than using a separate value for $-\infty$ I merely used a sufficently large negative float that the raising e to its power would give a result of 0. This works but is not an elegant solution. The value I use is 10^{-100}, so given that the normal weights for λs do not exceed 100, it seems unlikely that a tree that contains a feature with this weight will end up with a non-zero probability.

There is a possible source of errors though in the sampling algorithm: the sampler should not propose trees with a zero probability in the field. The current sampler does. It would be possible to filter them out, but this could result in quasi-infinite loops if the sampler gets stuck in an area of configuration space where almost all possibilities are outlawed – it could spend a long time trying to find the path out of this dead-end without producing any viable samples. Because of this worry, at the moment I let the samplers produce samplers with zero field weight in spite of the errors this may cause.

9.2 Sample runs

9.2.1 Fixed trees

I analyse here several runs of the program on some simple data generated from the most simple stochastic grammar (Figure 9.1) – one which just contains fixed

Left hand side	Probability	Right hand side
S	0.45	$(S((N((A\,sing)))(V((A\,sing))))$
S	0.45	$(S((N((B\,plur)))(V((B\,plur))))$
S	0.05	$(S((N((A\,sing)))(V((B\,plur))))$
S	0.05	$(S((N((B\,plur)))(V((A\,sing))))$

Figure 9.1: Stochastic grammar for the first data set

	feature	weight
	(A)	-6.295160579391303
	(A ((B)))	-1.0E+100
	(A ((V)))	-1.0E+100
	(A ((S)))	-1.0E+100
	(A ((N)))	-1.0E+100
	(A ((A)))	-1.0E+100
h	(N)	-25.8089925489358
	(N ((A)))	37.805572898847466
	(N ((B)))	11.708348648082046
	(N ((A) (B)))	-1.0E+100
	(S ((N ((B)))))	11.972531617096886
	(S ((V) (N ((B)))))	13.528229924395474
	(S ((N) (V) (N ((B)))))	13.528229924395474
	(S ((V ((B))) (N ((B)))))	3.9834961144932937

Figure 9.2: Features and weights induced in first run

trees. I generated 1000 trees from this and then ran the algorithm for up to 20 features.

Figure 9.2 shows the first features produced by a run on the data generated by the stochastic grammar . There are a number of interesting points.

- The last feature is the first non-local dependency it has noticed.
- The group of three weights (N ((A))),(N ((B))) and (N ((A) (B))) are very characteristic, the first two have very high weights and then the last blocks the undesired interaction of them.
- (S ((N) (V) (N ((B)))) is an error caused by a bug in the matching code that has since been removed.
- (N) ends up with a large negative weight. This is to counteract the effects of the other weights that have N in and have high positive values.
- It successfully identified (S ((V ((B))) (N ((B))))) as being an important non-local feature.

Figure 9.3 shows a second run on the same data. In this case because it chose S as the first feature it was able to generate a plausible set of features very rapidly. Note that although the algorithm is in principle deterministic, because of the errors in the MCMC estimates the path the algorithm takes through the feature space is always different – ie it behaves stochastically in

feature	weight
(S)	-25.85533348966001
(S ((V)))	-10.6054108237299
(S ((N) (V)))	0.36676849797799554
(S ((N) (V ((A)))))	81.61717252757934
(S ((N) (V ((B)))))	81.57684236494306
(S ((N ((A))) (V)))	79.49499554796994
(S ((N ((B))) (V)))	74.18548277881354
(S ((N)))	10.125171663575419
(S ((N ((B))) (V ((B)))))	3.9704218453125355
(S ((N ((A))) (V ((A)))))	6.1583134809698254
(S ((N ((B)))))	6.474874437699132
(S ((N ((A)))))	8.000655551893898
(S ((V ((B)))))	7.271510441625815
(S ((V ((A)))))	7.687600100497582
(S ((N ((B))) (V ((A)))))	2.768497418904808
(S ((S)))	-1.0E+100
(S ((N ((A))) (V ((B)))))	2.6549431314629763
(S ((B)))	-1.0E+100
(S ((A)))	-1.0E+100
(A)	-0.21890491870031464

Figure 9.3: Features and weights induced in second run

feature	weight
(B)	-3.88940E+1
(B ((N)))	-1.00000E+100
(B ((V)))	-1.00000E+100
(B ((A)))	-1.00000E+100
(B ((S)))	-1.00000E+100
(B ((B)))	-1.00000E+100
(N)	-4.69389E-1
(N ((B)))	-1.355967E+0
(S ((N ((B)))))	3.48183E+1
(S ((V) (N ((B)))))	4.62240E+1
(S ((V ((B))) (N ((B)))))	3.33548E+1
(N ((S)))	-1.00000E+100
(N ((V)))	-1.00000E+100
(N ((N)))	-1.00000E+100
(N ((A)))	1.152883E+0
(S ((N ((A)))))	2.80162E+0
(S ((V) (N ((A)))))	3.50810E+0
(S ((V ((A))) (N ((A)))))	4.00769E+0
(N ((A ((S)))))	-1.00000E+100
(N ((A) (B)))	-1.00000E+100

Figure 9.4: Features and weights induced in third run

Left hand side	Probability	Right hand side
S	0.8	$(S((N())(V()))$
S	0.1	$(S((V())(V()))$
S	0.1	$(S((V())(N())(S()))$
V	0.5	$(V((V0died)))$
V	0.3	$(V((V0kissed)(N()))$
V	0.2	$(V((V0kissed)(N())(N()))$
N	0.8	$(N((N0john)))$
N	0.1	$(N((N0mary)(S())))$
N	0.1	$(N((N0john)(S((V())(V())))))$

Figure 9.5: Stochastic grammar for the second data set

practice. The discussion in section 10.4.6 on the use of stochastic optimisation of the models should be read with this fact in mind.

On the third run (Figure 9.4) we can see again a different route is taken but the set of features again includes the most significant ones. If you look at the set of candidate features in the last iteration, one can gain an idea of how large the search space is: there are 120 candidates.

9.2.2 Simple STSG

For the next runs I chose a slightly more complex grammar (Figure 9.5). It is very straightforward except for the final production, which introduces a slight non-local dependency: S is much more likely to immediately dominate 2 V's when it appears below an N. The rest of the grammar is context-free. I wanted to find out whether the algorithm could detect a comparatively slight dependency of this kind, as opposed to the completely obvious ones in the previous data. On the first run, and subsequent ones it successfully did this, as can be seen from the features/weights induced (Figure 9.6). Of particular interest is the feature $(N((N0)(S((V)(V((V0)))))))$ which successfully captures the dependency in question. However as can be seen from the list of features, as S is the rarest atomic feature, it started by adding that one and then building features from that, which seemed to guide it more rapidly than it would otherwise have gone, so a degree of scepticism is in order with respect to these results. Subsequent runs should very similar patterns of behaviour.

Figure 9.7 shows ten trees produced by sampling from the field induced from the second STSG. It is clear that it has a long way to go before it has captured all significant generalisations.

9.2.3 The weights

9.2.4 SUSANNE data

Finally I ran the program on a subset of the SUSANNE corpus. As discussed above, I pruned the labels to the first character. The first 20 features, and the corresponding weights are listed in Figure 9.9. It is difficult to gloss the labels accurately when they are truncated but *very* roughly:

N is something headed by a noun

feature	weight
(S)	-2.73944E+1
(S ((V)))	5.30887E+0
(S ((N) (V)))	1.08563E+1
(S ((V ((V0)))))	2.20936E+1
(S ((N) (V ((N)))))	-1.177001E+1
(S ((N ((N0))) (V)))	2.75756E+1
(S ((V ((N) (V0)))))	2.67556E+1
(S ((N ((N0))) (V ((V0)))))	2.29481E+1
(S ((N) (V ((V0)))))	1.160467E+1
(S ((N ((N0))) (V ((N)))))	4.02404E+0
(S ((N ((N0))) (V ((N) (V0)))))	1.176258E+1
(S ((N) (V ((N) (V0)))))	2.70484E+0
(S ((V) (V ((N) (V0)))))	-1.361705E+0
(S ((V) (V ((V0)))))	4.17914E+0
(N ((S ((V) (V ((V0)))))))	-1.638327E+0
(S ((V ((V0))) (V ((N) (V0)))))	1.24761E+1
(N ((N0) (S ((V) (V ((V0)))))))	1.232277E+1
(S ((V ((N))) (V ((V0)))))	1.276307E+1
(S ((V ((N)))))	3.46537E+0
(N ((S ((V ((V0)))))))	4.31903E+0

Figure 9.6: Features and weights for the second data set

P is a pronoun

O is a root node – a paragraph

Y is punctuation

V is something headed by a verb

I is a preposition

A is a mixture of determiners and 'as *ldots* as' phrases.

S is a sentence

These rather inaccurate descriptions will, I hope, allow a rough understanding of the features induced. This took slightly over 24 hours to run. At the final iteration it was exploring over 2000 possible features, which is clearly too many for such an early stage of the algorithm. Nonetheless, and notwithstanding the other restrictions, it managed to induce some plausible features. In particular, the feature `(O ((Y) (S ((V) (N ((N)))))))` which says roughly that a paragraph can consist of a sentence and a punctuation mark, where the sentence has a verb phrase , and a noun or noun phrase dominating a noun or noun phrase, seems to be a useful feature.

9.3 Time/Space Efficiency

It is very slow: on the most trivial data set it takes about 2 hours to generate the first 20 features, and this increases to over 24 hours for a small fraction of

Samples

- (V ((V ((V ((N ((V0 DIED) (V KISSED))) (V JOHN) (V ((N0 JOHN) (V ((V0 KISSED))))))) (V ((N0 ((N ((N JOHN) (V DIED))) (N MARY))))) (V0 ((N ((N ((N DIED))) (V GAVE) (V0 JOHN))) (N0 KISSED) (V ((N ((V ((V0 ((N ((N KISSED))) (V0 KISSED))))))))))))) (V ((V0 ((V KISSED))) (N0 JOHN)))))

- (N ((S ((V ((V0 MARY) (N KISSED))) (V ((V0 MARY)))))))

- (N ((V ((N ((V ((V0 ((V0 ((V KISSED)))))))))))))

- (V ((N KISSED) (N0 KISSED) (S ((N ((S MARY))) (S ((N0 JOHN))) (N MARY))) (V ((S ((V GAVE))))) (N ((V ((V ((V MARY))))) (V0 ((S MARY) (S MARY) (N0 JOHN) (N DIED))))) (V JOHN) (N0 DIED)))

- (N0 ((V0 ((N ((V ((V JOHN)))))))))

- (N ((N0 ((N0 ((N ((N ((N ((N ((N0 ((V0 ((V ((V0 ((V0 KISSED)))))))))))))))))))))))

- (V ((V ((S KISSED) (V JOHN))) (N0 ((N0 MARY) (S ((S JOHN))) (V0 ((N0 MARY) (V0 DIED))))) (V MARY)))

- (V0 ((V ((N0 DIED) (N ((N KISSED))))) (S ((S ((N0 KISSED))) (S GAVE) (V0 JOHN))) (N MARY) (V0 ((V KISSED) (S GAVE))) (N ((V ((V ((N0 ((N JOHN))) (S ((V0 GAVE))))) (S ((V0 GAVE) (N0 JOHN))) (N0 ((N ((V0 ((V ((S ((V ((N0 MARY))))))) (V ((N JOHN) (V0 GAVE))))) (V0 ((N0 DIED))))) (V ((N0 ((N ((N DIED))))))))) (V ((S ((N0 ((V MARY))))))) (V0 KISSED))) (V ((N MARY) (V ((V0 ((S DIED))) (N ((V ((V0 KISSED))))) (N JOHN))) (N0 ((N JOHN) (N0 ((N0 ((V ((V0 JOHN))) (S GAVE))))))))) (N0 ((V ((V MARY) (V GAVE) (N0 ((V DIED))))))) (V KISSED))) (V0 MARY) (V0 ((V0 ((S MARY) (S MARY)))))))

- (V ((V0 ((N0 ((N JOHN))))) (N0 MARY) (V ((S JOHN) (S JOHN) (V ((V0 KISSED) (N0 GAVE))))) (V0 ((V ((V ((S DIED))))))) (N KISSED)))

- (N ((N0 ((S ((V ((V0 ((V0 ((V0 GAVE)))))))))))))

Figure 9.7: 10 samples produced from the random field induced from the second data set.

feature	15	16	17	18	19	20
(S)	-23.7	-23.8	-23.9	-24.0	-27.3	-27.4
(S ((V)))	7.60	7.73	7.81	7.9	5.14	5.30
(S ((N) (V)))	10.0	10.3	10.4	10.7	10.6	10.9
(S ((V ((V0)))))	24.1	24.2	24.4	24.5	21.8	22.1
(S ((N) (V ((N)))))	-9.60	-9.35	-9.15	-8.85	-12.0	-11.8
(S ((N ((N0))) (V)))	21.1	21.4	21.6	22.0	27.3	27.6
(S ((V ((N) (V0)))))	24.7	25.0	25.2	25.4	26.4	26.8
(S ((N ((N0))) (V ((V0)))))	16.5	16.8	17.0	17.3	22.6	22.9
(S ((N) (V ((V0)))))	10.7	10.9	11.1	11.4	11.3	11.6
(S ((N ((N0))) (V ((N)))))	-3.77	-3.42	-3.21	-2.86	3.76	4.02
(S ((N ((N0))) (V ((N) (V0)))))	3.94	4.29	4.50	4.85	11.5	11.8
(S ((N) (V ((N) (V0)))))	4.80	5.05	5.26	5.59	2.40	2.70
(S ((V) (V ((N) (V0)))))	-10.4	-10.4	-10.4	-10.4	-1.66	-1.36
(S ((V) (V ((V0)))))	4.16	4.41	4.56	4.82	3.92	4.18
(N ((S ((V) (V ((V0)))))))	5.77	5.98	6.13	6.37	-1.64	-1.64
(S ((V ((V0))) (V ((N) (V0)))))	0	3.76	3.76	3.76	12.5	12.5
(N ((N0) (S ((V) (V ((V0)))))))	0	0	3.96	3.96	12.3	12.3
(S ((V ((N))) (V ((V0)))))	0	0	0	3.76	12.5	12.8
(S ((V ((N)))))	0	0	0	0	3.19	3.47
(N ((S ((V ((V0)))))))	0	0	0	0	0	4.32

Figure 9.8: The changing weights for the last 6 iterations.

feature	weight
(N)	1.240786E+0
(N ((N)))	3.15806E+0
(P ((N ((N)))))	5.51206E+0
(N ((A) (N)))	8.21859E+0
(P ((I) (N ((N)))))	1.406619E+1
(N ((P ((I) (N ((N)))))))	9.02847E+0
(N ((N) (P ((I) (N ((N)))))))	7.92184E+0
(S ((N ((N)))))	7.47884E+0
(S ((V) (N ((N)))))	6.98811E+0
(S ((V ((V))) (N ((N)))))	5.92124E+0
(P ((I) (N ((A) (N)))))	8.48305E+0
(N ((P)))	3.16384E+0
(N ((P ((I)))))	6.07672E+0
(V)	1.446695E+0
(O ((S ((V) (N ((N)))))))	4.55567E+0
(N ((N) (P ((I)))))	6.24001E+0
(N ((N) (P ((N) (I)))))	4.87432E+0
(N ((P ((N) (I)))))	4.78136E+0
(O ((Y) (S ((V) (N ((N)))))))	4.03750E+0
(V ((V)))	1.996822E+0

Figure 9.9: The features and weights induced from the SUSANNE corpus

the SUSANNE corpus.

I have not been able to do any theoretical work on what the theoretical efficiency of these algorithms is.

Chapter 10

Conclusion

10.1 Overview

It appears to work well on the simple sorts of generated test data that I experimented with. The field of NLP is however littered with the remains of approaches that worked well in toy domains but failed to scale up to realistic tasks. The approach does seem in principle able to scale up – there are some serious problems though with the computational demands of the approach and with the data requirements. The testing with this artificial data, generated from a small well-defined stochastic grammar is not ideal to show the benefits of this sort of model. These models are designed to cope with rather noisier data sets. To a certain extent then, the choice of this sort of model represents an implicit meta-theoretical commitment to a view of a language slightly at odds with the usual view of a well-defined set of well-formed sentences.

10.2 Random fields

Random fields do have a number of very positive qualities.

The attraction of the DDL approach is that it defines a class of rather *ad hoc* models that have a solid statistical grounding. It is not clear that much of the theoretical work on random fields will have any relevance to this area of application, but the use of a well understood statistical model must be considered an improvement on the hastily improvised 'scores' used in much NLP work. Moreover it appears that for disambiguating language a model that assigns a probability to each tree is preferable to one that merely assigns comparative values to a set of alternative analyses of a particular string.

10.2.1 Robustness

A major appeal of this approach is its robustness. This model is intrinsically robust – to the extent that it is only through additions to the model (allowing λ to take the value of $-\infty$) that you can exclude ill-formed structures completely. This is important not only for the usual reasons, but also because the corpus of data used to generate the model is likely to be edited to a higher standard than much of the probable input to it. It thus seems desirable that the model should have a bit of extra 'give' in it to accommodate this possible increased laxity.

10.2.2 Context Freeness

In the context of this model, generative capacity in general and context freeness in particular are not an issue. However there clearly are valid generalisations to be made about the sets of admissible structures[Sav87]. The current model allows features to have unbounded support[1], but in any given model, with a finite set of features, there will be an upper bound on the depth of the support. Thus the model will have a restricted 'domain of locality'. However if the model were augmented with features one could certainly have it perform the necessary stack operations to perform arbitrary computations. So in this way the model seems not to have any restrictions on its generative capacity considered rather loosely.

There is another less formal way in which the context freeness property can be considered and that is as being analogous to the homogeneity of the field. The field is homogeneous when at every node there are the same features with the same weights (because technically they are distinct identical features). This property seems to capture some pre-theoretic intuition that for example , a noun phrase can appear anywhere in a sentence. It is difficult to see though how such a vague idea could be shown to have any empirical content.

10.3 Fundamental Problems

10.3.1 Sparse data

It is clear that the requirement for parsed data is a serious impediment to the development of large-scale models of English with this technique. There are sufficiently large corpora of unparsed data to expect to find examples of almost all syntactic constructions – lexical data are obviously much more demanding. The available corpora of parsed data are much smaller and it does not appear likely that they would be large enough to make statistically viable generalisations about all English constructions.

Moreover much of the model will in some sense be devoted to modelling the labelling methodology of the research group that designed the corpus. In the particular case of the Susanne corpus, the labelling methodology is specifically not intended to be linguistically 'correct' but merely to provide a *lingua franca* for the interchange of data and to facilitate cooperative research. Sampson says ([Sam95] page 8):

> ...it is more important that the recommended analysis of any particular linguistic form should be predictable than that it should be theoretically justifiable.

Both of these observations seem to point to the necessity for the algorithm to cope with raw, unparsed text, and to construct the constituent structure trees directly. This is clearly a non-trivial task. Experiments with analogous ideas with stochastic context free grammars using the inside-outside algorithm

[1] the support of a feature is the set of nodes that determine its value

have been rather unsuccessful. However the clustering algorithms presented in [BDPdS+92], which are based on mutual information metrics, and have a close theoretical affinity to the current work, might provide a start for this.

10.3.2 Sampler convergence

The various types of MCMC samplers discussed here form part of a very active research area. New techniques are constantly under development, and there are numerous articles which help with acquiring the 'craft skill' of getting good performance – that is rapid and accurate convergence. It is important to note some rather fundamental limitations of these methods. If one wished to develop a large scale model of English or any other language with this technique, one would in the later steps of the algorithm have a very large and complex field, with one would assume several thousand features, interacting in subtle and unpredictable ways to produce some of the more linguistically interesting constructions that have been so carefully studied by theoretical linguists of a traditional bent. The rarity of these constructions will give an idea of how large the sample drawn must be to get a good estimate – arguably millions of trees. It is clear that this modelling technique, using these MCMC samplers cannot deal with those sorts of 'interesting' constructions.

The attraction of the stochastic estimation techniques is that they work very generally. In specific cases it may be possible to calculate them directly, or to use very specialised samplers. In any event it appears that the general approach will not be able to cope with estimating expectations with a model that captures a significant fraction of the structure of a natural language.

However that does not mean that this technique is useless – rather it may mean that the 'interesting' data arise out of the interaction between this sort of rather limited statistical model, and a more complex reasoning component that handles the semantic side of things. This means that the task is separated into a statistical component that is tractable, and the intractable 'AI-complete'[Gaz96] component. From an engineering point of view this is ideal, since it does appear that NLP is in general AI-complete.

Alternatively one might take the view that the 'interesting' data is not real but rather an artifact of the methodology of a certain type of theoretical linguist, and does not reflect the reality of language use, or of 'authentic' data.

10.4 Future work

10.4.1 Efficiency

Given the time constraints for this project, I wrote it in a language (Common Lisp) that while suited to rapid prototyping is comparatively inefficient in both time and space. Rewriting it in a more efficient language would allow the sampling algorithms to run much longer. In addition the multi-path sampling algorithms can obviously be parallelised efficiently.

10.4.2 Pruning features

It may be the case that some features, though significant in the early phases of the algorithm may be superseded by more complex ones and become effectively irrelevant. This would be evidenced by their weights coming close to zero. In this case it would be a simple modification to the algorithm to have a pruning phase that would remove these irrelevant features, thus simplifying the field and accelerating the algorithm.

10.4.3 Surface constraints

I would like to look at combining surface type constraints and these tree structures. Some parts of English syntax are clearly dependent on the surface context – often the phonological context as in the case of the a/an alternation. Random fields can deal perfectly well with the combination of these rather orthogonal constraints.

10.4.4 Complex categories

The labels of the SUSANNE corpus seem to have an implicit attribute-value structure that could be teased out. It would be interesting to map these labels to some simple AV structures, and augment the set of field features to handle these attributes.

10.4.5 Annealing parsers

It should be possible to hook this up to an annealing parser. The only difficulty is that mentioned earlier, namely that the overall number of nodes would vary during the course of the annealing parse. There are a number of ways around this. Either one could try to estimate the normalisation constants, or one could stipulate that all the trees must be binary branching. The problem with the latter approach is that it would be more difficult to capture generalisations about the order of daughters categories (the LP rules). This could possibly be overcome by creating some new class of primitive feature to handle these cases.

A better approach might be to change the way that the field is defined. I divided the infinite space of all trees into finite groups – this means there can be no analytic concerns about the convergence of the normalisation factors. However it might well be possible to add some way of penalising large chains of unbranching nodes, that would be high enough to overwhelm the potential gain from features. In any event this is a serious flaw with this method of normalising the field.

10.4.6 Annealing models

Sampson mentions the use of annealing in the model, not just in the parser – this is an interesting possibility. The paradigm that DDL use is deterministic – but of course choosing a model is exactly the sort of non-linear optimisation that stochastic optimisation techniques are good for. So the possibility exists of using the Kullback-liebler divergence of the model as the score function and using some form of stochastic optimisation, such as simulated annealing or genetic algorithms to guide the development, rather than the greedy algorithm they suggest which is roughly analogous to gradient descent.

Bibliography

[Abn95] Steven Abney. Stochastic attribute-value grammars. Technical report, University of Tubingen, 1995.

[Abn96] Steven Abney. Statistical methods and linguistics. In Judith Klavans and Philip Resnik, editors, *The Balancing Act*. MIT Press, 1996.

[BDPDP96] Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, March 1996.

[BDPdS+92] Peter F. Brown, Vincent J. Della Pietra, Peter V. de Souza, Jenifer C. Lai, and Robert Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, 1992.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[Bod93] Rens Bod. Using an annotated corpus as a stochastic grammar. In *Proceedings of EACL-93*, 1993.

[Bre95] Chris Brew. Stochastic HPSG. In *Proceedings of EACL-95*, 1995.

[BS94] J. M. Bernardo and A. F. M. Smith. *Bayesian Theory*. Wiley, 1994.

[CG95] Siddartha Chib and Edward Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49:327–335, 1995.

[Cha93] Eugene Charniak. *Statistical Language Learning*. MIT Press, 1993.

[Chu88] Kenneth W. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the second conference on applied natural language processing*, 1988.

[CW97] J. Carroll and D. Weir. Encoding frequency information in lexicalised grammars. In *Proceedings of the 5th ACL/SIGPARSE International Workshop on Parsing Technologies (IWPT-97)*, pages 8–17, Cambridge, MA, 1997.

[DPDPL97] Stephen A. Della Pietra, Vincent J. Della Pietra, and John Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:380–393, 1997.

[Eis94] Andreas Eisele. Towards probabilistic extensions of constraint-based grammars. Technical Report Deliverable R1.2.B, DYANA-2, Unknown, 1994.

[Gam97] Dani Gamerman. *Markov Chain Monte Carlo*. Chapman & Hall, 1997.

[Gaz96] G Gazdar. Paradigm merger in natural language processing. In Robin Milner and Ian Wand, editors, *Computing Tomorrow: Future Research Directions in Computer Science*. Cambridge University Press, 1996.

[GG84] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 721–741, November 1984.

[GKPS85] G. Gazdar, E. Klein, G. Pullum, and I. Sag. *Generalised Phrase Structure Grammar*. Basil Blackwell, 1985.

[GS96] W Gale and G Sampson. Good-Turing frequency estimation without tears. Cognitive Science Research Paper CSRP 407, University of Sussex, 1996.

[Jay83] E. T. Jaynes. *Papers on Probability, Statistics and Statistical Physics*. Kluwer, 1983.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 3rd. edition, 1997.

[MOS92] M. I. Miller and J. A. O' Sullivan. Entropies and combinatorics of random branching processes and context-free languages. *IEEE Transactions on Information Theory*, 1992.

[Nea98] Radford M. Neal. Annealed importance sampling. Technical Report TR 9805 Department of Statistics, University of Toronto, 1998.

[PTL93] Fernando Pereira, Natali Tishby, and Lillian Lee. Distributional clustering of English words. In *Proceedings of the 31st annual meeting of the Association for Computational Linguistics*, 1993.

[Sam95] G Sampson. *English for the Computer*. Clarendon Press, 1995.

[Sam96] G Sampson. *Evolutionary Language Understanding*. Cassell, 1996.

[Sav87] Walter J. Savitch, editor. *The Formal Complexity of Natural Language*. D. Reidel, 1987.

[SS94] Andreas Stolcke and Jonathan Segal. Precise n-gram probabilities from stochastic context-free grammars. In *Proceedings of the 32nd annual meeting of the Association for Computational Linguistics*, 1994.

[Ste90] Guy L. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

Appendix A

Notation

Term	Explanation
$G_q(\alpha, g)$	The gain of adding feature g with weight α
$\tilde{p}$	The reference distribution from the corpus
Ω	The set of all configurations – labelled trees normally
ω	A particular configuration
$q_{\alpha g}$	The random field formed by adding feature g with weight α to the field q
MCMC	Markov Chain Monte Carlo
$q[g]$	The expectation of a function g with respect to the distribution q
$f_{\#}(\omega)$	The sum of all the feature values for a particular tree or configuration
$a_{m,i}^{(k)}$	The mth coefficient , in the kth iteration for the ith feature in the re-estimation equation.
$\delta(i, j)$	This is 1 if i and j are equal and 0 otherwise
π	The probability density function that you want to sample from in the Metropolis-Hastings algorithm
$q(x, y)$	The proposal density for the Metropolis-Hastings algorithm: the probability that the next one is y given the current one is x
f_i	The ith feature of the field
λ_i	The weight of the ith feature: in the range $[-\infty, \infty)$
β_i	e^{λ_i} which will be in the range $[0, \infty)$
$\|\omega\|$	The number of nodes in a tree ω
$F_q(\omega)$	The unnormalised field weight of a tree with respect to the field q

Appendix B

Program Trace

This is the unedited output of the third run of the program, on the first set of data. The results of this run are collected in Figure 9.4. The output is rather verbose, but when you have a very slow program, I feel it is psychologically important that it gives a reasonable idea of its progress.

```
? (setf rf (induce-field fc 20))

 Iteration number 0
 Feature (B) gain 0.000415 weight -3.287E-2
 Feature (V) gain 0.000236 weight 2.4393E-2
 Feature (A) gain 0.000084 weight 1.4581E-2
 Feature (N) gain 0.000006 weight -3.698E-3
 Feature (S) gain 0.000004 weight -3.305E-3
 Optimum feature is (B)
 Re-estimating .....Done
 feature (B) weight -4.13770E-2
 Iteration number 1
 Feature (B ((B))) gain 0.132732 weight -1.000E+100
 Feature (B ((V))) gain 0.164050 weight -1.000E+100
 Feature (V) gain 0.000170 weight 2.0844E-2
 Feature (B ((A))) gain 0.161578 weight -1.000E+100
 Feature (A) gain 0.000002 weight 2.447E-3
 Feature (B ((N))) gain 0.180563 weight -1.000E+100
 Feature (N) gain 0.000438 weight -3.319E-2
 Feature (B ((S))) gain 0.169484 weight -1.000E+100
 Feature (S) gain 0.000065 weight -1.2901E-2
 Optimum feature is (B ((N)))
 Re-estimating ....
 Sampled expectation is zero -- bailing out of re-estimation.Done
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight 1.117834E-2
 Iteration number 2
 Feature (B ((B))) gain 0.144217 weight -1.000E+100
 Feature (B ((V))) gain 0.217161 weight -1.000E+100
 Feature (V) gain 0.003265 weight -8.869E-2
```

```
Feature (B ((A))) gain 0.184043 weight -1.000E+100
Feature (A) gain 0.001278 weight -5.606E-2
Feature (N) gain 0.003103 weight 8.852E-2
Feature (B ((S))) gain 0.181282 weight -1.000E+100
Feature (S) gain 0.000454 weight -3.400E-2
Optimum feature is (B ((V)))
Re-estimating .....Done
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight 1.244607E-1
Iteration number 3
Feature (B ((B))) gain 0.150242 weight -1.000E+100
Feature (V) gain 0.001845 weight 6.546E-2
Feature (B ((A))) gain 0.235722 weight -1.000E+100
Feature (A) gain 0.010623 weight -1.6021E-1
Feature (N) gain 0.000000 weight 4.002E-4
Feature (B ((S))) gain 0.235343 weight -1.000E+100
Feature (S) gain 0.004546 weight -1.066E-1
Optimum feature is (B ((A)))
Re-estimating .....Done
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight 2.67204E-1
Iteration number 4
Feature (B ((B))) gain 0.115411 weight -1.000E+100
Feature (V) gain 0.000075 weight -1.3169E-2
Feature (A) gain 0.001898 weight -6.703E-2
Feature (N) gain 0.004890 weight -1.0483E-1
Feature (B ((S))) gain 0.282761 weight -1.000E+100
Feature (S) gain 0.026240 weight -2.612E-1
Optimum feature is (B ((S)))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight 7.75299E-1
Iteration number 5
Feature (B ((B))) gain 0.320756 weight -1.000E+100
Feature (V) gain 0.003384 weight 9.050E-2
Feature (A) gain 0.000483 weight 3.404E-2
Feature (N) gain 0.002483 weight -7.466E-2
Feature (S) gain 0.000107 weight -1.6305E-2
Optimum feature is (B ((B)))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation.Done
```

```
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight 1.168516E+0
Iteration number 6
Feature (V) gain 0.003392 weight -9.172E-2
Feature (A) gain 0.005376 weight -1.1332E-1
Feature (N) gain 0.020307 weight -2.1961E-1
Feature (S) gain 0.012009 weight -1.7428E-1
Optimum feature is (N)
Re-estimating .....Done
feature (N) weight -4.77177E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight 2.07088E+0
Iteration number 7
Feature (N ((B))) gain 0.361621 weight 1.8135E+0
Feature (N ((V))) gain 0.140642 weight -1.000E+100
Feature (V) gain 0.006728 weight -1.2874E-1
Feature (N ((A))) gain 0.296066 weight 1.483E+0
Feature (A) gain 0.012092 weight -1.6985E-1
Feature (N ((N))) gain 0.097723 weight -1.000E+100
Feature (N ((S))) gain 0.173758 weight -1.000E+100
Feature (S) gain 0.012812 weight -1.7755E-1
Optimum feature is (N ((B)))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (N ((B))) weight 1.842716E+0
feature (N) weight -4.87904E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight 2.11511E+0
Iteration number 8
Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((V))) gain 0.153617 weight -1.000E+100
Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
Feature (V ((N ((B))))) gain 0.122733 weight -1.000E+100
Feature (N ((V) (B))) gain -0.000000 weight -1.000E+100
Feature (V) gain 0.000606 weight 3.907E-2
Feature (N ((A))) gain 0.374352 weight 1.7286E+0
Feature (N ((B ((A))))) gain -0.000000 weight -1.000E+100
```

```
 Feature (A ((N ((B))))) gain 0.153617 weight -1.000E+100
 Feature (N ((A) (B))) gain -0.000000 weight -1.000E+100
 Feature (A) gain 0.000389 weight -3.173E-2
 Feature (N ((N))) gain 0.147225 weight -1.000E+100
 Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N ((B))))) gain 0.093871 weight -1.000E+100
 Feature (N ((N) (B))) gain -0.000000 weight -1.000E+100
 Feature (N ((S))) gain 0.135476 weight -1.000E+100
 Feature (N ((B ((S))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((B))))) gain 0.425817 weight 1.9891E+0
 Feature (N ((S) (B))) gain -0.000000 weight -1.000E+100
 Feature (S) gain 0.001486 weight -6.138E-2
 Optimum feature is (S ((N ((B)))))
 Re-estimating ....
 Sampled expectation is zero -- bailing out of re-estimation
 Sampled expectation is zero -- bailing out of re-estimation.Done
 feature (S ((N ((B))))) weight 2.01244E+0
 feature (N ((B))) weight 1.778861E+0
 feature (N) weight -4.96462E-1
 feature (B ((B))) weight -1.00000E+100
 feature (B ((S))) weight -1.00000E+100
 feature (B ((A))) weight -1.00000E+100
 feature (B ((V))) weight -1.00000E+100
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight 2.14652E+0
 Iteration number 9
 Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((V))) gain 0.084469 weight -1.000E+100
 Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
 Feature (V ((N ((B))))) gain 0.061344 weight -1.000E+100
 Feature (N ((V) (B))) gain 0.001001 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain 0.001001 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain 0.130109 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.492
 i + o is 1 0.492
 Feature (S ((V) (N ((B))))) gain 1.988758 weight 4.042E+0
 Feature (V) gain 0.035009 weight 3.005E-1
 Feature (N ((A))) gain 0.501775 weight 2.098E+0
 Feature (N ((B ((A))))) gain -0.000000 weight -1.000E+100
 Feature (A ((N ((B))))) gain 0.100594 weight -1.000E+100
 Feature (N ((A) (B))) gain 0.001101 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain 0.001101 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain 0.134675 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (A) gain 0.020013 weight 2.2365E-1
 Feature (N ((N))) gain 0.085013 weight -1.000E+100
```

```
 Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N ((B))))) gain 0.048350 weight -1.000E+100
 Feature (N ((N) (B))) gain 0.002102 weight -1.000E+100
 Feature (S ((N ((N) (B))))) gain 0.002102 weight -1.000E+100
 Feature (N ((S ((N ((B))))))) gain 0.108365 weight -1.000E+100
 Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S))) gain 0.206041 weight -1.000E+100
 Feature (N ((B ((S))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S) (B))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((N ((B))))))) gain 0.138573 weight -1.000E+100
 Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S) gain 0.032892 weight -3.057E-1
 Optimum feature is (S ((V) (N ((B)))))
 Re-estimating ....
 Sampled expectation is zero -- bailing out of re-estimation.Done
 feature (S ((V) (N ((B))))) weight 4.04219E+0
 feature (S ((N ((B))))) weight 2.00506E+0
 feature (N ((B))) weight 1.706526E+0
 feature (N) weight -5.33351E-1
 feature (B ((B))) weight -1.00000E+100
 feature (B ((S))) weight -1.00000E+100
 feature (B ((A))) weight -1.00000E+100
 feature (B ((V))) weight -1.00000E+100
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight 2.15706E+0
 Iteration number 10
 Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((N ((B))))) gain 0.020203 weight -1.000E+100
 Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.441
 i + o is 1 0.441
 Feature (S ((V ((B))) (N ((B))))) gain 1.750956 weight 3.970E+0
 Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((V))) gain 0.078719 weight -1.000E+100
 Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
 Feature (V ((N ((B))))) gain 0.080126 weight -1.000E+100
 Feature (N ((V) (B))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain 0.118784 weight -1.000E+100
 Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (V) gain 0.021754 weight 2.2987E-1
 Feature (N ((A))) gain 0.532175 weight 2.238E+0
 Feature (N ((B ((A))))) gain 0.010050 weight -1.000E+100
 Feature (A ((N ((B))))) gain 0.077097 weight -1.000E+100
```

```
 Feature (N ((A) (B))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain -0.000000 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain 0.145373 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.051
 i + o is 1 0.051
 Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
 Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (A) gain 0.032626 weight 2.899E-1
 Feature (N ((N))) gain 0.076557 weight -1.000E+100
 Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N ((B))))) gain 0.039885 weight -1.000E+100
 Feature (N ((N) (B))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((N ((B))))))) gain 0.108031 weight -1.000E+100
 Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S))) gain 0.177453 weight -1.000E+100
 Feature (N ((B ((S))))) gain 0.010050 weight -1.000E+100
 Feature (N ((S) (B))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((N ((B))))))) gain 0.130223 weight -1.000E+100
 Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S) gain 0.031644 weight -3.134E-1
 Optimum feature is (S ((V ((B))) (N ((B)))))
 Re-estimating .....Done
 feature (S ((V ((B))) (N ((B))))) weight 7.28489E+0
 feature (S ((V) (N ((B))))) weight 1.517336E+1
 feature (S ((N ((B))))) weight 5.38812E+0
 feature (N ((B))) weight 3.30334E+0
 feature (N) weight 8.53613E-2
 feature (B ((B))) weight -1.00000E+100
 feature (B ((S))) weight -1.00000E+100
 feature (B ((A))) weight -1.00000E+100
 feature (B ((V))) weight -1.00000E+100
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight -6.52008E+0
 Iteration number 11
 Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
```

```
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((V))) gain 0.199305 weight -1.000E+100
Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
Feature (V ((N ((B))))) gain 0.000700 weight -1.000E+100
Feature (N ((V) (B))) gain 0.005917 weight -1.000E+100
Feature (S ((N ((V) (B))))) gain 0.003807 weight -1.000E+100
Feature (V ((S ((N ((B))))))) gain 0.023985 weight -1.000E+100
Feature (S ((V) (N ((V) (B))))) gain 0.003807 weight -1.000E+100
Feature (S ((V ((V))) (N ((B))))) gain 0.001301 weight -1.000E+100
Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (V) gain 0.010628 weight -1.5132E-1
Feature (N ((A))) gain 0.152707 weight 9.961E-1
Feature (N ((B ((A))))) gain -0.000000 weight -1.000E+100
Feature (A ((N ((B))))) gain 0.004912 weight -1.000E+100
Feature (N ((A) (B))) gain 0.006320 weight -1.000E+100
Feature (S ((N ((A) (B))))) gain 0.005013 weight -1.000E+100
Feature (A ((S ((N ((B))))))) gain 0.019795 weight -1.000E+100
Feature (S ((A) (N ((B))))) gain 0.002102 weight -1.000E+100
Feature (S ((V) (N ((A) (B))))) gain 0.005013 weight -1.000E+100
Feature (S ((V ((A))) (N ((B))))) gain 0.157074 weight 4.089E+0
Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V) (N ((B))))) gain 0.002102 weight -1.000E+100
Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A) gain 0.003361 weight -8.348E-2
Feature (N ((N))) gain 0.170907 weight -1.000E+100
Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
Feature (N ((N ((B))))) gain 0.000300 weight -1.000E+100
Feature (N ((N) (B))) gain 0.002303 weight -1.000E+100
Feature (S ((N ((N) (B))))) gain 0.002202 weight -1.000E+100
Feature (N ((S ((N ((B))))))) gain 0.029223 weight -1.000E+100
Feature (S ((N) (N ((B))))) gain 0.004108 weight -1.000E+100
Feature (S ((V) (N ((N) (B))))) gain 0.002202 weight -1.000E+100
Feature (S ((V ((N))) (N ((B))))) gain 0.006018 weight -1.000E+100
Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((B))))) gain 0.004108 weight -1.000E+100
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
```

```
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S))) gain 0.242072 weight -1.000E+100
Feature (N ((B ((S))))) gain 0.001802 weight -1.000E+100
Feature (N ((S) (B))) gain 0.005817 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain 0.005013 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain 0.019693 weight -1.000E+100
Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (B))))) gain 0.005013 weight -1.000E+100
Feature (S ((V ((S))) (N ((B))))) gain 0.001401 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S) gain 0.015078 weight -1.9919E-1
Optimum feature is (N ((S)))
Re-estimating .....Done
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 1.94912E+1
feature (S ((V) (N ((B))))) weight 2.17996E+1
feature (S ((N ((B))))) weight 1.039388E+1
feature (N ((B))) weight 4.29771E-1
feature (N) weight -2.78385E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight -1.464029E+1
Iteration number 12
Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((N ((B))))) gain 0.020203 weight -1.000E+100
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((V))) gain 0.274174 weight -1.000E+100
Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((V) (B))) gain 0.030459 weight -1.000E+100
Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
```

```
 Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V) gain 0.048121 weight -3.480E-1
 Feature (N ((A))) gain 0.180506 weight 1.1106E+0
 Feature (N ((B ((A))))) gain -0.000000 weight -1.000E+100
 Feature (A ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A) (B))) gain -0.000000 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain -0.000000 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.051
 i + o is 1 0.051
 Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
 Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (A) gain 0.020345 weight -2.0816E-1
 Feature (N ((N))) gain 0.155602 weight -1.000E+100
 Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N ((B))))) gain 0.010050 weight -1.000E+100
 Feature (N ((N) (B))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((B ((S))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S) (B))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
```

```
 Feature (S) gain 0.000089 weight 1.4971E-2
 Optimum feature is (N ((V)))
 Re-estimating ....
 Sampled expectation is zero -- bailing out of re-estimation
 Sampled expectation is zero -- bailing out of re-estimation
 Sampled expectation is zero -- bailing out of re-estimation.Done
 feature (N ((V))) weight -1.00000E+100
 feature (N ((S))) weight -1.00000E+100
 feature (S ((V ((B))) (N ((B))))) weight 1.94912E+1
 feature (S ((V) (N ((B))))) weight 2.17996E+1
 feature (S ((N ((B))))) weight 1.039388E+1
 feature (N ((B))) weight 8.24634E-1
 feature (N) weight -1.589183E-1
 feature (B ((B))) weight -1.00000E+100
 feature (B ((S))) weight -1.00000E+100
 feature (B ((A))) weight -1.00000E+100
 feature (B ((V))) weight -1.00000E+100
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight -1.449842E+1
 Iteration number 13
 Feature (N ((B ((B))))) gain 0.030459 weight -1.000E+100
 Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
 Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((V) (B))) gain 0.040822 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain 0.008637 weight -1.000E+100
 Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((B))))))) gain 0.008637 weight -1.000E+100
 Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V) gain 0.013196 weight -1.6378E-1
 Feature (N ((A))) gain 0.131462 weight 9.044E-1
 Feature (N ((B ((A))))) gain -0.000000 weight -1.000E+100
 Feature (A ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A) (B))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain -0.000000 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain 0.011465 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
```

```
i + o is 0 0.051
i + o is 1 0.051
Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
Feature (A ((S ((V) (N ((B))))))) gain 0.011465 weight -1.000E+100
Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A) gain 0.014859 weight -1.7197E-1
Feature (N ((N))) gain 0.166409 weight -1.000E+100
Feature (N ((B ((N))))) gain 0.010050 weight -1.000E+100
Feature (N ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (B))) gain 0.040822 weight -1.000E+100
Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((S))))) gain 0.010050 weight -1.000E+100
Feature (N ((S) (B))) gain 0.094311 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S) gain 0.003292 weight -9.382E-2
Optimum feature is (N ((N)))
Re-estimating .....Done
feature (N ((N))) weight -1.00000E+100
feature (N ((V))) weight -1.00000E+100
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
feature (S ((V) (N ((B))))) weight 4.34798E+1
feature (S ((N ((B))))) weight 3.20741E+1
feature (N ((B))) weight -2.69871E+0
feature (N) weight -6.10354E-1
feature (B ((B))) weight -1.00000E+100
```

```
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight -3.93195E+1
Iteration number 14
Feature (N ((B ((B))))) gain 0.010050 weight -1.000E+100
Feature (B ((N ((B))))) gain 0.010050 weight -1.000E+100
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((V) (B))) gain 0.051293 weight -1.000E+100
Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V))) (N ((B))))) gain 0.001501 weight -1.000E+100
Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (V) gain 0.015059 weight -1.8653E-1
Feature (N ((A))) gain 0.175542 weight 1.1589E+0
Feature (N ((B ((A))))) gain -0.000000 weight -1.000E+100
Feature (A ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A) (B))) gain 0.064859 weight -1.000E+100
Feature (S ((N ((A) (B))))) gain 0.012883 weight -1.000E+100
Feature (A ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (N ((B))))) gain 0.003606 weight -1.000E+100
Feature (S ((V) (N ((A) (B))))) gain 0.012883 weight -1.000E+100
Feature (S ((V ((A))) (N ((B))))) gain 0.072607 weight 2.350E+0
Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V) (N ((B))))) gain 0.003606 weight -1.000E+100
Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A) gain 0.037865 weight -2.939E-1
Feature (N ((B ((N))))) gain 0.010050 weight -1.000E+100
Feature (N ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (B))) gain 0.030459 weight -1.000E+100
Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((B))))) gain 0.003707 weight -1.000E+100
Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
```

```
Feature (S ((V ((N))) (N ((B))))) gain 0.000900 weight -1.000E+100
Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((B))))) gain 0.003707 weight -1.000E+100
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((S))))) gain 0.020203 weight -1.000E+100
Feature (N ((S) (B))) gain 0.020203 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S) gain 0.004736 weight -1.0004E-1
Optimum feature is (N ((A)))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (N ((A))) weight 1.257121E+0
feature (N ((N))) weight -1.00000E+100
feature (N ((V))) weight -1.00000E+100
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
feature (S ((V) (N ((B))))) weight 4.39945E+1
feature (S ((N ((B))))) weight 3.25888E+1
feature (N ((B))) weight -2.44979E+0
feature (N) weight -5.21832E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight -3.92405E+1
Iteration number 15
Feature (N ((B ((B))))) gain 0.010050 weight -1.000E+100
Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((B))))) gain 0.030459 weight -1.000E+100
Feature (B ((N ((A))))) gain -0.000000 weight -1.000E+100
```

```
Feature (N ((B ((V))))) gain -0.000000 weight -1.000E+100
Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((V) (B))) gain 0.020203 weight -1.000E+100
Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V))) (N ((B))))) gain 0.001001 weight -1.000E+100
Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((V))))) gain 0.066888 weight -1.000E+100
Feature (V ((N ((A))))) gain 0.060387 weight -1.000E+100
Feature (N ((V) (A))) gain 0.020203 weight -1.000E+100
Feature (V) gain 0.000110 weight 1.5421E-2
Feature (N ((B ((A))))) gain 0.010050 weight -1.000E+100
Feature (A ((N ((B))))) gain 0.010050 weight -1.000E+100
Feature (N ((A) (B))) gain 0.039885 weight -1.000E+100
Feature (S ((N ((A) (B))))) gain 0.009142 weight -1.000E+100
Feature (A ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (N ((B))))) gain 0.001701 weight -1.000E+100
Feature (S ((V) (N ((A) (B))))) gain 0.009142 weight -1.000E+100
Feature (S ((V ((A))) (N ((B))))) gain 0.268373 weight 6.287E+0
Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V) (N ((B))))) gain 0.001701 weight -1.000E+100
Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((A))))) gain 0.114289 weight -1.000E+100
Feature (A ((N ((A))))) gain 0.112833 weight -1.000E+100
Feature (A) gain 0.030907 weight -2.512E-1
Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
Feature (N ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (B))) gain 0.030459 weight -1.000E+100
Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((B))))) gain 0.001601 weight -1.000E+100
Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N))) (N ((B))))) gain 0.002403 weight -1.000E+100
Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((B))))) gain 0.001601 weight -1.000E+100
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((N))))) gain 0.046253 weight -1.000E+100
Feature (N ((N ((A))))) gain 0.010050 weight -1.000E+100
Feature (N ((N) (A))) gain 0.010050 weight -1.000E+100
```

```
Feature (N ((B ((S))))) gain 0.010050 weight -1.000E+100
Feature (N ((S) (B))) gain 0.072571 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (N ((B))))) gain 0.001201 weight -1.000E+100
Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((B))))) gain 0.000600 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((B))))) gain 0.001201 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((S))))) gain 0.099489 weight -1.000E+100
Feature (S ((N ((A))))) gain 0.521343 weight 2.212E+0
Feature (N ((S) (A))) gain 0.040822 weight -1.000E+100
Feature (S) gain 0.002294 weight -7.232E-2
Optimum feature is (S ((N ((A)))))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (S ((N ((A))))) weight 2.34434E+0
feature (N ((A))) weight 1.247702E+0
feature (N ((N))) weight -1.00000E+100
feature (N ((V))) weight -1.00000E+100
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
feature (S ((V) (N ((B))))) weight 4.44246E+1
feature (S ((N ((B))))) weight 3.30189E+1
feature (N ((B))) weight -2.29552E+0
feature (N) weight -5.24332E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight -3.91934E+1
Iteration number 16
Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((B))))) gain 0.030459 weight -1.000E+100
Feature (B ((N ((A))))) gain 0.010050 weight -1.000E+100
Feature (B ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((V))))) gain 0.010050 weight -1.000E+100
```

```
 Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((V) (B))) gain 0.010050 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((V))))) gain 0.151055 weight -1.000E+100
 Feature (V ((N ((A))))) gain 0.030665 weight -1.000E+100
 Feature (N ((V) (A))) gain 0.010050 weight -1.000E+100
 Feature (S ((N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((A))))))) gain 0.038845 weight -1.000E+100
 Feature (S ((V) (N ((A))))) gain 1.103298 weight 3.508E+0
 Feature (V) gain 0.031451 weight 2.939E-1
 Feature (N ((B ((A))))) gain 0.010050 weight -1.000E+100
 Feature (A ((N ((B))))) gain 0.010050 weight -1.000E+100
 Feature (N ((A) (B))) gain 0.061875 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain 0.030459 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain 0.030459 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.051
 i + o is 1 0.051
 Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
 Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((A))))) gain 0.124430 weight -1.000E+100
 Feature (A ((N ((A))))) gain 0.067316 weight -1.000E+100
 Feature (A ((S ((N ((A))))))) gain 0.064965 weight -1.000E+100
 Feature (S ((A) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (A) gain 0.089840 weight -5.077E-1
 Feature (N ((B ((N))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N) (B))) gain 0.010050 weight -1.000E+100
 Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
```

```
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((N))))) gain 0.035731 weight -1.000E+100
Feature (N ((N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (A))) gain 0.010050 weight -1.000E+100
Feature (S ((N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((S))))) gain 0.020203 weight -1.000E+100
Feature (N ((S) (B))) gain 0.040822 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((S))))) gain 0.145141 weight -1.000E+100
Feature (N ((S) (A))) gain 0.020203 weight -1.000E+100
Feature (S ((N ((S) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((A))))))) gain 0.045416 weight -1.000E+100
Feature (S ((S) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S) gain 0.049028 weight -4.000E-1
Optimum feature is (S ((V) (N ((A)))))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (S ((V) (N ((A))))) weight 3.50810E+0
feature (S ((N ((A))))) weight 2.43588E+0
feature (N ((A))) weight 1.226977E+0
feature (N ((N))) weight -1.00000E+100
feature (N ((V))) weight -1.00000E+100
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
feature (S ((V) (N ((B))))) weight 4.44246E+1
feature (S ((N ((B))))) weight 3.30189E+1
feature (N ((B))) weight -1.983405E+0
feature (N) weight -5.03494E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
```

```
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight -3.91024E+1
 Iteration number 17
 Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((B))))) gain 0.020203 weight -1.000E+100
 Feature (B ((N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((A))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.041
 i + o is 1 0.041
 Feature (S ((V ((B))) (N ((A))))) gain 0.108853 weight 2.655E+0
 Feature (B ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (N ((B ((V))))) gain 0.010050 weight -1.000E+100
 Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((V) (B))) gain 0.051293 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((V))))) gain 0.096952 weight -1.000E+100
 Feature (V ((N ((A))))) gain 0.035731 weight -1.000E+100
 Feature (N ((V) (A))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((A))))))) gain 0.038741 weight -1.000E+100
 Feature (S ((V) (N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (V) gain 0.008362 weight 1.490E-1
 Feature (N ((B ((A))))) gain 0.010050 weight -1.000E+100
 Feature (A ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A) (B))) gain 0.072571 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain 0.051293 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain 0.051293 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
```

```
 i + o is 0 0.051
 i + o is 1 0.051
 Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
 Feature (A ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((A))))) gain 0.098495 weight -1.000E+100
 Feature (A ((N ((A))))) gain 0.057735 weight -1.000E+100
 Feature (A ((S ((N ((A))))))) gain 0.054351 weight -1.000E+100
 Feature (S ((A) (N ((A))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.467
 i + o is 1 0.467
 Feature (S ((V ((A))) (N ((A))))) gain 1.871593 weight 4.008E+0
 Feature (A ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((A) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (A) gain 0.047227 weight -3.727E-1
 Feature (N ((B ((N))))) gain 0.020203 weight -1.000E+100
 Feature (N ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((N) (B))) gain 0.051293 weight -1.000E+100
 Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((N))))) gain 0.041343 weight -1.000E+100
 Feature (N ((N ((A))))) gain 0.020203 weight -1.000E+100
 Feature (N ((N) (A))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((N) (A))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((N) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((N))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((N) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (N ((B ((S))))) gain 0.010050 weight -1.000E+100
 Feature (N ((S) (B))) gain 0.040822 weight -1.000E+100
 Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
```

```
Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((S))))) gain 0.122959 weight -1.000E+100
Feature (N ((S) (A))) gain 0.020203 weight -1.000E+100
Feature (S ((N ((S) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((A))))))) gain 0.056994 weight -1.000E+100
Feature (S ((S) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S) gain 0.010287 weight -1.7548E-1
Optimum feature is (S ((V ((A))) (N ((A)))))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (S ((V ((A))) (N ((A))))) weight 4.00769E+0
feature (S ((V) (N ((A))))) weight 3.50810E+0
feature (S ((N ((A))))) weight 2.55651E+0
feature (N ((A))) weight 1.179046E+0
feature (N ((N))) weight -1.00000E+100
feature (N ((V))) weight -1.00000E+100
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
feature (S ((V) (N ((B))))) weight 4.50651E+1
feature (S ((N ((B))))) weight 3.36594E+1
feature (N ((B))) weight -1.750556E+0
feature (N) weight -5.11218E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight -3.90231E+1
Iteration number 18
Feature (N ((B ((B))))) gain 0.010050 weight -1.000E+100
Feature (B ((N ((B))))) gain 0.010050 weight -1.000E+100
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((B))))) gain 0.020203 weight -1.000E+100
```

```
 Feature (B ((N ((A))))) gain 0.010050 weight -1.000E+100
 Feature (B ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((A))))) gain -0.000000 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.041
 i + o is 1 0.041
 Feature (S ((V ((B))) (N ((A))))) gain 0.108853 weight 2.655E+0
 Feature (B ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A))) (N ((B) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (N ((B ((V))))) gain 0.010050 weight -1.000E+100
 Feature (V ((N ((B))))) gain 0.010050 weight -1.000E+100
 Feature (N ((V) (B))) gain 0.051293 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain 0.013592 weight -1.000E+100
 Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((B))))))) gain 0.013592 weight -1.000E+100
 Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((V))))) gain 0.084034 weight -1.000E+100
 Feature (V ((N ((A))))) gain 0.031181 weight -1.000E+100
 Feature (N ((V) (A))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((A))))))) gain 0.049611 weight -1.000E+100
 Feature (S ((V) (N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A))) (N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V) gain 0.020707 weight 2.2274E-1
 Feature (N ((B ((A))))) gain 0.010050 weight -1.000E+100
 Feature (A ((N ((B))))) gain 0.020203 weight -1.000E+100
 Feature (N ((A) (B))) gain 0.072571 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain 0.040822 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain 0.004410 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain 0.040822 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.051
 i + o is 1 0.051
 Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
 Feature (A ((S ((V) (N ((B))))))) gain 0.004410 weight -1.000E+100
```

```
Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((A))))) gain 0.086539 weight -1.000E+100
Feature (A ((N ((A))))) gain 0.072033 weight -1.000E+100
Feature (A ((S ((N ((A))))))) gain 0.087848 weight -1.000E+100
Feature (S ((A) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (A) gain 0.030589 weight -2.781E-1
Feature (N ((B ((N))))) gain 0.010050 weight -1.000E+100
Feature (N ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (B))) gain 0.030459 weight -1.000E+100
Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((N))))) gain 0.051399 weight -1.000E+100
Feature (N ((N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (A))) gain 0.030459 weight -1.000E+100
Feature (S ((N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A))) (N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((S))))) gain 0.010050 weight -1.000E+100
Feature (N ((S) (B))) gain 0.051293 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain 0.002102 weight -1.000E+100
Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((B))))))) gain 0.002102 weight -1.000E+100
```

```
Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((S))))) gain 0.155952 weight -1.000E+100
Feature (N ((S) (A))) gain 0.010050 weight -1.000E+100
Feature (S ((N ((S) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((A))))))) gain 0.043638 weight -1.000E+100
Feature (S ((S) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((S) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A))) (N ((S) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((S) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((S) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S) gain 0.007370 weight -1.4961E-1
Optimum feature is (N ((A ((S)))))
Re-estimating ....
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation
Sampled expectation is zero -- bailing out of re-estimation.Done
feature (N ((A ((S))))) weight -1.00000E+100
feature (S ((V ((A))) (N ((A))))) weight 4.00769E+0
feature (S ((V) (N ((A))))) weight 3.50810E+0
feature (S ((N ((A))))) weight 2.69997E+0
feature (N ((A))) weight 1.18309E+0
feature (N ((N))) weight -1.00000E+100
feature (N ((V))) weight -1.00000E+100
feature (N ((S))) weight -1.00000E+100
feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
feature (S ((V) (N ((B))))) weight 4.58443E+1
feature (S ((N ((B))))) weight 3.44386E+1
feature (N ((B))) weight -1.478162E+0
feature (N) weight -4.72415E-1
feature (B ((B))) weight -1.00000E+100
feature (B ((S))) weight -1.00000E+100
feature (B ((A))) weight -1.00000E+100
feature (B ((V))) weight -1.00000E+100
feature (B ((N))) weight -1.00000E+100
feature (B) weight -3.89562E+1
Iteration number 19
Feature (N ((B ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((B) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (B ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
```

```
 Feature (S ((B) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((B))))) gain 0.010050 weight -1.000E+100
 Feature (B ((N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (N ((A))))) gain 0.010050 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
 i + o is 0 0.041
 i + o is 1 0.041
 Feature (S ((V ((B))) (N ((A))))) gain 0.108853 weight 2.655E+0
 Feature (B ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A))) (N ((B) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (B ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((B) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (N ((B ((V))))) gain 0.020203 weight -1.000E+100
 Feature (V ((N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((V) (B))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((N ((B))))))) gain 0.001902 weight -1.000E+100
 Feature (S ((V) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((B))))))) gain 0.001902 weight -1.000E+100
 Feature (S ((V ((B))) (N ((V) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((A ((V))))) gain 0.097613 weight -1.000E+100
 Feature (V ((N ((A))))) gain 0.023269 weight -1.000E+100
 Feature (N ((V) (A))) gain 0.020203 weight -1.000E+100
 Feature (S ((N ((V) (A))))) gain 0.010050 weight -1.000E+100
 Feature (V ((S ((N ((A))))))) gain 0.087630 weight -1.000E+100
 Feature (S ((V) (N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A))) (N ((V) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((V) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (V) gain 0.020051 weight 2.496E-1
 Feature (N ((B ((A))))) gain 0.010050 weight -1.000E+100
 Feature (A ((N ((B))))) gain 0.010050 weight -1.000E+100
 Feature (N ((A) (B))) gain 0.174353 weight -1.000E+100
 Feature (S ((N ((A) (B))))) gain 0.105361 weight -1.000E+100
 Feature (A ((S ((N ((B))))))) gain 0.012174 weight -1.000E+100
 Feature (S ((A) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((A) (B))))) gain 0.105361 weight -1.000E+100
Warning - insoluble polynomial - padding coefficients
```

```
i + o is 0 0.051
i + o is 1 0.051
Feature (S ((V ((A))) (N ((B))))) gain 0.141193 weight 2.768E+0
Feature (A ((S ((V) (N ((B))))))) gain 0.012174 weight -1.000E+100
Feature (S ((A) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((A) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((A))))) gain 0.082730 weight -1.000E+100
Feature (A ((N ((A))))) gain 0.089050 weight -1.000E+100
Feature (A ((S ((N ((A))))))) gain 0.068279 weight -1.000E+100
Feature (S ((A) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (A ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((A) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (A) gain 0.082617 weight -4.777E-1
Feature (N ((B ((N))))) gain 0.010050 weight -1.000E+100
Feature (N ((N ((B))))) gain 0.010050 weight -1.000E+100
Feature (N ((N) (B))) gain 0.040822 weight -1.000E+100
Feature (S ((N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((B))) (N ((N) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
Feature (N ((A ((N))))) gain 0.092444 weight -1.000E+100
Feature (N ((N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((N) (A))) gain 0.020203 weight -1.000E+100
Feature (S ((N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V) (N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((A))) (N ((N) (A))))) gain -0.000000 weight -1.000E+100
Feature (S ((V ((N) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
Feature (S ((N) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
Feature (N ((B ((S))))) gain 0.020203 weight -1.000E+100
Feature (N ((S) (B))) gain 0.051293 weight -1.000E+100
Feature (S ((N ((S) (B))))) gain -0.000000 weight -1.000E+100
Feature (S ((S ((N ((B))))))) gain 0.006018 weight -1.000E+100
```

```
 Feature (S ((S) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V) (N ((B))))))) gain 0.006018 weight -1.000E+100
 Feature (S ((S) (V) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((B))) (N ((S) (B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S) (B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V ((B))) (N ((B))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (V ((B))) (N ((B))))) gain -0.000000 weight -1.000E+100
 Feature (N ((S) (A))) gain 0.010050 weight -1.000E+100
 Feature (S ((N ((S) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((N ((A))))))) gain 0.045207 weight -1.000E+100
 Feature (S ((S) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V) (N ((S) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (V) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((A))) (N ((S) (A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((V ((S) (A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S ((V ((A))) (N ((A))))))) gain -0.000000 weight -1.000E+100
 Feature (S ((S) (V ((A))) (N ((A))))) gain -0.000000 weight -1.000E+100
 Feature (S) gain 0.000354 weight -3.435E-2
 Optimum feature is (N ((A) (B)))
 Re-estimating ....
 Sampled expectation is zero -- bailing out of re-estimation
 Sampled expectation is zero -- bailing out of re-estimation
 Sampled expectation is zero -- bailing out of re-estimation.Done
 feature (N ((A) (B))) weight -1.00000E+100
 feature (N ((A ((S))))) weight -1.00000E+100
 feature (S ((V ((A))) (N ((A))))) weight 4.00769E+0
 feature (S ((V) (N ((A))))) weight 3.50810E+0
 feature (S ((N ((A))))) weight 2.80162E+0
 feature (N ((A))) weight 1.152883E+0
 feature (N ((N))) weight -1.00000E+100
 feature (N ((V))) weight -1.00000E+100
 feature (N ((S))) weight -1.00000E+100
 feature (S ((V ((B))) (N ((B))))) weight 3.33548E+1
 feature (S ((V) (N ((B))))) weight 4.62240E+1
 feature (S ((N ((B))))) weight 3.48183E+1
 feature (N ((B))) weight -1.355967E+0
 feature (N) weight -4.69389E-1
 feature (B ((B))) weight -1.00000E+100
 feature (B ((S))) weight -1.00000E+100
 feature (B ((A))) weight -1.00000E+100
 feature (B ((V))) weight -1.00000E+100
 feature (B ((N))) weight -1.00000E+100
 feature (B) weight -3.88940E+1
#<RANDOM-FIELD #x862986>
? (describe-field-tex rf)
```

```
 \begin{tabular}{|l|l|}
 \hline feature & weight \\ \hline\hline
  (B) & -3.88940E+1 \\
  (B ((N))) & -1.00000E+100 \\
  (B ((V))) & -1.00000E+100 \\
  (B ((A))) & -1.00000E+100 \\
  (B ((S))) & -1.00000E+100 \\
  (B ((B))) & -1.00000E+100 \\
  (N) & -4.69389E-1 \\
  (N ((B))) & -1.355967E+0 \\
  (S ((N ((B))))) & 3.48183E+1 \\
  (S ((V) (N ((B))))) & 4.62240E+1 \\
  (S ((V ((B))) (N ((B))))) & 3.33548E+1 \\
  (N ((S))) & -1.00000E+100 \\
  (N ((V))) & -1.00000E+100 \\
  (N ((N))) & -1.00000E+100 \\
  (N ((A))) & 1.152883E+0 \\
  (S ((N ((A))))) & 2.80162E+0 \\
  (S ((V) (N ((A))))) & 3.50810E+0 \\
  (S ((V ((A))) (N ((A))))) & 4.00769E+0 \\
  (N ((A ((S))))) & -1.00000E+100 \\
  (N ((A) (B))) & -1.00000E+100 \\
 \hline \end{tabular}
NIL
?
```

Appendix C

Code

C.1 load.lsp

```
;;;; load.lsp
;;;; this file just loads the others

;;; rebinds the hash key for my uk keyboard

(def-fred-command (:meta #\3)
  #'(lambda (w) (let ((*current-character* #\#))
                  (ed-self-insert w))))

;;; some Lisp implementations
;; don't allow (setf (nthcdr ...))

(defun setfnthcdr (n l c)
   (setf (cdr (nthcdr (1- n) l)) c))

(defvar  *code-directory* "Macintosh HD:alex:random:")
;(defvar  *code-directory* "C:\\random\\")

(defvar *code-suffix* ".lsp")

(defvar *file-list*
  '("tree"
    "corpus"
    "feature"
    "poly"
    "field"
    "expect"
    "re-est"
    "m-h"
    "susanne"
    "fake"
    "proposal"))
```

```
(dolist (file *file-list*)
   (format t "~%Loading file ~a" file)
   (load ( concatenate 'string
           *code-directory*
           file
           *code-suffix*))
   (format t " ... Ok."))
```

C.2 tree.lsp

```
;;; tree.lsp

(defclass tree ()
    ((label
      :accessor label
      :initarg :label)))

(defclass complex-tree (tree)
    ((daughters
      :accessor daughters
      :initarg :daughters)))

(defmethod d-length ((tree complex-tree))
   (length (daughters tree)))

(defclass leaf (tree)
    ((word
      :accessor word
      :initarg :word)))

(defmethod daughters ((l leaf))
  nil)

(defmethod d-length ((tree leaf))
   0)

;;; a tree list is always a 2-element list
;;; either
;;; (label word)
;;; (label (tree1 ... treen))
;;; eg
;;; (NP ((DET the) (N cat)))


(defun read-tree-from-list (l)
    (let ((label (first l))
          (tail (second l)))
```

```
       (if (listp tail)
               ;; we recurse
          (make-instance 'complex-tree
                 :label label
                 :daughters (mapcar #'read-tree-from-list tail))
          (make-instance 'leaf
            :label label
            :word tail))))

(defmethod write-tree-to-list ((tree complex-tree))
   (list (label tree)
     (mapcar #'write-tree-to-list (daughters tree))))

(defmethod write-tree-to-list ((tree leaf))
   (list (label tree)
     (word tree)))

;; this works but ...

(defmethod duplicate-tree ((tree tree))
  (read-tree-from-list (write-tree-to-list tree)))

;;; returns the total number
;;; of nodes in the tree

(defmethod number-of-nodes ((tree tree))
  1)

(defmethod number-of-nodes ((tree complex-tree))
  (+ 1 (reduce #'+ (mapcar #'number-of-nodes (daughters tree)))))


;;; returns the number of
;;; non-terminal nodes in the tree

(defmethod number-of-words ((tree tree))
  1)

(defmethod number-of-words ((tree complex-tree))
  (reduce #'+ (mapcar #'number-of-words (daughters tree))))

;;; returns the nth-node in the tree
;;; according to a fairly arbitrary
;;; numbering system


;; returns a number if there are not enough in
;; the tree

(defmethod nth-node ((tree leaf)
```

```
                     (n integer))
  (if (= n 0)
    tree
    (1- n)))

(defmethod nth-node ((tree complex-tree)
                     (n integer))
  (if (= n 0)
    tree
    (let ((nn (1- n)))
      (dolist (daughter (daughters tree))
        (setf nn (nth-node daughter nn))
        (unless (integerp nn)
          (return)))
      nn)))

;;; same thing but for leaf nodes

(defmethod nth-leaf ((tree leaf)
                     (n integer))
  (if (= n 0)
    tree
    (1- n)))

(defmethod nth-leaf ((tree complex-tree)
                     (n integer))
  (let ((nn n))
      (dolist (daughter (daughters tree))
        (setf nn (nth-leaf daughter nn))
        (unless (integerp nn)
          (return)))
      nn))
```

C.3 corpus.lsp

```
;;; the corpus
;;; this may be both a real one or a fake one

(defclass corpus ()
    ((list-of-trees :accessor list-of-trees
      :initarg :contents
       :initform nil)
     (size :accessor size
      :initarg :size)))

(defun new-corpus ()
   (make-instance 'corpus
     :contents (mapcar #'read-tree-from-list
            *corpus*)))
```

```
;;; this picks n random trees from the corpus
;;; and displays them

(defmethod sample-from-corpus ((c corpus)
                               n)
   (let ((l (length (list-of-trees c))))
      (dotimes (i n)
        (let ((m (abs (random l))))
          (format t "~% sample number ~a ~a" m
           (write-tree-to-list (nth m (list-of-trees c))))))))

;;; this displays the first n trees
;;; from the corpus

(defmethod display-corpus ((c corpus)
                               n)
      (dotimes (i n)
         (format t "~% sample ~a"
           (write-tree-to-list (nth i (list-of-trees c))))))
```

C.4 feature.lsp

```
;;;; feature.lsp

(defclass feature ()
    ())

(defclass atomic-feature (feature)
    ())

;;; this is the only atomic feature used

(defclass label-feature (atomic-feature)
    ((label :accessor label
       :initarg :label)))

(defclass word-feature (atomic-feature)
    ((word :accessor word
       :initarg :word)))

(defclass word-label-feature (word-feature label-feature)
    ())

;;; this is the class of complex features

(defclass feature-tree (feature)
    ((head-label :accessor head-label
       :initarg :head-label)
```

```
     (daughter-list
       :accessor daughter-list
     :initarg :daughter-list)
     (order-type :accessor order-type
      :initarg :order-type)))

(defconstant *initial* 0)
(defconstant *medial* 1)
(defconstant *terminal* 2)
(defconstant *complete* 3)
(defconstant *unordered* 4)

(defmethod write-feature-to-list ((feature label-feature))
  (list (label feature)))

(defmethod write-feature-to-list ((feature feature-tree))
  (list (head-label feature)
        (mapcar #'write-feature-to-list
            (daughter-list feature))))

;;; this function is whether the feature
;;; is present at this particular point in the tree
;;; ie non-recursive.

;; default is it fails

(defmethod matches ((f feature)
                    (tree tree))
   nil)

;;; we use equal since the labels are strings

(defmethod matches ((f label-feature)
                    (tree tree))
   (equal (label f)
     (label tree)))

;;; only leafs can have words.

(defmethod matches ((f word-feature)
                    (l leaf))
   (equal (word f)
     (word l)))

(defmethod matches ((f word-label-feature)
                    (l leaf))
   (and
        (equal (word f) (word l))
        (equal (label f) (label l))))
```

```lisp
;;; the complex features can only match
;;; complex trees

(defmethod matches ((f feature-tree)
                    (tree complex-tree))

   (and (head-label f)
        (equal (head-label f)
               (label tree))
        (match-daughters
         (order-type f)
         (daughter-list f)
         (daughters tree))))

;;; initial

(defun every-match (features daughters)
   (dolist (f features t)
      (unless (and daughters
                   (matches f (first daughters)))
         (return))
      (pop daughters)))


(defmethod match-daughters ((type (eql 0))
                            features
                            daughters)
   (every-match  features daughters))

;;; medial


(defmethod match-daughters ((type (eql 1))
                            features
                            daughters)
;; we need to try each one
   (or (every-match features daughters)
       (and daughters
            (match-daughters features (cdr daughters)))))

;;; terminal

(defmethod match-daughters ((type (eql 2))
                            features
                            daughters)
   (let ((lf (length features))
         (ld (length daughters)))
      (and (>= ld lf)
           (every-match features
                        (nthcdr (- ld lf) daughters)))))
```

```
;;; complete

(defmethod match-daughters ((type (eql 3))
                            features
                            daughters)
   (and (= (length features)
           (length daughters))
        (every-match features daughters)))

;;; unordered-incomplete

(defmethod match-daughters ((type (eql 4))
                            features
                            daughters)
   ;;; this would be so easy in prolog
   ;;; but this ain't prolog
   (match-daughters-2
    features
    daughters
    nil
    ))

(defmethod match-daughters-2 (features
                               daughters
                               done-daughters)
   (if (null features)
      t
      (dolist (daughter daughters nil)
         ;; try to match the first daughter
         (when (and (not (member daughter done-daughters))
                    (matches
                     (first features)
                      daughter)
                    (match-daughters-2
                     (rest features)
                     daughters
                     (cons daughter done-daughters)))
            (return t)))))

;;; in general it will only happen once

(defmethod all-matches ((f feature)
                        (tree tree))
   (if (matches f tree)
      1 0))

(defmethod all-matches ((f feature)
                        (tree complex-tree))
   (+ (if (matches f tree)
```

```
         1 0)
      (reduce #'+ (mapcar #'(lambda (d) (all-matches f d))
                    (daughters tree)))))

;;; this test for equality
;;; so we don't get duplicates in
;;; the candidate list

(defmethod equal-feature ((f feature)
                          (g feature))
  nil)

(defmethod equal-feature ((f label-feature)
                          (g label-feature))
  (equal (label f)
         (label g)))

(defmethod equal-feature ((f word-feature)
                          (g word-feature))
  (equal (word f)
         (word g)))

(defmethod equal-feature ((f word-label-feature)
                          (g word-label-feature))
  (and (equal (word f)
              (word g))
       (equal (label f)
              (label g))))

;;; now for feature-trees

(defmethod equal-feature ((f feature-tree)
                          (g feature-tree))
  (cond ((equal (order-type f)
                *unordered*)
         (and (equal (head-label f)
                     (head-label g))
              (equal (order-type f)
                     (order-type g))
              (= (length (daughter-list f))
                 (length (daughter-list g)))
              ;; we need to check for permutations when
              (permute-equal (daughter-list f)
                             (daughter-list g))))
        (t
  (and (equal (head-label f)
              (head-label g))
       (equal (order-type f)
              (order-type g))
       (= (length (daughter-list f))
```

```
          (length (daughter-list g)))
       (every  #'equal-feature
               (daughter-list f)
               (daughter-list g))))))

;;; this is wrong as it says
;;; (a b b) is the same as (a a b)
;;; but it will do for now

(defmethod permute-equal ((fs list)
                          (gs list))
  ;; this returns true if there is
  ;; a permutation of fs which is
  ;; feature-equal to gs
  (every #'(lambda (f)
             (some #'(lambda (g) (equal-feature f g))
                   gs))
         fs))
```

C.5 poly.lsp

```
;;;; polymonial.lsp
;;; this has the code for the numerical solution of the
;;; polynomials

;;; this is just the newton method
;;; or newton-raphson

(defvar *max-polynomial-iterations* 200)

(defvar *min-polynomial-error* 1E-13)

(defvar *pad-value* 1)

;;; this needs to be improved so that it returns a positive root

(defmethod solve-polynomial ((polynomial array))
   (let ((x 2)
         (old-x 0)
         (grad-x 0)
         (f-x 0)
         (done nil)
         (start 2)
         (order (order polynomial)))
     (cond ((= 0 order)
            (break "constant polynomial"))
           ((= 1 order)
            ;; flat polynomial
            (/ (* -1 (aref polynomial 0))
```

```
               (aref polynomial 1)))
           ;; there should be the clause for
           ;; analytic solutions for n=2 and 3
           ;; but ...
           (t
            ;;; normal
      (dotimes (i *max-polynomial-iterations* x)
        ; (format t "~% loop ~a ~a ~a" i old-x x)
         ;(format t "~% error ~a " (abs (- x old-x)))
         (when (< (abs (- x old-x)) *min-polynomial-error*)
           (setf done t)
           (return x))
         (setf old-x x)
         (setf f-x (evaluate-polynomial polynomial x))
         (setf grad-x (evaluate-derivative polynomial x))
         (when (= grad-x 0)
            (format t "~% oops polynomial is flat ...")
            (break))
         ;(format t "~% x ~a y ~a" x f-x)
         (setf x (- x (float (/  f-x grad-x))))
         (when (< x 0)
           (format t "~% Error - x has gone negative")
           (incf start)
           (setf x start) ))
      (if done x (break "didn't converge"))))))


(defmethod evaluate-polynomial ((polynomial array)
                                x)
   (let* ((n (length polynomial))
         (answer 0)
         (k (- n 1)))
      (dotimes (i n answer)

         (setf answer
               (+
                  (* answer x)
                  (aref polynomial k)))
         (decf k))))

(defmethod evaluate-derivative ((polynomial array)
                                x)
   (let* ((n (length polynomial))
         (answer 0)
         (k (- n 1)))
      (dotimes (i (- n 1) answer)

         (setf answer
               (+
                  (* answer x)
```

```
                  (* k (aref polynomial k))))
         (decf k))))


(defmethod order ((polynomial array))
   (let ((n (length polynomial)))
      (loop
        (decf n)
        (unless (= 0
                   (aref polynomial n))
           (return n))
        (if (= n 0)
           (return 0)))))

;;; this just pads out the coefficients because
;;; the sample may be too small to show up
;;; the high frequencies.

(defmethod pad-polynomial ((polynomial array)
                           n)
   (let ((l (length polynomial)))
      (if (> n l)
         (break "can't pad the array"))
      (let ((c (ceiling n)))
         (loop
           (when (= c 0)
              (return))
           (if (= (aref polynomial c) 0)
              (setf (aref polynomial c)
                    *pad-value*)
              (return))
           (decf c)))))
```

C.6 field.lsp

```
;;;; top level functions for random field induction algorithm

(defclass random-field ()
    ((feature-weight-list :accessor feature-weight-list
       :initarg :fwl)
     (null-field :accessor null-field
      :initarg :null-field)
     (feature-cache :accessor feature-cache
                    :initform nil)))

(defmethod describe-field ((rf random-field))
  (dolist (fw (feature-weight-list rf))
    (format t "~% feature ~a weight ~,5E"
            (write-feature-to-list (feature fw))
```

```
            (weight fw))))

;;; this is just so I can paste it straight into
;;; a latex document

(defmethod describe-field-tex ((rf random-field))
  (format t "~% \\begin{tabular}{|l|l|}")
  (format t "~% \\hline feature & weight \\\\ \\hline\\hline")
  (dolist (fw (reverse (feature-weight-list rf)))
    (format t "~%  ~a & ~,5E \\\\"
            (write-feature-to-list (feature fw))
            (weight fw)))
  (format t "~% \\hline \\end{tabular}"))

;;; this is the null field

(defclass default-field ()
     ((label-array
       :accessor label-array
       :initarg :labels)
     (words
      :accessor words
      :initarg :words)
     (lengths
      :accessor lengths
      :initarg :lengths)
      (length-quotient :accessor length-quotient)))


;;; this just returns a new field that will
;;; have the right prior distribution of lengths etc.
;;; the right set of labels and so on

(defmethod create-field ((c corpus))
   (make-instance 'random-field
     :null-field (create-null-field c)
     :fwl '()))

(defmethod create-null-field ((c corpus))
   (let ((answer (make-instance 'default-field
                  :labels (collect-labels c)
                  :words (collect-words c)
                  :lengths (collect-lengths c))))
      (setf (length-quotient answer)
            (length-sum (lengths answer)))
      answer))

(defmethod collect-words ((c corpus))
   (let ((word-list ()))
      (dolist (tree (list-of-trees c))
```

```
      (setf word-list (add-words tree word-list)))
   (make-array (length word-list)
         :initial-contents word-list)))

(defmethod add-words ((tree complex-tree)
                      word-list)
   (dolist (daughter (daughters tree))
      (setf word-list (add-words daughter word-list)))
   word-list)

(defmethod add-words ((leaf leaf)
                      word-list)
   (pushnew (word leaf) word-list :test #'equal)
   word-list)

(defmethod collect-labels ((c corpus))
   (let ((label-list ()))
      (dolist (tree (list-of-trees c))
         (setf label-list (add-labels tree label-list)))
      (make-array (length label-list)
        :initial-contents label-list)))

(defmethod add-labels ((tree complex-tree)
                       label-list)
   (pushnew (label tree) label-list :test #'equal)
   (dolist (daughter (daughters tree))
      (setf label-list (add-labels daughter label-list)))
   label-list)

(defmethod add-labels ((leaf leaf)
                       label-list)
   (pushnew (label leaf)
     label-list
     :test #'equal)
   label-list)


;;; this one is slightly different
;;; we need to count up the number of lengths
;;; this doesn't need to be efficient.
;;; this will return a string of numbers like
;;; (100 20 10 1 )
;;; 100 of length zero, 20 of length 1 and so on.

(defvar *max-tree-length* 200)

(defmethod collect-lengths ((c corpus))
   (let ((length-array (make-array (+ 1 *max-tree-length*)
                        :initial-element 0)))
     (dolist (tree (list-of-trees c))
```

```
       (incf (aref length-array (number-of-nodes tree))))
     length-array))

(defmethod add-lengths ((tree complex-tree)
                         length-array)
   (add-length (d-length tree) length-array)
   (dolist (daughter (daughters tree))
      (add-lengths daughter length-array))
   length-array)

(defmethod add-lengths ((leaf leaf)
                       length-array)
  (add-length 0 length-array))

;;; this is destructive but who cares

(defun add-length (length length-array)
   (when (> length *max-tree-length*)
      (break "too long a tree"))
   (incf (aref length-array length)))


(defmethod length-sum ((length-array array))
   (let ((answer 0))
      (dotimes (i (length length-array))
         (setf answer (+ answer (aref length-array i))))
      answer))

;;; returns a random sample
;;; in the form of a corpus object
;;; using the Metropolis-Hastings algorithm

(defmethod field-weight ((tree tree)
                         (rf random-field))
   ;;; this returns the unnormalised field weight
   ;;; of tree wrt the rf random-field
   (let ((w 1))
      (dolist (fw (feature-weight-list rf) w)
         (setf w
               (* w (exp (* (weight fw)
                            (all-matches
                              (feature fw)
                              tree))))))))

;;; you can get overruns with the previous method
;;; this is also less expensive computationally

(defmethod log-field-weight ((tree tree)
                         (rf random-field))
   ;;; this returns the unnormalised field weight
```

```
   ;;; of tree wrt the rf random-field
   (let ((w 0))
      (dolist (fw (feature-weight-list rf) w)
         (setf w
               (+ w (* (weight fw)
                       (all-matches
                        (feature fw)
                        tree)))))))

(defmethod choose-n-trees (n (df default-field))
   (let ((answer ()))
      (dotimes (i n answer)
         (push (sample-once df) answer))))

;;; this samples a length from the prior distribution

(defmethod choose-length ((df default-field))
    (let ((r (1+ (random (length-quotient df))))
         (l 0)
         (la (lengths df)))
      (loop while (> r (aref la l))
        do (setf r (- r (aref la l)))
        do (setf l (+ l 1)))
      l))

(defmethod choose-label ((df default-field))
  (aref (label-array df)
        (random (length (label-array df)))))

(defmethod choose-word ((df default-field))
   (aref (words df)
        (random (length (words df)))))

(defmethod choose-node-num ((df default-field))
  (choose-length df))

(defclass feature-weight ()
    ((feature
       :accessor feature
       :initarg :feature)
     (weight
       :accessor weight
       :initarg :weight)))
```

C.7 expect.lsp

```
;;; this totals up all occurrences in the corpus
;;; and returns an expectation,
;;; or rather the list of things.
```

```
(defvar *max-feature-occurrence* 1000)

(defvar *sample-size* 2000)

(defvar *default-g-k* 100)

(defmethod calculate-expectation ((f feature)
                                  (c corpus))
   (let ((coefficients (make-array *max-feature-occurrence*
                         :initial-element 0))
         (trace 0))
      (dolist (tree (list-of-trees c) coefficients)
        (let ((num (all-matches f tree)))
          (incf trace)
          (when (= trace 100)
            ;(write "*")
            (setf trace 0))
         ; (when (> num 10)
          ;  (format t "~% ~a ~a" num (write-tree-to-list tree)))
         (incf (aref coefficients
                num))))))

(defmethod mean-feature ((f feature)
                         (c corpus))
   (let ((m 0)
         (n 0)
         (k 0)
         (coefficients (calculate-expectation f c)))
      (dotimes (i *max-feature-occurrence*)
         (setf k (aref coefficients i))
         (setf n (+ n k))
         (setf m (+ m
                    (* k i))))
      (float (/ m n))))

;;; this function returns the optimal beta
;;; for the feature

(defmethod find-optimum ((candidate feature)
                         (field-corpus corpus)
                         (sample-corpus corpus))
   (let ((sample-exp (mean-feature candidate sample-corpus))
         (field-exp (calculate-expectation candidate
                                           field-corpus))
         (polynomial (make-array *max-feature-occurrence* )))
     (cond ( (= 0 sample-exp)
            (list -1E100
                  0
```

```
                  (* -1 (log (/ (aref field-exp 0)
                                 (length-sum field-exp))))))
         (t
          ;; we do it the hard way
      ;;; now sample-exp is p tilde[g]
      ;;; field exp is (g_0 g_1 ... g_N)
     ;(format t "~% field-exp ~a " field-exp)
      (dotimes (k *max-feature-occurrence*)
         (setf (aref polynomial k)
               (* (aref field-exp k)
                  (- k sample-exp))))
      ;;; check to see whether soluble
      (let ((o (order polynomial)))
;        (format t"~% p[g] ~a N ~a" sample-exp o)
        (when (< o
                 sample-exp)
          (format t "~%Warning -")
          (format t "insoluble polynomial-padding coefficients")
          (dotimes (i (- (1+ (ceiling sample-exp))
                         o))
            (format t "~% i + o is ~a ~a" (+ i o) sample-exp)
            (setf (aref polynomial (+ i o 1))
                  (- (+ i o 1) sample-exp)))))
      (let* ((beta (solve-polynomial polynomial))
             (alpha (log beta)))
        (when (< beta 0)
          (break "polynomial should have been padded better"))
         (list alpha beta (gain alpha sample-exp field-exp)))))))


;;; returns the gain

(defmethod gain (alpha
                 sample-exp
                 (g-k-coefficients array))
   (let  ((gain 0)
          (n (length g-k-coefficients))
          (sum 0)
          (g-k 0))
      ;(setf alpha 0)
      (dotimes (k (1- n))
         (setf g-k (aref g-k-coefficients k))
         (setf sum (+ sum g-k))
         (when (> g-k 0)
           (setf gain (+ gain (* g-k (exp (* alpha k)))))))
      (setf gain (float (/ gain sum)))
      (setf gain (- (* alpha sample-exp) (log gain)))
      gain))
```

```
(defmethod choose-optimum-feature ((feature-list list)
                                   (rf random-field)
                                   (c corpus))
   (let ((field-corpus (sample-3  rf 100 1000 100))
         (max-gain '(0 0 0))
         (new-gain 0)
         (feature nil))
      (dolist (f feature-list)
         (setf new-gain (find-optimum f field-corpus c))
         (format t "~% Feature ~a gain ~,6F weight ~,3E"
                 (write-feature-to-list f)
                 (third new-gain)
                 (first new-gain))
         (when (> (third new-gain) (third max-gain))
            (setf max-gain new-gain
                  feature f)))
      (format t "~% Optimum feature is ~a"
              (write-feature-to-list feature))
      (make-instance 'feature-weight
        :feature feature
        :weight (first max-gain))))


(defmethod induce-feature ((rf random-field)
                           (c corpus))
   (let* ((candidates (candidate-features rf))
          (new-feature (choose-optimum-feature
                        candidates
                        rf
                        c)))
      (push new-feature (feature-weight-list rf))
      (re-estimate-importance rf c)))


(defvar *rf*)

(defmethod induce-field ((c corpus)
                         (n integer))
  (let ((rf (create-field c)))
    (setf *rf* rf)
    (dotimes (i n)
      (format t "~% Iteration number ~a" i)
      (induce-feature rf c)
      (describe-field rf))
    rf))
```

C.8 re-est.lsp

```
;;; re-est.lsp
```

```
;;; this does the iterative re-estimation
;;; this is the most computationally intensive part
;;; the convergence of the samplers is the big problem


(defvar *re-estimation-max-iterations* 500)
(defvar *re-estimation-error* 1E-5)
(defvar *re-estimation-burn-in* 1000)
(defvar *re-estimation-paths* 100)
(defvar *re-estimation-sample-size* 100)

;;; given a set of features
;;; we solve the same kind of polynomial equations
;;; and adjust them all
;;; the equations are slightly different

(defmethod max-hash-f ((c corpus)
                       (rf random-field))
   (let ((max 0)
         (features (mapcar #'feature (feature-weight-list rf))))
      (dolist (tree (list-of-trees c) max)
         (let ((n 0))
            (dolist (f features)
               (incf n (all-matches f tree)))
            (setf max (max n max))))))

;;;; this one does the iterative reestimation
;;;; using importance sampling on the same corpus
;;;; this means it is likely to converge in the case
;;;; when the MCMC samplers would not

(defmethod re-estimate-importance ((rf random-field)
                                   (c corpus))
  (format t "~% Re-estimating ....")
   (let* ((problem nil)
          (n (length (feature-weight-list rf)))
          (f-expectations
           (mapcar #'(lambda (f)
                       (mean-feature (feature f)
                        c))
             (feature-weight-list rf)))
          (feature-array
           (sample-and-discard rf
            *re-estimation-paths*
            *re-estimation-burn-in*
            *re-estimation-sample-size*))
          (number-of-samples
           (* *re-estimation-paths*
              *re-estimation-sample-size*))
          (initial-field-weight-array
```

```
          (calculate-field-weight-array rf feature-array))
         (max-hash-array (max-hash-array feature-array))
         (max-hash-f (max-of-array max-hash-array))
         (a-m-i (make-array (list (1+ max-hash-f) n)
                    :initial-element 0))
         (polynomial (make-array max-hash-f
                         :initial-element 0)))
    ;; the initial importance weights obviously are 1
    ;; set the a-m-i array
    ;; we only need to do this once

     ;; if nil is returned it means it did not converge in time
     (dotimes (k *re-estimation-max-iterations* nil)
        ;; at each iteration we calculate the
       (let* ((current-field-weight-array
(calculate-field-weight-array rf feature-array))
              (expectations f-expectations))
         (dotimes (i n)
           (dotimes (j max-hash-f)
             (setf (aref a-m-i j i) 0)))
         (dotimes (i n)
           (setf (aref a-m-i 0 i)
                 (* -1.0
                    (pop expectations))))
        ; (format t "~% Reestimating IS - iteration ~a" k)
         ;(write current-field-weight-array)
         ;(format t "~% step 1")
         (dotimes (sample-index number-of-samples)
            (let* ((f-hash (aref max-hash-array sample-index))
                   (current (aref current-field-weight-array
    sample-index))
                   (initial (aref initial-field-weight-array
                            sample-index))
                   (factor  (cond ((and (= 0 current)
                                        (= 0 initial))
                                   (/ 1 number-of-samples))
                                  ((not (= 0 initial))
                                   (/ current
                                      initial
                                      number-of-samples))
                                  (t
                                   (break "division by 0")))))

              (when (> f-hash 0)
                (dotimes (feature-index n)
                  (let ((feature-value (aref
                                         feature-array
                                         feature-index
                                         sample-index)))
                    (incf (aref a-m-i f-hash feature-index)
```

```
                        (* factor feature-value)))))))
          ;(format t "~% step 2")
          (let ((delta 0)
                (fwl (feature-weight-list rf)))

            (dotimes (i n)
               ;; copy polynomial coefficients to polynomial
               ;; and solve them
               (dotimes (m max-hash-f)
                  (setf (aref polynomial m)
                        (aref a-m-i m i)))
               ;; solve it
               (let ((fw (pop fwl)))
                 (cond ((= 0 (order polynomial))
                        ;; then it is flat and insoluble
                        (format t "~% Sampled expectation is zero")
                        (format t " -- bailing out of re-estimation")
                        (setf problem t))
                       ((> (weight fw) -100)
                        (let* ((raw-solution
                                (solve-polynomial polynomial))
                               (solution (log raw-solution)))
                          (when (<=  raw-solution 0)
                            (break t " Negative solution error"))
                          (when (> (weight fw) -1000)
                            (incf delta (abs  solution)))
                          (setf (weight fw)
                                (+ (weight fw)
                                   solution)))))))
            (when (or problem
                      (< delta *re-estimation-error*))
                  (return t))))))
   (format t ".Done"))

;; we don't need the actual trees just the weights.

(defmethod calculate-field-weight-array ((rf random-field)
                                         (feature-array array))
  (let* ((dim (array-dimensions feature-array))
         (answer (make-array (second dim)
                             :initial-element 1)))
    (dotimes (sample-index (second dim) answer)
      (let ((feature-index 0))
        (dolist (fw (feature-weight-list rf))
          (setf (aref answer sample-index)
                (* (aref answer sample-index)
                   (exp (* (weight fw)
                           (aref feature-array
                                 feature-index
                                 sample-index)))))
```

```
          (incf feature-index))))))

;;; the maximum row sum
;;; number of features , number of samples
;;; returns an array of values

(defmethod max-hash-array ((array array))
  (let* ((dim (array-dimensions array))
         (sum 0)
         (num-samples (second dim))
         (answer (make-array num-samples)))
    (dotimes (i num-samples answer)
      (setf (aref answer i)
            (progn
              (setf sum 0)
              (dotimes (j (first dim) sum)
                (incf sum (aref array j i))))))))

(defmethod max-of-array ((a array))
  (let ((max 0))
    (dotimes (i (length a) max)
      (setf max
            (max max (aref a i))))))


(defmethod sum-of-array ((a array))
  (let ((sum 0))
    (dotimes (i (length a) sum)
      (incf sum (aref a i)))))
```

C.9 m-h.lsp

```
;;;; multi-path random-walk Metropolis-Hastings sampler
;;;; (with annealing )
;; number of steps before convergence of distribution

(defvar *m-h-convergence* 10)
(setf *m-h-convergence* 250)

(defmethod sample-mp-rw-mh ((rf random-field) n)
  (let ((answer (make-instance 'corpus
                 :contents nil)))
    (dotimes (i n answer)
      ;; for each n we do a different loop
      (push (sample-once rf (choose-node-num (null-field rf)))
            (list-of-trees answer)))))

(defmethod sample-3 ((rf random-field)
```

```
                    (number-of-paths integer)
                    (burn-in integer)
                    (n integer))
  (let ((answer (make-instance 'corpus
                 :contents nil)))
    (dotimes (i number-of-paths answer)
      ;; for each n we do a different loop
      (sample-n-times #'(lambda (tree)
                          (push tree (list-of-trees answer)))
                      rf
                      (choose-node-num (null-field rf))
                      burn-in
                      n))))

(defmethod sample-and-discard ((rf random-field)
                               (number-of-paths integer)
                               (burn-in integer)
                               (n integer)
                               )
  (let* ((number-of-features (length (feature-weight-list rf)))
         (number-of-samples (* n number-of-paths))
         (answer (make-array (list number-of-features
                                   number-of-samples)
                             :initial-element 0))
         (sample-index 0))
    ;;; we return this array
    (dotimes (i number-of-paths answer)
      ;; for each n we do a different loop
      (sample-n-times #'(lambda (tree)
                          (calculate-feature-array
                           tree
                           (feature-weight-list rf)
                           sample-index answer)
                          (incf sample-index))
                      rf
                      (choose-node-num (null-field rf))
                      burn-in
                      n))))

;;; this one counts up each feature
;;; and updates the array

(defmethod calculate-feature-array ((tree tree)
                                    (fwl list)
                                    (sample-index integer)
                                    (answer array))
  (let ((feature-index 0))
    (dolist (fw fwl)
      (setf (aref answer feature-index sample-index)
            (all-matches (feature fw) tree))
```

```
      (incf feature-index))))

;;; we do the second-order stuff
;;; because sometimes we want to sample and discard the
;;; actual trees, retaining just some statistics.

(defmethod sample-n-times (func
                           (rf random-field)
                           (number-of-nodes integer)
                           (burn-in integer)
                           (n integer))
  (let ((last-sample nil)
        (last-field-weight 1)
        (new-field-weight 1)
        (new-sample (sample-initial-n (null-field rf)
                                      number-of-nodes))
        (rw nil))
    ;;; we start with the initial sample from the null field
    ;;; we then anneal for m-h-convergence iterations
    (dotimes (j (+ burn-in n))
      (setf last-sample new-sample)
      (setf last-field-weight new-field-weight)
      (setf rw (random-step rf last-sample))
      (setf new-sample (tree rw))
      (setf new-field-weight (field-weight new-sample rf))
      (when  last-sample
             ;;; this is the critical acceptance step
             ;;; that adjusts the probability
             (let ((acceptance-ratio (* (/ new-field-weight
                                           last-field-weight)
                                        (asymmetry rw))))

               (when
                 (> (random 1.0)
                    acceptance-ratio)
                 ;; scale this by the swapping asymmetry
                 ;; the coin has come up tails
                 ;; in this case we discard it.
            ;     (format t "d")
                 (setf new-sample last-sample
                       new-field-weight last-field-weight))))
      (when (>= j burn-in)
        (funcall func new-sample)))))

;;; this samples once correctly wrt to the
;;; random field for the fixed sub-field
;;; of a certain size

(defmethod sample-once ((rf random-field)
                        (number-of-nodes integer))
```

```
  (let ((last-sample nil)
        (last-field-weight 1)
        (new-field-weight 1)
        (new-sample (sample-initial-n (null-field rf)
                                      number-of-nodes))
        (rw nil))
    ;;; we start with the initial sample from the null field
    ;;; we then anneal for m-h-convergence iterations
    (dotimes (j *m-h-convergence*)
      (setf last-sample new-sample)
      (setf last-field-weight new-field-weight)
      (setf rw (random-step rf last-sample))
      (setf new-sample (tree rw))

      (setf new-field-weight (field-weight new-sample rf))
      (when
        (and last-sample
             ;;; this is the critical acceptance step
             ;;; that adjusts the probability
             (> (random 1.0)
                (* (/ new-field-weight last-field-weight)
                   (asymmetry rw)))
        ;; scale this by the swapping asymmetry
        ;; the coin has come up tails
        ;; in this case we discard it.
        ;(format t "d")
        (setf new-sample last-sample
              new-field-weight last-field-weight))))
    new-sample))


;;; a class that encapsulates
;;; possibilties for a random step
;;; probably doesn't need to be a class

(defclass random-walk ()
  ((tree
    :accessor tree
    :initarg :tree)
   (m :accessor m
      :initarg :m)
   (n :accessor n
      :initarg :n)))


(defmethod asymmetry ((rw random-walk))
  (float (/ (m rw) (n rw))))

;;; returns a random-walk object
;;; so we need to calculate the ratio of the
```

```
;;; transition probabilities
;;; which is one in the case of the label, and words,
;;; and is slighlty
;;; change this back to get words working

(defmethod random-step ((rf random-field)
                        (last-sample tree))
  (let ((switch (random 2)))
    (case switch
      (0
       ;; switch a label
       (swap-random-label rf last-sample))
      (1
       (swap-tree-structure rf last-sample)))))

;;; choose a random label
;;; and change it to something else
;;; return a random walk object
;;; we also need to recopy the whole tree,
;;; or we won't be able to back out.


(defmethod swap-random-label ((rf random-field)
                         (last-sample tree))
  (let ((n (random (number-of-nodes last-sample)))
        (new-sample (duplicate-tree last-sample)))
    ;(format t "~%  node is ~a" n)
    ;; now we iterate through
    (setf (label (nth-node new-sample n))
          (choose-label (null-field rf)))
    (make-instance 'random-walk
      :tree new-sample
      :m 1
      :n 1)))

;;; this isn't used here

(defmethod swap-random-word ((rf random-field)
                         (last-sample tree))
  (let ((n (random (number-of-words last-sample)))
        (new-sample (duplicate-tree last-sample)))
    ;; now we iterate through
    (setf (word (nth-leaf new-sample n))
          (choose-word (null-field rf)))
    (make-instance 'random-walk
      :tree new-sample
      :m 1
      :n 1)))
```

```
;; this is fairly tricky.


(defmethod swap-tree-structure ((rf random-field)
                                (last-sample tree))
  (let* ((forward-swaps
          (count-candidate-swaps last-sample))
         (reverse-swaps
          (count-candidate-converse-swaps last-sample))
         (total (+ forward-swaps reverse-swaps))
         (chosen (random total))
         (new-sample (duplicate-tree last-sample)))
   ; (format t "~% choices ~a ~a" forward-swaps reverse-swaps)
    (cond ((and (= 0 forward-swaps)
                (= 0 reverse-swaps))
           ;; we can't do it
           (make-instance 'random-walk
             :tree new-sample
             :m 1
             :n 1))
          ((< chosen forward-swaps)
           ;; swap forward
           (do-nth-forward-swap new-sample chosen total)
           )
          (t
           (setf chosen (- chosen forward-swaps))
           ;; swap back
           (do-nth-backward-swap new-sample chosen total rf)
           ))))

;;; returns a random walk object

(defmethod do-nth-forward-swap ((tree tree)
                                (n integer)
                                (possibles integer))
  ;; possibles is just something you will need for the
  ;; transition probabilities.
  ;; find the mother of it .
  (let* ((m-d (nth-mother-daughter tree n))
         (mother (first m-d))
         (daughter (second m-d))
         (l (length (daughters mother)))
         (sister nil)
         (i (random (1- l)))
     ;    (pre-num (number-of-nodes tree))
         )
    ; (write (write-tree-to-list tree))
    (dolist (sibling (daughters mother))
      (when (and (= i 0)
                 (not (eq daughter sibling)))
```

```
    (setf sister sibling)
    (return))
  (when (not (eq daughter sibling))
    (decf i)))
;; remove it
(setf (daughters mother)
      (delete daughter (daughters mother)))
;; now we have chosen the one to insert or whatever
(let* ((l2  (length (daughters sister)))
       (k (random (1+ l2))))
  ;; this list is where we insert the daughter
  ;; so there are l + 1 possibilities
  (cond ((= l2 0)
         ;; it was a leaf
        ; (format t "~% leaf to complex-tree")
         (change-class sister 'complex-tree)
         (setf (daughters sister) (list daughter)))
        ((= k 0)
         (push daughter  (daughters sister)))
        (t
         ;; put it in the right place
          (setfnthcdr k
                      (daughters sister)
                      (cons daughter
                            (nthcdr k (daughters sister))))))
         ;(push daughter (nthcdr k (daughters sister)))))
  ;; so now we just have to calculate the right ratios
  (let ((change 0))
    (when (= l 2)
      ;; we have lost one
      (decf change))
    (when (= l2 1)
      ;; you gain one
      (incf change))
    ;; then in the converse
    ;; put this in later
    ;(write (write-tree-to-list tree))
    ;(unless (= pre-num (number-of-nodes tree))
     ; (write (write-tree-to-list tree))
      ;(break "number change"))
    (when (or (= 0 possibles)
              (= l 1)
              (= l2 -1)
              (= l 0))
      (break "zero error"))
    (make-instance 'random-walk
      :tree tree
      :m (* possibles (1- l) (1+ l2))
      :n (* (+ possibles change) l ))))))
```

```
;; this looks for the nth sibling and returns a
;; a list (mother daughter)
;; n starts at 0

(defmethod nth-mother-daughter ((tree tree)
                                (n integer))
  (let ((answer (nth-mother-daughter-b tree n)))
    (when (integerp answer)
      (break "error"))
    answer))


(defmethod nth-mother-daughter-b ((tree leaf)
                                (n integer))
  n)

(defmethod nth-mother-daughter-b ((tree complex-tree)
                                (n integer))
  (let ((l (length (daughters tree))))
    (cond  ((and (> l 1)
                 (< n l))
            ;; bingo
            (list tree (nth n (daughters tree))))
           (t
            ;;
            (when (> l 1)
              (decf n l))
            (dolist (daughter (daughters tree))
              ;; for each daughter try it
              (setf n (nth-mother-daughter-b daughter n))
              (unless (integerp n)
                (return))
              )
            n))))


(defmethod count-candidate-swaps ((last-sample tree))
  ;; just count all the sisters
  0
  )

(defmethod count-candidate-swaps ((tree complex-tree))
  ;; just count all the sisters
  (let ((l (length (daughters tree))))
    (if (= l 1)
      ;; it is a singleton
      (count-candidate-swaps (first (daughters tree)))
      (+ (reduce #'+
                 (mapcar #'count-candidate-swaps
                         (daughters tree)))
```

```
        l))))


;; just the number of nodes -1 - daughters of the head node.
;; this would be zero if it is all in one local tree.

(defmethod count-candidate-converse-swaps ((tree complex-tree))
  ;; non-initial-mothers
  (- (number-of-nodes tree)
     (1+ (length (daughters tree)))))


;;; this is just the reverse
;;; take any node not in the initial local tree
;;; and re-attach it to it's grandmother

(defmethod do-nth-backward-swap ((tree tree)
                                 (n integer)
                                 (possibles integer)
                                 (rf random-field))
  ;; possibles is just something you will need for the
  ;; transition probabilities.
  ;; find the mother of it .
  (let* ((m-d (nth-grandmother-daughter tree n))
         (grandmother (first m-d))
         (mother (second m-d))
         (daughter (third m-d))
         (l (length (daughters grandmother)))
        ;  (pre-num (number-of-nodes tree))
         (i (random (1+ l))))
    ;; remove it from the daughter
    ;; destructively
    (setf (daughters mother)
          (delete daughter (daughters mother)))
    (unless (daughters mother)
      ;; it might be a leaf now
      (change-class mother 'leaf)
      (setf (word mother) (choose-word (null-field rf))))
    ;; add it to some appropriate thing.
    (if (= i 0)
      (push daughter  (daughters grandmother))
       ;;(push daughter (nthcdr i (daughters grandmother))))
      (progn (setfnthcdr i (daughters grandmother)
               (cons daughter
                     (nthcdr i (daughters grandmother))))))
  ;  (unless (= pre-num (number-of-nodes tree))
   ;       (write (write-tree-to-list tree))
    ;      (break "number change"))
    ;; done
    ;; so now we just have to calculate the right ratios
```

```
    (let ((change 0))
      (when (eq grandmother tree)
        ;; we have lost one of this sort
        (decf change))
      ;; we need some more things so check this later
      (make-instance 'random-walk
        :tree tree
        :m (* possibles (1- l) )
        :n (* (+ possibles change)
              l
              (1+ (length (daughters mother))))))))


;;; return a list with three elements
;;; (granny mother daughter)
;;; the nth such daughter

(defmethod nth-grandmother-daughter ((tree complex-tree)
                                     (n integer))
  (let ((answer (nth-g-d tree n)))
    (when (integerp answer)
      (break "error"))
    answer))

(defmethod nth-g-d ((tree complex-tree)
                                     (n integer))
  (let ((answer nil))
    (dolist (daughter (daughters tree))
      (dolist (granddaughter (daughters daughter))
        (when (= n 0)
          ;; we have found it
          (setf answer (list tree daughter granddaughter))
          (return))
        (decf n))
      (when answer
        (return)))
    (or answer
        ;; not with this one as root
        ;; so drop down a bit
        (progn
          (dolist (daughter (daughters tree) n)
            (setf n (nth-g-d daughter n))
            (unless (integerp n)
              ;; done

              (return n)))))))

(defmethod nth-g-d ((tree leaf)
                                     (n integer))
  n)
```

```
;;; sample-once for fixed N
;;; where N is the total number of nodes in the tree
;;; both terminal and non-terminal.
;;; it doesn't really matter what the distribution is
;;; as long as it is not too skew and it covers the whole space.


(defmethod sample-initial-n ((df default-field)
                             (n integer))
  (let ((array (make-array n :initial-element nil))
        (node-array (make-array n)))
    (dotimes (i (1- n))
      (push (1+ i) (aref array (random i))))
   ; (write array)
    ;; convert this into nodes
    (dotimes (i n)
      (if (aref array i)
        ;; non null
        (setf (aref node-array i)
              (make-instance 'complex-tree
                :label (choose-label df)))
        (setf (aref node-array i)
              (make-instance 'leaf
                :label (choose-label df)
                :word (choose-word df)))))
    ;(write node-array)
    ;; now map all the daughters
    (dotimes (i n)
      (if (aref array i)
        (setf (daughters (aref node-array i))
              (shuffle
               (mapcar #'(lambda (n)
                           (aref node-array n))
                       (aref array i))))))
    ;(write node-array)
    ;(write (write-tree-to-list (aref node-array 0)))
    (aref node-array 0)))


;;; very simple shuffling algorithm that should work enough

(defmethod shuffle ((a array))
  (let ((n (length a)))
    (dotimes (i n)
      (let ((index (random n))
            (old (aref a i)))
        (setf (aref a i)
              (aref a index)
              (aref a index)
```

```
            old)))
    a))

;; this is embarassingly inefficient
;; CHANGEME

(defmethod shuffle ((a list))
  (let ((n (length a)))
    (dotimes (i n)
      (let ((index (random n))
            (old (nth i a)))
        (setf (nth i a)
              (nth index a)
              (nth index a)
              old)))
    a))
```

C.10 susanne.lsp

```
;;; susanne.lsp
;;; this has to load susanne files
;;; and reconstruct the trees

;(defvar *susanne-directory* "F:\\random\\susanne\\")

(defvar *susanne-directory* "Macintosh HD:alex:random:")

;;; we just use a very small subset at the moment

(defvar *susanne-files* '("A01" "A02" ))

(defvar *max-corpus-line* 300)

(defclass corpus-line ()
    ((reference :accessor reference)
     (status :accessor status)
     (wordtag :accessor wordtag)
     (word :accessor word)
     (lemma :accessor lemma)
     (parse :accessor parse)))

;;; line is of type corpus-line

(defclass susanne-leaf (leaf)
    ((line :accessor line
       :initarg :line)))

(defclass susanne-tree (complex-tree)
    ((true-label :accessor true-label
```

```
      :initarg :true-label)))

;;; this should return nil when
;;; you reach eof

(defmethod read-corpus-line ((input stream))
   (let ((line (make-instance 'corpus-line))
         (surface (read-line-b  input))
         (mark1 12)
         (mark2 nil))
      (if (null surface)
         ;; eof
         nil
         (progn
      (setf (reference line)
            (subseq surface 0 9))
      (setf (status line)
            (aref surface 10))
      (setf mark2
            (position #\tab surface :start mark1))
      (setf (wordtag line)
            (subseq surface mark1 mark2))
      (setf mark1 (1+ mark2))
      (setf mark2
            (position #\tab surface :start mark1))
      (setf (word line)
            (subseq surface mark1 mark2))
      (setf mark1 (1+ mark2))
      (setf mark2
            (position #\tab surface :start mark1))
      (setf (lemma line)
            (subseq surface mark1 mark2))
      (setf mark1 (1+ mark2))
      (setf mark2
            (position #\tab surface :start mark1))
      (setf (parse line)
            (subseq surface mark1 mark2))
      line))))

;;; return a string that stops at the first cr
;;; we can't use read-line because it needs CR/LF
;;; return nil if eof

(defmethod read-line-b ((input stream))
   (let ((s (make-string *max-corpus-line*))
         (c nil))
      (dotimes (i *max-corpus-line* nil)
         (setf c (read-char input nil nil))
         (when (and (null c)
                    (= i 0))
```

```
            (return nil))
         (when (eql c
                    #\Linefeed)
            (if (> i 1)
               (return (subseq s 0 i))
               (return nil)))
         (setf (aref s i) c))
      ))

(defmethod read-ref ((input stream)
                     (line corpus-line))
   (let ((s (make-string 9)))
      (dotimes (i 9)
         (setf (aref s i)
               (read-char input)))
      (setf (reference line) s)))

(defmethod read-status ((input stream)
                        (line corpus-line))
   (read-tab input)
   (setf (status line)
         (read-char input)))

(defmethod read-tab ((input stream))
   (unless (eql (read-char input) #\tab)
      (error "Susanne format error")))

;;; this one just reads in a string stopping with white space

(defclass susanne-corpus (corpus)
    ((line-array :accessor line-array)))

(defmethod load-susanne ()
   (let ((corpus (make-instance 'susanne-corpus
                    :size 0)))
      (setf (line-array corpus)
            nil)
      (dolist (file *susanne-files*)
         (with-open-file
           (input (parse-namestring
                    (concatenate
                      'string
                      *susanne-directory*
                      file)))
           (read-susanne-corpus corpus input)))
      corpus))

;; do it iteratively with an explicit stack.
;; tree stack is the stack of unfinished constituents,
;; as we get closing brackets we pop 'em off
```

```
;; and at the end we stick the new tree into the corpus contents

(defmethod read-susanne-corpus ((corpus susanne-corpus)
                                (input stream))
   (let ((tree-stack nil)
         (num-trees 0))
      (loop
         (let ((current-line (read-corpus-line input)))
            (unless tree-stack
               (incf num-trees))
;            (when (> num-trees 20)
 ;              (return))
            (unless current-line
               (when tree-stack
                  (error "file ended with non-empty stack"))
               (return nil))
            ;; we have a current line
            ;;; now we loop through the ending
            (let* ((mark1 0)
                   (mark2 nil)
                   (parse (parse current-line))
                    (l (length parse)))
               (loop
                 ;; we look at what the current thing
                 ;; first check to see if we have finished
                 (when (= mark1 l)
                    (return))
                 (let ((c (aref parse mark1)))
                    (cond ((eql c #\[)
                           ;; we are starting a new constitutent
                           (setf mark2 (find-end parse
                                                 (1+ mark1)))
                           (let* ((label (subseq parse
                                                      (1+ mark1)
                                                      mark2))
                                  (new-tree
                                   (make-instance 'susanne-tree
                                     :label (map-label label)
                                     :true-label label
                                     :daughters nil)))
                              ;; we put the daughters on
                              ;; in reverse order and
                              ;; reverse them later
                              (when tree-stack
                                 (push new-tree
                                       (daughters
                                        (first tree-stack))))
                              (push new-tree tree-stack)
                              (setf mark1 mark2)))
                          ((eql c #\.)
```

```
                           ;; we are at a leaf
                           (let ((new-leaf
                                  (make-instance 'susanne-leaf
                                    :line current-line
                                    :label (map-label
                                            (wordtag current-line))
                                    :word (word current-line)))
                                 (closures (count #\]
                                             parse
                                             :start (1+ mark1)))
                                 (tr nil))
                              (push new-leaf
                                (daughters
                                 (first tree-stack)))
                              ;; we want tr to have wider scope
                              ;; than the following loop
                              ;; we have reached then end now
                              ;; we just need to carry on until
                              ;; the end of the line,
                              ;; reading the right brackets.
                              ;; pop em off one at a time, -
                              ;; this means they are finishes
                              ;; and remember to nreverse
                              ;; the daughters list as we go up
                              (dotimes (i closures)
                                 ;; we dont bother to check
                                ;; that the labels match
                                 (setf tr (pop tree-stack))
                                 (setf (daughters tr)
                                       (nreverse
                                         (daughters tr))))
                              ;; check if this last was top-level
                              (unless tree-stack
                                 (push tr
                                   (list-of-trees corpus))))
                           (return))))))))
      (incf (size corpus) num-trees)))

(defmethod find-end ((parse string)
                     start)
  (let ((k start))
    (dotimes (i
              (- (length parse)
                 start)
              nil)
      (let ((c (aref parse k)))
        (when (or (eql c #\[)
                  (eql c #\])
                  (eql c #\.))
          (return k)))
```

```
      (incf k))))

;; this function just remaps labels from one sort of thing
;; to another - for now we will just have them as
;; being the first letter

(defvar *simplify* t)

;; this just takes the first character

(defmethod map-label (label)
   (if *simplify*
      (subseq label 0 1)
      label))
```

C.11 fake.lsp

```
;;;;fake.lsp
;;; some functions for generating a fake corpus
;;; using a simple STSG

(defclass scs-grammar ()
  ((hash-table :accessor hash-table)
   (start-symbol :accessor start-symbol)))

(defmethod generate-one-tree ((grammar scs-grammar))
  (generate (start-symbol grammar) grammar))

(defclass scs-rule ()
  ((lhs :accessor lhs
        :initarg :lhs)
   (weights :accessor weights
            :initarg :weights)
   (trees :accessor trees
         :initarg :trees)))

;;; trees is an array of trees
;;; the root of each of them is lhs
;;; this returns a tree
;;; with start-category as the head  of it

(defmethod generate (start-category
                     (grammar scs-grammar))
  (expand-tree (choose-rule start-category grammar)
               grammar))

(defmethod choose-rule (category
                        (grammar scs-grammar))
  (let* ((rules (gethash category (hash-table grammar)))
```

```
         (r (random 1.0))
         (weights (weights rules))
         (i 0))
    (dotimes (j (length weights))
      (when (< r (aref weights j))
        ;; we have found it
        (setf i j)
        (return))
      (setf r (- r (aref weights j))))
    ;; so i is now thw index
    (aref (trees rules) i)))

;;; copy the tree
;;; if it ends in a non-terminal then
;;; expand it using grammar

(defmethod expand-tree ((tree complex-tree)
                        (grammar scs-grammar))
  (cond ((null (daughters tree))
         ;; we need to expand this one
        (expand-tree (choose-rule (label tree) grammar)
                     grammar)
         )
        (t
         ;; just copy
         (make-instance 'complex-tree
           :label (label tree)
           :daughters (mapcar #'(lambda (tree)
                                  (expand-tree tree grammar))
                              (daughters tree)))
         )))

(defmethod expand-tree ((leaf leaf)
                        (grammar scs-grammar))
  ;; we just copy it
  ;; maybe we don't need to copy it but better
  ;; safe than sorry
  (make-instance 'leaf
    :label (label leaf)
    :word (word leaf)))

;;; now we need some code to generate a scs-grammar

(defvar *scs-grammar*)

(defun create-scs-grammar (list start-symbol)
  (let ((g (make-instance 'scs-grammar)))
    (setf (hash-table g) (make-hash-table :test #'equal))
    (setf (start-symbol g) start-symbol)
    (dolist (r list)
```

```
      (let ((rule (read-scs-rule r)))
        (setf (gethash (lhs rule) (hash-table g))
              rule)))
    g))

;;; read-tree-from-list will work
;;; so this looks like
;;; (lhs ((0.9 tree1) (0.1 tree2)))
;;; where the label of tree1 and tree2 are both lhs

(defmethod read-scs-rule ((list list))
  (let* ((weights (mapcar #'car (second list)))
         (trees (mapcar #'(lambda (pair)
                            (read-tree-from-list (second pair)))
                        (second list)))
         (n (length weights)))
    (make-instance 'scs-rule
      :lhs (first list)
      :weights (make-array n :initial-contents weights)
      :trees (make-array n :initial-contents trees))))

(defmethod fake-corpus ((grammar scs-grammar)
                        (n integer))
  (let ((answer (make-instance 'corpus)))
    (dotimes (i n answer)
      (push (generate-one-tree grammar)
            (list-of-trees answer)))))
```

C.12 proposal.lsp

```
;;;; proposal.lsp
;;;; this has the code
;;;; for the feature proposal algorithms

;;; this returns the list of candidate features

(defmethod candidate-features ((rf random-field))
  (let* ((atomic-features (atomic-features rf))
         (field-features (mapcar #'feature
                                 (feature-weight-list rf)))
         (cl (make-instance 'candidate-list
               :done field-features)))
    (dolist (a atomic-features)
      (add-candidate a cl)
      (dolist (fw (feature-weight-list rf))
        (when (> (weight fw) -100)
          (compose-feature (feature fw) a cl))))
   ; (dolist (cand (contents cl))
    ;  (format t "~% candidate ~a" (write-feature-to-list cand)))
```

```
    (contents cl)))

;;; this is just an object that holds a stack
;;; and removes the duplicates

(defclass candidate-list ()
  ((contents :accessor contents
             :initform nil)
   (done :accessor done
         :initarg :done)))

(defmethod add-candidate ((f feature)
                          (cl candidate-list))
  (when (and (notany #'(lambda (g)
                         (equal-feature f g))
                     (contents cl))
             (notany #'(lambda (g)
                         (equal-feature f g))
                     (done cl)))
    (push f (contents cl))))


;;; this returns a list of the atomic features
;;; to start with just one for each label in the field

(defmethod atomic-features ((rf random-field))
  (let ((label-array (label-array (null-field rf)))
        (answer nil))
    (dotimes (i (length label-array) answer)
      (push (make-instance 'label-feature
              :label (aref label-array i))
            answer))))

;;;; this returns nothing
;;; it just adds them to the candidate list
;;; using add-candidate

(defmethod compose-feature ((f feature)
                            (atomic atomic-feature)
                            (cl candidate-list))

  nil)

;;; we just add these to any point in the tree.
;;; in this case

(defmethod compose-feature ((f label-feature)
                            (atomic atomic-feature)
                            (cl candidate-list))
  (add-candidate (make-instance 'feature-tree
```

```
                 :head-label (label f)
                 :daughter-list (list atomic)
                 :order-type *unordered*)
                 cl))


;; this is the problem
;; it only works to a fixed depth of two
;; this is just for coding convenience
;; and as an arbitrary limit on the search space


(defmethod compose-feature ((f feature-tree)
                            (atomic atomic-feature)
                            (cl candidate-list))
  ;;; this one just adds it to the sisters
  ;;; depth 1
  (add-candidate (make-instance 'feature-tree
                 :head-label (head-label f)
                 :daughter-list (cons atomic
                                      (daughter-list f))
                 :order-type *unordered*)
                 cl)
  ;;; depth 0
  (add-candidate (make-instance 'feature-tree
                 :head-label (label atomic)
                 :daughter-list (list f)
                 :order-type *unordered*)
                 cl)
  (dolist (daughter (daughter-list f))
    ;; we add it as a daughter of daughter
    (let ((new-tree
           (cond ((typep daughter 'feature-tree)
                  (make-instance 'feature-tree
                    :head-label (head-label daughter)
                    :daughter-list
                    (cons atomic
                      (daughter-list daughter))
                    :order-type *unordered*))
                 (t
                  (make-instance 'feature-tree
                    :head-label (label daughter)
                    :daughter-list (list atomic)
                    :order-type *unordered*)))))
      (add-candidate (make-instance 'feature-tree
                       :head-label (head-label f)
                       :daughter-list (substitute
                                       new-tree
                                       daughter
                                       (daughter-list f)
```

```
                   :test #'eq)
   :order-type *unordered*)
  cl))))
```