Dynamic Programming and the Graphical Representation of Error-Correcting Codes

Stuart Geman and Kevin Kochanek
Division of Applied Mathematics
Brown University
Providence, Rhode Island 02912
January, 2000

[1]Supported by Army Research Office contract DAAH04-96-1-0445, National Science Foundation grant DMS-9217655, and Office of Naval Research contract N00014-97-0249.

Abstract

Graphical representations of codes facilitate the design of computationally efficient decoding algorithms. This is an example of a general connection between dependency graphs, as arise in the representations of Markov random fields, and the dynamic programming principle. We concentrate on two computational tasks: finding the maximum-likelihood codeword and finding its posterior probability, given a signal received through a noisy channel. These two computations lend themselves to a particularly elegant version of dynamic programming, whereby the decoding complexity is particularly transparent. We explore some codes and some graphical representations designed specifically to facilitate computation. We further explore a coarse-to-fine version of dynamic programming that can produce an exact maximum-likelihood decoding many orders of magnitude faster than ordinary dynamic programming.

Key Words and Phrases: dynamic programming, graphical models, maximum-likelihood decoding, soft decoding

1 Introduction

By now, nearly everyone in the coding community knows about the tight relationship between the dependency structure on a set of random variables and the costs of computing certain functionals on their joint distribution. One way to deduce this relationship is to generalize, more-or-less straightforwardly, the principle of dynamic programming (Bellman [1]), but it can be arrived at from many other directions as well. In fact, an amazing history of discovery and re-discovery emerges from a diverse range of applications. Indeed, in one way or another, the "forward-backward" algorithm [2], "peeling" [3], "parsing" [4], the Viterbi algorithm [5], the "sum-product" algorithm [6], [7], [8], [9], "bucket elimination" [10], and "evidence propagation" [11], [12], [13], all rest on this very same dynamic-programming principle. Oftentimes there is a good reason for using one variation over another. With one approach, there may be a book-keeping scheme that avoids duplicating computations; with another, the computational flow may be particularly transparent or convenient. The best choice is typically dictated by the computational constraints and the functionals of interest.

We will concentrate here on two particular kinds of computations: computation of the most likely configuration of a collection of random variables, and this configuration's associated probability. In the context of maximum likelihood ("soft") decoding, we calculate these functionals under the *posterior* distribution—the conditional distribution on the transmitted codewords given the received signal. What codeword was most likely transmitted? What is the conditional probability that this maximum-likelihood codeword was in fact transmitted? In other words, we will compute the *maximum aposteriori* (MAP) estimator of the transmitted codeword and the (conditional) probability that the estimator is actually correct. MAP decoding gives the most likely codeword, and therefore minimizes the probability of a decoding error. If, alternatively, we seek to minimize the number of information-bit errors, then we would maximize the posterior marginal at each information bit. In general, these are not the same thing, and the better choice is clearly problem dependent. MAP makes sense when the codeword represents a logical unit, perhaps a product name, a price, a category, or an English word. On the other hand, a stream of information bits with no *a priori* structure calls for minimizing the bit error rate.

This paper is about using graphical representations of codes and probability models of channels to calculate the exact MAP decoding and its probability. The emphasis is on finding representations and algorithms that make these calculations computationally feasible. It should perhaps be pointed out that, although the connections between graphs, codes, and computation are currently the subjects of intense research, most of the effort is in a somewhat different direction. The emphasis, instead, is on iterative decoding algorithms that are generally not exact and whose convergence properties are generally not well understood, but often exhibit, nevertheless, spectacular performance. Many codes which would be impossible to decode exactly lend themselves effectively and efficiently to iterative decoding, in some cases even approaching channel capacity. Gallager's [14] low-density parity-check codes and associated iterative decoding algorithm are probably the first examples, but few if any recognized the power of Gallager's construction. Tanner [15] found clean graphical representations of Gallager and other related codes, and utilized these representations to address a host of design issues concering computational complexity and minimum distance. Tanner's graphical representations helped to set the stage for the rediscovery of Gallager codes, and for the many extensions that have evolved over the past few years. (But

see also Bahl et al. [16] for an early paper anticipating, rather remarkably, the modern viewpoint.) Of course the turbo-codes of Berrou et al. [17] also spurred this line of research, since these codes, as well, have natural graphical representations, come equipped with an iterative decoding algorithm, and appear to perform near optimally in certain regimes. Wiberg and collaborators (see [6] and [7]) picked up on Tanner graphs and turbo codes, and within a general treatment of codes on graphs made connections to soft decoding, general channel models (with memory), and iterative and non-iterative decoding algorithms. MacKay and Neal (see MacKay [18] for a thorough discussion) developed some of these same themes, and, additionally, introduced related codes and decoding methods that appear to be among the current best performers. The connections and common themes among all of these approaches, as well as to a broader framework including Bayesian inference, Markov random fields, belief propagation and the modern theory of expert systems, seem to have been first understood and most clearly formulated by Kschischang and Frey [19].

Our goal in this paper is twofold. We first present a brief tutorial on Markov random fields (MRF's), dependency graphs, and the non-iterative computation of various functionals on marginal and posterior distributions. In the remainder of the paper, we present some new results on how to apply these computational strategies to facilitate the maximum-likelihood decoding of graphical codes. As we have said, our focus is somewhat different from the current trend, in that we concentrate on representations that allow exact computation, and on new methods for reducing computational complexity. But in light of the often good performance of iterative methods, it would be of great interest to systematically compare iterative and exact computations in a decoding problem that lends itself to both approaches. We have not yet made these comparisons.

Section 2 is about generic computational issues on graphs. There is an especially transparent connection between graphs that represent dependency structures and the efficient calculation of a most likely configuration and its associated probability. This is essentially the "bucket elimination" algorithm, and it is perhaps the most straightforward of the various generalizations of dynamic programming. There is no need to triangulate, no need to construct junction trees, and no need to worry about cycles. Conveniently, the computational cost is readily calculated once a site-visitation schedule has been established. Furthermore, it is often the case that the most efficient

site-visitation schedule is immediately apparent from the general structure of the dependency graph.

In Section 3 we consider the simplest possible case: graphs with linear structure. We review their connection to convolutional codes and revisit Viterbi decoding from the MRF viewpoint. We point out that the (posterior) probability that the Viterbi-decoded signal is correct can be calculated as easily as the decoding itself, and we discuss extensions to channels with Markov memory, designed to model burst noise [20], [7]. In §4, we generalize to the case of tree-structured graphs and make a connection to production systems and context-free grammars. We reformulate Forney's squaring construction [21] to obtain a grammatical representation of the Reed-Muller and other related codes. Moreover, we discuss the computational complexity of soft decoding or of evaluating posterior probabilities for both memoryless and Markov communications channels. Finally, in §5 we introduce two (not necessarily exclusive) methods for reducing the computational demands of maximum likelihood decoding. The first is a "thinning" algorithm which controls computational costs by reducing information density. The second, coarse-to-fine dynamic programming [22], is a kind of multi-scale version of dynamic programming that produces a provably optimal configuration, often with greatly reduced computation. We present an exact coarse-to-fine algorithm for some "context-free" codes, including the Reed-Muller codes, and demonstrate thousands-fold improvements in decoding efficiency.

2 Dependency Graphs and Computing

Given a collection of random variables $X_1 \ldots, X_n$ and a probability distribution

$$P(x_1 \ldots, x_n) = \text{Prob}\{X_1 = x_1 \ldots X_n = x_n\},$$

how difficult is it to compute things like the marginal distribution on X_1, or the most likely configuration $x_1 \ldots, x_n$? More than anything else, the dependency relationships among the random variables dictate the complexity of computing these and other functionals of P. Dependency relationships can be conveniently represented with a graphical structure, from which the complexity of various computations can be more or less "read off." These graphical representations, and their connection to computing, are the foundation of modern expert systems [11], [12], [13] as well as of speech recognition

technologies [2], [23]. In fact, the connection is quite general, having emerged and re-emerged in these and many other application areas, such as genetics [3], coding theory [5], [17], computational linguistics [4], and image analysis [24].

This connection between graphs and computing is fundamental to the modern treatment of soft decoding, and we begin here with a brief tutorial. (See also Frey [25] for an excellent introduction to some of the same material.) Our approach to graphs is through Markov random fields, and our computational focus is on computing most-likely configurations and their corresponding probabilities.

2.1 Markov Random Fields and their Gibbs Representations

For the purpose of constructing dependency graphs, it is convenient to index random variables by a general finite index set S, rather than the more traditional set $\{1, 2 \ldots, n\}$. If state spaces are finite, then $X = \{X_s\}_{s \in S}$ is a finite vector of random variables with finite range. In most applications different components have different state spaces, but merely for the sake of cleaner notation, we will assume a common (and finite) range $\mathcal{R}$. Thus P is a probability on $\mathcal{R}^S$.

P is a *Markov random field* (MRF) with respect to a graph $\mathcal{G} = \{S, \mathcal{N}\}$ if P is strictly positive ($P(x) > 0, \ \forall x \in \mathcal{R}^S$) and if

$$P(x_s|_s x) = P(x_s|\{x_t\}_{t \in \mathcal{N}_s}) \tag{1}$$

for all x and all $s \in S$, where

- S indexes the nodes of $\mathcal{G}$
- $\mathcal{N} = \{\mathcal{N}_s\}_{s \in S}$ is the neighborhood structure of $\mathcal{G}$, meaning that $t \in \mathcal{N}_s$ if and only if $t \neq s$ and there is an edge connecting t and s in $\mathcal{G}$, and
- ${}_s x$ is shorthand for $\{x_t\}_{t \in S \setminus s}$.

The distribution on the random variable associated with any site $s \in S$, given the values of all other random variables, depends only on the values of the neighbors. This generalizes the familiar one-sided Markov property.

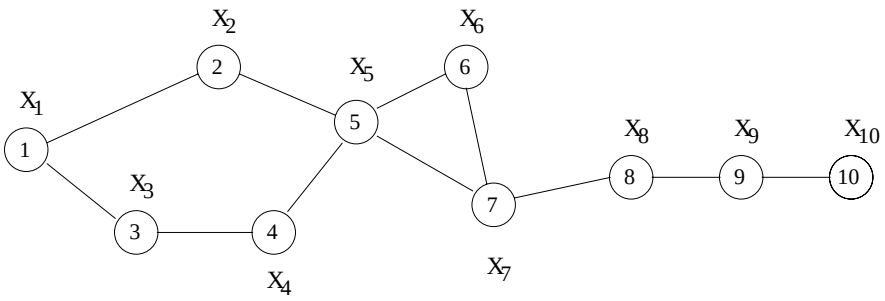

Figure 1: Dependency graph for a simple Markov random field.

An example with $S = \{1, 2 \ldots, 10\}$ is in Figure 1. The graph summarizes many (conditional) independence relationships. For instance,

$$P(x_7|x_1, x_2, x_3, x_4, x_5, x_6, x_8, x_9, x_{10}) = P(x_7|x_5, x_6, x_8)$$
$$P(x_1|x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) = P(x_1|x_2, x_3)$$

P is *Gibbs* with respect to a graph $\mathcal{G} = \{S, \mathcal{N}\}$ if P can be represented as

$$P(x) = \prod_{c \in \mathcal{C}} F_c(x_c) \tag{2}$$

where

- $\mathcal{C}$ is the set of cliques in $\mathcal{G}$, i.e. the set of fully connected subsets of S (including singletons),
- $x = \{x_s\}_{s \in S}$, $x_c = \{x_s\}_{s \in c}$, and
- each F_c is a positive function.

Such representations are certainly not unique: constants can be "shifted" (multiply one term, divide another) and F_c can be absorbed into $F_{c'}$ whenever $c \subseteq c'$.

The main tool for working with MRF's is the representation theorem of Hammersley and Clifford [26], [27]: P is MRF wrt $\mathcal{G}$ if and only if P is Gibbs wrt $\mathcal{G}$. One direction is easy: it is straightforward to verify that (2) has the required Markov property. But the other direction (MRF wrt $\mathcal{G} \Rightarrow$ Gibbs

wrt $\mathcal{G}$) is not easy. It does, though, make easy the proof that MRF's wrt linear graphs are Markov processes (i.e. that equation (1) implies the more familiar "one-sided" Markov property).

Referring to Figure 1, the cliques are the singletons, the pairs $\{\{1,2\}, \{1,3\}, \{3,4\}, \{2,5\}, \{4,5\}, \{5,6\}, \{5,7\}, \{6,7\}, \{7,8\}, \{8,9\}, \{9,10\}\}$, and the triple $\{5,6,7\}$. P MRF wrt the graph in Figure 1 means that P can be factored into terms each of which depends only on the components represented in one of these cliques. An analogous relationship appears in the study of Bayes nets, which use directed acyclic graphs (DAG's). If P "respects" a DAG (if P factors into conditional probabilities of individual daughter nodes given their parent nodes) then P is Markov wrt the corresponding undirected "moral" graph (turn arrows into edges and then connect all parents of each daughter).[1]

If our only interest is in graphical representations of Gibbs distributions, then strict positivity can be relaxed. In particular, even if we drop the condition $F_c(x_c) > 0$ in the definition of Gibbs distributions, we still have the Markov property (1) wrt $\mathcal{G}$, provided that we avoid conditioning on events with probability zero. The *computational* analysis of graphical models, as it turns out, relies only on the implication Gibbs $\Rightarrow$ MRF. Therefore, we shall proceed without the positivity constraint ($F_c > 0$), which would otherwise be troublesome in some of our applications.

2.2 Marginal and Posterior Distributions

In just about every application of MRF's, including coding, we are interested in making inferences about a subset of the variables given observations of the remaining variables. What makes MRF's useful in these applications is the fortunate fact that the *conditional distribution* on the unobserved variables given the observed variables (i.e. the *posterior* distribution) is a MRF on the subgraph of $\mathcal{G}$ obtained by restricting to "unobserved" sites. The conditional distribution on $X_1, X_2, X_5, X_7, X_8, X_9, X_{10}$, given X_3, X_4, and X_6—see Figure 1—is Markov wrt the subgraph at sites $\{1,2,5,7,8,9,10\}$, which in this case happens to be linear. Up to a constant, conditional distributions are just joint distributions with some of the variables fixed, so the statement about their

[1] Another variant is the Tanner graph (cf. [15], [25]), in which clique functions are represented explicitly as specially designated nodes in the dependency graph.

dependency structure follows immediately from the Gibbs representation.

Hidden Markov models (speech recognition [23], Kalman filters [28], etc.) and hidden Markov random fields [29], have dependency graphs like those in Figure 2, where we have labeled the observable variables of the model using y's, and the unobservable ones using x's, in order to distinguish them. In (a) and (b), the posterior distribution, $P(x|y)$, is Markov wrt the linear graph; in (c) it is Markov wrt the nearest-neighbor lattice graph.

The goal is usually to make inference about X, given Y. Therefore, it is significant that the graphical structure representing the distribution on X given Y is no more complicated than the original structure, since, as we shall see shortly, this structure determines computational complexity. In this regard, it is reassuring that models such as those in Figure 2 form a very rich class: the set of *marginal distributions* on Y, obtainable from finite-state space hidden nearest-neighbor Markov models is essentially everything (up to arbitrary approximation, see [29]). One way to understand this is to examine the graphical structure of the marginal distribution on Y. Consider the Gibbs representation: when the X variables are summed (integrated) out, new cliques are introduced. Two sites, s and t, in the Y subgraph will be connected under the marginal distribution on Y (after integrating out X) if s and t are already connected under the joint (X, Y) distribution, *or* if there exists a path, traveling strictly through the X variables, that connects s and t. So the marginal on Y, in the cases depicted in Figure 2, will in general define a fully connected graph! (Just check that there is always a path "through X" connecting any two components of Y.) This observation comes (easily—again!) from the Gibbs representation. (The condition for creating neighbors in a marginal distribution is necessary, but not quite sufficient—hence the disclaimer "in general." Consider, for example, $P(x, y_1, y_2) = P(x|y_1, y_2)P(y_1)P(y_2)$. Then Y_1 and Y_2 are marginally independent—$P(y_1, y_2) = P(y_1)P(y_2)$—yet they are neighbors in the joint distribution *and* they are connected by a path through X.)

2.3 Computation

2.3.1 Most Likely Configurations

Computing a most likely sequence of words given an acoustic signal, a most likely image restoration given a corrupted picture, or a most likely codeword

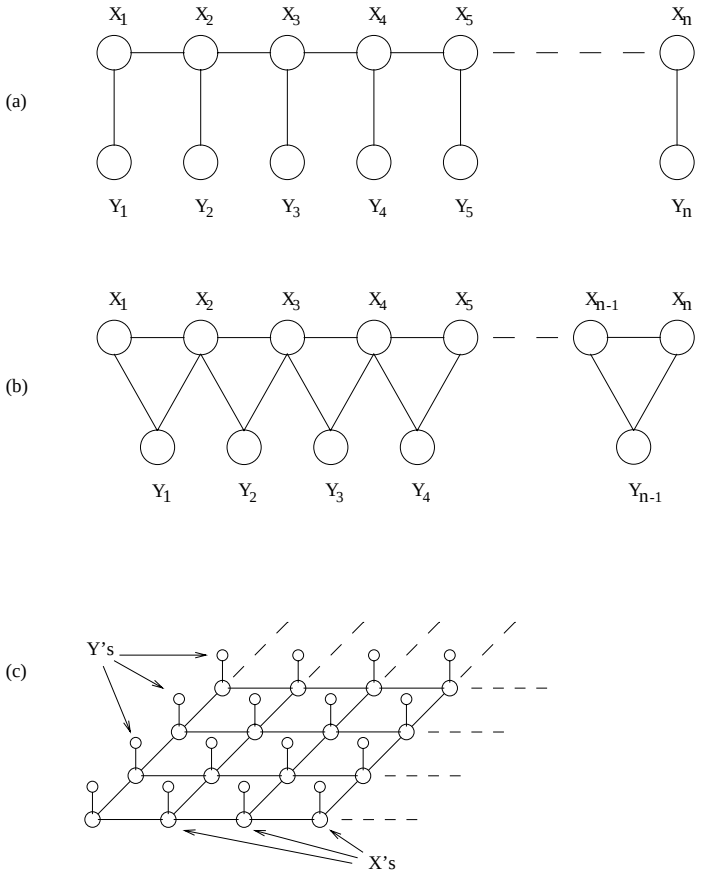

Figure 2: Hidden Markov models ((a) and (b)), and a hidden Markov random field (c).

given a channel output, means computing a most likely configuration under a posterior distribution. In light of our remarks about the dependency structure of conditional distributions (§2.2), it is evident that the generic problem is to maximize a probability distribution that is Gibbs relative to some given graph $\mathcal{G}$.

Consider again the simple example in Figure 1. If, for instance, $\mathcal{R} = \{1, 2 \ldots, 20\}$ then there are 20^{10} possible configurations and exhaustive search for the most likely one is impractical. On the other hand, we could use a kind of (generalized) dynamic programming: choose first an ordering of the nodes in $\mathcal{G}$, say 10, 9, 8, 7, 6, 5, 4, 2, 3, 1. In general, P has the factorization

$$\begin{aligned} P(x) &= F_{1,2}(x_1, x_2)F_{1,3}(x_1, x_3)F_{2,5}(x_2, x_5)F_{3,4}(x_3, x_4)F_{4,5}(x_4, x_5) \\ &\quad F_{5,6,7}(x_5, x_6, x_7)F_{7,8}(x_7, x_8)F_{8,9}(x_8, x_9)F_{9,10}(x_9, x_{10}) \end{aligned}$$

where, for convenience, we have absorbed sub-cliques into super-cliques. Following the site ordering, address x_{10} first: compute

$$\mathcal{R}_{10}(x_9) = \arg\max_{x \in \mathcal{R}} F_{9,10}(x_9, x)$$

and

$$C_{10}(x_9) = F_{9,10}(x_9, \mathcal{R}_{10}(x_9)).$$

Notice that x_9 "isolates" x_{10} from the other components, and $\mathcal{R}_{10}(v)$ is the value of x_{10} that participates in the most likely configuration, if that configuration involves $x_9 = v$. $C_{10}(v)$ is the corresponding contribution from terms involving x_{10}, evaluated at $x_9 = v$. Since x_9 is next on the list, and x_8 isolates x_9 and x_{10} from the other components, compute

$$\mathcal{R}_9(x_8) = \arg\max_{x \in \mathcal{R}} \{F_{8,9}(x_8, x)C_{10}(x)\}$$

and

$$C_9(x_8) = F_{8,9}(x_8, \mathcal{R}_9(x_8))C_{10}(\mathcal{R}_9(x_8))$$

to get, respectively, the value of x_9 participating in the optimal configuration (given x_8) as well as the contribution from terms involving x_9 and x_{10}. As for x_8:

$$\mathcal{R}_8(x_7) = \arg\max_{x \in \mathcal{R}} \{F_{7,8}(x_7, x)C_9(x)\}$$

and

$$C_8(x_7) = F_{7,8}(x_7, \mathcal{R}_8(x_7))C_9(\mathcal{R}_8(x_7)).$$

So far this is standard dynamic programming, but x_7 calls for a slight generalization. Since it takes *both* x_5 and x_6 to isolate x_7, x_8, x_9, and x_{10} from the remaining variables, compute

$$\mathcal{R}_7(x_5, x_6) = \arg\max_{x \in \mathcal{R}} \{F_{5,6,7}(x_5, x_6, x) C_8(x)\}$$

and

$$C_7(x_5, x_6) = F_{5,6,7}(x_5, x_6, \mathcal{R}_7(x_5, x_6)) C_8(\mathcal{R}_7(x_5, x_6)).$$

Proceeding in this way, always isolating what's been done from what hasn't been done, we compute

$$\begin{aligned}
\mathcal{R}_6(x_5) &= \arg\max_{x \in \mathcal{R}} C_7(x_5, x) \\
C_6(x_5) &= C_7(x_5, \mathcal{R}_6(x_5)) \\
\mathcal{R}_5(x_2, x_4) &= \arg\max_{x \in \mathcal{R}} \{F_{2,5}(x_2, x) F_{4,5}(x_4, x) C_6(x)\} \\
C_5(x_2, x_4) &= F_{2,5}(x_2, \mathcal{R}_5(x_2, x_4)) F_{4,5}(x_4, \mathcal{R}_5(x_2, x_4)) C_6(\mathcal{R}_5(x_2, x_4)) \\
\mathcal{R}_4(x_2, x_3) &= \arg\max_{x \in \mathcal{R}} \{F_{3,4}(x_3, x) C_5(x_2, x)\} \\
C_4(x_2, x_3) &= F_{3,4}(x_3, \mathcal{R}_4(x_2, x_3)) C_5(x_2, \mathcal{R}_4(x_2, x_3)) \\
\mathcal{R}_2(x_1, x_3) &= \arg\max_{x \in \mathcal{R}} \{F_{1,2}(x_1, x) C_4(x, x_3)\} \\
C_2(x_1, x_3) &= F_{1,2}(x_1, \mathcal{R}_2(x_1, x_3)) C_4(\mathcal{R}_2(x_1, x_3), x_3) \\
\mathcal{R}_3(x_1) &= \arg\max_{x \in \mathcal{R}} \{F_{1,3}(x_1, x) C_2(x_1, x)\} \\
C_3(x_1) &= F_{1,3}(x_1, \mathcal{R}_3(x_1)) C_2(x_1, \mathcal{R}_3(x_1)) \\
\mathcal{R}_1 &= \arg\max_{x \in \mathcal{R}} C_3(x)
\end{aligned}$$

$\mathcal{R}_1$ is the value of x_1 that participates in the most likely configuration (with the corresponding cost $C_1 = C_3(\mathcal{R}_1)$). Evidently, then, $\mathcal{R}_3(\mathcal{R}_1)$ is the corresponding value of x_3, and $\mathcal{R}_2(\mathcal{R}_1, \mathcal{R}_3(\mathcal{R}_1))$ the corresponding value of x_2, and so-on back through the graph.

The most likely configuration, x, is thereby computed with many fewer operations than required in a systematic (brute-force) search. How many operations are needed? The worst of it is in the computation of $\mathcal{R}_7$, $\mathcal{R}_5$, $\mathcal{R}_4$, and $\mathcal{R}_2$, each of which involves a triple loop (e.g., with regard to $\mathcal{R}_7$: for every x_5 and every x_6 find the best x_7) and hence order $|\mathcal{R}|^3$ operations.

Since there are 10 sites, and no site requires more operations than $O(|\mathcal{R}|^3)$ an upper bound on the computational cost is $O(10|\mathcal{R}|^3)$.

This procedure generalizes. For any graph $\mathcal{G}$ with n sites let $s_1, s_2 \ldots, s_n \in S$ be a "site visitation" schedule. How much work is involved in "visiting" a site? When visiting s_k, we need to compute the best x_{s_k} for each possible configuration on the set of sites *that isolate* $s_1, s_2 \ldots, s_k$ from the remaining sites. In other words, if N_k is the number of neighbors of $s_1, s_2 \ldots, s_k$ among $s_{k+1}, \ldots, s_n$ in the graph $\mathcal{G}$ (the "size of the boundary set of $s_1, s_2 \ldots, s_k$"), then there are $O(|\mathcal{R}|^{N_k+1})$ operations associated with the visit to k (N_k neighbors plus a loop through the states of x_{s_k}). Therefore, if $N_{\max} = \max\{N_1, N_2 \ldots, N_n\}$, then the maximum-likelihood configuration can be found in

$$O(n|\mathcal{R}|^{N\max+1})$$

operations.

Actually, things can be better than this. Suppose $s_1, s_2, \ldots, s_k$ is not connected (in the graph structure of $\mathcal{G}$). The maximization on x_{s_k} is not necessarily dependent upon all of the bounding variables of the set $x_{s_1} \ldots, x_{s_k}$. In fact, one need only fix the values of the variables at those sites that isolate the *particular connected component containing* s_k. Thus $N_{\max}$ should be interpreted, more favorably, as the *largest boundary of a connected component* created by the site visitation schedule $s_1 \ldots, s_n$.

This makes a substantial difference. In Figure 3, with visitation schedule 1,3,5,7,2,4,6,8,9, no connected set is generated with boundary size greater than one: $N_{\max} = 1$ and only $O(9|\mathcal{R}|^2)$ operations are needed. Of course order matters: any visitation schedule that starts with x_9 will immediately incur $O(|\mathcal{R}|^5)$ computations.

Remarks

1. None of this would really work on a big problem, say with $n = 100$, even if $N_{\max}$ were small. The C functions represent probabilities, and when k is large the probability of any configuration $x_{s_1}, x_{s_2} \ldots, x_{s_k}$ is exponentially small and would generate an underflow. Therefore, in practice we maximize the logarithm of P instead of P itself. Products (like $F_{2,9}(x_2, x_9)C_1(x_2)$ above) are replaced by sums (like $\log F_{2,9}(x_2, x_9) + \log C_1(x_2)$), but otherwise the procedure and the reasoning behind it are the same.

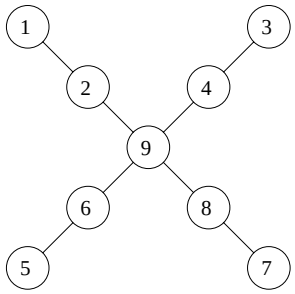

Figure 3: Star-shaped dependency graph.

2. Sometimes, an arg max will produce a tie. These can be decided arbitrarily, in which case one of possibly many maximum probability configurations will be found.

3. There is a more-or-less obvious modification that finds the k most likely configurations, for any $k \geq 1$.

4. Often an *optimal* ordering, in the sense of achieving the minimum $N_{\max}$, is transparent, more or less by inspection. But the problem in general, for arbitrary graphs, is NP-hard. See Arnborg et al. [30].

2.3.2 Marginal and Conditional Probabilities

Often the object of interest is a marginal or conditional probability distribution on a subset of the variables. What is the probability of congestive heart failure in a fifty-year-old male with swollen ankles and pneumonia? What is the probability that a particular decoding, for example the maximum-likelihood decoding, is actually correct given the output of a noisy channel and given that codewords are (say) *a priori* equally likely? These conditional probabilities are quotients of marginal probabilities—probabilities on configurations of subsets of the variables. The coding example requires the marginal probability of the observed channel output, and this involves a summation of probabilities over all possible inputs; for a medical application, we may need to compute marginal probabilities on small subsets of variables, associated with diseases like congestive heart failure and pneumonia, attributes like age and sex, and signs and symptoms like swollen ankles.

The general problem is therefore to compute the probability of a configuration of states for a subset of variables, given a Gibbs distribution on an associated graph $\mathcal{G}$. There is again a dynamic programming principle, operating in much the same way as in the calculation of a most likely configuration. This is probably best illustrated by example.

Consider again Figure 3, and suppose we wish to calculate the marginal distribution on (x_1, x_3). Then for each value of x_1 and x_3 we will need to sum out the other seven variables:

$$\begin{aligned} P(x_1, x_3) &= \sum_{x_2,x_4,x_5,x_6,x_7,x_8,x_9 \in \mathcal{R}} F_{1,2}(x_1, x_2) F_{2,9}(x_2, x_9) F_{3,4}(x_3, x_4) F_{4,9}(x_4, x_9) \\ &\quad F_{5,6}(x_5, x_6) F_{6,9}(x_6, x_9) F_{7,8}(x_7, x_8) F_{8,9}(x_8, x_9) \end{aligned}$$

The apparent computational cost is $|\mathcal{R}|^7$, but if we pay attention to the *order* of summation then this can be reduced to less than $7|\mathcal{R}|^2$. We again define a site visitation schedule, say 5,7,2,4,6,8,9, and this again dictates the sequence of calculations: define

$$\begin{aligned} T_5(x_6) &= \sum_{x \in \mathcal{R}} F_{5,6}(x, x_6) \\ T_7(x_8) &= \sum_{x \in \mathcal{R}} F_{7,8}(x, x_8) \\ T_2(x_1, x_9) &= \sum_{x \in \mathcal{R}} F_{2,9}(x, x_9) F_{1,2}(x_1, x) \\ T_4(x_3, x_9) &= \sum_{x \in \mathcal{R}} F_{4,9}(x, x_9) F_{3,4}(x_3, x) \\ T_6(x_9) &= \sum_{x \in \mathcal{R}} F_{6,9}(x, x_9) T_5(x) \\ T_8(x_9) &= \sum_{x \in \mathcal{R}} F_{8,9}(x, x_9) T_7(x) \end{aligned}$$

and then, finally,

$$\begin{aligned} P(x_1, x_3) &= T_9(x_1, x_3) \\ &= \sum_{x \in \mathcal{R}} T_8(x) T_6(x) T_4(x_3, x) T_2(x_1, x) \end{aligned}$$

In essence, the two schemes, for maximizing and for summing, are the same. Pick a visitation schedule, fix the variables on the boundary of the

connected set containing the current site, and then either maximize or sum. In either case, the number of elementary computations is no worse than

$$O(n|\mathcal{R}|^{N\max+1}).$$

As we have said, conditional probabilities are quotients of marginal probabilities, so conditional probabilities are also amenable to dynamic programming. But this won't always work! At least not when there are a large number of "observed" variables—variables upon which we condition. The problem is again numerical: the *a priori* probability of any one configuration of the observable variables is exponentially small, but this very same probability is the denominator of the quotient representing the desired conditional probability. If, for instance, a block code transmits 1024 bits, then the *unconditioned* probability of receiving any particular 1024-bit word is hopelessly small—much too small to be computed with a summation scheme like the one recommended above. On the other hand, the *conditional* probability of, say, the *most likely* transmitted word, given the received word, will typically be order one. What we are after is a ratio of two very small numbers, and we need to use caution to avoid gross numerical error.

One way around this is to mix the computations of the numerator and denominator in such a way as to avoid exponentially small terms. It turns out that this is easiest to do if we compute the *inverse* of the conditional probability, rather than the conditional probability itself. To illustrate, let us write x for the "unobservable" components and y for the "observable" components and (x, y) for the complete vector of variables. The Gibbs distribution has the form

$$P(x,y) = \prod_{c\in\mathcal{C}} F_c((x,y)_c)$$

and therefore

$$P(x|y) = \frac{\prod_{c\in\mathcal{C}} F_c((x,y)_c)}{\sum_{\tilde{x}\in\mathcal{R}^n}\prod_{c\in\mathcal{C}} F_c((\tilde{x},y)_c)},$$

where n is the dimension of x.

The inverse, $1/P(x|y)$, is then

$$\frac{1}{P(x|y)} = \sum_{\tilde{x}\in\mathcal{R}^n}\prod_{c\in\mathcal{C}}\left(\frac{F_c((\tilde{x},y)_c)}{F_c((x,y)_c)}\right).$$

Our only interest is in efficiently summing over $\tilde{x}$, so let us make things transparent by fixing x and y and writing

$$G_c(\tilde{x}_c) = \frac{F_c((\tilde{x}, y)_c)}{F_c((x, y)_c)}$$

in which case the problem becomes the evaluation of

$$\sum_{\tilde{x} \in \mathcal{R}^n} \prod_{c \in \mathcal{C}} G_c(\tilde{x}_c).$$

The clique structure is the same as we started with, and therefore so is the dynamic programming principle and the number of operations. This time, however, there are no numerical problems. Consequently, we will take this approach when, in §4.2, we compute some posterior probabilities of maximum-likelihood decodings.

3 Linear Graphs

Linear dependency graphs come up in many applications: speech recognition, convolutional coding, filtering and control, among others. In general there is an observation vector y and a "state vector" x, and a joint dependency structure like the one in Figure 2, (a) or (b).

In a speech recognition system, x_t might represent a portion of a phoneme uttered as part of a word, or perhaps even a pair of words, so that the state space is potentially quite large, representing the word or pair of words in addition to the phoneme and phoneme fraction. The observable y_t is some representation or encoding of the associated acoustic signal, or more precisely, the signal as it has been recorded by the microphone. In the speech application, this particular dependency graph comes out of the much-used hidden Markov model, under which

$$P(x_1, \ldots, x_n) = P_1(x_1) \prod_{i=2}^{n} P_i(x_i | x_{i-1})$$

and

$$P(y_1, \ldots, y_n | x_1, \ldots, x_n) \;\; = \;\; \prod_{i=1}^{n} Q_i(y_i | x_i)$$

or

$$P(y_1, \ldots, y_{n-1}|x_1, \ldots, x_n) \;\; = \;\; \prod_{i=1}^{n-1} Q_i(y_i|x_i, x_{i+1})$$

The joint distribution is clearly Gibbs with respect to the graph in Figure 2, (a) or (b), depending on which model is used for $P(y|x)$.

The object of interest is of course the configuration x, and as we have already noticed, its posterior distribution, given y, corresponds to a simple linear graph. So the computational complexity, whether computing a maximum *a posteriori* sequence $x_1, \ldots, x_n$, or the posterior probability of such a sequence, is no worse than $n|\mathcal{R}|^2$ (using a left-to-right or right-to-left visitation schedule). Furthermore, in many applications most transitions, $x_{i-1} \rightarrow x_i$, are ruled out, *a priori*, meaning that $P_i(x'|x'') = 0$ for most $x', x'' \in \mathcal{R}$. This can help substantially: if the software can be structured to efficiently ignore the zero transitions, then instead of $n|\mathcal{R}|^2$ operations, we can expect something more like $n \cdot k|\mathcal{R}|$, where k is the "arity" or number of possible transitions from a state $x \in \mathcal{R}$.

3.1 Convolutional Codes and Memoryless Channels

Convolutional codes also give rise to linear dependency graphs, though the neighborhood structure is generally richer than the nearest-neighbor system of first-order Markov processes. We shall formulate, here, convolutional decoding as an instance of dynamic programming for general graphs, and recover the well-known Viterbi algorithm [5], [31]. There's nothing new in this exercise (in particular, see [7] and [9]), but our general viewpoint does suggest some extensions that may be of some practical value. One could for instance perform exact maximum likelihood decoding even when the channel noise is not white. Such Markov dependency within the error process might afford a good model of bursting (see [20]). In this case, the dynamic programming principle still holds and the maximum likelihood decoding is still computable, at a modest increase in computational cost. Furthermore, the exact posterior probability of the maximum likelihood decoding is also computable, for about as much additional computation as was used for the decoding itself.

Recall that an (n, k, m) convolutional code is defined through a generator

matrix G of the form

$$G = \begin{pmatrix} G_0 & G_1 & G_2 & \dots & G_m & 0 & 0 & 0 & 0 & \dots \\ 0 & G_0 & G_1 & G_2 & \dots & G_m & 0 & 0 & 0 & \dots \\ 0 & 0 & G_0 & G_1 & G_2 & \dots & G_m & 0 & 0 & \dots \\ & & & \ddots & \ddots & \ddots & \dots & \ddots & & \end{pmatrix}$$

where each G_i is a $k \times n$ submatrix. For convenience, we will stick to binary codes, so that the elements of G are in $\{0, 1\}$.

To maintain the connection with §2, we introduce the following (somewhat unusual) notation: $\mathcal{G}_T$ will be the $(T+1)k \times (T+1)n$ upper-left submatrix of G, $x_i \in \{0,1\}^k$ will represent the $i'th$ block of information bits ($0 \le i \le T$), and $v_i \in \{0,1\}^n$ will represent the corresponding block of code (output) bits. If $x = (x_0 \dots x_T)$ and $v = (v_0 \dots v_T)$, then $v = x\mathcal{G}_T$.

Since cliques determine computational complexity, it is useful to observe that

$$\begin{aligned} v_0 &= x_0 G_0 \\ v_1 &= (x_0, x_1) \begin{pmatrix} G_1 \\ G_0 \end{pmatrix} \\ &\vdots \\ v_m &= (x_0 \dots x_m) \begin{pmatrix} G_m \\ \vdots \\ G_0 \end{pmatrix} \\ v_{i+m} &= (x_i \dots x_{i+m}) \begin{pmatrix} G_m \\ \vdots \\ G_0 \end{pmatrix} \quad 1 \le i \le T - m \end{aligned}$$

Thus the dependency of v_i on x has the form

$$v_i = F_i(x_{i-m}, x_{i-m+1}, \dots, x_i),$$

with the understanding that $F_i(Z_0, \dots, Z_m)$ depends only on the last $i+1$ arguments when $0 \le i \le m$.

Suppose the codeword v goes through a channel C and occasionally gets corrupted. Let v_{ij} be the $j'th$ bit of the $i'th$ code block ($j \in \{1, \dots, n\}, i \in$

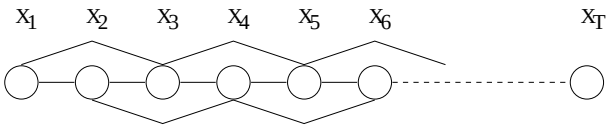

Figure 4: Second order posterior dependency of a convolutional code ($m = 2$).

$\{0, \ldots, T\}$) and let y_{ij} be the corresponding output bit. The usual channel model has the form

$$y_{ij} = C(v_{ij}, \eta_{ij})$$

where η_{ij} are independent and identically distributed ("iid"), so that

$$P(y_0, \ldots, y_T | v_0, \ldots, v_T) = \prod_{i=0}^{T} \prod_{j=1}^{n} P(y_{ij} | v_{ij})$$

Hence

$$\begin{aligned} P(y|x) &= P(y_0, \ldots, y_T | x_0, \ldots, x_T) \\ &= P(y_0, \ldots, y_T | v_0, \ldots, v_T) \\ &= \prod_{i=0}^{T} \prod_{j=1}^{n} P(y_{ij} | v_{ij}) \\ &= \prod_{i=0}^{T} \prod_{j=1}^{n} P(y_{ij} | (F_i(x_{i-m}, \ldots, x_i))_j) \\ &= \prod_{i=0}^{T} H_i(y_i, x_{i-m}, \ldots, x_i) \end{aligned}$$

where $H_i(y_i, x_{i-m}, \ldots, x_i) = \prod_{j=1}^{n} P(y_{ij} | (F_i(x_{i-m}, \ldots, x_i))_j)$.

The clique structure, and hence the computational complexity of dynamic programming, would be unchanged by any of a rich collection of prior models on x, governing the arrival of information bits. Since under the uniform prior, the posterior distribution, $P(x|y)$, is proportional to $P(y|x)$, maximum likelihood decoding is the same as maximum *a posteriori* (MAP) decoding, and the (maximal) cliques are of the form $(i-m, i-m+1, \ldots, i)$. For example, the dependency graph for the case $m = 2$ is depicted in Figure 4. The obvious site visitation schedule is $0, 1, 2, \ldots, T$, which incurs a maximum boundary

of m sites and hence a maximum computational cost of $2^{k(m+1)}$ ($|\mathcal{R}| = 2^k$ since $x_i \in \{0,1\}^k$) at any one site. With T sites the number of operations needed to compute the MAP (or maximum likelihood) decoding is, therefore, $O(T \cdot 2^{k(m+1)})$. This is of course the well-known Viterbi algorithm.

Suppose $\hat{x}$ ($\hat{v} = \hat{x}\mathcal{G}_T$) turns out to be the maximum likelihood block of information bits. It would be a good idea to compute the associated probability that $\hat{x}$ ($\hat{v}$) is correct: $P(\hat{x}|y)$. As we have already seen, more generally in §2.3.2, these probabilities are computed by a straightforward adaptation of the same dynamic programming (Viterbi) algorithm that gave us $\hat{x}$ in the first place.

3.2 Burst Errors

One way to model burst errors is with a Markov noise process $\eta_1, \eta_2, \ldots, \eta_{nT+n}$ where $\eta_t \in \{0,1\}$ and $\eta_t = 1$ represents a transmission error ($0 \to 1$ or $1 \to 0$). The typical state is presumably "0", but an occasional transition occurs to "1", and there is some tendency to stay at "1". If $p_{\alpha\beta} = \text{Prob}(\eta_{t+1} = \beta|\eta_t = \alpha)$ then the situation can be modeled by making p_{01} (the probability of initiating a burst) very small, and $p_{10} = 1/\mu$, μ being the average burst length. The channel model is then completed by introducing an initializing probability $p^o_\alpha = \text{Prob}(\eta_1 = \alpha)$, in which case the model for $\{\eta_t\}$ is

$$P(\eta_1, \eta_2, \ldots, \eta_{nT+n}) = p^o_{\eta_1} \prod_{t=1}^{nT+n-1} p_{\eta_t, \eta_{t+1}}.$$

Many elaborations are possible (as developed, for example, in [20], and connected to graphical models in [7]), including for instance state-dependent bursting in which statistics of $\{\eta_t\}$ depend on the transmitted data $\{v_t\}$, but the Markov model embodied in $p_{\alpha\beta}$ and p^o_α is sensible and, in any case, suitable for illustration.

What are the implications for computing maximum likelihood (or MAP) decodings? Introduce the error indicators

$$\xi_{ij} = \begin{cases} 0 & \text{if } y_{ij} = v_{ij} \\ 1 & \text{else} \end{cases}$$

Then

$$P(y|x) \;=\; P(y_0, \ldots, y_T|x_0, \ldots, x_T)$$

$$
\begin{aligned}
&= P(y_0, \ldots, y_T | v_0, \ldots, v_T) \\
&= \underbrace{p^o_{\xi_{0,1}} \prod_{j=1}^{n-1} p_{\xi_{0,j}\xi_{0,j+1}}}_{\substack{\text{function of} \\ (y_0, v_0)}} \prod_{i=1}^{T} \underbrace{p_{\xi_{i-1,n}\xi_{i,1}}}_{\substack{\text{function of} \\ (y_{i-1}, y_i, v_{i-1}, v_i)}} \underbrace{\prod_{j=1}^{n-1} p_{\xi_{i,j}\xi_{i,j+1}}}_{\substack{\text{function of} \\ (y_i, v_i)}} \\
&= \prod_{i=1}^{T} G_i(y_{i-1}, y_i, v_{i-1}, v_i) \\
&= \prod_{i=1}^{T} G_i(y_{i-1}, y_i, x_{i-m-1}, \ldots, x_i)
\end{aligned}
$$

in light of the relation between v_i and x_i.

Evidently, then, the situation is not much different from the simple iid channel model. This time the maximal cliques have the form $(i-m-1, i-m, \ldots, i)$, which is an expansion over the iid model by only one site, and therefore the most likely decoding and its posterior probability can be computed with $O(T \cdot 2^{k(m+2)})$ operations, or about 2^k times the decoding cost under a white-noise model.

4 Production Systems and Tree-Structured Graphs

From the computational viewpoint, the primary virtue of linear graphs is that the computational cost of dynamic programming grows linearly with the number of variables, even while the configuration space grows exponentially. More general lattice graphs, such as Z_d with $d \geq 2$ and nearest neighbor interactions, behave differently. A $T \times T \times \cdots \times T$ sub lattice of Z_d will achieve a maximum boundary of at least $N_{\max} = T^{d-1}$, no matter what the site visitation schedule. Computation therefore grows exponentially (in T) whenever $d \geq 2$.

In between the linear graph and the lattice graph (with $d \geq 2$) are tree-structured graphs, which fortunately also admit site visitation schedules with bounded maximum boundaries. As an example, consider the tree-structured (but cyclic) dependency graph on X in Figure 5.

Label the sites at level l, from left to right, by s^l_i $1 \leq i \leq 2^{p-l}$ and con-

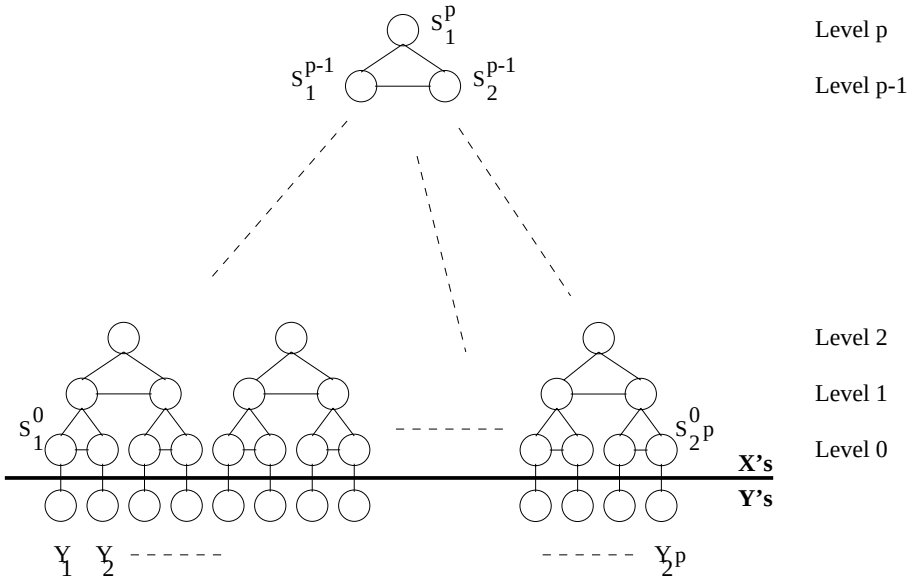

Figure 5: Tree-structured dependency graph. The leaf nodes of the X-graph represent a codeword, from a context-free code, and the Y nodes represent the output of a memoryless channel.

sider the "bottom-up, left-to-right" site visitation schedule: $s_1^0, s_2^0, \ldots, s_{2^p}^0$, $s_1^1, s_2^1, \ldots, s_{2^{p-1}}^1, \ldots, s_1^p$. The largest boundary encountered for any connected component has size $N_{\max} = 2$, *and this is independent of p, the depth of the tree.* Since there are $2^{p+1} - 1$ nodes, the number of dynamic programming operations for computing probabilities and most likely configurations of associated Gibbs distributions is $O(2^{p+1}|\mathcal{R}|^3)$. Thus computation grows linearly with the number of variables for Gibbs distributions on tree-structured graphs.

Because of their computational advantages, tree-structured dependencies are attractive modeling tools (e.g. ([32], [33]). They also come up naturally when working with *production systems*, which define the so-called "context-free" grammars studied in formal linguistics ([4]).

In this section we will introduce a suite of error-correcting codes that are based on, or in any case admit representations in terms of, production systems. (As we shall see, the approach turns out to be nothing more than a

reformulation of Forney's "Squaring Construction," [21]. See also Gore [34], for an earlier but less developed squaring-type construction.) In computational linguistics, the range of the (vector of) leaf-node variables is known as the "yield" or "language." In our application, the range is a set of permissible codewords rather than a set of well-formed sentences. In either application, whether to linguistics or coding, the tree structure is exploited to design efficient computational algorithms.

In way of illustration, let us examine some of the computational consequences of a formal grammar representation of the even-parity code. A context-free grammar (in "Chomsky normal form"—see [4]) consists of a finite set of non-terminal symbols, $\mathcal{N}$, a start symbol $S \in \mathcal{N}$, a finite set of terminal symbols $\mathcal{T}$, and, for every $A \in \mathcal{N}$, a finite set of production rules, each of the form

$$A \to BC \quad B, C \in \mathcal{N}$$

or

$$A \to t \quad t \in \mathcal{T}$$

In general $A \to \alpha \in (\mathcal{N} \cup \mathcal{T})^*$ is allowed, but a reduction to Chomsky normal form is always possible (again, see [4]). Typically, there is a multitude of production rules for each $A \in \mathcal{N}$. The language, or yield, of the grammar is the set of strings of terminals that can be derived from S through repeated application of the production rules. *Probabilistic* grammars include a collection of probability distributions, one for each $A \in \mathcal{N}$, that dictates the choice of production rules. This induces a probability distribution on the yield.

In linguistics, S usually denotes a parsed sentence, where the nonterminals represent sentence fragments, such as noun phrase, verb phrase, prepositional phrase, article, noun, and so-on, and the terminals represent lexical items, typically words. But suppose, instead, that $\mathcal{N} = \{E, O\}$, $\mathcal{T} = \{0, 1\}$, $S = E$, and the production rules are

$$E \to \begin{cases} EE \\ OO \\ 0 \end{cases} \qquad O \to \begin{cases} EO \\ OE \\ 1 \end{cases} \tag{3}$$

Then, evidently, the yield is exactly the set of all nonempty, finite, binary strings with an even number of ones.

We can connect this to Markov random fields, and block codes, by fixing a binary graph structure, assigning E $(= S)$ to the root node, assigning

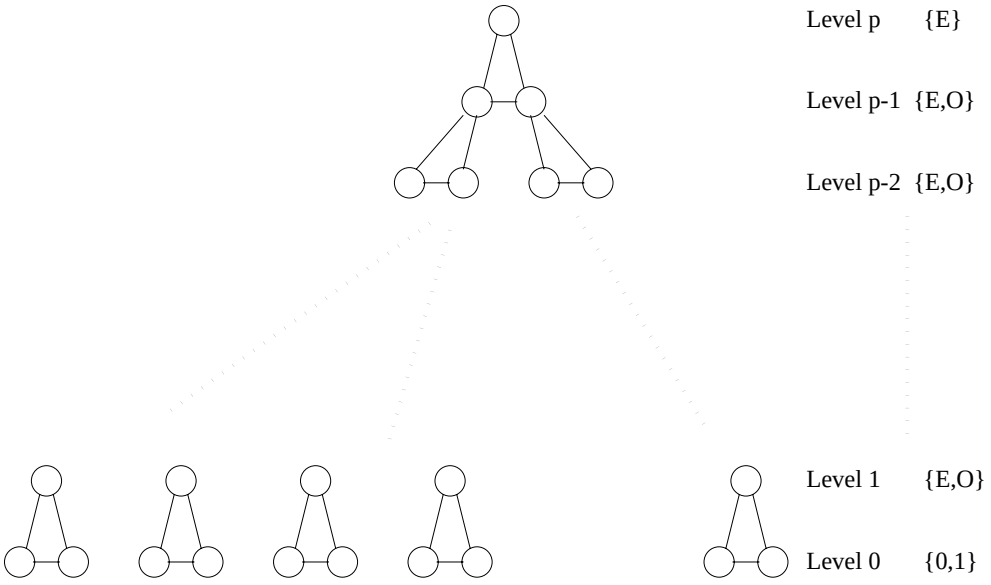

Figure 6: Even-parity code viewed as the yield of a context-free grammar.

terminal states to the leaf nodes, and assigning non-terminal states to the non-leaf nodes. Consider, for example, the balanced binary tree in Figure 6. Instead of (3), we adopt the production rules

$$E \to \begin{cases} EE \\ OO \end{cases} \qquad O \to \begin{cases} EO \\ OE \end{cases} \tag{4}$$

at levels ≥ 2, and

$$E \to \begin{cases} 00 \\ 11 \end{cases} \qquad O \to \begin{cases} 01 \\ 10 \end{cases} \tag{5}$$

at level 1. A sequence of $2^p - 1$ information bits can be turned into a state configuration on the graph by assigning each bit to one of the $2^p - 1$ triangles in the tree. The bit associated with the apical (root) triangle is used to choose between the productions $E \to EE$ and $E \to OO$. This fixes the states of the daughter sites (at level $p - 1$), and then two more information bits are read and two more productions are applied (to the level $p-1$ states), thereby determining the states of the four level $p-2$ sites. Encoding continues through the $2^p - 1$ information bits, resulting in a specification of states at

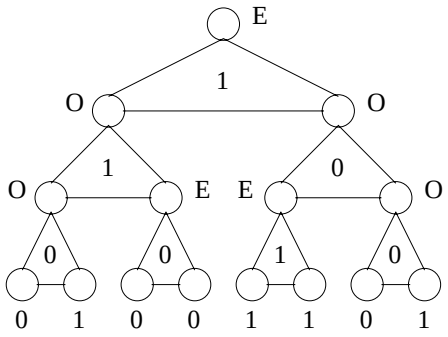

Figure 7: Even-parity code. The information sequence 1100010 is coded as 01001101.

every site in the graph. In this manner, a length 2^p even-parity codeword is produced at the leaf nodes, inducing a one-to-one correspondence between sequences of $2^p - 1$ information bits and length 2^p even-parity codewords. A simple example, with $p = 4$, is given in Figure 7, where information bits have been placed in the center of the triangles, and where the conventions "bit=0 $\Rightarrow$ apply first production" and "bit=1 $\Rightarrow$ apply second production" have been used to translate from information bits into states, via equations 4 and 5.

The random variables are $X_{s_i^l}$, $0 \le l \le p$, $1 \le i \le 2^{p-l}$, with state space $\{S\}$ when $l = p$, $\{E, O\}$ when $0 < l < p$, and $\{0, 1\}$ when $l = 0$. The production rules induce clique functions—one for each triangle. In most instances the "natural prior" is the uniform prior (in which case maximum-likelihood decoding is MAP decoding), which results, evidently, from a sequence of iid "balanced" (.5/.5) information bits. The corresponding clique functions are $F_c(x_{s_1^p}, x_{s_1^{p-1}}, x_{s_2^{p-1}}) = .5$ if $x_{s_1^{p-1}} = x_{s_2^{p-1}}$ and 0 otherwise ($x_{s_1^p}$ is always S), for the apical (root) node triangle; and $F_c(x_{s_i^l}, x_{s_{2i-1}^{l-1}}, x_{s_{2i}^{l-1}}) = .5$ if $x_{s_i^l}$ is the parity of $x_{s_{2i-1}^{l-1}} + x_{s_{2i}^{l-1}}$ and 0 otherwise, for the remaining triangles.

In summary, Figure 6 depicts a MRF ("X") in which the marginal distribution on leaves concentrates on the even parity code of length 2^p; the marginal distribution depends on the distribution on information bits; and in particular, the marginal distribution is uniform when the distribution on information bits is uniform.

Imagine now a channel, let us say memoryless for the time being, through which the leaf-node variables are transmitted and possibly corrupted, and eventually received as $y = (y_1, \ldots, y_{2^p})$. If for example

$$y_i = x_{s_i^0} + \eta_i$$

with $\eta_1, \eta_2, \ldots, \eta_{2^p}$ iid $N(0, \sigma^2)$, for some variance σ^2, then

$$p(y|x) = \prod_{i=1}^{2^p} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\{-(y_i - x_{s_i^0})^2/2\sigma^2\}$$

and the joint dependency graph has the structure depicted in Figure 5.

It is immediately evident that the posterior distribution has the same structure as the prior, and hence, in light of our earlier discussion about dynamic programming on trees, the most likely (or MAP) even-parity word $(\hat{x})$ can be computed in "linear time"—the number of operations grows linearly with the dimension $(2^p - 1)$ of the code. Of course there are other efficient ways to "soft decode" the even parity code, but bear in mind that the "reliability," $p(\hat{x}|y)$, is also calculated for about the same cost.

Obviously, it would be desirable to extend this representation beyond the even-parity code. One way to do this is to generalize the production system ((4), (5)) that generates the even parity code. In §4.1 we will formulate production systems that yield other (usually familiar) error-correcting codes. Then, in §4.2, we will study the computational implications for computing maximum likelihood (or MAP) decodings, and for computing the probability that the decoding is correct, given transmission across a memoryless channel. In §4.3 we will examine the extensions to Markov models for bursty channels.

4.1 Context-Free Codes

There are many ways to generalize. Here we will stick to binary codes, balanced, binary, tree-structured graphs, and a symbol set (state space) that depends only on level, l. We will assume that the number of productions per symbol also depends only on the level, and furthermore we will restrict ourselves to block codes with minimum distance 2^α, for some $\alpha = 0, 1, 2, \ldots, p$. If the number of productions per symbol is always a power of two, then the encoding scheme that we used for the even parity code generalizes directly: just "peel off" the number of bits needed to specify a production (this

depends only on the level—not on the state), working down from the top. Obviously, there are other, less restrictive ways to map information bits into productions, but we will confine ourselves to this simple case by devising production rules in which the number of choices depends only on the level and is always a power of two.

Look again at Figure 6. At every (non-terminal) node, each possible state can itself be thought of as representing a code—namely the set of derivable terminal sequences of the *subtree* defined by the chosen node. Site s_i^l, for example, is the root node for the terminal sequence of sites $s^0_{2^l(i-1)+1}, \ldots, s^0_{2^l i}$, and $x_{s_i^l} = E$ (resp. O) if and only if $x_{s^0_{2^l(i-1)+1}}, \ldots, x_{s^0_{2^l i}}$ has even (resp. odd) parity. In this way, an E at level l represents the even parity code of length 2^l, and an O represents the corresponding odd parity code.

More generally, let A_i^l $i \in \{0, 1, 2, \ldots, m-1\}$ be the nonterminal symbols (states) of a level-l site. (Later, m will depend on l, $m = m^l$.) If we want a code with minimum distance 2^α, then, evidently, the yield of A_i^l, as expressed at the terminal sites $s^0_{2^l(i-1)+1}, \ldots, s^0_{2^l i}$, must itself be a code with minimum distance no smaller than 2^α. Taking a hint from the even parity code, suppose that each A_i^l represents a length 2^l, distance 2^α, code (just as E and O at level l represent length 2^l, distance $2^1 = 2$, codes), and that *the distance between any two of these codes*, $d(A_i^l, A_k^l)$ $i \neq k$, is at least $2^{\alpha-1}$ (just as the even and odd parity codes are distance $2^{1-1} = 1$ apart). Then the productions

$$A_i^{l+1} \rightarrow A^l_{(i+j) \bmod m} A_j^l \qquad j = 0, 1, \ldots, m-1$$

define m level $l+1$ symbols ($i = 0, 1, \ldots, m-1$), each of which represents a length 2^{l+1}, distance 2^α, subtree code, and, furthermore, $d(A_i^{l+1}, A_k^{l+1}) \geq 2^{\alpha-1}$ whenever $i \neq k$. Each code A_i^{l+1} is just a union of Cartesian products, so the construction could be written, more traditionally, as

$$A_i^{l+1} = \bigcup_{j=0}^{m-1} A^l_{(i+j) \bmod m} \times A_j^l$$

where $\times$ represents Cartesian product (concatenation):

$$A \times B = \{(a, b) : a \in A, b \in B\}$$

What we have is just one example of Forney's (twisted) "squaring construction" [21]. The even-parity code is a special case with $A_0^l = E$, $A_1^l =$

O, and $m = 2$. There is nothing special about the particular "twisting" $(i + j) \bmod m$, and in fact other ways of combining i and j are possible and often desirable, as we shall see shortly. The point for now is that such constructions can be iterated, "bottom up," thereby defining symbols and productions all the way up to the root node at level p (where any of the m states A_i^p $i = 0, 1, \ldots, m-1$ could be used as a start symbol). These productions define a tree-structured dependency graph, and through this, a ready-made computational engine for soft decoding and other decoding calculations.

Of course the construction assumes that we start with a set of level-l codes, A_i^l, with certain distance properties and certain separations. One way to get at this is to generalize even further: replace the set of separated codes A_i^l $i = 0, 1 \ldots m-1$, by a hierarchy of codes with a hierarchy of separations. This more elaborate representation falls out of a hierarchical representation for $\mathbf{Z}_2^{2^l}$ itself, which we will now construct.

The elements of $\mathbf{Z}_2^{2^l}$ will be represented by symbols $A^l_{i_0,i_1,\ldots,i_l}$. The integer coefficients $i_0, \ldots, i_l$ are restricted by $0 \le i_k \le m_k^l - 1$, where $m_k^l = 2^{\binom{l}{k}}$, for each $k = 0, \ldots, l$. There are, then, $\prod_{k=0}^{l} m_k^l = 2^{2^l}$ vectors $(i_0, \ldots, i_l)$; we will set up a one-to-one correspondence between these and the 2^{2^l} elements of $\mathbf{Z}_2^{2^l}$.

The correspondence is built inductively. Start by representing $\mathbf{Z}_2^1 = \mathbf{Z}_2^{2^0}$ with the symbols $A_{i_0}^0$ $0 \le i_0 \le 1$ ($m_0^0 = 2^{\binom{0}{0}} = 2$):

$$A_0^0 = 0, \quad A_1^0 = 1$$

Now build the representation for $\mathbf{Z}_2^{2^l}$ from an already-built representation for $\mathbf{Z}_2^{2^{l-1}}$, through the formula:

$$\begin{aligned} A^l_{i_0,\ldots,i_l} \quad = \\ & A^{l-1}_{c(\lfloor i_1/m_1^{l-1} \rfloor, i_0, m_0^{l-1}), c(\lfloor i_2/m_2^{l-1} \rfloor, i_1, m_1^{l-1}), \ldots, c(\lfloor i_{l-1}/m_{l-1}^{l-1} \rfloor, i_{l-2}, m_{l-2}^{l-1}), c(i_l, i_{l-1}, m_{l-1}^{l-1})} \\ & \times A^{l-1}_{\lfloor i_1/m_1^{l-1} \rfloor, \lfloor i_2/m_2^{l-1} \rfloor, \ldots, \lfloor i_{l-1}/m_{l-1}^{l-1} \rfloor, i_l} \end{aligned} \tag{6}$$

where $\lfloor x \rfloor = \sup\{n : n \text{ integer}, \ n \le x\}$ and $c(n, m, k) = (n + m) \bmod k$. In way of example, let us work through the particulars when $l = 1$ ($\mathbf{Z}_2^2$) and

$l = 2$ ($\mathbf{Z}_2^4$):

$$\begin{array}{rclcrcl} A^1_{0,0} & = & A^0_{c(0,0,2)} \times A^0_0 & = & A^0_0 \times A^0_0 & = & (00) \\ A^1_{0,1} & = & A^0_{c(1,0,2)} \times A^0_1 & = & A^0_1 \times A^0_1 & = & (11) \\ A^1_{1,0} & = & A^0_{c(0,1,2)} \times A^0_0 & = & A^0_1 \times A^0_0 & = & (10) \\ A^1_{1,1} & = & A^0_{c(1,1,2)} \times A^0_1 & = & A^0_0 \times A^0_1 & = & (01) \end{array}$$

and

$$A^2_{i_0,i_1,i_2} = A^1_{c(\lfloor i_1/2 \rfloor, i_0, 2), c(i_2,i_1,2)} \times A^1_{\lfloor i_1/2 \rfloor, i_2} \Longrightarrow$$

$$\begin{array}{cccc} A^2_{0,0,0} = (0000) & A^2_{0,1,0} = (1100) & A^2_{0,2,0} = (1010) & A^2_{0,3,0} = (0110) \\ A^2_{0,0,1} = (1111) & A^2_{0,1,1} = (0011) & A^2_{0,2,1} = (0101) & A^2_{0,3,1} = (1001) \end{array}$$

$$\begin{array}{cccc} A^2_{1,0,0} = (1000) & A^2_{1,1,0} = (0100) & A^2_{1,2,0} = (0010) & A^2_{1,3,0} = (1110) \\ A^2_{1,0,1} = (0111) & A^2_{1,1,1} = (1011) & A^2_{1,2,1} = (1101) & A^2_{1,3,1} = (0001) \end{array}$$

The representation is useful because it makes explicit a hierarchy of distance properties, as can be verified inductively:

$$d(A^l_{i_0,\ldots,i_{\alpha-1},i_\alpha,\ldots,i_l}, A^l_{i_0,\ldots,i_{\alpha-1},j_\alpha,\ldots,j_l}) \geq 2^\alpha \tag{7}$$

whenever $j_\alpha \neq i_\alpha$. An immediate consequence is that $\mathbf{Z}_2^{2^l} = \{A^l_{i_0 \ldots i_l} : 0 \leq i_k \leq m^l_k - 1\}$ $\forall l \geq 0$, since each $A^l_{i_0 \ldots i_l}$ is evidently distinct and there are just the right number of them. Beyond this, the distance properties lead more-or-less directly to a hierarchy of codes and separations (as we shall see momentarily), and this hierarchy, in turn, leads to the representation of various codes in terms of production rules. The reader may recognize the representation as a version of Forney's iterated squaring construction [21], albeit with a new notation. The combinatorial function $c(n, m, k)$ is not particularly special, and many others could be used (corresponding to different "twistings"). In [21] Forney introduced an especially canonical twisting (see appendix §A) for which we will later develop an exact coarse-to-fine version of the dynamic programming algorithm (in §5). But let us first develop the sought-after grammatical representation, since this is independent of the particular twisting (and, hence, combinatorial function) used.

Right away we get a hierarchy of codes and separations, just by taking unions over indices: for each l and each $i_0, \ldots, i_{\alpha-1}$ ($\alpha \leq l$), define

$$A^l_{i_0,\ldots,i_{\alpha-1}} = \bigcup_{i_\alpha,\ldots,i_l} A^l_{i_0,\ldots,i_l} \tag{8}$$

Evidently, in light of (7), $A^l_{i_0,\ldots,i_{\alpha-1}}$ is a distance 2^α code (with $\prod_{i=\alpha}^{l} m^l_\alpha$ codewords), and furthermore

$$d(A^l_{i_0,\ldots,i_{k-1},i_k,\ldots,i_{\alpha-1}}, A^l_{i_0,\ldots,i_{k-1},j_k,\ldots,j_{\alpha-1}}) \geq 2^k \tag{9}$$

whenever $j_k \neq i_k$. What's more, the construction of $\{A^l_{i_0,\ldots,i_l}\}$ from $\{A^{l-1}_{i_0,\ldots,i_{l-l}}\}$ (see equation (6)) translates into a construction of $\{A^l_{i_0,\ldots,i_{\alpha-1}}\}$ from $\{A^{l-1}_{i_0,\ldots,i_{\alpha-l}}\}$: If $\alpha \leq l$ then

$$A^l_{i_0,\ldots,i_{\alpha-1}} = \bigcup_{i=0}^{m^{l-1}_{\alpha-1}-1}$$

$$A^{l-1}_{c(\lfloor i_1/m_1^{l-1}\rfloor, i_0, m_0^{l-1}), c(\lfloor i_2/m_2^{l-1}\rfloor, i_1, m_1^{l-1}), \ldots, c(\lfloor i_{\alpha-1}/m_{\alpha-1}^{l-1}\rfloor, i_{\alpha-2}, m_{\alpha-2}^{l-1}), c(i, i_{\alpha-1}, m_{\alpha-1}^{l-1})}$$

$$\times A^{l-1}_{\lfloor i_1/m_1^{l-1}\rfloor, \lfloor i_2/m_2^{l-1}\rfloor, \ldots, \lfloor i_{\alpha-1}/m_{\alpha-1}^{l-1}\rfloor, i}$$

We have therefore, for any $\alpha \geq 0$, a production system:

$$A^l_{i_0,\ldots,i_{\alpha-1}} \rightarrow$$

$$A^{l-1}_{c(\lfloor i_1/m_1^{l-1}\rfloor, i_0, m_0^{l-1}), \ldots, c(\lfloor i_{\alpha-1}/m_{\alpha-1}^{l-1}\rfloor, i_{\alpha-2}, m_{\alpha-2}^{l-1}), c(i, i_{\alpha-1}, m_{\alpha-1}^{l-1})} A^{l-1}_{\lfloor i_1/m_1^{l-1}\rfloor, \ldots, \lfloor i_{\alpha-1}/m_{\alpha-1}^{l-1}\rfloor, i}$$

(for $l \geq \alpha,\ \forall i = 0, 1, \ldots, m^{l-1}_{\alpha-1} - 1$)

$$A^l_{i_0 \ldots i_l} \rightarrow$$

$$A^{l-1}_{c(\lfloor i_1/m_1^{l-1}\rfloor, i_0, m_0^{l-1}), \ldots, c(\lfloor i_{l-1}/m_{l-1}^{l-1}\rfloor, i_{l-2}, m_{l-2}^{l-1}), c(i_l, i_{l-1}, m_{l-1}^{l-1})} A^{l-1}_{\lfloor i_1/m_1^{l-1}\rfloor, \ldots, \lfloor i_{l-1}/m_{l-1}^{l-1}\rfloor, i_l}$$

(for $l < \alpha,$ productions are "singletons")

(10)

which, along with the conventions $A^0_0 = 0$ and $A^0_1 = 1$, yields a distance 2^α code $A^l_{i_0 \ldots i_{\alpha-1}}$ for every $l \geq \alpha$ and every $i_0 \ldots i_{\alpha-1}$, $0 \leq i_k \leq m^l_k - 1$. The meaning of the $l < \alpha$ case is that, from level $\alpha - 1$ down, no more information bits are encoded; the codeword is already determined by the configuration at level $\alpha - 1$.

We started with the even-parity code. To recover the even-parity code, as a special case, just take $\alpha = 1$ and identify A^l_0 with E and A^l_1 with O for all $l \geq 1$.

Reed–Muller codes: a canonical class of context-free codes. Since, as Forney has shown [21], the Reed–Muller codes can be derived from an iterated squaring construction, it is natural to seek a grammatical representation for them. It turns out that the Reed–Muller grammar is identical to the grammar presented above, with the sole exception being that a different choice of combinatorial function (or twisting) is required. We present a brief derivation of this fact in appendix A. In section §5, we will exploit the grammatical structure of the Reed–Muller codes to develop a more efficient maximum likelihood decoder based on the notion of coarse-to-fine dynamic programming.

4.2 Memoryless Channels

The random variables situated at the leaf nodes, $x_{s_1^0}, x_{s_2^0}, \ldots, x_{s_{2^p}^0}$, make up the codeword—the bits that are actually transmitted. Returning to the channel models of §3, the simpler model corrupts bits independently (memoryless channel):

$$y_i = c(x_{s_i^0}, \eta_i) \qquad 1 \leq i \leq 2^p$$

where $y_1, \ldots, y_{2^p}$ represents the received signal and $\eta_1, \ldots, \eta_{2^p}$ is an independent noise process. In this case

$$P(y|x) = \prod_{i=1}^{2^p} P(y_i|x_{s_i^0})$$

and we get the joint (X, Y) dependency graph depicted in Figure 5.

We ran some experiments with the Reed-Muller RM(2,6) code ($p = 6$, $\alpha = 4$), transmitted at low signal-to-noise ratio through a Gaussian channel: $p(y_i|x_{s_i^0}) \sim N(2x_{s_i^0} - 1, 1)$. Following tradition, we have assumed BPSK modulation ($0 \rightarrow -1$, $1 \rightarrow 1$). For the memoryless channel, the dependency graph of the posterior, $P(x|y)$, is the same as the dependency graph of the prior, $P(x)$, and is tree-structured under the grammatical representation of RM(2,6). See Figure 5, where for the purpose of organizing the computation we can simply ignore the sites belonging to the received vector, y. In each of 50 trials we computed both the maximum likelihood decoding, $\hat{x}$, and the associated "reliability," $p(\hat{x}|y)$. As suggested earlier, we used the "bottom-up" site visitation schedule $s_1^0, \ldots, s_{64}^0$, $s_1^1, \ldots, s_{32}^1$, $s_1^2, \ldots, \ldots, s_4^4$, s_1^5, s_2^5, s_1^6, which

RM(2,6) [64,22,16] AWGN with BPSK modulation, $\sigma = 1$
Dynamic Programming operations: 79,231/decoding
(equal number for $p(\hat{x}|y)$)

Trial	$d(\tilde{x}, \hat{x})$	$p(\hat{x}\|y)$	Trial	$d(\tilde{x}, \hat{x})$	$p(\hat{x}\|y)$	Trial	$d(\tilde{x}, \hat{x})$	$p(\hat{x}\|y)$
1	0	1.0000	7	0	1.0000	25	0	1.0000
2	0	0.9880	8	0	0.9844	30	0	1.0000
3	16	0.3790	9	0	0.9548	35	0	0.9983
4	0	1.0000	10	0	0.9981	40	0	0.9994
5	0	0.9990	15	0	1.0000	45	0	0.9998
6	0	0.9974	20	0	0.9999	50	0	0.9688

Table 1: RM(2,6) with BPSK modulation and added white Gaussian noise. Typical decodings and their posterior probabilities. First column is trial number; second column is Hamming distance between the maximum-likelihood decoding and the correct codeword; third column is the probability assigned to the decoding under the posterior distribution. At $\sigma = 1$, maximum likelihood yielded one decoding error (in trial #3) in 50 trials. Notice that the low posterior probability (.3790) signals an error. The next lowest probability was .5866, belonging to a correct decoding.

makes for about 80,000 operations per decoding cycle and per posterior probability calculation. In computing $p(\hat{x}|y)$, we were careful to avoid underflow, by following the recipe given at the end of §2.3.2.

The results are summarized in Table 1, where $\tilde{x}$ represents the correct decoding and $d(\tilde{x}, \hat{x})$ is the Hamming distance to the maximum likelihood decoding. Notice that the single decoding error is well anticipated by the low posterior probability, $P(\hat{x}|y) < .4$.

4.3 Burst Errors

What are the computational implications of a channel model with Markov memory? Let us take another look at the model for burst errors discussed

earlier in §3.2, but this time in the computational context of a tree-structured prior.

The channel model introduced in §3.2 represents bursts by a Markov error process $\eta_i \in \{0,1\}$ such that $\eta_i = 1 \leftrightarrow y_i \neq x_{s_i^0}$. In terms of the initial probability distribution, p_α^o $\alpha \in \{0,1\}$, on η_1, and the transition probability matrix $\{p_{\alpha\beta}\}_{\alpha,\beta\in\{0,1\}}$ for $\text{Prob}(\eta_{t+1} = \beta | \eta_t = \alpha)$, the channel model can be written

$$P(y|x) = P(y_1, \ldots, y_{2^p} | x_{s_1^0} \ldots x_{s_{2^p}^0}) = p_{\xi_1} \prod_{i=1}^{2^p-1} p_{\xi_i \xi_{i+1}}$$

where $\xi_i = 0$ if $y_i = x_{s_i^0}$, and $\xi_i = 1$ otherwise. Of course there are other models, that perhaps make more sense in a particular application ([20]), but the computational costs will be the same or similar for many of these variations.

Since $P(x,y) = P(y|x)P(x)$, we get the joint dependency graph, for (X,Y), by starting with the (tree-structured) dependency graph for X, and appending with sites for Y and with cliques introduced in the channel model, $P(y|x)$. Since ξ_i is a function of $x_{s_i^0}$ and y_i, and since ξ_{i+1} is a function of $x_{s_{i+1}^0}$ and y_{i+1}, $p_{\xi_i\xi_{i+1}}$ introduces a clique made up of the four sites associated with the four variables $x_{s_i^0}, y_i, x_{s_{i+1}^0}$ and y_{i+1}. Taking this into account, and taking into account the tree-structured prior on X, we arrive at the joint (X,Y) dependency structure depicted in Figure 8(a). As we have noted before, conditioning does not change the graph structure. The *posterior* dependency graph comes from simply removing the sites representing the received signal, Y, as is depicted in Figure 8(b).

Compare the dependency graph on X in Figure 5 to Figure 8(b). Channel memory introduces new neighbors among the leaf nodes. An optimal ordering of site visits is no longer obvious, and may in fact depend upon the various sizes of state spaces at the various sites. But there is a particular ordering that does the job at a cost of no more than 16 times the cost of decoding the simpler memoryless channel analyzed earlier. Before writing down this particular visitation schedule, which may appear at first glance to be rather complex and arbitrary, it might be better to first describe, less formally, the resulting dynamic programming process.

The optimization (soft decoding) is based on subtrees. The subtree rooted at s_i^l ($2 \leq l \leq p$ and $1 \leq i \leq 2^{p-l}$) is made up of s_i^l and the two "daughter" subtrees rooted at s_{2i-1}^{l-1} and s_{2i}^{l-1}. Suppose that for each of the two daugh-

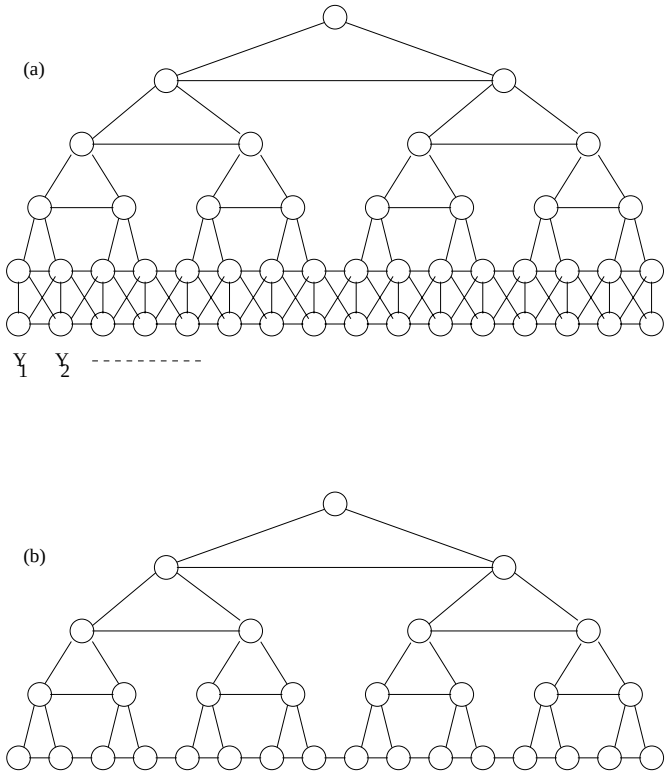

Figure 8: Tree-structured code through a Markov channel. (a) Joint (X, Y) dependency structure. (b) Dependency structure on X, under $P(x|y)$.

ter subtrees the optimal "interior" has been calculated, which is to say the optimal assignments of states conditioned upon all possible assignments at the triangles' respective corners—the root (s^{l-1}_{2i-1} or s^{l-1}_{2i}) and, for each root, the corresponding pair of corners sitting at level 0 (($s^0_{(i-1)2^l+1}, s^0_{(i-1)2^l+2^{l-1}}$) or ($s^0_{(i-1)2^l+2^{l-1}+1}, s^0_{i2^l}$)). Then the optimal interior of the level-l subtree can be computed by "merging" the daughter subtrees. The merging involves visiting (maximizing over) the right-most corner of the left daughter subtree, the left-most corner of the right daughter subtree, the root node of the left daughter subtree, and then finally the root node of the right daughter subtree. Merging continues, "upwards", until eventually the optimal interior of the entire graph is calculated, conditioned on the configuration at the graph corners s^p_1,

s_1^0, and $s_{2^p}^0$. The global optimum is then computed after a maximization at each of these three remaining sites.

The procedure is summarized more formally through a set of do-loops defining the visitation schedule:

Do $l = 2, p$ % loop over levels
  Do $i = 1, 2^{p-l}$ % loop over sites at level l
% Create a connected component of "interior" sites of the subtree
% rooted at s_i^l by "merging" the subtrees rooted at s_{2i-1}^{l-1} and s_{2i}^{l-1}
    Visit $s^0_{(i-1)2^l+2^{l-1}}$ % visit the right-most corner
    % of the subtree rooted at s_{2i-1}^{l-1}
    Visit $s^0_{(i-1)2^l+2^{l-1}+1}$ % visit the left-most corner
    % of the subtree rooted at s_{2i}^{l-1}
    Visit s_{2i-1}^{l-1} % visit the root node
    % of the left subtree
    Visit s_{2i}^{l-1} % visit the root node
    % of the right subtree
  End do
End do
% All that remains are the corners of the tree...
Visit s_1^0
Visit $s_{2^p}^0$
Visit s_1^p

In effect, every production from s_i^l incurs a 16-fold increase in computation cost: the two remaining interior leaf nodes (right node of the left daughter and left node of the right daughter), with four configurations, are visited once for each of the four configurations of the two leaf nodes that sit at the corners of the s_i^l subtree.

As usual, the ordering (and computational analysis) applies equally well for probability calculations, including the posterior $p(\hat{x}|y)$.

5 Thinning and Exact Coarse-to-Fine

Dynamic programming can get out of hand, computationally. The graph structure, for example, may be bad: there may be no site visitation schedule that avoids large boundaries. The two-state two-dimensional Ising model,

on the $N \times N$ square lattice, is the prototypical example (cf. [35]). The worst loops have at least $O(2^{N+1})$ operations, no matter in what order sites are visited, and interesting lattices begin with $N = 100$ or even $N = 1000$. But even a good graph structure does not guarantee a feasible computational problem. Sometimes the state spaces are too large. The codes of §3 and §4 have good graph structures but in some instances very large state spaces.

How large are the state spaces under the grammatical representations of the Reed-Muller-like codes introduced in §4? Fix p (code length 2^p) and fix $\alpha \leq p$ (code distance 2^α). The computation of the number of states at a particular site does require some care, since not every available symbol is actually used. At level p there are $m_0^p \cdot m_1^p \cdots m_{\alpha-1}^p$ symbols, but only one is actually used—the designated "start" or "sentence" symbol, S. Still, the number of symbols employed at a given site is independent of the *particular* start symbol, and in fact depends only on the level, l (see [36]):

$$N^l = \#\text{ states of a level-}l\text{ site} = \begin{cases} 1 & l = p \\ \prod_{k=\max(0,l+\alpha-p)}^{\min(l,\alpha-1)} m_k^l & 0 \leq l \leq p-1 \end{cases}$$

This leads to large state spaces, even for more-or-less modest values of p. The Reed-Muller code RM(3,7), with $p = 7$ (length 128) and $\alpha = 4$ (distance 16), generates N^l = 1, 4, 16, 256, 16,384, 1,048,576, 1,048,576, 1 states, at levels l = 0, 1, 2, 3, 4, 5, 6, 7, respectively. The worst of it, from the dynamic programming viewpoint, would appear to be the loops associated with the level 5-6 cliques, which are triangles involving one level-6 site and two level-5 sites. Naively, there are $1,048,576^3 = 2^{60}$ operations in this loop, and this is of course infeasible.

But the calculation is naive since the clique function is *zero* for the vast majority of these triples. Non-zero contributions are made, only, by those pairs of level-5 symbols that represent an allowed production from a level-6 symbol. The number of operations associated with a triangle is the number of symbols at the apex level times the number of productions per symbol. If, therefore, we loop first over productions, given a symbol at the apex, and then over symbols, then the number of operations will be $N^l \cdot I^l$, where I^l is the number of productions per level-l symbol. This latter number is independent of both the particular symbol and the particular level-l site, as is evident from the production formulas. For the Reed-Muller codes, $I^l = m_{\alpha-1}^{l-1}$ for $l \geq \alpha$ and 1 for $l < \alpha$, so that in the particular example $p = 7,\ \alpha = 4$ there are 1,

1, 1, 2, 16, 1024, and 1,048,576 productions for levels 1, 2, 3, 4, 5, 6, and 7, respectively.

The product, $N^l \cdot I^l$, is biggest when $l = 6$: each of the two level-6 sites contributes about $1,048,576 \times 1024 = 2^{30}$ operations. The cost of decoding, or of evaluating a posterior probability, is about

$$\sum_{i=1}^{7} (2^{7-l}) N^l I^l$$

since 2^{7-l} is the number of sites at level l, and this is about 2^{31} operations. This may be feasible, but it would be impractical in most applications.

We will suggest a few remedies. One (coarse-to-fine dynamic programming) is more or less generic, in that it applies in principle to any problem of finding a most likely configuration under a probability with a given graphical dependency structure. Another (thinning) is special to the problem at hand: decoding the grammatical codes introduced in §4.

5.1 Thinning

As we have just seen, the computational cost of maximum likelihood decoding or computing a posterior probability for a typical context-free code like $RM(p - \alpha, p)$ transmitted across a memoryless channel is a simple function of the state space cardinality and number of productions. Although the CTFDP algorithm (see §5.2) can reduce decoding complexity, sometimes by several orders of magnitude, the really large context-free codes are still undecodable, at least from a practical point of view.

Consider for example the Reed–Muller code RM(6,10). With an information rate of 848/1024 and distance 16, its (maximum-likelihood) decoding complexity exceeds the level 9 contribution of $2m_3^9 m_3^8 = 2^{141}$ operations! RM(6,10) is patently undecodable. But suppose one systematically pruned its grammar, discarding productions (and associated information bits) to reduce decoding complexity. What sort of code would emerge from this process? Alternatively, one could imagine imposing strict limits on the cardinality of each level's state space and inquiring whether the resulting system remained a consistent context-free grammar. These equivalent approaches yield a family of context-free codes that we will refer to as *thinned* codes.

We present a brief introduction to thinned codes in appendix §B (though a fuller treatment is available in [36]). The thinned Reed–Muller code $RM^{(n)}(p-$

$\alpha, p)$, defined to be a subcode of $RM(p-\alpha, p)$ with the number states (or productions) at any level not exceeding 2^n, is readily decodable by the exact maximum likelihood decoding algorithms of §4 in at most (loosely) 2^{p+2n} operations. For example, $RM^{(8)}(6, 10)$, a linear $[1024, 440, 16]$ subcode of the undecodable $[1024, 848, 16]$ code RM(6,10), is decodable in approximately 2^{22} operations. In other words, by discarding only half the information bits from RM(6,10) we can decode the remaining thinned code at least 2^{120} times faster. Moreover, using the coarse-to-fine approach of §5.2, $RM^{(8)}(6, 10)$ can be decoded an additional 3 to 30 times faster depending on the signal-to-noise (see Table 2).

As a general rule, thinned Reed–Muller codes are poor codes in terms of coding gain and other performance measures. However, they are useful in the context of context-free codes, because they allow one to vary decoding complexity (often by orders of magnitude) by simply altering a single parameter. One speculative direction for future inquiry is the following problem. Given a thinned Reed–Muller code with a known and manageable decoding complexity, can one find a set of combinatorial functions (or twistings) that optimize the coding gain of the iterated squaring construction?

A far more promising approach than thinning for the maximum-likelihood decoding of context-free codes is the method of coarse-to-fine dynamic programming.

5.2 Coarse-to-Fine Dynamic Programming

Coarse-to-fine dynamic programming (CTFDP) is what Pearl [37] would call an "admissible heuristic," meaning that it is a variation on dynamic programming that is meant to save operations in a typical problem (it is a "heuristic"), but, at the same time, it is guaranteed to solve the optimization problem (it is "admissible"). The well-known A^* is an admissible heuristic, as is the Iterated Complete Path (ICP) algorithm of Kim and Kopac [38], [39]. Coarse-to-fine dynamic programming is a kind of hierarchical, or multi-resolution, version of ICP invented by C. Raphael [22]. We employ it here in order to reduce (sometimes dramatically) the computational cost of maximum-likelihood decoding. Unfortunately, the ideas behind the method do not extend in any obvious way to the problem of computing posterior probabilities.

The topology of the dependency graph is irrelevant, but the idea of

CTFDP is perhaps best illustrated with a simple one-dimensional lattice. Let $X_0, X_1, \ldots, X_n$ have joint probability distribution

$$P(x) = \prod_{i=1}^{n} F_i(x_{i-1}, x_i),$$

and let $X_0, X_1, \ldots, X_n$ have common finite state space (range) $\mathcal{R}$. The idea is to coarsen $\mathcal{R}$ into a small number of "super states," and to perform multiple passes of dynamic programming on super states, successively refining super states on each pass. Formally, for each "coarsening" $q = 0, 1, \ldots, M$ we define a partition $\{\mathcal{R}_i^q\}_{i=1}^{m_q}$ of $\mathcal{R}$ with m_q elements in such a way that $\mathcal{R}$ is recovered at coarsening $q = 0$, and $\{\mathcal{R}_i^q\}$ refines $\{\mathcal{R}_i^{q+1}\}$:

1. ($\{\mathcal{R}_i^q\}$ partitions $\mathcal{R}$): $\mathcal{R}_i^q \bigcap \mathcal{R}_j^q = \emptyset \ \forall i \neq j$, and $\forall q \bigcup_{i=1}^{m_q} \mathcal{R}_i^q = \mathcal{R}$;

2. ($\{\mathcal{R}_i^q\}$ refines $\{\mathcal{R}_i^{q+1}\}$): for every $q = 0, 1, \ldots, M-1$ and every $i \in \{1, 2, \ldots, m_q\}$ there exists $j \in \{1, 2, \ldots, m_{q+1}\}$ such that $\mathcal{R}_i^q \subseteq \mathcal{R}_j^{q+1}$

3. ($\{\mathcal{R}_i^0\}$ recovers $\mathcal{R}$): $m_0 = |\mathcal{R}|$.

So $\mathcal{R}_i^0$, $i = 1, 2, \ldots, m_0$, are just the individual elements of $\mathcal{R}$.

Now suppose that for any i, and any $\mathcal{R}_j^q$ and $\mathcal{R}_{j'}^{q'}$, we could find a "heuristic cost" $H_i(\mathcal{R}_j^q, \mathcal{R}_{j'}^{q'})$ such that

1.
$$H_i(\mathcal{R}_j^q, \mathcal{R}_{j'}^{q'}) \geq \max_{x_{i-1} \in \mathcal{R}_j^q, \ x_i \in \mathcal{R}_{j'}^{q'}} F_i(x_{i-1}, x_i)$$

and

2.
$$H_i(\mathcal{R}_j^0, \mathcal{R}_{j'}^0) = F_i(\mathcal{R}_j^0, \mathcal{R}_{j'}^0)$$

(remember that $\mathcal{R}_j^0$ and $\mathcal{R}_{j'}^0$ are singletons).

Then we could perform dynamic programming on super states, starting with the partition $\{\mathcal{R}_i^M\}_{i=1}^{m_M}$, and using the costs $H_i(\mathcal{R}_j^M, \mathcal{R}_{j'}^M)$. Since presumably $m_M << m_0 = |\mathcal{R}|$, this first dynamic programming would be quick. We now refine: each (super) state along the optimal path (as chosen by the previous

dynamic programming pass) is refined into its subset of super states from the next lower level of coarsening:

$$\mathcal{R}_j^M \rightarrow \{\mathcal{R}_i^{M-1}\}_{\{i:\mathcal{R}_i^{M-1}\subseteq\mathcal{R}_j^M\}}$$

Now the state spaces are larger than at the first pass, but still, presumably, not nearly as large as $\mathcal{R}$. Another dynamic programming pass generates another sequence of super states, and the refinement/dynamic-programming cycle continues. Evidently, a path will eventually consist only of single states, $\mathcal{R}_i^0$, and evidently, in light of properties (1) and (2) characterizing the heuristic, this single-state path solves the original optimization problem.

Figure 9 describes how this process might proceed on a simple linear graph. Super states are delineated by boundary marks, and the super states along an optimal path are indicated by darkened circles. Here, $n = 5$, $|\mathcal{R}| = 8$, $M = 2$, and the refinements are all binary:

$$\mathcal{R}_1^2 = \{1,2,3,4\} \quad \mathcal{R}_2^2 = \{5,6,7,8\}$$

$$\mathcal{R}_1^1 = \{1,2\} \quad \mathcal{R}_2^1 = \{3,4\} \quad \mathcal{R}_3^1 = \{5,6\} \quad \mathcal{R}_4^1 = \{7,8\}$$

and $\mathcal{R}_i^0,\ i = 1,\ldots,8$, are the individual states $1,2,3,4,5,6,7$, and 8. The path chosen in panel (d) consists only of single states, so the process ends here, meaning that

$$argmax_x \prod_{i=1}^{n} F(x_{i-1}, x_i) = (4,7,7,4,6,5).$$

Coarse-to-fine dynamic programming may or may not find the optimal solution *efficiently*. There seems to be a rather subtle relationship between the structure of the problem at hand, and the savings won (or lost!) in a coarse-to-fine implementation. As it turns out, in the case of squaring constructions the relationship is often highly favorable, and it is worthwhile, therefore, to look at generalizations beyond the simple one-dimensional lattice.

It is clear enough how to proceed for more general graphs. Introduce a hierarchy of super states at every node, and define a heuristic cost for every clique function:

$$H_c(\mathcal{R}_c) \geq \max_{x_c \in \mathcal{R}_c} F_c(x_c)$$

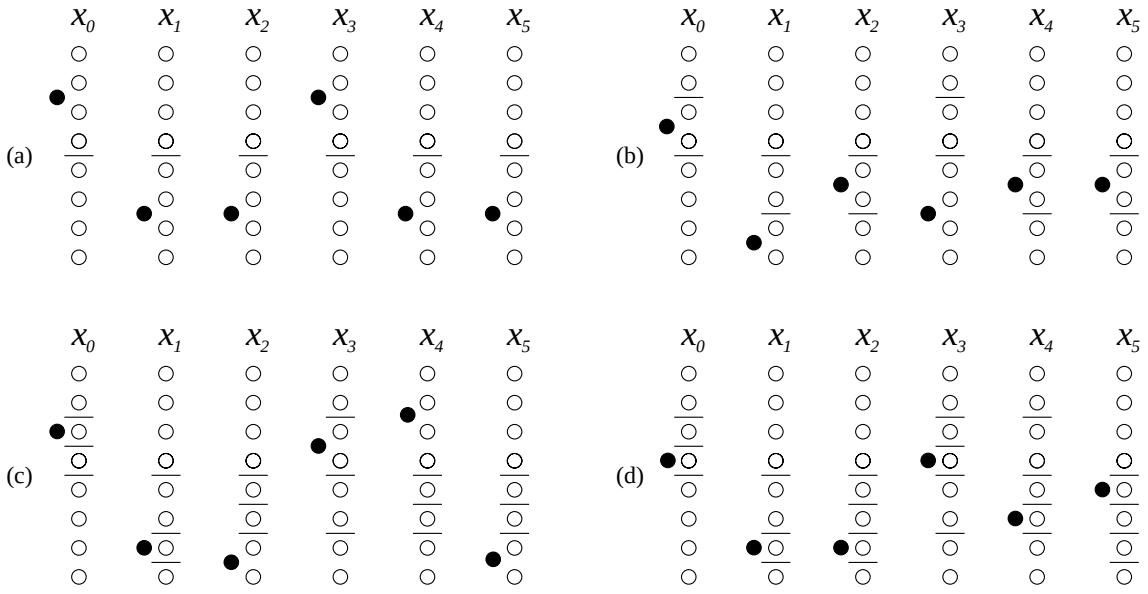

Figure 9: Coarse-to-fine dynamic programming. Darkened circles indicate a chosen path of super states. States along the chosen path are refined and dynamic programming is repeated. In panel d, dynamic programming yields a path of singletons, at which point the optimal path has been computed.

where $\mathcal{R}_c$ is a vector of super states, with one component super state for each site $s \in c$, and where $x_c \subseteq \mathcal{R}_c$ means component-wise membership. Our first choice for a heuristic cost would naturally be

$$H_c(\mathcal{R}_c) = \max_{x_c \in \mathcal{R}_c} F_c(x_c), \tag{11}$$

if this were actually computable at a less than prohibitive cost. As we shall see shortly (§5.3), this "ideal heuristic" has a simple analytic representation for the (canonical) squaring construction, and hence for codes of the type developed in §4 as well as the variants of these discussed in §5.1. This is perhaps a little surprising, and it has the fortunate consequence that CTFDP for these codes is particularly convenient and sometimes spectacularly efficient.

5.3 Exact Heuristics

Recall from §4 that with grammar-based (squaring) constructions, clique functions are (up to a multiplicative constant) just $0-1$ valued, indicating unallowed or allowed productions respectively. The "ideal heuristic" suggested in equation (11), applied to the clique $x_c = (x_{s_i^l}, x_{s_{2i-1}^{l-1}}, x_{s_{2i}^{l-1}})$, is then also binary, simply indicating the existence of an allowed production within the coarsened states:

$$H_c(\mathcal{R}_c) = H_c(A, B, C) = \begin{cases} 1 & \text{if } \exists \text{ production } x_{s_i^l} \to (x_{s_{2i-1}^{l-1}}, x_{s_{2i}^{l-1}}) \\ & \text{with } x_{s_i^l} \in A,\ x_{s_{2i-1}^{l-1}} \in B, \text{ and } x_{s_{2i}^{l-1}} \in C \\ 0 & \text{otherwise} \end{cases}$$

for any three super states A, B, and C at the three sites s_i^l, s_{2i-1}^{l-1}, and s_{2i}^{l-1}. It is useful to think of this as defining "super state productions": the production $A \to BC$ is allowed if there is a corresponding production among the constituent states.

With this choice of super state productions for a context-free code $\mathcal{C}$, any given set of state space partitions on the underlying graph—possibly consisting of super states of varying degrees of coarseness and possibly differing from node to node at any level—uniquely determines a *super code* containing $\mathcal{C}$. Clearly, any codeword in $\mathcal{C}$ can be derived from this super state grammar; given the codeword $\mathbf{c} \,\epsilon\, \mathcal{C}$, there exists a super state derivation tree that corresponds (by the definition of super state productions) to the codeword's original state derivation tree and has the bits of $\mathbf{c}$ as its terminal assignments. Thus, each such super state grammar generates a super code containing $\mathcal{C}$.

The CTFDP algorithm proceeds by progressively refining the super state grammar. Given the solution of the previous DP problem—an optimal derivation tree corresponding to a minimum cost super codeword, we determine whether the optimal derivation tree contains any non-singlet super states. If so, we refine these super states, recompute the super state productions, solve the new DP problem, and again examine the optimal derivation tree. If not, we stop: the current optimal derivation tree represents the minimum cost codeword. Since the final derivation tree contains only states from $\mathcal{C}$'s own grammar, it certainly generates a codeword (in $\mathcal{C}$). Moreover, this codeword is by definition the minimum cost super codeword in the final super code—a code that contains $\mathcal{C}$ itself.

		AVG(CTFDP)/DP Operations			
Code	DP Operations	$\sigma = 0.3$	$\sigma = 0.5$	$\sigma = 0.8$	$\sigma = 1.0$
RM(2,5)	3,007	0.8134	0.8136	1.0266	1.693
RM(2,6)	79,231	0.1199	0.1199	0.123	0.2127
RM(2,7)	4,606,719	0.006903	0.006904	0.007029	0.0099
RM(3,7)	4,425,388,799	1.136e-5	1.137e-5	1.214e-4	—
$RM^{(10)}(4,8)$	12,887,551	0.003262	0.003263	0.0052	0.0811
$RM^{(8)}(6,10)$	4,236,287	0.03261	0.03262	0.0663	0.3306

Table 2: The performance of CTFDP relative to DP. For six representative thinned Reed–Muller codes, the table compares the number of decoding operations required by exact DP with the average number of operations from a series of CTFDP decoding simulations. Codewords were transmitted across a memoryless AWGN channel (standard deviation σ) with BPSK modulation.

Although this CTFDP algorithm must eventually produce a solution to the given optimization problem, it need not necessarily outperform standard DP. For the procedure to converge rapidly, the number of refinements and subsequent DP computations must be minimal. This suggests that super states should consist of aggregations of "similar" states so that their costs more closely reflect those of their constituents. In addition, the determination of super state productions must not be too computationally demanding. Remarkably, at least in the case of Reed–Muller codes, one can in fact find a natural choice of super states that addresses these concerns, resulting in a substantially faster maximum-likelihood decoder.

Using the partitioning scheme introduced in appendix §C, we can implement a CTFDP version of maximum-likelihood decoding for the Reed–Muller codes. Table 2 presents the average ratio of CTFDP to DP operations from 50 trials with each of four RM codes and two thinned RM codes (see §5.1). Except for the very smallest code RM(2,5), the CTFDP algorithm computes the maximum likelihood codeword substantially faster on average with an efficiency increasing as the code size or signal-to-noise ratio increases. In the case of RM(3,7), the coarse-to-fine procedure is five orders of magnitude faster than the effectively impractical DP approach!

References

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, N.J., 1957.

[2] L.R. Bahl, F. Jelinek, and R.L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-5:179–190, 1983.

[3] C. Cannings, Thompson E.A., and H.H. Skolnick. Probability functions on complex pedigrees. *Advances in Applied Probability*, 10:26–61, 1978.

[4] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass., 1979.

[5] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Information Theory*, IT-13:260–269, 1967.

[6] N. Wiberg, H.-A. Loeliger, and R. Kötter. Codes and iterative decoding on general graphs. *European Trans. Telecom.*, 6:513–525, 1995.

[7] N. Wiberg. *Codes and decoding on general graphs*. PhD thesis, Department of Electrical Engineering, Linköping University, Sweden, 1996.

[8] G.D. Forney Jr. On iterative decoding and the two-way algorithm. In *Proc. Intl. Symp. Turbo Codes and Related Topics*, Brest, France, 1997.

[9] F. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. Technical report, Department of Electrical and Computer Engineering, University of Toronto, 1998.

[10] R. Dechter. Bucket elimination: a unifying framework for probabilistic inference. In M.I. Jordan, editor, *Learning in Graphical Models*, pages 75–104. The MIT Press, 1999.

[11] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, San Francisco, 1988.

[12] S.L. Lauritzen. *Graphical Models*. Oxford Univ. Press, Oxford, 1996.

[13] B.D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[14] R.G. Gallager. *Low-Density Parity-Check Codes*. PhD thesis.

[15] R.M. Tanner. A recursive approach to low complexity codes. *IEEE Trans. Inform. Theory*, IT-27:533–547, 1981.

[16] L.R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, IT-20:284–287, 1974.

[17] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: Turbo-codes. In *ICC'93*, pages 1064–1070, Geneva, Switzerland, 1993.

[18] D.J.C. Mackay. Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 45:399–431, 1999.

[19] F. Kschischang and B. Frey. Iterative decoding of compound codes by probability propagation in graphical models. *IEEE Journal on Selected Areas in Communications*, 16:219–230, 1998.

[20] L.N. Kanal and A.R.K. Sastry. Models for channels with memory and their applications to error control. *Proceedings of the IEEE*, 66:724–744, 1978.

[21] G.D. Forney Jr. Coset codes-part II: binary lattices and related codes. *IEEE Transactions on Information Theory*, 34:1152–1187, 1988.

[22] C.S. Raphael. Coarse-to-fine dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. To appear.

[23] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE*, 77:257–286, 1989.

[24] Y. Amit and A. Kong. Graphical templates for model registration. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18:225–236, 1996.

[25] B.J. Frey. *Graphical Models for Machine Learning and Digital Communication*. MIT Press, Cambridge, MA, 1998.

[26] J. Hammersley and P. Clifford. Markov fields on finite graphs and lattices. Technical report, University of California, Berkeley, 1968.

[27] G. Winkler. *Image Analysis, Random Fields, and Dynamic Monte Carlo Methods: A Mathematical Introduction*. Springer Verlag, 1995.

[28] G. Kallianpur. *Stochastic Filtering Theory*. Springer-Verlag, New York, 1980.

[29] H. Künsch, S. Geman, and A. Kehagias. Hidden Markov random fields. *Annals of Applied Probability*, 5:577–602, 1995.

[30] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[31] G.D. Forney Jr. The Viterbi algorithm. *Proceedings of the IEEE*, 61:268–278, 1973.

[32] P.W. Fieguth and A.S. Willsky. Fractal estimation using models on multiscale trees. *IEEE Trans. on Signal Processing*, 44:1297–1300, 1996.

[33] M. Meila. *Learning with Mixtures of Trees*. PhD thesis, Massachusetts Institute of Technology, 1999.

[34] W.C. Gore. Further results on product codes. *IEEE Transactions on Information Theory*, IT-16:446–451, 1970.

[35] R. Kindermann and J. Snell. *Markov Random Fields and Their Applications*. American Mathematical Society, Providence, RI, 1980.

[36] K. Kochanek. *Dynamic Programming Algorithms for Maximum Likelihood Decoding*. PhD thesis, Division of Applied Mathematics, Brown University, 1998.

[37] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.

[38] A. Kam and G. Kopec. Document image decoding by heuristic search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18:945–950, 1996.

[39] A. Kam and G. Kopec. The iterated complete path algorithm. Technical report, Xerox Palo Alto Research Center, 1995.

APPENDIX

A Reed–Muller Grammars

In this appendix, we present a brief derivation of the Reed–Muller grammar. We refer the reader to Kochanek [36] for a fuller account.

Originally in §4.1, our indexing system for grammatical symbols was designed to reflect a geometric hierarchy of minimum distances and separations. In the linear context both the indexing scheme and distance properties of symbols emerge naturally from the algebra of Reed–Muller codes. Each binary string in $\mathbf{Z}_2^{2^p}$ can be uniquely expressed as a sum of coset representatives,

$$B^p_{i_0,i_1,\ldots,i_p} = \sum_{k=0}^{p} \mathbf{c}^{(k,p)}_{i_k}$$

where $0 \leq i_k \leq m^p_k - 1$ and $0 \leq k \leq p$. We choose our coset representatives for the quotient group $RM(p-k,p)/RM(p-k-1,p)$ according to the scheme

$$\mathbf{c}^{(k,p)}_{i_k} = \mathbf{i}_k \overline{G}_{\partial RM}(p-k,p)$$

where $\mathbf{i}_k$ is the $\binom{p}{k}$-bit binary representation of the integer i_k and $\overline{G}_{\partial RM}(p-k,p)$ is the $\binom{p}{k} \times 2^p$ generator matrix of p-fold Kronnecker products of the form $\mathbf{g}_{j_1} \otimes \mathbf{g}_{j_2} \otimes \cdots \otimes \mathbf{g}_{j_p}$ of weight 2^k ordered lexicographically by label $j_1 j_2 \cdots j_p$—largest first. Note that the generators $\mathbf{g}_0 = (10)$ and $\mathbf{g}_1 = (11)$ serve as a basis for $\mathbf{Z}_2^2$ as do their corresponding p-fold Kronnecker products for $\mathbf{Z}_2^{2^p}$. For a complete discussion of set and group partitions of $\mathbf{Z}_2^{2^p}$ (upon which much of this discussion is based) see [21].

One can verify by induction [36] that these symbols have a familiar hierarchical structure. In fact, the iterative rule for constructing symbols is

$$B^p_{i_0,i_1,\ldots,i_p} \;=\; \mathbf{g}_1 \otimes B^{p-1}_{x_0,x_1,\ldots,x_{p-1}} + \mathbf{g}_0 \otimes B^{p-1}_{z_0,z_1,\ldots,z_{p-1}}$$

where $x_k = \lfloor i_{k+1}/m^{p-1}_{k+1} \rfloor$ and $z_k = i_k \bmod m^{p-1}_k$. But by an application of the additivity rule for this linear representation of $\mathbf{Z}_2^{2^p}$,

$$B^p_{i_0,i_1,\ldots,i_p} + B^p_{j_0,j_1,\ldots,j_p} = B^p_{g(i_0,j_0),g(i_1,j_1),\ldots,g(i_p,j_p)},$$

the grammatical hierarchy of §4.1 emerges:

$$B^p_{i_0,i_1,\ldots,i_p} \quad = \quad B^{p-1}_{g(x_0,z_0),g(x_1,z_1),\ldots,g(x_{p-1},z_{p-1})} \times B^{p-1}_{x_0,x_1,\ldots,x_{p-1}}.$$

Notice that the combinatorial function c has been replaced by the alternative $f(n,m,k) = g(n, m \bmod k)$ where $g(\cdot,\cdot)$ represents the bitwise exclusive OR operator.

Thus we see that the derivation of the Reed–Muller grammar is simply a matter of choosing the appropriate twisting. Instead of c we introduce a "canonical" twisting f which corresponds to Forney's "iterated group squaring construction" and essentially serves to linearize the grammatical hierarchy. To distinguish the two developments, we label the Reed–Muller symbols with the letter B (instead of A), and everywhere replace c by f. Then for the Reed–Muller grammar,

1. Equation (6) (with B in place of A and f in place of c) constructs (inductively) $\{B^p_{i_0,\ldots,i_p}\}$, a linear representation of $\mathbf{Z}_2^{2^p}$;

2. Equation (8) (with B in place of A) produces a collection of distance 2^α codes $\{B^p_{i_0,\ldots,i_{\alpha-1}}\}$, representing $RM(p-\alpha,p) = B^p_{\underbrace{0,0,\ldots,0}_{\alpha}}$ and its cosets in $\mathbf{Z}_2^{2^p}$; and

3. The productions in equation (10) (with B in place of A and f in place of c) define a grammatical realization of the codes $\{B^p_{i_0,\ldots,i_{\alpha-1}}\}$.

B Thinning

For the class of context-free codes constructed in §4.1 and generated by Reed–Muller-like grammars, one can bound the size of the state space without disrupting the coherence of the resulting context-free grammar. Suppose, for example, we impose an upper bound of 2^n on the number of allowed productions in an intractably large context-free grammar of this type. The yield of this reduced grammar is a subcode of the original one—a *thinned* code. We define the thinned Reed–Muller code $RM^{(n)}(p-\alpha,p)$ to be the

code generated by the start symbol $B^{p,n}_{\underbrace{0,0,\ldots,0}_{\alpha}}$ and productions:

$$B^{l,n}_{i_0,i_1,\ldots,i_{K(l)}} = \begin{cases} \bigcup_{x=0}^{I^l-1} B^{l-1,n}_{g(x_0,z_0),\ldots,g(x_{\alpha-2},z_{\alpha-2}),g(x,z_{\alpha-1})} \times B^{l-1,n}_{x_0,\ldots,x_{\alpha-2},x} & \alpha \le l \le p \\ B^{l-1,n}_{g(x_0,z_0),\ldots,g(x_{l-2},z_{l-2}),g(x_{l-1},z_{l-1})} \times B^{l-1,n}_{x_0,\ldots,x_{l-2},x_{l-1}} & 1 \le l \le \alpha-1 \end{cases}$$

with terminals $B^{0,n}_{i_0} = i_0$ and $K(l) = \min(l, \alpha - 1)$. Note that the number of productions is now $I^l = \min(2^n, m^{l-1}_{\alpha-1})$ (adopting the convention that $\binom{l}{k} \triangleq 0$ for $l < 0$ or $l > k$). A careful examination of its grammar (see [36]) reveals $RM^{(n)}(p-\alpha, p)$ to be a linear $[2^p, \sum_{l=1}^{p} 2^{p-l} \log_2 I^l, 2^\alpha]$ subcode of $RM(p-\alpha, p)$ with $N^l \le 2^n$ symbols (or states) at level l. Of course, if we set $n \ge \binom{p-1}{\alpha-1}$ the unthinned grammar for $RM(p-\alpha, p)$ is recovered.

The decoding complexity of thinned codes is controlled by the limiting parameter n. By definition, the number of productions is bounded above by 2^n. Surprisingly, this condition induces a similar restriction on the state space cardinality. Thus, the number of operations required to decode the thinned Reed–Muller code $RM^{(n)}(p-\alpha, p)$ is loosely upper bounded by 2^{p+2n} (2^p sites, $\le 2^n$ states per site, $\le 2^n$ productions per state), a large but fixed multiple of the code length.

We conclude our discussion of thinning by observing that one could as easily have thinned a general context-free code—simply by making the appropriate substitutions $f \to c$; such codes would exhibit all of the above features except linearity.

C State Space Partitions

To implement a CTFDP version of a given DP problem, one must first partition the problem's state spaces into hierarchies of super states. In this appendix, we present one particularly effective choice of super states for thinned Reed–Muller codes.

We begin by introducing a more compact notation for the thinned Reed–Muller grammar. The set of allowed symbols at level l for the code $RM^{(n)}(p-\alpha, p)$ can be re-expressed as $\{B^{l,n}_i | 0 \le i \le N^l - 1\}$, where the integer label i is derived from the indices $i_0, \ldots, i_{K(l)}$ of $B^l_{i_0,\ldots,i_{K(l)}}$ ($K(l) = \min(l, \alpha - 1)$) by concatenation of binary representations. Specifically, let $\mathbf{i}_k$ $0 \le k \le K(l)$

be the $\binom{l}{k}$-bit binary expansion of i_k, and let $\mathbf{i} = \mathbf{i}_0\mathbf{i}_1\cdots\mathbf{i}_{K(l)}$ be the concatenation of these expansions. Then i is the integer with binary expansion $\mathbf{i}$ ($i \leftrightarrow \mathbf{i} = \mathbf{i}_0\mathbf{i}_1\cdots\mathbf{i}_{K(l)}$). The label i denotes the corresponding *state* of an allowed symbol. The number of states at level l is N^l (see [36]) whereas the number of productions allowed at level l is $I^l = \min(2^n, m_{K(l)}^{l-1})$.

Among the advantages of this scheme is that single and multiple productions can be jointly expressed as

$$B_i^{l,n} = \bigcup_{j=0}^{I^l-1} B_{g(x(i)\wedge j, z(i))}^{l-1,n} \times B_{x(i)\wedge j}^{l-1,n}$$

where the auxiliary integers $x(i)$ and $z(i)$ are defined by the correspondences

$$x(i) \leftrightarrow \mathbf{x}(i) = \mathbf{x}_0\mathbf{x}_1\cdots\mathbf{x}_{K(l)}$$

and

$$z(i) \leftrightarrow \mathbf{z}(i) = \mathbf{z}_0\mathbf{z}_1\cdots\mathbf{z}_{K(l)}$$

and the binary operator $\wedge$ is the same as g itself—bitwise exclusive OR, introduced for notational convenience. In these expressions $\mathbf{z}_k$ is the $\binom{l-1}{k}$-bit binary expansion of $z_k = i_k \bmod m_k^{l-1}$ and $\mathbf{x}_k$ is the $\binom{l-1}{k}$-bit expansion of $x_k = \lfloor i_{k+1}/m_{k+1}^{l-1} \rfloor$ (or 0 if $k = K(l)$). Note that for single productions (i.e. $K(l) = l$), the strings $\mathbf{x}_{K(l)}$ and $\mathbf{z}_{K(l)}$ have length $\binom{l-1}{l} \triangleq 0$ and can therefore be ignored; however, for multiple productions ($K(l) < l$) they cannot be ignored.

A further distinction of this scheme is that the multitude of productions for the Reed–Muller code $RM(p-\alpha, p)$ can be readily computed from a comparatively small set of stored integers—the $2(p+1)$ parameters $\{N^l, I^l | 0 \le l \le p\}$ and the set of $2\sum_{l=1}^{p} N^l$ auxiliary x's and z's, one pair of integers for each state at each level $l \ge 1$. But of far greater consequence is the foundation we have established for constructing a simple system of state space partitions for thinned Reed–Muller grammars.

The basic partition at level l is constructed as a succession of binary refinements of the set of N^l states. There are $\log_2 N^l + 1$ coarsenings, $q \in \{0, 1, \ldots, \log_2 N^l\}$, each with $N^{l,q} \triangleq \lfloor N^l/2^q \rfloor$ *super states* denoted by the pair (q, i) at coarseness q. Specifically, we define the coarsened symbols

$$B_i^{l,n,q} \triangleq \{B_k^{l,n} | \lfloor k/2^q \rfloor = i\}$$

for $0 \leq i \leq N^{l,q} - 1$.

The extraordinary feature of our choice of super states is that they inherit the underlying structure of the Reed–Muller grammar. In fact, these coarsened symbols obey the recursive set relation (proved in [36]):

$$B_i^{l,n,q} = \bigcup_{j=0}^{I^{l,q}-1} B_{g(\overline{x}\wedge j,\overline{z})}^{l-1,n,\overline{q}} \times B_{\overline{x}\wedge j}^{l-1,n,\overline{q}} \tag{12}$$

where $\overline{x}$ and $\overline{z}$ are simple functions of the auxiliary integers $x(i)$ and $z(i)$ respectively, while $\overline{q}$ and $I^{l,q}$ depend only on the level l and coarseness q. Although this coarsening scheme is somewhat cumbersome to express mathematically, its computational implementation is straightforward and facilitates the remarkably fast CTFDP decoding algorithm presented in §5.3.

For example, since the hierarchy of coarsened symbols retains the underlying Reed–Muller structure, the run-time computation of super state productions required by the CTFDP procedure is trivial. If (q, i), (q_L, L), and (q_R, R) are super states at the respective sites s_i^l, s_{2i-1}^{l-1}, and s_{2i}^{l-1}, then $(q, i) \rightarrow [(q_L, L), (q_R, R)]$ is an allowed super state production if and only if L shares a (suitably sized—see [36]) binary prefix with $g(\overline{x} \wedge j, \overline{z})$ and R shares a binary prefix with $\overline{x} \wedge j$ for some $0 \leq j \leq I^{(l,q)} - 1$; for if this condition is met, there is an allowed state production contained within the postulated super state production. We are thus able to compute super state productions by inspection!