Codes on Graphs: Normal Realizations

G. David Forney, Jr., *Fellow, IEEE*

***Abstract*—A generalized state realization of the Wiberg type is called normal if symbol variables have degree 1 and state variables have degree 2. A natural graphical model of such a realization has leaf edges representing symbols, ordinary edges representing states, and vertices representing local constraints. Such a graph can be decoded by any version of the sum-product algorithm. Any state realization of a code can be put into normal form without essential change in the corresponding graph or in its decoding complexity.**

Group or linear codes are generated by group or linear state realizations. On a cycle-free graph, there exists a well-defined minimal canonical realization, and the sum-product algorithm is exact. However, the Cut-Set Bound shows that graphs with cycles may have a superior performance–complexity tradeoff, although the sum-product algorithm is then inexact and iterative, and minimal realizations are not well-defined. Efficient cyclic and cycle-free realizations of Reed–Muller (RM) codes are given as examples.

The dual of a normal group realization, appropriately defined, generates the dual group code. The dual realization has the same graph topology as the primal realization, replaces symbol and state variables by their character groups, and replaces primal local constraints by their duals. This fundamental result has many applications, including to dual state spaces, dual minimal trellises, duals to Tanner graphs, dual input/output (I/O) systems, and dual kernel and image representations. Finally, a group code may be decoded using the dual graph, with appropriate Fourier transforms of the inputs and outputs; this can simplify decoding of high-rate codes.

***Index Terms*—Codes, dual codes, graphical models, group codes, state realizations, sum-product algorithm.**

I. Introduction

THE subject of codes defined on graphs is one of intense current interest.

Briefly, this subject developed as follows. The low-density parity-check (LDPC) codes of Gallager [19] were the first such codes. Gallager also invented what today is called the sum-product algorithm for *a posteriori* probability (APP) decoding.

Some years later, Tanner [42] founded the general study of codes defined on graphs, introduced generalized constraints, and proved the optimality of the sum-product and min-sum algorithms for decoding codes defined on cycle-free graphs.

Modern interest in this subject reawakened with the work of Wiberg, Loeliger, and Koetter [46], [47], who rediscovered Tanner's results and introduced states into Tanner graphs, thus allowing connections to trellises and to turbo codes, and with the rediscovery of LDPC codes by Spielman *et al.* [41] and MacKay *et al.* [31]. Connections to Bayesian networks and Markov random processes were then recognized [32], [26]. Recently, the subject has been abstracted and refined to a high degree under the rubrics of "factor graphs" [27] or "the generalized distributive law" [1], and further connections have been made, e.g., to fast Fourier transforms and to Kalman filtering.

Unlike some of this prior work, this paper makes a clear distinction between a realization of a code or system, namely, a mathematical characterization of the code/system as in system theory, and a graphical model of a realization.

We begin with realizations of the Wiberg type, which we call generalized state realizations. Generalized state realizations are constructed from three kinds of elements: symbols, states, and constraints. Each constraint is local, in the sense that it involves only a certain subset of the symbol and state variables. The full behavior of a generalized state realization is the set of all symbol/state configurations that satisfy all of the local constraints. The code generated by the realization is the set of all symbol configurations (codewords) that appear as part of some symbol/state configuration in the full behavior.

A generalized state realization has a natural graphical model, as shown in Wiberg *et al.* [46], [47] (now called a factor graph [27]). Symbols, states, and constraints are represented by three different kinds of vertices. A state or symbol vertex is connected to a constraint vertex if the corresponding state or symbol variable is involved in the corresponding constraint. Consequently, the graph is bipartite.

We restrict attention to *normal realizations*, defined as generalized state realizations in which the degree of a state variable (the number of constraints in which it is involved) is restricted to two, and the degree of a symbol variable is restricted to one. We note that these restrictions are already satisfied by conventional state realizations (trellises) and tail-biting realizations. We show how to "normalize" any generalized state realization.

Normal realizations have a natural graphical model, which we call a *normal graph*, in which constraints are represented by vertices, state variables are represented by ordinary edges, and symbol variables are represented by leaf edges ("half-edges"). We show that the normal graph of a normalized generalized state realization has the same topology as the factor graph of the original realization.

Like a factor graph, a normal graph has a natural associated "decoding algorithm," namely, the sum-product algorithm. If the graph is finite and cycle-free, then the algorithm is finite and exact. If the graph has cycles, then the algorithm becomes iterative and approximate, but empirically often performs well—e.g., for turbo codes or LDPC codes.

We show that normalization of a generalized state realization does not affect its decoding complexity. Moreover, in de-

Manuscript received December 10, 1998; revised August 15, 2000. The material in this paper was presented in part at the IEEE International Symposium on Information Theory, Sorrento, Italy, June 2000.

The author is with the Laboratory for Information and Decision Systems, MIT, Cambridge, MA 02139 USA (e-mail: forney@lids.mit.edu).

Communicated by F. R. Kschischang, Associate Editor for Coding Theory.
Publisher Item Identifier S 0018-9448(01)00726-X.

coding normal graphs, there is a nice separation of the functions of each graph component: symbol edges are purely for input/output (I/O), state edges are purely for communications (message passing), and constraint nodes are purely for computation.

Linear codes are generated by linear realizations, and group codes are generated by group realizations. We will see that a linear or group code defined on a cycle-free graph has a canonical minimal realization, as in the conventional trellis case. However, the question of how to minimize realizations on graphs with cycles remains largely open.

As examples, we present efficient realizations of Reed–Muller (RM) codes on graphs with cycles. Moreover, we show how these realizations may be made cycle-free by clustering, which yields realizations as in [13] that are slightly more efficient than minimal trellis realizations.

We give a fundamental duality theorem for normal realizations of linear or group codes/systems. We define the dual of a normal linear or group normal realization as follows: the dual realization has the dual (character group) state and symbol alphabets, which are the same as the primal alphabets in the linear or finite-group case, and dual (orthogonal) constraints. The graph of the dual realization therefore has the same topology as that of the primal realization. We then show that the dual realization generates the dual (orthogonal) linear or group code, regardless of whether the graph of the realization is cycle-free.

This fundamental duality theorem depends essentially on precisely the degree restrictions that we have imposed on normal realizations, and thus strongly supports restricting attention to such realizations. It also supports our growing belief that the fundamental results of linear system theory are most transparent in a group-theoretic setting, and do not depend on time invariance.

This theorem has myriad applications, of which we mention a few, including dual state spaces, dual minimal trellises (tail-biting too), duals to Tanner graphs, dual I/O systems, and dual kernel and image representations.

Finally, we show how a primal code may be decoded using the dual graph, after appropriate Fourier transform operations on the inputs and outputs.

II. Codes, Realizations, and Graphical Models

In this section we will establish notation and terminology for codes, and review Tanner-type and Wiberg-type realizations and their corresponding graphical models.

A. Codes

We denote variables by upper case letters, values of variables by corresponding lower case letters, and the alphabets of variables by corresponding upper case script letters.

For example, a symbol variable A_k takes values $a_k \in \mathcal{A}_k$ in a symbol alphabet $\mathcal{A}_k$. In coding, $\mathcal{A}_k$ is often a vector space over a finite field (e.g., the set $(\mathbb{F}_2)^n$ of all binary n-tuples) or a finite Abelian group.

A *symbol configuration space* $\mathcal{A}$ is a Cartesian product

$$\mathcal{A} = \prod_{k \in I_{\mathcal{A}}} \mathcal{A}_k$$

of a collection $\{\mathcal{A}_k,\, k \in I_{\mathcal{A}}\}$ of symbol alphabets, where $I_{\mathcal{A}}$ is any discrete index set. The elements of $\mathcal{A}$ are denoted by $\boldsymbol{a} = \{a_k,\, k \in I_{\mathcal{A}}\} \in \mathcal{A}$, and will be called *symbol configurations.* If all alphabets $\mathcal{A}_k$ are vector spaces over a field $\mathbb{F}$, then $\mathcal{A}$ is a direct-product vector space. If all alphabets $\mathcal{A}_k$ are groups, then $\mathcal{A}$ is a direct-product group.

A *code* $\mathcal{C}$ is any subset $\mathcal{C} \subseteq \mathcal{A}$ of a symbol configuration space. The elements of $\mathcal{C}$ will be called *valid* symbol configurations, or *codewords.*

Whereas in system theory the index set $I_{\mathcal{A}}$ is usually ordered and regarded as a time axis, here $I_{\mathcal{A}}$ will not necessarily be ordered. In coding, two codes that differ only by a permutation of the index set $I_{\mathcal{A}}$ are usually regarded as equivalent; they have the same minimum distance and the same performance on a memoryless channel. Consequently, we may represent the "time axis" $I_{\mathcal{A}}$ by an unordered discrete set (or by a graph) rather than by a subinterval of the integers.

For simplicity, we will usually assume that the index set $I_{\mathcal{A}}$ is finite—i.e., that $\mathcal{C}$ is a *block code.* (See the note at the end of Section VII-B.)

A *linear code* $\mathcal{C} \subseteq \mathcal{A}$ is a vector subspace of a direct-product symbol configuration space $\mathcal{A} = \prod_{k \in I_{\mathcal{A}}} \mathcal{A}_k$, where each symbol alphabet $\mathcal{A}_k$ is a vector space over a given field $\mathbb{F}$.

A *group code* $\mathcal{C} \subseteq \mathcal{A}$ is a subgroup of a direct-product symbol configuration space $\mathcal{A} = \prod_{k \in I_{\mathcal{A}}} \mathcal{A}_k$, where each symbol alphabet $\mathcal{A}_k$ is a group.

A linear code is evidently a group code. As our principal results are essentially group-theoretic, we will generally state and prove them for the group case. However, recognizing that most readers will be more familiar with linear codes, we will consistently translate our results into linear terms, and most of our examples will be linear.

We will use the $(8, 4, 4)$ extended Hamming code as a running example, to exhibit various styles of realizations. This code is a binary linear block code consisting of a subset of 16 elements of the space $\mathcal{A} = (\mathbb{F}_2)^8$ of binary 8-tuples. In one coordinate ordering, the codewords are the set of binary linear combinations of the four generators

$$\{11110000,\, 00111100,\, 00001111,\, 01011010\}.$$

B. Behavioral Realizations and Tanner Graphs

A code $\mathcal{C} \subseteq \mathcal{A}$ may be characterized as the set of configurations $\boldsymbol{a} \in \mathcal{A}$ that satisfy a certain set of constraints. For example, a linear code $\mathcal{C}$ may be characterized as the set of $\boldsymbol{a} \in \mathcal{A}$ that satisfy a certain set of parity checks.

Gallager's LDPC codes are based on such a representation [19], [20]. Tanner formalized and generalized this notion [42]. Nowadays such a representation might be called "behavioral," since it is specified by local constraints as in the behavioral system theory of Willems [48], [49].

Formally, a *behavioral realization* of a code $\mathcal{C} \subseteq \mathcal{A}$ is described by a set $\mathcal{C}_i$, $i \in I_{\mathcal{C}}$, of *local constraints* ("local codes," "local behaviors"), where $I_{\mathcal{C}}$ is a second discrete index set that need have no relation to the symbol index set $I_{\mathcal{A}}$. Each local

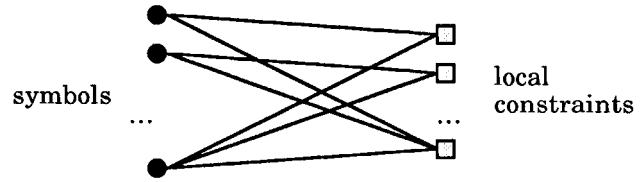

Fig. 1. Tanner graph.

constraint $\mathcal{C}_i$ *involves* some subset of the symbol variables, indexed by a certain subset $I_{\mathcal{A}}(i)$ of the symbol index set $I_{\mathcal{A}}$, and defines a subset

$$\mathcal{C}_i \subseteq \mathcal{A}_i = \prod_{k \in I_{\mathcal{A}}(i)} \mathcal{A}_k$$

of the corresponding local Cartesian product set $\mathcal{A}_i$. The local constraint thus defines a set of *valid* local configurations ("local codewords") $\boldsymbol{a}_{|I_{\mathcal{A}}(i)} = \{a_k, k \in I_{\mathcal{A}}(i)\} \in \mathcal{C}_i$, where the notation $\boldsymbol{a}_{|I_{\mathcal{A}}(i)}$ denotes the projection of a configuration $\boldsymbol{a}$ onto the symbol variables A_k indexed by $I_{\mathcal{A}}(i)$. The code $\mathcal{C}$ is then the set of all configurations that satisfy all local constraints

$$\mathcal{C} = \{\boldsymbol{a} \in \mathcal{A} | \boldsymbol{a}_{|I_{\mathcal{A}}(i)} \in \mathcal{C}_i \text{ for all } i \in I_{\mathcal{C}}\}.$$

In a *linear behavioral realization*, each symbol alphabet $\mathcal{A}_k$ is a vector space over a field $\mathbb{F}$, and each local code $\mathcal{C}_i$ is a *subspace* of the direct-product vector space $\mathcal{A}_i$. The code $\mathcal{C}$ is then a linear code—i.e., a subspace of $\mathcal{A}$.

In a *group behavioral realization*, each symbol alphabet $\mathcal{A}_k$ is a group, and each local code $\mathcal{C}_i$ is a *subgroup* of the direct-product group $\mathcal{A}_i$; the code $\mathcal{C}$ is then a group code—i.e., a subgroup of $\mathcal{A}$.

For example, a linear code $\mathcal{C} \subseteq \mathcal{A}$ may be characterized as the set of $\boldsymbol{a} \in \mathcal{A}$ that satisfy the parity-check equations $\langle \boldsymbol{a}, \boldsymbol{h}_i \rangle = 0$ for a certain set of check configurations $\{\boldsymbol{h}_i \in \mathcal{A}, i \in I_{\mathcal{C}}\}$. The symbol variables A_k that are involved in the ith check are those for which $h_{ik} \neq 0$. Each local code $\mathcal{C}_i$ is then a linear $(n_i, n_i - 1, 2)$ single-parity-check (SPC) code, whose length n_i, called the *degree* of $\mathcal{C}_i$, is equal to the number of symbols involved in $\mathcal{C}_i$.

A behavioral realization has a natural graphical model, which in coding is called a *Tanner graph.* A Tanner graph is a bipartite graph in which a first set of vertices represents the symbol variables $\{A_k, k \in I_{\mathcal{A}}\}$, a second set of vertices represents the local constraints $\{\mathcal{C}_i, i \in I_{\mathcal{C}}\}$, and a symbol vertex is connected to a constraint vertex by an edge if the corresponding symbol variable is involved in the corresponding local constraint. Fig. 1 illustrates a generic Tanner graph.

For example, the binary linear $(8, 4, 4)$ code is self-dual, and therefore may be characterized as the set of binary 8-tuples that satisfy the four parity-check equations

$$\begin{aligned} a_1 + a_2 + a_3 + a_4 &= 0 \\ a_3 + a_4 + a_5 + a_6 &= 0 \\ a_5 + a_6 + a_7 + a_8 &= 0 \\ a_2 + a_4 + a_5 + a_7 &= 0. \end{aligned}$$

The Tanner graph of this behavioral realization is shown in Fig. 2. The four parity-check constraints are represented by square vertices labeled by "+" signs. Clearly, there is no simpler Tanner graph for this code.

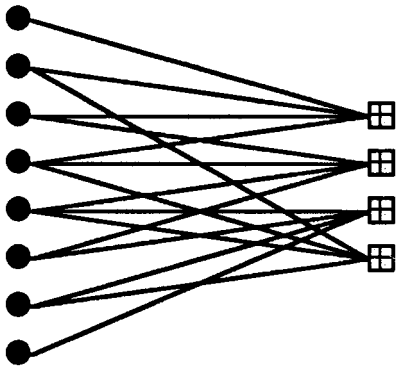

Fig. 2. Tanner graph for (8, 4, 4) code.

C. Generalized State Realizations and Factor Graphs

Wiberg *et al.* [46], [47] rediscovered Tanner's work and extended it to include state variables, a step whose importance can hardly be overstated. Connections were thereby made to the theory of trellis realizations, which correspond to conventional system-theoretic state realizations of codes. Moreover, new kinds of generalized state realizations were thereby defined, differing from those traditionally considered in system theory.

Formally, a *generalized state realization* of a code $\mathcal{C}$ is defined by three sets of elements:

- a set of symbol variables $\{A_k, k \in I_{\mathcal{A}}\}$ with alphabets $\mathcal{A}_k$;
- a set of state variables $\{S_j, j \in I_{\mathcal{S}}\}$ with alphabets (state spaces) $\mathcal{S}_j$;
- a set of local constraints $\{\mathcal{C}_i, i \in I_{\mathcal{C}}\}$.

The three index sets $I_{\mathcal{A}}$, $I_{\mathcal{S}}$, and $I_{\mathcal{C}}$ are discrete, unordered, and in general unrelated to each other. The *symbol configuration space* is the Cartesian product $\mathcal{A} = \prod_{k \in I_{\mathcal{A}}} \mathcal{A}_k$, and the *state configuration space* is the Cartesian product $\mathcal{S} = \prod_{j \in I_{\mathcal{S}}} \mathcal{S}_j$. Each local constraint $\mathcal{C}_i$ *involves* a subset of the symbol and state variables indexed by a certain subset $I_{\mathcal{A}}(i)$ of the symbol index set $I_{\mathcal{A}}$ and a certain subset $I_{\mathcal{S}}(i)$ of the state index set $I_{\mathcal{S}}$, respectively, and defines a subset

$$\mathcal{C}_i \subseteq \mathcal{A}_i \times \mathcal{S}_i = \left(\prod_{k \in I_{\mathcal{A}}(i)} A_k \right) \times \left(\prod_{j \in I_{\mathcal{S}}(i)} S_j \right)$$

of the corresponding local Cartesian product configuration space $\mathcal{A}_i \times \mathcal{S}_i$. The local constraint thus defines a set of *valid* local configurations ("local codewords")

$$(\boldsymbol{a}_{|I_{\mathcal{A}}(i)}, \boldsymbol{s}_{|I_{\mathcal{S}}(i)}) = \{\{a_k, k \in I_{\mathcal{A}}(i)\}, \{s_j, j \in I_{\mathcal{S}}(i)\}\} \in \mathcal{C}_i.$$

The *full behavior* $\mathfrak{B} \subseteq \mathcal{A} \times \mathcal{S}$ is the set of all global symbol/state configurations that satisfy all local constraints

$$\mathfrak{B} = \{(\boldsymbol{a}, \boldsymbol{s}) \in \mathcal{A} \times \mathcal{S} | (\boldsymbol{a}_{|I_{\mathcal{A}}(i)}, \boldsymbol{s}_{|I_{\mathcal{S}}(i)}) \in \mathcal{C}_i, i \in I_{\mathcal{C}}\}.$$

Finally, the *code* generated by the generalized state realization is the projection $\mathcal{C} = \mathfrak{B}_{|\mathcal{A}}$ of the full behavior $\mathfrak{B} \subseteq \mathcal{A} \times \mathcal{S}$ onto the symbol configuration space $\mathcal{A}$. In other words, the code

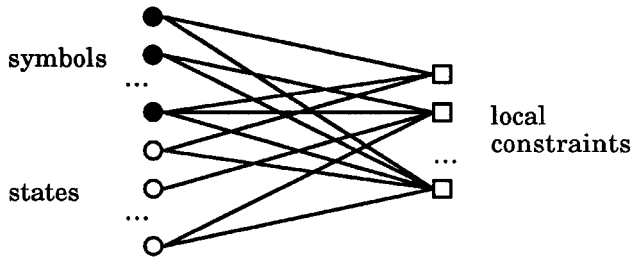

Fig. 3. Factor graph of a generalized state realization.

$\mathcal{C}$ is the set all $\boldsymbol{a} \in \mathcal{A}$ that occur as part of some valid configuration $(\boldsymbol{a}, \boldsymbol{s}) \in \mathfrak{B}$.

In still other words, if the full behavior $\mathfrak{B}$ is regarded as a symbol/state code, then $\mathcal{C}$ is the code obtained from $\mathfrak{B}$ by "puncturing" all state variables.

In a *linear generalized state realization*, each symbol alphabet $\mathcal{A}_k$ and state alphabet $\mathcal{S}_j$ is a vector space over a field $\mathbb{F}$, and each local code $\mathcal{C}_i$ is a *subspace* of the direct-product vector space $\mathcal{A}_i \times \mathcal{S}_i$. The full behavior $\mathfrak{B}$ and the code $\mathcal{C}$ generated by a linear realization are linear codes.

In a *group generalized state realization*, each symbol alphabet $\mathcal{A}_k$ and state alphabet $\mathcal{S}_j$ is a group, and each local code $\mathcal{C}_i$ is a *subgroup* of the direct-product group $\mathcal{A}_i \times \mathcal{S}_i$. The full behavior $\mathfrak{B}$ and the code $\mathcal{C}$ generated by a group realization are group codes.

The *degree* of a local constraint $\mathcal{C}_i$ is defined as the number $|I_{\mathcal{A}}(i)| + |I_{\mathcal{S}}(i)|$ of symbol and state variables that are involved in $\mathcal{C}_i$. Conversely, the *degree* of a symbol or state variable is defined as the number of local constraints in which it is involved.

A generalized state realization has a natural graphical model, now called a *factor graph* [27]. A factor graph is a bipartite graph in which three types of vertices represent, respectively, symbol variables A_k, state variables S_j, and local constraints $\mathcal{C}_i$. A symbol or state vertex is connected to a constraint vertex by an edge if the corresponding symbol or state variable is involved in the corresponding local constraint. Thus, the degree of a vertex is the degree of the corresponding variable or constraint as defined above. Edges are in principle unlabeled and undirected, although an edge may sometimes be labeled by the state or symbol variable to which it is connected [27]. Fig. 3 illustrates a generic factor graph of a generalized state realization.

Another possible graphical model for a generalized state realization is a hypergraph—i.e., a graph in which edges may have arbitrary degrees. Variables may be associated with hypergraph edges and local constraints with vertices, or *vice versa*. A hypergraph edge of degree greater than two may be drawn as a star (as in block diagrams) or as a clique (as in Markov random fields). However, a hypergraph edge of degree one needs to be represented by a special symbol (e.g., the "dongle" symbol introduced later in Fig. 10).

D. Symbol Versus State Variables

It may at first appear that there is little difference between a behavioral realization of a code $\mathcal{C}$ as described earlier and a generalized state realization as just described, except that the variables have been partitioned into two types. What is the difference between symbol and state variables?

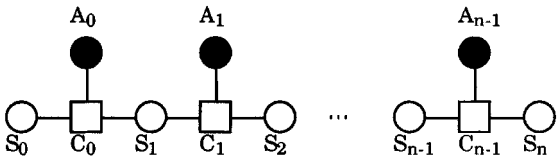

Fig. 4. Factor graph of a conventional state realization.

The basic difference is that state variables do not appear in codewords. A generalized state realization of a code $\mathcal{C}$ is obtained by first realizing a full behavior $\mathfrak{B}$, and then puncturing the state variables.

Symbol variables are given *a priori* as part of the description of $\mathcal{C}$. State variables are auxiliary variables that are introduced by the designer to improve the realization in some sense.

Symbol variables are visible (manifest, observable), whereas state variables are hidden (latent, unobservable). In coding, symbol variables are transmitted over a channel and are observed at the receiver in the presence of noise, whereas state variables are part of the internal realization of the code and are neither transmitted nor observed.

E. Conventional State Realizations

A conventional state realization is a good first example of a generalized state realization.

In a *conventional state realization*, the symbol index set $I_{\mathcal{A}}$ is regarded as an ordered time axis and is identified with a subinterval of the integers; e.g., $I_{\mathcal{A}} = [0, n)$. In coding, the ordering of $I_{\mathcal{A}}$ is a design choice.

The state and constraint index sets are taken to be essentially the same time axis, e.g., $I_{\mathcal{S}} = [0, n]$ and $I_{\mathcal{C}} = [0, n)$. The initial and final state variables are unary (single-valued): $|\mathcal{S}_0| = |\mathcal{S}_n| = 1$. The local constraints $\mathcal{C}_k$ specify the configurations $(s_k, a_k, s_{k+1}) \in \mathcal{S}_k \times \mathcal{A}_k \times \mathcal{S}_{k+1}$ that can actually occur; i.e., which state transitions $(s_k, s_{k+1}) \in \mathcal{S}_k \times \mathcal{S}_{k+1}$ can occur, and which symbols $a_k \in \mathcal{A}_k$ may be associated with each possible transition.

Thus, each symbol variable A_k is involved in one local constraint $\mathcal{C}_k$, and each state variable (except S_0 and S_n) is involved in two local constraints, namely, $\mathcal{C}_k$ and $\mathcal{C}_{k-1}$.

Fig. 4 shows a factor graph of a conventional state realization. The graph consists of a chain of "trellis sections" corresponding to the constraints $\mathcal{C}_k$, and is evidently cycle-free.

A trellis diagram is a more detailed graphical model of a conventional state realization which can be drawn when all state spaces $\mathcal{S}_k$ are finite, as is usually the case in coding. A trellis diagram is a directed graph with a set of $|\mathcal{S}_k|$ vertices at each time k corresponding to the states in $\mathcal{S}_k$. A local constraint $\mathcal{C}_k$ is represented by a trellis section, consisting of a set of edges ("branches") from s_k to s_{k+1} labeled by a_k for each valid local configuration $(s_k, a_k, s_{k+1}) \in \mathcal{C}_k$. Each path in the trellis from the unique root vertex $s_0 \in \mathcal{S}_0$ to the unique goal vertex $s_n \in \mathcal{S}_n$ then corresponds to a valid configuration $(\boldsymbol{a}, \boldsymbol{s})$ in the full behavior $\mathfrak{B}$. The code generated by the trellis diagram is the set of all path label sequences, namely, the projection $\mathcal{C} = \mathfrak{B}_{|\mathcal{A}}$.

For example, for the $(8, 4, 4)$ code, it is known that the minimal state complexity profile of any trellis diagram is obtained with the coordinate ordering given earlier, and is equal

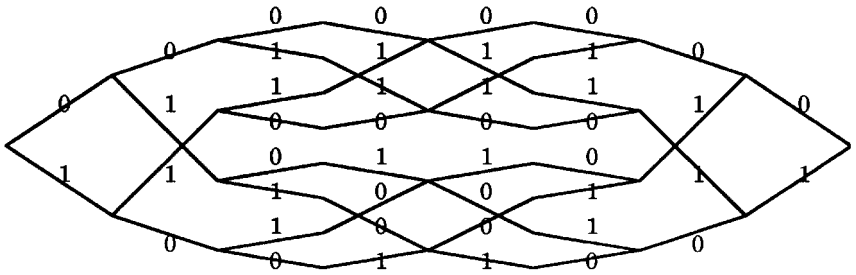

Fig. 5. Trellis diagram for $(8, 4, 4)$ code.

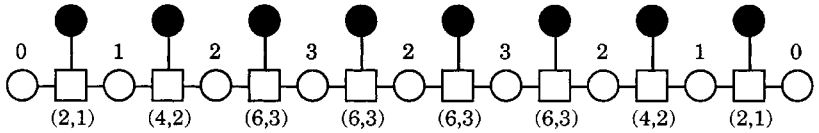

Fig. 6. Factor graph for $(8, 4, 4)$ code.

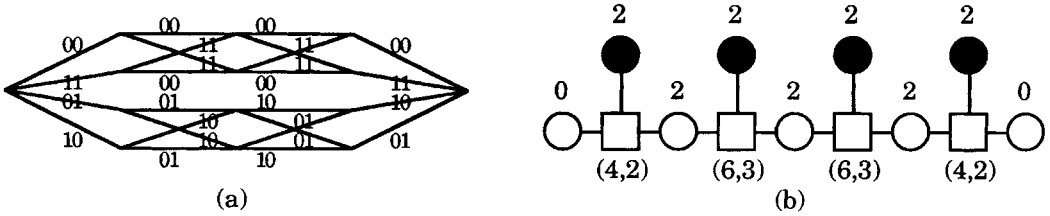

Fig. 7. Four-section realizations for $(8, 4, 4)$ code. (a) Trellis diagram. (b) Factor graph.

to $(|\mathcal{S}_0|, \cdots, |\mathcal{S}_8|) = (1, 2, 4, 8, 4, 8, 4, 2, 1)$. This trellis diagram is shown in Fig. 5.

The corresponding factor graph is shown in Fig. 6. Each state variable is labeled by the dimension of its alphabet. (All symbol variables have dimension 1.) Each local constraint (trellis section) is a binary linear block code, labeled by its length and dimension (n, k).

It is well known that a nicer trellis diagram is obtained by taking symbols two at a time to obtain a four-section trellis diagram in which each branch is labeled by a pair of symbols, as shown in Fig. 7(a). This reduces the state space dimension profile to $(1, 4, 4, 4, 1)$. The corresponding sectionalized factor graph is shown in Fig. 7(b). Because the local constraint codes (trellis sections) do not increase in size, this sectionalized realization is no more difficult to decode, which is not the case in general.

F. Tail-Biting State Realizations

A tail-biting state realization is a good first example of an unconventional generalized state realization (cf. [8]).

In a *tail-biting state realization*, the symbol index set $I_{\mathcal{A}}$ is regarded as a circular time axis and is identified with $\mathbb{Z}_n$, the integers modulo $n = |I_{\mathcal{A}}|$. All index arithmetic is performed in $\mathbb{Z}_n$; i.e., modulo n. Again, the correspondence between $I_{\mathcal{A}}$ and $\mathbb{Z}_n$ is a design choice.

The state and constraint index sets are taken as the same circular time axis; i.e., $I_{\mathcal{S}} = I_{\mathcal{C}} = \mathbb{Z}_n$. Again, the local constraints $\mathcal{C}_k$ specify the valid configurations

$$(s_k, a_k, s_{k+1}) \in \mathcal{S}_k \times \mathcal{A}_k \times \mathcal{S}_{k+1}.$$

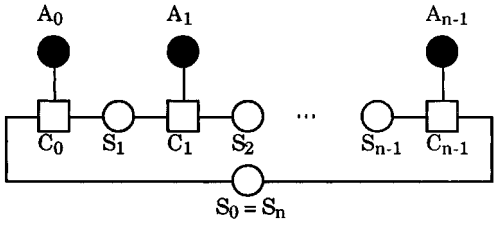

Fig. 8. Factor graph of a tail-biting state realization.

In particular, $\mathcal{C}_{n-1}$ specifies the valid configurations

$$(s_{n-1}, a_{n-1}, s_0) \in \mathcal{S}_{n-1} \times \mathcal{A}_{n-1} \times \mathcal{S}_0.$$

Thus, each symbol variable A_k is involved in one local constraint $\mathcal{C}_k$, and each state variable is involved in precisely two local constraints, namely, $\mathcal{C}_k$ and $\mathcal{C}_{k-1}$. A symbol configuration $\boldsymbol{a}$ is a valid codeword if and only if there exists a symbol/state configuration $(\boldsymbol{a}, \boldsymbol{s})$ that satisfies all local constraints.

Fig. 8 shows a factor graph of a tail-biting state realization. The graph consists of a circular chain of n "trellis sections" corresponding to the constraints $\{\mathcal{C}_k, k \in \mathbb{Z}_n\}$, and has a single cycle.

For example, Fig. 9 shows a minimal four-section tail-biting realization for the $(8, 4, 4)$ code [8], both as a trellis diagram and as a factor graph. Note that a different coordinate ordering is used. The state complexity profile is $(2, 4, 2, 4)$, which in this case, hardly improves over that of a conventional state realization.

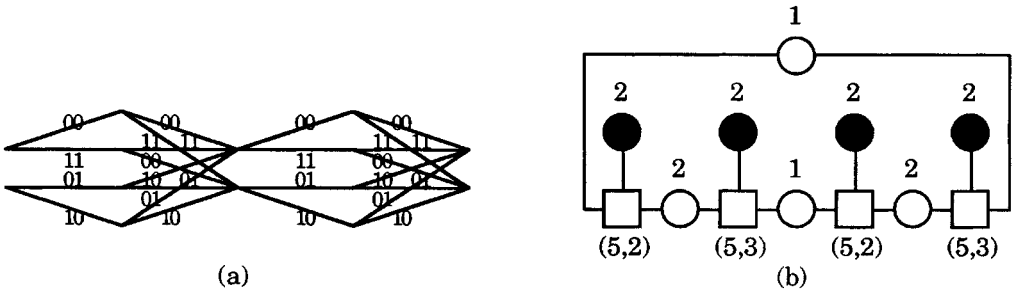

Fig. 9. Four-section tail-biting realizations for $(8, 4, 4)$ code. (a) Trellis. (b) Factor graph.

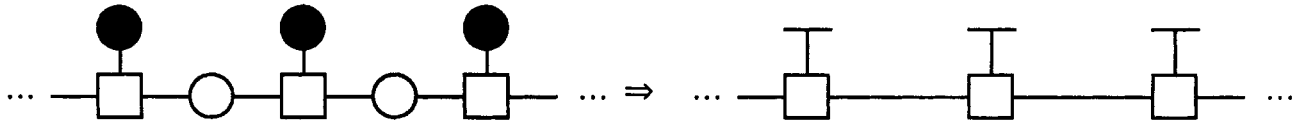

Fig. 10. Factor graph and normal graph of a conventional state realization.

III. Normal Realizations and Normal Graphs

We define a *normal realization* simply as a generalized state realization in which all symbol variables have degree 1 and all state variables have degree 2. In other words, each symbol variable is involved in precisely one local constraint, and each state variable is involved in precisely two local constraints. No restriction is imposed on constraint degrees.

Because of these degree restrictions, a normal realization has a natural graphical model that is different from a factor graph, which we call a *normal graph*. In a normal graph, variables are represented by edges and constraints by vertices.

We note immediately that the variables in a conventional state realization satisfy these degree restrictions, except for the initial and final state variables S_0 and S_n. But these state variables are unary and therefore may be neglected, since a function of a unary variable is actually independent of that variable. Excluding S_0 and S_n, therefore, we have the following.

Theorem 3.1: A conventional state realization is a normal realization.

Similarly, by inspection of Fig. 4, we have the following.

Theorem 3.2: A tail-biting state realization is a normal realization.

More generally, however, these degree restrictions may at first appear to impose significant constraints. However, we will show that

- restriction to normal realizations incurs no loss of generality, because every state realization can be "normalized" by a certain replication procedure;
- restriction to normal realizations incurs no essential cost in complexity, because the normal graph of the "normalized" realization is essentially the same as the factor graph of the original generalized state realization.

A. Normal Graphs

A normal realization has a natural graphical model, which we call a *normal graph* (or, after a while, simply a "graph"). It is defined as follows.

- Each local constraint $\mathcal{C}_i$ is represented by a vertex.
- Each state variable S_j, which by definition is involved in two local constraints, is represented by an ordinary edge between the two corresponding vertices.
- Each symbol variable A_k, which by definition is involved in one local constraint, is represented by a "leaf edge" (or "half-edge") connected to the corresponding constraint vertex.

The resulting normal graph is a *graph with leaves*, comprising a set of vertices representing local constraints, a set of ordinary edges representing state variables, each incident on two vertices, and a set of *leaf edges* representing symbol variables, each incident on only one vertex. It differs from a factor graph in the following respects.

- Whereas a factor graph represents state and symbol variables by vertices, a normal graph represents them by edges. This choice is unconventional for graphical models of state realizations, although conventional for, e.g., block diagrams.
- Whereas a factor graph is bipartite, a normal graph has only one kind of vertex, and there are no restrictions on graph topology.
- Whereas the edges of a factor graph need not be labeled, an edge of a normal graph is labeled by the state or symbol variable that it represents.
- Whereas a factor graph is an ordinary graph, a normal graph is a graph with leaves. This is the most elementary type of hypergraph, namely, a hypergraph in which edge degrees must be 1 or 2.

The factor graph of a normal realization is by definition a factor graph in which all symbol vertices have degree 1 and all state vertices have degree 2. A factor graph satisfying these constraints will be called a *normal factor graph.*

For example, Fig. 10 illustrates the factor graph of a conventional state realization, which is already normal, and the corresponding normal graph. Leaf edges in the normal graph are represented by a special symbol, called a "dongle." The normal

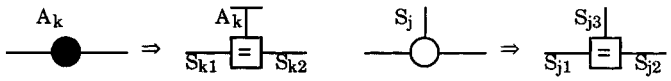

Fig. 11. Conversion of symbol and state variables: factor graph to normal graph.

graph is evidently simpler; however, we will see that both graphs have the same decoding complexity.

We like the term "half-edge" for the degree-1 edges that represent symbol variables. Note that any portion of a normal graph may be regarded as a subsystem. We may isolate the subsystem by cutting a set of boundary edges, so that each such edge (state) becomes a pair of "half-edges" (symbols), one in the disconnected subsystem and one in the remaining system. (See, e.g., Fig. 38.) Conversely, given a system, each half-edge may potentially be coupled to some half-edge in the external environment. Thus, half-edges (symbols) represent the coupling between a system and its environment, whereas ordinary edges (states) represent internal connections.

The special "dongle" symbol for such a half-edge that is introduced in Fig. 10 is intended to be reminiscent of a railroad-car coupler, symbolizing the coupling function described in the previous paragraph. However, any special symbol will do, including a filled circle as in factor graphs.

Note that a graph with leaves may be converted into an ordinary graph by merging leaf edges into their associated vertices (yielding "vertices with hair"). Thus any normal graph may be converted into an ordinary graph, with an implicit symbol variable possibly associated with each constraint node. The topological properties of a graph with leaves and this associated ordinary graph are clearly essentially the same. In particular, the two graphs will have the same properties of connectedness and/or cycle-freedom.

B. Normalization by Replication

We now show that any generalized state realization of $\mathcal{C}$ may be "normalized" to become a normal realization that generates the same code, via the following replication procedure.

Normalization by Replication: Given a generalized state realization of a code $\mathcal{C}$ with symbol variables $\{A_k,\, k \in I_{\mathcal{A}}\}$, state variables $\{S_j,\, j \in I_{\mathcal{S}}\}$ and local codes $\{\mathcal{C}_i,\, i \in I_{\mathcal{C}}\}$, where the symbol and state variables are indexed by $I_{\mathcal{A}}(i)$ and $I_{\mathcal{S}}(i)$, respectively.

1) (Symbol Replication): For each $i \in I_{\mathcal{C}}$, for each $k \in I_{\mathcal{A}}(i)$ create a replica S_{ki} of A_k with alphabet $\mathcal{A}_k$ and replace A_k by S_{ki} in the local constraint $\mathcal{C}_i$. Then for each $k \in I_{\mathcal{A}}$, add a local constraint $\mathcal{C}_k$ that constrains all replicas S_{ki} to be equal to each other and to A_k.
2) (State Replication): For each $i \in I_{\mathcal{C}}$, for each $j \in I_{\mathcal{S}}(i)$ create a replica S_{ji} of S_j with alphabet $\mathcal{S}_j$ and replace S_j by S_{ji} in the local constraint $\mathcal{C}_i$. Then for each $j \in I_{\mathcal{S}}$, add a local constraint $\mathcal{C}_j$ that constrains all replicas S_{ji} to be equal to each other, and delete S_j.

The corresponding conversions from the factor graph of the original state realization to the normal graph of the normalized realization are illustrated in Fig. 11. In the original factor graph, symbol variables are represented by filled vertices and state variables by open vertices. In the normal graph, repetition constraints are represented by vertices labeled by the symbol "=," and leaf edges are represented by the special "dongle" symbol.

The result of this procedure will be called a *normalized state realization.* Notice the following.

- The set $\{A_k,\, k \in I_{\mathcal{A}}\}$ of symbol variables is preserved.
- The set of state variables is replaced by
$$\{\{S_{ki},\, k \in I_{\mathcal{A}}(i)\},\, \{S_{ji},\, j \in I_{\mathcal{S}}(i)\},\, i \in I_{\mathcal{C}}\}.$$
- The set of local constraints is augmented by a new set
$$\{\{\mathcal{C}_k,\, k \in I_{\mathcal{A}}\},\, \{\mathcal{C}_j,\, j \in I_{\mathcal{S}}\}\}.$$
The new local constraints are *repetition constraints* that force all variables involved to be equal.

The following is clear by construction.

- The normalized state realization generates the same code $\mathcal{C}$, since each new state variable S_{ki} may be replaced by the symbol variable A_k to which it is equal, and each new state variable S_{ji} may be replaced by the common value S_j of all the S_{ji}.
- Each symbol variable A_k is now involved in only one local constraint, namely, the new constraint $\mathcal{C}_k$.
- Each state variable S_{ji} or S_{ki} is now involved in precisely two local constraints, namely, the original constraint S_i and a new constraint $\mathcal{C}_j$ or $\mathcal{C}_k$, respectively.

Fig. 12 shows how the bipartite factor graph of a generic generalized state realization is converted to the normal graph of the corresponding normalized state realization. The new normal graph looks exactly like the original factor graph, except for the addition of leaf edges to represent symbol variables. All state or symbol vertices are converted into repetition constraint vertices.

We will see when we discuss decoding that the similarity of these graphs implies that their decoding complexity is essentially the same.

We see also from Fig. 12 that graph topology and global graph-theoretic properties are not essentially affected by this conversion; e.g., graphs that are connected remain connected, and graphs that are cycle-free remain cycle-free.

To summarize, we have the followiwng theorem.

Theorem 3.3: Given a generalized state realization of a code $\mathcal{C}$, normalization yields a normal realization of $\mathcal{C}$. The normal graph of the normalized realization has essentially the same topology as the factor graph of the original realization, and may be decoded with essentially the same decoding complexity.

In other words, perhaps somewhat surprisingly, there is no loss of generality or efficiency in restricting attention to normal realizations.

C. Further Remarks on Normalization

We remark first that normal realizations that are derived by normalization are not the most general normal realizations, because they must satisfy certain restrictions. Namely, in a normal graph of a normalized realization, certain vertices must be repetition vertices, and the graph must be bipartite. Conventional

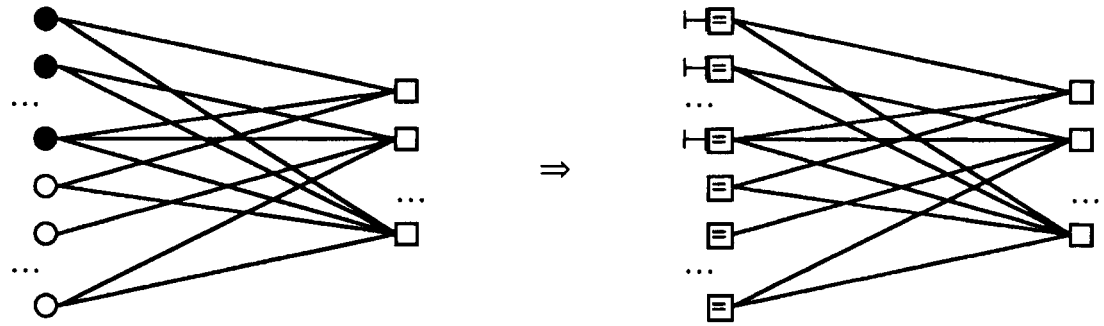

Fig. 12. Conversion of the factor graph of a generic generalized state realization to the normal graph of a normalized realization.

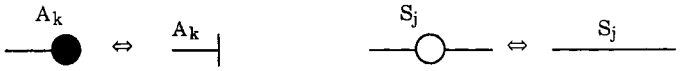

Fig. 13. Conversions for normalized symbol and state variables.

and tail-biting state realizations are examples of normal realizations that are not subject to such restrictions.

Second, note that a state variable that has degree 2 or a symbol variable that has degree 1 does not need to be replicated in the normalization process. Therefore, in these cases the conversion process from a factor graph to a normal graph may be simplified as follows.

1) (Normalized Symbol Variables): For each original symbol vertex A_k of degree 1, replace the symbol vertex and its attached edge by a leaf edge representing A_k.
2) (Normalized State Variables): For each original state vertex S_j of degree 2, replace the state vertex and its two attached edges by a single ordinary edge representing S_j.

These conversions are illustrated in Fig. 13. The symbol "$\Leftrightarrow$" is used because these conversions are evidently invertible.

The conversions of Fig. 13 may be used to convert back and forth between a normal factor graph and a normal graph. Which graphical model is preferable? As we saw earlier in Fig. 10, a normal factor graph looks much like the corresponding normal graph, except for redundant state vertices in the middle of ordinary edges. While these redundant state vertices complicate the graphical model unnecessarily, we will see that in decoding no computation occurs at a state vertex of degree 2; messages simply pass through it. From a complexity viewpoint, therefore, normal graphs and normal factor graphs are essentially equivalent. Aesthetically, we believe that normal graphs are a bit nicer.

D. Block Diagrams

Another familiar type of graphical model is a block diagram. In a block diagram, the edges represent variables, and the vertices ("blocks") typically represent I/O systems, in which some of the incident edges represent inputs to the block and the remaining edges represent outputs. These distinctions are typically indicated by placing arrows on the edges, making a block diagram into a directed graph.

An I/O system defines a local code, but the converse does not hold. Given a local code $\mathcal{C}_i$, a subset of the variables involved in $\mathcal{C}_i$ is called an *information set* if the projection of $\mathcal{C}_i$ onto that subset is a bijective (one-to-one and onto) map. The projection map is then invertible, and the inverse map may be composed with the projection onto the remaining variables (called a *check set* in coding theory) to give an I/O map from the variables in the information set (the "inputs") to the variables in the check set (the "outputs"). Conversely, if no projection of $\mathcal{C}_i$ onto any subset of its variables is bijective, then no such I/O map exists.

If the local code is a group code, then the projection of $\mathcal{C}_i$ onto an information set must be an isomorphism, and the resulting I\O map a homomorphism.

If a local code $\mathcal{C}_i$ has an information set, then we may optionally regard the corresponding vertex in a normal graph as an I\O system. This may be indicated by labeling all incident edges by appropriately directed arrows. If all edges in the graph can be so labeled consistently, then the resulting graph may be regarded as a block diagram. (For an example, see Fig. 29 below.) From a behavioral perspective, such a labeling is purely for illustrative purposes, and does not affect the associated normal realization nor the code that it generates.

E. Pairwise Markov Random Fields (MRFs)

In this section we briefly sketch some intriguing parallels between normal graphs and pairwise Markov random fields (MRFs). An MRF is a undirected graphical model in which vertices correspond to random variables, and the joint probability distribution of the random variables factors into the product of potential functions defined on the maximal cliques of the graph (see, e.g., [27] or [45], this issue). Thus, an MRF is essentially a factor graph, with functions (constraints) represented by cliques rather than by vertices [27]. Alternatively, an MRF could be regarded as a hypergraph in which cliques represent hypergraph edges.

A pairwise MRF is an MRF in which the degrees of all functions are restricted to 1 or 2. Any MRF may be "normalized" to a pairwise MRF via a replication procedure [45].

The pairwise restriction allows a number of remarkable results to be proved [50]. One can define a "Bethe free energy" function on the MRF graph such that the sum-product algorithm ("belief propagation") minimizes this free energy approximately (exactly, if the MRF graph is cycle-free). It follows that the fixed points of the sum-product algorithm are the zero-gradient points of the free energy, and, since the free energy is bounded below, that the sum-product algorithm always has a fixed point, regardless of graph topology.

These results follow from restricting function (constraint) degrees to 2 or less, just as the results of this paper result from

restricting variable degrees to 2 or less. Either restriction may be made without loss of generality or essential change to graph topology. These two restrictions are clearly mirror images of one another, and both seem very useful. However, it is clearly not possible to restrict both function (constraint) and variable degrees to 2 or less, because the resulting graph could only consist of disconnected simple cycles and chains.

IV. Graph-Theoretic Properties of Realizations

A normal realization is completely characterized by its normal graph, and *vice versa*. Therefore, we will henceforth identify a normal realization with its normal graph. We may also omit the word "normal." We may therefore say that a code is realized or generated by a certain graph. Moreover, we may say that a realization has certain graph-theoretic properties, such as connectedness or cycle-freedom.

In this section, we will discuss important graph-theoretic properties of realizations and the corresponding properties of codes.

We first show that a code has a disconnected realization if and only if it is the Cartesian product of shorter codes. In this case, we may consider the shorter codes independently.

Secondly, we consider cut sets. A connected graph is cycle-free if and only if every edge is by itself a cut set. Every cut set of a realization induces a certain decomposition of the code into a union of Cartesian product codes.

From this cut-set decomposition follows a lower bound (the Cut-Set Bound) on the product of the sizes of the state spaces in any cut set in terms of the minimal state-space size in a certain conventional trellis realization. In the linear or group case, this lower bound may be evaluated via the State Space Theorem. If the graph is also cycle-free, then there exists a canonical realization in which all of these lower bounds are met with equality—i.e., a minimal realization.

The Cut-Set Bound shows that state-space sizes cannot be reduced substantially unless the realization has cycles, which motivates the consideration of realizations with cycles.

In the following section, we will develop a general exact decoding algorithm for cycle-free realizations, called the sum-product algorithm. This development will also be based on cut sets.

A. Connectedness and Independence

Suppose that a realization is disconnected. Then the code generated by the realization is the Cartesian product of the codes generated by the components of the realization. Conversely, if a code can be represented as a Cartesian product of shorter codes, then the disjoint union of any set of realizations of the shorter codes is a realization of the original code.

Thus, a realization of a code can be disconnected if and only if the code can be decomposed into the Cartesian product of shorter codes. The component codes are then independent of each other, and may be considered separately. Therefore, we will henceforth consider only codes that cannot be so decomposed, whose realizations are necessarily connected.

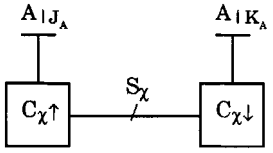

Fig. 14. Aggregated view of a realization with a cut set χ.

B. Cut Sets and Conditional Independence

In this subsection, we introduce the important topic of cut sets, and prove a Cut-Set Decomposition Lemma, which can be interpreted as a statement of conditional independence.

A *cut set* of a connected graph is a minimal set of edges such that removal of that set partitions the graph into two disconnected subgraphs. A connected graph is *cycle-free* if and only if every ordinary edge is by itself a cut set.

In a normal graph, a cut set consists of a set of ordinary (state) edges, and may be specified by the corresponding subset $\chi \subseteq I_{\mathcal{S}}$ of state indices. The cut-set alphabet is then the Cartesian product $\mathcal{S}_\chi = \prod_{j\in\chi} \mathcal{S}_j$, with values $\boldsymbol{s}_\chi = \{s_j, j \in \chi\}$ and size $|\mathcal{S}_\chi| = \prod_{j\in\chi} |\mathcal{S}_j|$. Of course, if the cut set is a single edge, then the cut-set alphabet is simply the associated state space $\mathcal{S}_j$ with values s_j and size $|\mathcal{S}_j|$.

Since removal of a cut set partitions a graph into two disconnected subggraphs, the index sets $I_{\mathcal{A}}$, $I_{\mathcal{S}} - \chi$, and $I_{\mathcal{C}}$ are correspondingly partitioned into disjoint index sets, say $I_{\mathcal{A}} = \{J_{\mathcal{A}}, K_{\mathcal{A}}\}$, $I_{\mathcal{S}} - \chi = \{J_{\mathcal{S}}, K_{\mathcal{S}}\}$, and $I_{\mathcal{C}} = \{J_{\mathcal{C}}, K_{\mathcal{C}}\}$.

Fig. 14 gives a high-level view of a realization with a cut set χ. On one side of χ, the local codes $\{\mathcal{C}_i, i \in J_{\mathcal{C}}\}$ and state variables $\{S_j, j \in J_{\mathcal{S}}\}$ are aggregated into an "upstream" code $\mathcal{C}_{\chi\uparrow}$ that constrains the upstream symbol variables $\mathcal{A}_{|J_{\mathcal{A}}}$ and the cut-set alphabet S_χ. Similarly, an aggregated "downstream" code $\mathcal{C}_{\chi\downarrow}$ constrains the downstream symbol variables $\mathcal{A}_{|K_{\mathcal{A}}}$ and S_χ.

As noted in [27], fixing a variable (or conditioning on that variable) effectively removes the corresponding edge in the corresponding graph. The reason is that a function of a unary (single-valued) variable does not actually depend on that variable.

If $\mathcal{S}_\chi$ is a cut set of a normal graph, therefore, fixing $\mathcal{S}_\chi = \boldsymbol{s}_\chi$ partitions the graph of Fig. 14 into two disconnected subgraphs. Every cut set $\mathcal{S}_\chi$ of the graph thus corresponds to an instance of *conditional independence.*

In detail, for each $\boldsymbol{s}_\chi$, the code $\mathcal{C}_\chi(\boldsymbol{s}_\chi)$ generated by the disconnected graph (i.e., the set of all codewords *consistent with* $\mathcal{S}_\chi = \boldsymbol{s}_\chi$) is a Cartesian product

$$\mathcal{C}_\chi(\boldsymbol{s}_\chi) = \mathcal{C}_{\chi\uparrow}(\boldsymbol{s}_\chi) \times \mathcal{C}_{\chi\downarrow}(\boldsymbol{s}_\chi)$$

where the codes $\mathcal{C}_{\chi\uparrow}(\boldsymbol{s}_\chi) \subseteq \mathcal{A}_{|J_{\mathcal{A}}}$ and $\mathcal{C}_{\chi\downarrow}(\boldsymbol{s}_\chi) \subseteq \mathcal{A}_{|K_{\mathcal{A}}}$ are generated independently by the two graph components. The code $\mathcal{C}$ generated by the original connected graph is then the union of these Cartesian product codes:

Lemma 4.1 (Cut-Set Decomposition Lemma): If a code $\mathcal{C}$ has a realization with a cut set χ as in Fig. 14, then $\mathcal{C}$ has the following decomposition:

$$\mathcal{C} = \bigcup_{\boldsymbol{s}_\chi \in \mathcal{S}_\chi} \mathcal{C}_{\chi\uparrow}(\boldsymbol{s}_\chi) \times \mathcal{C}_{\chi\downarrow}(\boldsymbol{s}_\chi). \tag{4.1}$$

C. The Cut-Set Bound

Observe that Fig. 14 may be interpreted as a conventional two-section state realization (trellis) of $\mathcal{C}$ on the two-section time axis $\{J_\mathcal{A}, K_\mathcal{A}\}$, with central state space $\mathcal{S}_\chi$. The two trellis sections are determined by the two local constraints

$$\mathcal{C}_{\chi\uparrow} = \{(\boldsymbol{a}_{|J_\mathcal{A}}, \boldsymbol{s}_\chi) | \boldsymbol{s}_\chi \in \mathcal{S}_\chi, \boldsymbol{a}_{|J_\mathcal{A}} \in \mathcal{C}_{\chi\uparrow}(\boldsymbol{s}_\chi)\}$$

and

$$\mathcal{C}_{\chi\downarrow} = \{(\boldsymbol{a}_{|K_\mathcal{A}}, \boldsymbol{s}_\chi) | \boldsymbol{s}_\chi \in \mathcal{S}_\chi, \boldsymbol{a}_{|K_\mathcal{A}} \in \mathcal{C}_{\chi\downarrow}(\boldsymbol{s}_\chi)\}$$

with the initial and final state spaces being implicit. Such a two-section trellis realization implies the decomposition of Lemma 4.1.

In other words, the alphabet $\mathcal{S}_\chi$ of a cut set χ is a sufficient state space for a partition of the "time axis" $I_\mathcal{A}$ of $\mathcal{C}$ into a "past" $J_\mathcal{A}$ and "future" $K_\mathcal{A}$.

From this observation follows the important Cut-Set Bound [47], [8].

Theorem 4.2 (Cut-Set Bound): Let $I_\mathcal{A}$ be the symbol index set of a code $\mathcal{C}$, let χ be a cut set in the graph of a normal realization of $\mathcal{C}$, and let $J_\mathcal{A}$ and $K_\mathcal{A}$ be the two disjoint subsets of symbol indices in the graph partition induced by removal of the edges in χ. Then $|\mathcal{S}_\chi| = \prod_{j\in\chi} |\mathcal{S}_j|$ is lower-bounded by the minimum central state space size in any two-section conventional state realization of $\mathcal{C}$ on the two-section time axis $\{J_\mathcal{A}, K_\mathcal{A}\}$.

Proof: Under these hypotheses, (4.1) holds, so there exists a two-section realization of $\mathcal{C}$ on the two-section time axis $\{J_\mathcal{A}, K_\mathcal{A}\}$ with central state space $\mathcal{S}_\chi$. □

D. The State Space Theorem

When $\mathcal{C}$ is a linear or group code, the minimum state-space size of the Cut-Set Bound is easily determined via the State Space Theorem [16], [48].

In discrete-time system theory, a state space $\mathcal{S}_k$ arises from a cut of an ordered index set (time axis) $I \subseteq \mathbb{Z}$ at time k, which partitions I into two disjoint subsets, the "past" $k^- = \{i \in I | \, i < k\}$ and the "future" $k^+ = \{i \in I | \, i \geq k\}$. Here we consider more general two-way partitions of an *unordered* index set I into disjoint subsets $\{J, K\}$, where $K = I - J$.

Given a linear or group code $\mathcal{C}$ defined on an index set I and a subset $J \subseteq I$, we define its *projection* $\mathcal{C}_{|J} = \{\boldsymbol{a}_{|J} | \boldsymbol{a} \in \mathcal{C}\}$ and *cross section* $\mathcal{C}_J = \{\boldsymbol{a}_{|J} | \boldsymbol{a} \in \mathcal{C}, \boldsymbol{a}_{|K} = \boldsymbol{0}\}$, where $K = I - J$ and $\boldsymbol{0}$ denotes the all-zero (identity) configuration. It is immediate that $\mathcal{C}_{|J}$ and $\mathcal{C}_J$ are vector spaces or groups, and that $\mathcal{C}_J$ is a subspace or normal subgroup of $\mathcal{C}_{|J}$.

The following easy but extremely important result was shown in [16]. (For linear systems, this result was essentially given in [48].)

Theorem 4.3 (State Space Theorem): Given a linear or group code $\mathcal{C}$ defined on an index set I, let $\{J, K\}$ be any two-way partition of I. Then the following three quotient groups are isomorphic:

1) the $\{J, K\}$-induced state space $\mathcal{S}_{JK} = \mathcal{C}/(\mathcal{C}_J \times \mathcal{C}_K)$;
2) the J-induced state space $\mathcal{S}_J = \mathcal{C}_{|J}/\mathcal{C}_J$;
3) the K-induced state space $\mathcal{S}_K = \mathcal{C}_{|K}/\mathcal{C}_K$.

Any minimal state space corresponding to any cut such that J is the past and K is the future must be isomorphic to $\mathcal{S}_{JK}$.

More precisely, define the $\{J, K\}$-induced, J-induced, and K-induced state maps as the natural maps σ_{JK}: $\mathcal{C} \to \mathcal{S}_{JK}$, σ_J: $\mathcal{C}_{|J} \to \mathcal{S}_J$, and $\sigma_K : \mathcal{C}_{|K} \to \mathcal{S}_K$, respectively. Then there exist isomorphisms ι_J: $\mathcal{S}_J \to \mathcal{S}_{JK}$ and ι_K: $\mathcal{S}_K \to \mathcal{S}_{JK}$ such that the following diagram commutes, where π_J and π_K are the projections onto J and K, respectively:

$$\begin{array}{ccccc} \mathcal{C}_{|J} & \xleftarrow{\pi_J} & \mathcal{C} & \xrightarrow{\pi_K} & \mathcal{C}_{|K} \\ \downarrow\sigma_J & & \downarrow\sigma_{JK} & & \downarrow\sigma_K \\ \mathcal{S}_J & \xrightarrow{\iota_J} & \mathcal{S}_{JK} & \xleftarrow{\iota_K} & \mathcal{S}_K \end{array}$$

This implies that there exists a state realization of $\mathcal{C}$ with the following full behavior:

$$\mathfrak{B} = \{(\boldsymbol{a}_{|J}, s_{JK}, \boldsymbol{a}_{|K}) \in \mathcal{C}_{|J} \times \mathcal{S}_{JK} \times \mathcal{C}_{|K} | \\ \iota_J(\sigma_J(\boldsymbol{a}_{|J})) = s_{JK} = \iota_K(\sigma_K(\boldsymbol{a}_{|K}))\}$$

or, equivalently, that $\mathcal{C}$ may be decomposed as follows:

$$\mathcal{C} = \bigcup_{s_{JK} \in \mathcal{S}_{JK}} \mathcal{C}_{|J}(s_{JK}) \times \mathcal{C}_{|K}(s_{JK}) \tag{4.2}$$

where

$$\mathcal{C}_{|J}(s_{JK}) = \{\boldsymbol{a}_{|J} \in \mathcal{C}_{|J} | \iota_J(\sigma_J(\boldsymbol{a}_{|J})) = s_{JK}\}$$

and

$$\mathcal{C}_{|K}(s_{JK}) = \{\boldsymbol{a}_{|K} \in \mathcal{C}_{|K} | \iota_K(\sigma_K(\boldsymbol{a}_{|K})) = s_{JK}\}$$

are the cosets of $\mathcal{C}_J$ and $\mathcal{C}_K$ that map to s_{JK}, respectively, and $\mathcal{C}_{|J}(s_{JK}) \times \mathcal{C}_{|K}(s_{JK})$ is the corresponding coset of $\mathcal{C}_J \times \mathcal{C}_K$ in $\mathcal{C}$.

If $\mathcal{C}$ is a linear code, then this development may be summarized as follows. $\mathcal{C}$ has a generator matrix of the following form:

$$G_\mathcal{C} = \begin{bmatrix} G_{\mathcal{C}_J} & 0 \\ 0 & G_{\mathcal{C}_K} \\ G_{\mathcal{C}_{|J}/\mathcal{C}_J} & G_{\mathcal{C}_{|K}/\mathcal{C}_K} \end{bmatrix}$$

where $G_{\mathcal{C}_J}$ and $G_{\mathcal{C}_K}$ are generator matrices for the cross sections $\mathcal{C}_J$ and $\mathcal{C}_K$, and $(G_{\mathcal{C}_J}, G_{\mathcal{C}_{|J}/\mathcal{C}_J})$ and $(G_{\mathcal{C}_K}, G_{\mathcal{C}_{|K}/\mathcal{C}_K})$ are generator matrices for the projections $\mathcal{C}_{|J}$ and $\mathcal{C}_{|K}$. The dimension of the minimal state space $\mathcal{S}_{JK}$ is the dimension of $G_{\mathcal{C}_{|J}/\mathcal{C}_J}$ or of $G_{\mathcal{C}_{|K}/\mathcal{C}_K}$.

From (4.2) we see that $\mathcal{S}_{JK}$ may be used as the central state space in a two-section state realization of $\mathcal{C}$. Moreover, because the cosets $\mathcal{C}_{|J}(s_{JK}) \times \mathcal{C}_{|K}(s_{JK})$ are disjoint, it is clear that no smaller state space is possible.

E. The Cut-Set Bound for Linear or Group Codes

For linear or group codes, the State Space Theorem implies the following.

Corollary 4.4 (Cut-Set Bound for Linear or Group Codes): Let $I_\mathcal{A}$ be the symbol index set of a linear or group code $\mathcal{C}$, let χ be a cut set in the graph of a normal realization of $\mathcal{C}$, and let $J_\mathcal{A}$ and $K_\mathcal{A}$ be the two disjoint subsets of symbol indexes in the graph partition induced by removal of the edges in χ. Then $|\mathcal{S}_\chi| = \prod_{j\in\chi} |\mathcal{S}_j|$ is lower-bounded by the size of the $\{J_\mathcal{A}, K_\mathcal{A}\}$-induced state space $\mathcal{S}_{J_\mathcal{A}K_\mathcal{A}} = \mathcal{C}/(\mathcal{C}_{J_\mathcal{A}} \times \mathcal{C}_{K_\mathcal{A}})$ of $\mathcal{C}$: $|\mathcal{S}_\chi| \geq |\mathcal{S}_{J_\mathcal{A}K_\mathcal{A}}|$. □

F. Canonical Realizations on Cycle-Free Graphs

If a normal graph representing a code $\mathcal{C}$ is cycle-free, then every ordinary edge is a cut set; i.e., every state variable S_j partitions the symbol index set $I_{\mathcal{A}}$ into two disjoint subsets, $J_{\mathcal{A}}$ and $K_{\mathcal{A}}$. For linear or group codes, the State Space Theorem defines the $\{J_{\mathcal{A}}, K_{\mathcal{A}}\}$-induced state space

$$\mathcal{S}_{J_{\mathcal{A}}K_{\mathcal{A}}} = \mathcal{C}/(\mathcal{C}_{J_{\mathcal{A}}} \times \mathcal{C}_{K_{\mathcal{A}}})$$

as the canonical state space associated with this partition.

We, therefore, define a *canonical normal realization* as follows. Define $\mathcal{S}_j = \mathcal{S}_{J_{\mathcal{A}}K_{\mathcal{A}}}$, define σ_j as the natural state map $\sigma_j: \mathcal{C} \rightarrow \mathcal{S}_j$, and define $\boldsymbol{\sigma}: \mathcal{C} \rightarrow \mathcal{S}$ as the composition of these state maps. Define the full behavior $\mathfrak{B}$ as the set of all configurations

$$\mathfrak{B} = \{(\boldsymbol{a}, \boldsymbol{s}) \in \mathcal{C} \times \mathcal{S} | \boldsymbol{\sigma}(\boldsymbol{a}) = \boldsymbol{s}\} = \{(\boldsymbol{a}, \boldsymbol{\sigma}(\boldsymbol{a})), \boldsymbol{a} \in \mathcal{C}\}.$$

Then $\mathcal{C} = \mathfrak{B}_{|\mathcal{A}}$, so the realization generates $\mathcal{C}$. Moreover, the lower bound on S_j of Corollary 4.4 is met everywhere with equality.

Such a realization is minimal in the sense of meeting the Cut-Set Bound for all state spaces $\mathcal{S}_j$. Moreover, it is a linear or group realization, since the state spaces are all vector spaces or groups and the local codes $\mathcal{C}_i$ are all homomorphic images of $\mathcal{C}$:

$$\mathcal{C}_i = \{(\boldsymbol{a}_{|I_{\mathcal{A}}(i)}, (\boldsymbol{\sigma}(\boldsymbol{a}))_{|I_{\mathcal{S}}(i)}), \boldsymbol{a} \in \mathcal{C}\}.$$

For example, the graph of a conventional state realization is cycle-free. For this case, our canonical state realization reduces to the canonical conventional state realization of [16].

G. Realizations on Graphs with Cycles

On cycle-free graphs, each edge is a cut set, and the size of the state space associated with that edge is lower-bounded by the Cut-Set Bound to be at least as great as that of a state space in some conventional state realization (trellis). Therefore, no dramatic reduction in complexity over conventional realizations can be expected by using more general cycle-free realizations.

On the other hand, on graphs with cycles, cut sets generally comprise multiple state variables, and the minimum complexity implied by the Cut-Set Bound may be spread across these variables; therefore, dramatic reductions in state complexity may be obtained.

For example, consider a tail-biting realization, with a circular graph as shown in Fig. 15. Here, every cut set comprises two edges. Consequently, the minimum size of both such state spaces can be as small as the *square root* of the value given by the Cut-Set Bound [46].

It has been shown, for example, that whereas in any conventional state realization of the $(24, 12, 8)$ Golay code the size of the state space at the trellis midpoint must have at least 256 states, there exists a tail-biting realization in which the size of each state space is only 16 [8]. Simulations using this tail-biting realization have shown a performance degradation of only about 0.1 dB relative to maximum-likelihood (ML) decoding [36].

On the other hand, the question of finding "minimal" tail-biting realizations is in general quite difficult (see, e.g., [25]), even for this simplest case of a graph with cycles.

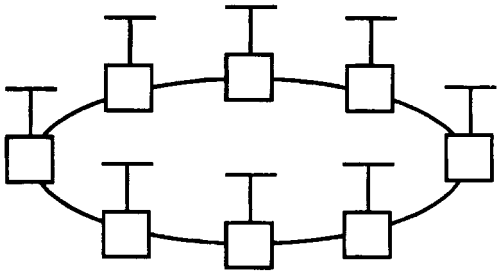

Fig. 15. Normal graph of a tail-biting realization.

H. Normal Realizations of Capacity-Approaching Codes

Spectacular progress has recently been made on capacity-approaching coding schemes, using codes that are naturally defined on graphs.

For example, the original turbo codes of Berrou *et al.* [6] are realized as shown in Fig. 16. A data bit sequence is duplicated and used as the information bits in two low-complexity rate-$1/2$ convolutional codes, in one case after being permuted in a long, pseudo-random permuter Π. The code symbols are the data sequence and the two parity sequences, so the code rate is $1/3$.

Given received symbol APPs, each convolutional code may be decoded by the sum-product decoding algorithm of the next section to give updated APPs, which may be relayed via the data-bit repetition constraints to the other convolutional decoder. Decoding continues in this way for 10–20 iterations. On an additive white Gaussian noise (AWGN) channel, performance within 0.35 dB of the Shannon limit has been achieved.

A turbo code is thus a long, powerful, pseudo-random code with cycles comprised of simpler cycle-free codes. Its length and pseudo-randomness are induced by the permuter, which does not require any computations in decoding. The decoding operations are performed iteratively on the component low-complexity, cycle-free graphs. Note that the overall code graph must have cycles, since, as we have seen from the Cut-Set Bound, the state spaces could not be small otherwise.

As another example, Gallager's LDPC codes [19] are realized by a Tanner normal graph, as shown in Fig. 17. Connections between data bits and checks are realized by a long, pseudo-random permuter Π.

Decoding alternates between the repetition and parity-check constraints. The overall graph has cycles, but they can be made very long. With highly irregular repetition-constraint degrees on an AWGN channel, performance within 0.005 dB of the Shannon limit has now been achieved [10].

Finally, remarkably good results have been achieved even with the simple "repeat–accumulate" (RA) code of Fig. 18 [12]. Here the data bits are simply repeated n times, permuted in a long, pseudo-random permuter Π, and then used as the information bits in a trivial rate-$1/1$ 2-state "accumulate" code, whose states are the code symbols. On an AWGN channel, with turbo-like decoding, performance within 1.0 dB of the Shannon limit has been achieved.

These remarkable results show that there are many possible routes to channel capacity. The common element is a long, pseudo-random permuter, which makes the codes long and pseudo-random without itself incurring any computational

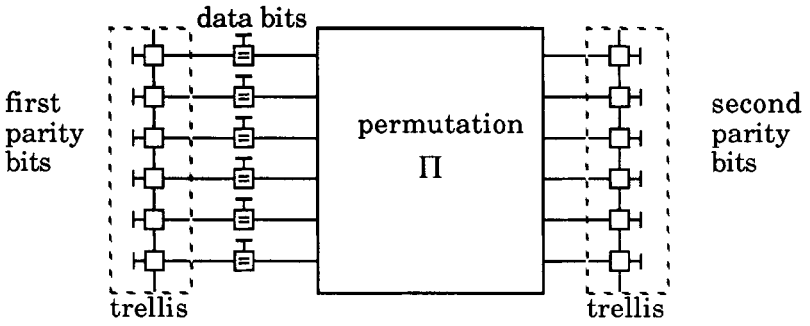

Fig. 16. Normal graph of a classical turbo code.

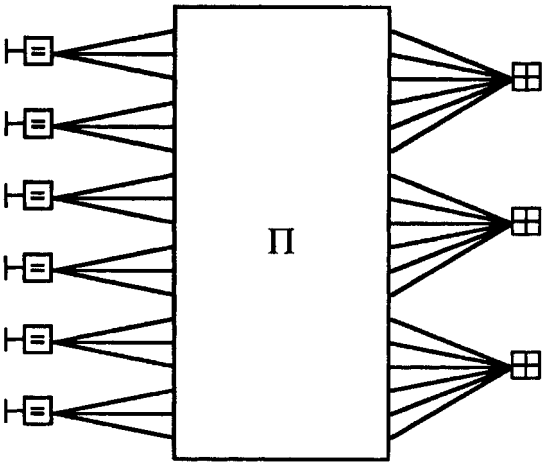

Fig. 17. Normal graph of a LDPC code.

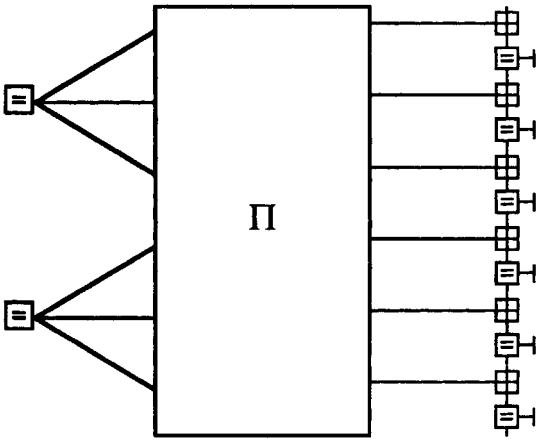

Fig. 18. Normal graph of an RA code.

complexity. Outside of the permuter, two (or more) low-complexity cycle-free codes are used—it does not seem to matter much which, or which bits are tapped to be code symbols. The overall code must have cycles, but with a long permuter the cycles can be made very long, so that the graph structure within some radius of any vertex can be made locally cycle-free. The sum-product decoding algorithm then operates almost as though on a cycle-free graph.

V. The Sum-Product Algorithm

One of the most important properties of a graphical model of a realization of a code is that it specifies an associated "decoding" algorithm (actually, a class of decoding algorithms): namely, the sum-product algorithm.

By now, there are many expositions of the sum-product algorithm; e.g., in the context of factor graphs, see [27]. The reader who is already familiar with the factor-graph algorithm may adapt it to normal graphs simply by applying the factor-graph algorithm to the equivalent normal factor graph. We give a brief development of the sum-product algorithm for normal graphs here not only to make our presentation self-contained, but also because we believe that the development is somewhat simpler in the normal graph context, mainly because it focuses on edges and cuts.

The sum-product algorithm is exact only on cycle-free graphs. However, the sum-product update rule may be used for approximate iterative decoding on graphs with cycles. We observe that deterministic and/or erased variables can sometimes have the effect of "breaking" cycles.

We observe that with normal graphs there is a clean functional separation between the elements of the graph and the elements of the sum-product algorithm: leaf edges (symbols) are purely for I\O, ordinary edges (states) are purely for communications, and all computation takes place at vertices (local codes).

Finally, we observe that graph fragments, including individual vertices (local codes), may be decoded as lower level "modules" or "subroutines."

A. The Sum-Product Algorithm on Cycle-Free Graphs

The decoding problem solved by the sum-product algorithm is as follows.

Assume that a code $\mathcal{C}$ defined on a finite index set $I_{\mathcal{A}}$ has a normal realization whose graph is finite, connected and *cycle-free.* In other words, every (ordinary) edge is a cut set.

Assume further that observations are made on all symbols $\{A_k,\ k \in I_{\mathcal{A}}\}$, resulting in a set of input weight vectors

$$\{\boldsymbol{i}_k = \{i_k(a_k),\ a_k \in A_k\} | k \in I_{\mathcal{A}}\}$$

e.g., likelihood vectors with $i_k(a_k) = p(y_k|a_k)$, where y_k is the kth observation. The *weight* of a codeword $\boldsymbol{a} \in \mathcal{C}$ is then defined as the componentwise product

$$w(\boldsymbol{a}) = \prod_{k \in I_{\mathcal{A}}} i_k(a_k)$$

e.g., with likelihood vectors, $w(\boldsymbol{a}) = p(\boldsymbol{y}|\boldsymbol{a})$.

The sum-product algorithm computes for every value a_k of every symbol variable A_k the sum

$$w_k(a_k) = \sum_{\boldsymbol{a} \in \mathcal{C}_k(a_k)} w(\boldsymbol{a})$$

where $\mathcal{C}_k(a_k)$ is the set of codewords that are consistent with a_k; i.e., whose kth symbol has value a_k. Similarly, for every value s_j of every state variable S_j, it computes

$$w_j(s_j) = \sum_{\boldsymbol{a} \in \mathcal{C}_j(s_j)} w(\boldsymbol{a})$$

where $\mathcal{C}_j(s_j)$ is the set of codewords that are consistent with s_j.

If weight vectors are likelihood vectors and codewords are equiprobable, then the weight vector $\boldsymbol{w}_k = \{w_k(a_k)\}$ is proportional up to a scale factor α to the APP vector $\{p(A_k = a_k|\boldsymbol{y})\}$. We write this using the notation "$\equiv_\alpha$" ("equivalent up to scaling") as

$$\boldsymbol{w}_k \equiv_\alpha \{p(A_k = a_k|\boldsymbol{y})\}.$$

The sum-product algorithm is therefore sometimes called the *APP decoding algorithm* [19]. In the context of Bayesian networks, it is called *belief propagation* [34]. In general, we will assume that two weight vectors that are proportional up to scaling are equivalent.

For the purposes of this development, it is convenient to regard every leaf edge as being terminated by a leaf vertex, as in the factor graph picture. Leaf edges then become ordinary edges. For uniformity, we may regard these new ordinary edges as representing a set of additional state spaces S_k corresponding to the symbol alphabets A_k. The state (edge) index set thus becomes $I'_{\mathcal{S}} = I_{\mathcal{A}} \cup I_{\mathcal{S}}$.

The algorithm is based on two fundamental principles that we will now develop.

1) (Upstream/Downstream Decomposition): For each $j \in I_{\mathcal{S}}{}'$, the jth edge weight vector $\boldsymbol{w}_j$ is the componentwise product of "upstream" and "downstream" weight vectors $\boldsymbol{w}_{j\uparrow}$ and $\boldsymbol{w}_{j\downarrow}$ (see (5.3)), which may be computed independently on the two disconnected components of the graph that results from deleting the jth edge.
2) (Sum-Product Update Rule): The components of an upstream weight vector $\boldsymbol{w}_{j\uparrow}$ are given by a sum of products of components of the next-further upstream weight vectors, consistent with the upstream local code $\mathcal{C}_i$ (see (5.4)).

For the first principle, we use the Cut-Set Decomposition Lemma (Lemma 4.1). For each $j \in I_{\mathcal{S}}{}'$, the jth edge is a cut set whose removal partitions the graph into two disconnected components, which we label arbitrarily as "upstream" and "downstream." Every codeword $\boldsymbol{c} \in \mathcal{C}$ may be correspondingly decomposed into upstream and downstream components: $\boldsymbol{c} = (\boldsymbol{c}_\uparrow, \boldsymbol{c}_\downarrow)$. By Lemma 4.1, $\mathcal{C}$ may be decomposed as a union of Cartesian products

$$\mathcal{C} = \bigcup_{s_j \in S_j} \mathcal{C}_{j\uparrow}(s_j) \times \mathcal{C}_{j\downarrow}(s_j)$$

where $\mathcal{C}_{j\uparrow}(s_j)$ and $\mathcal{C}_{j\downarrow}(s_j)$ are the sets of upstream and downstream components consistent with s_j, respectively.

We may now apply an elementary Cartesian product lemma:

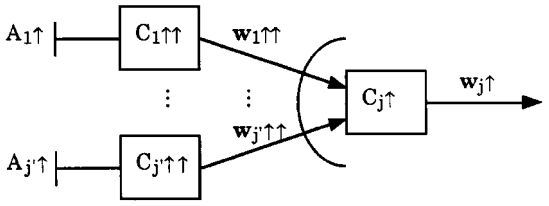

Fig. 19. Local configuration for sum-product update rule.

Lemma 5.1 (Cartesian-Product Distributive Law): If X and Y are disjoint discrete sets and $f(x)$ and $g(y)$ are any two functions defined on X and Y, then

$$\sum_{(x,y) \in X \times Y} f(x)g(y) = \left(\sum_{x \in X} f(x) \right) \left(\sum_{y \in Y} g(y) \right).$$

This Cartesian-product distributive law holds whenever $f(x)$ and $g(y)$ take values in any commutative semiring and arithmetic is performed in this semiring (cf. end of this subsection), and is the fundamental reason why the sum-product algorithm is "fast," as pointed out in [1].

Taking $X = \mathcal{C}_{j\uparrow}(s_j)$ and $Y = \mathcal{C}_{j\downarrow}(s_j)$ in Lemma 5.1, we conclude that

$$w_j(s_j) = w_{j\uparrow}(s_j) w_{j\downarrow}(s_j) \tag{5.3}$$

i.e., the jth-edge weight vector $\boldsymbol{w}_j$ is the componentwise product of upstream and downstream weight vectors $\boldsymbol{w}_{j\uparrow}$ and $\boldsymbol{w}_{j\downarrow}$, where, e.g.

$$w_{j\uparrow}(s_j) = \sum_{\boldsymbol{c}\uparrow \in \mathcal{C}_{j\uparrow}(s_j)} w(\boldsymbol{c}_\uparrow).$$

To compute weights, the algorithm proceeds iteratively according to the following sum-product update rule. Let $\mathcal{C}_i$ be the local code corresponding to the vertex upstream of the jth edge. Let $I'_{\mathcal{S}}(i) = I_{\mathcal{S}}(i) - \{j\}$ denote the set of indexes of all other edges incident on the ith vertex; then the set $\mathcal{C}_{j\uparrow}(s_j)$ of codewords upstream of the jth edge that are consistent with s_j is

$$\mathcal{C}_{j\uparrow}(s_j) = \bigcup_{\mathcal{C}_i(s_j)} \prod_{j' \in I_{\mathcal{S}}(i)'} \mathcal{C}_{j'\uparrow\uparrow}(s_{j'})$$

where $\mathcal{C}_i(s_j)$ is the set of codewords consistent with $S_j = s_j$, and $\mathcal{C}_{j'\uparrow\uparrow}(s_{j'})$ is the set of codewords upstream of the j'th edge that are consistent with $s_{j'}$. This situation is illustrated in Fig. 19.

Now suppose that we know all upstream weight vectors $\{\boldsymbol{w}_{j'\uparrow\uparrow}, j' \in I_{\mathcal{S}}(i)'\}$. Then, again using Lemma 5.1, it follows that the components of the weight vector $\boldsymbol{w}_{j\uparrow}$ are given by the following *sum-product update rule*:

$$w_{j\uparrow}(s_j) = \sum_{\mathcal{C}_i(s_j)} \prod_{j' \in I_{\mathcal{S}}(i)'} w_{j'\uparrow\uparrow}(s_{j'}). \tag{5.4}$$

The computational complexity of the sum-product update rule is evidently proportional to $|\mathcal{C}_i|$, since for every codeword in $\mathcal{C}_i$ a product of $\delta_i - 1$ upstream weight vectors must be computed and added to a running sum, where δ_i is the degree of the ith vertex. This involves $(\delta_i - 2)|\mathcal{C}_i|$ multiplications, and must be done once for the δ_i outgoing edges from $\mathcal{C}_i$. Therefore, the total number of multiplications associated with the ith node is $\delta_i(\delta_i - 2)|\mathcal{C}_i|$.

Note that for a conventional state realization, $|\mathcal{C}_i|$ is the "branch complexity" of the ith trellis section, which measures decoding complexity more precisely than the "state complexity" [44].

If the local code $\mathcal{C}_i$ is a repetition code, then the sum-product update rule (5.4) reduces to the *product update rule*

$$w_{j\uparrow}(s_j) = \prod_{j' \in I_{\mathcal{S}}(i)'} w_{j'\uparrow\uparrow}(s_j) \tag{5.5}$$

i.e., $\boldsymbol{w}_{j\uparrow}$ is just the componentwise product of the upstream weight vectors $\boldsymbol{w}_{j'\uparrow\uparrow}$, all of which have the same alphabet $\mathcal{S}_j$. In many descriptions of the sum-product algorithm for factor graphs, the product update rule is stated as a separate rule for symbol and state vertex updates.

Note further that if a repetition vertex has degree 2, then there is only one upstream edge with index j', and we have simply a "pass-through" $\boldsymbol{w}_{j\uparrow} = \boldsymbol{w}_{j'\uparrow\uparrow}$ with no computation. This explains why we can suppress degree-2 repetition vertices (e.g., redundant state vertices).

There are many possible schedules for this iteration [27]. In the *sequential schedule*, the sum-product update rule is computed at the jth edge as soon as all upstream weight vectors are known. In a finite cycle-free graph, the upstream graph from the jth edge is a tree with finite depth d, and the trees from the upstream edges have maximum depth $d-1$. Therefore, starting with leaf edges, whose upstream depth is $d = 1$ and whose upstream weight vector $\boldsymbol{w}_{k\uparrow}$ is the input weight vector $\boldsymbol{i}_k$, all weight vectors at depth d may be computed at the dth cycle of the algorithm. The algorithm completes with the correct answer in $d_{\max}$ cycles, where the graph *diameter* $d_{\max}$ is the maximum depth of any edge in either direction. The sum-product update rule is computed once for each value of each edge in each direction.

Alternatively, in the *parallel schedule*, the two weight vectors of every edge are updated on every cycle, using whatever upstream weight vectors are available. Weight vectors at depth d again become correct on the dth cycle. The parallel schedule is particularly suitable for asynchronous and even analog implementations [30].

The sum-product algorithm generalizes to any commutative semiring in which a "sum" and "product" that satisfy the associative, distributive, and commutative rules of ordinary arithmetic can be defined [1], [27]. An important example is the *min-sum semiring* $\{\mathbb{R}_{\geq 0} \cup \infty\}$, with "addition" defined as taking the minimum, and "multiplication" defined as the usual sum; the additive and multiplicative identities are then ∞ and 0, respectively. In the min-sum semiring, the sum-product algorithm becomes the *min-sum algorithm*, which computes for every a_k

$$w(a_k) = \min_{\boldsymbol{a} \in \mathcal{C}_k(a_k)} w(\boldsymbol{a})$$

where $w(\boldsymbol{a}) = \sum_k i_k(a_k)$ is a componentwise sum of input weights $i_k(a_k)$.

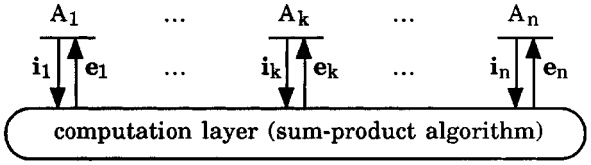

Fig. 20. Sum-product algorithm realized by an I/O layer and a computation layer.

If the weights are negative log likelihoods, $i_k(a_k) = -\log p(y_k|a_k)$, then $w(\boldsymbol{a}) = -\log p(\boldsymbol{y}|\boldsymbol{a})$, and the min-sum algorithm becomes a per-symbol *maximum-likelihood* (ML) decoding algorithm.

In summary, we have the following theorem.

Theorem 5.2: Given a finite, cycle-free normal graph of diameter $d_{\max}$ and weight vectors $\boldsymbol{w}_k$ for each symbol variable A_k, the sum-product algorithm computes the sum of the weights of all codewords that are consistent with all values of all state and symbol variables in $d_{\max}$ clock cycles. The weights and the computations may be in any commutative semiring. One vector (message) of size $|\mathcal{S}_j|$ is computed and sent each way along the jth edge for each $j \in I_{\mathcal{S}}{}'$. The number of computations at the ith vertex is of the order of $\delta_i(\delta_i - 2)|\mathcal{C}_i|$, where δ_i is the vertex degree.

From Theorem 5.2, we see that the state complexity $|\mathcal{S}_j|$ is a measure of the communications complexity, or "bandwidth," of the jth edge. The computational complexity at the ith node is proportional to the constraint complexity $|\mathcal{C}_i|$, or more precisely to $\delta_i(\delta_i - 2)|\mathcal{C}_i|$.

B. The Sum-Product Algorithm on Graphs with Cycles

On a graph with cycles, because the sum-product update rule is local, it may still be used in an iterative decoding algorithm. One must then specify an initialization rule, a schedule, and a stopping rule. Optimality is not in general guaranteed, nor even convergence. However, such approximate iterative sum-product algorithms often work very well, and indeed are commonly used to decode capacity-approaching codes such as turbo codes and LDPC codes.

Alternatively, one may develop a more elaborate exact algorithm by generalizing the sum-product algorithm and its update rule to more general cut sets.

For example, for any graph with cycles, one can find a configuration of cut sets starting with the empty set "downstream" and ending with the empty set "upstream," such that each successive cut set moves one vertex from the "downstream" to the "upstream" side of the cut—i.e., one can sweep a cut-set boundary through the graph, crossing one vertex at a time. The sum-product update rule may then be generalized to a rule for computing weight vectors $\boldsymbol{w}_{\chi\uparrow}$ in terms of previously computed "upstream" weight vectors $\boldsymbol{w}_{\chi'\uparrow\uparrow}$; the resulting algorithm will be exact. (This idea is closely related to the "junction tree" method for factor graphs [1].)

However, such a sweep establishes a cycle-free graph structure on the symbol index set $I_{\mathcal{A}}$. The sizes of the state spaces will be at least equal to the sizes of the state spaces in some conventional state realization (trellis), according to the Cut-Set

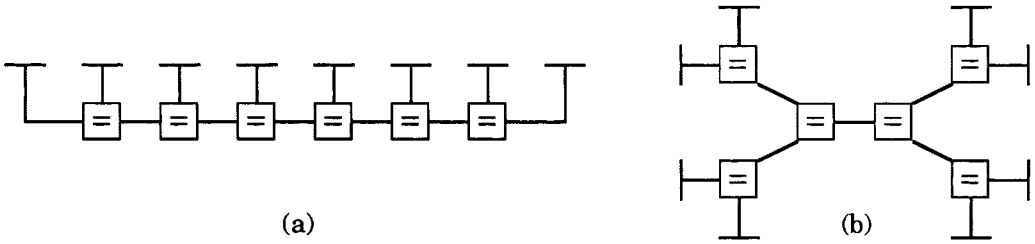

Fig. 21. Realizations of eight-symbol repetition constraint. (a) Chain (trellis). (b) Tree.

Bound, so the complexity advantage of a realization on a graph with cycles will be lost.

We conjecture that *any* exact decoding algorithm for a code $\mathcal{C}$ must be at least as complex as the sum-product algorithm applied to some cycle-free normal graph for $\mathcal{C}$ (or its dual; see Section IX). Therefore, in general it will be more attractive to use lower complexity graph representations with cycles and approximate iterative decoding, provided that performance is not too poor.

C. Deterministic and Erased Variables

A variable S_j for which only one of the upstream weights $w_{j\uparrow}(s_j)$ is nonzero will be called (upstream) *deterministic*, since $\boldsymbol{w}_{j\uparrow}$ is equivalent up to scaling to the probability vector of a deterministic random variable. If $\boldsymbol{w}_{j\uparrow}$ has this property, then so does $\boldsymbol{w}$, independent of $\boldsymbol{w}_{j\downarrow}$ (provided that $w_{j\downarrow}(s_j) \neq 0$ for the s_j for which $w_{j\uparrow}(s_j) \neq 0$), so the downstream weight vector $\boldsymbol{w}_{j\downarrow}$ does not need to be computed.

If all variables in an information set of a local code $\mathcal{C}_i$ are upstream deterministic, then only one codeword has nonzero upstream weight, and the remaining variables in the complementary check set must also be upstream deterministic.

A variable S_j for which all of the upstream weights $w_{j\uparrow}(s_j)$ are equal will be called an (upstream) *erasure*, since $\boldsymbol{w}_{j\uparrow}$ is equivalent up to scaling to the probability vector of an equiprobable random variable. If $\boldsymbol{w}_{j\uparrow}$ has this property, then $\boldsymbol{w}$ is proportional up to scaling to $\boldsymbol{w}_{j\downarrow}$, $\boldsymbol{w} \equiv_\alpha \boldsymbol{w}_{j\downarrow}$, so up to scale equivalence $\boldsymbol{w}$ depends only on the downstream weight vector $\boldsymbol{w}_{j\downarrow}$.

If all variables in a check set of a local code $\mathcal{C}_i$ are upstream erasures, then all codewords have equal upstream weight, and the remaining variables in the complementary information set must also be upstream erasures.

Deterministic variables and erasures can thus make computations of the upstream or downstream components of certain weight vectors unnecessary. In this sense, they can effectively convert a undirected graph into a partially directed graph. This may have the effect of eliminating some or all cycles in the original graph, and thus may allow exact decoding of a graph which is not cycle-free. For example, these properties may allow us to use a graph with cycles to realize an encoder I/O system, as we will see in Section VI.

For decoding of graphs with cycles, it may therefore be useful to let an intelligent supervisor make tentative "hard decisions" or "erasures" on certain variables, perhaps in various configurations, to speed convergence. Making tentative hard decisions is called "pinning" in [27].

D. Functional Separation

Normal graphs yield a clean separation of functions in sum-product decoding.

All computations take place at vertices (local codes). The function of ordinary edges (states) is purely communications (message passing). The function of leaf edges (symbols) is purely input of weight vectors $\boldsymbol{i}_k = \boldsymbol{w}_{k\uparrow}$ ("intrinsic information") from observation of symbols A_k, and ultimately output of weight vectors $\boldsymbol{e}_k = \boldsymbol{w}_{k\downarrow}$ ("extrinsic information") computed from the rest of the graph. In integrated circuit terminology, the local codes, state spaces, and symbols represent logic, interconnect, and I/O, respectively.

Conceptually, one can think of the vertices and ordinary edges as a "computation layer," connected by the leaf edges to an "I/O layer." Given a set of input weight vectors $\boldsymbol{i}_k$ at the I/O layer, the computation layer returns a set of weight vectors $\boldsymbol{e}_k$ for each symbol that have been computed by the sum-product algorithm. This viewpoint is illustrated in Fig. 20.

E. Decoding Graph Fragments

Another nice feature of the sum-product algorithm is that it can be executed on any fragment of a graph. Decoding a graph fragment can then be viewed as a lower level "subroutine" that may be called during the decoding of the original graph. This leads to recursive implementations.

In particular, each local code $\mathcal{C}_i$ may be realized by a cycle-free state realization, which may then be decoded exactly by the sum-product algorithm.

For example, if $\mathcal{C}_i$ is an n-symbol repetition constraint (i.e., in the linear case, an $(n, 1, n)$ repetition code), then $\mathcal{C}_i$ may be represented by the conventional state realization of a repetition code, which is essentially a chain of $n-2$ three-symbol repetition codes, as shown in Fig. 21(a). This will be simpler to decode than one n-symbol code [1]. Even better, $\mathcal{C}_i$ may be represented by a binary tree of $n-2$ three-symbol repetition codes, as shown in Fig. 21(b). Compared to the chain, the tree realization has the same space complexity, but less time complexity. Similarly efficient realizations may be used for decoding n-symbol SPC $(n, n-1, 2)$ codes.

In Section IV-C, we will show how to decode graph fragments using dual graphs.

VI. Normal Realizations of Reed–Muller Codes

At this point we pause to present a family of normal realizations for Reed–Muller (RM) codes. RM codes are known to have rather simple conventional state realizations (trellises), and their minimal state complexity profiles are known [13], [44].

These realizations are quite efficient: they have space complexity of the order of $n \log_2 n$ and time complexity of the order of $\log_2 n$. They use the simplest possible components: binary state variables, and $(3, 2, 2)$ and $(3, 1, 3)$ local codes. However, they do have cycles, which degrade decoding performance. They are nonetheless useful for encoding.

If cycles are removed by clustering, then we obtain somewhat more complicated cycle-free realizations that meet the Cut-Set Bound everywhere, and are slightly simpler than the standard trellis realizations of RM codes. They turn out to specify the ML decoding algorithms of [13].

A. Definition of RM Codes Via Hadamard Transforms

The basic building block of RM codes is the 2×2 Hadamard transform over the binary field $\mathbb{F}_2$, defined by the 2×2 matrix

$$H_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Given a binary 2-tuple $\boldsymbol{a} = (a_0, a_1)$, its Hadamard transform is the binary 2-tuple $\boldsymbol{A} = \boldsymbol{a}H_2$. In detail, $\boldsymbol{A} = (A_0, A_1) = (a_0, a_0+a_1)$. Note that $(H_2)^2 = I_2$, so the Hadamard transform of $\boldsymbol{A} = \boldsymbol{a}H_2$ is $\boldsymbol{a} = \boldsymbol{A}H_2$.

The $2^m \times 2^m$ Hadamard transform is defined by the m-fold tensor product $H_{2^m} = (H_2)^{\otimes m}$. For example

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The weights of the rows of H_{2^m} are powers of 2, and the number of rows of weight 2^r is the combinatorial coefficient $\binom{m}{r}$. Again, H_{2^m} is self-inverse.

The RM code RM (r, m) of length 2^m and minimum distance $d = 2^{m-r}$ is defined as the code generated by the rows of H_{2^m} of weight d or greater. In other words, the codewords are the words $\boldsymbol{a} = \boldsymbol{A}H_{2^m}$, where the components of the information vector $\boldsymbol{A}$ corresponding to rows of weight less than d are zero, and the remaining components are free.

B. Normal Realizations of RM Encoders

A normal realization of an I/O realization of RM(r, m) (an encoder for RM(r, m)) may therefore be obtained from a normal realization of the Hadamard transform relation $\boldsymbol{A} = \boldsymbol{a}H_{2^m}$, which in turn may be based on realizations of the elementary 2×2 transform $(A_0, A_1) = (a_0, a_0 + a_1)$.

Such an elementary realization is shown in Fig. 22. Here there is one binary state variable s, and two constraints

$$s = a_0 = A_0$$
$$s + a_1 + A_1 = 0.$$

Note that there are no arrows in this "behavioral" realization. Both $\boldsymbol{a}$ and $\boldsymbol{A}$ are information sets; either may be taken as the "input," and then its Hadamard transform ($\boldsymbol{A}$ or $\boldsymbol{a}$, respectively) is the "output." (Interestingly, this is the controlled-NOT gate, the standard two-qubit gate of quantum computation [3].)

A realization of the relation $\boldsymbol{A} = \boldsymbol{a}H_{2^m}$ may now be obtained by connecting $m2^m$ such elementary realizations in tensor product fashion, as illustrated for $m = 3$ in Fig. 23. Again, this yields a "fast" realization of either the 8×8 Hadamard transform or its inverse.

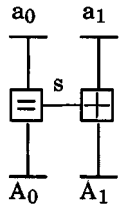

Fig. 22. Realization of $(A_0, A_1) = (a_0, a_0 + a_1)$.

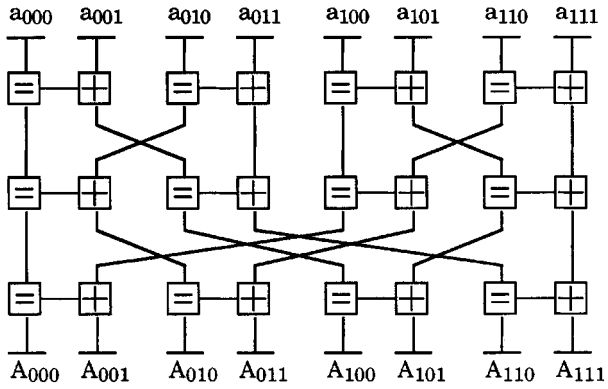

Fig. 23. Realization of $\boldsymbol{A} + \boldsymbol{a}H_8$.

There are three nontrivial RM codes of length 8, which are obtained by fixing certain of the transform components to zero and letting the others range freely. With these assignments, Fig. 23 becomes an encoder for any of these codes.

i) RM$(0, 3) = (8, 1, 8)$: A_{000} free; all others set to zero.
ii) RM$(1, 3) = (8, 4, 4)$: $A_{111} = A_{110} = A_{101} = A_{011} = 0$; all others free.
iii) RM$(2, 3) = (8, 7, 2)$: $A_{111} = 0$; all others free.

C. Normal Realizations of RM Codes

In the realizations above, both $\boldsymbol{a}$ and $\boldsymbol{A}$ represent visible symbols. In other words, Fig. 23 is a realization of the full I/O behavior $\mathfrak{B} = \{(\boldsymbol{a}, \boldsymbol{A}) | \boldsymbol{a} = \boldsymbol{A}H_{2^m}\}$. Particularly for decoding, we desire a realization of the code $\mathcal{C} = \mathfrak{B}_{|\mathcal{A}}$ in which only the code symbols $\boldsymbol{a}$ are visible, while the "input symbols" $\boldsymbol{A}$ become latent (driving) state variables.

The components of $\boldsymbol{A}$ should then be regarded as state edges connected to degree-1 local constraints, which restrict them either to be zero or to be a free element of $\mathbb{F}_2$. In Fig. 24, degree-1 constraint vertices labeled by "0" and "F" indicate zero and free constraints, respectively. We may then use the following "reduction rules," illustrated in Fig. 24.

a) If $A_0 = A_1 = 0$, then $a_0 = a_1 = 0$.
b) If $A_1 = 0$, then $a_0 = a_1 = A_0$.
c) If A_0 is free and $A_1 = 0$, then $a_0 = a_1$.
d) If A_0 is free, then $a_0 + a_1 = A_1$.
e) If A_0 and A_1 are free, then a_0 and a_1 are free.

Using these reduction rules, we obtain the reduced realizations shown in Figs. 25–27.

We note that the realizations of the $(8, 1, 8)$ and $(8, 7, 2)$ codes are the minimal binary-tree realizations of repetition and

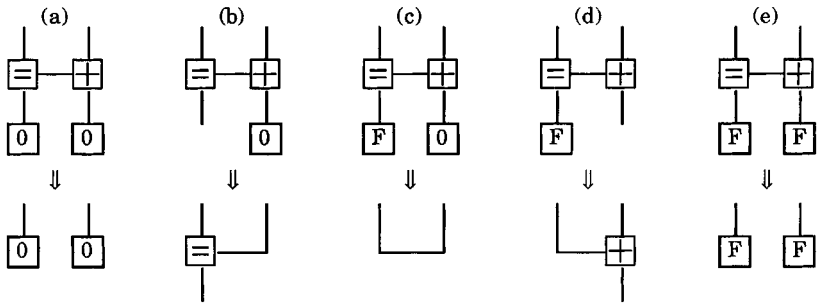

Fig. 24. Reduction rules to minimize elementary realizations.

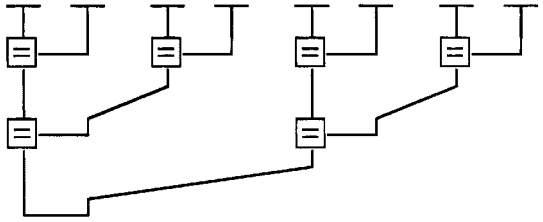

Fig. 25. Reduced realization of $(8, 1, 8)$ code.

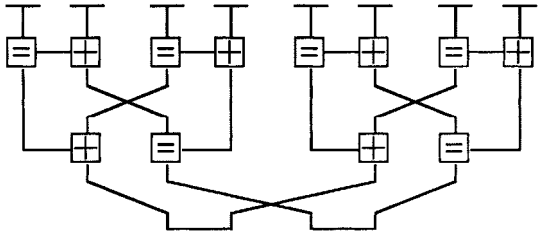

Fig. 26. Reduced realization of $(8, 4, 4)$ code.

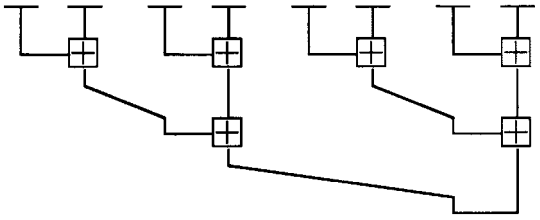

Fig. 27. Reduced realization of $(8, 7, 2)$ code.

SPC codes that were illustrated in Fig. 21. They each involve only six $(= 2^m - 2)$ degree-three, binary-valued constraints of parity-check or repetition type, respectively. These graphs are cycle-free, and their diameter is only $4\ (= 2m - 2)$.

The realization of the $(8, 4, 4)$ code is novel, as far as we know. It involves only 12 degree-three, binary-valued constraints of parity-check or repetition type, which is likely to be the most efficient possible for this code. The diameter is only 4. However, the graph is not cycle-free. The minimum cycle girth is 6. Simulations of iterative sum-product decoding on this graph have shown performance about 0.2 dB worse than exact ML decoding [9].

Fig. 28 shows a reduced realization for the $(16, 5, 8)$ RM code. It involves 34 degree-three, binary-valued constraints; its diameter is 6; and the minimum girth of any cycle is 7. Decoding simulations on this graph have shown performance about 0.4 dB worse than ML decoding [9].

Similar reduced realizations may be developed for all RM codes. These representations are quite efficient: for a code of length $n = 2^m$, the number of elementary realizations needed ("space complexity") is upper-bounded by $n \log_2 n$, and the diameter ("time complexity") is upper-bounded by $2 \log_2 n$.

However, except for SPC and repetition codes, these realizations have cycles, whose girth is lower-bounded by 6. We therefore do not have good bounds on the number of iterations needed or on performance with the sum-product decoding algorithm. Performance deteriorates as the codes become more complicated; e.g., decoding simulations of the reduced graph of the $(32, 16, 8)$ code have shown performance about 1.0 dB worse than ML decoding [9].

Although they have cycles, these realizations may nonetheless be used for encoding. Fig. 29 shows, for example, how the $(8, 4, 4)$ realization can implement an encoder with information set $\{a_{000}, a_{001}, a_{010}, a_{100}\}$ and check set $\{a_{011}, a_{101}, a_{110}, a_{111}\}$ by edge directing, as explained in Section V-C. This technique works for any RM realization of this type and any information set.

D. Cycle-Free Realizations Via Clustering

We now show how these realizations may be made cycle-free by clustering ("sectionalization").

Consider the realization of the $(8, 4, 4)$ code in Fig. 26. We may obtain an alternative quaternary realization for the $(8, 4, 4)$ code by clustering as shown in Fig. 30.

If we cluster pairs of bits, then the top-row clusters are inconsequential, because one incident quaternary variable is just a permutation of the other. No computation occurs at this level, just a pass-through of weights with relabeling according to this permutation.

At the next level, the two clusters may each be regarded as a realization of a certain simple self-dual $(6, 3)$ binary linear block code.

The result is the realization of Fig. 31. The four symbols and the single state are all 4-valued. The two constraints are identical and are defined by a certain $(6, 3)$ binary code. The realization is cycle-free, and has maximum constraint (branch) complexity 8. Indeed, this is the minimal conventional sectionalized four-state realization (trellis) for this code (see Fig. 7).

Similarly, we obtain the cycle-free realization for the $(16, 5, 8)$ code shown in Fig. 32. Here the 4-bit sections represent $(4, 3)$ codes, which in turn are represented in "divide-by-2" fashion. (We omit the inconsequential $(4, 2)$ codes at the

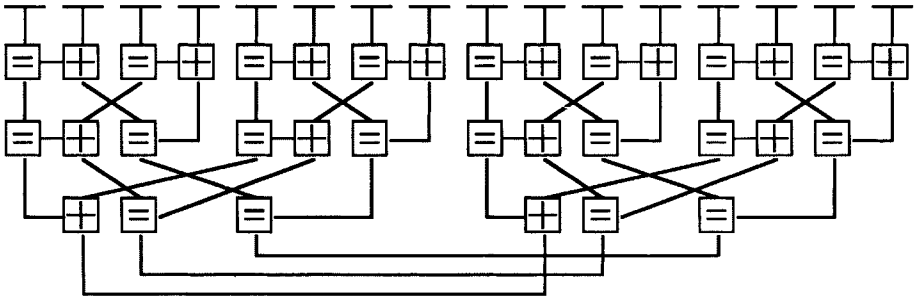

Fig. 28. Reduced realization of $(16, 5, 8)$ code.

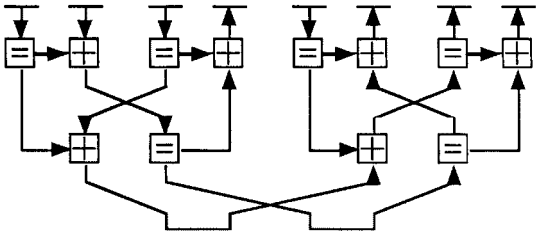

Fig. 29. Encoder for $(8, 4, 4)$ code.

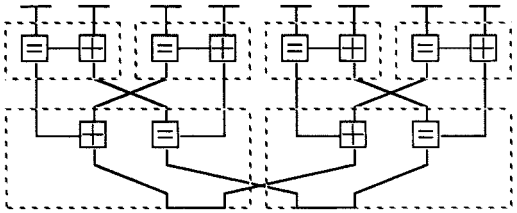

Fig. 30. Clustering of $(8, 4, 4)$ realization.

2-bit section level.) Each of the two halves (8-bit sections) is constrained to lie in the $(8, 4, 4)$ code. The 3-bit state variable represents a coset of $(8, 1, 8)$ in $(8, 4, 4)$ [13]. The parameters of each constraint code are as shown.

The Cut-Set Bound is met for all possible cut sets in this realization. Decoding is slightly faster than for the standard minimal four-section, eight-state trellis for this code because of the tree-like "divide-by-2" structure. The dominant constraint codes have complexity 16, equal to the maximum branch complexity in the minimal trellis for this code.

Similarly, by grouping 4-tuples of bits, we obtain a cycle-free "four-section, 64-state" realization for the $(32, 16, 8)$ code that is shown in Fig. 33. A closely related "three-section, 64-state" realization for the $(24, 12, 8)$ Golay code is shown in Fig. 34. In each case, the dominant constraint complexity is 512, as in the minimal trellis for these codes. Again, the Cut-Set Bound is met everywhere. Note how the $(24, 12, 8)$ realization cleverly avoids any cuts that would partition the symbols into two equal parts, which would require a state complexity of 256.

It is interesting to notice these realizations specify precisely the "divide-by-2" ML decoding algorithms that were proposed in [13] to decode RM codes. These algorithms are slightly more efficient than Viterbi algorithm decoding of the minimal trellis.

Future work might explore whether there exists some intermediate clustering technique that reduces but does not eliminate cycles, and that obtains near-ML decoding performance.

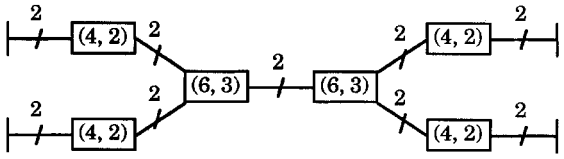

Fig. 31. Minimal sectionalized realization (trellis) for $(8, 4, 4)$ code.

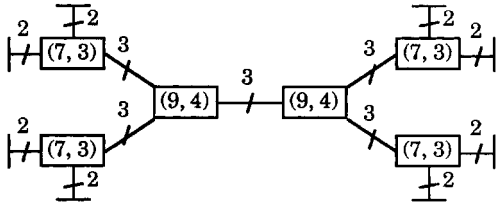

Fig. 32. Minimal sectionalized realization for $(16, 5, 8)$ code.

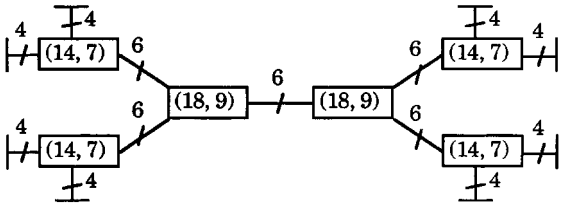

Fig. 33. Minimal sectionalized realization for $(32, 16, 8)$ code.

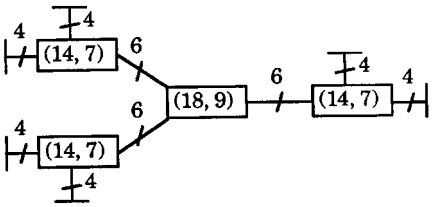

Fig. 34. Minimal sectionalized realization for $(24, 12, 8)$ Golay code.

VII. Dual Codes and Realizations

In this section we develop our most fundamental result: dual normal group realizations (suitably defined) generate dual group codes. This result depends on precisely the degree restrictions that characterize normal realizations, and has as corollaries some of the most basic dynamical duality results for linear or group codes and realizations.

Our notion of duality is Pontrayagin (character group) duality. Pontryagin duality theory applies to locally compact Abelian (LCA) groups such as finite Abelian groups and real Euclidean n-space $\mathbb{R}^n$, and underlies all Fourier analysis [35], [40], [23],

[14]. For linear codes, it reduces to the usual notions of duality in terms of orthogonality.

We will briefly recapitulate some definitions and results from the duality theory of group codes, which extends that of linear codes [17], [18], [15]. Under Pontryagin duality, a group code $\mathcal{C}$ has a well-defined dual code $\mathcal{C}^\perp$, which is the usual orthogonal code when $\mathcal{C}$ is actually linear over a finite field $\mathbb{F}$ or over $\mathbb{R}$.

Our objective is to define a dual to a normal group realization of $\mathcal{C}$ such that the dual realization necessarily generates the dual code $\mathcal{C}^\perp$.

A. Pontryagin Duality

We discuss a number of basic dualities of Pontryagin duality theory: character group duality, orthogonal subgroup duality, direct product duality of two kinds, quotient group duality, and sum/intersection duality.

The *character group* $G\hat{}$ of an LCA group G is the group under composition of all continuous homomorphisms h: $G \to \mathcal{C}$ from G into the complex unit circle S^1. The fundamental Pontryagin duality theorem is that $G\hat{}$ is LCA and that the character group of $G\hat{}$ may be taken as G: $G\hat{}\hat{} = G$. Indeed, if one defines a "pairing" $\langle\, , \rangle$: $G\hat{} \times G \to S^1$ by $\langle h, g\rangle = h(g)$, then the characters g: $G\hat{} \to S^1$ of $G\hat{}$ are precisely the characters defined by $g(h) = \langle h, g\rangle$.

If G is isomorphic to a finite cyclic group $\mathbb{Z}_m$, then its character group $G\hat{}$ is isomorphic to $\mathbb{Z}_m$, and the pairing $\langle h, g\rangle$ may be written explicitly as $\langle h, g\rangle = \omega^{gh}$, where $\omega = e^{2\pi i/m}$ is a complex mth root of unity, both G and $G\hat{}$ are identified with $\mathbb{Z}_m$, and the product gh is taken in $\mathbb{Z}_m$. Similarly, if $G = \mathbb{R}$, then its character group $G\hat{}$ is isomorphic to $\mathbb{R}$, and the pairing $\langle h, g\rangle$ may be written explicitly as the Fourier kernel $\langle h, g\rangle = e^{-2\pi igh}$.

In these two cases, G and $G\hat{}$ are isomorphic; however, this is not true in general. For example, by quotient group duality (see below), the character group of the integers $\mathbb{Z}$ is the torus $\mathbb{R}/\mathbb{Z}$. In general, the character group of a discrete group is compact and *vice versa*, so the character group of a discrete group is discrete if and only if it is also compact, i.e., finite.

If $G = G_1 \times \cdots \times G_n$ is the direct product of a finite collection of LCA groups $\{G_i,\, 1 \le i \le n\}$, then its character group is the direct product $G\hat{} = G_1\hat{} \times \cdots \times G_n\hat{}$ of the corresponding character groups $\{G_i\hat{},\, 1 \le i \le n\}$. Each group element may be written as $\boldsymbol{g} = (g_1, \ldots, g_n)$, each character may be written as $\boldsymbol{h} = (h_1, \ldots, h_n)$, and the pairing $\langle \boldsymbol{h}, \boldsymbol{g}\rangle$ may be written componentwise as

$$\langle \boldsymbol{h}, \boldsymbol{g}\rangle = \prod_i \langle h_i, g_i\rangle.$$

Since every finite Abelian group is a finite direct product $G = G_1 \times \cdots \times G_n$ of finite cyclic groups G_i, it follows that its character group $G\hat{} = G_1\hat{} \times \cdots \times G_n\hat{}$ is isomorphic to itself. Similarly, the character group of $\mathbb{R}^n$ is isomorphic to $\mathbb{R}^n$.

If G and $G\hat{}$ are dual character groups, then $g \in G$ and $h \in G\hat{}$ are said to be *orthogonal* if $\langle h, g\rangle = 1$ (the identity of $\mathcal{C}$). If S is a subgroup of G, then the *orthogonal subgroup* (annihilator) of S is the set $S^\perp$ of all $h \in G\hat{}$ that are orthogonal to all $g \in S$. The second basic Pontryagin duality theorem is that $S^\perp$ is a closed subgroup of $G\hat{}$, and that $S^{\perp\perp}$ is the closure $\overline{S}$ of S in G.

We will therefore consider only closed subgroups in this paper. In particular, we will refine our definition of a group code $\mathcal{C} \subseteq \mathcal{A}$ to require $\mathcal{C}$ to be a *closed* subgroup of the direct product group $\mathcal{A}$ in its natural topology. In the case of finite Abelian block codes, the topology of $\mathcal{A}$ is discrete, and all subgroups are closed.

With this condition, the orthogonal subgroup of $S^\perp$ is S: $S^{\perp\perp} = S$. Thus, a closed subgroup $S \subseteq G$ is completely characterized by its orthogonal subgroup $S^\perp \subseteq G\hat{}$.

If $S = S_1 \times \cdots \times S_n$ is the direct product of a finite collection of subgroups $\{S_i \subseteq G,\, 1 \le i \le n\}$, then its orthogonal subgroup is the direct product $S^\perp = (S_1)^\perp \times \cdots \times (S_n)^\perp$ of the corresponding orthogonal groups $\{(S_i)^\perp \subseteq G\hat{},\, 1 \le i \le n\}$.

Another elementary duality result is that if $S \subseteq G$ and $S^\perp \subseteq G\hat{}$ are orthogonal subgroups, then $S^\perp$ may be identified with the character group of the quotient group G/S, so $S^\perp$ is isomorphic to G/S. In the finite Abelian case, this implies $|S||S^\perp| = |G|$.

More generally, if $T \subseteq S \subseteq G$, then $S^\perp \subseteq T^\perp \subseteq G\hat{}$, and $T^\perp/S^\perp$ is isomorphic to S/T. We call this *quotient group duality*. In the finite Abelian case, this implies $|S||S^\perp| = |T||T^\perp|$.

Finally, if S and T are subgroups of G and $S^\perp$ and $T^\perp$ are the orthogonal subgroups in $G\hat{}$, then the sum $S + T$ (the smallest closed subgroup containing both S and T) and the intersection $S^\perp \cap T^\perp$ are orthogonal subgroups, as are $S \cap T$ and $S^\perp + T^\perp$.

If all groups are p-ary groups (i.e., isomorphic to $(\mathbb{Z}_p)^n$ for some prime p), then all groups may be regarded as vector spaces over the p-ary field $\mathbb{F}_p$, and all subgroups as linear subspaces. The group definitions of orthogonality and duality then reduce to the standard definitions for linear vector spaces over $\mathbb{F}_p$.

B. Dual Configuration Spaces and Dual Codes

If $\mathcal{A} = \prod_{I_\mathcal{A}} \mathcal{A}_k$ is a group symbol configuration space defined on a finite index set $I_\mathcal{A}$ with LCA symbol alphabets $\{\mathcal{A}_k,\, k \in I_\mathcal{A}\}$, then the *dual symbol configuration space* is defined as its character group $\mathcal{A}\hat{} = \prod_{I_\mathcal{A}} \mathcal{A}_k\hat{}$, where $\mathcal{A}_k\hat{}$ is the character group of $\mathcal{A}_k$.

If $\mathcal{C} \subseteq \mathcal{A}$ is a (closed) subgroup of $\mathcal{A}$, then the *dual group code* is defined as the orthogonal subgroup $\mathcal{C}^\perp \subseteq \mathcal{A}\hat{}$. In the finite Abelian case, we have $|\mathcal{C}||\mathcal{C}^\perp| = |\mathcal{A}|$ by quotient group duality. We will say that $\mathcal{C}$ is *high-rate* if $|\mathcal{C}| > |\mathcal{C}^\perp|$ and *low-rate* if $|\mathcal{C}| < |\mathcal{C}^\perp|$.

Note that duality of configuration spaces is defined in a character group sense, whereas duality of group codes is defined in an orthogonal subgroup sense.

Two simple but useful examples of dual group codes are the following.

1) If $\mathcal{C} = \mathcal{A}$ (the *universe code*), then $\mathcal{C}^\perp = \{\mathbf{0}\}$ (the *trivial code* containing only the all-zero codeword $\mathbf{0} \in \mathcal{A}\hat{}$). This follows since there is only one character orthogonal to all elements of a group G, which is called the *principal character* and denoted by $0 \in G\hat{}$.

2) If $\mathcal{A} = G^n$ and

$$\mathcal{C} = \{(g, g, \ldots, g) | g \in G\}$$

(the *repetition code* of length n over G), then $\mathcal{A}^\wedge = (G^\wedge)^n$ and

$$\mathcal{C}^\perp = \left\{ \boldsymbol{h} \in \mathcal{A}^\wedge \Big| \sum_k h_k = 0 \right\}$$

(the *zero-sum code* of length n over $G^\wedge$). This follows since for $\boldsymbol{g} \in \mathcal{C}$, by the bihomomorphic properties of characters

$$\langle \boldsymbol{h}, \boldsymbol{g} \rangle = \prod_{k=1}^{n} \langle h_k, g \rangle = \left\langle \sum_{k=1}^{n} h_k, g \right\rangle$$

which equals 1 for all $g \in G$ if and only if $\sum_k h_k$ is the principal character $0 \in G^\wedge$.

(*Note*: The technical difficulty with an infinite index set $I_\mathcal{A}$ is that the character group of an infinite direct product $\prod_{I_\mathcal{A}} A_k$, with an appropriate product topology, is the infinite direct sum $\bigoplus_{I_\mathcal{A}} A_k{}^\wedge$, namely the set of all elements in the direct product $\prod_{I_\mathcal{A}} A_k{}^\wedge$ that have only a finite number of nonzero components, with an appropriate sum topology [24]. Our duality theorems remain true in the infinite case if we impose a global finiteness restriction in either the primal or dual domain; however, such a global restriction cannot be depicted in graphical models.)

C. Conditioned Code Duality

We now state a new fundamental lemma from which will follow all of our duality results.

We first prove a special case of this lemma, called projection/cross-section duality. In this subsection, the symbol index set will be denoted simply by I, and will be partitioned into two disjoint subsets J, K. We write $I = J \cup K$, with $J \cap K = \emptyset$ understood implicitly. We recall that $\mathcal{C}_{|J} = \{\boldsymbol{a}_{|J} | \boldsymbol{a} \in \mathcal{C}\}$ denotes the projection of $\mathcal{C}$ onto J, and $\mathcal{C}_J = \{\boldsymbol{a}_{|J} | \boldsymbol{a} \in \mathcal{C}, \boldsymbol{a}_{|K} = \boldsymbol{0}_{|K}\}$ denotes the cross section of $\mathcal{C}$ in J.

Theorem 7.1 (Projection/Cross-Section Duality): If $\mathcal{C}$ and $\mathcal{C}^\perp$ are dual group codes defined on a partitioned index set $I = J \cup K$, then the cross section $\mathcal{C}_J$ and the projection $(\mathcal{C}^\perp)_{|J}$ are dual group codes.

Proof: Because pairings are defined componentwise, we have

$$\begin{aligned}\langle \boldsymbol{b}, \boldsymbol{a} \rangle &= \prod_I \langle b_k, a_k \rangle = \left(\prod_J \langle b_k, a_k \rangle \right) \left(\prod_K \langle b_k, a_k \rangle \right) \\ &= \langle \boldsymbol{b}_{|J}, \boldsymbol{a}_{|J} \rangle \langle \boldsymbol{b}_{|K}, \boldsymbol{a}_{|K} \rangle.\end{aligned}$$

We therefore have the following logical chain:

$$\begin{aligned}\boldsymbol{a}_{|J} \in \mathcal{C}_J &\iff (\boldsymbol{a}_{|J}, \boldsymbol{0}_{|K}) \in \mathcal{C} \\ &\iff (\boldsymbol{a}_{|J}, \boldsymbol{0}_{|K}) \perp \mathcal{C}^\perp \iff \boldsymbol{a}_{|J} \perp (\mathcal{C}^\perp)_{|J}\end{aligned}$$

where we have used the definition of the cross section $\mathcal{C}_J$, orthogonal subgroup duality, and the fact that

$$\langle \boldsymbol{b}, (\boldsymbol{a}_{|J}, \boldsymbol{0}_{|K}) \rangle = \langle \boldsymbol{b}_{|J}, \boldsymbol{a}_{|J} \rangle$$

respectively. □

Fig. 35. Normal graphs of $(\mathcal{C}|\mathcal{D})_{|J}$ and $(\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J}$.

Now let $\mathcal{C} \subseteq \mathcal{A}$ be a group code defined on I, and let $\mathcal{D} \subseteq \mathcal{A}_{|K} = \prod_K \mathcal{A}_k$ be a group code defined on K. Define the *conditioned code* $(\mathcal{C}|\mathcal{D})_{|J}$ to be the projection onto J of the set of all codewords in $\mathcal{C}$ whose projection onto K is in $\mathcal{D}$

$$(\mathcal{C}|\mathcal{D})_{|J} = \{\boldsymbol{a}_{|J} | \boldsymbol{a} \in \mathcal{C}, \boldsymbol{a}_{|K} \in \mathcal{D}\}.$$

It is clear that a conditioned code is a group code.

Note that we may write $(\mathcal{C}|\mathcal{D})_{|J}$ as either a projection or a cross section

$$(\mathcal{C}|\mathcal{D})_{|J} = (\mathcal{C} \cap (\mathcal{A}_{|J} \times \mathcal{D}))_{|J} = (\mathcal{C} + (\{\boldsymbol{0}\}_{|J} \times \mathcal{D}))_J$$

since, by definition, $\boldsymbol{a}_{|J} \in (\mathcal{C}|\mathcal{D})_{|J}$ if and only if there is some $\boldsymbol{a}_{|K} \in \mathcal{D}$ such that $(\boldsymbol{a}_{|J}, \boldsymbol{a}_{|K}) \in \mathcal{C}$.

For example, the projection $\mathcal{C}_{|J}$ is equal to the conditioned code $(\mathcal{C}|\mathcal{A}_{|K})_{|J}$, and the cross section $\mathcal{C}_J$ is equal to the conditioned code $(\mathcal{C}|\{\boldsymbol{0}\}_{|K})_{|J}$.

Our fundamental lemma is then as follows.

Theorem 7.2 (Conditioned Code Duality): If $\mathcal{C}$ and $\mathcal{C}^\perp$ are dual group codes defined on a partitioned index set $I = J \cup K$, and $\mathcal{D}$ and $\mathcal{D}^\perp$ are dual group codes defined on K, then the conditioned codes $(\mathcal{C}|\mathcal{D})_{|J}$ and $(\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J}$ are dual group codes.

Proof: We have the following duality chain:

$$\begin{aligned}((\mathcal{C}|\mathcal{D})_{|J})^\perp &= ((\mathcal{C} + (\{\boldsymbol{0}\}_{|J} \times \mathcal{D}))_J)^\perp \\ &= ((\mathcal{C} + (\{\boldsymbol{0}\}_{|J} \times \mathcal{D}))^\perp)_{|J} \\ &= (\mathcal{C}^\perp \cap (\{\boldsymbol{0}\}_{|J} \times \mathcal{D})^\perp)_{|J} \\ &= (\mathcal{C}^\perp \cap ((\mathcal{A}^\wedge)_{|J} \times \mathcal{D}^\perp))_{|J} \\ &= (\mathcal{C}^\perp | \mathcal{D}^\perp)_{|J}\end{aligned}$$

where we have used one definition for $(\mathcal{C}|\mathcal{D})_{|J}$, projection/cross-section duality, sum/intersection duality, direct product duality (with $(\{\boldsymbol{0}\}_{|J})^\perp = (\mathcal{A}^\wedge)_{|J}$), and, finally, a second definition for $(\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J}$.

Fig. 35 illustrates the dual conditioned codes $(\mathcal{C}|\mathcal{D})_{|J}$ and $(\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J}$ by normal graphs. These graphs are examples of the dual graphs that will be defined in the next subsection.

D. Dual Realizations and Dual Graphs

Let a normal realization for $\mathcal{C}$ be given with symbol variables $\{A_k, k \in I_\mathcal{A}\}$, state variables $\{S_j, j \in I_\mathcal{S}\}$, and local codes $\{\mathcal{C}_i, i \in I_\mathcal{C}\}$, where the index subsets of the symbol and state variables that are involved in the ith local code $\mathcal{C}_i$ are $I_\mathcal{A}(i)$ and $I_\mathcal{S}(i)$, respectively. We now aim to define a dual realization in such a way that the dual realization necessarily generates the dual code $\mathcal{C}^\perp$.

It is natural to attempt to define such a dual realization as follows.

1) The dual symbol alphabets are the duals (character groups) $\{(\mathcal{A}_k)^\wedge, k \in I_\mathcal{A}\}$ of the primal symbol alphabets.
2) The dual state alphabets are the duals (character groups) $\{(\mathcal{S}_j)^\wedge, j \in I_\mathcal{S}\}$ of the primal state alphabets.

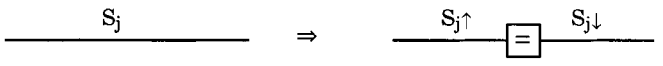

Fig. 36. Modifying a state edge by inserting a repetition vertex.

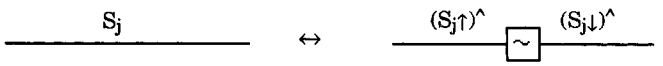

Fig. 37. Dualizing a state edge; a sign inverter appears in the middle of the dual edge.

3) The dual local codes are the duals (orthogonal subgroups) $\{(\mathcal{C}_i)^\perp, i \in I_\mathcal{C}\}$ of the primal local codes, where the index subsets of the symbols and state spaces that are involved in the ith local code $(\mathcal{C}_i)^\perp$ are again $I_\mathcal{A}(i)$ and $I_\mathcal{S}(i)$, respectively.

This realization is evidently normal. Its graph has the same topology as the graph of the primal realization. The alphabets of the variables are replaced by their character groups, and the local codes labeling the vertices are replaced by their dual (orthogonal) codes. We recall that the character group of a finite Abelian group or of a vector space over a finite field is isomorphic to the original alphabet.

Unfortunately, this realization does not always generate the dual code $\mathcal{C}^\perp$ (except when all groups are binary). We also need a trick that was used by Mittelholzer [33] to construct dual conventional state realizations (trellises): invert the sign of each dual state space $(\mathcal{S}_j)^\wedge$ in precisely one of its two appearances in the dual local codes $\mathcal{C}_i^\perp$. The mathematical necessity for this sign inversion will become apparent below.

To describe this trick precisely, let us replace each state variable S_j by two replicas $S_{j\uparrow}$ and $S_{j\downarrow}$ with the same alphabet $\mathcal{S}_j$ in the two local codes in which it is involved, where the choice between $S_{j\uparrow}$ and $S_{j\downarrow}$ may be made arbitrarily, and then require that the two replicas be equal. The original realization is thus modified as follows.

1) The symbol variables are still $\{A_k, k \in I_\mathcal{A}\}$.
2) The state variables are now $\{S_{j\uparrow}, j \in I_\mathcal{S}\}$ and $\{S_{j\downarrow}, j \in I_\mathcal{S}\}$.
3) The local codes now include both the original local codes $\{\mathcal{C}_i, i \in I_\mathcal{C}\}$ and also length-2 repetition codes $\{\mathcal{C}_j, j \in I_\mathcal{S}\}$, which impose the constraints $\{s_{j\uparrow} = s_{j\downarrow}, j \in I_\mathcal{S}\}$. Each state variable $S_{j\uparrow}$ or $S_{j\downarrow}$ is now involved in one local code $\mathcal{C}_i$ and one repetition code $\mathcal{C}_j$.

Graphically, this amounts to replacing each state edge S_j by the concatenation of a new state edge $S_{j\uparrow}$, a degree-2 repetition constraint $\mathcal{C}_j$: $s_{j\uparrow} = s_{j\downarrow}$, and a new state edge $S_{j\downarrow}$, as shown in Fig. 36. The resulting realization is still normal and still generates the same code, although it now has a redundant repetition vertex in the middle of each edge.

The *dual realization* is now defined as follows.

1) The dual symbol alphabets are the duals $\{(\mathcal{A}_k)^\wedge, k \in I_\mathcal{A}\}$ of the primal symbol alphabets.
2) The dual state variables are the duals

$$\{(\mathcal{S}_{j\uparrow})^\wedge = (\mathcal{S}_j)^\wedge, j \in I_\mathcal{S}\}$$

and

$$\{(\mathcal{S}_{j\downarrow})^\wedge = (\mathcal{S}_j)^\wedge, j \in I_\mathcal{S}\}$$

of the primal state alphabets.

3) The dual local codes are the duals $\{(\mathcal{C}_i)^\perp, i \in I_\mathcal{C}\}$ and $\{(\mathcal{C}_j)^\perp, j \in I_\mathcal{S}\}$ of the primal local codes, respectively. Since the dual of a repetition code is a zero-sum code, the dual codes $(\mathcal{C}_j)^\perp$ impose the constraints $\{t_{j\uparrow} = -t_{j\downarrow}, j \in I_\mathcal{S}\}$, where $t_{j\uparrow}, t_{j\downarrow} \in (\mathcal{S}_j)^\wedge$.

In the graph of the dual realization, a state edge in the original realization is thus replaced by the concatenation of a state edge $S_{j\uparrow}$, a degree-2 zero-sum constraint $(\mathcal{C}_j)^\perp$: $t_{j\uparrow} = -t_{j\downarrow}$, and a state edge $S_{j\downarrow}$. We call such a constraint a *sign inverter*, and depict it as shown in Fig. 37.

The code generated by the modified original realization may be characterized as follows:

$$\begin{aligned}\mathcal{C} = \{&(\boldsymbol{a}, \boldsymbol{s}_{j\uparrow}, \boldsymbol{s}_{j\downarrow}) \\ &\in \mathcal{A}\times\mathcal{S}\times\mathcal{S} | (\boldsymbol{a}, \boldsymbol{s}_{j\uparrow}, \boldsymbol{s}_{j\downarrow})_{|I_\mathcal{A}(i)\cup I_\mathcal{S}\uparrow(i)\cup I_\mathcal{S}\downarrow(i)} \in \mathcal{C}_i, i \in I_\mathcal{C}; \\ &\boldsymbol{s}_{j\uparrow} = \boldsymbol{s}_{j\downarrow}, j \in I_\mathcal{S}\}.\end{aligned}$$

Observe that $\mathcal{C}$ has the form of a conditioned code $\mathcal{C} = (\mathcal{P}|\mathcal{R})_{|I_\mathcal{A}}$, where

- $\mathcal{P} \subseteq \mathcal{A}\times\mathcal{S}\times\mathcal{S}$ denotes the set of $(\boldsymbol{a}, \boldsymbol{s}_{j\uparrow}, \boldsymbol{s}_{j\downarrow})$ that satisfy all local code constraints $\mathcal{C}_i$;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the length-2 repetition code over $\mathcal{S}$ that enforces the equality $\boldsymbol{s}_{j\uparrow} = \boldsymbol{s}_{j\downarrow}$.

Moreover, because the codes $\mathcal{C}_i$ have no variables in common, all constraints are independent; i.e., $\mathcal{P}$ is simply the direct product

$$\mathcal{P} = \prod_{i\in I_\mathcal{C}} \mathcal{C}_i.$$

In other words, the code $\mathcal{P}$ is generated by the graph obtained by cutting every ordinary edge in the original graph for $\mathcal{C}$ into two half-edges, thus creating a completely disconnected graph in which every local code $\mathcal{C}_i$ and its associated half-edges lies in a distinct disconnected component. The relation $\mathcal{R}$ then glues the two half-edges of a cut edge back together via a repetition vertex, thus reconstructing a graph for $\mathcal{C} = (\mathcal{P}|\mathcal{R})_{|I_\mathcal{A}}$ that is equivalent to the original graph. This sequence of transformations is depicted in Fig. 38.

Similarly, the code $\mathcal{C}'$ generated by the dual realization has the form of a conditioned code $\mathcal{C}' = (\mathcal{P}'|\mathcal{R}')_{|I_\mathcal{A}}$, where

- $\mathcal{P}' \subseteq A^\wedge \times \mathcal{S}^\wedge \times \mathcal{S}^\wedge$ denotes the set of $(\boldsymbol{b}, \boldsymbol{t}_{j\uparrow}, \boldsymbol{t}_{j\downarrow})$ that satisfy all constraints $(\mathcal{C}_i)^\perp$;
- $\mathcal{R}' \subseteq \mathcal{S}^\wedge \times \mathcal{S}^\wedge$ is the length-2 zero-sum code over $\mathcal{S}^\wedge$ that enforces the equality $\boldsymbol{t}_{j\uparrow} = -\boldsymbol{t}_{j\downarrow}$.

Again, in $\mathcal{P}'$ the codes $(\mathcal{C}_i)^\perp$ have no variables in common, so $\mathcal{P}'$ is simply the direct product $\mathcal{P}' = \prod_{i\in I_\mathcal{C}} (\mathcal{C}_i)^\perp$. But by direct product duality, this is the dual of $\mathcal{P}$: $\mathcal{P}' = \mathcal{P}^\perp$. Similarly, $\mathcal{R}'$ is the dual of $\mathcal{R}$: $\mathcal{R}' = \mathcal{R}^\perp$. Thus, $\mathcal{C}' = (\mathcal{P}^\perp|\mathcal{R}^\perp)_{|I_\mathcal{A}}$, which, by conditioned code duality, implies that $\mathcal{C}' = \mathcal{C}^\perp$.

Therefore, we have proved our main result.

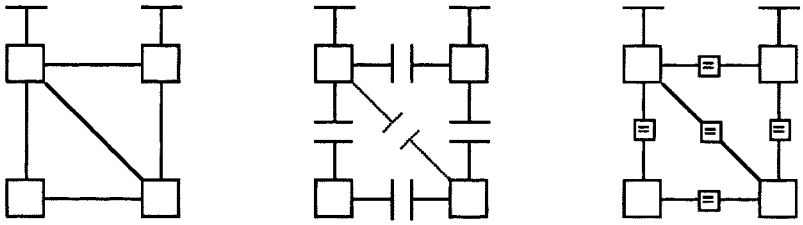

Fig. 38. Transformations. (a) A graph for $\mathcal{C}$. (b) A graph for $\mathcal{P}$. (c) A graph for $(\mathcal{P}|\mathcal{R})_{|I_{\mathcal{A}}}$.

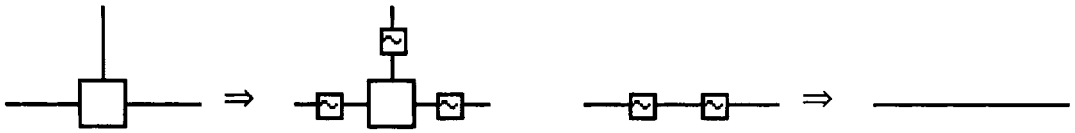

Fig. 39. Reduction rules for sign inverters.

Theorem 7.3 (Dual Realizations Generate Dual Codes): If $\mathcal{C}$ and $\mathcal{C}'$ are codes generated by dual normal realizations as defined above, then $\mathcal{C}$ and $\mathcal{C}'$ are dual codes: $\mathcal{C}' = \mathcal{C}^{\perp}$.

Note that the restrictions that symbol variables have degree 1 and state variables have degree 2 are essential to Theorem 7.3; i.e., it can hold only for normal realizations.

Theorem 7.3 is remarkable for its generality. In particular, it is one of the few general theorems known for codes generated by graphs with cycles.

If all groups are binary, so that $-g = g$, then Theorem 7.3 holds for our original attempt at a dual realization, since it differs from our ultimate dual realization only by sign inversions.

The graph of the dual realization is called the *dual graph.* It is the same as the graph of the original realization, except that alphabets of variables are replaced by their character groups (if necessary), codes are replaced by their dual codes, and sign inverters are inserted into edges as shown in Fig. 37 (if necessary). In the linear or finite Abelian cases, alphabets are unchanged, and in the binary case, inverters are unnecessary. Thus, given a normal graph realizing a code $\mathcal{C}$, a dual graph that realizes $\mathcal{C}^{\perp}$ may be obtained by trivial graphical manipulations.

The following rules may be used to minimize proliferation of sign inverters (see Fig. 39).

- Sign inverters may be inserted into every edge incident on any vertex; this has no effect since $\mathcal{C}_i = -\mathcal{C}_i$ for any Abelian group code $\mathcal{C}_i$.
- Two consecutive sign inverters are equivalent to a repetition vertex and may be suppressed.

VIII. Applications

Theorem 7.3 has innumerable applications. In this section we give some examples.

A. Dual Trellises, State Spaces, Tail-Biting Trellises, and RM Codes

Theorem 8.1 (Dual Minimal Cycle-Free Realizations): Let $\mathcal{C}$ and $\mathcal{C}^{\perp}$ be dual group codes. Let $\mathcal{C}$ have a minimal realization on a cycle-free graph. Then $\mathcal{C}^{\perp}$ has a minimal realization on the same graph, in which the state spaces are the character groups of the primal state spaces.

Proof: The existence of such a dual realization for $\mathcal{C}^{\perp}$ follows directly from Theorem 7.3. Moreover, each state space in the dual realization is necessarily as small as possible, else the dual of the dual realization would be a realization for $\mathcal{C}$ with smaller state spaces than a minimal realization on that graph, which is impossible. □

Again, we remind the reader that in the linear or finite Abelian cases, the character group of G is isomorphic to G, so that the same state spaces may be used in the dual realization.

Corollary 8.2 (Dual Trellises): (Mittelholzer [33]) Let $\mathcal{C}$ and $\mathcal{C}^{\perp}$ be dual group codes over finite Abelian groups. A minimal trellis for $\mathcal{C}^{\perp}$ is obtained by dualizing a minimal trellis for $\mathcal{C}$. Both minimal trellises have the same state complexity profile. The constraints (trellis sections) for $\mathcal{C}^{\perp}$ are the duals of the constraints for $\mathcal{C}$.

Proof: Specialize the previous theorem to trellises and finite Abelian groups. □

Corollary 8.3 (Dual State Spaces): Let $\mathcal{C}$ and $\mathcal{C}^{\perp}$ be dual group codes defined on I. Let $I = J \cup K$ be any two-way partition of the index set I, and let the corresponding $\{J, K\}$-induced state space of $\mathcal{C}$ be S_J. Then the $\{J, K\}$-induced state space of $\mathcal{C}^{\perp}$ is isomorphic to the character group of S_J.

Proof: Specialize the previous theorem to any cycle-free graph in which there exists an edge whose removal induces the given two-way partition. □

Theorem 8.3 is not difficult to prove directly using quotient group duality and projection/cross-section duality [18], [17], [15], but it is interesting that it falls out of this general development.

Theorem 8.4 (Dual Tail-Biting Realizations): Let $\mathcal{C}$ and $\mathcal{C}^{\perp}$ be dual group codes. Suppose that $\mathcal{C}$ has a tail-biting realization with state spaces S_j. Then $\mathcal{C}^{\perp}$ has a tail-biting realization with state spaces $(S_j)^{\wedge}$.

Proof: Follows directly from Theorem 7.3. □

Theorem 8.5 (Dual RM Codes):

$$\mathrm{RM}(r, m) = \mathrm{RM}(m - r - 1, m)^{\perp}.$$

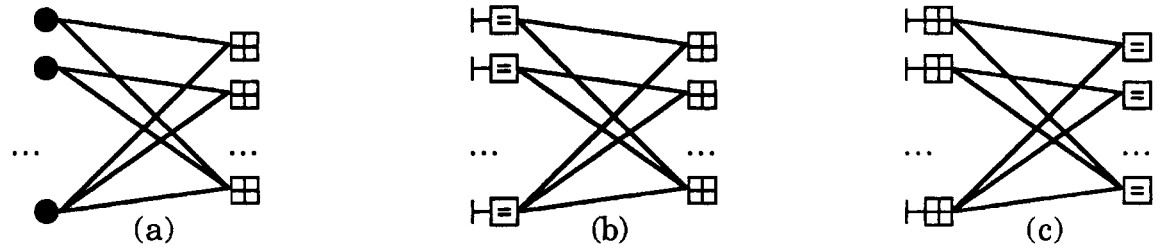

Fig. 40. (a) Tanner graph. (b) Normal Tanner graph. (c) Dual Tanner graph.

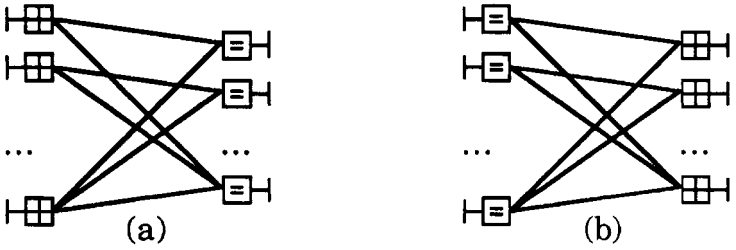

Fig. 41. Realizations of (a) encoder for $\mathcal{C}^\perp$ and (b) syndrome-former for $\mathcal{C}$.

Proof: Consider the realizations for RM(r, m) and RM$(m-r-1, m)$ given in Section VI. It can be seen by inspection that dualizing one (by interchanging repetition and zero-sum (parity-check) constraints, as well as free and zero inputs) gives the left–right mirror image of the other. Therefore, the two realizations are dual and must generate dual codes. □

B. Duals to Tanner Graphs, Encoders, and Syndrome Formers

A Tanner graph is a factor graph without state variables. An $r \times n$ parity-check matrix H^T for an $(n, n-r)$ binary linear block code $\mathcal{C}$ is a binary matrix such that

$$\mathcal{C} = \{\boldsymbol{a}H^T = \mathbf{0} | \boldsymbol{a} \in (\mathbb{F}_2)^n\}.$$

The parity-check matrix realization for $\mathcal{C}$ leads to a Tanner graph with n binary symbol variables, r SPC local codes representing the r rows (parity checks) of H^T, and edges corresponding to the nonzero entries of H^T, as illustrated in Fig. 40(a).

The corresponding normal graph is illustrated in Fig. 40(b). SPC codes are designated by a "+" symbol, while repetition codes are as usual designated by "=."

Since the dual of an SPC code is a repetition code and *vice versa*, the dual to this normal Tanner graph is the normal graph shown in Fig. 40(c), which we will call a *dual Tanner graph.* By regarding the binary state variables associated with each of the r repetition constraints as "inputs," we see that the dual Tanner graph generates the dual (n, r) binary linear block code $\mathcal{C}^\perp = \{\boldsymbol{u}H, \boldsymbol{u} \in (\mathbb{F}_2)^r\}$, whose generator matrix is H. The n SPC vertices serve to sum the contributions to each of the n code symbols from each of the r inputs.

Note that this dual "generator-matrix" realization cannot be represented by a Tanner graph.

In behavioral system theory, realizations such as those in Fig. 40(b) and (c) are called "kernel" and "image" representations, respectively. The next subsection will discuss the duality between kernels and images in a more general setting.

Two related realizations are illustrated in Fig. 41. If we make the input symbols of Fig. 40(c) visible, then we obtain a realization of an input/output system ("encoder") for $\mathcal{C}^\perp$:

$$(\mathcal{C}^\perp)_{\mathrm{I/O}} = \{(\boldsymbol{u}, \boldsymbol{u}H), \boldsymbol{u} \in (\mathbb{F}_2)^r\}.$$

The dual of this realization realizes an I/O system called a "syndrome-former" for $\mathcal{C}$:

$$(\mathcal{C})_{\mathrm{SF}} = \{(\boldsymbol{a}, \boldsymbol{a}H^T), \boldsymbol{a} \in (\mathbb{F}_2)^n\}.$$

These dual realizations are a special case of the more general dual I/O systems to be discussed in the next subsection.

C. Adjoints and Dual I/O Systems

In this subsection we show that the dual of a homomorphic I/O system is the adjoint I/O system, with inputs and outputs trading places. For linear systems, these results are classical.

The *adjoint* of a homomorphism φ: $G \to U$ is the unique homomorphism φ^*: $U^\wedge \to G^\wedge$ such that $\langle v, \varphi(g)\rangle = \langle \varphi^*(v), g\rangle$ for all $g \in G$, $v \in U^\wedge$. The adjoint character $\varphi^*(v)$ is simply the unique character in $G^\wedge$ whose values are given by $\varphi^*(v)(g) = \langle v, \varphi(g)\rangle$. Evidently the adjoint of φ^* is φ: $\varphi^{**} = \varphi$.

Now let us consider a group code $\mathcal{C} \subseteq G \times U$ for which the first symbol is an information set. Then $\mathcal{C}$ is isomorphic to G, and may be written in the form of an I/O system

$$\mathcal{C} = \{(g, \varphi(g)), g \in G\}$$

where the I/O map φ: $G \to U^\wedge$ is a homomorphism.

The dual code $\mathcal{C}^\perp \subseteq G^\wedge \times U^\wedge$ is then evidently

$$\mathcal{C}^\perp = \{(-\varphi^*(v), v), v \in U^\wedge\}$$

since $(h, v) \in G^\wedge \times U^\wedge$ is orthogonal to all codewords in $\mathcal{C}$ if and only if

$$\langle -h, g\rangle = \langle v, \varphi(g)\rangle, \qquad \text{for all } g \in G$$

but this implies $-h = \varphi^*(v)$, by the definition of the adjoint φ^*.

Fig. 42 shows block diagrams of the I/O system representing $\mathcal{C}$ and the dual I/O system representing $\mathcal{C}^\perp$. Note that in the dual

Fig. 42. Block-diagram realizations of (a) homomorphic I/O system and (b) dual I/O system.

Fig. 43. Dual graphs for image and kernel representations, showing $\mathrm{Im}\,\varphi = (\mathrm{Ker}\,\varphi^*)^\perp$

system the I/O map φ is replaced by its negative adjoint $-\varphi^*$, the input and output are interchanged, and the directions of all edges are reversed. An arrow has been placed on each block to indicate the direction of the homomorphism.

In a group block diagram, all edges are directed, and every block may be written as a homomorphic I/O system as above by letting G be the direct product of all input alphabets and U be the direct product of all output alphabets. We thus arrive at the following general prescription for dualizing a group block diagram.

1) Replace all homomorphisms by their negative adjoints, and reverse their direction.
2) Replace all edge alphabets by their character groups, and reverse each edge direction.
3) Put sign inverters in the middle of all internal edges.

By Theorem 7.3, the dual block diagram then generates the dual code (system).

As we shall see in an example below, if the primal system is causal, then I/O reversal makes the dual system anticausal. In order to represent the dual system as a causal system, we must reverse the direction of time (negate time indexes $k \in I \subseteq \mathbb{Z}$). Thus, the well-known principle of time reversal in dual systems fundamentally follows from I/O reversal.

A fundamental duality result for adjoints is that the kernel of a homomorphism φ is the dual (orthogonal subgroup) of the image of its adjoint φ^*, and *vice versa*. This result follows directly from Theorem 7.3 applied to the dual graphs shown in Fig. 43. In system theory, these are called image and kernel representations of Im φ and Ker φ^*, respectively. Fig. 41(c) and (b) are examples of standard image (generator-matrix) and kernel (parity-check-matrix) representations in a coding context.

We next consider three elementary types of homomorphisms and their adjoints, and show how every homomorphism and its adjoint can be built up from such elementary homomorphisms.

Consider first an isomorphism ψ; then its adjoint ψ^* is evidently also an isomorphism. Consider next a natural map $\pi\colon G \to G/K$; then its adjoint π^* is the injection $\iota\colon K^\perp \to G\hat{}$, where $K^\perp$ is identified with the character group $(G/K)\hat{}$ of G/K. (This follows from the fact that the characters in $K^\perp$ are constant on the cosets of K in G, $\langle v, \pi(g)\rangle = \langle v, g\rangle$ for all $g \in G$, $v \in K^\perp$, which is essentially why $K^\perp$ may be taken as the character group of G/K.) Similarly, the adjoint of an injection $\iota\colon G \to U$ is the natural map $\pi\colon U\hat{} \to U\hat{}/G^\perp$, where $U\hat{}/G^\perp$ acts as the character group of G.

By the fundamental theorem of homomorphisms, any homomorphism $\varphi\colon G \to U$ with kernel K and image V may be

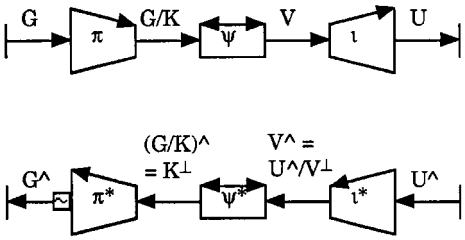

Fig. 44. Block-diagram realizations of homomorphism $G \xrightarrow{\pi} G/K \xleftarrow{\psi} V \xrightarrow{\iota} U$, and adjoint $U\hat{} \xrightarrow{\iota^*} V\hat{} = U\hat{}/V^\perp \xleftarrow{\psi^*} (G/K)\hat{} = K^\perp \xrightarrow{\pi^*} G\hat{}$.

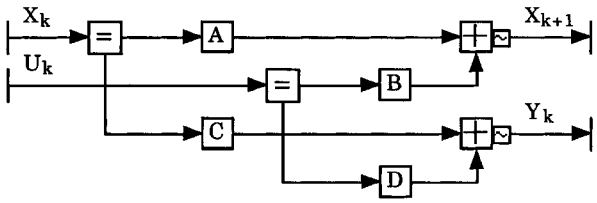

Fig. 45. Block diagram of $(A,\ B,\ C,\ D)$ input/state/output realization.

written as the composition of a natural map $\pi\colon G \to G/K$, an isomorphism from G/K to V, and an injection $\iota\colon V \to U$. A block diagram is shown in Fig. 44. By block-diagram duality, the adjoint $\varphi\colon U\hat{} \to G\hat{}$ may correspondingly be written as the composition of a natural map $\iota^*\colon U\hat{} \to U\hat{}/V^\perp = V\hat{}$, an isomorphism from $U\hat{}/V^\perp$ to $K^\perp = (G/K)\hat{}$, and an injection $\pi\colon K^\perp \to G\hat{}$, as shown also in Fig. 44. (Sign inverters have been minimized.)

Notice again that the kernel $V^\perp$ of φ^* is the dual of the image V of φ, and the image $K^\perp$ of φ^* is the dual of the kernel K of φ.

For another example, consider the standard (A, B, C, D)-matrix linear input/state/output realization over a field $\mathbb{F}$ given by the equations

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k + Du_k$$

where x_{k+1} and x_k are state n-tuples, u_k is an input m-tuple, y_k is an output p-tuple, and $(A,\ B,\ C,\ D)$ are matrices of appropriate dimension.

A block diagram of this realization is shown in Fig. 45. Note that explicit three-symbol repetition vertices are used in place of "degree-3 hyperedges." We also convert each ordinary block-diagram adder representing, e.g., $x_{k+1} = Ax_k + Bu_k$ to a zero-sum constraint followed by a sign-inverter, representing, e.g., $-x_{k+1} + Ax_k + Bu_k = 0$.

To dualize this block diagram, we use the fact that the adjoint of a matrix homomorphism, for example, $B\colon \mathbb{F}^m \to \mathbb{F}^n$, is the transpose matrix homomorphism, for example, $B^T\colon \mathbb{F}^n \to \mathbb{F}^m$. We also use the duality between repetition and zero-sum constraints, and the sign inverter reduction rules given in Section VII-D. We then obtain the dual block diagram of Fig. 46.

Fig. 46 has the same functional form as Fig. 45. If we reverse the direction of time in the dual diagram (negate time indexes) to make the system causal, then Fig. 46 represents the linear input/state/output realization given by the equations

$$x_{-k}\hat{} = -A^T x_{-k-1}\hat{} - C^T y_{-k}\hat{}$$
$$u_{-k}\hat{} = -B^T x_{-k-1}\hat{} - D^T y_{-k}\hat{}$$

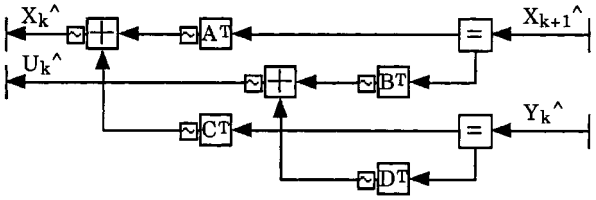

Fig. 46. Block diagram of dual input/state/output realization.

where $x_{-k}\hat{}$ and $x_{-k-1}\hat{}$ are state n-tuples, $y_{-k}\hat{}$ is an input p-tuple, $u_{-k}\hat{}$ is an output m-tuple, and $(-A^T, -B^T, -C^T, -D^T)$ are the dual (A, B, C, D) matrices.

Note that even if the (A, B, C, D) matrices vary with time, this procedure still realizes the dual system. This observation generalizes the usual result for linear time-invariant systems.

The same development clearly holds when inputs, states, and outputs are LCA groups, and (A, B, C, D) are group homomorphisms, if $(-A^T, -B^T, -C^T, -D^T)$ denote negative adjoints. (Such systems were studied under the rubric "finite group homomorphic sequential systems in [7].) However, homomorphic block diagrams are not the most general realization of a group code; see next subsection. Indeed, canonical state realizations need not be homomorphic [16].

D. General Two-Symbol Group Codes

Now let $\mathcal{C} \subseteq G \times U$ be a general two-symbol group code. $\mathcal{C}$ cannot be represented as an I/O system unless the projection onto the first or the second coordinate is an isomorphism. (This is an example of how behavioral realizations are more general than block-diagram realizations.)

The general situation is completely described by the State Space Theorem. Let H be the projection of $\mathcal{C}$ onto the first coordinate, and K the cross section. Similarly, let V be the projection of $\mathcal{C}$ onto the second coordinate, and W the cross section. Then $(g, u) \in G \times U$ is a codeword in $\mathcal{C}$ if and only if $g \in H$, $u \in V$, and the coset $g + K$ of K in H corresponds to the coset $u + W$ of W in V under the natural isomorphism of the State Space Theorem.

A graph of a realization of $\mathcal{C}$ according to this description is shown in Fig. 47. All of the blocks (vertices) in the graph are elementary homomorphic I/O systems, but the graph cannot be made into a block diagram in general because there is no consistent way of directing the edges unless $H = G$ and $W = \{0\}$, or alternatively $V = U$ and $K = \{0\}$.

Fig. 48 shows an alternative, equivalent realization, in which the injective and natural map blocks have been interchanged on each side. This makes no difference; we may generate, e.g., H/K by first restricting to H and then reducing modulo K, or, equivalently, by first reducing modulo K and then restricting G/K to H/K.

By dualizing Fig. 48, we obtain a realization for $\mathcal{C}^\perp$ as shown in Fig. 49. All but one sign inverter in the dual realization has been eliminated.

The dual realization again has the form of Fig. 47. We see that the projections onto the first and second coordinates are $K^\perp$ and $W^\perp$, the cross sections are $H^\perp$ and $V^\perp$, and the state space is isomorphic to $K^\perp/H^\perp$ or $W^\perp/V^\perp$; i.e., to the character group of the primal state space. All of this was already known from the dual state space theorem [18], [17], [15], but the situation becomes completely transparent in this pictorial proof based on Theorem 7.3.

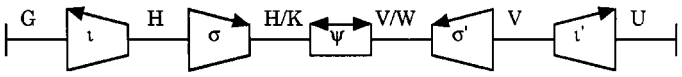

Fig. 47. Realization of general two-symbol group code $\mathcal{C}$.

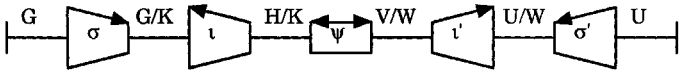

Fig. 48. Alternative realization of general two-symbol group code $\mathcal{C}$.

(Parenthetically, the appearance of the sign inverter in Fig. 49 hints at the difficulties that are encountered in generalizing these results to the non-Abelian case.)

E. A Behavioral Control Theorem and Its Dual

As a final application, we rederive, sharpen, and dualize a pretty behavioral control theorem of Trentelman and Willems [43]. Recall the definition of a conditioned code $(\mathcal{C}|\mathcal{D})_{|J}$, illustrated in Fig. 35(a)

$$(\mathcal{C}|\mathcal{D})_{|J} = \{\boldsymbol{a}_{|J} | \boldsymbol{a} \in \mathcal{C}, \boldsymbol{a}_{|K} \in \mathcal{D}\}.$$

In a control context, we make the following interpretation.

- $\mathcal{C}$ is the behavior of a linear or group I/O system, called a *plant*;
- $\mathcal{A}_{|J}$ are a set of observable variables, called the *to-be-controlled variables*;
- $\mathcal{A}_{|K}$ are a set of connectable variables, called the *control variables*;
- $\mathcal{D}$ is the behavior of a *controller* with variables $\mathcal{A}_{|K}$;
- $(\mathcal{C}|\mathcal{D})_{|J}$ is the *controlled behavior* after the controller is connected to the plant.

The question is, given the full behavior $\mathcal{C}$, what controlled behaviors $(\mathcal{C}|\mathcal{D})_{|J}$ are possible? In view of the coset decomposition (4.2) of the State Space Theorem, the answer is clear.

Theorem 8.6: Let $\mathcal{C}$ be a linear or group plant behavior governing a set of variables partitioned into to-be-controlled variables $\mathcal{A}_{|J}$ and control variables $\mathcal{A}_{|K}$, and let $\mathcal{S}_J = \mathcal{C}_{|J}/\mathcal{C}_J$ be the J-induced state space of $\mathcal{C}$, where $\mathcal{C}_{|J}$ and $\mathcal{C}_J$ are, respectively, the projection and cross section of $\mathcal{C}$ onto J. Then a controlled behavior $(\mathcal{C}|\mathcal{D})_{|J}$ is achievable by choice of a controller behavior $\mathcal{D}$ if and only if $(\mathcal{C}|\mathcal{D})_{|J}$ is a union of cosets of $\mathcal{C}_J$ in $\mathcal{C}_{|J}$.

Proof: In view of (4.2), any collection of cosets of $\mathcal{C}_J$ in $\mathcal{C}_{|J}$ may be determined by selecting the corresponding collection of cosets of $\mathcal{C}_K$ in $\mathcal{C}_{|K}$ as control variables. Conversely, no finer partition of nor addition to the to-be-controlled variables is possible.

Evidently, the maximal controlled behavior $(\mathcal{C}|\mathcal{D})_{|J}$ is the projection $\mathcal{C}_{|J}$, which is obtained if the controller behavior is free, i.e., $(\mathcal{C}|\mathcal{D})_{|J} = \mathcal{C}_{|J}$ if $\mathcal{D} = \mathcal{A}_{|K}$, and the minimal controlled behavior $(\mathcal{C}|\mathcal{D})_{|J}$ is a single coset of the cross section $\mathcal{C}_J$, which is obtained if the controller behavior is unary (deterministic); e.g., $(\mathcal{C}|\mathcal{D})_{|J} = \mathcal{C}_J$ if $\mathcal{D} = \{\mathbf{0}\}$.

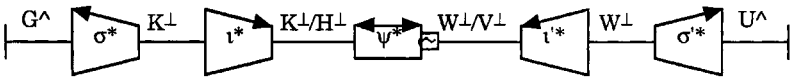

Fig. 49. Dual realization of dual two-symbol group code $\mathcal{C}^\perp$.

If we require that the controller behavior $\mathcal{D}$ be a linear or group behavior, then we obtain the theorem of Trentelman and Willems [43], extended to the group case.

Theorem 8.7 [43]: Let $\mathcal{C}$ be a linear or group plant behavior governing a set of variables partitioned into to-be-controlled variables $\mathcal{A}_{|J}$ and control variables $\mathcal{A}_{|K}$, and let $S_J = \mathcal{C}_{|J}/\mathcal{C}_J$ be the J-induced state space of $\mathcal{C}$, where $\mathcal{C}_{|J}$ and $\mathcal{C}_J$ are, respectively, the projection and cross section of $\mathcal{C}$ onto J. Then, a controlled behavior $(\mathcal{C}|\mathcal{D})_{|J}$ is achievable by choice of a linear or group controller behavior $\mathcal{D}$ if and only if $(\mathcal{C}|\mathcal{D})_{|J}$ is a group and

$$\mathcal{C}_J \subseteq (\mathcal{C}|\mathcal{D})_{|J} \subseteq \mathcal{C}_{|J}.$$

Proof: If $\mathcal{D}$ is a group, then the conditioned code $(\mathcal{C}|\mathcal{D})_{|J}$ is a group. Since $\{\mathbf{0}\} \subseteq \mathcal{D} \subseteq \mathcal{C}_{|K}$, (4.2) implies that $\mathcal{C}_J \subseteq (\mathcal{C}|\mathcal{D})_{|J} \subseteq \mathcal{C}_{|J}$. Conversely, if $(\mathcal{C}|\mathcal{D})_{|J}$ is a group and $\mathcal{C}_J \subseteq (\mathcal{C}|\mathcal{D})_{|J} \subseteq \mathcal{C}_{|J}$, then the collection of cosets of $\mathcal{C}_K$ corresponding to $(\mathcal{C}|\mathcal{D})_{|J}$ is a subgroup of $\mathcal{C}_{|K}$. □

As noted in [43], this theorem implies that control problems may be reduced to finding an acceptable controlled behavior $(\mathcal{C}|\mathcal{D})_{|J}$ that lies between given minimal and maximal controlled behaviors $\mathcal{C}_J$ and $\mathcal{C}_{|J}$; an explicit controller $\mathcal{D}$ need not be found.

Finally, by projection/cross-section duality, we have the following duality between minimal and maximal controlled behaviors.

Theorem 8.8: Let $\mathcal{C}$ be a linear or group plant behavior governing a set of variables partitioned into to-be-controlled variables $\mathcal{A}_{|J}$ and control variables $\mathcal{A}_{|K}$, and let $\mathcal{C}_{|J}$ and $\mathcal{C}_J$ be the maximal and minimal controlled behaviors, respectively. Then the minimal and maximal controlled behaviors of the dual plant behavior $\mathcal{C}^\perp$ are $(\mathcal{C}_{|J})^\perp$ and $(\mathcal{C}_J)^\perp$, respectively. Thus, a controlled behavior $(\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J}$ is achievable by choice of a linear or group controller behavior $\mathcal{D}^\perp$ if and only if $(\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J}$ is a group and $(\mathcal{C}_{|J})^\perp \subseteq (\mathcal{C}^\perp|\mathcal{D}^\perp)_{|J} \subseteq (\mathcal{C}_J)^\perp$.

IX. Dualizing the Sum-Product Algorithm

On a conventional state realization (trellis), the sum-product algorithm is called the "BCJR" or "forward–backward" algorithm. Several authors [22], [2], [21], [39] have observed that APPs can be computed for binary linear codes by transforming the likelihood weights and running the BCJR algorithm on the dual-code trellis, which can be simpler when the dual code is low-rate. Berkmann has extended this result to linear codes over nonbinary fields [4] and over rings $\mathbb{Z}_m$ [5].

Here we generalize this result to any cycle-free normal group realization of any finite Abelian group code $\mathcal{C}$. The cycle-free condition is required in order that the sum-product algorithm be an exact APP decoding algorithm. The finite Abelian condition is primarily to simplify exposition; analogous results hold for general LCA groups.

A. Fourier Transforms and the Poisson Summation Formula

The *Fourier transform* of a real- or complex-valued function $\{f(g), g \in G\}$ of a group G is the function $\{F(h), h \in G^\wedge\}$ of its character group $G^\wedge$ defined by

$$F(h) = \sum_{g \in G} \langle h, g \rangle f(g), \qquad h \in G^\wedge$$

where $\langle h, g \rangle = e^{-2\pi i g h}$ is the Fourier kernel defined previously [15].

The key to the use of the dual graph to compute APPs is the powerful and general Poisson Summation Formula (also the key to the MacWilliams identities), which says that the sum of a function over a subgroup $H \subseteq G$ is proportional to the sum of its Fourier transform over the orthogonal subgroup $H^\perp \subseteq G^\wedge$ [15].

Theorem 9.1 (Poisson Summation Formula): Let $\{f(g), g \in G\}$ be a real- or complex-valued function defined on a finite Abelian group G, let $\{F(h), h \in G^\wedge\}$ be its Fourier transform, and let $H \subseteq G$ and $H^\perp \subseteq G^\wedge$ be orthogonal subgroups. Then

$$\sum_{g \in H} f(g) = \frac{1}{|H^\perp|} \sum_{h \in H^\perp} F(h).$$

B. The Sum-Product Algorithm on the Dual Realization

As previously noted, if all code configurations $\boldsymbol{a} \in \mathcal{C}$ are equiprobable, then the kth APP vector

$$\boldsymbol{p}_k = \{p(A_k = a_k|\boldsymbol{y}), a_k \in \mathcal{A}_k\}$$

given an observed vector $\boldsymbol{y}$ is proportional to (written $\equiv_\alpha$)

$$\begin{aligned} \boldsymbol{p}_k &\equiv_\alpha \{p(\boldsymbol{y}|A_k = a_k), a_k \in \mathcal{A}_k\} \\ &\equiv_\alpha \left\{ \sum_{\boldsymbol{a} \in \mathcal{C}} \delta(\boldsymbol{a}_{|k} = a_k) p(\boldsymbol{y}|\boldsymbol{a}), a_k \in \mathcal{A}_k \right\} \end{aligned} \tag{9.1}$$

where the indicator function $\delta(\boldsymbol{a}_{|k} = a_k)$ restricts the sum to $\mathcal{C}(a_k) = \{\boldsymbol{a} \in \mathcal{C} | \boldsymbol{a}_{|k} = a_k\}$. We assume as before that the channel is memoryless so that $p(\boldsymbol{y}|\boldsymbol{a}) = \prod_k i_k(a_k)$, where $\boldsymbol{i}_k = \{i_k(a_k) = p(y_k|a_k)\}$ is the kth *intrinsic information vector.*

Since the input weight $i_k(a_k)$ appears in every term in (9.1), we may extract it and write $p_k(a_k) \equiv_\alpha i_k(a_k) e_k(a_k)$, where

$$e_k(a_k) = \sum_{\boldsymbol{a} \in \mathcal{C}} \delta(\boldsymbol{a}_{|k} = a_k) \left(\prod_{k' \neq k} i_{k'}(\boldsymbol{a}_{|k'}) \right), \qquad a_k \in \mathcal{A}_k$$

are the components of the kth *extrinsic information vector* $\boldsymbol{e}_k$ [21].

Viewed as a black box, the sum-product algorithm operating on a finite cycle-free graph takes the intrinsic information vector

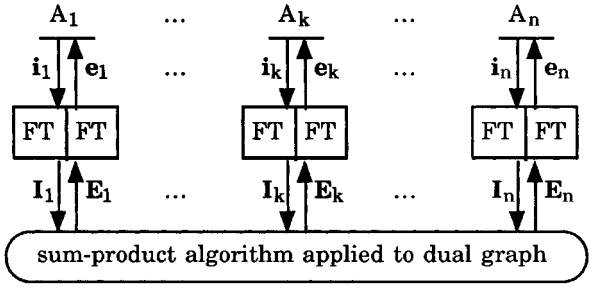

Fig. 50. Computing APPs using the sum-product algorithm on the dual graph.

$\boldsymbol{i}_k$ as its input at the kth I/O terminal, and in a finite time returns the extrinsic information vector $\boldsymbol{e}_k$, as illustrated in Fig. 20.

Applying the Poisson Summation Formula, we have

$$e_k(a_k) \equiv_\alpha \sum_{\boldsymbol{b}\in\mathcal{C}^\perp} \Delta_{a_k}(\boldsymbol{b}_{|k}) \left(\prod_{k'\neq k} I_{k'}(\boldsymbol{b}_{|k'}) \right)$$

where $\Delta_{a_k}(\boldsymbol{b}_{|k})$ and $I_{k'}(\boldsymbol{b}_{|k'})$ are, respectively, the Fourier transforms of $\delta(\boldsymbol{a}_{|k} = a_k)$ and $i_{k'}(a_{k'})$, and we have noted that the Fourier transform of a product function

$$\delta(\boldsymbol{a}_{|k} = a_k) \left(\prod_{k'\neq k} i_{k'}(a_{k'}) \right)$$

is the corresponding product function, namely

$$\Delta_{a_k}(\boldsymbol{b}_{|k}) \left(\prod_{k'\neq k} I_{k'}(\boldsymbol{b}_{|k'}) \right).$$

Explicitly, we can evaluate the transform of $\delta(\boldsymbol{a}_{|k} = a_k)$ as

$$\Delta_{a_k}(\boldsymbol{b}_{|k}) = \sum_{\boldsymbol{a}_{|k}\in\mathcal{A}_k} \langle \boldsymbol{b}_{|k}, \boldsymbol{a}_{|k}\rangle \delta(\boldsymbol{a}_{|k} = a_k) = \langle \boldsymbol{b}_{|k}, a_k\rangle.$$

Breaking up the sum over $\boldsymbol{b} \in \mathcal{C}^\perp$ into terms for which $\boldsymbol{b}_{|k} = b_k$ for each $b_k \in \hat{\mathcal{A}_k}$, we obtain

$$\begin{aligned} e_k(a_k) &\equiv_\alpha \sum_{b_k\in\hat{\mathcal{A}_k}} \langle b_k, a_k\rangle \sum_{\boldsymbol{b}\in\mathcal{C}^\perp} \delta(\boldsymbol{b}_{|k} = b_k) \left(\prod_{k'\neq k} I_{k'}(\boldsymbol{b}_{|k'}) \right) \\ &= \sum_{b_k\in\hat{\mathcal{A}_k}} \langle b_k, a_k\rangle E_k(b_k) \end{aligned} \tag{9.2}$$

where we define

$$E_k(b_k) = \sum_{\boldsymbol{b}\in\mathcal{C}^\perp} \delta(\boldsymbol{b}_{|k} = b_k) \left(\prod_{k'\neq k} I_{k'}(a_{k'}) \right).$$

Now observe that $\boldsymbol{E}_k$ is the kth extrinsic information vector for the dual code $\mathcal{C}^\perp$ if the likelihood weight vectors are $\boldsymbol{I}_k$. Thus, (9.2) expresses the primal extrinsic information vector $\boldsymbol{e}_k$ as the Fourier transform of the dual extrinsic information vector $\boldsymbol{E}_k$.

It follows that we can compute APPs using the sum-product algorithm on the dual graph if we simply Fourier-transform the intrinsic (likelihood) vector inputs and the extrinsic information vector outputs as shown in Fig. 50.

It should be noted that the transformed likelihood vectors may take values in a different range than the original likelihood vectors, which are nonnegative real. Unless the alphabet groups are all binary (isomorphic to $(\mathbb{Z}_2)^n$ for some n), the transformed likelihoods will, in general, be complex; even if they are binary, their transforms will, in general, have negative components. While such a change in range may not cause problems in a digital implementation, it could pose problems in analog implementations of the sum-product algorithm as in [30].

If the symbol alphabets are two-valued, then since

$$(w_0, w_1) \equiv_\alpha (1, w_1/w_0)$$

a two-valued likelihood vector can be represented simply by the likelihood ratio $\lambda = w_1/w_0$. Since the Fourier transform of $(1, \lambda)$ is

$$(1+\lambda, 1-\lambda) \equiv_\alpha (1, (1-\lambda)/(1+\lambda))$$

this leads to the "tanh rule" for transforming likelihood ratios [22], [2], [21]

$$\lambda \mapsto \Lambda = \frac{1-\lambda}{1+\lambda} = \tanh(e^{\lambda/2})$$

which is its own inverse (i.e., $\lambda = (1-\Lambda)/(1+\Lambda)$). Note, however, that the range of λ is $[0, \infty]$, whereas the range of its transform Λ is $[-1, 1]$.

Note also that a deterministic likelihood vector transforms to an erasure, and vice versa. This general property holds also for nonbinary variables.

C. Dualizing the Sum-Product Update Rule

The sum-product algorithm may be used to decode any cycle-free graph fragment. Therefore, the Fourier transform method of the previous section may be used to compute the sum-product update rule, as previously observed in [37].

In words, the dual sum-product update rule can be expressed as follows.

1) Transform the upstream weight vectors $\boldsymbol{w}_{j'\uparrow\uparrow}$ to their Fourier transforms $\boldsymbol{W}_{j'\uparrow\uparrow}$.
2) Execute the sum-product update rule (5.4), using the transforms $\boldsymbol{W}_{j'\uparrow\uparrow}$ and the dual local code $\mathcal{C}_i^\perp$.
3) Transform the resulting vector to obtain a vector proportional to the weight vector $\boldsymbol{w}_{j\uparrow}$.

This yields the following equation, which we call the *dual sum-product update rule*:

$$\boldsymbol{w}_{j\uparrow} \equiv_\alpha \left\{ \sum_{s'_j\in\mathcal{S}_j} \langle s'_j, s_j\rangle \sum_{\boldsymbol{s}'\in\mathcal{C}_i^\perp} \delta(\boldsymbol{s}'_{|j} = s'_j) \prod_{j'\in I'_{\mathcal{S}}(i)} W_{j'\uparrow\uparrow}(\boldsymbol{s}'_{|j'}) \right\}. \tag{9.3}$$

The dual sum-product update rule may be less complex than the sum-product update rule if $\mathcal{C}_i$ is a high-rate code and Fourier transforms are easy to compute. For example, if $\mathcal{C}_i$ is a binary $(n, n-1, 2)$ SPC code, then the input weight λ may be transformed by the "tanh rule," and the dual sum-product update rule may be executed by a simple componentwise multiplication, because the dual code $\mathcal{C}_i^\perp$ is a simple binary $(n, 1, n)$ repetition code.

When the sum-product algorithm is used for APP decoding, it suffices to compute weight vectors up to a scale factor. If exact weight vectors are desired, then the exact scale factor may be obtained from the Poisson Summation Formula.

X. Conclusion

Normal realizations lead to graphical representations of codes that are nicer than factor graphs in some respects. There

is only one vertex type, and there are no topological restrictions. A clean functional separation is obtained between operations in the sum-product decoding algorithm. Computational complexity is associated entirely with vertices, and is appropriately measured by the sizes of local codes. Edges are sites for cuts and for communication between graph components; these properties accord with the essence of the system-theoretic notion of "state."

Most strikingly, we have shown that duals of normal realizations generate dual codes. We have seen that this fundamental result underlies a great many other duality results.

It would be interesting to explore new classes of code constructions using normal realizations. Any such code may be decoded with iterative versions of the sum-product algorithm. Careful deployment of state spaces may yield simpler realizations and better decoding performance.

For example, with classical LDPC codes, there are no explicit state spaces, and all implicit state spaces are binary. We believe that the performance improvements observed from using nonbinary symbol alphabets [11] may be attributable to the implicit use of larger state spaces. (However, we note that near-capacity results have now been obtained [38], [10] with binary LDPC codes.) Similarly, turbo codes might benefit by using nonbinary state spaces to connect component codes.

Finally, one would like to define classes of canonical realizations on graphs with cycles that are in some sense "minimal." The work of [25] on minimal tail-biting realizations shows that this will not be a straightforward problem.

ACKNOWLEDGMENT

The author is grateful to R. Koetter, F. R. Kschischang, H.-A. Loeliger, T. Mittelholzer, D. Spielman, N. Wiberg, and A. Vardy for various comments and suggestions, particularly to H.-A. Loeliger and T. Mittelholzer for pointers to [33], to F. R. Kschischang and D. Spielman for proofs of the results of Section IX in the binary case, to A. Vardy for a proof of the nonexistence of a conjectured tetrahedral realization of the binary Golay code, and to S.-Y. Chung for the RM decoding simulations reported in Section VI. He would also like to thank the authors of [1] for preprints of their papers, and the reviewers for their helpful comments.

REFERENCES

[1] S. M. Aji and R. J. McEliece, "The generalized distributive law," *IEEE Trans. Inform. Theory*, vol. 46, pp. 325–343, Mar. 2000.

[2] G. Battail, M. C. Decouvalære, and P. Godlewski, "Replication decoding," *IEEE Trans. Inform. Theory*, vol. IT-25, pp. 332–345, May 1979.

[3] C. H. Bennett and P. W. Shor, "Quantum information theory," *IEEE Trans. Inform. Theory*, vol. 44, pp. 2724–2742, Oct. 1998.

[4] J. Berkmann, "On turbo decoding of nonbinary codes," *IEEE Commun. Lett.*, vol. 2, pp. 94–96, Apr. 1998.

[5] ——, "A symbol-by-symbol MAP decoding rule for linear codes over rings using the dual code," in *Proc. 1998 IEEE Intl. Symp. Information Theory*, Cambridge, MA, Aug. 1998, p. 90.

[6] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes (I)," in *Proc. ICC '93*, Geneva, Switzerland, June 1993, pp. 1064–1070.

[7] R. W. Brockett and A. S. Willsky, "Finite group homomorphic sequential systems," *IEEE Trans. Automat. Contr.*, vol. AC-17, pp. 483–490, 1972.

[8] A. R. Calderbank, G. D. Forney Jr., and A. Vardy, "Minimal tail-biting trellises: The Golay code and more," *IEEE Trans. Inform. Theory*, vol. 45, pp. 1435–1455, July 1999.

[9] S.-Y. Chung, private communication, 1999.

[10] S.-Y. Chung, G. D. Forney Jr., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB from the Shannon limit," *IEEE Commun. Lett.*, submitted for publication.

[11] M. C. Davey and D. J. C. MacKay, "'Low-density parity-check codes over $GF(q)$," *IEEE Commun. Letters*, vol. 2, pp. 165–167, June 1998.

[12] D. Divsalar, H. Jin, and R. J. McEliece, "Coding theorems for 'turbo-like' codes," in *Proc. 1998 Allerton Conf. Communication, Control and Computers*, Allerton, IL, Sept. 1998, pp. 201–210.

[13] G. D. Forney Jr., "Coset codes—II: Binary lattices and related codes," *IEEE Trans. Inform. Theory*, vol. 34, pp. 1152–1187, Sept. 1988.

[14] ——, "Transforms and groups," in *Codes, Curves and Signals: Common Threads in Communications*, A. Vardy, Ed. Norwood, MA: Kluwer, 1998, ch. 7.

[15] ——, "Group codes and behaviors," in *Dynamical Systems, Control, Coding, Computer Vision*, G. Picci and D. S. Gilliam, Eds. Zürich, Switzerland: Birkhauser, 1999, pp. 301–320.

[16] G. D. Forney Jr. and M. D. Trott, "The dynamics of group codes: State spaces, trellis diagrams and canonical encoders," *IEEE Trans. Inform. Theory*, vol. 39, pp. 1491–1513, Sept. 1993.

[17] ——, "Controllability, observability and duality in behavioral group systems," in *Proc. 34th Conf. Decision and Control*, vol. 3, New Orleans, LA, Dec. 1995, pp. 3259–3264.

[18] ——, "The dynamics of group codes: Dual abelian group codes and systems," *IEEE Trans. Inform. Theory*, submitted for publication.

[19] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[20] ——, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[21] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 429–445, Mar. 1996.

[22] C. R. Hartmann and L. D. Rudolph, "An optimum symbol-by-symbol decoding rule for linear codes," *IEEE Trans. Inform. Theory*, vol. IT-22, pp. 514–517, Sept. 1976.

[23] E. Hewitt and K. A. Ross, *Abstract Harmonic Analysis I*, 2nd ed. New York: Springer, 1979.

[24] S. Kaplan, "Extension of the Pontrjagin duality—I: Infinite products," *Duke Math. J.*, vol. 15, pp. 649–658, 1948.

[25] R. Kötter and A. Vardy, "Factor graphs: Construction, classification and bounds," in *Proc. 1998 IEEE Int. Symp. Information Theory*, Cambridge, MA, Aug. 1998, p. 14.

[26] F. R. Kschischang and B. J. Frey, "Iterative decoding of compound codes by probability propagation in graphical models," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 219–230, Feb. 1998.

[27] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inform. Theory*, vol. 47, pp. 498–519, Feb. 2001.

[28] H.-A. Loeliger, G. D. Forney Jr., T. Mittelholzer, and M. D. Trott, "Minimality and observability in group systems," *Linear Alg. Appl.*, vol. 205–206, pp. 937–963, July 1994.

[29] H.-A. Loeliger and T. Mittelholzer, "Convolutional codes over groups," *IEEE Trans. Inform. Theory*, vol. 42, pp. 1660–1686, Nov. 1996.

[30] H.-A. Loeliger, F. Lustenberger, M. Helfenstein, and F. Tarköy, "Probability propagation and decoding in analog VLSI," *IEEE Trans. Inform. Theory*, vol. 47, pp. 837–843, Feb. 2001.

[31] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low-density parity-check codes," *Electron. Lett.*, vol. 32, pp. 1645–1646, Aug. 1996. Reprinted in *Electron. Lett.*, vol. 33, pp. 457–458, Mar. 1997.

[32] R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 140–152, Feb. 1998.

[33] T. Mittelholzer, "Convolutional codes over groups: A pragmatic approach," in *Proc. 33d Allerton Conf. Communiccation, Control and Computers*, Allerton, IL, Sept. 1995, pp. 380–381.

[34] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*. San Mateo, CA: Kaufmann, 1988.

[35] L. Pontryagin, *Topological Groups*. Princeton, NJ: Princeton Univ. Press, 1946.

[36] A. Reznik, "Iterative decoding of codes defined on graphs," M.Sc. thesis, MIT, Cambridge, MA, May 1998.

[37] T. J. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, pp. 599–618, Feb. 2001.

[38] T. J. Richardson, A. Shokrollahi, and R. Urbanke, "Design of provably good low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 619–637, Feb. 2001.
[39] S. Riedel, "New symbol-by-symbol MAP decoding algorithm for high-rate convolutional codes that uses reciprocal dual codes," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 175–185, Feb. 1998.
[40] W. Rudin, *Fourier Analysis on Groups*. New York: Wiley, 1990.
[41] M. Sipser and D. A. Spielman, "Expander codes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 1660–1686, Nov. 1996.
[42] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. IT-27, pp. 533–547, Sept. 1981.
[43] H. Trentelman, "A truly behavioral approach to the H_∞ control problem," in *The Mathematics of Systems and Control: From Intelligent Control to Behavioral Systems*, J. W. Polderman and H. Trentelman, Eds. Groningen, Germany: U. Groningen, 1999, pp. 177–190.
[44] A. Vardy, "Trellis structure of codes," in *Handbook of Coding Theory*, V. Pless and W. C. Huffman, Eds. Amsterdam, The Netherlands: Elsevier, 1998.
[45] Y. Weiss and W. T. Freeman, "On the optimality of solutions of the max-product belief propagation algorithm in arbitrary graphs," *IEEE Trans. Inform. Theory*, vol. 47, pp. 736–744, Feb. 2001.
[46] N. Wiberg, "Codes and decoding on general graphs," Ph.D. dissertation, Univ. Linköping, Linköping, Sweden, 1996.
[47] N. Wiberg, H.-A. Loeliger, and R. Kötter, "Codes and iterative decoding on general graphs," *Euro. Trans. Telecomm.*, vol. 6, pp. 513–525, Sept./Oct. 1995.
[48] J. C. Willems, "Models for dynamics," in *Dynamics Reported*, U. Kirchgraber and H. O. Walther, Eds. New York: Wiley, 1989, vol. 2, pp. 171–269.
[49] ——, "Paradigms and puzzles in the theory of dynamical systems," *IEEE Trans. Automat. Contr.*, vol. 42, pp. 259–294, Mar. 1991.
[50] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Bethe free energy, Kikuchi approximations and belief propagation algorithms," Mitsubishi Elec. Rsch. Lab., Cambridge, MA, preprint, June 2000.