J Stat Phys (2011) 145:860–890
DOI 10.1007/s10955-011-0384-7

# Message-Passing Algorithms for Inference and Optimization

## “Belief Propagation” and “Divide and Concur”

**Jonathan S. Yedidia**

Received: 4 April 2011 / Accepted: 18 October 2011 / Published online: 29 October 2011

**Abstract** Message-passing algorithms can solve a wide variety of optimization, inference, and constraint satisfaction problems. The algorithms operate on factor graphs that visually represent and specify the structure of the problems. After describing some of their applications, I survey the family of belief propagation (BP) algorithms, beginning with a detailed description of the min-sum algorithm and its exactness on tree factor graphs, and then turning to a variety of more sophisticated BP algorithms, including free-energy based BP algorithms, “splitting” BP algorithms that generalize “tree-reweighted” BP, and the various BP algorithms that have been proposed to deal with problems with continuous variables.

The Divide and Concur (DC) algorithm is a projection-based constraint satisfaction algorithm that deals naturally with continuous variables, and converges to exact answers for problems where the solution sets of the constraints are convex. I show how it exploits the “difference-map” dynamics to avoid traps that cause more naive alternating projection algorithms to fail for non-convex problems, and explain that it is a message-passing algorithm that can also be applied to optimization problems. The BP and DC algorithms are compared, both in terms of their fundamental justifications and their strengths and weaknesses.

**Keywords** Message-passing algorithms · Factor graphs · Belief propagation · Divide and concur · Difference-map · Optimization · Inference · Constraint satisfaction

## 1 Introduction

This paper is a tutorial introduction to the important “Belief Propagation” (BP) and “Divide and Concur” (DC) algorithms. The tutorial will be informal, and my main goal is to explain the fundamental ideas clearly.

Iterative message-passing algorithms like BP and DC have an amazing range of applications, and it turns out that their theory is deeply connected to concepts from statistical physics. BP algorithms are already very well-known, and have been discussed in depth in

J.S. Yedidia (✉)
Disney Research Boston, 222 3rd Street, Suite 1101, Cambridge, MA 02142, USA
e-mail: yedidia@disneyresearch.com

some excellent reviews [2, 38, 74]. The DC algorithm [27] has important advantages compared with BP, but is so far much less known and used.

I have a couple target audiences in mind. One target reader is learning about message-passing algorithms for the first time; this paper can serve as a gentle introduction to the field. Another target reader might already have an knowledge of at least some aspects of BP, but may not be familiar with recent advances in our understanding of BP, or with projection-based algorithms like DC, which are not usually described as message-passing algorithms. For such readers, I will present an overview of different BP algorithms, and show that DC can be interpreted as a message-passing algorithm that can be easily compared and contrasted with other BP algorithms.

## 2 Inference and Optimization Problems

No algorithm can be discussed in isolation from the problem it is solving. BP and DC Message-passing algorithms are used to solve *inference* problems, *optimization* problems, and *constraint satisfaction* problems.

In an inference problem, one takes as input some noisy or ambiguous measurements, and tries to infer from those measurements the likely state of some hidden part of the world. In general, it is impossible to make those inferences with complete certainty, but one can at least try to obtain, within a model of the world and the measurements, the most *probable* state of the hidden part of the world [46, 63].

As an example, consider the “channel coding” problem which is fundamental to information theory [22, 60, 64]. We want to transmit a message consisting of some sequence of bits across a noisy channel that might corrupt those bits. To deal with that noise, additional bits that depend on the message are appended to it by an encoder before it is transmitted. The task of the decoder is one of probabilistic inference: it tries to compute, from all the possibly noisy received bits, the most probable message that might have been transmitted.

Computer vision is another example of a field where probabilistic inference problems are ubiquitous [17, 21, 66]. In a typical scenario, one obtains images or videos from one or more cameras, and wants to infer something about the scene being captured. For a computer, photographic images are simply two-dimensional matrices of color intensities, and the scene is a hidden three-dimensional world of objects that must somehow be inferred from those inherently ambiguous measurements. A computer vision probabilistic inference system uses a statistical model that connects the camera measurements to the scene quantities of interest (e.g. the depth or the identity of objects), and tries to find the most probable interpretation of the scene quantities given the measurements.

Speech recognition [28, 31, 58] and language understanding [33, 48] systems are similar. Here one obtains measurements of a sound signal, and tries to infer the most probable sequence of words, or meanings, consistent with those sounds.

In statistical physics, one deals with mathematically analogous problems [39, 51]. If one is given the energy function for some magnetic system or some macromolecule, the most probable configuration of the system is the one that has the lowest energy. Of course, as physicists are well aware, the lowest energy configuration may be the most probable, while not being a *typical* configuration. To obtain probabilities about typical configurations, one must also take into account the entropy of the system. In probabilistic inference, this distinction corresponds to the difference between two tasks the system might be asked to perform. First, it might be asked to obtain the one most probable state of the entire system, or second, one might ask for the marginal probabilities of particular hidden variables, after summing over the probabilities of all possible configurations.

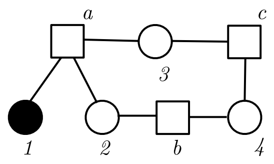


**Fig. 1** A toy factor graph with one observed variable node (variable 1), three hidden variable nodes, and three factor nodes

A probabilistic inference problem can be converted into a statistical physics problem by defining an energy function using Boltzmann's Law:

$$p(X) \propto \exp(-E(X)). \tag{1}$$

That is, if we are given a statistical model (say for coding, or vision, or speech recognition) that assigns probabilities $p(X)$ to states $X$, we can completely equivalently think of it as assigning energies $E(X)$, and proceed using any method of our choice from statistical physics. Or, if we are given a physical system with an energy function $E(X)$, we can fruitfully apply the probabilistic inference algorithms that have been invented by computer scientists and electrical engineers. Mathematically, many of the problems studied by statistical physicists and computer scientists are completely equivalent [51].

## 3 Factor Graphs

The task of inferring the most probable state of a system is actually the optimization problem of finding the minimum energy of that system. I will now describe a very convenient data structure called a *factor graph* [40, 43] which can be used to visualize and precisely define such optimization problems. A variety of other "graphical models" exist (e.g. Bayesian networks [56], Markov random fields [25], normal factor graphs [19, 43]), and have their advantages, but all these models can be easily converted into the standard factor graphs I will discuss [76].

Factor graphs are bipartite graphs containing two types of nodes: *variable* nodes and *factor* nodes. See Fig. 1 for our first example of a toy factor graph.

The variable nodes, usually denoted using circles, represent the variables in the optimization problem. A variable might be discrete—that is, it can only take on a finite number of states, like an Ising spin, which can only be in the two states that we might call "up" and "down"—or it might have a continuous range of possible states.

If a variable's state is known (or "observed"), we fill in the corresponding circle in the factor graph; otherwise we use an open circle to denote a so-called "hidden" variable. The factor graph of Fig. 1 has three hidden variables and one observed variable.

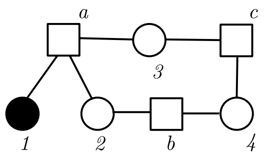
The "factor" nodes in a factor graph show how the overall objective ("energy" or "cost") function of our optimization problem breaks up—factorizes—into local terms. We draw an edge between each factor node representing a local cost function, and the variables that are involved in that local cost function. For example, for the factor graph of Fig. 1, the overall "cost" $C(x_1, x_2, x_3, x_4)$ for a configuration $(x_1, x_2, x_3, x_4)$ will be

$$C(x_1, x_2, x_3, x_4) = C_a(x_1, x_2, x_3) + C_b(x_2, x_4) + C_c(x_3, x_4). \tag{2}$$

More generally, if there are $M$ local cost functions, we can write the overall cost function $C(X)$ as

$$C(X) = \sum_{a=1}^{M} C_a(X_a) \tag{3}$$

| $x_1$ | $x_2$ | $x_3$ | $C_a$ |
|---|---|---|---|
| 0 | 0 | 0 | ∞ |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | ∞ |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | ∞ |
| 1 | 1 | 0 | ∞ |
| 1 | 1 | 1 | 0 |



| $x_2$ | $x_4$ | $C_b$ |
|---|---|---|
| 0 | 0 | 1.2 |
| 0 | 1 | 1.7 |
| 0 | 2 | 3.2 |
| 1 | 0 | 1.9 |
| 1 | 1 | 0.6 |
| 1 | 2 | 1.4 |

| $x_3$ | $x_4$ | $C_c$ |
|---|---|---|
| 0 | 0 | 0.4 |
| 0 | 1 | 1.9 |
| 0 | 2 | 0.2 |
| 1 | 0 | 4.9 |
| 1 | 1 | 0.3 |
| 1 | 2 | 2.4 |

**Fig. 2** A factor graph, along with the lookup tables associated with each local cost function. Note that for this factor graph the four variables are all discrete. Factor *a* is a "hard" constraint, that either allows or disallows different local configurations, while factors *b* and *c* are "soft" factors

where $X_a$ is the set of variables involved in the $a$th local cost function.

By itself, a factor graph just lets one visualize the relationships between the variables and local functions in the problem; but it does not give the full information one needs to compute the cost associated with a configuration. To fill out that information, we would need to provide a table or explicit function for each factor node. For example, in Fig. 2, we supplement the factor graph from Fig. 1 with lookup tables that specify the local costs associated with each factor. Notice that in this example some of the local costs for factor $a$ are infinite; this means that the corresponding configurations are forbidden and represents a "hard" constraint. For factors $b$ and $c$, all the costs are finite, so these factors represent "soft" preferences.

The factor graph we have been looking at is just a toy example; we normally are interested in problems where there are a large number of hidden variables—at least hundreds, maybe thousands or millions. In that case, there will be an exponentially huge number of possible states of the system, so while it is always easy to compute the energy of one configuration, finding the lowest energy state, or summing over all the states, can be a very hard problem.

In fact, it should be obvious that our formulation is so general that many NP-hard optimization problems can be described in this way. Thus, we certainly are not going to be able to specify here an algorithm that is guaranteed to find the lowest cost state of an arbitrary factor graph, while using computational resources that only grow polynomially with the size of a factor graph. Nevertheless, it turns out that the BP and DC message-passing algorithms are often successfully used on problems that are NP-hard, and can in fact often efficiently find the global optimum for those problems. The point is that a problem might be NP-hard, and yet specific instances of it that we care about may be in an easier regime than the worst case, and often be solved by a clever algorithm. For example, the problem of finding the optimal decoding of a low-density parity-check (LDPC) code is NP-hard, and yet efficient state-of-the-art BP decoders succeed sufficiently often that they closely approach the Shannon limit of possible channel coding performance [60, 64].

## 4 Example Factor Graphs and Applications

Let's take a look at just a few examples of how factor graphs can represent interesting problems. To simplify the factor graphs in the following examples, we take advantage of the fact that we can absorb any observed variable nodes as parameters in the factors they are attached to, leaving only hidden variables in the factor graph (see Fig. 3).

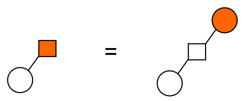


**Fig. 3** Any observed variable nodes in a factor graph can be absorbed as parameters in the factor nodes that they are connected to, leaving only "hidden" variable nodes, and factor nodes that depend on the observations

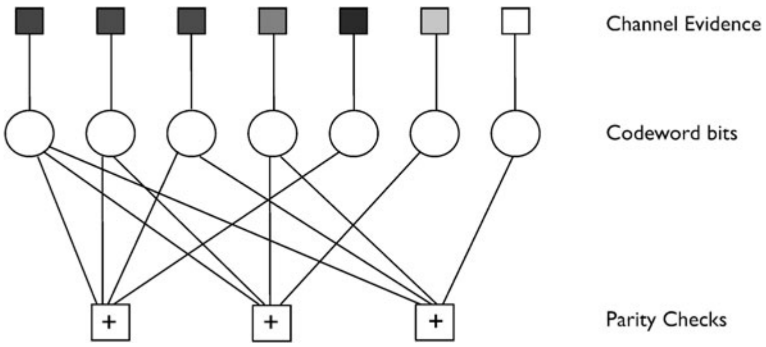


**Fig. 4** A factor graph for the $(N = 7, k = 4)$ Hamming code, which has seven codeword bits, of which the *left-most four* are information bits, and the *last three* are parity bits

In the following examples, we will also describe some properties of BP message-passing algorithms, anticipating our more extensive discussion in future sections.

## 4.1 Error Correcting Codes

Our first example is a factor graph for an error correcting code—the simple $(N = 7, k = 4)$ Hamming code shown in Fig. 4. In this code, there are seven "hidden" variable nodes that represent the seven unknown transmitted bits. The first four of those bits are information bits that encode the original message, the other three are additional parity bits that can be computed from the information bits using the parity check factor nodes. The three parity check factor nodes are hard constraints that force the sum of the bits connected to them to equal 0 modulo 2. There are also seven "soft" channel evidence factor nodes that give the *a priori* probability that each of the hidden codeword bits is equal to a one or zero, given the observed received bits.

The goal will be to find the most likely values of the seven hidden transmitted bits, given the channel evidence and the fact that they must be consistent with the parity check constraints.

Such factor graphs were introduced into coding theory in 1981 by Tanner [68], to describe and visualize the low-density parity-check (LDPC) codes and the BP decoder for LDPC codes that had been introduced earlier by Gallager in 1963 [23]. LDPC codes were given their name because each parity check is only connected to a small number of codeword bits. LDPC codes and their factor graphs are similar to the Hamming code in Fig. 4, except that the number of codeword bits is usually on the order of a few thousand in practical LDPC codes, and the codeword bits are not simply divided into information bits and extra parity check bits. BP decoders of LDPC codes are very practically significant, because if they are properly designed, their performance can closely approach the Shannon limit, and they can be implemented in modern hardware [60, 64].

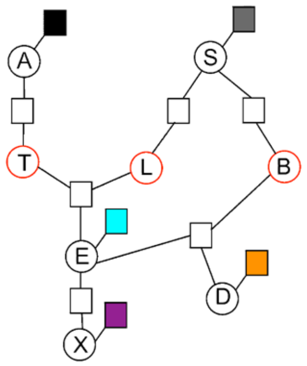


**Fig. 5** A toy factor graph representing an "expert system" for a medical diagnosis problem. The hidden variables represent the unknown possible diseases (e.g. "*L*" represents lung cancer), or information about the patient such whether he is a smoker ("*S*") or test results. The factors represent known statistical relationships (e.g. a smoker is more likely to have lung cancer). The goal is to obtain the best estimate of the probabilities of possible diseases given the available information

## 4.2 Diagnosis

Our next example (see Fig. 5) is a toy factor graph representing a medical diagnosis problem. The hidden variable nodes in this factor graph labeled "T," "L," and "B" represent different possible diagnoses of some patient (e.g. Tuberculosis, Lung Cancer or Bronchitis). The other variables may represent some information about the patient (e.g. "S" represents how much the patient smokes). The factor nodes encode the known statistical relationships between diseases and other variables.

This factor graph is adapted from an example in [41] that originally used a Bayes network, a graphical model that uses directed edges and that has the advantage of explicitly encoding *conditional* probabilities. The name "Belief Propagation" was in fact introduced by Pearl [56] for a version of the BP algorithm working on Bayes networks. It is easy to convert between different graphical models [76], and one reason that I am focusing here on standard factor graphs is that BP algorithms are easier to describe on standard factor graphs because they are undirected.

This example highlights an important point about probabilistic inference algorithms—one is often interested in more detailed information than the overall most probable configuration. For example, one might want an accurate estimate of the marginal probability that the patient has a particular disease. As we shall see, different BP algorithms are designed to give answers to different types of questions—for the kind of marginal probabilities we want here, we should use the "sum-product" version of BP, which we will describe in more detail later in this paper.

## 4.3 Computer Vision

Our next example (see Fig. 6) is a cartoon factor graph depicting the way factor graphs are used in low-level computer vision. The colored factor nodes in this example represent image intensity pixels that would be captured by a camera. The hidden variable nodes are some variables about the scene that we would like to infer (for example, the depth from the camera associated with each pixel). These hidden variables have some local probabilities given the observations, but there are also correlations between them—for example, if the

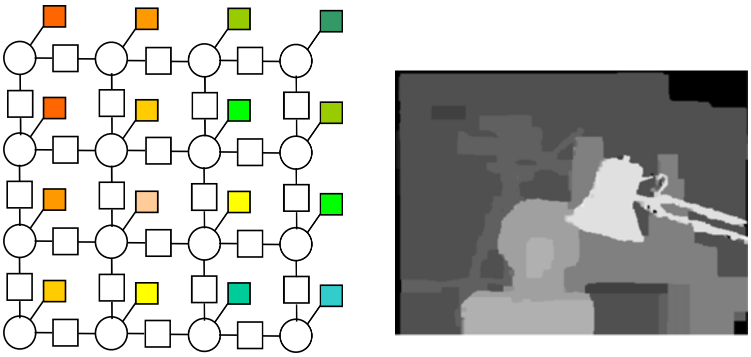


**Fig. 6** (Color online) An illustration of a factor graph equivalent to a pairwise Markov random field (*left*) as is often used to representing a computer vision problem, and a depth map returned by a BP algorithm [17] using stereo images (*right*). The variable nodes in the factor graph could represent unknown depth values, for which there exists some local evidence (the *colored factor nodes*), and which are statistically correlated (depths at particular pixels tend to be similar to nearby pixels)

depth has a certain value at one pixel, it is likely (but not guaranteed) to have similar values at neighboring pixels.

Our goal, as usual, will be to find the most probable values of the hidden scene variables given what our cameras observe.

Graphical models called "pairwise Markov random fields," which are equivalent to factor graphs like that shown in Fig. 6, were introduced into computer vision by Geman and Geman [25], and have become increasingly popular in the last 25 years [17, 21, 66], although the models used in practice are more complicated than the simplified cartoon shown in this figure. In a factor graph equivalent to a pairwise Markov random field, each factor node is connected to no more than two variable nodes.

One significant practical problem for BP algorithms is that the hidden variables of interest in computer vision are typically continuous, and are often severely quantized for the sake of efficiency when BP (or competing algorithms like "graph cuts" [8]) are run. This means that one often sees quantization artifacts in the results. The depth map on the right hand side of Fig. 6 is the result of running a BP algorithm on a pairwise Markov random field for stereo vision [17], and shows such artifacts.

Much recent research has focused on improving BP's performance or efficiency for problems with continuous variables (see Sect. 12), but nevertheless the fact that the Divide and Concur algorithm works naturally with continuous quantities to any desired precision makes it an attractive potential alternative to BP for computer vision applications.

It should be emphasized that factor graphs only give a principled way of *representing* an optimization or probabilistic inference problem. You then need to separately choose an algorithm to *solve* it.

Many different variants of message-passing algorithms exist, and of course there are many other optimization algorithms (e.g. simulated annealing) that can be used once the problem is represented as a factor graph. I focus here on message-passing algorithms because they are often particularly powerful and efficient.

One should be careful to cleanly separate the model of an optimization or inference problem from the algorithm being used on that model. When a clean separation is made, one can determine whether it is the model or the optimization algorithm that needs improvement if one obtains an inadequate solution. For example, artifacts obtained using a particular

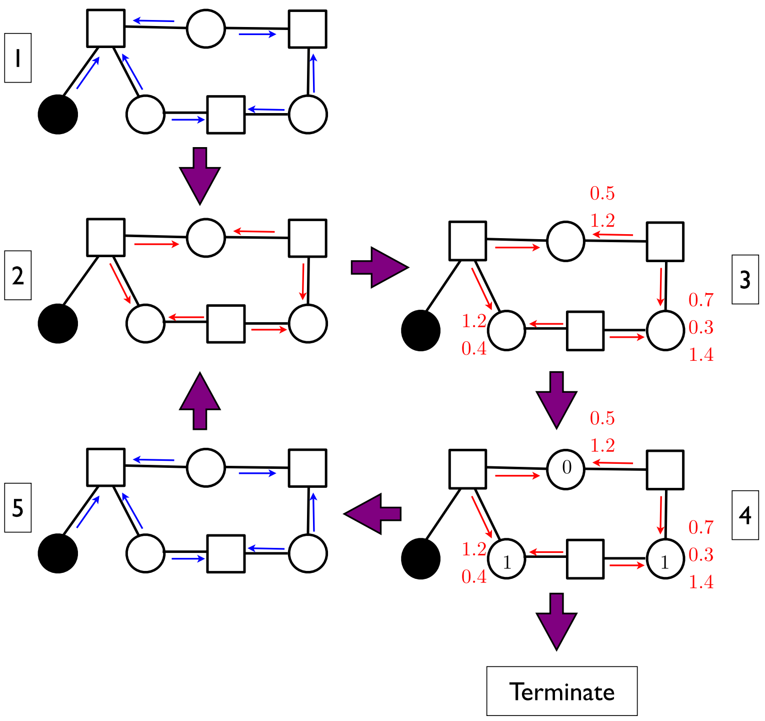


**Fig. 7** (Color online) The overall structure of message-passing algorithms. The algorithms iterate between steps, indicated by numbers in rectangles. In *step 1*, messages from variable nodes to factor nodes (in *blue*) are initialized to random or non-informative values. In *step 2*, the factor nodes compute from the incoming messages new outgoing messages (in *red*). In *step 3*, those messages are converted into beliefs, which in BP are generally represented as a cost for each possible state (the *red numbers*). In *step 4*, the beliefs are thresholded to their lowest cost state (represented by the *number inside the variable node*), and a termination condition is checked. In *step 5*, the beliefs and incoming messages are used to compute new outgoing messages from the variable nodes, and then one returns to *step 2*, and the cycle continues

stereo vision model (that was in principle NP-hard to optimize) were for a long time blamed on the inability of optimization algorithms to find the global optimum. Eventually though, it was found that even the *provably globally optima* found by BP-based algorithm were not completely adequate, demonstrating that it was the model that needed to be improved [50].

## 5 Message-Passing Algorithms

Let's turn now to the overall structure of message-passing algorithms, as they operate on factor graphs. This overall structure is shared by different classes of BP algorithms and by DC algorithms.

Figure 7 breaks the structure down into five steps. Message-passing algorithms get their name from the fact that in each step, messages are sent between nodes in the factor graph.

In the first step, the messages from variable nodes to factor nodes are initialized. The initial messages could be random, but more often, one uses non-informative messages from the hidden nodes, and messages that correspond to the observations from the observed nodes. Messages should be thought of as a variable telling its neighboring factors what it thinks its state is, or what it thinks would be the cost for it to be in each of its possible states. A non-informative message simply sets the costs of each possible state to be equal.

The factor nodes take in all the messages from their neighboring variable nodes, and in the second step, they compute messages that go back out to neighboring hidden variable nodes. These messages take into account what the variable nodes have told them, and the statistical relationships encoded by a factor node. The messages tell the neighboring variables what state they should be in, or what the costs will be for being in all their possible states.

In the third step, the hidden variable nodes inspect all the incoming messages, and compute a "belief" about what their state should be. In BP algorithms, this belief takes the form of a cost, or a probability, associated with each possible state of a variable. In the example shown in Fig. 7, the three hidden variable nodes have respectively two, two, and three possible states, so the beliefs are similarly vectors (represented by the columns of red numbers) of the same lengths giving the costs of each state. Thus in a BP algorithm, if a variable is continuous valued, the belief associated with it must be a function of that continuous variable. In DC algorithms, the belief (and the messages) are just a single number, representing the current best guess for the state of that variable node.

The fourth step of a message passing algorithm is necessary for BP algorithms but not for DC, and involves thresholding the beliefs to obtain a single best guess for a variable node. In our example, the beliefs represent costs, and the lowest cost state is chosen for each variable node (for example, the bottom right variable node has costs 0.7 for state 0, 0.3 for state 1, and 1.4 for state 2, so state 1 has the lowest cost for that variable).

After the fourth step is completed, one has obtained a guess for the overall configuration of the factor graph, and one can use that guess to check for a termination condition. For example, in BP-based decoders of LDPC codes, one can check whether the guess is a legal codeword that satisfies all the parity-check constraints. Or one can check whether the guess or the beliefs have changed from previous iterations, or whether a maximum number of iterations has been reached. If the termination condition is satisfied, one outputs the current guess.

Otherwise, in the fifth step, the variable nodes will compute new messages to send back to the factor nodes, based on their beliefs and the messages that they received. Then one goes back to the second step, and the cycle repeats.

Notice that all the factor nodes can compute their outgoing messages in step 2 in parallel, and similarly all the variable nodes can compute their outgoing messages in parallel in step 5. That makes these algorithms attractive for parallel implementation, either in hardware or software, and has contributed significantly to their popularity.

## 6 Message and Belief Update Rules

Message-passing algorithms differ in the details of and justifications for their message-update rules. I will begin by presenting one particular BP algorithm, the "min-sum" algorithm. This algorithm uses messages and beliefs that represent the costs for each variable to be in its different possible states. One sometimes sees the "min-sum" algorithm in a different guise, and referred to as the "max-product" BP algorithm. "Max-product" BP is equivalent to "min-sum" BP; the only (completely superficial) difference is that messages and beliefs are represented as probabilities rather than costs.

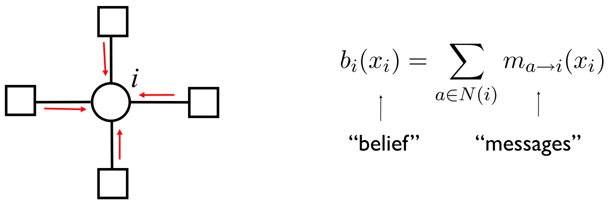


**Fig. 8** The belief update rule for the min-sum BP algorithm says that the belief at a variable node is simply the sum of incoming messages from neighboring factor nodes

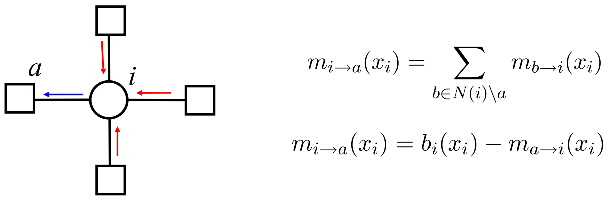

a
i
$m_{i\to a}(x_i) = \sum_{b \in N(i)\backslash a} m_{b\to i}(x_i)$
$m_{i\to a}(x_i) = b_i(x_i) - m_{a\to i}(x_i)$

**Fig. 9** (Color online) The variable-to-factor message update rule in min-sum BP says that the outgoing (*blue*) message is the sum of all the incoming (*red*) messages on edges other than the edge of the outgoing message

I begin with the belief update rules for min-sum BP, which relate the beliefs $b_i(x_i)$ at the variable node $i$ to the messages $m_{a\to i}(x_i)$ coming into node $i$ from neighboring factor nodes $a$ (see Fig. 8). The rule is very simple: the belief is the sum of all the messages:

$$b_i(x_i) = \sum_{a \in N(i)} m_{a\to i}(x_i). \tag{4}$$

Here we use the notation $x_i$ to represent the possible states of variable node $i$, and $N(i)$ to be the set of factor nodes neighboring node $i$.

This rule is easy to understand if we think of the factor nodes as giving independent information about different parts of the graph. For example, if node $i$ is a binary variable neighboring two factor nodes, and the first factor thinks it will cost $A$ more for node $i$ to be in state 1 than state 0, and the second thinks it will cost $B$ more to be in state 1, than it is natural to conclude that overall it will cost $A + B$ more to be in state 1, so long as the two factor nodes are using information from independent parts of the graph.

Turning next to the message update rule for a message $m_{i\to a}(x_i)$ from a variable $i$ to a factor node $a$ (see Fig. 9), we see that the message depends on all messages coming into variable node $i$ from neighboring factor nodes $b$ except for the one coming in from the target factor node $a$:

$$m_{i\to a}(x_i) = \sum_{b \in N(i)\backslash a} m_{b\to i}(x_i). \tag{5}$$

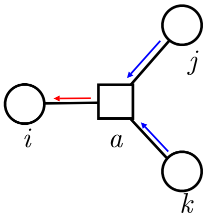
Again, the message out about the costs of the possible states of node $i$ is just a sum of incoming messages from other parts of the graph. Note that if the belief has already been computed, we can use it to more efficiently compute the outgoing message:

$$m_{i\to a}(x_i) = b_i(x_i) - m_{a\to i}(x_i). \tag{6}$$

To complete our collection of update rules, we need the rule updating messages from factor to variable nodes. It's a little more complicated than the other rules, but still easy to understand. Let's look at the case when the factor node $a$ is connected to three variable nodes $i$, $j$, and $k$ (see Fig. 10). The message update rule

$$m_{a\to i}(x_i) = \min_{x_j, x_k}[C_a(x_i, x_j, x_k) + m_{j\to a}(x_j) + m_{k\to a}(x_k)]. \tag{7}$$

The three terms in this equation can be understood as follows. The message $m_{a\to i}(x_i)$ should have node $a$ tell node $i$ what its costs would be for being in each of its possible



$$m_{a\to i}(x_i) = \min_{x_j, x_k} \left[ C_a(x_i, x_j, x_k) + m_{j\to a}(x_j) + m_{k\to a}(x_k) \right]$$

**Fig. 10** The factor-to-variable message update rule for a message from factor $a$ to variable $i$ depends on the local cost function $C_a$, and the incoming variable-to-factor messages on other edges

states. For each choice of state for node $i$, we will arrange the other nodes attached to $i$ to be in their best states given that choice (that is the explanation for the minimization over $x_j$ and $x_k$). One part of the cost is the cost associated with $a$ itself (the $C_a(x_i, x_j, x_k)$ term). The other parts of the cost are what it costs to place the $x_j$ and $x_k$ variables in the states that $x_i$ would like them to be in, given by the incoming messages $m_{j\to a}(x_j)$ and $m_{k\to a}(x_k)$.

If $X_a$ is the set of all variables attached to factor node $a$, and $X_a \backslash x_i$ is all those variables except for $x_i$, then the general update rule for messages from factor nodes to variable nodes, generalizing (7), is

$$m_{a\to i}(x_i) = \min_{X_a \backslash x_i} \left[ C_a(X_a) + \sum_{j \in N(a)\backslash i} m_{j\to a}(x_j) \right]. \tag{8}$$

The form of (8) gives the min-sum algorithm its name. If the beliefs and messages represent probabilities instead of costs, we would obtain an equivalent message update rule with the sums replaced by products, and the minimization replaced by a maximization, so that equivalent algorithm is called the "max-product" algorithm.

We have written these rules as equations, but in fact they are normally used as iterative update rules. Only at a fixed point will they become equalities.

## 7 Example of the Min-Sum BP Algorithm

Let's take a look at a concrete example of the min-sum BP algorithm in action, using as an example a decoder of the binary Hamming code with the factor graph shown in Fig. 4. Each of the codeword bit variables $x_i$, with the index $i$ running from 1 to 7, will have two possible states: $x_i = 0$ or $x_i = 1$.

For this factor graph, there are ten factor nodes, of which seven are channel evidence factor nodes, and three are parity check factor nodes. The seven channel evidence factor nodes each have a soft cost function $C_i(x_i)$ associated with them, representing the relative cost that a bit is a 0 or 1, given what was received from the channel. As an example, suppose that

$$\begin{aligned}
C_1(x_1 = 0) &= 0.0; \qquad & C_1(x_1 = 1) &= 3.0 \\
C_2(x_2 = 0) &= 0.0; \qquad & C_2(x_2 = 1) &= 2.0 \\
C_3(x_3 = 0) &= 0.0; \qquad & C_3(x_2 = 1) &= 2.5 \\
C_4(x_4 = 0) &= 0.0; \qquad & C_4(x_2 = 1) &= 5.4 \\
C_5(x_5 = 0) &= 0.0; \qquad & C_5(x_2 = 1) &= 4.0
\end{aligned}$$

$$C_6(x_6 = 0) = 0.2; \qquad C_6(x_2 = 1) = 0.0$$
$$C_7(x_7 = 0) = 0.7; \qquad C_7(x_2 = 1) = 0.0.$$

Based just on the channel evidence, the first five bits would prefer to be 0, while the last two bits would prefer to be 1, although the preferences of the last two bits are less strong than those of the first five. The three parity check factor nodes, whose cost function we denote by $C_A(x_1, x_2, x_3, x_5)$, $C_B(x_1, x_2, x_4, x_6)$ and $C_C(x_1, x_3, x_4, x_7)$, introduce additional constraints. We assign a zero cost to the legal configurations of the connected bits, where the modulo-two sum is equal to 0, and an infinite cost to the configurations where the modulo-two sum is equal to 1. For example, for the first parity check node, we have that $C_A(x_1 = 0, x_2 = 0, x_3 = 0, x_5 = 0)$, $C_A(x_1 = 0, x_2 = 0, x_3 = 1, x_5 = 1)$, and so on are all equal to zero, while $C_A(x_1 = 0, x_2 = 0, x_3 = 0, x_5 = 1)$, $C_A(x_1 = 0, x_2 = 0, x_3 = 1, x_5 = 0)$ and so on are all infinite.

Let us now see how the beliefs and messages are updated, following the overall structure of a message-passing algorithm shown in Fig. 7. In step 1, all the messages from variables to factors are initialized to zero. In step 2, we compute messages from factor nodes to variable nodes using (8). Because each channel evidence node is connected to only a single variable node, the message update rule from evidence factor nodes to variable nodes simplifies to the simple rule $m_{a_i \to i}(x_i) = C_i(x_i)$ and these messages stay constant through all iterations. On the other hand, because all messages into the parity check factor nodes are initially zero, it is easy to work out that all the initial messages out of the parity nodes will also be zero.

At step 3, we compute the beliefs at the variable nodes, using (4). Because the messages from the parity nodes are zero at this stage, the beliefs will initially be equal to the messages from the evidence nodes: $b_i(x_i) = m_{a_i \to i}(x_i) = C_i(x_i)$.

At step 4, we threshold these beliefs; that is, we assign the first 5 bits the value 0, and the last two bits the value 1, because these are currently the lowest cost values according to the beliefs. Let us assume that our termination condition is that the thresholded beliefs correspond to a codeword (that is, whether they satisfy all the parity checks in the code). For our current bit values, the second and third parity checks are violated, so we need to continue.

At step 5, we compute messages from variable nodes to factor nodes. The messages from variable nodes to evidence nodes are in fact irrelevant because they are never used to compute anything else, so we focus on the relevant messages from the variable nodes to the parity check nodes. Initially, all the messages coming from the parity check nodes are zero, so the messages from the variable nodes to the factor nodes will equal the beliefs at the variable nodes, which at this point equal the evidence cost functions. For example, $m_{1 \to A}(x_1) = m_{1 \to B}(x_1) = m_{1 \to C}(x_1) = b_1(x_1) = C_1(x_1)$.

Now we cycle back to step 2, and the computations become more interesting. We need to update the messages from the parity checks to the variable nodes, using (8). As an example, let's focus on the message $m_{A \to 1}(x_1)$ from check $A$ to variable 1. This will depend on the incoming messages to check $A$ from variables 2, 3, and 5. The hard parity check constraint assigns an infinite cost to choices of the three variables $x_2$, $x_3$, and $x_5$ whose sum do not equal $x_1$ modulo-two. We find that $m_{A \to 1}(x_1 = 0)$ equals

$$\begin{aligned} &\min\left[C_A(0, x_2, x_3, x_5) + m_{2 \to A}(x_2) + m_{3 \to A}(x_3) + m_{5 \to A}(x_5)\right] \\ &\quad = \min\left[(0.0 + 0.0 + 0.0), (0.0 + 2.5 + 4.0), (2.0 + 0.0 + 4.0), (2.0 + 2.5 + 0.0)\right] \\ &\quad = 0.0, \end{aligned}$$

while $m_{A \to}(x_1 = 1)$ equals

$$\begin{aligned}&\min\left[C_A(1,x_2,x_3,x_5)+m_{2\to A}(x_2)+m_{3\to A}(x_3)+m_{5\to A}(x_5)\right]\\&\quad=\min\left[(0.0+0.0+4.0),(0.0+2.5+0.0),(2.0+0.0+0.0),(2.0+2.5+4.0)\right]\\&\quad=2.0.\end{aligned}$$

If one thinks about what the min-sum algorithm is doing here, it makes intuitive sense. For each of the two possible states of variable 1, it assigns the associated states of variables 2, 3, and 5 such that they are consistent with the state of 1, while costing as little as possible.

It is straightforward to similarly compute all the other messages from parity checks to variable nodes. We find

$$\begin{aligned}m_{A\to 1}(x_1)&=(0.0,2.0)\\m_{A\to 2}(x_2)&=(0.0,2.5)\\m_{A\to 3}(x_3)&=(0.0,2.0)\\m_{A\to 5}(x_5)&=(0.0,2.0)\\m_{B\to 1}(x_1)&=(0.2,0.0)\\m_{B\to 2}(x_2)&=(0.2,0.0)\\m_{B\to 4}(x_4)&=(0.2,0.0)\\m_{B\to 6}(x_6)&=(0.0,2.0)\\m_{C\to 1}(x_1)&=(0.7,0.0)\\m_{C\to 3}(x_3)&=(0.7,0.0)\\m_{C\to 4}(x_4)&=(0.7,0.0)\\m_{C\to 7}(x_7)&=(0.0,2.5),\end{aligned}$$

where we have represented the messages as vectors in an obvious way.

Now we return to step 3, and update the beliefs, summing together these new messages from the check nodes, as well as the constant messages from the channel evidence nodes. For example

$$\begin{aligned}b_1(x_1)&=m_{a_1\to 1}(x_1)+m_{A\to 1}(x_1)+m_{B\to 1}(x_1)+m_{C\to 1}(x_1)\\&=(0.0,3.0)+(0.0,2.0)+(0.2,0.0)+(0.7,0.0)\\&=(0.9,5.0).\end{aligned}$$

The full set of beliefs at this stage will be

$$\begin{aligned}b_1(x_1)&=(0.9,5.0)\\b_2(x_2)&=(0.2,4.5)\\b_3(x_3)&=(0.7,4.5)\\b_4(x_4)&=(0.9,5.4)\\b_5(x_5)&=(0.0,6.0)\\b_6(x_6)&=(0.2,2.0)\\b_6(x_7)&=(0.7,2.5).\end{aligned}$$

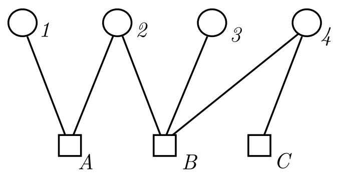


**Fig. 11** A factor graph with no cycles used to illustrate the fact that min-sum BP finds an optimal configuration for such factor graphs

When we return to step 4 and threshold the beliefs, all the bit values will be set to zero, and since this is a codeword consistent with the parity checks, the algorithm will terminate and output the all-zeros codeword.

Notice that the BP algorithm takes advantage of "soft" information. In our example, the algorithm flipped two bits from the values that the channel evidence would have preferred, because the preference at those two bits was weak. If instead we had first assigned each bit to the preferred value from the channel evidence, and then tried to flip the minimum number of bits to find a codeword (many inferior "classical" decoders work in such a fashion, because they cannot use soft cost information), we would have chosen to leave bits 6 and 7 alone, and flipped bit 4 to the value 1 to reach a codeword. This would have only flipped one bit from the preferences of the channel evidence, but it would have cost 5.4 overall instead of 0.9, so it would have been a worse choice.

## 8 Exactness of BP for Tree Factor Graphs

We have already given some intuitive explanations for the form of the min-sum update rules, but a more rigorous justification for the rules is that on a graph with no cycles, applying the min-sum rules will provably give the lowest-cost configuration, using an amount of memory and time that only scales linearly with the number of nodes in the factor graph. I will not prove that fact here, and instead just give an example showing how that works.

Consider the tree factor graph shown in Fig. 11. Suppose that we want to compute the best state for variable node 1 in the optimal configuration. We can get that by computing $b_1(x_1)$, and thresholding it to the lowest cost state. By (4), that belief will be given by $b_1(x_1) = m_{A\to 1}(x_1)$. Now, we can replace $m_{A\to 1}(x_1)$ by using the min-sum update rule (8). If we continually replace messages with cost functions and other messages using the message update rules, we find:

$$\begin{aligned}
b_1(x_1) &= m_{A\to 1}(x_1) \\
&= \min_{x_2} [C_A(x_1, x_2) + m_{2\to A}(x_2)] \\
&= \min_{x_2} [C_A(x_1, x_2) + m_{B\to 2}(x_2)] \\
&= \min_{x_2,x_3,x_4} [C_A(x_1, x_2) + C_B(x_2, x_3, x_4) + m_{3\to B}(x_3) + m_{4\to B}(x_4)] \\
&= \min_{x_2,x_3,x_4} [C_A(x_1, x_2) + C_B(x_2, x_3, x_4) + m_{C\to 4}(x_4)] \\
&= \min_{x_2,x_3,x_4} [C_A(x_1, x_2) + C_B(x_2, x_3, x_4) + C_C(x_4)] .
\end{aligned}$$

In the end, we see that $b_1(x_1)$ gives the exact minimal overall cost for each possible state of the first node, and a similar result would hold starting with any other node.

Intuitively, the reason that BP algorithms are exact on trees is that a message from a node summarizes everything that is happening on the branch of the tree beyond that node. Thus, in our example, the message from node 2 to node $A$ compactly and exactly summarizes everything that we know about nodes $B$, $C$, 3, and 4.

In fact, if should be clear from this example that the min-sum algorithm is a "dynamic programming" algorithm [12] when run on trees, and only needs an amount of computation and memory that scales linearly with the number of nodes in the factor graph.

As we have seen, the min-sum algorithm finds, for a factor graph with no cycles, the lowest-cost or highest-probability configuration of the system. Recall that the max-product algorithm is equivalent to the min-sum version, with the only difference being that messages are represented as probabilities rather than costs. A small modification of the max-product algorithm can be used to obtain exact marginal probabilities $p_i(x_i)$ for the variable nodes being in each of their possible states, averaged over all possible configurations of the system. That modification, which simply replaces the "max" in the factor-to-variable message-update rule with a sum, gives the "sum-product" version of BP. The beliefs in sum-product BP are precisely equal to the desired marginal probabilities so long as the factor graph has no cycles.

## 9 Early History of BP Algorithms

It might be worthwhile at this point to review the history, through the end of the 20th century, of BP algorithms. BP algorithms have in fact been independently invented many times, and applied to a wide range of problems, which is a natural consequence of the fact that they provide an exact solution to problems whose factor graph has no cycles, and such problems are very common. The fundamental commonality between the different versions of the algorithm introduced in different fields has only gradually and relatively recently been understood.

In fact, one could trace the genesis of BP ideas back to the original introduction and solution of the one-dimensional Ising model in 1925 [30], and the development of the transfer matrix approach [5] that was subsequently used to exactly solve a variety of models in statistical mechanics. Statistical physicists will recognize that transfer matrix computations have essentially the same form as BP message update computations, although the transfer matrix method is not usually described as an algorithm.

Because the exactness of the BP algorithm on chains still holds if the variables are continuous, and BP messages can be computed efficiently and exactly for Gaussian models, it turns out that Kalman's 1960 solution of the Kalman filtering problem [35], for which the factor graph represents a temporal chain rather than a spatial one, is also an instance of a BP algorithm [44].

In 1967, Viterbi [70] introduced an algorithm, which would now be viewed as a min-sum or max-product BP algorithm, which gave an exact decoder of convolutional codes. In 1973, Forney [20] made clear the significance of the Viterbi algorithm as an efficient way of finding optimal state sequences for a wide range of problems that can be represented using one-dimensional factor graphs, and the Viterbi algorithm has since had enormous practical importance. Forney also introduced an important graphical visualization of the Viterbi algorithm, the "trellis" diagram (see Fig. 12). Trellis diagrams can be used to effectively track the values of messages as a BP algorithm proceeds forward in time, or equivalently along the variables in a chain factor graph [19]. Each edge in the trellis corresponds to a choice for a value of a one or more variables in the factor graph. For sufficiently simple error-correcting codes, one can obtain optimal decoders from their trellis diagrams; and the important issue often becomes how to represent a code so as to obtain a minimal trellis diagram [69].

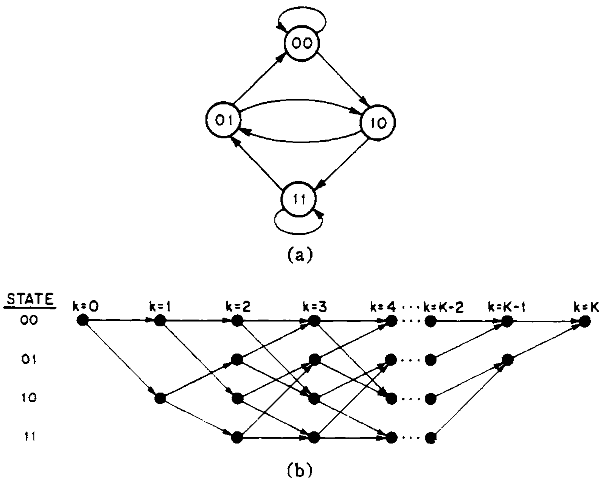


**Fig. 12** An illustration from Forney's paper [20] on the Viterbi algorithm, showing a state diagram for a four-state shift-register process (**a**), and the corresponding trellis diagram given evidence for the state of the process from time 0 to time $k$ (**b**)

The BCJR algorithm (from a modern-perspective, sum-product BP on a one-dimensional chain) was introduced in 1974 and shown to minimize symbol error rates in convolutional codes [1]. This algorithm was also later used for exactly computing marginal probabilities in many other one-dimensional problems, sometimes being called the "forward-backward" algorithm in the context of solving one-dimensional hidden Markov models [58], where it is widely used in bioinformatics [14] and speech recognition [31].

In fact, new problems that can be solved exactly using BP algorithms are continually being discovered. For instance, one can interpret binary decision diagrams (BDDs), introduced by Bryant in 1986 [10], and zero-suppressed binary decision diagrams (ZDDs), introduced by Minato in 1993 [54], as generalizations of trellis diagrams that allow many different types of exact computations to be made for *general* factor graphs over discrete variables, using as little memory and time as possible. Knuth's remarkable recent tutorial [37] describes in detail the algorithms used to efficiently construct BDDs and ZDDs, and surveys the many different kinds of combinatorial problems that they can now efficiently solve, in conjunction with BP algorithms.

From the modern perspective, Gallager's introduction in 1963 of the sum-product algorithm as a decoder of LDPC [23] codes was a particularly significant conceptual breakthrough because it demonstrated that BP algorithms could also serve as effective approximate algorithms even on factor graphs with cycles. In his seminal 1988 book introducing the term "belief propagation," and introducing its application to Bayesian networks for artificial intelligence [56], Pearl focused mostly on BP on tree-like factor graphs, but also mentioned in an exercise that it might be applied to factor graphs with loops. As Pearl pointed out, BP algorithms are perfectly well-defined on factor graphs with cycles, even though they are not necessarily exact. When McEliece, MacKay, and Cheng pointed out in 1998 [49] that Turbo decoders, introduced in 1993 [6], were also an instance of a highly effective, albeit approximate BP algorithm, it became clear that BP provides a "very attractive general methodology for devising low-complexity iterative decoding algorithms," although as they pointed out in their conclusion, it was still mysterious at that point why BP so often gave good approximations for factor graphs with cycles.

## 10 Approaches Based on Free Energies

Some insight into why BP gives good approximations on factor graphs with cycles was obtained when my colleagues Bill Freeman and Yair Weiss and I showed [76, 77] that the fixed points obtained by sum-product BP were identical to the stationary points of a variational free energy, the so-called "Bethe free energy."

Our approach starts with basic definitions from statistical physics. If we have a factor graph with an overall cost function $C(X)$ as defined in (3), then the we can define a corresponding probability distribution over all the states by $p(X) = \exp(-C(X))/Z$, where $Z$ is the partition function $Z = \sum_X \exp(-C(X))$. We can introduce a trial probability, or "belief" function $b(X)$ which is intended to approximate $p(X)$, and a variational free energy $F(b)$, with $F(b) = U(b) - S(b)$ where $U(b)$ is the variational average energy:

$$U(b) = \sum_X b(X)C(X) \tag{9}$$

and $S(b)$ is the variational entropy:

$$S(b) = -\sum_X b(X)\ln b(X). \tag{10}$$

It follows directly from our definitions that

$$F(b) = -\ln Z + D(b||p) \tag{11}$$

where

$$D(b||p) = \sum_X b(X)\ln \frac{b(X)}{p(X)} \tag{12}$$

is the Kullback-Liebler divergence between $b(X)$ and $p(X)$. Since $D(b||p)$ is always non-negative, and is zero precisely when $b(X)$ equals $p(X)$, we see that $F(b) \geq -\ln Z$, with equality when $b(X) = p(X)$. Thus, we can use the procedure of trying to minimize $F(b)$ to recover the true $p(X)$.

In fact, it is not normally tractable to compute the full Gibbs free energy using beliefs over all states of all the nodes in the system, but this variational argument inspires approximations that use beliefs over regions comprising only a limited number of variable nodes. The simplest such approximation, called the Bethe approximation, uses beliefs over "large" regions consisting of the variable nodes attached to a single factor node, and "small" regions consisting of single variable nodes [77].

If we use $b_a(X_a)$ to denote a multi-node belief over all the variable nodes attached to the factor node $a$ (intended to approximate the multi-node marginal probability $p_a(X_a)$) then the Bethe free energy $F_B$ is given by $F_B = U_B - S_B$, where the Bethe average energy $U_B$ is

$$U_B = \sum_a \sum_{X_a} b_a(X_a)C_a(X_a) \tag{13}$$

and the Bethe entropy $S_B$ turns out to be

$$S_B = -\sum_a \sum_{X_a} b_a(X_a)\ln b_a(X_a) + \sum_i (d_i - 1)\sum_{x_i} b_i(x_i)\ln b_i(x_i). \tag{14}$$

Here, $d_i$ is the number of factor nodes neighboring variable node $i$.

The Bethe Free energy $F_B$ is a functional of the beliefs $b_a(X_a)$ and $b_i(x_i)$, which must satisfy the normalization conditions (for all $a$ and $i$):

$$\sum_{X_a} b_a(X_a) = \sum_{x_i} b_i(x_i) = 1 \tag{15}$$

and the marginalization conditions for variable nodes $i$ neighboring factor nodes $a$:

$$b_i(x_i) = \sum_{X_a \backslash x_i} b_a(X_a) \tag{16}$$

where $X_a \backslash x_i$ denotes all variables attached to factor node $a$ except $x_i$.

The marginalization condition turns out to be fundamental: the Lagrange multipliers that enforce it when minimizing the Bethe Free energy turn out to equal linear combinations of the fixed-point messages in sum-product BP.

The Bethe average energy $U_B$ is actually exactly equal to the true average energy of the system given correct beliefs, but the Bethe entropy $S_B$ is only an approximation. Improving that approximation, using larger regions of nodes to compute the entropy, gives better "Kikuchi" [36] or "region graph" free energies, and corresponding "generalized belief propagation" algorithms that gave more accurate marginal probabilities than the sum-product algorithm [77].

Building on these ideas, Chertkov and Chernyak developed another significant approach to improving BP, that begins with sum-product BP or the equivalent Bethe approximation as a zeroth order approximation, and obtains a "loop" series that systematically improves on that approximation [11].

Given the fact that spin glasses and related models can be described using factor graphs, it was perhaps unsurprising that the well developed statistical physics of disordered system, and in particular the "replica" and "cavity field" methods [52] would be related to BP message-passing algorithms [51]. From this perspective, it is clear that the sum-product BP equations only describe a single minimum of the free energy of the system, and ignore any fracturing of phase space such as often occurs in disordered systems. An important and surprising breakthrough exploiting this insight was the development of another improved message-passing algorithm called "survey propagation," based on heuristic ideas that try to take phase-space fracturing into account [53]. Survey propagation has been shown to solve NP-hard random satisfiability problems very effectively, even very close to the satisfiability threshold where they are particularly difficult [9].

On the other hand, one can take the point of view that the Bethe free energy that underlies sum-product BP would be more convenient to work with if it was always convex as a function of the beliefs, as it is when the factor graph is a tree. Wainwright, Jaakkola, and Willsky [71] derived new message-passing algorithms by replacing the Bethe entropy approximation with concave entropies to obtain "convexified" free energies which are guaranteed to have a unique global minimum. They showed that their approach could be used to find upper bounds on the log partition function (or lower bounds on the Helmholtz free energy) of the system.

## 11 Min-sum Algorithms Based on "Splitting"

Wainwright, Jaakkola, and Willsky also introduced related "tree-reweighted" BP algorithms, in both sum-product [72] and max-product [73] form, and proved several powerful theorems about them (see also [65] for extensions of tree-reweighted BP algorithms to solve linear programming problems). These algorithms were originally introduced in the context of pairwise Markov random fields rather than standard factor graphs. I prefer therefore to consider instead the insightful recent formulation of Ruozzi and Tatikonda [61], which generalizes the tree-reweighted max-product algorithm in a way that directly connects to the min-sum BP algorithm on standard factor graphs.

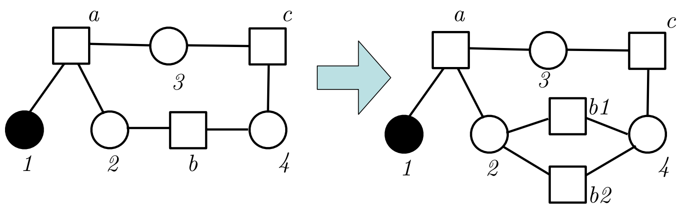


**Fig. 13** One can obtain a new factor graph modeling exactly the same cost function by splitting a factor node into two identical factor nodes, each taking half the cost of the original, and connected to the same variable nodes

Ruozzi and Tatikonda start from the idea that it is easy to create equivalent factor graphs for the same overall cost function by "splitting" either factor or variable nodes. For example, as illustrated in Fig. 13, we can split the factor node $b$ into two factor nodes $b_1$ and $b_2$, each of which has the same neighboring nodes as $b$ and gets exactly half of the cost associated with the original node $b$. Doing this gives us a factor graph that models exactly the same original cost function as the original cost function, but notice that the min-sum algorithm associated with new factor graph is different from the original one.

Let's assume that we initialize all messages from the same variable node but to different split copies of a factor node to be equal to each other, and similarly for messages from different split copies of a factor node. In that case, the messages to and from split copies of a factor node will continue to maintain that symmetry, and can be identified with the messages to and from the factor node in the original factor graph. So in fact, we can translate the message-passing algorithm on the factor graph with split factor nodes into an induced message-passing algorithm on the original factor graph. If each variable node $i$ is split $k_i$ times, and each factor node $a$ is split $k_a$ times, it is easy to verify that the new induced message update rules are

$$m_{i\to a}(x_i) = (k_a - 1)m_{a\to i}(x_i) + \sum_{b\in N(i)\setminus a} k_b m_{b\to i}(x_i) \tag{17}$$

and

$$m_{a\to i}(x_i) = \min_{X_a\setminus x_i}\left[\frac{C_a(X_a)}{k_a} + (k_i - 1)m_{i\to a}(x_i) + \sum_{j\in N(a)\setminus i} k_j m_{j\to a(x_k)}\right]. \tag{18}$$

Notice that the message $m_{i\to a}(x_i)$ now depends directly on the message $m_{a\to i}(x_i)$ coming in the opposite direction on the same edge. This results from splitting: a message to one of the copies of $a$ depends on the messages from the other copies of $a$.

Even though the message update rules (17) and (18) were originally derived using splitting factors $k_a$ and $k_i$ that were positive integers, they are also perfectly well-defined for any real values of $k_a$ and $k_i$, including real values that are less than 1. Ruozzi and Tatikonda show that if $k_a$ and $k_i$ are chosen appropriately (e.g. for a regular graph where each variable node has degree $d$, choose $k_i = 1$, and $k_a$ to be a positive real less than $1/d$), then one can prove some remarkable theorems about the resulting message-passing algorithm.

In particular, if a fixed point of the message-passing update rules is reached, and the resulting single node beliefs each have a unique lowest cost state for that node, then the overall state obtained by combining those single-node states is provably a *global* optimum! This is a surprising theorem, because it holds for arbitrary factor graphs, even those with cycles. Moreover, with the same choice of splitting constants, one can devise simple schedules that provably converge. Of course, the condition that each single-node belief must have a unique lowest cost state at the fixed point (no ties are allowed) is an important loophole that will often prevent a "splitting" message-passing algorithm from giving the globally optimal solution for an NP-hard problem in a difficult regime.

## 12 BP for Factor Graphs with Continuous Variables

Because BP messages and beliefs track the cost of every possible state of a variable, it is much easier in practice to apply BP to problems where the variables all have a small number of possible states. Nevertheless, the BP update rules are well-defined for continuous variables; it is just that the messages and beliefs must be full functions of the variables. There are certain circumstances when these functions can still be computed with efficiently.

First, suppose one tries to parameterize a message or belief function that represents a probability as a Gaussian (or equivalently as a quadratic if we use costs instead of probabilities); in this case one would only need to store its mean and variance. It turns out that there are a variety of important local cost functions such that locally, the max-product and sum-product BP algorithms preserve a Gaussian form [43]. The famous Kalman filtering problem [35] is constructed from local cost functions that all preserve Gaussians, and moreover the factor graph is a chain, so that a Gaussian BP algorithm parameterizing messages using means and variances is exact, and is equivalent to the Kalman smoothing algorithm [43, 44].

For graphs with cycles, a BP algorithm may not converge, and in general its fixed points will no longer exact, even if the local cost functions preserve the Gaussian form of messages. However, in the important special case of a pairwise Markov random field that represents a Gaussian distribution of many variables, Weiss and Freeman [75] and Rusmevichientong and Van Roy [62] proved that if the BP algorithm converges, the calculated means (but not the variances) are exact. The conditions under which Gaussian BP converges [47], and how to fix its convergence properties [32], has been the subject of much recent work.

For certain problems, one can sometimes show that min-sum BP will preserve other functional forms. For example Gamarnik, Shah, and Wei recently showed that for the min-cost network flow problem, the messages preserve a piecewise linear form [24], and proved that min-sum BP converges to the optimal solution.

If one does not have local cost functions that preserve any nice form for the messages, dealing with continuous variables becomes very difficult. Nevertheless, one can begin with the assumption that messages have some nice form (for example Gaussians), then proceed by computing the outgoing messages using the sum-product or max-product BP update rules, and convert the results back into Gaussians by retaining only their means and variances. A systematic way to do this is provided by the popular "Expectation Propagation" algorithm [55].

"Non-parametric BP" is a similar but more accurate method that uses mixtures of Gaussians, and efficient sampling techniques [67]. Finally, the "Particle Belief Propagation" approach [29] builds on the "Particle Filter" sampling technique [67] first developed for temporal processes. However, the drawback of relying on sampling is that it becomes increasingly computationally intensive to obtain more precise results.

As you can see, there are lots of options for dealing with continuous variables in the context of BP, which is not really surprising, because continuous variables are ubiquitous in inference and optimization problems that we care about. For an extensive review, with an emphasis on Gaussian BP for signal processing applications, see [45].

Consider however, the most simple and naive approach imaginable: instead of keeping track of the costs of every possible value of a variable, why don't we just use our single best

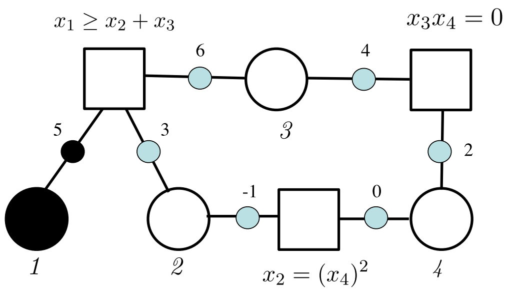


**Fig. 14** (Color online) A constraint graph with three hidden variables, one observed variable, and three hard constraints. The *little blue* "*beads*" placed on the edges represent replicas of the neighboring variable node, which can temporarily take different values, but must be equal at a solution

current guess for the value as a message? It might seem unlikely that such an approach can give good results, but in fact with sufficient ingenuity it can; such an approach underlies the Divide and Concur algorithm, which we turn to now.

## 13 Divide and Concur

The DC algorithm was introduced by Gravel and Elser [27], and builds upon considerable earlier work done by Elser with his students on the use of "difference-map" dynamics in iterative projection algorithms [16]. As I did with the min-sum BP algorithms, I will begin by explaining how to implement DC, and then discuss its justifications.

Like BP, DC can be used to solve optimization and inference problems, but it is easier to begin by considering its application to *constraint satisfaction* problems. In a constraint satisfaction problem, we are looking for a configuration of variables such that all constraints are satisfied. In the language of factor graphs, all "cost" functions are "hard": their only possible values are zero (the constraint is satisfied) or infinite (the constraint is not satisfied).

To explain the DC algorithm it is standard to introduce "replicas" or "copies" of each variable; we introduce one replica of each variable for each constraint it is involved in. (These "replicas" have nothing to do with the "replica method" used for averaging over disorder in statistical physics [51], or the copies used in Ruozzi and Tatikonda's "splitting" method previously described [61].) Of course, the different replicas of the same variable eventually have to equal each other, but temporarily we can allow them to be unequal while they satisfy different constraints. Essentially we are lifting our original problem to a higher dimensional space where it is easier to solve.

In Fig. 14, I show a small example of a "constraint graph," which is useful for visualizing the DC algorithm. A constraint graph is like a factor graph, except that the factor nodes have been replaced with constraint nodes that represent hard constraints. These constraint nodes can represent arbitrary and possibly non-linear constraints on the neighboring variables.

Note that the variables in a constraint graph should always be thought of as real (continuous) numbers. However, problems with discrete variables can easily be handled by simply adding the appropriate constraints to the constraint graph.

Our example constraint graph explicitly represents the replicas of variable as little "beads" on the edges between constraint and variable nodes. Next to each replica bead there is a real number that is the current value of the replica. Notice that the replicas are associated with the edges of the constraint graph, just as BP messages are associated with the edges of factor graphs.

The DC algorithm is built from two "projections" on the replica values. The "Divide" projection does the most natural thing imaginable to satisfy the constraints: it moves the

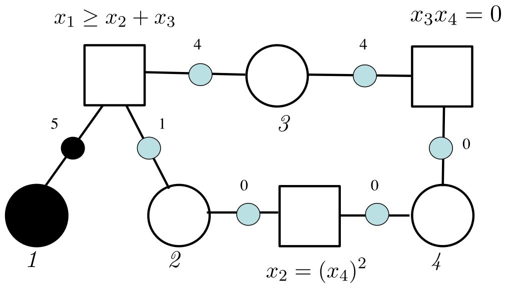


**Fig. 15** State of the replicas of the constraint graph in Fig. 14 after applying a Divide projection

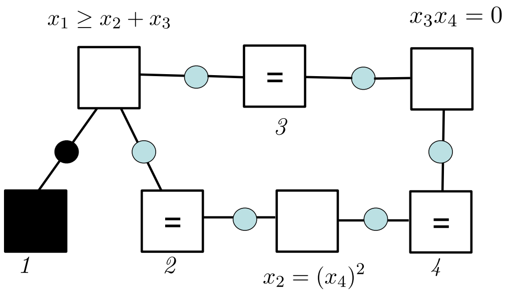


**Fig. 16** A "normal" version of the constraint graph from Fig. 14, where the variable nodes are replaced with constraint nodes that impose equality on neighboring replicas

replica values from their current values to the nearest values that satisfy all the constraints. The "Concur" projection also is completely natural: it averages the replica values that belong to the same variable.

Let us use our example constraint graph, with its replica values, to illustrate how a "Divide projection" works. In the top left, we have a constraint $x_1 \geq x_2 + x_3$, and the current replica values are 5 for the replica of $x_1$, 3 for the replica of $x_2$, and 6 for the replica of $x_3$. We know that $x_1$ is an observed variable, so we leave it fixed, but otherwise we want to move the replicas of $x_2$ and $x_3$ as little as possible, but make them satisfy the constraint $x_1 \geq x_2 + x_3$.

When I say "move as little as possible," of course I must specify a metric. The choice of metric is a crucial issue, but for the moment, let's just use a standard Euclidean metric.

Using this metric, the Divide projection would satisfy its constraint by moving the replicas of $x_2$ and $x_3$ to the values 1 and 4 respectively. At the same time, and in parallel, we can project the replicas around the other constraints ($x_3 x_4 = 0$ and $x_2 = (x_4)^2$) to their nearest values which satisfy those constraints. Note that each replica is connected to only one constraint, so that all these local Divide projections can be done fully in parallel. The overall Divide projection of all the replicas is simply the concatenation of all the local projections.

Figure 15 shows the results of applying the Divide projection to the constraint graph from Fig. 14. It is usually easy to write a subroutine to compute each local divide projection at a particular constraint node, so the DC algorithm divides the overall problem of simultaneously satisfying all the constraints into a lot of small problems of projecting to the nearest solution of a single constraint.

The Concur projection can also be thought of as making the smallest move that satisfies a constraint; now the constraint is that different replicas of the same variable must be equal, and the smallest move is to move all the replicas to the average. Thus we could as well have drawn our constraint graph in the form shown in Fig. 16, where we have replaced the hidden variable nodes with constraints that impose equality on the neighboring replicas, and now the Divide and Concur projections would work in exactly the same way.

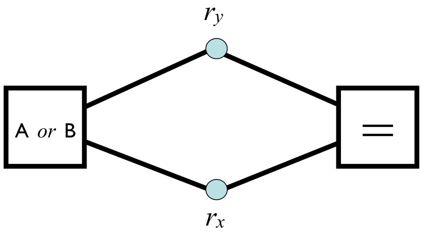


**Fig. 17** A toy "normal" constraint graph used to illustrate replica dynamics

Forney introduced such so-called "normal" versions of standard factor graphs [19], which, while being equivalent to the standard version, have several advantages compared with standard factor graphs in terms of the insight they give. For example, using normal factor graphs, one sees that BP in fact has only one message update rule, just applied to different types of factors. Normal factor graphs are also well-suited for hierarchical modeling (it is easy to create a super-factor by enclosing a group of factors in a box), and they are compatible with standard block diagrams [43].

Returning to the DC algorithm, the simplest way to combine the Divide and Concur projections would be just to alternate between them—the so-called "alternating projections" algorithm. However, that is often a bad idea, as explained in the next section.

## 14 Traps in Alternating Projections

Let us denote the vector of all the values of the replicas in a constraint graph at iteration $t$ by $\mathbf{r}_t$, and the replica values obtained by applying the Divide projection to $\mathbf{r}_t$ to be $P_D(\mathbf{r}_t)$. Then the alternating-projections algorithm would iteratively apply the Divide projection $P_D$ and the Concur projection $P_C$; that is it would obtain $\mathbf{r}_{t+1}$ using the rule

$$\mathbf{r}_{t+1} = P_C(P_D(\mathbf{r}_t)). \tag{19}$$

The problem with alternating projections is that the algorithm can be trapped in a simple cycle, where the replica vector first satisfies the Divide constraints but not the Concur constraints, and then satisfies the Concur constraints but not the Divide constraints, and then goes right back to where it was before satisfying the Divide constraints.

Consider the somewhat contrived "normal" constraint graph [78] shown in Fig. 17. This constraint graph has two replicas, denoted $r_x$ and $r_y$, and two constraints: an equality constraint on the right corresponding to a variable node, and a constraint that the two replicas are either at point $A$, which is at $(r_x = 0, r_y = 0)$ or at point $B$, which is at $(r_x = 3, r_y = 1)$. We can consider the Divide projection to move the replica vector to the nearest of points $A$ and $B$, and the Concur projection to set $r_x$ and $r_y$ to their mean value.

The only solution that satisfies all the constraints is the point $A$, where $r_x = r_y = 0$. But let's see what happens when we start at some point near $B$ like point $D$ in Fig. 18. The Divide projection takes us to the nearest point of $A$ or $B$, which is $B$, then the Concur projection takes us to the nearest point on the diagonal line, which is $C$, then we go back to $B$, and so on. We never find the true solution at $A$.

## 15 Difference-Map Dynamics

To make progress, we need a way to turn pairs of points in replica space where the Divide constraints and Concur constraints come close, but do not intersect (like points $B$ and $C$

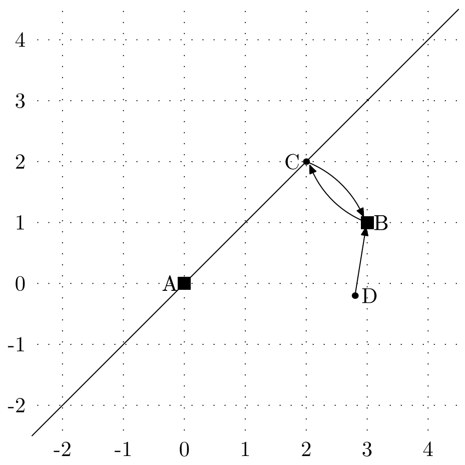


**Fig. 18** An example of a trap resulting from alternating projections. If one alternately projects to the nearest point that satisfies the constraint to be at $A$ or $B$, and then the nearest point where the replica values are equal (the *diagonal line*), one may be trapped in a short cycle ($B$ to $C$ to $B$ and so on) and never find the true solution at $A$

in Fig. 18) into repellers in the dynamics rather than traps. The "Difference-Map" (DM) dynamics does that [16]. We first consider a particular version of DM, where the replica update rule is

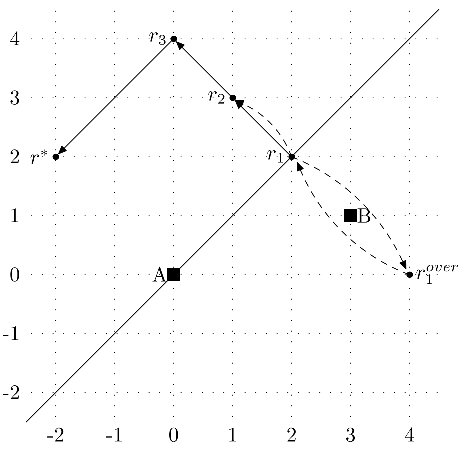
$$\mathbf{r}_{t+1} = P_C(\mathbf{r}_t + 2[P_D(\mathbf{r}_t) - \mathbf{r}_t]) - [P_D(\mathbf{r}_t) - \mathbf{r}_t]. \tag{20}$$

To parse this complicated looking equation, it is useful to think of the difference-map dynamics as breaking up into a three-step process [78]. The expression $[P_D(\mathbf{r}_t) - \mathbf{r}_t]$ represents the change to the current values of the replicas resulting from the Divide projection. In the first step, the values of the replicas move *twice* the desired amount indicated by the Divide projection. We can refer to these new values of the replicas as the "overshot" values $\mathbf{r}_t^{\text{over}} = \mathbf{r}_t + 2[P_D(\mathbf{r}_t) - \mathbf{r}_t]$. Next the Concur projection is applied to the overshot values to obtain the "concurred" values of the replicas $\mathbf{r}_t^{\text{conc}} = P_C(\mathbf{r}_t^{\text{over}})$. Finally the overshoot (that is, the extra motion in the first step) is subtracted from the result of the Concur projection to obtain the replica value for the next iteration $\mathbf{r}_{t+1} = \mathbf{r}_t^{\text{conc}} - [P_D(\mathbf{r}_t) - \mathbf{r}_t]$.

In Fig. 19 we return to our previous example to illustrate that the DM dynamics do not get stuck in a trap. Suppose that we now start initially at point $\mathbf{r}_1 = (2, 2)$. The Divide projection would take us to point $B$, but the overshoot takes us twice as far to $\mathbf{r}_1^{\text{over}} = (4, 0)$. The Concur projection takes us back to $\mathbf{r}_1^{\text{conc}} = (2, 2)$. Finally, the amount by which we overshot is subtracted so that $\mathbf{r}_2 = (1, 3)$. The next full iteration takes us to $\mathbf{r}_3 = (0, 4)$ (substeps are tabulated in Fig. 19). Now however, we are closer to $A$ then to $B$. Therefore, the next overshoot takes us to $\mathbf{r}_3^{\text{over}} = (0, -4)$, from which we would move to $\mathbf{r}_3^{\text{conc}} = (-2, -2)$, and $\mathbf{r}_4 = (-2, 2)$. Finally, at $\mathbf{r}_4$ we have reached a fixed point in the dynamics, because $\mathbf{r}_5 = \mathbf{r}_4$.

It can be proven that if a fixed point $\mathbf{r}^*$ in the DM dynamics is reached such that $\mathbf{r}_{t+1} = \mathbf{r}_t = \mathbf{r}^*$, then that fixed point must *correspond* to a solution $\mathbf{r}_{sol}$ that can be obtained using $\mathbf{r}_{sol} = P_D(\mathbf{r}^*)$. Thus, applying one more Divide projection to our fixed point, we arrive at our solution at $(0, 0)$.

Of course, the DM dynamics is not a panacea. It is possible for example for the replica vector to fall into a more complicated cycle and fail to find a fixed point. More generically, it is believed that when DM fails to converge, it typically follows chaotic dynamics. Empirically, though, it is now clear that the DM dynamics can effectively solve a great variety of problems (e.g. graph coloring, solving Diophantine equations, Sudoku, spin glass



| $t$ | $\boldsymbol{r}_t$ | $P_D(\boldsymbol{r}_t)$ | $\boldsymbol{r}_t^{over}$ | $\boldsymbol{r}_t^{conc}$ |
|---|---|---|---|---|
| 1 | $(2,2)$ | $(3,1)$ | $(4,0)$ | $(2,2)$ |
| 2 | $(1,3)$ | $(3,1)$ | $(5,-1)$ | $(2,2)$ |
| 3 | $(0,4)$ | $(0,0)$ | $(0,-4)$ | $(-2,-2)$ |
| 4 | $(-2,2)$ | $(0,0)$ | $(2,-2)$ | $(0,0)$ |
| 5 | $(-2,2)$ | | | |

**Fig. 19** An example showing how DM dynamics avoids traps. If we start at the point $r_1$, an alternating projections dynamics would be trapped between point $B$ and $r_1$, and never find the solution at $A$. DM dynamics will instead be repelled from the trap and move to $r_2$ (via the three sub-steps denoted with *dashed lines* $r_1^{\text{over}}$, $r_1^{\text{conc}} = r_1$, and $r_2$), then move to $r_3$, and then end at the fixed point $r_4 = r^*$, which corresponds to the solution at $A$

ground states, phase retrieval, random satisfiability, sphere packing, heteropolymer folding) that would be insoluble with the more naive alternating projections approach [16, 26].

It is natural to wonder to what extent the DM (20) can be modified, to perhaps improve convergence; for example can that funny-looking 2 that defines how much one overshoots be changed into a parameter? It turns out the value 2 is a good choice: smaller values and you don't always escape from a trap, larger ones and you start shooting away at an exponentially growing rate, which can cause problems.

However, you might also notice that (20) does not treat the Divide and Concur projections symmetrically. What if we swap the roles of $P_C$ and $P_D$? It turns out that works fine, typically about as well as the original version. So is there a parameterized version of DM that lets us move smoothly from the original version given by (20) to the version with $P_C$ and $P_D$ swapped? Such a parameterization has indeed been devised [16, 27]. For many problems a parameter value that gives a version of DM part-way between the original and the swapped version, but still rather close to either the original or swapped version, works best [26].

## 16 DC as a Message-passing Algorithm

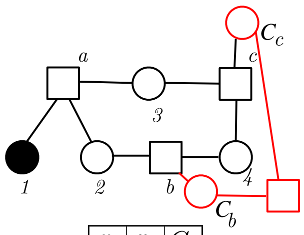
Nevertheless, the standard DM dynamics given by (20) has an important conceptual advantage—it makes it easier to see that DC is a message-passing algorithm closely analogous to BP [78]. Recall that each iteration of the standard DM dynamics can be interpreted as a three-step process: first overshoot, then concur, then correct.

The overshoot computation is done using the Divide projection at the constraint nodes. One can interpret the replica values before the overshoot as "messages" from the variable nodes to the factor nodes, and the resulting overshot values as messages from the factor nodes to the variable nodes.

The second step is to concur the overshot replica values, and make them equal at each variable node. This exactly parallels the BP step where one computes a belief at each variable from the incoming messages.

**Fig. 20** (Color online) A constraint graph derived from the factor graph shown in Fig. 2. The *red circles* represent new cost variable nodes corresponding to the soft cost functions in the original factor graph. The *red square* represents a global constraint on the maximum summed cost

| $x_1$ | $x_2$ | $x_3$ | $C_a$ |
|---|---|---|---|
| 0 | 0 | 0 | ∞ |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | ∞ |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | ∞ |
| 1 | 1 | 0 | ∞ |
| 1 | 1 | 1 | 0 |



| $x_2$ | $x_4$ | $C_b$ |
|---|---|---|
| 0 | 0 | 1.2 |
| 0 | 1 | 1.7 |
| 0 | 2 | 3.2 |
| 1 | 0 | 1.9 |
| 1 | 1 | 0.6 |
| 1 | 2 | 1.4 |

| $x_3$ | $x_4$ | $C_c$ |
|---|---|---|
| 0 | 0 | 0.4 |
| 0 | 1 | 1.9 |
| 0 | 2 | 0.2 |
| 1 | 0 | 4.9 |
| 1 | 1 | 0.3 |
| 1 | 2 | 2.4 |

Finally, the third step is to correct the concurred replica values neighboring variable nodes by subtracting the original overshoot. This parallels the BP step (see (6)) where one computes the messages from variable nodes to factor nodes by subtracting the incoming message from the belief.

To summarize, although the details of the update rules are different, the DC overshot replica values correspond to BP messages from factor nodes to variable nodes, the DC concurred replica values correspond to BP beliefs, and the DC corrected replica values correspond to BP messages from variable nodes to factor nodes [78].

## 17 DC for Optimization

We have seen how to use DC to solve *constraint satisfaction* problems; now I will show that *optimization* problems can be converted into constraint satisfaction problems. In some cases, this is relatively easy; if we know some conditions on the optimum configuration (e.g. the stationarity conditions) we can impose those as constraints.

Another possibility, which may be less elegant but which is always available, is to introduce "cost variables" corresponding to the cost functions of a factor graph. Consider for example the *factor* graph that we first introduce in Fig. 2, and the *constraint* graph derived from it shown in Fig. 20. For each soft local cost, we introduce a new variable corresponding to that cost, and modify the soft factor into a corresponding hard constraint. For example, for the constraint $b$ in the constraint graph illustrated, we require that if $x_2$ and $x_4$ equal 0, then the cost variable $C_b$ must equal 1.2.

All the cost variables can then be tied together in a new global hard constraint, which says that the sum of the cost variables must be less than some desired maximum cost. If we like, we can continually tighten the desired maximum cost in an outer loop on the algorithm.

This technique, which can generally convert an optimization problem into a constraint satisfaction problem to which DC can be applied, has been used to develop DC decoders for LDPC codes [26, 78], and DC algorithms that optimize heteropolymer energy functions [15]. Incidentally, similar ideas have long been used to convert optimization problems into decision problems in the theory of computational complexity [57].

## 18 Advantages and Disadvantages of DC Compared with BP

As an optimization or constraint satisfaction algorithm, DC presents several notable advantages compared with BP algorithms. First of all, as we have emphasized, DC has no difficulties dealing with continuous variables.

A second, less obvious, advantage is that DC, unlike BP, performs very well even when the hidden variables have no good “local evidence.” BP algorithms often converge to a fixed point with non-informative beliefs in the absence of local evidence.

Consider, for example, the problem of packing hard spheres (or other geometrical objects) as densely as possible into a finite volume such as the interior of a cube, for which DC is the state-of-the-art algorithm [34]. Because there is no local evidence saying that any particular sphere should be in a particular part of the volume, if one applied BP the messages sent from variables representing sphere centers to the constraints would start off generally non-informative, simply “saying” that the cost of being anywhere in the cube was equal. Those messages would be a fixed point of the dynamics: the BP algorithm would get no traction. Even starting with random messages, a BP algorithm for this problem would nearly inevitably converge to non-informative messages. DC on the other hand, is forced to make a single guess for each message at all times. Even starting with some initial random guesses for the positions of the sphere centers, it gradually works its way to a solution that satisfies all the hard constraints.

A third advantage of DC is that it is much easier to introduce complicated hard constraints, as might naturally arise in computer vision or control problems.

A fourth advantage of DC is that it cannot be trapped at a fixed point that is not a solution of the problem. BP, on the other hand, can converge to non-solutions, such as the “trapping sets” that cause “error floors” in BP decoders of some LDPC codes [59]. DC LDPC decoders do not get stuck in traps, as we would expect, but unfortunately they often decode to codewords that are less likely than the transmitted codeword, a failure mode not seen in BP [78]. Working from the idea of combining the advantages of BP and DC, my colleagues Yige Wang, Stark Draper and I recently proposed a hybrid “difference-map BP” decoder that heuristically imports a difference-map dynamics into a min-sum BP decoder. Difference-map BP has proven to significantly improve error floor performance compared to standard BP decoders, and also turns out to be closely related to the “splitting” BP algorithm [78].

DC also has some disadvantages in comparison with BP. Most significantly, it fundamentally only tracks a single value for each variable, so it cannot compute marginal probabilities like the sum-product algorithm does, or properly account for a probabilistic weighted sum over states.

DC also is often somewhat slow in converging to a solution. A related issue is that the convergence rate of DC depends crucially on the metric chosen. Even if we restrict ourselves to the natural Euclidean metric, it is very important how one scales the different variables in a problem (e.g. for a particular problem, it matters significantly whether one measures distances for a particular variable in units of millimeters or kilometers, and costs in units of dollars or cents). Currently, the best way to scale variables in DC is somewhat of a mystery.

Finally, unlike many versions of BP, DC is *not* guaranteed to give correct answers for constraint graphs or factor graphs without cycles. It does, however, have its own set of guarantees, which we turn to next.

## 19 Justifications for and History of DC

Let’s begin by considering when the alternating projections algorithm can be guaranteed to find a satisfying solution to a constraint satisfaction problem. Suppose that we have two constraints that we want to satisfy. In our case we consider the collection of Divide constraints to be one constraint, and the collection of Concur constraints to be the other constraint. Each of the two constraints can be considered to be sets of points in some space (in our case consider that to be the space of replica vectors).

A set of points is defined to be *convex* if the segment of points connecting any two points in the set is also contained in the set. It turns out (see the monograph by Bauschke and Combettes [3] for a full mathematical treatment of the topics discussed in this section) that if two sets of points are convex, alternately projecting from a point in one set to the nearest point in the other is guaranteed to converge a point in the intersection of the two sets. In that case, the alternating projections algorithm is also known as the "Projections onto Convex Sets" algorithm. It follows that if the sets of replica vectors that satisfy the Divide and Concur constraints are each convex, than the alternating projections algorithm will converge to a solution.

Notice however that the alternating projections algorithm is only guaranteed to be weakly convergent—the algorithm sometimes bounces from one convex set to the other, getting ever closer to the intersection of the two sets, but never actually reaching it in a finite number of iterations. Think for example, of the case when the two convex sets are each a line that intersects at a point.

Just as for alternating projections, one can prove that the DM projection dynamics also always converges to the intersection of two convex sets, and in some cases (but not always) it accelerates convergence compared to alternating projections. In fact, for some cases where alternating projections converges only weakly, DM converges in a finite number of iterations. So for the problem of finding the intersection of two convex sets, the DM is a handy alternative algorithm to alternating projections: it also always converges to a solution, and sometimes is much faster.

It is easy to show that the set of replica values satisfying the Concur constraints is automatically convex. It is also true that an intersection of convex sets is convex, which means that if all the sets of replica values satisfying the local Divide constraints are convex, the overall Divide constraint is also convex.

Historically, the "difference-map" dynamics given by (20) were first investigated in 1956 by Douglas and Rachford [13] as an improved iterative method for solving partial differential equations, and extended into a projection operator splitting method by Lions and Mercier in 1979 [42]. The surprisingly successful application of the DM algorithm to the *non-convex* phase retrieval problem [18] sparked the more recent investigations into why DM dynamics should also be useful for non-convex problems [4], their application to many other problems [16], and finally the formulation of the DC algorithm [27], which makes clear that the approach can be applied to general constraint satisfaction and optimization problems, albeit without the guarantees that obtain when it is applied to convex problems.

To summarize, just as BP algorithms converge to exact solutions on tree factor graphs but still can give very useful results for problems defined on general graphs, the DC algorithm converges to exact solutions when the constraint solution sets are convex, but can still give very useful results for problems defined using general constraints.

## 20 Conclusions and Future Research

In this paper, I have tried to present a tutorial to BP and DC algorithms, which will help introduce the reader to a very active area of current research. These algorithms turn out to have a similar message-passing structure which is ideally suited to parallelization.

Both sets of algorithms can be used to find good approximate solutions to a very wide set of problems, and exact solutions to a more limited, but still very important sub-set. In a general sense, I believe that an important goal of future research is to expand these sets of applications. For example, the set of problems that can be exactly solved using BP can be fruitfully expanded by using BDDs and ZDDs [37]. Many interesting convex problems can be solved exactly using DC or other projection-based algorithms; an important issue for future research is how efficient these algorithms are compared to more conventional approaches [7].

The scope for future research concerning approximate solutions is even greater. In particular, the ease with which DC algorithms deal with continuous variables and complicated constraints opens up many new possibilities. To take one example, control problems often involve optimizing functions of many continuous variables over complicated constraints. So while these problems have not heretofore been a good fit for BP algorithms, it is certainly conceivable that DC algorithms could be profitably used on such problems. A crucial and fundamental issue that could ultimately determine the success of such efforts will be whether we can learn to optimize DC algorithms so that they reach a solution as quickly as possible.

**Acknowledgements** I thank Veit Elser for many helpful discussions about the Divide and Concur algorithm, and Stark Draper, Yige Wang, Bill Freeman, and Yair Weiss for enjoyable collaborations on the subjects discussed here.

## References

1. Bahl, L., Cocke, J., Jelinek, F., Raviv, J.: Optimal decoding of linear codes for minimizing symbol error rate. IEEE Trans. Inf. Theory **20**, 284–287 (1974)
2. Barber, D.: Bayesian Reasoning and Machine Learning. Cambridge University Press, Cambridge (2011)
3. Bauschke, H.H., Combettes, P.L.: Convex Analysis and Monotone Operator Theory in Hilbert Spaces. Springer, Berlin (2011)
4. Bauschke, H.H., Combettes P.L., Luke D.R.: Phase retrieval, error reduction algorithm, and Fienup variants: a view from convex optimization. J. Opt. Soc. Am. A **19**, 1334–1345 (2002)
5. Baxter, R.J.: Exactly Solved Models in Statistical Mechanics. Academic Press, San Diego (1982)
6. Berrou, C., Glavieux, A., Thitimajshima, P.: Near Shannon limit error-correcting coding and decoding: turbo-codes. In: Proc. 1993 IEEE Int. Conf. on Comm., pp. 1064–1070 (1993)
7. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2004)
8. Boykov, Y., Veksler, O., Zabih, R.: Fast approximate energy minimisation via graph cuts. IEEE Trans. Pattern Anal. Mach. Intell. **29**, 1222–1239 (2001)
9. Braunstein, A., Mézard, M., Parisi, G.: Survey propagation: an algorithm for satisfiability. Random Struct. Algorithms **27**, 201–226 (2005)
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**, 677–691 (1986)
11. Chertkov, M., Chernyak, M.: Loop series for discrete statistical models on graphs. J. Stat. Mech. (2006) doi:10.1088/1742-5468/2006/06/P06009
12. Cormen, C.H., Leiserson, C.E., Rivest, R.L., Stein, C.: An Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009). Chap. 15
13. Douglas, J., Rachford, H.H.: On the numerical solution of heat conduction problems in two or three space variables. Trans. Am. Math. Soc. **82**, 421–439 (1956)
14. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)
15. Elser, V., Rankenburg, I.: Deconstructing the energy landscape: constraint-based algorithms for folding heteropolymers. Phys. Rev. E **73**, 026702 (2006)
16. Elser, V., Rankenburg, I., Thibault, P.: Searching with iterated maps. Proc. Natl. Acad. Sci. USA **104**, 418–423 (2007)
17. Felzenszwalb, P.F., Huttnelocher, D.P.: Efficient belief propagation for early vision. Int. J. Comput. Vis. **70**, 41–54 (2006)
18. Fienup, J.R.: Phase retrieval algorithms: a comparison. Appl. Opt. **21**, 2758–2769 (1982)

19. Forney, G.D.: Codes on graphs: normal realizations. IEEE Trans. Inf. Theory **47**, 520–548 (2001)
20. Forney, G.D.: The Viterbi algorithm. Proc. IEEE **61**, 268–278 (1973)
21. Freeman, W.T., Pasztor, E.C., Charmichael, O.T.: Learning low-level vision. Int. J. Comput. Vis. **40**, 25–47 (2000)
22. Gallager, R.G.: Information Theory and Reliable Communication. Wiley, New York (1968)
23. Gallager, R.G.: Low-Density Parity-Check Codes. MIT Press, Cambridge (1963)
24. Gamarnik, D., Shah, D., Wei, Y.: Belief propagation for min-cost network flow: convergence and correctness. In: Proc. of the 2010 ACM-SIAM Symp. on Discrete Algorithms (2010)
25. Geman, S., Geman, D.: Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. IEEE Trans. Pattern Anal. Mach. Intell. **6**, 721–741 (1984)
26. Gravel, S.: Using symmetries to solve asymmetric problems. Ph.D. dissertation, Cornell University, Ithica, NY (2009)
27. Gravel, S., Elser, V.: Divide and concur: a general approach to constraint satisfaction. Phys. Rev. E **78**, 036706 (2008)
28. Hershey, J.R., Rennie, S.J., Olsen, P.A., Kristjansson, T.T.: Super-human multi-talker speech recognition: a graphical modeling approach. Comput. Speech Lang. **24**, 45–66 (2010)
29. Ihler, A., McAllester, D.: Particle belief propagation. In: Proc. of the 12th Int. Conf. on Artificial Intelligence and Statistics (2009)
30. Ising, E.: A contribution to the theory of ferromagnetism. Z. Phys. **31**, 253 (1925)
31. Jelinek, F.: Statistical Methods for Speech Recognition. MIT Press, Cambridge (1997)
32. Johnson, J.K., Bickson, D., Dolev, D.: Fixing convergence of Gaussian belief propagation. In: Proc. Int. Symposium Inform. Theory (2009)
33. Jurafsky, D.: Martin, J.H: Speech and Language Processing. Prentice Hall, Upper Saddle River (2000)
34. Kallus, Y., Elser, V., Gravel, S.: Method for dense packing discovery. Phys. Rev. E **82**, 056707 (2010)
35. Kalman, R.E.: A new approach to linear filtering and prediction problems. J. Basic Eng. **82**, 35–45 (1960)
36. Kikuchi, R.: A theory of cooperative phenomena. Phys. Rev. **81**, 988–1003 (1951)
37. Knuth, D.E.: The Art of Computer Programming, vol. 4A. Combinatorial Algorithms. Addison-Wesley, New York (2011). Sect. 7.1.4
38. Koller, D., Friedman, N.: Probabilistic Graphical Models. MIT Press, Cambridge (2009)
39. Krauth, W.: Statistical Mechanics: Algorithms and Computations. Oxford University Press, Oxford (2006)
40. Kschischang, F.R., Frey, B.J., Loeliger, H.-A.: Factor graphs and the sum-product algorithm. IEEE Trans. Inf. Theory **47**, 498–519 (2001)
41. Lauritzen, S.L.: Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems. J. R. Stat.Soc., Ser. B **50**, 157–194 (1988)
42. Lions, P.-L., Mercier, B.: Splitting algorithms for the sum of two nonlinear operators. SIAM J. Numer. Anal. **16**, 964–979 (1979)
43. Loeliger, H.-A.: An introduction to factor graphs. IEEE Signal Proc. Mag., 28–41 (2004)
44. Loeliger, H.-A.: Least squares and Kalman filtering on Forney graphs. In: Blahut, R.E., Koetter, R. (eds.) Codes, Graphs, and Systems, pp. 113–135. Kluwer Academic, Norwell (2002)
45. Loeliger, H.-A., Dauwels, J., Hu, J., Korl, S., Ping, L., Kschischang, F.: The factor graph approach to model-based signal processing. Proc. IEEE **95**, 1295–1322 (2007)
46. MacKay, D.J.C.: Information Theory, Inference, and Learning Algorithms. Cambridge University Press, Cambridge (2003)
47. Malioutov, D.M., Johnson, J.K., Willsky, A.S.: Walk-sums and belief propagation in Gaussian graphical models. J. Mach. Learn. Res. **7**, 2031–2064 (2006)
48. Manning, C.D., Schutze, H.: Foundations of Statistical Natural Language Processing. MIT Press, Cambridge (1999)
49. McEliece, R.J., MacKay, D.J.C., Cheng, J.F.: Turbo decoding as an instance of Pearl's "belief propagation" algorithm. IEEE J. Sel. Areas Commun. **16**, 140–152 (1998)
50. Meltzer, T., Yanover, C., Weiss, Y.: Globally optimal solutions for energy minimization in stereo vision using reweighted belief propagation. In: Int. Conference on Computer Vision (2005)
51. Mézard, M., Montanari, A.: Information, Physics, and Computation. Oxford University Press, Oxford (2009)
52. Mézard, M., Parisi, G., Virasaro, M.A.: Spin Glass Theory and Beyond. World Scientific, Singapore (1987)
53. Mézard, M., Parisi, G., Zecchina, R.: Analytic and algorithmic solution of random satisfiability problems. Science **297**, 812–815 (2002)
54. Minato, S.-E.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. 30th ACM/IEEE Design Automation Conference, pp. 272–277 (1993)

55. Minka, T.P.: Expectation propagation for approximate Bayesian inference. In: Proc. of the 17th Conf. on Uncertainty in Artificial Intelligence, pp. 362–369 (2001)
56. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Francisco (1988)
57. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1993)
58. Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**, 257–286 (1989)
59. Richardson, T.: Error floors of LDPC codes. In: Proc. 41st Allerton Conf. Commun. Contr., Comput., Monticello, IL (2003)
60. Richardson, T., Urbanke, R.: Modern Coding Theory. Cambridge University Press, Cambridge (2008)
61. Ruozzi, N., Tatikonda, S.: Convergent and correct message passing schemes for optimization problems over graphical models. Available at http://arxiv.org/abs/1002.3239 (2010)
62. Rusmevichientong, P., Van Roy, B.: An analysis of belief propagation on the turbo decoding graph with Gaussian densities. IEEE Trans. Inf. Theory **47**, 745–765 (2001)
63. Russell, S., Norvig, P.: Artificial Intelligence, A Modern Approach, 3rd edn. Prentice Hall, Upper Saddle River (2009)
64. Ryan, W.E., Lin, S.: Channel Codes: Classical and Modern. Cambridge University Press, Cambridge (2009)
65. Sontag, D., Meltzer, T., Globerson, A., Jaakkola, T., Weiss, Y.: Tightening LP relaxations for MAP using message-passing. In: Uncertainty in Artificial Intelligence (2008)
66. Sudderth, E.B., Freeman, W.T.: Signal and Image processing with belief propagation. IEEE Signal Process. Mag. **25**, 114–141 (2008)
67. Sudderth, E.B., Ihler, A., Isard, M., Freeman, W.T., Willsky, A.S.: Non-parametric belief propagation. Commun. ACM **53**, 95–103 (2010)
68. Tanner, R.M.: A recursive approach to low complexity codes. IEEE Trans. Inf. Theory **27**, 533–547 (1981)
69. Vardy, A.: Trellis structure of codes. In: Pless, V.S., Huffman, W.C. (eds.) Handbook of Coding Theory, vol. 2, pp. 1989–2118. Elsevier, Amsterdam (1998)
70. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory **13**, 260–269 (1967)
71. Wainwright, M.J., Jaakkola, T., Willsky, A.S.: A new class of upper bounds on the log partition function. IEEE Trans. Inf. Theory **51**, 2313–2335 (2005)
72. Wainwright, M.J., Jaakkola, T., Willsky, A.S.: Tree-based reparametrization framework for analysis of sum-product and related algorithms. IEEE Trans. Inf. Theory **45**, 1120–1146 (2003)
73. Wainwright, M.J., Jaakkola, T., Willsky, A.S.: MAP estimation via agreement on (hyper)trees: message-passing and linear programming approaches. IEEE Trans. Inf. Theory **51**, 3697–3717 (2005)
74. Wainwright, M.J., Jordan, M.I.: Graphical models, exponential families, and variational inference. Faund. Trends Mach. Learn. **1**, 1–305 (2008)
75. Weiss, Y., Freeman, W.T.: On the optimality of the max-product belief propagation algorithm on arbitrary graphs. IEEE Trans. Inf. Theory **47**, 736–744 (2001)
76. Yedidia, J.S., Freeman, W.T., Weiss, Y.: Understanding belief propagation and its generalizations. In: Lakemeyer, G., Nebel, B. (eds.) Exploring Artificial Intelligence in the New Millenium, pp. 239–270. Morgan Kaufmann, San Francisco (2003)
77. Yedidia, J.S., Freeman, W.T., Weiss, Y.: Constructing free energy approximations and generalized belief propagation algorithms. IEEE Trans. Inf. Theory **51**, 2282–2312 (2005)
78. Yedidia, J.S., Wang, Y., Draper, S.C.: Divide and concur and difference-map BP decoders for LDPC codes. IEEE Trans. Inf. Theory **57**, 786–802 (2011)