Using Prior Knowledge to Improve Distributed Hill Climbing

Roger Mailler
SRI International
Artificial Intelligence Center
333 Ravenswood Ave., Menlo Park, CA 94025
mailler@ai.sri.com

Abstract

The Distributed Probabilistic Protocol (DPP) is a new, approximate algorithm for solving Distributed Constraint Satisfaction Problems (DCSPs) that exploits prior knowledge to improve the algorithm's convergence speed and efficiency. This protocol can most easily be thought of as a hybrid between the Distributed Breakout Algorithm (DBA) and the Distributed Stochastic Algorithm (DSA), because like DBA, agents exchange "improve" messages to control the search process, but like DSA, actually change their values based on a random probability. DPP improves upon these algorithms by having the agents exchange probability distributions that describe the likelihood of having particular "improve" values. These distributions can then be used by an agent to estimate the probability of having the best improve value among its neighbors or to compute the error caused by not informing other agents of changes to its improve value. This causes the protocol to use considerably fewer messages than both DBA and DSA, does not require a user to choose a randomness parameter like DSA, and allows DPP to more quickly converge onto good solutions. Overall, this protocol is empirically shown to be very competitive with both DSA and DBA.

1. Introduction

Distributed hill-climbing algorithms are a very powerful tool for solving numerous real-world problems including distributed scheduling, resource allocation, and distributed optimization. Currently, the two primary algorithms used to solve these problems are the Distributed Breakout Algorithm (DBA) [7] and the Distributed Stochastic Algorithm (DSA) [1, 8].

Both of these algorithms have been shown to be quite robust, and in fact DBA has been shown to take no more than $O(n^2)$ time to converge to a solution on acyclic graphs of size n [9]. However, both of these protocols possess some undesirable properties. DBA, for example, uses a strict locking mechanism to prevent agents from simultaneously changing their values. This mechanism slows the early convergence of the protocol and requires that each agent send a message to each of its neighbors at every step. DSA, on the other hand, requires the user to specify a parallelism parameter p that is used to control the likelihood that agents will change their values. The setting of p can have dramatic effects on the behavior of the protocol and can be quite problem specific. For instance, on very dense problems, having high values of p can cause the protocol to converge more quickly, but the same setting on a sparse problem will cause it to oscillate unnecessarily.

This paper presents a new algorithm that can best be described as a hybrid between the DSA and DBA protocols. The Distributed Probabilistic Protocol (DPP) employs a dynamically adjusted combination of control and randomness to allow the protocol to rapidly hill climb in the early stages of execution and then, in later stages, to control the search process in order to prevent unnecessary message passing and solution thrashing. This is done without requiring the user to specify any parameters at all and automatically adjusts to problems of different types.

These improvements over DSA and DBA are obtained by having the agents exchange improve value probability distributions with their neighbors. This allows the agents to compute how likely a neighbor is to change its value at each step without explicitly communicating. In addition, it allows each agent to determine when it would be most useful to tell another agent about a change to its local state. This means that, unlike the DBA protocol, DPP communicates *improve* messages only when it is deemed necessary and unlike the DSA protocol, allows an agent to individually adjust the likelihood of making a value change based on its problem and current state. Empirical testing of DPP has shown it to be competitive with both the DSA and DBA protocol while requiring substantially fewer messages.

Section 2 presents a formalization of the distributed constraint satisfaction problem that is used to describe and compare the three protocols throughout the rest of the paper. In Section 3, the DBA protocol is described and discussed.

Section 4 describes the DSA protocol. In Section 5, the DPP protocol is introduced, including an example of its execution on a simple problem. Section 6 discusses both the setup and results of empirical testing that has been done to compare the three protocols. Finally, the paper closes with some concluding remarks and some future directions for this work.

2. Distributed Constraint Satisfaction

A Constraint Satisfaction Problem (CSP) consists of the following [6]:

- a set of *n* variables $V = \{x_1, \ldots, x_n\}$
- discrete, finite domains for each of the variables $D = \{D_1, \ldots, D_n\}$
- a set of m constraints $R = \{R_1, \ldots, R_m\}$ where each $R_i(d_{i1}, \ldots, d_{ij})$ is a predicate on the Cartesian product $D_{i1} \times \cdots \times D_{ij}$ that returns true iff the value assignments of the variables satisfy the constraint

The problem is to find an assignment $A = \{d_1, \ldots, d_n | d_i \in D_i\}$ such that each of the constraints in R is satisfied. CSP has been shown to be NP-complete, making some form of search a necessity.

In the distributed case (DCSP), each agent is assigned one or more variables along with the constraints on those variables. The goal of each agent, from a local perspective, is to ensure that each of the constraints on its variables is satisfied. Clearly, each agent's goal is not independent of the goals of the other agents in the system. In fact, in all but the simplest cases, the goals of the agents are strongly interrelated. For example, in order for one agent to satisfy its local constraints, another agent, potentially not directly related through a constraint, may have to change the value of its variable.

In this paper, for the sake of clarity, each agent is assigned a single variable and is given knowledge of the constraints on that variable. Since each agent is assigned a single variable, the agent is referred to by the name of the variable it manages. Also, this paper considers only binary constraints that are of the form $R_i(x_{i1}, x_{i2})$. It is fairly easy to extend all the algorithms presented in this paper to handle more general problems where these restrictions are removed.

In addition, throughout this paper the word *neighbor* is used to refer to agents that share constraints. In other words, if an agent A has a constraint R_i that contains a variable owned by some other agent B, then agent A and agent B are considered neighbors.

```
when received (ok?, $x_j, d_j$) do
  if $mode$ == wait_improve
    add message to queue;
    return;
  else
    add $(x_j, d_j)$ to agent_view;
    when received ok? messages from all neighbors do
      send_improve;
      $mode \leftarrow$ wait_improve;
    end do;
  end if;
end do;

procedure send_improve
  $current\_eval \leftarrow$ evaluation value of $current\_value$;
  $improve_i \leftarrow$ possible maximum improvement;
  $new\_value \leftarrow$ the value which yields the best improvement;
  send (improve, $x_i$, $improve_i$, $current\_eval$) to neighbors;
end send_improve;
```

Figure 1. Procedures of the *wait_ok?* mode in Distributed Breakout.

3. Distributed Breakout Algorithm

The Distributed Breakout Algorithm (DBA) is a distributed adaptation of the centralized Breakout algorithm [3]. DBA works by alternating between two modes. The first mode (see Figure 1) is called the *wait_ok?* mode. During this mode, the agent collects *ok?* messages from each of its neighbors. Once this has happened, the agent calculates the best new value for its variable along with the improvement in its local evaluation. The agent then sends out an *improve* message to each of its neighbors and changes to the *wait_improve* mode.

In the *wait_improve* mode (see Figure 2), the agent collects *improve* messages from each of its neighbors. Once all the messages have been received, the agent checks to see if its improvement is the best among its neighbors. If it is, the agent changes its value to the new improved value. If the agent believes it is in a quasi-local-minimum (QLM), it increases the weights on all of its violated constraints. Finally, the agent sends *ok?* messages to each of its neighbors and changes back to the *wait_ok?* mode. The algorithm starts up with each agent sending *ok?* messages and going into the *wait_ok?* mode.

A QLM is a weaker condition than what is typically thought of as a local minimum. For an agent to be in a QLM it needs to have at least one violated constraint, have an improve value of 0, and have each of its neighbors report a 0 improve value as well. Although a QLM sometimes unnecessarily reports a minimum, it never fails to report one when it truly exists.

Because of the strict locking mechanism employed in the algorithm, the overall behavior of the agents is to simultaneously switch back and forth between the two modes. So, if one or more of the agents reacts slowly or messages are delayed, the neighboring agents wait for the correct mes-

```
when received (improve, $x_j, improve_j, eval$) do
  if $mode$ == wait_ok
    add message to queue;
    return;
  else
    record message;
    when received improve messages from all neighbors do
      send_ok;
      clear $agent\_view$;
      $mode \leftarrow$ wait_ok;
    end do;
  end if;
end do;

procedure send_ok
  if $improve_i$ is better than all of my neighbors
    $current\_value \leftarrow new\_value$;
  end if;
  when in a quasi-local-minimum do
    increase the weights on all violated constraints;
  end do;
  send (ok?, $x_i, current\_value$) to neighbors;
end send_ok;
```

Figure 2. Procedures of the *wait_improve* mode in Distributed Breakout.

```
procedure main
  while (not terminated) do
    update agent_view with incoming ok? $(x_j, d_j)$ messages;
    $new\_value \leftarrow$ choose_value;
    if $new\_value \neq d_i$ do
      $d_i \leftarrow new\_value$;
      send ((ok?, $(x_i, d_i)$) to all $x_j \in neighbors$;
    end if;
  end do;
end main;

procedure choose_value
  if $d_i$ has no conflicts do
    return $d_i$;
  $v \leftarrow$ the value with the least conflict ($v \neq d_i$);
  if $v$ has the same or fewer conflicts than $d_i$
     and $random < p$ do
    return $v$;
  else
    return $d_i$;
end choose_value;
```

Figure 3. Procedures of the DSA-B algorithm.

sage to arrive. This makes the protocol's communication usage very predictable because in each mode, each agent sends exactly one message to each of its neighbors, meaning that if there are m constraints, exactly $2m$ messages are transmitted during each step.

The locking mechanism is also very sensitive to message loss or to an agent failing. When this occurs, it can cause a cascading effect throughout the entire network of agents until a timeout condition is reached. In highly dynamic environments, this can become quite problematic unless the agents have some method of detecting the loss or failure, in which case they can use the DynDBA variant of the protocol [2].

The locking mechanism in DBA can be very beneficial because it does not allow neighboring agents to change their values at the same time, thus preventing thrashing. However, it can also prevent opportunities for additional parallelism because it limits the number of variables that can change at each *wait_improve* step to at most half when they are in a fully connected problem. These limitations effectively allow at most 25% of the variables to change during any individual step of the protocol's execution.

Two variants of the DBA protocol have been created to improve its overall parallelism and prevent pathological behavior by introducing randomness [9]. The weak-probabilistic DBA protocol (DBA-WP) uses randomness to break ties when two neighboring agents have the same improve value. The result is that either one agent, both agents, or neither of the agents change values when this situation occurs.

The strong-probabilistic DBA protocol (DBA-SP) attempts to improve parallelism by allowing agents to change their values with some probability when they can improve, but do not have the best improve value among their neighbors. This technique helps to improve parallelism because, in many situations, an agent's neighbor with the best improve value does not have the best improve value amongst its neighbors. This causes agents to wait unnecessarily for a neighbor to change when the neighbor has no intention of changing.

Overall, neither of these variants significantly improve the performance of DBA. However, they are able to escape certain types of worst-case situations where the strict locking mechanism causes infinite loops in the protocol's behavior.

4. Distributed Stochastic Algorithm

The Distributed Stochastic Algorithm (DSA) is one of a class of algorithms based on the idea that at each step, each variable should change to its best value with some small probability $p \in [0, 1]$. Because each variable changes with p probability, the likelihood of two neighbors changing at the same time is p^2, so as long as p is selected correctly, the protocol will hill climb to a better state.

The DSA algorithm has a number of variants. Figure 3 details the DSA-B variant. DSA-B follows the basic rule of DSA by changing values with probability p when it reduces the number of constraint violations, but also changes the value with p probability when the number of constraint violations is not improved. In this way, the DSA-B variant is able to escape certain types of local minima in the search space by making lateral moves.

The DSA protocol is quite popular because it is by far the easiest protocol to implement. However, it is also one of the hardest to tune because it requires the user to specify p.

```
procedure **main**
    **initialize**;
    **while** (not terminated) **do**
        update *agent_view* with ok? $(x_j, d_j)$ messages;
        update *agent_view* with improve $(x_j, improve_j)$ messages;
        **calculate_improve**;
        **send_ok**;
        **send_improve**;
    **end do**;
**end main**;

procedure **initialize**
    $pdf_i \leftarrow$ **calc_improve_pdf**; (see text)
    send ((**init**, $(x_i, d_i, pdf_i)$) to all $x_j \in neighbors$;
    **while** (not received **init** from $x_j \in neighbors$) **do**
        update *agent_view* with incoming init $(x_j, d_j, pdf_j)$ messages;
**end initialize**;
```

Figure 4. Main and initialize procedures of the DPP algorithm.

```
procedure **calculate_improve**
    $v \leftarrow$ the value with the least conflict $(v \neq d_i)$;
    **if** $d_i$ has no conflicts **or** $v$ has more conflicts than $d_i$**do**
        $new\_value = d_i$;
        $improve_i \leftarrow 0$;
    **else**
        $new\_value = v$;
        $improve_i \leftarrow$ difference in conflicts between $d_i$ and $v$;
    **end if**;
**end calculate_improve**;

procedure **send_ok**
    **if** $new\_value \neq d_i$ **do**
        $p \leftarrow$ **calc_change_probability**; (see text)
        **if** $random < p$
            $d_i \leftarrow new\_value$;
            send ((**ok?**, $(x_i, d_i)$) to all $x_j \in neighbors$;
            $improve_i \leftarrow 0$;
        **end if**;
    **end if**;
**end send_ok**;

procedure **send_improve**
    **for all** $x_j \in neighbors$ **do**
        $p \leftarrow$ **calc_improve_probability**$(x_j)$; (see text)
        **if** $random < p$
            send ((**improve**, $(x_i, improve_i)$) to $x_j$;
            store $(x_j, improve_i)$;
        **end if**;
    **end do**;
**end send_improve**;
```

Figure 5. Calculate_improve, send_ok, and send_improve procedures of the DPP algorithm.

The process of choosing this value can require a great deal of empirical testing because it is problem specific. Higher values of p cause the protocol to exhibit a rapid decrease in the number of constraint violations, which, depending on the problem, can level off far from an optimal solution. Lower values of p tend to correct violations more slowly, but often end up with a better solution in the end.

One of the greatest benefits of the DSA protocol is that it uses considerably fewer messages than the DBA protocol, because agents communicate only when they change their values. So, as the protocol executes and the number of violations is reduced, so is the number of messages.

5. Distributed Probabilistic Protocol

Conceptually, the Distributed Probabilistic Protocol (DPP) is a hybrid of the DSA and DBA protocols that aims to merge the benefits of both algorithms while correcting their weakness. The DPP protocol uses a dynamic mixture of randomness and direct control that changes based on the structure and current state of the problem to mitigate the effects of asynchrony. The key insight in this protocol is to understand that it is not always necessary for an agent to receive improve messages from all of its neighbors in order for it to determine that it is or is not the best agent to make a value change. Consider, as an example, an agent that has an improve value of 3 when each of its neighbors has a maximum possible improve value of 2.

By extending this idea to the next logical step, it can be seen that if an agent were to have a probability distribution (PDF) of the possible improves of its neighbors, then it could calculate the likelihood that it has the best improve value among its neighbors without communicating at all. If we were to use this likelihood as a basis for randomly determining when to change an agent's value, then we would end up with a DSA-like protocol where each agent's probability p_i is dictated by the improve distributions of its neighbors and its current improve value.

This process can be further enhanced by considering the use of explicit improve messages like those used in DBA. One way to do this would be to have agents communicate their improve values to their neighbors whenever they change. However, this creates an unnecessarily large communication overhead because many of the agents will not have actually changed their improve values since the last time they were sent.

Another, more interesting, use of the improve message is to consider it as a way to manage prediction errors in an agent's neighbor. In other words, if agent A sends a PDF to agent B that says, in all likelihood, that it will have an improve value of 1, and agent A actually has an improve value of 3, there is a certain probability that agent B will choose to do the wrong thing because it has inaccurate knowledge. However, if agent A sends an improve message that tells agent B that it has an improve value of 3, then the error is reduced considerably. Improve messages can, therefore, be sent out with a probability that is associated with the prediction error of an agent's neighbor.

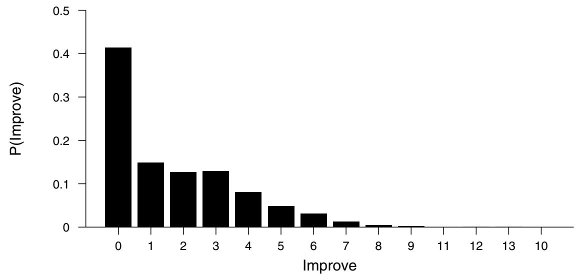

Figure 6. Example improve PDF of a variable with 13 neighbors in a 3-coloring problem.

5.1. The Algorithm

The core DPP algorithm can be seen in Figures 4 and 5. The algorithm starts, like DBA and DSA, by randomly selecting a value for the agent's variable. Unlike the other protocols, the DPP algorithm then calculates the improve PDF for its variable and sends an *init* message to its neighbors.

In general, it is very difficult to come up with a closed-form solution to calculating the improve PDF for a variable. Because of this, improve PDFs can be created by exhaustive enumeration or by employing some form of statistical sampling over the possible configuration space of the constraints on an agent's variable. Although for all the tests conducted in this paper exhaustive enumeration was used, it is believed that statistical sampling can be done without significantly altering the results of this work. An example of an improve PDF for a variable with 13 neighbors in a 3-coloring problem can be seen in Figure 6.

Another potentially promising way of calculating the improve value is to estimate it by using the PDF of the number of constraint violations which can often be computed in a computationally tractable manner. The logic behind this approach is that the number of constraint violations is strongly correlated to the improve value an agent can obtain (certainly acts as an upper bound). So, in principle, the one agent would always over-estimate the improve of its neighbors and vice-versa. This method was not implemented for this paper and is considered to be future work.

After sending the *init* messages, the agent waits for the *init* messages of its neighbors before proceeding to the main control loop. The main loop of DPP is very similar to that of DSA. During each loop, the agent updates its local state with any messages that have been received. It then determines what the next value of its variable should be based on this view and sends *ok?* messages to its neighbors if it decides to change its value. Unlike DSA, however, DPP may then send *improve* messages to one or more of its neighbors in an effort to reduce their prediction error.

The **calculate_improve** procedure of the protocol can be seen in Figure 5. This procedure is very similar to the way both DSA and DBA determine the best values and improve values for their variables. Like DSA-B, DPP will sometimes choose a new value for its variable when it does not actually improve the overall state of the problem. So, essentially, DPP has a lateral escape strategy like DSA.

It is conceivable that DPP could be modified to more closely resemble DBA by allowing the agents to change the weights of their constraints when they find that they have a high probability of being in a QLM. However, when the weights change, it also changes the improve PDF and subsequently increases the prediction error of the agent's neighbors. So, unless a new PDF is generated and sent, the result would be that the agent would need to send improve messages more frequently in order for the system to continue to operate correctly. This change is considered to be additional work that is planned for the future.

The **send_ok** procedure of the protocol is the next to be executed within the loop and can be seen in figure 5. The role of this procedure is to determine whether or not, given the agent's current information and its improve value, it should change its value. Like the DBA protocol, agents try not to change their values when they expect a neighbor with a higher improve value to change. The difference is that in DPP this is probabilistically determined based on the agent's neighbor's improve PDF and the current improve value of the agent. More information about how this is done can be found in Section 5.2.

The final procedure of the protocol, **send_improve**, can also been seen in figure 5. The goal of this procedure is to send out *improve* messages to the agent's neighbors with some probability based on the error created by not telling them. To do this, whenever an agent sends an improve value to another agent, it stores the last value that it has told that agent. This value is then used to compute the error caused by the difference between the current improve value and the one stored for the agent. Currently, improve messages are do not altered the improve PDF of stored by the receiving agent. It is conceivable that this could be done and is considered a future research direction for this work. More information about this calculation can be found in Section 5.2

The main loop continues until some predetermined termination condition is met. Like DSA, this is currently just a fixed number of executions of the loop. The termination condition could easily be changed to match DBA's termination condition or incorporate a quiescence detection algorithm such as the ones presented in [4] or [5].

5.2. Calculating Improve and Change Probabilities

Two of the most important components of the DPP protocol are how the improve and change probabilities are computed in the **calc_improve_probability** and **calc_change_probability** procedures. The **calc_improve_probability** procedure is used to determine the probability of an agent Y incorrectly determining that it has an improve value either less than or greater than X's. The equation for calculating this probability is given in equation 1.

$$P(improve_X^t = j | X = j \wedge improve_X^{t-1} = i) = |P(Y \leq j) - P(Y \leq i)| \quad (1)$$

This equation essentially says that the probability of sending an improve message for variable X at time t with an improve value of j given that the improve value of X is j and that the previous value that was sent to Y was i is the difference in the likelihood that the variable Y would change given j instead of i. This probability encapsulates the combined effects of agent Y committing a Type I (believing $improve_X^t < improve_Y^t$ when $improve_X^t > improve_Y^t$) or Type II error (believing $improve_X^t > improve_Y^t$ when $improve_X^t < improve_Y^t$) in decision making with regard to X.

In situations where no previous improve value has been sent to Y, the probability becomes more complicated because Y is estimating the improve value of X based on the improve PDF for X. Therefore, equation 1 becomes equation 2 where n is the largest value in the range of the improve value of X.

$$P(improve_X^t = j | X = j \wedge improve_X^{t-1} = null) = |P(Y \leq j) - \sum_{i=0}^{n} P(X = i) * P(Y \leq i)| \quad (2)$$

The **calc_change_probability** procedure is used to calculate the probability that the local variable X has the highest improve value among its neighbors. This is done by taking the product over all neighbors Y of equation 3.

$$P(Y \leq x | improve_Y^t = i) = \sum_{j=0}^{x} P(Y = j | improve_Y^t = i) \quad (3)$$

The $P(Y = j | improve_Y^t = i)$ can be calculated using Bayes theorem as in equation 4.

$$P(Y = j | improve_Y^t = i) = \frac{P(Y = j) * P(improve_Y^t = i | Y = j)}{P(improve_Y^t = i)} \quad (4)$$

The denominator of equation 4 can then be expanded to equation 5 using The Law of Total Probability.

$$P(improve_Y^t = i) = \sum_{k=0}^{n} P(Y = k) * P(improve_Y^t = i | Y = k) \quad (5)$$

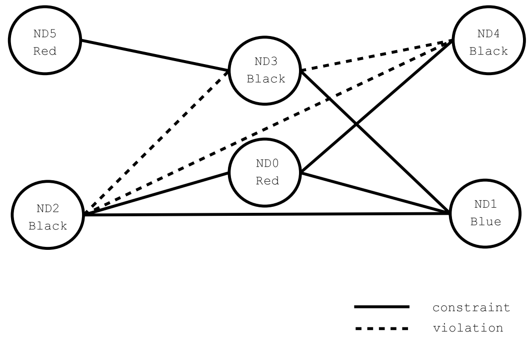

Figure 7. Example 3-coloring problem with six variables and nine constraints.

This results in the numerator and denominator of equation 4 being related to the conditioned probability $P(improve_Y^t = i | Y = k)$. This is the probability that agent Y would not communicate that its improve value has changed from i to k at time t. This actually can be expanded to equation 6 which is 1 minus the probability that Y will communicate about the change (see equation 1).

$$P(improve_Y^t = i | Y = k) = 1 - P(improve_Y^t = k | Y = k \wedge improve_Y^{t-1} = i) \quad (6)$$

This means that if agent X knows agent Y's improve PDF and agent Y behaves according to the protocol, agent X can accurately predict the probability that Y will have an improve value less than its own, even when Y has not sent X an improve message for a long period of time. Obviously, this is a very useful property of the protocol.

For efficiency reasons, these equations can be reduced considerably. Equations 3 and 4, for example, can be combined so that the denominator of equation 4 need be calculated only once in computing equation 3. For the sake of clarity, these reductions are not included in this paper.

5.3. Example

Consider the 3-coloring problem presented in Figure 7. In this problem, there are six agents, each with a variable and nine constraints between them. Because this is a 3-coloring problem, each variable can be assigned only one of the three available colors {Black, Red, or Blue}. The goal is to find an assignment of colors to the variables such that no two variables, connected by a constraint, have the same color.

In this example, three constraints are in violation: (ND2,ND3), (ND2,ND4), and (ND3,ND4). Following the protocol, at startup each of the agents computes the improve PDF for its variable and sends it along with its current value to all its neighbors. For example, ND0, ND1, and ND4 compute the following PDF:

- $P(improve_x = 0) = 0.52$
- $P(improve_x = 1) = 0.22$

- $P(improve_x = 2) = 0.22$
- $P(improve_x = 3) = 0.04$

ND2 and ND3, on the other hand, compute PDFs of

- $P(improve_x = 0) = 0.49$
- $P(improve_x = 1) = 0.25$
- $P(improve_x = 2) = 0.15$
- $P(improve_x = 3) = 0.10$
- $P(improve_x = 4) = 0.01$

After exchanging PDFs, each agent computes its best value and its associated improve value.

- ND0 has a 0 improve.
- ND1 has a 0 improve.
- ND2 has an improve of 1.
- ND3 has an improve of 1.
- ND4 has an improve of 2.
- ND5 has a 0 improve.

The agents then execute the **send_ok** procedures. ND4, for example, computes a change probability of 0.80 because its current improve value is 91% likely to be better than ND2's, 91% likely to be better than ND3's, and 97% likely to be better than ND0's. In this round, ND4 chooses to change its value to Blue and sends out *ok?* messages to its neighbors. ND4 then computes its improve probability with an improve value of 0. Since it has never sent an improve message to any of its neighbors, it assumes that they are using its prior improve PDF and so computes the following error probabilities:

- $P(error)$ for ND2 = 0.16
- $P(error)$ for ND3 = 0.16
- $P(error)$ for ND0 = 0.17

Through random chance, ND4 does not send any improve messages in this round. Also quite randomly, no other agent changes its value, until some time in the future when ND3 makes a lateral move to the color Red at which point ND5 changes to Black and the problem is solved.

6. Evaluation

To test the DPP protocol, the DSA, DBA, and DPP protocols were implemented in a distributed 3-coloring domain. The DSA algorithm was the DSA-B variant described in [8] with a $p = 0.60$, which was reported to be one of the best across a number of different trial settings.

The test series consisted of randomly generated graphs with $n = \{200, 400, 600\}$ variables and $m = \{2n, 4n\}$ constraints. For each setting of n and m, 30 problems were created and each of the protocols was run on them. Each algorithm was given 500 cycles of execution time. During a cycle, each agent was given the opportunity to process its incoming messages, change its value, and queue up messages for delivery during the next cycle. The actual amount of execution time per cycle varied depending on the cycle, the problem, and the protocol.

The algorithms were compared on two main factors. Namely, during each cycle, the number of current violations and the number of messages transmitted were measured. These values were

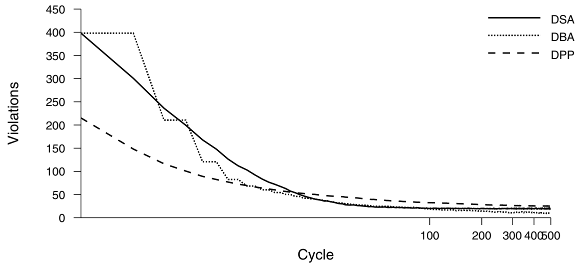

Figure 8. Anytime characteristics of DSA, DBA, and DPP for 600 variable 3-coloring problems with 1200 constraints.

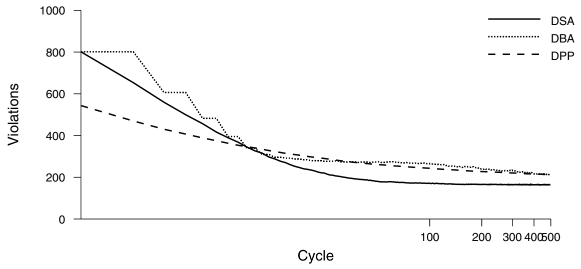

Figure 9. Anytime characteristics of DSA, DBA, and DPP for 600 variable 3-coloring problems with 2400 constraints.

used to plot the anytime curves of the protocol as well as measure the number of messages that are used in relation to the number of violations.

The results from the 600 variable tests can be seen in Figures 8, 9, 10, and 11. Overall, the anytime characteristics of DPP are quite good. At first, the protocol rapidly decreases the number of constraint violations, which tends to slow down after a short period of time. This behavior is to be expected from the protocol because as the number of violations decrease, so do the relative change probabilities for the agents. In addition, the current implementation of the protocol does not have a robust local minimum escape technique like the one used in DBA. It would be interesting to see how DPP would perform if it too were to adjust the weights of constraints to escape local minima.

Table 1 shows the average CPU time required by the most heavily loaded agent during each cycle for the various algorithms, a value referred to as that average parallel cycle time. As can be seen, DPP requires less time per cycle on average that either of the other algorithms. The most likely reason for this is that the cost of processing the extra messages in DSA and DBA tend to dominate their per cycle run time. This average value takes into account the cost of initially computing the improve PDFs in DPP which, at first is very dominate, but tends to be absorbed by the very fast cycles as the protocol progresses past the first cycle.

In the end, on both the easy and hard problems, DPP is com-

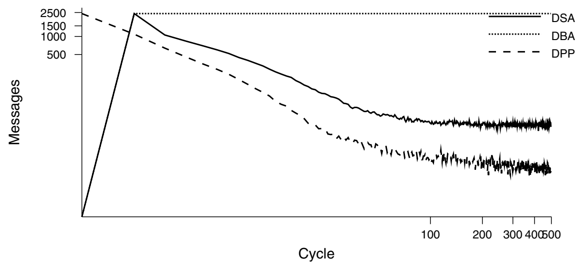

Figure 10. Message usage of DSA, DBA, and DPP for 600 variable 3-coloring problems with 1200 constraints.

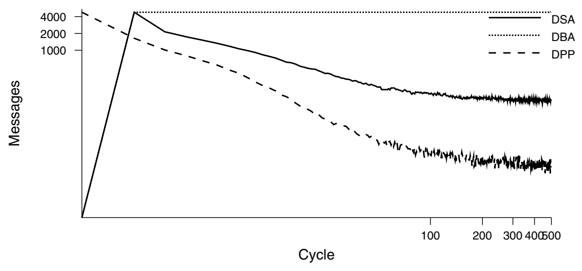

Figure 11. Message usage of DSA, DBA, and DPP for 600 variable 3-coloring problems with 2400 constraints.

petitive with both DBA and DSA even though it uses considerably fewer messages. This is a very encouraging result, especially if one of the limitations of the domain is network bandwidth. The results for the 200- and 400-variable problems were very similar to the results of the larger problems.

7. Conclusions

This paper presents a new protocol, called the Distributed Probabilistic Protocol (DPP), which uses a combination of randomness and control to rapidly converge on good solutions while using very few messages. The central idea behind this protocol is the exchange of improve probability distributions in order to circumvent unnecessary communication among the agents.

Overall, this protocol has a very strong, early hill-climbing nature that tends to taper off as the agents near a good solution. Empirically, DPP has been shown to be competitive with both the DSA and DBA protocols while using considerably fewer messages.

There is a considerable amount of future work to be done on this protocol. First and foremost, additional testing needs to be conducted to determine the algorithms' properties in a variety of domains. In addition, several improvements can be made to the protocol. These include adding DBA-like constraint weighting and termination detection mechanisms. The early testing on this protocol shows that it has a great deal of potential and is a substantial contribution to the state of the art.

Variables	Constraints	DSA	DBA	DPP
200	400	5.77	31.90	5.14
200	800	10.44	30.00	6.10
400	800	7.85	29.10	7.38
400	1600	11.70	33.98	7.83
600	1200	15.15	30.22	41.05
600	2400	12.60	9.73	10.03

Table 1. The average parallel cycle time (milliseconds) of DSA, DBA, and DPP on various problems.

References

[1] S. Fitzpatrick and L. Meertens. *Distributed Sensor Networks: A Multiagent Perspective*, chapter Distributed Coordination Through Anarchic Optimization, pages 257–294. Kluwer Academic Publishers, 2003.

[2] R. Mailler. Comparing two approaches to dynamic, distributed constraint satisfaction. In *Proceedings of Fourth International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2005)*, pages 1049–1056, July 2005.

[3] P. Morris. The breakout method for escaping local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 40–45, 1993.

[4] G. Tel. *Introduction to Distributed Alogrithms*. Cambridge University Press, 2000.

[5] M. Wellman and W. Walsh. Distributed quiescence detection in multiagent negotiation. In *In AAAI-99 Workshop on Negotiation: Settling Conflicts and Identifying Opportunities*, 1999.

[6] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.

[7] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *International Conference on Multi-Agent Systems (ICMAS)*, 1996.

[8] W. Zhang, G. Wang, and L. Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *Proceedings of the AAAI Workshop on Probabilistic Approaches in Search*, pages 53–59, 2002.

[9] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 352–357, 2002.