Adaptive and Non-Adaptive Distribution Functions for DSA

Melanie Smith
University of Tulsa
800 S. Tucker Drive
Tulsa, OK 74104
melanie@utulsa.edu

Sandip Sen
University of Tulsa
800 S. Tucker Drive
Tulsa, OK 74104
sandip-sen@utulsa.edu

Roger Mailler
University of Tulsa
800 S. Tucker Drive
Tulsa, OK 74104
mailler@utulsa.edu

ABSTRACT

Distributed hill-climbing algorithms are a powerful, practical technique for solving large Distributed Constraint Satisfaction Problems (DSCPs) such as distributed scheduling, resource allocation, and distributed optimization. Although incomplete, an ideal hill-climbing algorithm finds a solution that is very close to optimal while also minimizing the cost (i.e. the required bandwidth, processing cycles, etc.) of finding the solution. The Distributed Stochastic Algorithm (DSA) is a hill-climbing technique that works by having agents change their value with probability p when making that change will reduce the number of constraint violations. Traditionally, the value of p is constant, chosen by a developer at design time to be a value that works for the general case, meaning the algorithm does not change or learn over the time taken to find a solution. In this paper, we replace the constant value of p with different probability distribution functions in the context of solving graph-coloring problems to determine if DSA can be optimized when the probability values are agent-specific. We experiment with non-adaptive and adaptive distribution functions and evaluate our results based on the number of violations remaining in a solution and the total number of messages that were exchanged.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Algorithms, Design, Performance

Keywords

Distributed Constraint Satisfaction, Hill Climbing

1. INTRODUCTION

Distributed hill-climbing algorithms are very powerful tools for solving numerous real-world problems including distributed scheduling, resource allocation, and distributed optimization. These problems can be easily mapped to distributed constraint satisfaction, and like DSCPs, they must be solved using algorithms that can make decisions about how to best improve the global state of the problem from an agent's limited, local perspective. The ultimate goal of distributed constraint satisfaction is to find a solution, if one exists, while also minimizing the cost (i.e. the required bandwidth, processing cycles, etc.) [3, 7]. Complete algorithms, such as Asynchronous Weak Commitment (AWC) [13], Asynchronous Backtracking (ABT) [14], and Asynchronous Partial Overlay (APO) [6], are guaranteed to find a solution if one exists, but tend not to be very scalable. In practice, however, one must accept a close-enough solution, especially if the problem is large or the solution needs to be derived quickly. Hill-climbing algorithms tend to work very quickly even on large problems, but do not guarantee that they will find a solution if there is one. One of the most powerful algorithms from this class is the Distributed Stochastic Algorithm (DSA) [4, 16].

DSA is a hill-climbing technique that works by having agents change their value with probability p when making that change will reduce the number of constraint violations. DSA requires the user to specify p. Traditionally, the value of p is constant, chosen by a developer at design time to be a value that works for the general case, meaning the algorithm does not adapt its behavior based on the problem's characteristics. In fact, 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. Because p's value is so crucial to the success of finding a good solution, we believe choosing p to be more agent- and problem-specific will improve the solution and the process by which the solution is found.

In this paper, we investigate different probability functions for the Distributed Stochastic Algorithm in the context of solving graph-coloring problems. First, we examine four non-adaptive techniques that define p as a function of how much improvement an agent can have. The second set of techniques involves adaptation, where the function used to compute p is modified over time based on the agent's experiences.

Section 2 presents a formalization of the distributed constraint satisfaction problem that is used as the basis for this paper. Section 3 gives a detailed description of DSA. Sections 4 and 5 discuss our non-adaptive and adaptive ap-

.

proaches, and Section 6 discusses both the setup and results of empirical testing that has been done to compare these adaptations to DSA. Finally, the paper closes with some concluding remarks and some future directions for this work.

2. DISTRIBUTED CONSTRAINT SATISFACTION

A Distributed Constraint Satisfaction Problem (DCSP) consists of the following [14]:

- a set of n variables $V = \{x_1, \dots, x_n\}$
- a set of k Agents $A = \{a_1, \dots, a_k\}$
- discrete, finite domains for each of the variables $D = \{D_1, \dots, D_n\}$
- a set of m constraints $R = \{R_1, \dots, R_m\}$ where each $R_i(d_{i1}, \dots, d_{ij})$ 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, $S = \{d_1, \dots, d_n \mid d_i \in D_i\}$, such that each of the constraints in R is satisfied. DCSP, like its centralized counterpart, has been shown to be NP-complete, making some form of search a necessity [2].

In DCSP, each agent is assigned one or more variables along with 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. For each of the agents, achieving this 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, either by changing the algorithm or by changing the problem as done in [1].

Definition 1. *A binary CSP is a CSP where all of the constraints in R are of the form $R_i(x_{i1}, x_{i2})$.*

Definition 2. *The constraint graph of a binary CSP is a graph $G = < V, E >$ where V is the set of variables in the CSP and E is the set of edges representing the set of constraints in R (i.e. $R_i(x_{i1}, x_{i2}) \in R \Rightarrow (x_{i1}, x_{i2}) \in E$).*

Additionally, 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.

3. RELATED WORK

3.1 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 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. As long as p is selected correctly, the protocol will hill climb to a better state.

```
procedure main
  while (not terminated) do
    update agent_view with incoming
      ok? $(x_j, d_j)$ messages;
    new_value ← choose_value;
    if new_value $\neq d_i$ do
      $d_i$ ← 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 $d_i$;
  $v$ ← 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 1: The procedures of the DSA-B algorithm.

The DSA algorithm has a number of implementation variants. Figure 1 details DSA-B, which follows the basic rule of DSA by changing values with probability p when it reduces the number of constraint violations. However, it also changes the value with p probability when the number of constraint violations remains the same (i.e. its improve value is 0). 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. 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 can level off far from an optimal solution depending on the problem. 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 other protocols like the Distributed Breakout Algorithm [15] because agents communicate only when they change their values. As the protocol executes and the number of violations decrease, so do the number of messages. However, while DSA converges on a solution in a reasonable amount of time, finding a better solution in less time while using even fewer messages is important.

Manipulating DSA's probability variable allows the algorithm to vary its results. Some studies suggest that the most general value is about $p = 0.3$ [16]. However, these values are simple constants and do not change based on the state of the problem.

3.2 Distributed Breakout Algorithm

The Distributed Breakout Algorithm (DBA) [15] is a dis-

```
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 all ok? messages 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 best value ;
  send (improve, $x_i$, $improve_i$, $current\_eval$);
end send_improve;
```

Figure 2: The procedures of the *wait_ok?* mode in Distributed Breakout.

```
when received (improve, $x_j, improve_j, eval$) do
  if $mode$ == wait_ok
    add message to queue;
    return;
  else
    record message;
    when received all improve messages 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 3: The procedures of the *wait_improve* mode in Distributed Breakout.

tributed adaptation of the Centralized Breakout Algorithm [9]. DBA works by alternating between two modes. The first mode (see figure 2) is called the *wait_ok?* mode where 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 3), the agent collects *improve?* messages from each of its neighbors. Once all of the messages have been received, the agent checks to see if its improvement is the best among its neighbors. If it is, it changes its value to the new improved value. If the agent believes it is in a quasi-local-minimum (QLM), it increase the weights on all its of 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.

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 message 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. Thus, if there are m constraints, exactly $2m$ messages are transmitted during each step.

Conversely, the locking mechanism in DBA can be very beneficial because it does not allow neighboring agents to change their values at the same time, which prevents oscillations. 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 1/4 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 [17]. 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, 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 value with some probability when they can improve, but don't have the best improve among their neighbors. This technique helps to improve parallelism because in many situations, the agent's neighbor with the best improve doesn't have the best improve among its neighbors. This causes agents to wait unnecessarily for their neighbor to change when their neighbor has no intention of actually doing so.

3.3 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 that inspired the creation of the protocol is that an agent doesn't necessarily need 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.

DPP works by having agents exchange probability distributions (PDF) that describe the likelihood they are going to have a particular improve value given the configuration of the constraints on their variable(s). This allows the agents to estimate the likelihood that it has the best improve value among its neighbors without communicating at all. Using this likelihood as a basis for randomly determining when to change an agent's value, we 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 is further enhanced by considering the use of explicit improve messages like those used in DBA. Unlike DBA, DPP sends out improve messages with a probability

```
procedure **choose_value**
  **if** $d_i$ has no conflicts **do**
    **return** $d_i$;
  $v \leftarrow$ the value with the least conflict ($v \neq d_i$);
  $improve \leftarrow d_i - v$;
  $maxImprove \leftarrow$ number of neighbors for agent;
  $p \leftarrow P(improve, maxImprove)$;   (see text)
  **if** $improve \geq 0$ **and** $random < p$ **do**
    **return** $v$;
  **else**
    **return** $d_i$;
**end choose_value**;
```

Figure 4: The choose_value procedure of the Non-Adaptive DSA-B algorithm.

that is associated with its estimate of a neighbor having a prediction error of its improve value. This means that if agent X knows agent Y's improve PDF and agent Y behaves according to the protocol, agent X can even 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.

As a result of these modifications, DPP uses considerably fewer messages than both DSA and DBA, does not require a user to define p values as in DSA, and more quickly converges onto good solutions. The drawback to DPP is that calculating the initial PDF function can be very difficult because it often does not have a closed-form solution. Because of this, improve PDFs are 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, both of which cause a steep overhead when starting up the algorithm.

4. NON-ADAPTIVE DSA

DPP's inspiration was that the probability of an agent changing its value should be associated with how much improvement it expects to have. Similarly motivated, we decided to investigate versions of the DSA protocol that determine the value of p as a function of the current improve value for an agent. In the non-adaptive version of the protocol, we altered the DSA-B algorithm to update p based on a function, as shown in Figure 4. The algorithm uses the same **main** procedure as DSA, but changes the **choose_value** procedure to calculate an *improve* value and uses that value to determine p. Like DBA and DPP, the improve value for a variable is simply the difference between the current number of conflicts and the number of conflicts for the best possible value. The *maxImprove* value, which is used to normalize the functions that calculate p, is the maximum total cost of all of the variable's constraints. This assumes that the maximum improvement occurs when all constraints are in conflict and changing the variable's value causes all the conflicts to be resolved. The value of p gets returned by the $P(improve, maxImprove)$ function, and depending on whether an improvement can be made, the agent's value is changed with probability p.

For our tests, we tried four non-adaptive functions to compute p: linear, sub-linear, super-linear, and Weibull. Figure 5 shows a graph of these non-adaptive P functions. We initially chose the linear function on the basis that there should be a higher probability of changing values if there is a higher

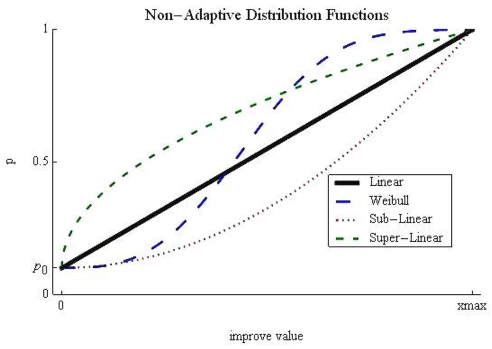


Figure 5: Non-Adaptive Distribution Functions: Linear, Sub-Linear, Super-Linear, and Weibull Distributions

improve value associated with that move. The other three functions were chosen to examine whether variations on the linear function would be better suited than using the simple linear function.

4.1 Distribution Functions

For our first function, we looked at a simple linear function on the amount of improve. Basically, when the linear function has a positive slope and a node has a high improve value (i.e. changing colors would allow a large number of conflicts to be abated), there is a good chance that the color will change. Using a normalized linear distribution instead of a constant value for p allows the change probability to be higher the more an agent can improve. We set $p_0 = 0.1$ to give some probability of change for the case where lateral movement occurs. As a reminder, lateral movements occur when multiple values have an improve of 0 and the agent can switch between the values without violating any constraints. The linear technique forms the basis of all the other functions we evaluate in this paper. The normalized linear function is as follows:

$$P(imp, maxImp) = (1 - p_0)\left(\frac{imp}{maxImp}\right) + p_0 \qquad (1)$$

A sub-linear function is similar to a linear one, but keeps the probably of the agent changing its value low until a sufficiently large improvement value is likely. Below is the sub-linear function that we used in this paper:

$$P(imp, maxImp) = (1 - p_0)\left(\frac{imp}{maxImp}\right)^2 + p_0 \qquad (2)$$

The opposite of sub-linear is super-linear. This function has the characteristic that the agents have a higher-than-linear likelihood of changing their value, except in the cases where they expect 0 or maximum improve. The following is the super-linear function we use:

$$P(imp, maxImp) = (1 - p_0)\sqrt{\frac{imp}{maxImp}} + p_0 \qquad (3)$$

The Weibull distribution, part of the family of Sigmoid functions, is a combination of the sub- and super-linear cases, acting sub-linear until the amount of improvement is

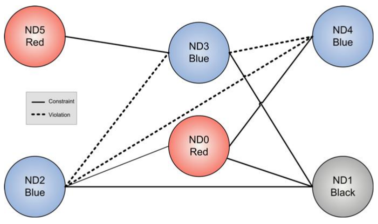


Figure 6: Example 3-coloring problem with six variables and nine constraints.

approximately half of the maximum possible improvement and then switching into a super-linear function. This means for small improvements, the likelihood of an agent changing its value is slim, but for large improvements, the likelihood is quite high. Below is the function used for our tests:

$$P(imp, maxImp) = (1-p_0)\left(1 - e^{-\left(\frac{imp}{0.5\times maxImp}\right)^{maxImp}}\right) + p_0 \tag{4}$$

4.2 Example

Consider the 3-coloring problem presented in Figure 6. 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 sends its current value to all its neighbors in an ok? message. After receiving all the messages, each agent determines their *improve*, *maxImprove*, and p values. In this example, p is calculated using equation 1.

- ND0 has improve = 0, maxImprove = 3, and p = 0.1
- ND1 has improve = 0, maxImprove = 3, and p = 0.1
- ND2 has improve = 1, maxImprove = 4, and p = 0.325
- ND3 has improve = 1, maxImprove = 4, and p = 0.325
- ND4 has improve = 2, maxImprove = 3, and p = 0.7
- ND5 has improve = 0, maxImprove = 1, and p = 0.1

After finding the probability of change, a random number is generated such that if the random number is less than p, the value for the node actually changes. Every node that changes sends a message to all its neighbors and the process starts over again until execution ends.

5. ADAPTIVE DSA

In addition to looking at non-constant, although static functions for determining the value of p, we also investigated two methods that allow the function to adapt based on experience. The learning problem that the agents encounter in this algorithm is to learn a mapping from their improve value at a time t to a probability that determines whether they should be the one that changes their value at t. In situations where this leads to an action being produced, the agent is rewarded based on the relative goodness of the action (i.e. how much improvement is actually made).

Like most learning methods, including Temporal difference (TD) learning [10], the general form of the update we use in this paper can be seen in equation 5. Basically there is an error term calculated at each time step that is used to move the probability by some small amount, α, toward the correct value.

5.1 Algorithm

The modifications needed to support the ability to adapt the function used to determine p are fairly simple. First, we introduce two global variables, a probability array, *prob*, of size *maxImprove* that holds the function values for each improve value and an integer *predImprove* that holds the predicted improvement value of the agent from the previous cycle. While still operating like the DSA algorithm, we alter the **main** procedure to initialize *maxImprove*, *prob*, and *lastPred*. Each time the main loop cycles, we save the predicted improve from the previous cycle so that we can compare the actual change to what was predicted. To find the actual change (*actualImprove*), we count the number of conflicts before and after the messages are processed and take the difference. The probability array is updated if the last predicted value is greater than 0, meaning we made a change to our variable's value on the last cycle. Each of our approaches introduces a new **update_prob** method that does the probability array updating.

The **choose_value** procedure is also changed in the adaptive algorithm, although minimally. We initialize *predImprove* to -1 every time the method is called to indicate that no change is made. If a change is made, *predImprove* is set to the *improve* value. This value is what is saved to compare in the next iteration to the actual improve of the agent and trigger the **update_prob** procedure call.

5.2 Update Methods

The *discrete update* function limits the impact that the error value has to only correct the probablity assocated with the agent's last predication. To initialize the *prob* array, we calculate the linear value for each unit using equation 1. As the problem is solved, the probabilities are changed to reflect whether the decision made by the agent was a good decision.

After each cycle, the probability array is updated by taking into account the current value of p and adding a fraction of the difference between the actual and predicted improve values. The **update_prob** procedure for the discrete algorithm takes as input the actual improve value, the last predicted improvement, and the maximum improvement possible for the particular agent. The procedure changes the global *prob* array by changing the $prob_{pred}$ to increase (or decrease) by a constant, α, times the normalized difference in the predicted value and the actual value at time t. None of the other values in the *prob* array are affected, and if no change is made to the agent's value, the $prob_{pred}$ value will not change. In our evaluation, we set $\alpha = 0.3$.

$$prob_{pred}^{t} \leftarrow prob_{pred}^{t-1} + \alpha\left(\frac{actual - pred}{max}\right) \tag{5}$$

The *exponential decay update* technique updates to the *prob* array just like in the discrete method, but also updates other values in the array based on an exponential decay. Again, the array is initialized using the linear function (equation 1) to initialize the probability array.

The **update_prob** procedure for the exponential decay update algorithm contains a loop that iterates through the *prob* array, adjusting each value a slight amount based on how far from the *pred* position it is. For example, if the *pred* is 4, and i is 2, then the value for $prob_i$ will be adjusted by a factor of α^3, as will the value of $prob_6$ because it is the same distance from the center at 4. The further away from one another i and *pred* are, the smaller the factor, and the smaller the change in the probability. One could easily think of this updating technique as being similar in nature to a radial basis function [8] with each function centered at an individual improve value. As you adjust one of the kernels, it affects the probabilities in an exponentially decaying manner based on its distance from the prediction.

$$prob_i^t \leftarrow prob_i^{t-1} + \alpha^{(|pred-i|+1)} \left(\frac{actual - pred}{max} \right) \qquad (6)$$

6. EVALUATION

To test our DSA variants, we implemented each in a distributed 3-coloring domain. The DSA algorithm was the DSA-B variant and the test series consisted of randomly generated graphs with $n = \{100, 200, 300, 400, 500\}$ variables and $m = \{2.0n, 2.3n, 2.7n\}$ constraint densities to cover under-constrained, normally constrained, and over-constrained environments. For each setting of n and m, 30 problems were created and each of the probability distributions were used, both adaptive and non-adaptive. Each run 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 distribution function.

The test cases were compared on two main factors. During each cycle, the number of current violations and the number of messages transmitted were measured. These values were used to plot the graphs shown in Figures 7 and 10. Although not shown here, for the adaptive cases, we kept track of the probability array values for each cycle to see how the probability functions were affected over time. As expected, the exponential decay update technique changes more often and to a greater degree because more values in *prob* change during each cycle. One thing we noticed is that even though the values changed more, they still didn't change very much from their initial values. In future work, we plan to use the resulting function from one run and using it as the input to the next run so that each successive run would be improving the final function instead of starting over with the linear values. This would make it work much more like classical reinforcement learning because the agents would get multiple trials in addition to multiple updates.

6.1 Total Messages Received

In terms of total messages sent/received, all of our algorithms used less than half of the number of messages that traditional DSA (see Figure 7) uses for all combinations of nodes and edge densities. This seems to indicate that adapting the p values to be more situation specific facilitated significantly more effective communication between agents. Thus, it is our conjecture that it will require half the transmission bandwidth and allow for a more scalable solution.

In comparison to DPP, based on the results presented in [5], the amount of messaging is about the same. However, using any of our distribution functions alleviates the need to calculate the initial PDF function. We plan to do a more empirical comparison between DPP and our DSA variant in the future.

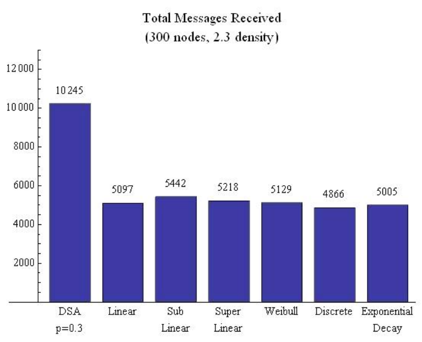


Figure 7: Number of messages sent for all algorithms at 500 nodes and 2.3 edge density.

6.2 Total Conflicts

Even though our approach dramatically improves the communication cost, none of our alterations to DSA showed consistent improvement to the solution found by normal DSA as far as the total conflicts are concerned. Table 1 shows the conflicts remaining for 2.3 density and all nodes after 500 cycles. Out of our six probability functions, a clear leader did not emerge, although they were within a standard deviation of one another and DSA.

One possible reason for finding a slightly worse solution is that when traditional DSA has an agent with only a small or no improvement possible, the probability of having it change values is still fairly high at $p = 0.3$, whereas with our experiments, we set $p_0 = 0.1$, which is significantly lower than traditional DSA. The probability of an agent having only a small or no amount of improve for any particular time cycle is also fairly high, meaning that the 0.2 difference in initial probability values is likely a significant factor. In many cases, a probability function that has a high probability of change in the early part of the run is more likely to do well overall because the agent has the chance to hill-climb to a better state. In the cases where the starting value is too low, the agent may not have had the opportunity to find a better solution because it has already hill climbed into such a bad state that there is no escape.

6.3 Further Analysis & Experimentation

To determine if the difference in p_0 is the cause of our solutions coming out with slightly more constraint violations, we ran traditional DSA again with $p = 0.1$ instead of $p = 0.3$. In this case, the solutions ended up consistently worse with the lower value of p because there is a smaller likelihood that an agent will change its value no matter how much it can improve. Our probability functions result in conflict curves over time that fall between the traditional DSA curves for $p = 0.1$ and $p = 0.3$. This implies that there are two separate components at play in finding good solutions with DSA-B: lateral movement and hill-climbing.

To test this hypothesis, we augmented the traditional DSA algorithm to give $p_0 = 0.1$ and $p_{improve \geq 1} = 0.3$ to segregate the approach into lateral movement and hill-climbing portions. Figure 8 shows all three traditional DSA algorithms

Table 1: Remaining Conflicts after 500 cycles for 2.3 density

Algorithm	100 nodes	200 nodes	300 nodes	400 nodes	500 nodes
DSA	6.1	12	15.9	22.5	28.6
Linear	6.0	12.8	18.4	24.8	30.7
Sub-Linear	7.1	12.5	20.1	24.9	30.2
Super-Linear	6.6	12.8	19.1	24.7	29.4
Weibull	6.9	12	18.6	23.6	30.8
Discrete	5.9	11.9	17.5	24	29.5
Contextual Discrete	6.2	12.4	17.9	24.3	31.0

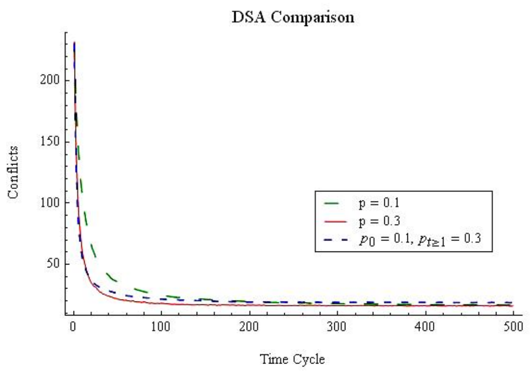


Figure 8: Conflicts over time for DSA with $p = 0.1, 0.3$ and DBH with $p_0 = 0.1$ and $p_{1...i} = 0.3$.

with different static values for p at 300 nodes and 2.3 density. The higher the value for p_0, the faster the number of remaining conflicts falls due to the higher probability of lateral motion. We also notice that the higher the number of nodes, the more of an impact the lateral motion has. In Figure 9, we show the average distribution of the *improve* value for an agent with 4 neighbors. Because $improve = 0$ occurs more frequently than larger improve values, we know that lateral movement plays a large part in finding a good solution. This isn't entirely surprising as it has been reported numerous times that randomness in centralized hill-climbing searches has a fairly significant impact on the overall solution quality [11].

Examining the hill-climbing portion of the runs, we look at the slope of the conflict lines in Figure 10. The thicker line is DSA with $p = 0.3$, and the other lines are our non-adaptive and adaptive results. The slope of each line indicates the effectiveness of the hill-climbing part of the algorithm. Traditional DSA flattens out as time goes on with little to no slope while our algorithms have a more defined slope as time progresses. We believe that this indicates that our hill-climbing methods are more effective than normal DSA, but that our choice of p for the lateral movement case was sub-optimal. Our adaptive algorithms have a steeper slope than our non-adaptive algorithms, indicating that having the values for p evolve as the problem is solved improves the hill-climbing method over finding p based on a static function. In addition, using static functions results in a more defined slope than having p defined as a static constant. In future work, we will explore this discovery and experiment with different p_0 values over each of our distribution functions.

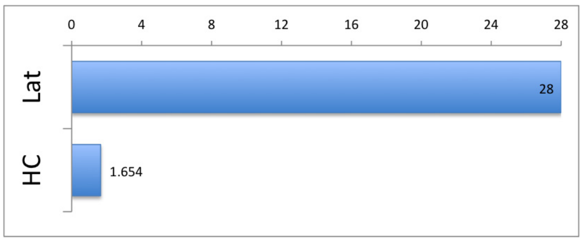


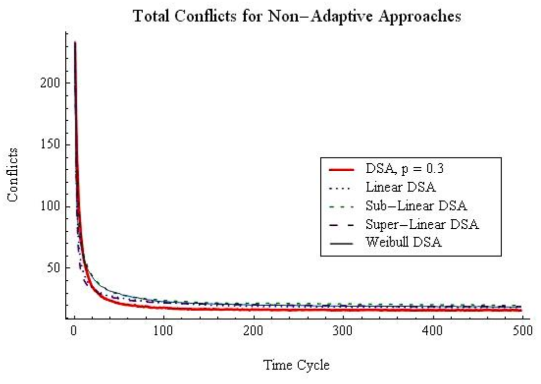
Figure 9: Average number of Lateral Moves (Lat) vs. Hill-Climbing Moves (HC) for graph-coloring agents with four neighbors.

7. CONCLUSION

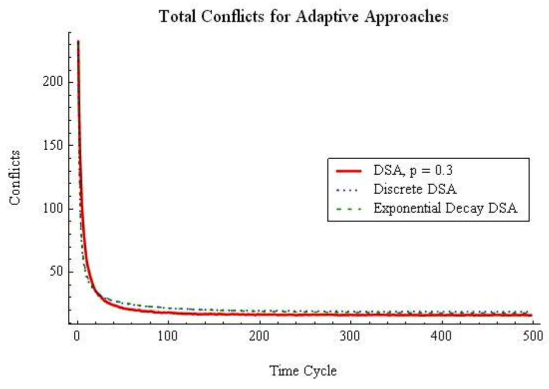
This paper presents different adaptive and non-adaptive alterations to the Distributed Stochastic Algorithm (DSA), which turns the traditional constant p into an situation-specific function for p based on the predicted improvement of the agent during a time cycle. By allowing p to change and adapt, we reduce the number of messages needed to communicate between agents by more than half. In discovering the heavy influence of lateral movement, we believe that given a more optimal p_0 value, our DSA variants should improve even further. Other work we have done includes experimenting with fine-tuning the lateral movement probability [12].

As agent systems become more complex and start to evolve more autonomously, and as this extends into complex software systems, cutting communication costs (i.e. bandwidth and throughput needs) may become more desirable than finding a more optimal solution if the difference is within a tolerable range. In cases like these, using any of our techniques would make a dramatic impact on the networking footprint required by traditional DSA, even without an optimal p_0 probability for lateral movement.

Because of the significant reduction in messaging, we believe fine-tuning the lateral movement probability in our distribution functions can find more optimal solutions. Running more tests in a variety of domains would help determine how the algorithms adapt to more than just graph-coloring problems. Also testing the adaptive approaches using different initialization vectors may result in finding a more optimal probability function. We are also planning on incorporating more complex machine learning, where each successive run takes the probability function from previous runs as the initial value, allowing the simulation to improve upon a function that starts off in a more optimal state than the static functions we used in this paper.



(a)



(b)

Figure 10: Conflicts Remaining Over Time: (a) Non-Adaptive and (b) Adaptive

8. ACKNOWLEDGMENTS

The authors gratefully acknowledge support of the Defense Advanced Research Projects Agency under DARPA grants HR0011-07-C-0060. Views and conclusions contained in this document are those of the authors and do not necessarily represent the official opinion or policies, either expressed or implied of the US government or of DARPA.

9. REFERENCES

[1] F. Bacchus and P. van Beek. On the conversion between non-binary constraint satisfaction problems. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 311–318, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[2] A. Bulatov, A. Krokhin, and P. Jeavons. The complexity of maximal constraint languages. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 667–674, New York, NY, USA, 2001. ACM.

[3] B. Faltings. Chapter 20 distributed constraint programming. In P. v. B. Francesca Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 699 – 729. Elsevier, 2006.

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

[5] R. Mailler. Using prior knowledge to improve distributed hill climbing. In *Proceedings of the 2006 International Conference on Intelligent Agent Technology (IAT 2006)*, 2006.

[6] R. Mailler and V. Lesser. Using Cooperative Mediation to Solve Distributed Constraint Satisfaction Problems. *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.

[7] A. Meisels. *Distributed search by constrained agents: algorithms, performance, communication.* Springer, 2008.

[8] J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. *Neural Comput.*, 1(2):281–294, 1989.

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

[10] A. G. B. Richard S. Sutton. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, Massachusetts, 1999.

[11] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, 1994.

[12] M. Smith and R. Mailler. Getting What You Pay For: Is Exploration in Distributed Hill Climbing Really Worth It? *Int'l Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, 2010.

[13] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP-95)*, Lecture Notes in Computer Science 976, pages 88–102. Springer-Verlag, 1995.

[14] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

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

[16] 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.

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