DRATS:
Dynamically Re-Allocated Team Search

A Thesis presented

by

Andrew Garrod Bosworth

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 5, 2004

Contents

Chapter 1

Introduction

Throughout computer science there are a number of "really hard problems" that require expensive searches to solve [7, 11, 22, 25]. These problems are so common that better methods of solving them are always in demand. As a result, there has been a great deal of research into developing efficient algorithms and data structures tailored to the set of NP-complete problems [17, 12]. Perhaps as a result of this heavy research, however, substantial gains in algorithmic efficiency are increasingly difficult to attain. To accelerate these searches further we are forced to look for alternative improvements beyond the scope of standalone algorithms, such as exploiting memory, metacomputation, or parallel computing.

In the domain of these search algorithms, the obvious naive method of exploiting parallel computation is to simply run the same algorithm with different random seeds on multiple machines in parallel until one of the agents finds a complete solution. While this method does speed up the search, it does not seem to take full advantage of parallel computing by isolating each agent. In the *Cooperative Search* paradigm agents searching in parallel share information in order to more quickly arrive at a global solution [8, 9, 20]. Specifically, Cooperative Search allows agents swap partial solutions to different parts of the problem via a global blackboard. Cooperative Search is more effective than naive parallel search, but it can suffer from a lack of search diversity when all the agents are working on the same part of the problem. The *TeamSearch* and *DRATS* algorithms presented in this dissertation will remedy this problem and ultimately improve both the average performance and worst case performance of the Cooperative Search paradigm.

1.1 Background

1.1.1 Really Hard Problems and Local Search

The aforementioned "hard problems" are a subset of the class of NP-complete problems and are found in remarkably different domains [7, 17, 35]. Despite having different origins, they can all be formulated in terms of variables, values, and constraints in what is known as a constraint satisfaction problem [29, 34, 24]. These problems are considered solved only when each variable is assigned a value and no constraints are violated. Testing to see if a given assignment is a solution, therefore, is computationally cheap. Among the most effective methods of solving these problems are local search algorithms [27, 28]. These algorithms begin with a random complete assignment of values to variables and then move to neighboring assignments by changing a single value at a time. Local Search continues considering different neighboring assignments until a solution is found. While this basic algorithmic description is amazingly simple, it is also remarkably fast and effective. There are, nonetheless, many tricks for improving the performance of these algorithms.

1.1.2 Solution Landscapes, Hill-climbing and Memory

The most common trick used to improve local search performance is to provide the algorithm, or agent, with the evaluation function that determines the quality of its current assignment (by the number of constraints violated, for example). With this information available, the agent can consider a neighboring assignment and only choose to accept it if it improves upon his current one. This method is called hill climbing because, if you imagine the space of assignments as a landscape where higher quality (more complete) solutions are at a higher elevation, the agent is literally searching for the top of the hill. Hill climbing is the most common form of local search and has been used effectively on almost all types of NP-hard problems [39]. The downside of hill climbing is that the agent could get stuck at the top of a small hill (a local optima) and be unable to get to the highest point in the landscape (the global optima). To solve this problem, most algorithms are implemented with random restart where an agent will select a random new solution if it gets stuck in a local optima for some predetermined number of steps. The alternative to this sort of restart mechanism is to allow the agent to accept poorer solutions under certain circumstances, an approach that was first used by Simulated Annealing [23] in the Traveling Salesman domain.

Local search agents are also more effective when given a small amount of memory to record their history of assignments. It has been shown, for example, that agents do well if they record hints, or partial assignments that violate no constraints [8]. These hints can then be re-incorporated into an agents' future assignment, as an alternative to picking a random assignment from which to restart. Alternatively, Agents in Tabu Search [18] record recently visited assignments and do not return to them for as long as the assignment is in memory, thus preventing immediate backtracking and propelling themselves to new areas of the solution landscape. Tabu Search has been used to solve a wide range of hard optimization problems such as Job Shop Scheduling, Graph Coloring, the Traveling Salesman Problem and the Capacitated Arc Routing Problem.

1.1.3 Parallelism and Cooperative Search

Running multiple search agents in parallel improves global performance to a point, but this naive formulation may not take full advantage of the potential of parallel computing. It has been shown that allowing the agents to communicate with one another is more effective [20]. Under the Cooperative Search paradigm [8], agents share a global blackboard upon which they post hints. Hints are simply partial solutions generated by removing inconsistent assignments until no constraints are violated. Agents post these to the blackboard with some probability when they find an improved partial solution. Meanwhile, another agent may be stuck at a local optima and, with some probability, decide to integrate a hint from the blackboard rather than restart. When taking a hint of the blackboard the agent overwrites some of his own assignments with the assignments in the hint, thus combining a partial solution of his own with a partial solution of some other agent. Although this will not always produce a solution with fewer constraint violations, it will generally allow the agent to move toward a solution by searching in a reduced space of possibilities. Hogg and Williams [20] present a simple statistical argument for this tendency:

> "Suppose we view the agents as making a series of choices. Let p_{ij} be the probability that agent i makes choice j correctly (i.e. in the context of its previous choices, this one continues a path to a solution, e.g., by selecting a useful hint). The probability that the series of choices for agent i is correct is then just $p_i = \prod_i p_{ij}$. With sufficient diversity in the hints and agents' choices to prevent the p_{ij} from being too correlated, and viewing them as random variables, this

> multiplicative process results in a lognormal distribution [38] for agent performance...hence there is an increased likelihood that at least one agent will have much higher than average performance, leading to an improvement in group performance." [20] (For a more formal analysis, refer to Clearwater *et al.*, 1991.)

Thus, using hints we manage to preserve work done by agents on unsuccessful (but still potentially useful) searches. Notice, however, that one of the important assumptions of this argument is the maintenance of diversity in the search.

1.2 Description of Project

We observed that Cooperative Search occasionally performed remarkably poorly despite good performance on average. While this is a normal feature of local search, and indeed all search algorithms on such hard problems, it was more noticeable in Cooperative Search than in naive parallel local search. The intuition of many researchers, including the developers of Cooperative Search, is that this is a feature of poor search diversity among the agents [9, 20]. Once a majority of the agents are in one area of the search space, information sharing has a tendency to keep most of the agents in that area thus preventing them from finding a solution somewhere else in the search space. In response to this need we developed TeamSearch, which is simply several smaller instances of Cooperative Search run in parallel. This division forced the search algorithm to maintain diversity as different teams could not communicate with one another.

The effectiveness of TeamSearch in general, however, does not tell us the optimal division of agents among teams. It seems likely that at different points in the search teams have different needs in terms of number of agents. It may be possible to learn the characteristics of the problem and use that information to dynamically modify the search procedure. Testing this hypothesis, however, required information on the value of an agent to a specific team at a given point in the search. Such information had not previously been gathered. Furthermore, it wasn't necessarily clear that such data would be generalizable enough to be useful even if it were available.

Assuming we were able to gather useful data, knowing which data to gather was not obvious. We were forced to determine how to define the value of an agent. In a satisfaction solver the only actual measure of good or bad is how quickly a solution is found. In a

parallel algorithm such as Cooperative Search or TeamSearch, this means that the first agent to finish is the only one that matters. Of course, agents share information in these paradigms so agents may help another agent find the solution. Thus, the value of an agent to the search must measure not just how likely an agent is to find the correct solution itself, but also how it can contribute to another agent's search. We ultimately decided that the most important factors determining how valuable an agent is at a given point in search are how complete its solution is and how many other agents are on its team.

1.3 Summary of Results

We show that it is possible to take information from previous searches in a specific problem domain and use that information to more effectively organize cooperating agents and thereby speed up parallel local search. We developed an algorithm which uses metadeliberation to dynamically reallocate agents across different teams, each running Cooperative Search in parallel. We call this algorithm Dynamically Re-Allocated Team Search (DRATS) and found that it outperformed Cooperative Search by 14% in the average number of local search rounds required to solve hard problems. The main contributions are the following:

- We develop a framework for measuring the value of an agent to search.
- We provide an algorithm for re-allocating agents across teams using metadeliberation. This algorithm is extensible and could be easily made adaptive or anytime.
- We demonstrate that it is possible to gather data from prior searches in a domain and use it to benefit future searches.
- Our DRATS algorithm improves both the average performance (14%) and worst case performance (22%) of Cooperative Search.

1.4 Related Work

The research done by Clearwater, Huberman, Hogg, and Williams [8, 9, 20] represents the foundation upon which the DRATS algorithm was conceived. While those authors and several others [13, 15, 16] mentioned dynamic restructuring as a potentially fruitful area of research, none followed through to study the idea.

Another approach to adapting Cooperative Search involves allowing different agents to run different local search algorithms. In at least one formulation, the algorithms being run are selected over time on the basis of algorithm performance [37, 14]. Much like DRATS, this idea relies implicitly on gathering as much information as possible and using metacomputation during the search to more efficiently allocate resources [15, 30]. Meta-deliberation in computers is akin to planning in humans and involves using a small amount of time at some point in a process in order to save a great deal of time overall [34]. Related to this field is the study of bounded rationality, which deals specifically with the question of how long to metadeliberate before moving ahead with computation [36, 41, 19]. There is also a related class of algorithms known as anytime algorithms which can produce an answer whenever requested but will continue attempting to improve that answer indefinitely [5, 21]. Local search algorithms when applied to optimization problems are anytime.

There are several distributed constraint satisfaction problem solvers, some of which include dynamic agent reprioritization [48, 44]. These algorithms assign each agent a single variable or a set of variables and then agents communicate to find a solution that satisfies every constraint. The advantage these distributed algorithms have over their centralized counterparts is that groups of agents will often develop consistent partial solutions which can later be joined with other partial solutions to solve larger parts of the problem. This is precisely the same sort of mechanism that gives Cooperative Search and advantage over naive independent search! Although they employ similar tricks to gain an advantage, these families of algorithms were developed separately. Furthermore, it isn't clear how to take aspects of "distributed CSP" algorithms and use them to improve local search algorithms.

Market Oriented Programming is a distributed optimization framework which lends itself to dynamic resource allocation by simulating markets to distribute work optimally [46, 6]. Much like local search, these methods can be used to optimize or even solve NP-hard problems. Although not used in this paper, economic models such as those in Market Oriented Programming could theoretically be used to define the value of an agent and optimize the global structure with market simulation.

1.5 Outline

The second chapter will address the specific problem domain we have chosen as well as giving an example encoding. The third chapter will discuss Cooperative Search and the

implementation of our algorithms. The fourth chapter will go into greater depth on the foundation and, subsequently, the effectiveness of TeamSearch, the first attempt at increasing efficiency of Cooperative Search. In Chapter Five, discussion will turn toward DRATS, in which team sizes shift dynamically. The pieces will ultimately be brought together in Chapter Six where we compare the performance of DRATS with TeamSearch and Cooperative Search on in the Uniform Random 3-SAT domain. Finally, Chapter Seven will provide a discussion of the results as well as addressing the open questions raised by this research.

Chapter 2

Problem Selection

This chapter first describes what sort of hard problems we were looking for and where we found those problems. This is followed by a detailed discussion of our chosen problem domain and a description of how our problem instances were generated. Finally, we will give a sample problem from a different domain and convert it to our chosen domain and format.

2.1 Finding Hard Problems

Not every constraint satisfaction problem is difficult, of course, and there is a great deal of literature dedicated simply to finding hard problems [47]. In general, the hardest problems lie in the "phase transition region" between underconstrained problems, which have many solutions, and overconstrained problems, which have no solutions [7]. Finding appropriate problems was further complicated by the fact that we are primarily concerned with the number of steps required to *solve* the problem. As a result, we must deal with a set of problems that can be solved to completion. Fortunately, there are many resources for obtaining hard problems in a wide variety of domains.

SATLIB is an online library of benchmark satisfiability (SAT) problems of varying size and from various domains [2]. Any SAT problem can be formulated as Constraint Satisfaction Problem (CSP) and they are ideal for testing local search algorithms [4]. All of the problems on this site are in CNF encoded in the DIMACS standard format and can be easily parsed [1]. (We chose to use a parser written by Yumi K. Tsji and Allen Van Gelder,

downloadable from the Rugters DIMACS depository [45].) The specific problem set we selected was from the Uniform Random 3-SAT domain described below. We chose to use small (but difficult) problems to make large scale testing less time consuming. Specifically, each of the 1000 solvable instances has 21 variables and 91 clauses of length 3. The problems were all found to lie in the difficult phase transition region.

2.2 Uniform Random 3-SAT

We chose to use 3-SAT problems because it is among the most studied problem domains in the whole family of satisfaction problems, making it a good benchmark for comparison to a wide range of other problems [31]. It is also a difficult domain; 3-SAT has been shown to be the most intractable version of the k-SAT family [42]. Of course, 3-SAT is also NP-complete meaning that search results on this problem can be generally applied to all other NP-hard problems. SATLIB provides a good definition of the problem domain as well as how the instances we used were generated:

> "Uniform Random-3-SAT is a family of SAT problems distributions obtained by randomly generating 3-CNF formulae in the following way: For an instance with n variables and k clauses, each of the k clauses is constructed from 3 literals which are randomly drawn from the $2n$ possible literals (the n variables and their negations) such that each possible literal is selected with the same probability of $\frac{1}{2}n$. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological (i.e., they contain a variable and its negation as a literal). Each choice of n and k thus induces a distribution of Random-3-SAT instances. Uniform Random-3-SAT is the union of these distributions over all n and k." [2]

Using parameters described by Cheesemen *et al.* [7], the random 3-SAT instances generated are more likely to fall in the difficult phase transition region. Once generated, all instances were searched and unsatisfiable instances were rejected.

2.3 Sample Problem Encodings

As mentioned above, the SAT problems are encoded in Conjunctive Normal Form (CNF). This format consists of a number of clauses, where a clause is a disjunction of a number of boolean variables or their negations. For example, if you have boolean variables x_i, which can only represent *true* or $false$, then an example of a CNF equation would be:

$$(x_1 \vee \neg x_4) \wedge (x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

The symbol $\vee$ represents boolean `or`, $\wedge$ stands for boolean `and`, and $\neg x_i$ stands for the negation of x_i. If there are n clauses labeled C_j, then the problem is satisfiable only if assignments can be given to each variable to make the following formula true:

$$C_1 \wedge C_2 \wedge ... \wedge C_m$$

To demonstrate the proximity of 3-SAT problems to other domains, Figure 2.1 gives an example of a Graph Coloring problem which is then converted to a CNF encoded 3-SAT problem. Graph Coloring is another NP-hard domain where a set of vertexes are are connected to one another by a set of edges and the goal is to color each node using as few colors as possible such that each vertex such that no two connected vertices are the same color. The satisfaction version of this problem provides a specific number of colors and asks whether or not the graph can be colored with them.

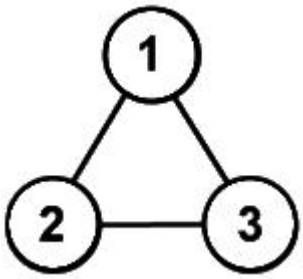

Figure 2.1: A *very* simple graph coloring problem with three connected vertices labeled 1, 2, and 3.

Let's first consider the 3-coloring version of this graph, which means we will ask the question whether or not it can be colored with 3 different colors. Our variables in CNF must be boolean so we can't just make a single variable for each vertex. Instead, we make

a new variable for each combination of vertices nd colors. Thus node 1 will be converted into three distinct boolean variables, $1R$, $1G$, and $1B$, where R, G, B stand for the three possible colors of Red, Blue, and Green. With three vertices and three possible colors we will have 9 variables. A simple encoding of the constraint that no two adjacent nodes can have the same color would look like this:

$$(1R \vee \neg 2R \vee \neg 3R) \wedge (\neg 1R \vee 2R \vee \neg 3R) \wedge (\neg 1R \vee \neg 2R \vee 3R) \wedge$$
$$(1G \vee \neg 2G \vee \neg 3G) \wedge (\neg 1G \vee 2G \vee \neg 3G) \wedge (\neg 1G \vee \neg 2G \vee 3G) \wedge$$
$$(1B \vee \neg 2B \vee \neg 3B) \wedge (\neg 1B \vee 2B \vee \neg 3B) \wedge (\neg 1B \vee \neg 2B \vee 3B)$$

This isn't enough to encode the whole problem, however, because this encoding doesn't preclude the possibility of a variable being assigned more than one color! To handle this shortcoming we add the following clauses and complete the encoding.

$$(1R \vee \neg 1G \vee \neg 1B) \wedge (\neg 1R \vee 1G \vee \neg 1B) \wedge (\neg 1R \vee \neg 1B \vee 1B) \wedge$$
$$(2R \vee \neg 2G \vee \neg 2B) \wedge (\neg 2R \vee 2G \vee \neg 2B) \wedge (\neg 2R \vee \neg 2B \vee 2B) \wedge$$
$$(3R \vee \neg 3G \vee \neg 3B) \wedge (\neg 3R \vee 3G \vee \neg 3B) \wedge (\neg 3R \vee \neg 3B \vee 3B)$$

In the 3-coloring case, there is a clearly a solution because there are as many colors as there are nodes! Indeed, we intentionally chose an example which has a fairly straightforward translation to 3-SAT, but any NP-complete problem is equivalent by definition [11]. Table 2.1 gives some possible valid assignments; it should be clear how to interpret the results in terms of coloring!

$1R$	$1G$	$1B$	$2R$	$2G$	$2B$	$3R$	$3G$	$3B$
$True$	$False$	$False$	$False$	$True$	$False$	$False$	$False$	$True$
$False$	$True$	$False$	$False$	$False$	$True$	$True$	$False$	$False$
$False$	$False$	$True$	$False$	$True$	$False$	$True$	$False$	$False$

Table 2.1: Valid 3-Colorings of Figure 2.1. Notice that no two vertices have $True$ values for the same color and that no two colors have $True$ values for the same vertex.

It may not be immediately clear how to encode the same exact graph as a different problem, so we will include its encoding as a 2-coloring problem below. It should be mentioned,

however, that 2-coloring, in general, is *not* an NP-complete problem! Of course, Figure 2.1 isn't exactly the hardest problem to begin with, so it shouldn't be a serious concern.

$$(1R \vee \neg 2R \vee \neg 3R) \wedge (\neg 1R \vee 2R \vee \neg 3R) \wedge (\neg 1R \vee \neg 2R \vee 3R) \wedge$$
$$(1G \vee \neg 2G \vee \neg 3G) \wedge (\neg 1G \vee 2G \vee \neg 3G) \wedge (\neg 1G \vee \neg 2G \vee 3G)$$

For the sake of brevity we will forgo the additional clauses needed to limit each vertex to only being assigned one color. It should be clear, just by looking at Figure 2.1, that the 2-coloring version of this problem is *not* satisfiable. Indeed, if you consider any assignment of variables to values in this most recent encoding you will see that none satisfy all of the clauses.

Chapter 3

Cooperative Search

Although Cooperative Search is just one of the local search algorithms we will consider, it is also the foundation upon which the other algorithms are based. Both *TeamSearch* and *DRATS* work within the same basic framework as Cooperative Search with minor additions and modifications. For each of the algorithms, the actual search agents are identical with the exception of the team to which they belong and other agents with whom they share hints. This chapter describes the agents and various parameters shared by all the algorithms. Finally, it will review the performance of Cooperative Search on the Uniform Random 3-SAT problems described in the previous chapter.

3.1 Algorithm Details

The implementation of Cooperative Search generally follows the algorithm described in Hogg and Williams [20], roughly reproduced in Figure 3.1. The search described for each agent is a random restart hill-climbing search. Neighboring solutions are chosen using heuristic repair which forces agents to only change the values of variables currently involved in a constraint violation [32]. (The constants in the algorithm are actually parameters and their values are justified in the next section.)

The only notable difference between this implementation and prior implementations comes with hint generation; rather than remove variable values randomly until the assignment is consistent our agents only remove variable values currently involved in a constraint violation.

CoopSearch()

definition: a_i^s = agent i's current solution
definition: a_i^n = a neighboring solution to a_i^s; a single conflicted variable is different
definition: a_i^r = the number of rounds since agent i restarted or improved its solution

```
for each agent i do
    a_i^s ← RandomSolution()
repeat
    for each agent i do
        if a_i^r > 10 then
            with probability 0.1
                a_i^s ← GetHint()
            otherwise
                a_i^s ← RandomSolution()
        a_i^n ← NeighborSolution(a_i^s)
        if SolutionQuality(a_i^n) > SolutionQuality(a_i^s) then
            a_i^s ← a_i^n
            with probability 0.5
                PostHint(a_i^s)
        else
            a_i^r ← a_i^r + 1
until solution found
```

Figure 3.1: The Cooperative Search Algorithm. For all algorithms, access to the correct problem instance is assumed.

3.1.1 Parameter Selection

The majority of parameter values used were provided by Hogg and Williams and used again in this research for consistency [20]. The blackboard holds 100 hints, agents post hints with probability 0.5, and agents read hints (instead of restarting) with probability 0.1. To find the optimal restart frequency, however, it was necessary to test extensively to ensure the optimal values for our problem domain were used. Each test consisted of running 25 complete searches on each of 1000 instances while recording the number of rounds it took for a solution to be found. This test showed that restarting (or taking a hint) after being stuck for 10 search steps was optimal regardless of team size. (To see a graph of this data, refer to Figure 4.1.)

We chose to do all of our tests using 20 agents, a value which is on the transition between substantial performance gains (1-15 agents) and trivial gains (25+ agents). (See Figure 3.2.) Using more agents provides a global reduction in search steps, but the additional bookkeeping cost in our serial implementation made the use of more agents infeasible.

3.2 Evaluation

Cooperative Search and the algorithms described in the following chapters are serial simulations of parallel algorithms. This makes implementation and testing more straightforward but can potentially complicate evaluation. Each algorithm could be evaluated serially or in parallel. A good example of a serial measurement is the *total* running time of the algorithm. In our serial implementation, this is theoretically the sum of the running times of all the agents. A good example of parallel measurement is the number of rounds the fastest agent took to solve the problem.

In general, we decided to measure performance by the number of rounds before the first agent solved the problem. This measurement provides a better indicator of potential performance in a true distributed environment. When comparing DRATS to the other algorithms, however, using rounds is not sufficient because it does not account for the cost of metadeliberation. To measure the impact of the extra computation we recorded the average running time *per round* of each search algorithm. A similar value could be attained by taking the total running time and dividing it by the total number of rounds searched by all agents.

3.3 Performance

One of the reasons that Cooperative Search was an improvement over naive parallel independent search was that adding agents to independent search showed diminishing efficiency returns, as we see in Figure 3.2. Nonetheless, we still see diminishing performance returns for each additional agent, a common feature of parallel systems called Amdahl's law [3]. Thus, while it is clear that adding more agents will continue to improve the global efficiency of the search, they will do so to an increasingly limited extent.

The substantial gains achieved by the first additional agents, however, indicates that smaller teams using fewer agents might make more effective use of the processing power

available than one team with all the agents. Furthermore, such performance gains could potentially be reinforced by gains in search diversity. Indeed, Clearwater *et al.* [8] cite the maintenance of search diversity as one of the primary components of gaining a superlinear speedup as the number of agents increase and escaping Amdahl's law.

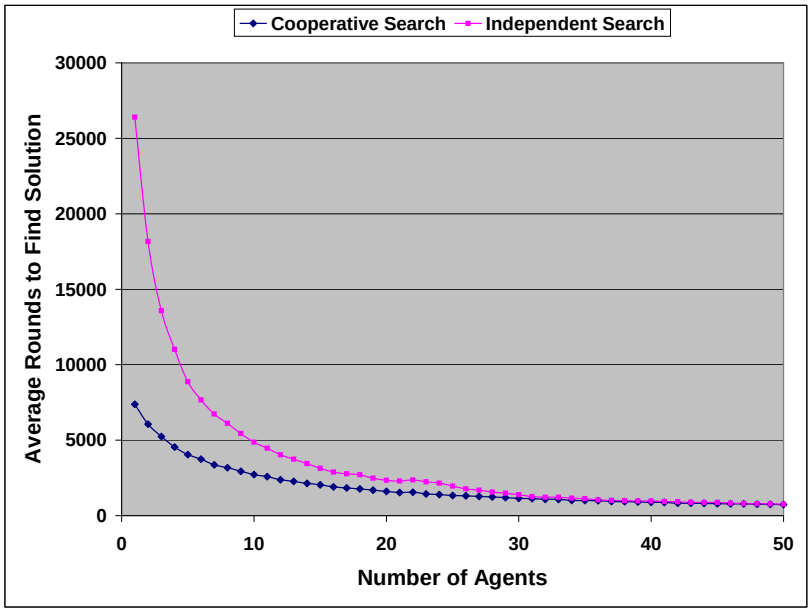

Figure 3.2: Search performance of Cooperative Search compared to naive Independent search with varying numbers of agents. 25,000 searches were run on the Uniform Random 3-SAT domain for each number of agents.

With 20 agents, Cooperative Search averaged 1880 rounds of search before finding a solution, with standard error of 3,857 rounds. This is compared in Figure 3.2 to naive independent search which required 2343 rounds, on average, and had a standard deviation of 4,256 rounds. An interesting feature of this data is that 3,148 of the searches (about 12.5% of the tests) required more than 10,000 search steps to find a solution. This is actually a slight *increase* in the number of long searches when compared to naive independent search which had only 3,079. (For a graphical depiction of this Cooperative Search data relative to TeamSearch refer to Figures 4.3 and 4.4.)

Chapter 4

TeamSearch

The potential problem with Cooperative Search is that if all the agents were to end up in one area of the search space and all the hints were from that same area, agents taking hints would not be exploring new areas of the problem space. Instead, the agents would be stuck and continue to reinforce their own position. Even though its average performance is better than naive parallel independent search (see Figure 3.2), occasionally Cooperative Search has spikes of poor performance that are even worse than those of parallel independent search.

We believe that the reason for these spikes is precisely the scenario described above where the search has lost diversity. Unfortunately, this conjecture is difficult to test because there is no established method for measuring the amount of diversity in a search. Nonetheless, it is an idea held widely enough that the original researchers were aware of this and emphasized a need to maintain diversity in the search [9, 20], although they did not address how to achieve this goal. *TeamSearch*[1] was developed to keep the search diverse by running multiple instances of Cooperative Search in parallel but without communication between the teams.

4.1 Implementation

TeamSearch uses the same number of agents as Cooperative Search and simply divides them onto different teams. The only difference is that there is no global blackboard. Instead, each agent belongs to a different team with its own private blackboard. Agents may only

[1] J.S. Levin was the co-developer of the TeamSearch algorithm as well as my partner in the early development of DRATS.

read and write from the blackboard associated with their own team. If the number of agents was not evenly divisible by the number of teams, leftover agents were simply distributed to teams one at a time until none remained, leaving some teams a single agent short. The implementation of TeamSearch and a brief discussion of its differences from Cooperative Search are given in Figure 4.2.

4.1.1 Verifying Parameters

It isn't necessarily clear that the optimal parameter values for a Cooperative Search with 20 agents are also optimal for smaller searches. For that reason, we tested the effect of team size on the parameters. Once again, the test consisted of running 25 complete searches on each of 1000 instances while recording the number of rounds it took for a solution to be found. The results showed that the optimal restart frequency, despite minor variation, is generally consistent across all team sizes. Figure 4.1 shows that restarting or taking a hint after being stuck for 10 rounds is optimal for nearly all team sizes.

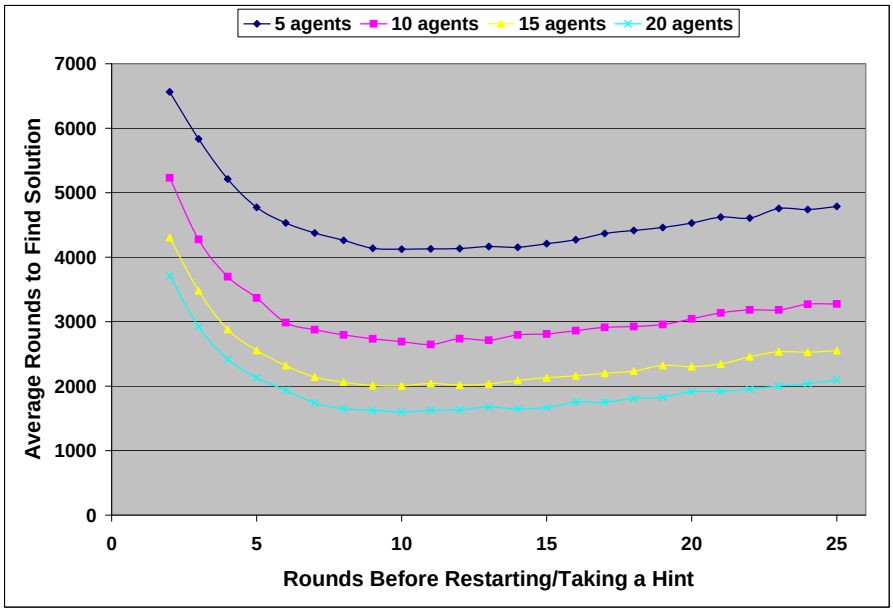

Figure 4.1: Performance of Cooperative Search as restart frequency and team size vary. Each data point represents 25,000 complete searches in the Uniform Random 3-SAT domain.

We decided to provide each team with a full sized blackboard (holds 100 hints) irrespective of the number of agents on their team. Another possibility that we did not explore

would be fixing the amount of global blackboard space and dividing it among the teams either evenly or by number of agents. Although we are not presenting the results here, our preliminary testing in this area indicated that the effectiveness of a blackboard did not vary substantially with the number of agents using it.

4.2 Performance

Our experiments show that smaller parallel Cooperative Searches outperform one large Cooperative Search with the same number of total agents. Over 100,000 searches, a search with 20 agents divided among 6 different teams took 5% fewer rounds, on average, than a single team performing Cooperative Search. The standard deviation for the number of rounds was also reduced by 8%. Figure 4.3 clearly shows the substantial performance improvement achieved by dividing the agents between two teams. Further division improves the results until the average number of agents per team is just over 3, at which point further subdivision of the agents hurts performance.

When reading the charts keep in mind that Cooperative Search is just a special case of TeamSearch where there is only one team. Conversely, if the number of teams is equal to the number of agents the search is very similar to naive independent search. The difference is that each agent will still have its own blackboard with which it may exchange hints. This configuration has been shown to be a "slight improvement" over independent search because agents communicating with prior versions of themselves are effectively not dissimilar from two entirely different agents communicating with each other [20]. Clearwater *et al.*, who refer to this search as "non-cooperative search," found Cooperative Search to be more effective [8]. For this reason, our research focuses on team configurations with at least an average of 2 agents per team.

In line with our predictions, a substantial part of the average gains TeamSearch saw over Cooperative Search were related to an 13% reduction in the number of exceptionally slow instances, as we see in Figure 4.4.

TeamSearch(C, N)

definition: a_i^s = agent i's current solution
definition: a_i^n = a neighboring solution to a_i^s; a single conflicted variable is changed
definition: a_i^r = the number of rounds since agent i restarted or improved its solution
definition: a_i^t = the team to which agent i belongs
definition: $\{a_1^t, a_2^t, ..., a_n^t\}$ is a complete configuration for n agents
input : A complete configuration C
input : Number of rounds to search N

```
n ← 0
repeat
    for each agent i do
        if a_i^r > 10 then
            with probability 0.1
                a_i^s ← GetHint(a_i^t)
            otherwise
                a_i^s ← RandomSolution()
        a_i^n ← NeighborSolution(a_i^s)
        if SolutionQuality(a_i^n) > SolutionQuality(a_i^s) then
            a_i^s ← a_i^n
            with probability 0.5
                PostHint(a_i^s, a_i^t)
        else
            a_i^r ← a_i^r + 1
    n ← n + 1
until n ≥ N or solution found
```

Figure 4.2: The TeamSearch Algorithm. This algorithm is almost identical to the Cooperative Search algorithm (Figure 3.1) except that it accepts a team configuration C as an argument and agents exchange hints with a blackboard specific to their team. It also accepts an argument N that limits the number of rounds. This is useful for the DRATS algorithm (Figure 5.1). Agents are initialized with random solutions and maintain their solutions between calls.

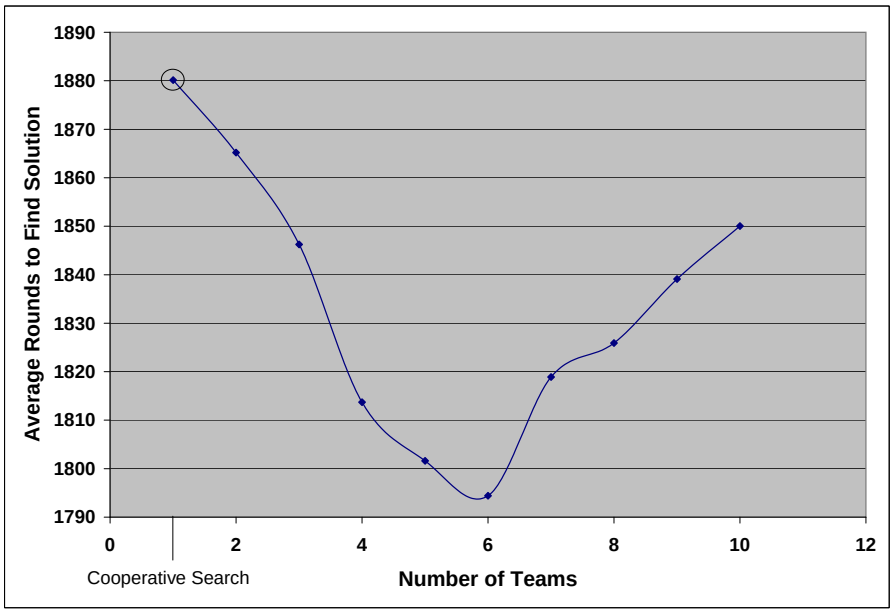

Figure 4.3: Performance of TeamSearch varying the number of teams. In each search there were 20 total agents evenly divided among the teams. 25,000 searches were run on the Uniform Random 3-SAT domain for each team configuration.

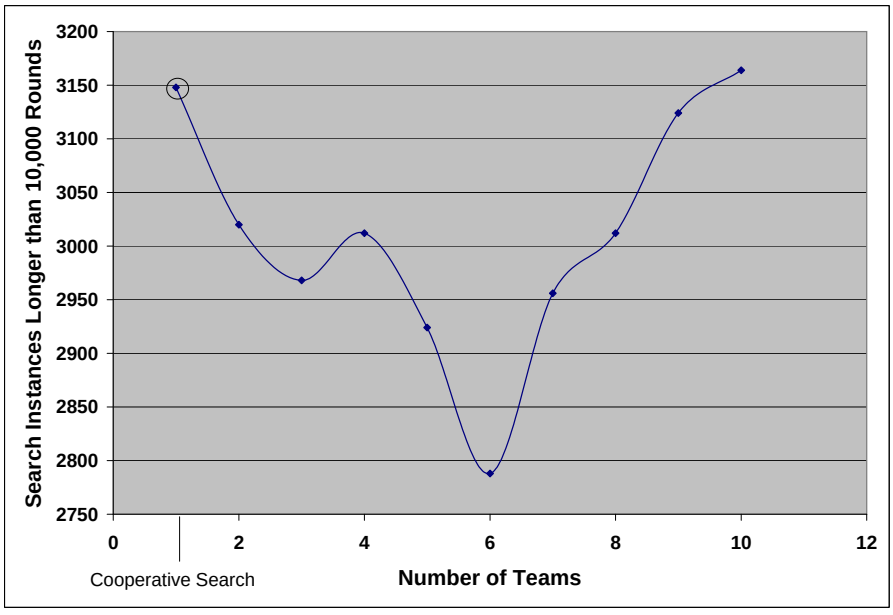

Figure 4.4: TeamSearch instances requiring more than 10,000 rounds to find a solution as the number of teams varies. Data was taken from 25,000 complete searches of the Uniform Random 3-SAT domain per team size. For reference, the average performance of TeamSearch with 6 teams was 1,800 rounds.

Chapter 5

DRATS

Despite the speedup achieved from TeamSearch, we conjectured that with its fixed topography it was unlikely that any single configuration is ideal for all problem instances or even for all times during a single search. Optimizing the team configuration to the specific search being run was the idea behind Dynamically Re-Allocated Team Search (*DRATS*), an algorithm that allocates agents to different teams dynamically during the search to optimize the configuration. Such dynamic reorganization was proposed in the original Clearwater *et al.* paper [8], but ours appears to be the first study.

The chief difficulty is to effectively measure the marginal value of each agent given the global state of the search. By gathering data on the ultimate effect of changes in the size of individual teams at different times during search we approximate a function from solution completeness (in percentage) and team size to the value of a given agent. This provides information on the *expected* marginal value of a single agent for a specific team at a specific point in a given search. With that data in hand we use statistical methods to calculate the expected global value of different team configurations and dynamically reallocate team size to maximize that global expectation.

5.1 Modeling the Problem

Possible factors affecting the ability of any given agent to contribute to a team include the quality of its solution, the number of other agents on its team, and the quality of hints on its team's blackboard. We showed in Chapters Two and Three that the number of agents

on a team has a clear correlation with team performance. We are also able to measure the quality of a given agent's solution. There is no clear correlation, however, between the quality of hints on a blackboard and search progress, making blackboard analysis an ineffective measure of a team's solution quality [20]. Our model enables us to learn the relationship from team size and solution quality to the value of an agent to a given team.

In our model of TeamSearch, although every agent provides *some* amount of value to the team to which it belongs, each team will value that agent differently depending on how many agents they already have and depending on the quality of each of those agent's solutions. Although it is clear that every team will want as many agents as possible, the diminishing marginal value of each additional agent (see Figure 3.2) dictates that a global optimization based on each team's value will tend to spread agents out rather than let them accumulate. As users, we are only interested in the final result of the algorithm as a whole rather than the performance of any individual agents or teams.

Rather than learn an explicit function from team size and solution quality to value, we use a data intensive approach [10]. We constructed an offline model of the performance of agents in TeamSearch in our domain. We recorded the progress of each agent every 50 steps in 20 million searches (1 million searches with each team size). Using that data as a statistical model of algorithm performance we sample from that data at run time to obtain an estimate of the value of various team/agent configurations.

This model relies implicitly on the ability to measure the quality of an agent's solution, but there is more than one way to do this. One measure would be the number of constraints that a solution violated (the fewer, the better). The alternative method is to consider the number of variables that are violating at least one constraint. In our problems, there were 91 constraints and only 21 variables so we could conceivably gain greater specificity in our measurements if we used the number of constraint violations as an indicator. Furthermore, if we were studying constraint optimization problems, as opposed to satisfaction problems, our primary goal would be to minimize constraint violations. However, in the domain of constraint satisfaction problems it seemed more appropriate to use the number of variables in violation as a measure. The behavior we observed in the domain of Uniform Random 3-SAT is that changing one variable seemed to drastically change the number and membership of violated constraints. In contrast, changing one variable seemed to have a much more conservative effect on the number of variables involved in constraint violations and therefore provide more consistent data. There are certainly legitimate arguments for either method,

but we chose to measure quality by the number of variables in violation of constraints.

5.2 Data Collection

At the beginning of each search our data gathering algorithm records the solution quality of each agent. Every 50 steps hereafter, the algorithm briefly halts all agents from searching while it records the search progress of the agent alongside the size of the agents team and its solution quality at the beginning of that search period. The DRATS reallocation mechanism interrupts search in exactly the same way every 50 steps, but DRATS shifts agents around rather than recording their progress. (See Figure 5.2.) With that in mind, it is worth noting that reallocation and data gathering could *both* be implemented in the same algorithm, allowing DRATS to both reallocate during search and also continue learning about its domain. Using this method, DRATS could easily be considered an adaptive algorithm.

5.2.1 Method

These tests involved all 1000 instances being searched to completion (without an upper bound on run time), with 100 runs for each instance. We then repeated this test with teams of every size in the range of 1 to 20. Every 50 rounds of searching, the global algorithm halts briefly to record the progress each agent has made in the previous 50 steps. In reality, this "progress" is not always positive because an agent might get stuck and restart in an inferior area of the search space. Along with the initial and final solution completeness we record the size of that agent's team. The data is written to file in an easily parsable format (outlined in the next section) for future reference by our reallocation algorithm.

It is important to realize that this entire data gathering phase is done "offline" as a part of the precomputation necessary to take advantage of the DRATS algorithm. Gathering the data took a great deal longer than any set of experiments presented in this research, so the burden of requiring this precomputational strain should not be taken lightly. That said, a brief analysis of our results indicates that it may be possible to take advantage of dynamic reorganization with less data (see Figure 6.4). Furthermore, as noted above, DRATS could gather data and reorganize simultaneously allowing it to improve its ability over time.

5.2.2 Data Format

Between the precomputation phase and the actual DRATS testing we kept the data (over 3GB worth) stored in a simple plain text format (Table 5.1). The data for each team size was kept in a separate file named `N.dat` where N is the number of agents. Each of these 20 files was exactly 100 lines long with each line representing a percentage point between 0 and 99. Whenever an agent completed 50 rounds of search, his current solution progress (in percentage) would be added to the end of the line representing his initial solution progress.

0	C_0	d_0^0	d_1^0	...	$d_{C_0}^0$
1	C_1	d_0^1	d_1^1	...	$d_{C_1}^1$
...					
99	C_{99}	d_0^{99}	d_1^{99}	...	$d_{C_{99}}^{99}$

Table 5.1: File Format for Storing DRATS Data. Note that the leftmost column represents line number and is not part of the file. C_i holds the number of values on line i while d_k^j is the k^{th} data point on line j.

When running DRATS, the data from all the files was loaded into an array before any search began to make sampling as fast as possible.

5.2.3 Cross Validation and the Question of Overfitting

When using the same data to both learn a function and test that function one runs the risk of "overfitting," or learning the *specific* data in the set rather than a more general function thus casting a shadow of doubt upon any results [33]. For this reason, we had initially planned on using 10-fold cross validation to do our testing [43]. We gathered data from each 100 instances separately and intended to use $\frac{9}{10}$ of the instances as data to guide a DRATS search running on the remaining instances. A brief analysis of our data, however, relieved us of the burden of cross-validation. No 900 instance subset of the data set differed by more than 1% from the parameters of the Normal distribution of the data set as a whole (shown in Figure 6.4). In other words, there does not seem to be enough variation in the data to justify cross-validation. Moreover, we are not explicitly fitting a function but rather sampling from the collected data which exempts us from the worry of learning the noise in the data.

5.3 The DRATS Reallocation Algorithm

Once we have the necessary data we can use statistical methods to calculate the expected value of adding or removing an agent from a team given the team's size and the agent's current search progress. The algorithm we use to restructure the teams on the basis of these valuations is given in Figure 5.2. This algorithm fits within the TeamSearch algorithm given in Figure 4.2 by interrupting search every 50 rounds to execute a reallocation routine. Figure 5.1 gives a high level view of how DRATS interacts with Cooperative Search.

DRATS(C)

definition: C = the current configuration

$N \leftarrow 0$
repeat
  `TeamSearch(`C, *50*`)`
  $C =$ `reallocate(`C`)`
until *solution found*

Figure 5.1: The DRATS algorithm, parameterized by the initial configuration C.

Note that the first line of the reallocation algorithm (Figure 5.2) restricts the configurations our reallocation algorithm will consider. Given that our algorithm is essentially taking away from computation time that could be dedicated to searching we want to limit its computational footprint. One possible reallocation algorithm would be to use dynamic programming to consider every configuration of agents and teams, but this would be computationally expensive, even with just 20 agents. Instead, we limit the changes in configuration to one agent per reallocation. Thus, if the initial configuration involves two teams of 5 agents, a reallocation will only consider team configurations of 4 & 6, 1 & 4 & 5, as well as the original 5 & 5. This will allow teams to have some continuity through reallocation and subsequently improve the likelihood of our data being an effective measure of long-term performance. Moving one agent, however, does not preclude adding a new team with a single agent or contracting a team out of existence! Figure 5.3 shows just a few of the many reallocation options available to a specific six agent search.

Even with the limitation on team configurations there remains a question of which agents to move. One system would be to try every combination of different agents on different teams. While this might provide the optimal performance gain, it will again leave a larger footprint on the overall running time of the algorithm. Our solution will instead pick the

reallocate(C°)

definition: a_i^t is the i^{th} agents team
definition: a_i^s is the i^{th} agents solution
definition: $\{a_1^t, a_2^t, ..., a_n^t\}$ is a complete configuration for n agents
input : The current configuration C°
output : The optimized configuration C'

```
1 for each configuration c with at most one T_i different from C° do
2     Do 100 times
3         for each unique team t ∈ c do
4             for each agent i ∈ t do
5                 x_i ← sample(sizeof(t), SolutionQuality(a_i^s))
6         Max_t ← max(x_i)
7     Mean_c ← mean(Max_t)
8 C' ← argmax(Mean_c)
```

Figure 5.2: The DRATS Reallocation Algorithm. This algorithm is passed the current configuration C° and returns a new configuration C'.

agent on each team with the lowest solution quality and try moving him. This naive heuristic could easily be replaced with a more sophisticated one in future work.

Notice that when we calculate the value of each team we are taking only the max value out of the recorded samples. Because this study involves only satisfaction problems, teams only care about the quality of the best solution out of all of their agents. By sampling once for each agent given that agent's current solution quality and then taking the max, we get the expected maximal value of the team after an additional 50 steps.

This algorithm is using metadeliberation to optimize performance. Occasionally the algorithm will stop searching for a solution to the problem and reorganize in the hope that the benefit of a more efficient team configuration will outweigh the cost of metadeliberation. When given the option we always chose to make the metacomputational aspects of DRATS as cheap as possible, but there will undoubtedly still be a footprint in the average CPU time per round. This data is presented in Table 6.1 in Chapter Six along with discussion.

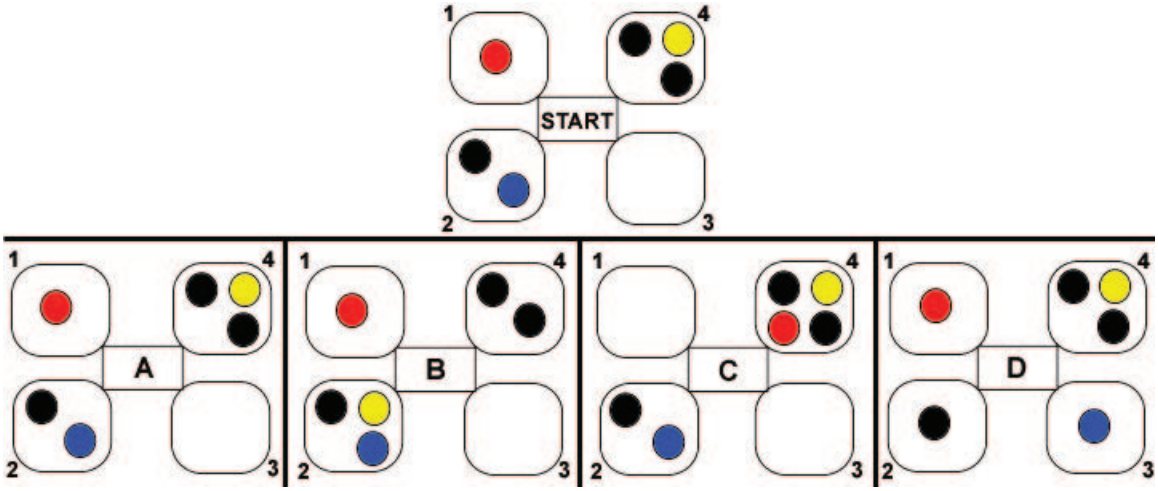

Figure 5.3: A graphical representation of the DRATS algorithm. Each of the rounded numbered boxes represents a team, notice that some teams can have no agents meaning they are effectively non existent. From the configuration labeled START a single round of DRATS reallocation could result in any of the four positions below (as well as many others). In configuration A, no changes have been made. Configuration B has the same number of teams with the same sizes but the yellow agent has been swapped from team 4 to team 2. Team 1 is contracted in configuration C as its only agent, red, is shifted to team 4. Finally, a new single agent team is created when the blue agent goes to team 3 in configuration D.

Chapter 6

Experimental Results

We compare performance of the DRATS algorithm with TeamSearch, Cooperative Search, and with independent agents searching in parallel. All the algorithms were run with optimal parameters for 20 agents. Each algorithm was run 100 times on each of the 1000 instances of Uniform Random 3-SAT and allowed to run until a solution was found. Previously, data was collected offline to provide statistics to control DRATS.

6.1 Performance

DRATS outperformed all of the other algorithms, requiring 14% fewer rounds, on average, than Cooperative Search and 8% fewer than TeamSearch with 6 teams. This reduction in average search time was accompanied by a substantial reduction in the number of searches requiring more than 10,000 rounds to complete (22% reduction vs. Cooperative Search, 9% reduction vs. TeamSearch). Additionally, Table 6.1 contains the average amount of CPU time required to run each of the four algorithms.[1] The DRATS algorithm takes 5% more time per round, on average, than the other searches. This additional time makes sense because DRATS improves search performance using metacomputation, which has some additional computational cost. Fortunately, the increase in computational time for each round is compensated by the reduction in the number of rounds required to solve the problems, as we see in the average running time.

When looking at these results keep in mind that all measurements are in in parallel.

[1] This does not include the substantial processor time required to gather the reference data.

	Independent	Cooperative Search	TeamSearch	DRATS
Average Rounds	2343	1880	1794	1654
Standard Deviation	4256	3857	3575	3317
Searches >10k Rounds	3079	3148	2788	2569
CPU Time per round	949μs	958μs	959μs	1007μs
Average Running Time	462.6s	376.2s	358.2s	352.8s

Table 6.1: Search performance by algorithm. Measured from 100,000 complete searches with each algorithm in the Uniform Random 3-SAT domain.

The average running time, for example, measures the average running time of each agent during the search. Of course, the measurements dealing with rounds are also parallel and taken from each individual agent. Note that metadeliberation in DRATS is not particularly optimized. Our reallocation scheme represents a first attempt and there is most likely a great deal of room for improvement.

6.2 Reallocation Analysis

While running the DRATS algorithm we kept track of the reallocations occurring every 50 steps of search. After 100,000 complete searches this yielded a substantial amount of information which may give us some insight into what configurations the algorithm tended to find valuable at different times in the search. As mentioned earlier, each search began divided among 6 teams so that there were 2 teams with four agents and 4 teams with three agents. Figure 6.1 shows the progression of the average team size alongside the average size of the largest and smallest teams.

Our TeamSearch data (see Figure 4.3) led us to expect DRATS to prefer configurations averaging just over 3 agents per team. Instead, the algorithm seems to have selected an average of 2 agents per team but the average is misleading as the algorithm clearly doesn't spread the agents out evenly. Instead, it seems to select configurations with one large team and several small teams, often with only a single agent. We did not consider such an asymmetric distribution in our experiments on optimal static TeamSearch configurations. Figure 6.2 shows how different team sizes are represented at different places in the search.

As we see in Figure 6.3, the reallocation that takes place during an average length search maintains a relatively high number of agents per team (just over 3). Nonetheless, we can

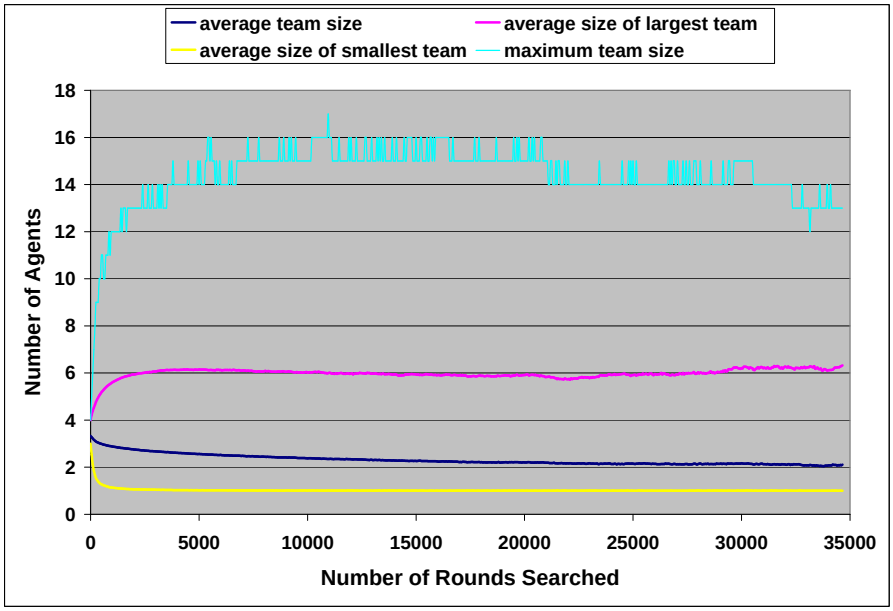

Figure 6.1: Average, minimum, and maximum seam sizes during DRATS search. Only searches shorter than 35,000 rounds are shown because fewer than 100 searches went beyond this threshold. Data is from 100,000 total searches in the Uniform Random 3-SAT domain.

already see a tendency of the algorithm to want single agent teams around in addition to a few larger teams. After about 2,000 rounds, as is clear in Figure 6.3, the DRATS algorithm seems to consistently move toward a configuration containing one large team with between 4 and 7 agents, one 3 agent team, two or three 2 agent teams, and about five single agent teams. Notice, however, that while this configuration is stable, the membership of the teams is still very fluid. Thus, the team that was largest after 5,000 steps is only of medium size by 10,000 steps. The changing membership of these teams even as the number of teams and their sizes remains relatively fixed is doubtlessly affecting the performance of the algorithm.

It should be noted that the limitation of this data is that it all started in the same configuration. If the configuration were allowed to continuously change across searches it might eventually arrive at a stable state which could be considered an optimal configuration for a static TeamSearch. Alternatively, we might find that the algorithm chooses different configurations toward the beginning of the search that our fixed starting topography and conservative reallocation policy don't allow us to see. This is an area for exploration in future work.

One might observe in Figure 6.3 that agents tend to leave teams "lower" in the graph

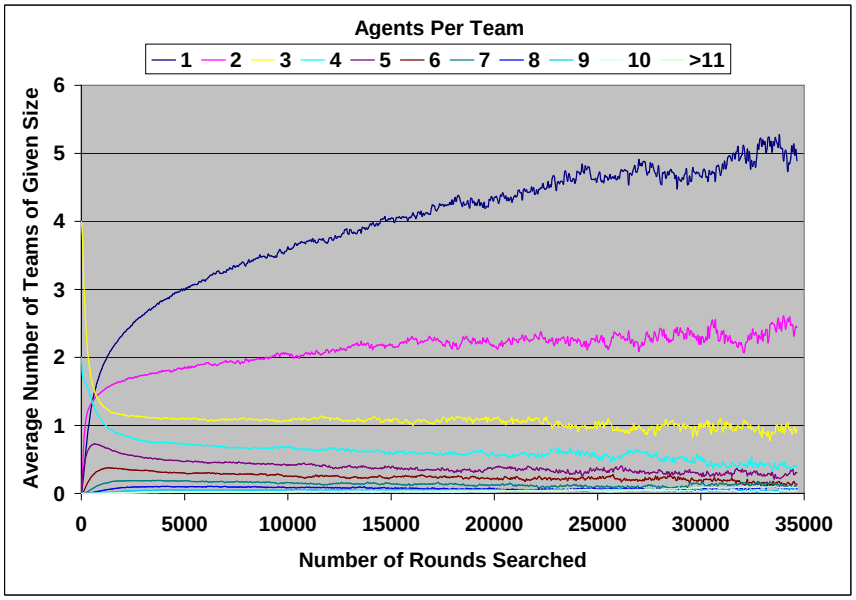

Figure 6.2: Frequency of various team sizes during search with DRATS. Data taken from 999,900 searches in the Uniform Random 3-SAT domain.

and join the teams "above" them. This is a feature of how we chose to break ties in our algorithm. If the DRATS reallocation algorithm finds more than one configuration with the same estimated mean value, it chooses the first one found (see Figure 5.2, line 8). As it happens, our algorithm explored solutions that moved agents away from the first few teams before it explored those that moved agents toward the first few teams. This clearly had an effect on the specific features of reallocation (especially when viewed graphically) but we expect that it had no effect on the actual algorithmic performance.

6.3 Data Analysis

Although the data we used to run DRATS wasn't specifically gathered for analysis, there were a number of interesting features that merit brief discussion. The most noticeable feature was an almost perfectly Normal distribution of agents' initial solution quality, as shown in Figure 6.4. By virtue of the fact that each initial solution is the final solution for the previous 50 rounds, the same distribution also applies to the distribution of expected solution quality *after* 50 rounds of computation. Averaged across all team sizes, the distribution illustrates an average agent is 48 percent complete, with a standard deviation of

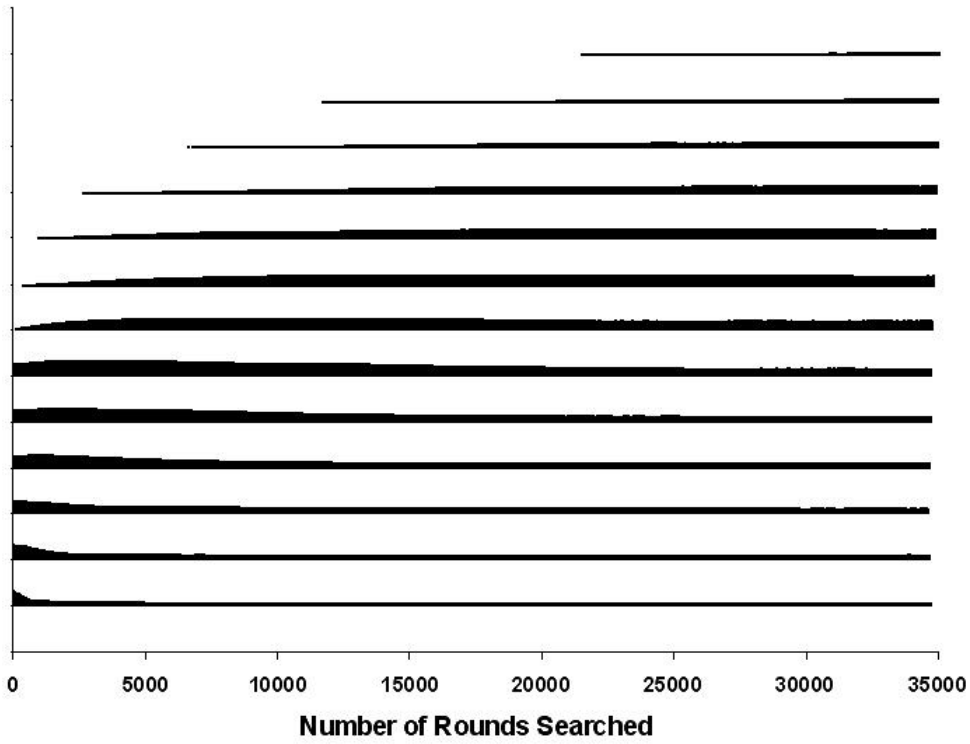

Figure 6.3: This chart depicts graphically the average redistribution of agents in the first 35,000 rounds of DRATS search on the Uniform Random 3-SAT domain. In the initial configuration, there are four teams of 3 and two teams of 4, as indicated by the initial bar heights of the "bottom" six teams. From there one agent could be moved every 50 rounds. The larger the bar (vertically) the more agents it had on average at that time in the search. The total number of agents at any vertical section is always 20.

just over 15 percent.

The algorithm clearly spends the majority of its time with halfway complete solutions, which makes sense given that solutions that violate every constraint may be as rare as complete solutions. As we expected, however, there is some variation between team sizes. Although densely packed, Figure 6.4 still shows that the curve for teams with 20 agents is shifted slightly to the right relative to the smaller teams. This indicates that, on average, a team with 20 agents will be tend to produce slightly better solutions.

To more carefully analyze differences between teams we plotted the mean solution quality against the standard deviation for each agent in Figure 6.5. For this particular graph we averaged the final solution quality of each team across all initial solution qualities. The circled values form the efficient frontier, in terms of mean and standard deviation, and

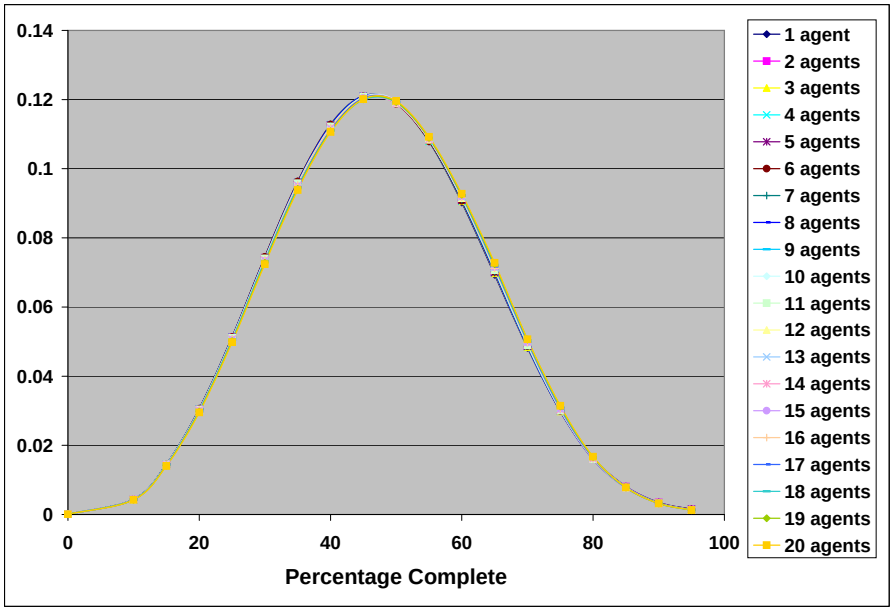

Figure 6.4: Volume of instances of agents beginning *and* ending at various solution qualities in DRATS plotted by team size and averaged over 100,000 instances.

dominate all other team sizes. The circled teams are, with one exception, the largest team sizes we considered. This graph is averaged over all initial solution qualities, and thus fails to capture the complexity that we hoped to exploit with DRATS in the first place which is that the initial solution quality *does* affect the optimal number of agents on a team.

For this reason we made Figure 6.6, which is another mean vs. standard deviation plot, but rather than averaging all initial solution qualities together, we bucketed initial solution qualities into groups of 5% and added an extra data point for each group within each team size. Figure 6.7 is the same plot but with team sizes grouped together to show trends more clearly. The arrow in these figures indicates the direction of data points of increasing initial solution quality within each team size.

These graphs are perhaps our most effective tool for understanding the trends we see in DRATS. It is clear that the large teams that tend to dominate gain most of their advantage when beginning with more complete solutions. Teams with 5 to 8 agents have a clear advantage over larger teams with initial solutions less than 45% complete. Teams of size 1 to 4 also do very well in this area of the search space. The transition occurs around 45%, beyond which larger teams begin to dominate performance. The fact that agents of

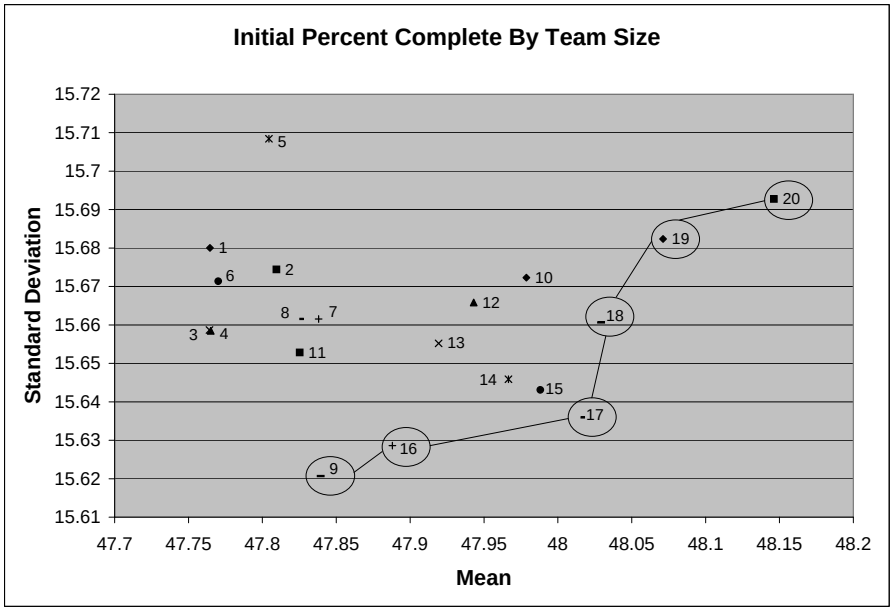

Figure 6.5: A mean vs. standard deviation plot of expected solution quality after 50 steps by team size for DRATS. This data is averaged across all initial solution qualities from 100,000 instances for each team size. The circled values denote team sizes on the efficient frontier in terms of mean and standard deviation.

all solution qualities are likely to occur throughout search may be the reason that DRATS seems to prefer having a diverse range of team sizes.

All this data still leaves us wondering why DRATS chooses to keep so many single agent teams. Teams with only one agent are still dominated, in terms of mean and standard deviation, at all points by other team sizes. We can only assume that there is some data not represented. The most likely candidate is the advantage gained by maintaining search diversity. Although diversity is difficult to measure, intuition tells us that there should be some advantage to keeping a few agents exploring the search space on their own. The advantage for the global search comes when these lone agents are brought into larger teams enabling them to share their partial solutions, which happens often (see Figure 6.3). A study of the *correlation* between the values of agents working on their own and the values of agents on larger teams may show that the advantage of single agents is not be what they do on their own but rather the diverse solutions they provide to larger teams.

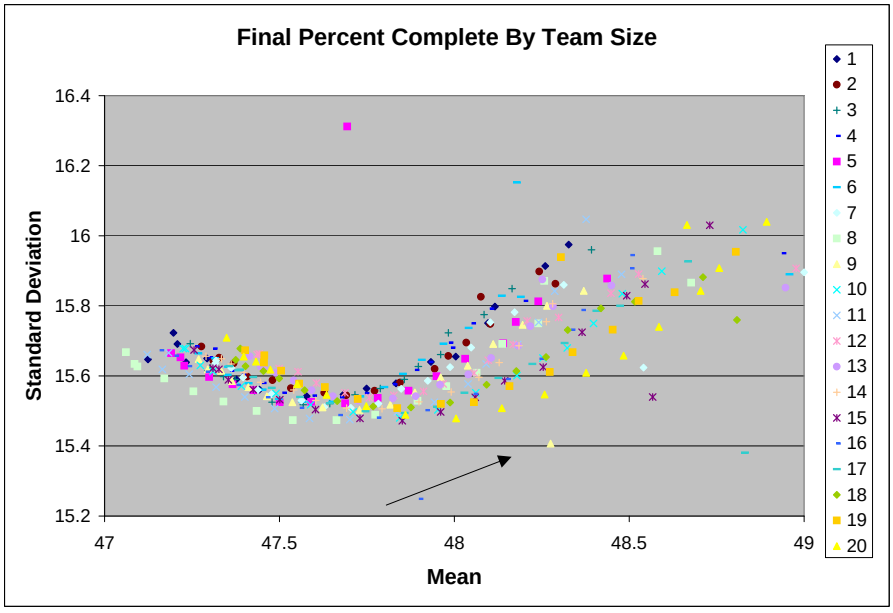

Figure 6.6: A mean vs. standard deviation plot of expected solution quality after 50 steps by team size in DRATS. Initial solution quality is broken down into groups of 5% starting with 0-5% and ending with 95-100%. The arrow indicates the direction of increasingly complete initial solutions. This data is taken from 100,000 searches for each team size on the Uniform Random 3-SAT domain

6.4 Return to TeamSearch

The consistency with which the algorithm selects the same configurations leads us to wonder whether the optimal DRATS configuration is actually independent of the progress of agents in the search. It seems possible that the one large team, many small team distribution that DRATS favors is the optimal *static* configuration. We tested this hypothesis with TeamSearch using 1 team with six agents, 1 team with three agents, 3 teams with two agents, and 5 teams with only one agent. After 100,000 searches, this configuration averaged 1802 rounds to find a solution with a standard deviation of 3612 rounds. This performance is very close to the best TeamSearch performance, but not particularly close to DRATS (see Table 6.1). Clearly, there *is* an advantage to dynamic reallocation beyond just finding good configurations for TeamSearch. We find this to be a reassuring observation.

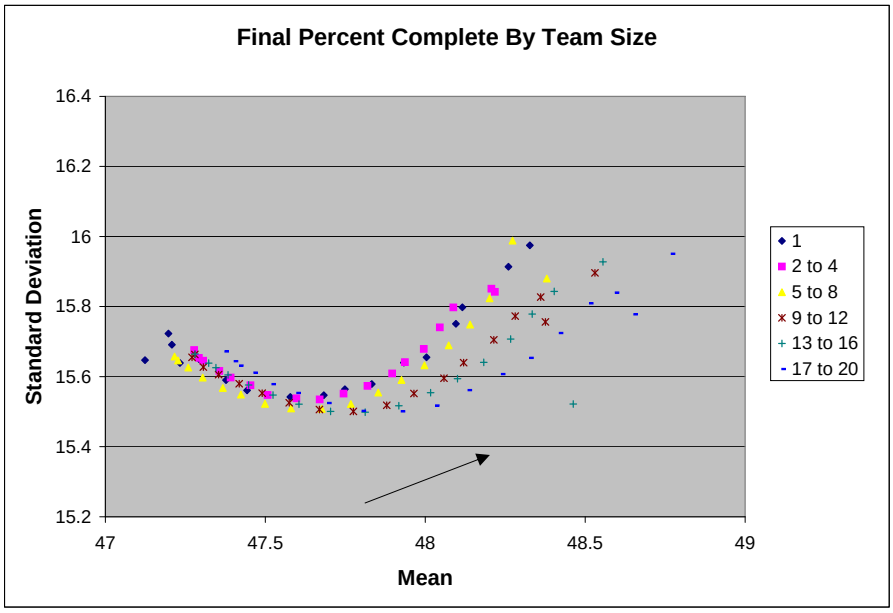

Figure 6.7: A mean vs. standard deviation plot of expected solution quality after 50 steps be team size in DRATS. Teams are grouped to more clearly demonstrate trends. Initial solution quality is again broken down into groups of 5% starting with 0-5% and ending with 95-100%. The arrow indicates the direction of increasingly complete initial solutions. This data is taken from 100,000 searches for each team size on the Uniform Random 3-SAT domain

Chapter 7

Discussion

This chapter will first explore the deeper implications of the success of the DRATS algorithm and then address its shortcomings along with areas for future research.

7.1 Implications

The DRATS algorithm successfully reduced the average number of steps required to solve some really hard problems. It is the product of a substantial exploration of the Cooperative Search paradigm and is proof that there is still a great deal to be learned about how agents can be made more effective when teamed up with other agents and allowed to communicate. The full potential of agents cooperating in parallel has most likely not yet been realized.

In addition to being a faster algorithm than its predecessors, DRATS also gives support to metadeliberation as a viable option for increasing the efficiency of local search algorithms. Similarly, this research has shown that it may be possible to use domain specific information from prior searches effectively. This attractive idea is not intuitive and has not been previously explored in the field of local search, although it has been applied to combinatorial auctions [26]. The question remains whether or not this ability to gather data varies with the domain or whether we should always be expect to gain some insight if we look at enough problems.

Perhaps the most interesting, although least explicable, lesson learned from DRATS is that mixed team size configurations seem to work most effectively in the cooperative paradigm. The algorithm shows a very clear preference for many small teams, one medium

team, and one large team. This preference resulted in a faster algorithm so we have reason to believe it is a legitimate boon to performance. Furthermore, we tested the static version of the preferred configuration and found that its performance did not match that of its dynamic version. We can only speculate about the reason that this configuration works, but it may be that the singleton agents are swapped in and out of the larger teams in order to maintain a balance between diversity and cooperation. Thus while a team of agents works on the best solutions found so far, other agents are exploring other areas of the search space. Unfortunately, research to date has found no clear way of providing any quantitative results to measure diversity nor the effect of cooperation, so conjecture remains our best tool.

7.2 Remaining work

This research opened many more doors than it closed. Although this first foray into using metadeliberation in local search met with moderate success, the efficiency gains are unlikely sufficient to justify the substantial amount of preprocessing that must be done to make DRATS effective. The balance between metacomputation, precomputation, and search may need further exploration before it can be fully understood and optimally handled [40]. Further work must also be done to determine how useful data from prior searches is, as well as how valuable a single agents contribution is in a cooperative environment.

We observed in Figure 6.4 that the distribution of initial agent solution qualities was almost perfectly Normal. This observation indicates that it may be possible to reduce the per round computational expense of DRATS by simulating sampling rather than actually dealing with raw data. As long as it could be shown that the normal distribution we saw wasn't domain specific or problem specific, the algorithm would only need to learn the parameters of the curve. This adaptive method of allowing learning to take place "in the loop" would make DRATS more extensible to other domains.

There may be more effective methods of reorganizing teams. Using dynamic programming to optimize the global team structure, for example, might be worth the added computational cost. It may also be possible to continue moving agents until the configuration is relatively stable for a number of consecutive attempts at swaps to assure the optimal configuration for that point in the search. Our current implementation, in contrast, seems to move toward a stable state but we can't be sure how that would be different earlier in the search if agents had more mobility.

Our limitation of only moving one agent per restructuring made comparison to previous results more straightforward but may not have been drastic enough to gain the full benefits of dynamic reorganization. Primarily, it may stifle the very diversity that these algorithms initially set out to maintain. It may be effective to swap agents between teams even if team sizes are kept the same! Similarly, the choice to always swap out the worst agent on a team was intuitive but might easily be replaced by some superior heuristic for agent selection. Work also needs to be done to determine the optimal frequency for reorganization.

Perhaps the most potent direction DRATS could take would be to focus on the highly related field of distributed optimization problems. Distributed Constraint Satisfaction Problems are only concerned with complete solutions, whereas optimization problems can generally continue to improve with more searching. The contribution of DRATS to this field would not be limited to dynamic reallocation but would include some gauge of how likely the search is to improve, thus helping a user decide when further computation is unlikely to improve the solution a great deal [19].

Acknowledgments

I must first thank Jimmy Levin who was my partner in developing TeamSearch as well as early versions of DRATS and without whom I would most likely never have graduated. My eternal gratitude goes to Professor David C. Parkes who has advised me in my studies, my research, my writing, and my love of Spanish tapas. My thanks go to Professor Avi Pfeffer and the Mind, Brain, and Behavior program for guiding my research in Artificial Intelligence. I would also like to thank Professor Norman Ramsey and his writing group for helping me more effectively understand how poorly I am capable of expressing my ideas. I owe a debt of gratitude and processor time to Harvard University, FAS, EECS, and DEAS, whose computers I compromised for an entire month to complete this research.

Finally, on a more personal level, thank you Dave Troiano, my scientifically dubious partner in thesising. Thanks Mike, Mante, James, and other members of my room/the Harvard Football Team. Thanks cabot-open and thefacebook.com for keeping me in steady supply of enemies and friends, respectively. Thanks to my family for pretending to understand what I do. Most of all, thank you Sara Clark for always being there.

Bibliography

[1] Satisfiability suggested format. WWW, April 2004. www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/.

[2] Satlib - benchmark problems. WWW, April 2004. www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.

[3] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AIFPS 1967 Spring Joint Computer Conference*, Atlantic City, New Jersey, 1967.

[4] H. Bennaceur. The satisfiability problem regarded as a constraint satisfaction problem. In *In Proceedings of ECAI-96*, pages 155–159, 1996.

[5] M. Boddy and T. Dean. Solving time-dependent planning problems. In *In Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 979–984, 1989.

[6] A. Chavez, A. Moukas, and P. Maes. Challenger: A multiagent system for distributed resource allocation. In *Proceedings of First International Conference on Autonomous Agents*, Marina del Rey, USA, 1997. ACM Press.

[7] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of IJCAI91*, pages 331–337, San Mateo, CA, 1991. Morgan Kaufmann.

[8] S.H. Clearwater, B.A. Huberman, and T. Hogg. Cooperative solution of constraint satisfaction problems. *Science*, 254:1181–1183, 1991.

[9] S.H. Clearwater, B.A. Huberman, and T. Hogg. Cooperative problem solving. In B. Huberman, editor, *Computation: The Micro and the Macro View*, pages 33–70, Singapore, 1992. World Scientific.

[10] P.R. Cohen. *Empirical Methods for Artificial Intelligence.* MIT Press, Cambridge, MA, 1995.

[11] S.A. Cook. The complexity of theorem proving procedures. In *Proceedings, Third Annual ACM Symposium on the Theory of Computing, ACM*, pages 151–158, New York, 1971.

[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Cambridge, MA, 2001.

[13] K.S. Decker, E.H. Durfee, and V.R. Lesser. Evaluating research in cooperative distributed problem solving. In M.N. Huhns, editor, *Distributed Artificial Intelligence, Volume II*, pages 487–519. Pitmann, 1989.

[14] K.S. Decker and V.R. Lesser. Designing a family of coordination algorithms. In *13th International Workshop on Distributed Artificial Intelligence*, pages 65–84, Lake Quinalt, Washington, 1994.

[15] E.H. Durfee, V.R. Lesser, and D.D. Corkill. Cooperation through communication in a distributed problem solving network. In M.N. Huhns, editor, *Distributed Artificial Intelligence*, pages 29–58, Los Altos, California, USA, 1987. Morgan Kaufmann, Inc.

[16] E.H. Durfee and Y. So. The effects of runtime coordination startegies within static organizations. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[17] M. Garey and D. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[18] F. Glover. Tabu search – part 1. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[19] E.A. Hansen and S. Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1-2):139–157, 2001.

[20] T. Hogg and C.P. Williams. Solving the really hard problems with cooperative search. In *Proceedings of AAAI93*, pages 231–236. AAAI Press, 1993.

[21] E.J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *In Proc. 3rd AAAI Workshop on Uncertainty in Artificial Intelligence*, pages 429–444, 1987.

[22] R.M. Karp. Reducibility among combinatorial problems. In J.W. Thatcher and R.E. Miller, editors, *Complexity of Computer Computations*. Plenum Press, 1972.

[23] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[24] W.A. Kornfeld and C.E. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man and Cybernetics*, 11:24–33, 1981.

[25] L.A. Levin. Universal search problems (in russian). *Problemy Peredachi Informatsii*, 3(9):265–266, 1973.

[26] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. Technical report, Stanford University, 2002.

[27] S. Lin. Computer solutions of the traveling salesman problem. *BSTJ*, 44(10):2245–2260, 1965.

[28] S. Lin and B.W. Kernighan. An effective heuristic for the traveling-salesman problem. *Oper. Res.*, 21:495–516, 1973.

[29] A.K. Mackworth. Constraint satisfaction. In S. Shapiro and D. Eckroth, editors, *Encyclopedia of A.I.*, pages 205–211. John Wiley and Sons, 1987.

[30] D.L. Maremen and V. Lesser. Problem structure and subproblem sharing in multi-agent system. In *3rd International Conference on Multi-Agent Systems*, 1998.

[31] G. Mehmet. From 3-sat to 2+p,3-sat, 2001.

[32] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 52:161–205, 1992.

[33] J.E. Moody. The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In J.E. Moody, S.J. Hanson, and R.P.

Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 847–854, 1992.

[34] A. Newell and H. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[35] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization.* Prentice-Hall, Inc, Englewood Cliffs, NJ, 1982.

[36] D.C. Parkes. Bounded rationality. Technical report, Univ. of Pennsylvania, 1997.

[37] D.C. Parkes and B.A. Huberman. Multi-agent cooperative search for portfolio selection. *Games & Economic Behavior*, 35:124–165, 2001.

[38] S. Redner. Random multiplicative processes; an elementary tutorial. *American Journal of Physics*, 58(3):267–273, 1990.

[39] E. Rich and K. Knight. *Artificial Intelligence, 2nd Edition.* McGraw-Hill, New York, New York, 1991.

[40] S. Russell and E. Wefald. *Do the right thing: studies in limited rationality.* Artificial Intelligence. MIT Press, 1991.

[41] S. Russell and E. Wefald. Principles of metareasoning. *Artificial Intelligence*, 49:361–395, 1991.

[42] B. Sellman, D.G. Mitchell, and H.J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

[43] M. Stone. Asymptotics for and against cross-validation. *Biometrika*, 64:29–35, 1977.

[44] T.A.Montgomery and E.H.Durfee. Search reduction in hierarchical distributed problem solving. *Group Decision and Negotiation*, 2:301–317, 1993.

[45] Y.K. Tsji and A.V. Gelder. Dimacs cnf parser. World Wide Web, April 2004. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances.

[46] M.P. Wellman. Market-oriented programming: Some early lessons. In S. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation.* World Scientific, 1996.

[47] C.P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.

[48] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), 1998.