Better Than The Best: The Power of Cooperation

Tad Hogg Bernardo A. Huberman

1992 Lectures in Complex Systems, pp. 165–184, Addison-Wesley 1993

Abstract

We show that when agents cooperate in a distributed search problem, they can solve it faster than any agent working in isolation. This is accomplished by having agents exchange hints within a computational ecosystem. We present a quantitative assessment of the value of cooperation for solving constraint satisfaction problems through a series of experiments. Our results suggest an alternative methodology to existing techniques for solving constraint satisfaction problems in computer science and distributed artificial intelligence.

1 Introduction

The development of parallelism in computation allows a number of agents to share the work required to solve computational problems. The potential speedup offered by this approach has led to a large effort devoted to the design of parallel algorithms and architectures. In spite of its obvious advantages however, the effective use of concurrency is fraught with difficulties. Most of them stem from the fact that the experience gained from programming single processor machines cannot be simply extrapolated to large number of computational agents. This is because parallel computing involves a number of new issues: how tasks can be usefully divided among many agents, how one program can exploit the knowledge generated by another, and how the agents can communicate efficiently with each other. These issues are particularly important for large scale distributed processing in which individual agents operate largely without central controls. If the task to be performed is easily decomposed into fairly independent subtasks, requiring little communication, a parallel implementation is relatively easy. However, this is not always possible, and also misses the potential of using communication to significantly help with individual subtasks.

Some insights into how these issues can be effectively addressed for more complex cases can be gained from studying the way human societies go about solving problems of collective interest. Although the individuals differ from these computational agents in many important aspects, they nevertheless face the same general problems of coordination and communication described above. In human societies, the benefit of cooperation underlies the existence of firms, scientific and professional communities, and the use of committees charged with solving particular problems. It is often observed that groups of people can solve a problem more effectively than any single individual acting alone. This suggests implementing those mechanisms that seem to work among humans in a computational context.

The existence of computational ecologies [10] provides a natural framework for using these methods because they share a number of key features in common with human societies. These include asynchronous independent agents that solve problems from their local perspective involving uncertain and delayed information that they can retrieve from the system. A number of attempts at collective problem solving from this perspective have been made. These include work by several authors who have pointed out the beneficial effects of cooperation on hard problems by constructing models in which agents cooperate to accomplish a task [2, 5, 8, 14, 19].

It is important to understand what is meant by cooperation in a computational context. Cooperation involves a collection of agents that interact by communicating information, or *hints*, to each other while solving a problem. The most natural way to think of cooperation is as a collection of independent processes, possibly running on separate processors. However, it is always possible to have a single computational process that, in effect, multiplexes among the procedures followed by this diverse set of agents. In this way, a single agent could also obtain the benefit of cooperation discussed here. This ability of one computational

process to emulate a collection of other processes is quite distinct from other cases of cooperation, e.g., human societies, where individuals have differing skills that are not easily transferred to others. What is most important to the increase in performance is the diversity of approaches available by having many agent processes.

The information exchanged between the agents may be incorrect at times, and should alter the behavior of the agents receiving it. An example of cooperative problem solving is the use of a genetic algorithm [6] to find states of high value in some space of possibilities. In a genetic algorithm members of a population of states exchange pieces of themselves or mutate to create a new population, often containing states of high value. Another example is a neural network, where the output of one neuron affects the behavior of the neuron receiving it.

In what follows we will concentrate on a particular type of computational task, that of search. This is an important general task which arises for problems in which no algorithmic method is known for directly constructing a solution. Instead one must examine a large number of alternative candidate states in order to identify a satisfactory solution. Typically, the number of states to consider grows exponentially as larger problems are considered, making these problems considerably more difficult than, say, many numerical operations such as linear algebra or solution of differential equations whose computational cost generally grows as a fairly low-degree polynomial as problems scale up. Because of the huge number of states to consider in a search, many heuristic methods have been developed to guide the selection of states to consider. While not always correct, by guiding the search toward states that are more likely to lead to solutions, they can considerably reduce the time required to find a solution. Most heuristics are meant to improve individual searches. By contrast, the cases that we will discuss highlight the potential of cooperative methods which can be thought of as heuristics in which information obtained by one agent is used to guide the search of another. We also present a number of more practical issues that arise in applying cooperation to problems in computer science and distributed artificial intelligence [5].

As a concrete illustration of the value of cooperation for search, we solve discrete constraint satisfaction problems. These are problems in which values from a finite set must be assigned to a finite set of variables such that a number of conditions (the constraints) are satisfied. Constraint satisfaction problems lie at the heart of human and computer problem solving [13, 16, 18]. Examples are scheduling, navigating through a maze, and crossword puzzles, to name a few. A *complete state* in the search is a set of assignments for all the variables and a *partial state* has only some of variables assigned.

To evaluate the usefulness of cooperation in computational problems, we examine its behavior for two specific problems. At one extreme, cryptarithmetic with a simple individual search method, shows how even very simple methods can benefit from an exchange of information. By contrast, our second example, graph coloring, a computationally hard problem, illustrates how simple hints can be used in conjunction with an effective heuristic search method.

2 Cooperative Searches

A simple explanation of the success of cooperation is given by observing that hints change the way different agents find the solution by combining them with their own current state. Although not always successful, those cases in which hints do combine well allow the agent to proceed to a solution by searching in a reduced space of possibilities. Even if many of the hints are not successful, this results in a larger variation of performance and hence can still improve the performance of the group when measured by the time for the first agent to finish.

The speed at which an agent can solve the problem depends on the initial conditions and the particular sequence of actions it chooses as it moves through a search space. This sequence relies on the knowledge, or heuristics, that an agent has about which state should be examined next. The better the agent is able to utilize the heuristics, the quicker it will be able to solve the problem. When many agents work on the same problem, this knowledge can include hints from other agents suggesting where solutions are likely to be.

Cooperative search methods are based on modifying individual search methods. A useful distinction is whether a method is *complete* or *incomplete*. Complete methods systematically examine states and are guaranteed to either eventually find a solution or terminate when no solution exists. By contrast, incomplete methods explore more opportunistically and may miss some states in the search space; hence they can never

guarantee a solution does not exist. For parallel searches, a further issue is whether to split the search space among the agents. In the simplest case, each agent examines the entire search space. However, this can mean a single state is examined by more than one agent during the search. This can be avoided by partitioning the search space into disjoint parts and assigning one to each agent. In this partitioned search, agents only examine states in their assigned part of the space thus avoiding unnecessary duplicate examination of the states. Restricting each agent to examine a state at most once, as well as partitioning the search space so that a state is not examined by more than one agent, improve performance somewhat, but far less than the enhancement due to cooperation [3].

2.1 The use of hints

There are a number of search methods an individual agent can use to solve a problem, as well as a variety of methods for combining the partial information obtained from other agents. These choices determine if hints build on each other and if so, how does the search improves.

2.1.1 searching with complete states

The most straightforward search method is generate and test. In this case, at each step an agent generates a complete state and tests whether it is a solution. This generation can be done in a simple pre-specified order or new states can be generated randomly. In random generation, states can be selected completely at random (which we refer to as random selection with replacement) or the selection can be made only from states that have not yet been examined. The latter case avoids some unnecessary search and guarantees the search will terminate after all search states are examined, but does introduce an additional requirement of storing previously examined states and the cost of checking that they are not subsequently generated.

Other restrictions on the generation of new states are possible as well. For instance, the assignments to all the variables can be replaced in one step (which we refer to as "jumping" around the search space) or some assignments can remain unchanged, with the extreme case being a change to only a single assignment ("walking"). Walking rather than jumping through the space preserves the property that an agent near or far from a solution is still fairly near or far after one step.

There are more sophisticated methods that share the same basic strategy, i.e., start from some randomly selected initial state make a series of small adjustments to the state attempting to satisfy all the constraints. If these adjustments do not produce a solution, a new initial state is selected. Examples of this strategy include simulated annealing [12], heuristic repair [17], as well as simple hill climbing. By contrast, generate and test makes no adjustments, and simply tests the initial state itself.

With this search strategy, if hints are only used to guide the selection of the initial state and each new hint completely overwrites the old state, there will be no build up of progress from one hint to another. Alternatively, if the new hint modifies just part of the state, then successive hints could correspond to a kind of random walk in the state space in which there is (at least for the lucky agents) an overall bias to move successive initial states closer to a solution which is eventually found by the local adjustments.

2.1.2 constructing solutions from partial states

Other search methods rely on a more systematic exploration of the space, attempting to construct a complete solution by incrementally extending partial solutions. Such a hierarchical construction of a solution, combined with some backtrack scheme when further progress is impossible, allows for pruning regions of the search space that would be unproductive. With this depth-first search method, some ordering of the variables is selected (e.g., either fixed in advance or chosen randomly) and partial states are constructed using this ordering until a full solution is found or enough assignments are made to violate one of the constraints, indicating that there is no solution corresponding to this partial state. Where these constraint violations occur well before all assignments have been made, backtracking avoids a considerable amount of unnecessary search.

A simple illustration of the resulting tree structured search is shown in Fig. 1a. Specifically this is for a constraint problem with three variables, v_1, v_2, and v_3, each of which can take on the values 1 or 2. The nodes in the tree represent the variables, and the links from a node represent the two choices for the values to assign to that node's variable (corresponding to the value 1 for the left branch and 2 for the right

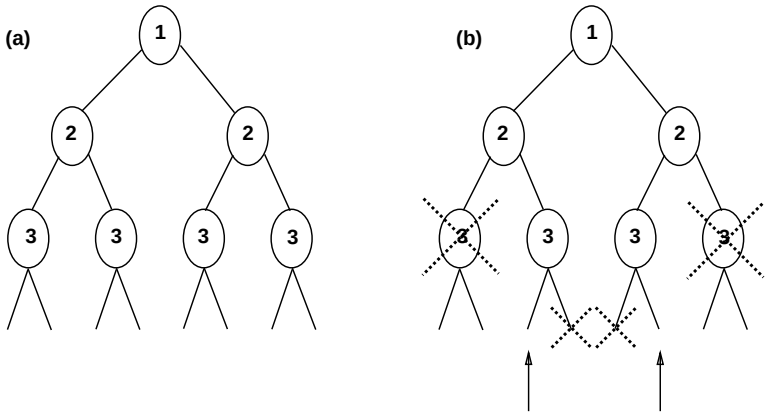

Figure 1: a) Illustration of the tree structured search space resulting from three variables, v_1, v_2, and v_3, (corresponding to the nodes in the tree) each with two possible values (corresponding to the branches). Searches generally start at the top of the tree and examine successive branches until a complete state (corresponding to a leaf, at the bottom of the tree) that satisfies the constraints is found. b) The pruning of the search tree for the constraint problem $\{v_1 \neq v_2, v_2 \neq v_3\}$. The crosses indicate those states that violate one or more of the constraints. The arrows point to the leaves corresponding to solutions, i.e., $\{v_1 = 1, v_2 = 2, v_3 = 1\}$ and $\{v_1 = 2, v_2 = 1, v_3 = 2\}$.

branch). The leaves of the tree correspond to complete search states in which each variable has a value. For example, the leftmost leaf corresponds to the assignments $\{v_1 = 1, v_2 = 1, v_3 = 1\}$. Partial states, in which some variables are not assigned, are found higher in the tree (in the ordering illustrated here, variable v_1 is assigned first, v_2 next and v_3 last). Adding consideration of these partial states means that these search methods could potentially examine more states than the those that use only complete states. However, this increase in total states is usually more than offset by the ability to prune many states at one time high in the tree. This pruning is illustrated in Fig. 1b for the constraints $\{v_1 \neq v_2, v_2 \neq v_3\}$, i.e., the values for the first two variables, and the last two, are required to be different. For example, the leftmost pruned node is due to the partial state $\{v_1 = 1, v_2 = 1\}$ which already violates the first constraint so there is no need to consider possible values for the third variable.

These basic methods can be improved with the use of heuristics to guide the selection of states. An important class of heuristics uses information obtained in prior steps of the search. Such heuristics are fairly readily modified for cooperative search allow us to directly evaluate the effect of cooperation. Specifically, in a noncooperative search, an agent using such a method could only use information that it had previously found itself, while cooperative search allows the agent to use information found by others as well.

Hints can naturally be used to guide the ordering of backtrack choices, which can be viewed as moving in a tree structure. When a hint gives the correct choice for an agent, the remaining choices are, in effect, pruned. More generally, these hints can give large partial solutions from other regions of the search space. This is the case, for example, when putting together a puzzle by working on different regions and then combining them. Genetic algorithms are another instance of this general strategy.

2.1.3 diversity

A more interesting possibility is to have a group of agents use different search methods. Such diverse communities are particularly well-suited for the use of cooperation since a particular agent may not be able to utilize all the information it generates, whereas another agent, using a different strategy, can. For example, a systematic backtrack search method may rapidly find promising regions of the search space but take a long time to finally reach a solution when this requires some changes to choices made early in the backtracking. This could be quickly fixed by other methods that make adjustments opportunistically with no prespecified ordering. Thus the exchange of information among methods can improve performance beyond that possible without cooperation.

The effectiveness of these hints will depend on the search choices made by the agents. For example, as the search progresses, agents may find better partial solutions so that hint quality increases over time.

Conversely, as agents get near the solution, hints become less important since they will tend to duplicate partial solutions already found, or in fact incorrect hints may even become more detrimental.

2.2 Implementation issues

From this general discussion of using hints with various search methods, we now turn to a number of implementation issues and how they were resolved in our experiments. While there are many ways to address these issues, we made fairly simple choices. We can expect further improvements from more sophisticated use of hints, but the choices made here illustrate the potential of this method and have many direct correspondences with a wide range of constraint satisfaction problems. As a note of caution in developing more sophisticated strategies, the choices made should tend to promote high diversity among the agents [7, 9] so there will be many opportunities to try hints in different promising contexts. This means that some choices that appear reasonable when viewed from the perspective of a single agent, could result in lowered performance for the group as a whole.

2.2.1 cooperative search

There are two basic steps in implementing a cooperative search based on individual algorithms. First, the algorithms themselves must be modified to enable them to produce and incorporate information from other agents, i.e., read and write hints. We should note that the first step may, in itself, change the performance of the initial algorithm or its characteristics (e.g., changing a complete search method into an incomplete one). Since this may change the absolute performance of the individual algorithm, a proper evaluation of the benefit of cooperation should compare the behavior of multiple agents, exchanging hints, to that of a single one running the same, modified, algorithm, but unable to communicate with other agents. In that way, the effect of cooperation, due to obtaining hints from other agents, will be highlighted.

The second step concerns decisions as to exactly what information to use as hints, when to read them, etc. must be made. The hints consist of any useful information concerning regions of the search space to avoid or likely to contain solutions. A simple choice for constraint satisfaction problems is to use partial solutions, i.e., partial states whose assignments do not violate any constraint. We must also specify the organizational structure, i.e., which agents communicate with each other. In our experiments, all hints were written to a central blackboard, so each agent could access the results of any other agent. Hierarchical organizations more suitable to larger populations have also been studied [3].

The next major question is when during its search should an agent produce a hint. Generally, agents should tend to write hints that are likely to be useful in other parts of the search space. Possible methods to use include only writing the largest partial solutions an agent finds (i.e., at the point it is forced to backtrack) or only if the hint is comparable in size to those already on the blackboard.

Another set of complementary questions has to do with the time at which an agent decides to read a hint from the blackboard, which one should it choose and how should it make use of the information for its subsequent search. Once again, there are a number of reasonable choices which have different benefits in avoiding search and costs in their evaluation, as well as more global consequences for the diversity of the agent population. For instance agents could select hints whenever a sufficiently good hint is available, or whenever the agent is about to make a random choice in its search method (i.e., use the hint to break ties), or whenever the agent is in some sense stuck, e.g., needing to backtrack, or at a local optimum of a hill climbing search method. For deciding which available hint to use, methods range from random selection [2] to picking one that is a good match, in some sense, to the agent's current state.

A final issue concerns the memory requirements for the hints. To avoid the potential of an unbounded growth in the size of the blackboard, one can limit the number of hints it could store. Once this limit is reached, some hints have to be discarded. For our experiments, the oldest (i.e., added to the blackboard before any others) of the smallest (i.e., involving the fewest assignments) hints were overwritten with new hints. We found that relatively small blackboards were sufficient to obtain significantly better performance than the independent searches.

2.2.2 performance measures

Before turning to our experimental comparison of cooperating and non-cooperating agents, we must specify how the performance of a group of agents is to be measured. The appropriate performance measure depends on the nature of the problem [3]. In many cases, one is interested in finding a single solution to the problem and each agent is individually capable of finding a complete solution. This means that the search is completed as soon as one agent finds a solution. The appropriate overall performance measure is then just the time required until some agent in the group finds a solution.

As a simple performance criterion we use the number of search steps required for the first agent to find a solution. However, we should note that this ignores the additional overhead involved in selecting and incorporating hints. Including such costs doesn't change the qualitative observation of cooperative improvement in simple cases [3]. Whether this remains true for the more sophisticated search methods remains open and is ultimately best addressed by comparing execution times of careful implementations of the algorithms. Moreover, an actual parallel implementation would also face possible communication bottlenecks at the central blackboard though this is unlikely to be a major problem with the small blackboards considered here due to the relatively low reading rate and the possibility of caching multiple copies of the blackboard which are only slowly updated with new hints. Nevertheless, the improvement in the number of search steps reported below, as well as comparisons of the execution time of our unoptimized code, suggest the cooperative methods are likely to be beneficial for large, hard problems.

3 Cryptarithmetic

For our first example, we consider a very simple search method, used in the familiar problem of solving cryptarithmetic codes. These problems require finding a unique digit assignment to each of the letters of a word addition so that the numbers represented by the words add up correctly. An example is the sum: DONALD + GERALD = ROBERT, which has one solution, given by $A = 4$, $B = 3$, $D = 5$, $E = 9$, $G = 1$, $L = 8$, $N = 6$, $O = 2$, $R = 7$, $T = 0$. In general, if there are n letters then there are 10^n possible states. However, the requirement of a unique digit for each letter means that there are $\binom{10}{n}$ ways to choose the values and $n!$ ways to assign them to the letters, which reduces the total number of search states to $n!\binom{10}{n} = 10!/(10-n)!$. Thus the above example, which has 10 letters has 10! states in its search space.

Solving a cryptarithmetic problem involves performing a search. Although clever heuristics can be used to rapidly solve the particular case of cryptarithmetic [15], our purpose is to address the general issue of cooperation in parallel search using cryptarithmetic as a simple example. Thus we focus on simple search methods, without clever heuristics that can lead to quick solutions by a single agent. This is precisely the situation faced with more complex constraint problems where searches remain long even with the best available heuristics.

The basic search paradigm we have used in the cryptarithmetic problem is random generate and test with replacement. We used hints consisting of letter-digit assignments in columns that add correctly. These hints were posted to a blackboard. Agents used the available hints to select their next state. In a noncooperative search, an agent using this method could only use hints that it had previously found so that each agent had a separate blackboard. Cooperative search allowed the agent to use hints found by others as well, using a single blackboard.

For each search step, an agent chooses a hint randomly from the blackboard and replaces assignments in its current state with those specified by the hint. If there are no hints, it chooses a random letter-digit assignment using random generate and test. Once the agent obtains the new state it generates and posts all possible hints from its state, if any. Thus, assignments that work for more than one column are posted as several different hints. When random states are generated by jumping, rather than single letter replacements, there is a greater possibility of generating more hints faster but at the expense of frequently overwriting partially correct states.

As an example of this search method consider an agent solving the problem AB + AC = DE. This problem has $10!/5! = 30240$ possible states and 144 solutions (determined by exhaustive search). In the first step, each agent selects a random set of letter-digit assignments such that no digit is assigned to more than one letter. Suppose the letter-digit assignments, or state, of the first agent are: $A = 4$, $B = 2$, $C = 7$, $D = 3$, $E = 9$. In this case the assignments do not correspond to a solution since $42 + 47$ does not equal

39. However, the rightmost column, $B + C = E$ $(2 + 7 = 9)$, does add up correctly so that the agent's state is *partially* (or *locally*) correct. Partial correctness includes cases where a carry has been brought over from the previous column or may be sent to the next column. Note that although a particular column may be locally correct, it may not lead to a solution. In this example, the agent has one column correct (3 letters: B, C and E). If these letter assignments do lead to a solution then there are only two letters that need to be assigned from 7 possible choices. Thus the agent went from a search space of size 30240 to one of $7!/5! = 42$ states, a reduction by a factor of nearly 1000. In a cooperative search, this reduction could also be used by other agents, perhaps in other regions of the search space where this hint is more successfully used.

3.1 Results

As a specific case, we examine the effect of cooperation for groups of 100 agents solving the problem WOW + HOT = TEA. This problem, with 6 distinct letters, has 151200 search states and 82 different solutions. The comparative performance of cooperation is illustrated in Table 1.

search method	relative time	relative deviation
cooperative	1	.87
independent, with memory	7.5	.49
independent, no memory	23.9	1

Table 1: Average performance of 10 trials of 100 agents solving WOW + HOT = TEA for different search methods. The relative time is the average time required for the first agent of the group to find a solution, divided by the average time required for the cooperative case. The relative deviation is the standard deviation in the time to first solution divided by the average time for each method. The benefit of cooperation, i.e., sharing hints among the agents, is shown by the comparison between the cooperative case and that where the agents used the same method, i.e., had memory, but did not share it. The last row shows, for comparison, the theoretical performance of the unmodified random generate-and-test method.

It also worthwhile to note the effect of cooperation as the problems become more difficult. One way of measuring the difficulty of problems is by the ratio of the number of states in the search space, T, to the number of solutions, S. Table 2 below shows the relative speed for the first finisher of 100 agents for four problems of vastly different complexities. The data for the cooperative case came from experiments while the behavior of the noncooperative case was obtained theoretically [3] by noting that each random generate and test step has probability S/T to find a solution. Note that as the problem becomes more difficult the importance of cooperation and use of memory in speedup is increased. The relative increase becomes even more startling when one considers that the fraction of hints posted on the blackboard that are subsets of *any* of the solutions (not necessarily the one found first) decreases as the problems become more complex. Thus the high performance is due to some agents finding combinations of hints that lead to solutions even though the full hints are rarely part of a solution.

Another way of studying the effect of cooperation vs. problem complexity is to vary the effectiveness of the search performed by the agent itself, i.e., the *self-work*, without utilizing the hints from the other agents. For example, suppose that when the agents are not using hints they perform a depth-first backtrack search, each using a randomly selected ordering of the variables. During the depth-first search the agents have the opportunity to prune partial states which do not lead to any solution. For example, if some columns do not add up correctly there is no point in considering assignments to uninstantiated letters for this state. Whenever a hint is used it overwrites the current partial state, in the same manner as for the agents using simple generate and test, so that there may be very large jumps through the search space and the resulting search is no longer complete. We can simulate the effect of this pruning by probabilistically pruning partially assigned states that are known not to lead to a solution. (We can do this with cryptarithmetic by generating all the solutions ahead of time.) When the probability of pruning is small this corresponds to difficult problems because the agents must instantiate nearly all the letters of an incorrect assignment before pruning. The results of this study, which are shown in Fig. 2, show the greater relative importance of cooperation for harder problems.

problem	ratio of speeds	T/S	fraction of hints that are subsets of solutions
AB + AC = DE	7	210	0.9–1.0
WOW + HOT = TEA	45	1844	0.5–0.6
CLEAR + WATER = SCOTT	145	181440	0.1–0.2
DONALD + GERALD = ROBERT	315	3628880	0.004

Table 2: Scaling of cooperative performance for cryptarithmetic problems of increasing difficulty for 100 agents. The second column is the ratio of speeds between the cooperative search and that of independent agents with no memory, and represents an average over about 100 trials for the first three cases, and a few trials for the last case. (Note that the entry for the second problem is 45, compared to 23.9 of Table 1, since this was from a separate run of the experiment and indicates the degree of statistical fluctuation in the cooperative search.) The fourth column shows the range in the fraction of hints on the final blackboard that were subsets of some solution for some of these cooperative searches. Typically these were added just before the end of the search by the agent that found the first solution.

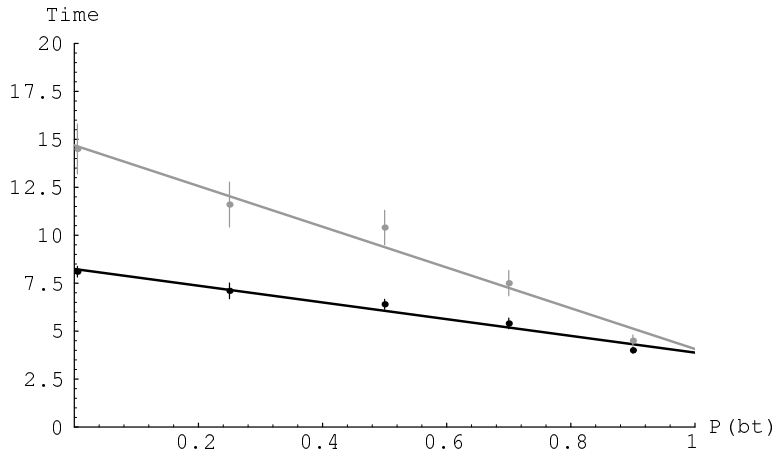

Figure 2: Cooperation works best for harder problems. Time to first solution as a function of decreasing problem hardness. Specifically, the plot shows the average time to first solution for 100 agents solving AB + AC = DE as a function of the probability of pruning, $P(bt)$, a state that is known not to lead to a solution. The left side of the plot corresponds to "hard" problems where pruning of the search space is very poor, and the right side of the plot corresponds to "easy" problems where pruning is very effective. The light line is for the case of noncooperating agents, in this case a depth-first search. The dark line is for the case where the agents spend 80% of their time doing depth-first self-work and 20% cooperating, i.e., using hints from the blackboard. The lines show the best linear fits to the data. The data points correspond to the average solution time from 50–100 runs. The error bars are the error of the mean.

In summary, these results show the value of cooperation in solving a relatively easy constraint satisfaction problem using very simple search methods. There remains the question of how this methodology can be used in solving harder problems.

4 Graph Coloring

The distinction between easy and hard problems is important in determining the feasibility of computations, and a great deal of research has been devoted to it [4]. An important distinction among problems is based on how rapidly the number of elementary operations required to solve them grows as the problems are scaled up to larger instances. Particularly whether the scaling is dominated by polynomial or exponential growth. An elementary operation could typically be an arithmetic operation for a numerical problem or the examination of a single state in a search problem.

A surprising result is that sometimes the difference between these classes of problems is extremely subtle.

For instance, consider two given nodes of a graph, which consists of a number of nodes and links between them. The problem of deciding whether there is a path between them, i.e., a series of distinct linked nodes that connect the two given nodes, whose total length is less than a given bound M can be solved in polynomial time with respect to the number of nodes in the graph. On the other hand, the similar problem of whether there is a path with length greater than M has no known solution in polynomial time. However, if one is given a path whose length is claimed to be larger than M so that such a path exists, there is an algorithm that will quickly verify that the answer is correct, namely to count the links in the path and check that the length is indeed larger than M. This procedure operates in time which is linear in the length of the path which in turn is no more than the total number of nodes in the graph. This is an example of a simple yes or no problem in which an affirmative answer can be verified in polynomial time, even though there may be no way to actually construct the answer readily.

Such problems are said to belong to the class NP (for nondeterministic polynomial). Conceptually, these problems can be rapidly solved by a *nondeterministic* algorithm, i.e., one which can somehow guess the correct answer, and then rapidly verify it. Actual implementations, however, are deterministic and appear to be unable to solve the problem in polynomial time. Note that NP includes all problems in P, the class of problems for which there is a deterministic polynomial time algorithm. Whether NP is in fact the same as P remains an open question.

Although the class NP is based on the ability to easily verify solutions, it can also be shown to include many optimization problems whose solutions would seem more difficult to check. For instance, corresponding to the path problems mentioned above are the optimization problems of determining the *shortest* and *longest* paths between the vertices, respectively. The shortest path can be found in polynomial time, but there is no known rapid solution (i.e., short of checking all possible paths) for determining the longest one. In the latter case, being given a path which is claimed to be the longest is difficult to directly verify since not only must its length be determined but it must also be compared to all other possible paths. However, this latter problem does in fact belong to NP because it can be transformed into a series of verifiable problems involving specified bounds on the lengths such that the total time to verify all the subproblems is still polynomial. Another example of such a problem is the travelling salesman problem, in which a collection of cities and distances between them is given, and the task is to find the shortest path which visits each city. Among the problems in the class NP, some are known to be at least as hard, up to a polynomial factor, as any other problem in the class. In this sense, these so-called NP-complete problems constitute the most difficult problems in NP. As far as currently known, the solution cost grows exponentially in the worst case as the size of the problem increases.

As our second example of cooperative search, we consider such an NP-complete problem, that of graph coloring. The problem consists of coloring the nodes in a graph, from a limited set of colors, in such a way that no two adjacent nodes (i.e., nodes linked by an edge in the graph) have the same color. An example of such a colored graph is shown in Fig. 3. Graph coloring has received considerable attention and a number of search methods have been developed [11]. Paradoxically, although their are some graphs that are very hard to color, among graphs of a given size there is considerable variation in the difficulty of finding a solution, and most of them can be colored (or determined to have no coloring) quite rapidly with existing heuristic methods.

For this problem, the average degree of the graph γ (i.e., the average number of edges coming from a node in the graph) distinguishes relatively easy from harder problems, on average. For the case of 3-coloring, (i.e., when 3 different colors are available) which we focus on in this paper, the region of hardest problems is empirically observed to occur near [1] $\gamma = 5$. We used the Brelaz search heuristic [11] which effectively finds colorings by assigning most constrained nodes first (i.e., those with the most distinct colored neighbors), breaking ties by choosing nodes with the most uncolored neighbors (ties that remain after applying this criterion are broken randomly). For the chosen node, the smallest available color is examined first, with successive colors considered when the search is forced to backtrack. As an example, for the graph shown in Fig. 3, this heuristic would first color the central node with four neighbors, then randomly select one of those neighbors to color (since after the first node is colored, each of its neighbors will have the same number of uncolored neighbors), etc; continuing until a complete coloring is found or the search is forced to backtrack because no consistent coloring is possible for the next node selected. By focusing attention on the most constrained nodes first, this will generally rapidly determine if a proposed partial coloring is inconsistent, thus pruning unproductive searches high in the tree and avoiding substantial wasted effort. This heuristic

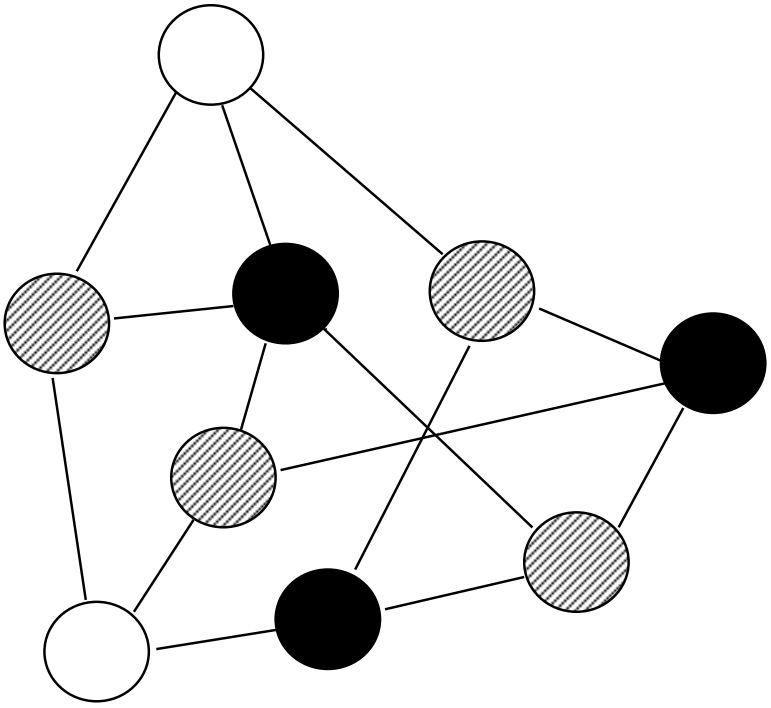

Figure 3: A graph with nine nodes colored with three colors (black, gray, and white) such that no two adjacent nodes have the same color.

is considerably more efficient than simple generate-and-test or backtracking with a random ordering of the nodes.

To generate a collection of hard problems we examined a large number of random graphs. Trivial cases with underconstrained nodes were removed by ensuring each node had at least three edges. Notice that nodes with fewer edges are underconstrained in that they can always be colored differently from the nodes they are linked to when there are three available colors. The resulting graphs were searched repeatedly with the Brelaz heuristic, and only those with high search cost were retained. Moreover, to correspond with the cooperative methods used for the cryptarithmetic example and simplify the use of hints, we considered only graphs that did in fact have solutions. In addition to high average cost for solution with the Brelaz heuristic, these graphs also had a large variance in the cost of repeated searches due to different choices made at tie points. This variance gives rise to improved performance of using multiple independent searches in parallel, stopping when the first one finishes. The experiments reported here show the additional benefit from exchanging hints.

At any point in a backtracking search, the current partial state is a consistent coloring of some subset of the graph's nodes. When writing a hint to the blackboard, the Brelaz agents simply wrote their current state. Specifically, each agent independently wrote its current state at each step with a fixed probability q.

Each time the agent was about to expand a node in its backtrack search, it would instead, with probability p, attempt to read a compatible hint from the blackboard, i.e., a hint whose assignments were: 1) consistent with those of the agent (up to a permutation of the colors[1]) and 2) specified at least one node not already assigned in the agent's current state. Frequently, there was no such compatible hint (especially when the agent was deep in the tree and hence had already made assignments to many of the nodes), in which case the agent continued with its own search.

When a compatible hint was found, its overlap with the agent's current state was used to determine a permutation of the hint's colors that made it consistent with the state. This permutation was applied to the remaining colorings of the hint and then used to extend the agent's current state as far as possible

[1]We thus used the fact that, for graph coloring, any permutation of the color assignments for a consistent set of assignments is also consistent.

(ordering the new nodes as determined by the Brelaz heuristic), and retaining necessary backtrack points so that the overall search remained complete. In effect, this hint simply replaced decisions that the Brelaz heuristic would have made regarding the initial colors to try for a number of nodes. Thus, this amounts to a fairly conservative use of hints, compared to the backtrack search for cryptarithmetic in Fig. 2; where hints overwrote the agent's state without retaining backtrack points.

4.1 Results

The experimental results show the benefit of cooperation for graph coloring using a variety of search methods [8]. In Fig. 4, we compare the performance of a group of 10 independent and 10 cooperative agents, all using the same Brelaz search algorithm described above. We generated a set of graphs whose search cost was one to three orders of magnitude more than the minimum possible. To highlight the benefit of cooperation, beyond that achieved with multiple runs of independent agents, we compare the cooperative case with the same number of agents running independently. Note that in both cases, cooperation gives better performance than simply taking the best of 10 independent agents. Moreover, cooperation appears to be more beneficial as problem hardness (measured by the performance of a group of independent agents) increases. We obtained a few graphs of significantly greater hardness than those shown here which confirm this trend.

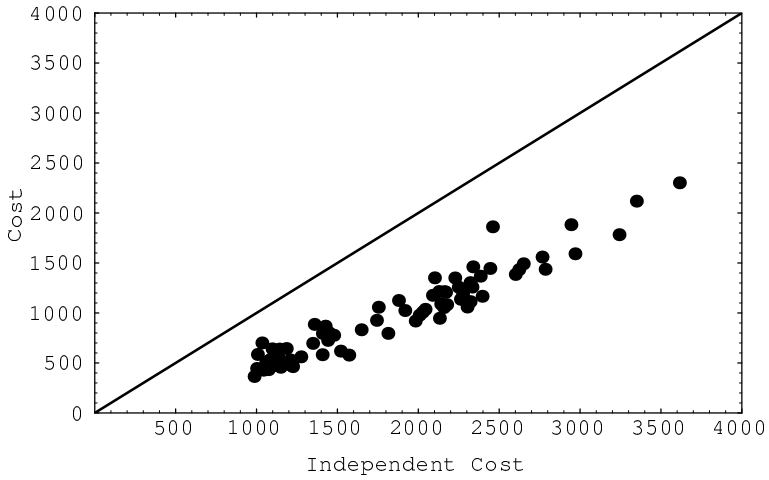

Figure 4: Performance of groups of 10 cooperating agents using the Brelaz search method on a range of graphs vs. the performance of a group of 10 independent agents using the same method. The performance values used for each graph are the median, over 10 trials, of the search steps required for the first agent in the group to find a solution. For comparison, the line shows the performance of the independent agents. In these experiments, the blackboard was limited to hold 100 hints, and we used $p = 0.5$, $q = 0.1$ and graphs with 100 nodes.

example	independent search cost	T/S	avg. hint size	fraction of hints that are subsets of solutions
A	3614	9×10^{42}	64.2	0.02
B	985	7×10^{41}	42.3	0.05

Table 3: Extreme cases from Fig. 4. Note that the search space for this problem has $3^{100} \approx 5 \cdot 10^{47}$ states, giving much larger values of T/S than for the cryptarithmetic problems. The number of solutions was found by exhaustive search. The fourth column shows the average size (i.e., number of colored nodes) of the hints on the blackboard at the time the solution was found. The fifth shows the fraction that are subsets of a solution.

As with cryptarithmetic, most hints on the blackboard are not subsets of solutions. As an example, for two of the cases shown in Fig. 4, Table 3 shows the number of hints on the final blackboard (for a single

search method	relative time A	relative time B	relative deviation A	relative deviation B
cooperative	1	1	0.03	0.14
independent, with memory	1.6	2.7	0.03	0.05
independent, no memory	1.8	3.1	0.06	0.11

Table 4: Performance for the examples, A and B, given in Table 3 for different search methods. The relative time is the median time required for the first agent of the group to find a solution, divided by the median for the cooperative case. The relative deviation is the standard deviation in the time to first solution divided by the median time for each method. The benefit of cooperation, i.e., sharing hints among the agents, is shown by the comparison between the cooperative case and that where the agents used the same method, i.e., had memory, but did not share it. The last row shows, for comparison, the performance of the unmodified backtrack using the Brelaz heuristic. Note that the deviations for this backtrack search method are considerably smaller than for the generate and test search used for the cryptarithmetic example.

run) that are subsets of solutions. Note that unlike the cryptarithmetic case, here the blackboard is limited to 100 hints. Finally, Table 4 shows the speedup obtained for some of the graph coloring cases. These are considerably less than obtained starting from the simple generate and test search with cryptarithmetic, but are comparable to the speedup obtained with a backtrack search shown in Fig. 2.

Similar cooperative improvements are obtained for other search methods [8], including heuristic repair [17], in which changes are made to complete colorings that minimize the number of violated constraints, and a mixed group of agents in which some use the Brelaz heuristic with backtracking as described above while others use heuristic repair.

5 Discussion

We have shown how cooperating agents working towards the solution of a constraint satisfaction problem can lead to a marked increase in the speed with which they solve it compared to their working in isolation. A summary of the cases studied is shown in Table 5.

search problem	cryptarithmetic	graph coloring
individual method	random generate and test	backtracking using Brelaz heuristic
blackboard size	unlimited	100 hints, old ones overwritten with new ones
hints	digits for some letters that added correctly	consistent colors for some nodes
when to write a hint	whenever some columns added correctly	randomly with probability $q = 0.1$ at each step
when to read a hint	every step when hint was available	randomly with probability $p = 0.5$ at each step a compatible hint was available
how to use a hint	overwrite current state	extend current state

Table 5: Comparison of cooperative search methods used for cryptarithmetic and graph coloring, except that the cryptarithmetic results shown in Fig. 2 use simple backtrack and only use hints on some of the search steps.

In our implementation we defined hints in terms of information that moved the agents toward a region of the space that could have a solution. Another possibility is for hints to contain information that tends to move them away from regions that can have no solutions. More generally, any search algorithm that agents may use will have parameters that will affect the benefit of cooperation. Another consideration is when are the hints most useful for problem solving. At the beginning of a problem the hints provide crucial information for starting the agents off on a plausible course, but will usually be fairly nonspecific. Near the end of the problem however, there are likely to be many detailed hints but also of less relevance to the agents since they may have already discovered that information themselves. This suggests that typical cooperative

searches will both start and end with agents primarily working on their own and that the main benefit of exchanging hints will occur in the middle of the search.

This work suggests an alternative to the current mode of constructing task-specific computer programs that deal with constraint satisfaction problems. Rather than spending all the effort in developing a monolithic program or perfect heuristic, it may be better to have a set of relatively simple cooperating processes work concurrently on the problem while communicating their partial results. This would imply the use of "hint engineers" for coupling previously disjoint programs into interacting systems that are able to make use of each others (imperfect) knowledge.

This new methodology may be particularly useful in areas of artificial intelligence such as design, qualitative reasoning, truth maintenance systems and machine learning. Researchers in these areas are just starting to consider the benefits brought about by massive parallelism and concurrency, and our work suggest the additional benefits that could be obtained from cooperation.

In closing, we have seen how computational ecosystems can be used to solve complex problems by exploiting the benefit of cooperation in a distributed context. We believe this is just the beginning; one can envision systems where the demands of a particular task will dynamically spawn new processes to work on promising avenues while deleting those agents that are not making much progress. This will require new programming methodologies for resource allocation in these systems. The spread of these ecosystems will make it easier to program them in order to use cooperative methods for the solution of even harder problems.

6 Acknowledgments

We thank S. Clearwater and C. Williams for helpful discussions.

References

[1] Peter Cheeseman, Bob Kanefsky, and William 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.

[2] Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg. Cooperative solution of constraint satisfaction problems. *Science*, 254:1181–1183, 1991.

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

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] Les Gasser and Michael N. Huhns, editors. *Distributed Artificial Intelligence*, volume 2. Morgan Kaufmann, Menlo Park, CA, 1989.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, NY, 1989.

[7] Tad Hogg. The dynamics of complex computational systems. In W. Zurek, editor, *Complexity, Entropy and the Physics of Information*, volume VIII of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 207–222. Addison-Wesley, Reading, MA, 1990.

[8] Tad Hogg and Colin P. Williams. Solving the really hard problems with cooperative search. In Haym Hirsh et al., editors, *AAAI Spring Symposium on AI and NP-Hard Problems*, pages 78–84. AAAI, 1993.

[9] Bernardo A. Huberman. The performance of cooperative processes. *Physica D*, 42:38–47, 1990.

[10] Bernardo A. Huberman and Tad Hogg. The behavior of computational ecologies. In B. A. Huberman, editor, *The Ecology of Computation*, pages 77–115. North-Holland, Amsterdam, 1988.

[11] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May-June 1991.

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

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

[14] William A. Kornfeld. The use of parallelism to implement heuristic search. Technical Report 627, MIT AI Lab, 1981.

[15] Jean-Louis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

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

[17] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI-90*, pages 17–24, Menlo Park, CA, 1990. AAAI Press.

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

[19] H. Penny Nii, Nelleke Aiello, and James Rice. Experiments on Cage and Poligon: Measuring the performance of parallel blackboard systems. In Les Gasser and Michael N. Huhns, editors, *Distributed Artificial Intelligence*, volume 2, pages 319–383. Morgan Kaufmann, San Mateo, CA, 1989.