Collaborative Learning for Constraint Solving

Susan L. Epstein
Department of Computer Science
CUNY
epstein@cs.hunter.cuny.edu

Eugene C. Freuder
Department of Computer Science
UNH
ecf@cs.unh.edu

Abstract

Although constraint programming offers a wealth of strong, general-purpose methods, in practice a complex, real application demands a person who selects, combines, and refines various available techniques for constraint satisfaction and optimization. Although such tuning produces efficient code, the scarcity of human experts slows commercialization. The necessary expertise is of two forms: constraint programming expertise and problem-domain expertise. The former is in short supply, and even experts can be reduced to trial and error prototyping; the latter is difficult to extract. The project described here seeks to automate both the application of constraint programming expertise and the extraction of domain-specific expertise. It applies FORR, an architecture for learning and problem-solving, to constraint solving. FORR develops expertise from multiple heuristics. A successful case study is presented on coloring problems.

1. Introduction

Difficult constraint programming problems require human experts to select, combine and refine the various techniques currently available for constraint satisfaction and optimization. These people "tune" the solver to fit the problems efficiently, but the scarcity of such experts slows commercialization of this successful technology. The few initial efforts to automate the production of specialized software have thus far focused on choosing among methods or constructing special purpose algorithms [1-4].

Although a properly-touted advantage of constraint programming is its wealth of good, general-purpose methods, at some point complex, real applications require human expertise to produce a practical program. This expertise is of two forms: constraint programming expertise and problem domain expertise. The former is in short supply, and even experts can be reduced to trial and error prototyping; the latter is difficult to extract. This project seeks to automate both the application of constraint programming expertise and the extraction of domain-specific expertise.

Our goal is to automate the construction of problem-specific or problem-class-specific constraint solvers. Each solver will be a learned, collaborative "community" of heuristics appropriate for their problem or problem class. Both the way in which they collaborate and some of the heuristics themselves will be learned.

This paper reports initial results from a case study that applies a well-tested, collaborative, problem-solving architecture to the domain of constraint programming. The architecture permits swift establishment of a well-provisioned base camp from which to explore this research frontier more deeply. Section 2 presents some minimal background. Section 3 presents an initial, successful case study. Section 4 outlines further opportunities and challenges. Section 5 is a brief conclusion.

2. Background

We provide here some minimal background information on CSP's and on the *FORR* (FOr the Right Reasons) architecture. Further details will be provided on a need-to-know basis during our description of the case study.

2.1 CSP

Constraint satisfaction problems involve a set of variables, a *domain* of values for each variable, and a set of *constraints* that specify which combinations of values are allowed [5-8]. A *solution* is a value for each variable, such that all the constraints are satisfied. For example, graph coloring problems are CSP's: the variables are the graph vertices, the values are the available colors, and the constraints specify that neighboring vertices cannot have the same color. The basic CSP paradigm can be extended in various directions, for example to encompass optimization or uncertainty. Solution methods generally involve some form of search, often interleaved with some form of inference.

Many practical problems – such as resource allocation, scheduling, configuration, design, and diagnosis – can be modeled as constraint satisfaction problems. The technology has been widely commercialized, in Europe even more so than in the U.S. This is, of course, an NP-hard problem area, but there are powerful methods for solving difficult problems. Artificial intelligence, operations research, and algorithmics all have made contributions. There is considerable interest in constraint programming languages. Although we take an artificial intelligence approach, we expect our results to have implications for constraint programming generally.

Constraint satisfaction problem classes can be defined by "structural" or "semantic" features of the problem. These parameterize the problem and establish a multidimensional problem space. We will seek to synthesize specialized solvers that operate efficiently in different portions of that space.

2.2 FORR

FORR is a problem-solving and learning architecture for the development of expertise from multiple heuristics. It is a *mixture of experts* decision maker, a system that combines the opinions of a set of procedures called *experts* to make a decision [9, 10]. This approach is supported by evidence that people integrate a variety of strategies to accomplish problem solving [11-13].

A FORR-based artifact is constructed for a particular set of related tasks called a *domain*, such as path finding in mazes [14] or game playing [15]. A FORR-based program develops expertise during repeated solution attempts within a *problem class*, a set of problems in its domain (e.g., contests at the same game or trips with different starting and ending points in the same maze).

FORR-based applications have produced expert-level results after as few as 20 experiences in a problem class. Learning is relatively fast because a FORR-based application begins with prespecified, domain-specific knowledge. To some extent, a FORR-based application resembles a person who already has substantial general expertise in a domain, and then develops expertise for a new problem class. Such a person is already aware of general principles that may support expert behavior, and also recognizes what is important to learn about a new class, how to acquire that information, and how to apply it. In FORR, that information is called *useful knowledge,* and the decision principles are called *Advisors.* FORR learns *weights* to reflect the reliability and utility of Advisors.

Useful knowledge is knowledge that is possibly reusable and probably correct. In path finding, for example, a dead-end is a particular kind of useful knowledge, an *item.* Each item of useful knowledge is expected to be relevant to every class in the domain. The *values* for a particular useful knowledge item, however, are not known in advance; dead-ends, for example, must be learned, and they will vary from one maze to another. This is what is meant by *problem-class-specific* useful knowledge.

A FORR-based program learns when it attempts to solve a problem, or when it observes an external expert solve one. The program is provided in advance with a set of useful knowledge items. Each item has a name (e.g., "dead-end"), a learning algorithm (e.g., "detect backing out"), and a trigger (e.g., "learn after each trip"). A learning trigger may be set for after a decision, after a solution attempt, or after a sequence of solution attempts. When a useful knowledge item triggers, its learning algorithm executes, and the program acquires problem-class-specific useful knowledge. Note that there is no uniform learning method for useful knowledge items — in this sense, FORR truly supports multi-strategy learning.

FORR organizes Advisors into a hierarchy of tiers (see Figure 1), based upon their correctness and the nature of their response. A FORR-based program begins with a set of prespecified Advisors intended to be *problem-class-independent*, that is, relevant to most classes in the domain. Each Advisor represents some domain-specific principle likely to support expert behavior.

Each Advisor is represented as a time-limited procedure that accepts as input the current problem-solving state, the legal actions from that state, and any useful knowledge that the program has acquired about the problem class. Each Advisor produces as output its opinion on any number of the current legal actions. An opinion is represented as a *comment*, of the form

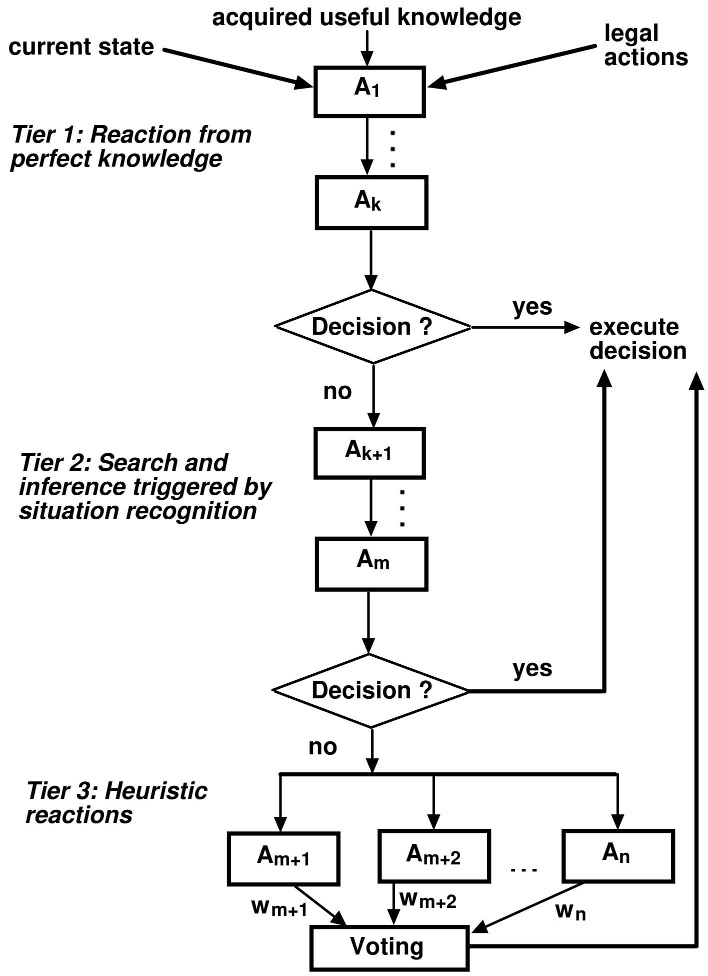

Figure 1: The FORR architecture organizes and manages Advisors.

<*strength, action, Advisor*> where *strength* is an integer in [0, 10]. A comment expresses an Advisor's support for (strength > 5), or discouragement of (strength < 5), a particular action. Comments may vary in their strength, but an Advisor may not comment more than once on any action in the current state.

Our work applies FORR to CSP. To apply FORR to a particular application domain, one codes definitions of problem classes and useful knowledge items, along with algorithms to learn the useful knowledge. In addition, one postulates Advisors, assigns them to tiers, and codes them as well. Effective application of the architecture requires a domain expert to provide such insights. The feasibility study of the next section was generated relatively quickly. The future work outlined in Section 4, however, is expected to require substantial changes to FORR.

3. Case Study

This case study on graph coloring is provided to introduce the basic approach we are pursuing, and to demonstrate its potential. We understand that there is a vast literature on graph coloring; we do not wish to give the erroneous impression that we believe that this study makes a serious contribution to it.

3.1 GC, the Graph Colorer

Graph Colorer (*GC*) is a FORR-based program for the specific CSP problem domain of graph coloring. We developed it as a proof of concept demonstration that the FORR architecture is a suitable basis for a research program aimed at learning collaborative algorithms attuned to classes of similar problems. GC includes only a few Advisors and learns only weights, but its results are quite promising. For GC, a problem class is the number of vertices and edges in a *c*-colorable graph, and a problem is an instance of such a graph. For example, a problem class might be specified as 4-colorable on 20 vertices with 10% *edge density.* ("Percentage edge density" here actually refers to percentage of possible edges above a minimal n-1, in this case 10% edge density means 19 + 17 = 36 edges.) A problem in that class would be a particular 4-colorable graph on 20 vertices with 36 edges. Problems are randomly generated, and are guaranteed to have at least one solution. There are, of course, a great many potential graphs in any given problem class.

GC basically simulates a standard CSP algorithm, *forward checking*. A world state for GC is a legally, partially (or fully) colored graph. On each iteration, GC either selects a vertex to color or, if a vertex has already been selected, colors it. Color selection is random. Our objective was to have GC learn an efficient way to select the vertices. In CSP terms, we wanted to acquire an efficient variable ordering heuristic. After a color is chosen for a vertex, that color is removed from the domain of neighboring vertices. If after a coloring iteration, some vertex is left without any legal colors, then the state is automatically transformed by retracting that coloring and removing it from the legal colors that vertex may subsequently assume. If necessary, vertices can be "uncolored" to simulate backtracking. Thus, given enough time and space, GC is *complete*, that is, is capable of finding a solution.

GC has two tier-1 Advisors. In *tier 1*, FORR maintains a presequenced list of prespecified, always correct Advisors, denoted by $A_1 \ldots A_k$ in Figure 1. A FORR-based artifact begins the decision making process there, with the current position, the legal actions from it, and any useful knowledge thus far acquired about the problem class. When a tier-1 Advisor comments positively on an action, no subsequent Advisors are consulted, and the action is executed. When a tier-1 Advisor comments negatively on an action, that action is eliminated from consideration, and no subsequent Advisor may support it. If the set of possible actions is thereby reduced to a single action, that action is executed.

GC's two tier-1 Advisors are Victory and Later. If only a single vertex remains uncolored and that vertex has been selected and has at least one legal coloring, *Victory* colors it. If an iteration is for vertex selection, *Later* opposes coloring any vertex whose degree is less than the number of colors that could legally be applied to it, on the theory that consideration of such a vertex can be delayed. Typically with FORR, the first tier does not identify an action, and control passes to *tier 2*, denoted by $A_{k+1} \ldots A_m$ in Figure 1. Tier-2 Advisors plan, and may recommend sequences of actions, instead of a single action. GC does not yet incorporate tier-2 Advisors.

If neither the first nor the second tier produces a decision, control passes to *tier 3*, denoted by $A_{m+1} \ldots A_n$ in Figure 1. In FORR, all tier-3 Advisors are heuristic and consulted in parallel. A decision is reached by combining their comments in a process called *voting*. When control resorts to tier 3, the action that receives the most support during voting is executed, with ties broken at random. Originally, voting was simply a tally of the comment strengths. Because that process makes tacit assumptions that are not always correct, voting can also be weighted.

GC has nine tier-3 Advisors, eight of which encapsulate a single primitive, naive approach to selecting a vertex. *Random Color* is the only coloring Advisor, so GC always selects a legal color for a selected vertex at random. Each of the remaining tier-3 Advisors simply tries to minimize or maximize a basic vertex property. *Min Degree* supports the selection of uncolored vertices in increasing degree order with comment strengths from 10 down. *Max Degree* is its dual, rating in decreasing degree order. *Min Domain* supports the selection of uncolored vertices in increasing order of the number of their current legal colors, again with strengths descending from 10. *Max Domain* is its dual. *Min Forward Degree* supports the selection of uncolored vertices in increasing order of their fewest uncolored neighbors, with strengths from 10 down. *Max Forward Degree* is its dual. *Min Backward Degree* supports the selection of uncolored vertices in increasing order of their fewest colored neighbors, with strengths from 10 down. *Max Backward Degree* is its dual. The use of such heuristic, rather than absolutely correct, rationales in decision making is supported by evidence that people *satisfice*, that is, they make decisions that are good enough [16]. Although satisficing solutions are not always optimal, they can achieve a high level of expertise. See, for example, [17].

Arguably these eight properties are simply the most obvious properties one could ascribe to vertices during the coloring process, making it all the more remarkable that the experiments we carried out were able to use them to such good effect. They also correspond naturally to properties of the "constraint graph" and "search tree" associated with general CSP's, providing additional resonance to the case study. Of course, a skeptical reader might be concerned that, consciously or not, we have "biased" our set of Advisors here. Even if that were so, we would respond that it is still up to FORR to learn how to use the Advisors appropriately, and that the ability to incorporate our expertise into the FORR architecture by specifying appropriate Advisors is a feature, not a bug.

Although a FORR-based program begins with a set of problem-class-independent, tier-3 Advisors, there is no reason to believe that they are all of equal significance or reliability in a particular problem class. Therefore, FORR uses a weight-learning algorithm called *PWL* (*Probabilistic Weight Learning*) to learn problem-class-specific weights for its tier-3 Advisors. The premise behind PWL

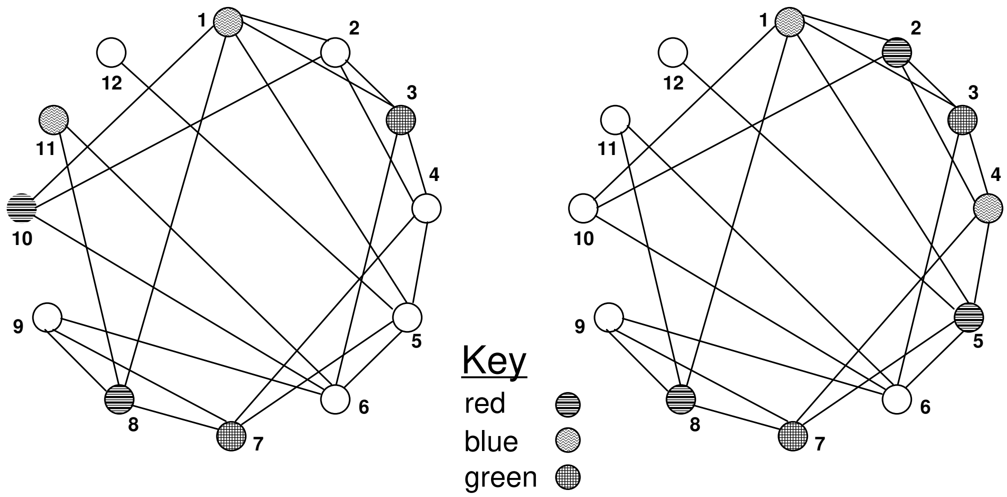

Figure 2: Two partially 3-colored graphs.

is that the past reliability of an Advisor is predictive of its future reliability.

Initially, every Advisor has a weight of .05 and a *discount factor* of .1. Each time an Advisor comments, its discount factor is increased by .1, until, after 10 sets of comments, the discount factor reaches 1.0, where it remains. Early in an Advisor's use, its weight is the product of its learned weight and its discount factor; after 10 sets of comments, its learned weight alone is referenced. In tier 3 with PWL, a FORR-based program chooses the action with the greatest support:

$$\underset{j}{\mathit{argmax}} \left\{ \sum_i \omega_i w_i s_{ij} \right\} \text{ where } \begin{cases} d = \text{number of opinions } i \text{ has generated} \\ \omega_i = \begin{cases} 0.1 * d \text{ if } d < 10 \\ 1 \text{ otherwise} \end{cases} \end{cases}$$

If an Advisor is correct, its wisdom will gradually be incorporated. If an Advisor is incorrect, its weight will diminish as its opinions are gradually introduced, so that it has little negative impact in a dynamic environment.

During testing, PWL drops Advisors whose weights are no better than random guessing. This threshold is provided by a non-voting tier-3 Advisor called *Anything.* Anything comments only for weight learning, that is, it never actually participates in a decision. Anything comments on one action 50% of the time, on two actions 25% of the time, and in general on n actions $(0.5)^n$% of the time. Each of Anything's comments has a randomly-generated strength in {0, 1, 2, 3, 4, 6, 7, 8, 9, 10}. An Advisor's weight must be at least .01 greater than Anything's weight to be consulted during testing. During testing, provisional status is also eliminated (i.e., ω_i is set to 1), to permit infrequently applicable but correct Advisors to comment at full strength. In summary, PWL fits a FORR-based program to correct decisions, learning to what extent each of its tier-3 Advisors reflects expertise. Because problem-class-specific Advisors can also be acquired during learning, PWL is essential to robust performance.

To get some sense of how GC behaves, consider the partially 3-colored graph in Figure 2(a). (The graph was used in a different context in [18].) Six of the vertices are colored, and the next vertex should now be selected for coloring. Since vertex 6 has no legal color, however, the most recently selected vertex will be uncolored. Now consider the partially colored graph in Figure 2(b). Since the number of possible colors for vertex 12 is 2, Later will eliminate vertex 12 as an immediate choice for coloring, and the remaining uncolored vertices will be considered by tier 3. For example, Min Degree would support the selection of vertex 11 with a strength of 10, and the selection of vertices 9 and 10 with a strength of 9. Similarly, Max Backward Degree would support the selection of vertices 9 and 10 with a strength of 10, and vertices 6 and 11 with a strength of 9. When the comments from all the Advisors are tallied, vertices 6 and 11 would receive maximum support, so without weights GC would choose one of them at random to color. If GC were using PWL, however, the strengths would be multiplied by the weights learned for the Advisors before tallying them.

3.2 Experimental design and results

Performance in an experiment was averaged over 10 runs. Each *run* consisted of a learning phase and a testing phase. In the *learning phase*, GC learned weights while it attempted to color each of 100 problems from the specified problem class. In the *testing phase*, weight-learning was turned off, and GC tried to color 10 additional graphs from the same class. Multiple runs were used because GC

Table 1: A comparison of GC's performance, averaged over 10 runs. Time is in seconds per solved problem; retractions is number of backtracking steps per solved or unsolved problem.

Vertices	Edges	Learning	Solutions	Time	Retractions
20	10%	no	95%	0.22	22.28
20	10%	yes	100%	0.11	0.00
20	20%	no	35%	1.11	418.16
20	20%	yes	83%	0.48	79.63
20	30%	no	12%	1.40	631.60
20	30%	yes	41%	1.43	427.05
50	10%	no	1%	3.23	815.82
50	10%	yes	46%	1.02	414.29
50	10%	transfer	32%	4.16	428.54
50	10%	bootstrap	40%	3.62	382.18
50	20%	no	0%	—	—
50	20%	yes	0%	—	—
50	20%	transfer	26%	5.09	486.61
50	20%	bootstrap	20%	4.51	519.89

learning can get stuck in a "blind alley", where there are no successes from which to learn. Thus a fair evaluation averages behavior over several runs. This is actually conservative, as we argue below that one could reasonably utilize the best result from multiple runs.

Problems were generated at random, for both learning and testing. Although there is no guarantee that any particular set of graphs was distinct, given the size of the problem classes the probability that a testing problem was also a training problem is extremely small. The fact that the training set varied from one run to another is, as we shall see, an advantage.

We ran experiments on five different problem classes: 4-colorable graphs on 20 vertices with edge densities of 10%, 20%, and 30%, and 4-colorable graphs on 50 vertices with edge densities of 10% and 20%. (Edge densities were kept relatively low so that enough 4-colorable graphs could be readily produced by our CSP problem generator. Those classes contain 36, 53, 70, 167, and 285 undirected edges, respectively.) To speed data collection, during both learning and testing, GC was permitted no more than 1000 task steps for the 20-vertex graphs, and 2000 task steps for the 50-vertex graphs. (A task step is either the selection of a vertex, the selection of a color, or the retraction of a color.)

We evaluated GC on the percentage of testing problems it was able to solve, and on the time it required to solve them. As a baseline, we also had GC attempt to color 100 graphs in each problem class without weight-learning. These results appear in Table 1 as "no learning." As Table 1 shows, weight learning ("yes" in Table 1) substantially improved GC's performance in all but the largest graphs. With weight learning, GC solved more problems and solved them faster. With weight learning, the program also did far less backtracking and required 32%-72% fewer steps per task.

An unanticipated difficulty was that, in the 50-vertex-20%-density class, GC was unable to solve any problem within the 2000-step limit, and therefore could not train its weights and improve. We therefore adapted the program so that it could learn in two other environments. With *transfer* learning, GC learned on small graphs but tested on larger graphs of the same density. With *bootstrap* learning, GC learned first on 50 small graphs of a given density, then learned on 50 larger graphs of the same density, and then tested on the larger graphs. Table 1 reports the result of both bootstrap learning and transfer learning between 20-vertex and 50-vertex-classes of the same density (e.g., from 20-vertex-20%-density to 50-vertex-20%-density).

3.3 Discussion

The most interesting results from our case study are reflected in the resultant weights. In the 20-vertex-10%-density experiment, on every run only the Advisors Max Degree, Min Domain, and Min Backward Degree had weights high enough to qualify them for use during testing. Inspection indicated that in the remaining experiments, runs were either *successful* (able to color correctly at least 5 of the 10 test graphs), or *unsuccessful* (able to color correctly no more than 2 test graphs). The 8 successful runs in the 20-vertex 20%-density experiment solved 95% of their test problems. In the 20-vertex-30%-density experiment, the 6 successful runs solved 65% of their test problems. On the 50-vertex-10%-density graphs, the 6 successful runs colored 76.7% of their test graphs. Inspection indicates that a run either starts well and goes

Table 2: Learned weights for those GC vertex-selection Advisors active during testing, averaged across successful runs in five different experiments. 50-vertex values are from bootstrap learning.

Advisor	**20-10%**	**20-20%**	**20-30%**	**50-10%**	**50-20%**
Max Degree	0.678	0.678	0.743	0.547	0.678
Min Domain	0.931	0.841	0.713	0.841	0.723
Min Backward Degree	0.943	—	—	—	—
Max Backward Degree	—	0.862	0.724	0.852	0.716

Table 3: A performance comparison of GC with the Brelaz heuristic. "GC best" is the top-performing runs with GC. Brelaz comment frequencies are provided for Min Domain (MD) and Max Forward Degree (MFD).

		Number of solutions			**Time in seconds**		**Comment frequency**	
Vertices	**Density**	**Brelaz**	**GC**	**GC best**	**Brelaz**	**GC**	**MD**	**MFD**
20	10%	86%	100%	100.0%	0.33	0.11	15.99	34.01
20	20%	26%	83%	95.0%	1.19	0.48	3.77	64.41
50	10%	0%	46%	76.7%	1.71	1.23	11.27	13.11

on to succeed, or goes off in a futile direction. Rather than wait for learning to recover, multiple runs are an effective alternative. As used here then, GC can be thought of as a restart algorithm: if one run does not result in an effective algorithm for the problem class, another is likely to do so.

For each problem class, the Advisors on which GC relied during testing in successful runs appear in Table 2. Together with their weights, these Advisors constitute an algorithm for vertex selection while coloring in the problem class. Observe that different classes succeed with different weights; most significantly the sparsest graphs prefer the opposite Backward Degree heuristic to that preferred by the others.

The differences among ordinary GC learning on 50-vertex-10%-density graphs, and transfer and bootstrap learning from them with 20-vertex-10%-density graphs, are statistically significant at the 95% confidence level: ordinary learning produces the best results, followed by bootstrap learning (where weights learned for the smaller graphs are tuned), followed by transfer learning (where weights for the smaller graphs are simply used). This further indicates that 20-vertex-10%-density graphs and 50-vertex-10%-density graphs lie in different classes, with regard to appropriate heuristics. Although solution of 50-vertex-20%-density graphs was only possible with with transfer or bootstrap learning, these are not our only recourses. We could also extend the number of steps permitted during a solution attempt substantially, on the theory that we can afford to devote extended training time to produce efficient "production" algorithms.

In this study, we attempted to "seed" GC with an "impartial" set of alternative vertex characteristics. Two factors previously considered individually by constraint researchers in a general CSP context as variable ordering heuristics, minimal domain size and maximal degree, were selected in all successful runs. Moreover the combination of the two is consistent with the evidence presented in [19] that maximizing domain-size/degree is a superior CSP ordering heuristic to either minimizing domain size or maximizing degree alone. Given the relatively recent vintage of this insight, its "rediscovery" by FORR is impressive. Min Backward Degree corresponds to the "minimal width" CSP variable ordering heuristic, and again FORR was arguably insightful in weighting this so heavily for the 20-10 case, since it can guarantee a backtrack-free search for tree-structured problems [20]. The success of Max Backward Degree for the other classes may well reflect its correlation with both Min Domain (the domain will be reduced for each differently colored neighbor) and Max Degree.

In a final experiment we implemented the classic Brelaz heuristic for graph coloring within FORR by simply eliminating any vertex that does not have minimum domain in tier 1 and then voting for vertices with maximum forward degree in tier 3. Table 3 shows the results. Note that GC, learning from experience, does considerably better.

4. Future Work

GC is a feasibility study for our planned Adaptive Constraint Engine (ACE). ACE will support the automated construction of problem-class-specific constraint solvers in a number of different problem domains. Automating constraint solving as a learned collaboration among heuristics presents a number of specific opportunities and challenges. FORR offers a concrete approach to these opportunities and challenges; in turn, ACE provides new opportunities and challenges to extend the FORR architecture.

4.1 Opportunities

We anticipate a range of opportunities along four dimensions:

- **Algorithms:** Algorithmic devices known to the CSP community can be specified as individual Advisors for ACE. Advisors can represent varying degrees of local search or different search methods (e.g., backjumping), and they can represent heuristic devices for variable ordering or color selection. ACE could be modified to employ other search paradigms, including stochastic search.
- **Domains:** ACE will facilitate the addition of domain-specific expertise at varying degrees of generality, and in various fields. For example, we might discover variable ordering heuristics for a class of graphs or general graphs, for employee scheduling problems or general scheduling problems.
- **Change:** We will begin by learning good algorithms for a static problem or problem class, that is, good weights for a set of prespecified Advisors. In practice, however, problems change. For example, a product configuration problem changes when a new product model is introduced. ACE will offer opportunities to adapt to such change. Furthermore, ACE should be able to adapt to changing conditions during a single problem-solving episode. (The FORR architecture has proved resilient in other dynamic domains.)
- **Discovery:** We can select among standard techniques, for example, minimal domain variable ordering. We can combine these techniques, through weighting and voting schemes. Most exciting, we can learn new techniques, in the form of useful knowledge and new Advisors. These will include planners at the tier-2 level. Some preliminary work on learning new Advisors based on relative values of graph properties (Later is an example of such an Advisor, albeit prespecified here) has shown both improved solution rates and considerable speedup.

4.2 Challenges

Exploring these opportunities will require progress in several areas. Basically, we need to provide the elements of a collaborative learning environment. FORR permits us to begin addressing this challenge quickly and concretely.

- **Advice:** Many interesting issues arise in appropriately combining advice. With variable ordering heuristics, for example, we can now move beyond using secondary heuristics to break ties, or combining heuristics in crude mathematical combinations. Ordering advice can be considered in a more flexible and subtle manner. The challenge lies in using this new power intelligently and appropriately. In particular, this may require new voting schemes, such as partitioning FORR's tier 3 into prioritized subsets. Such higher order control could be learned.
- **Reinforcement:** Opportunities to learn can come from experience or from expert advice. ACE will provide a mechanism to generalize experience computed from exhaustive analysis or random testing. It will also provide a mechanism for knowledge acquisition from constraint programming experts and domain experts. In particular, we expect that ACE will be able to extract from domain expert decisions knowledge that the experts could not impart directly in a realizable form, addressing the knowledge acquisition problem for constraint programming. Specific reinforcement schemes, analysis, and experimental protocols are required to accomplish this. For example, what is the proper definition of an "optimal" variable ordering choice, and what forms of experiment or experience will come closest to modeling optimality?
- **Modeling:** We need languages for expressing general constraint solving knowledge and domain specific expertise. Such languages will support discovery of useful knowledge and new Advisors. They will enable us to learn the context in which tools are to be brought to bear. For example, a grammar has been formulated for a language that compares relative values (e.g., <, =) of vertex properties (e.g., degree, number of colored neighbors); this grammar can be used to formulate learned Advisors.

Modeling constraint solving and domain knowledge to facilitate discovery presents perhaps the most exciting combination of opportunity and challenge. The feasibility study already gives us a glimpse of this capability. The features we used, involving domain size and degree, are basic features of a constraint graph model of a problem, and of the coloring domain in particular. We simply described possible variations on those features, and let GC "discover" which ones most effectively contributed to control of variable ordering during search.

More broadly, we envision modeling constraint satisfaction search as movement through a space of sets of potential solutions (the power set of the Cartesian product of the variable domains). Conventional algorithms may be viewed as special cases of movement through that space. Backtrack search operates by fixing one value at a time and moving to the Cartesian product of the remaining (pruned) domains. Hill climbing operates by moving from one singleton set to another. ACE can explore the vast realm of intermediate algorithms. We envision expanding FORR to operate with sets of possible states, an extension of the architecture that would facilitate exploration of algorithms modeled in this manner.

In FORR, not all Advisors are prespecified. Given a language from which to develop them, and a learning method, a FORR-based program can acquire and integrate new, problem-class-specific Advisors into tier 3. For example, *Hoyle*, the FORR-based game player, has learned two different kinds of game-specific Advisors from perceptual data [21]. The Advisors Hoyle learns provide insight into the nature of a game, extend its representational ability, and substantially improve the program's performance. The mechanism for learning new Advisors is sketched in [22] and detailed in [21].

This work will motivate numerous enhancements to FORR. For example, we hope to learn to sequence the tier-1 Advisors, rather than prespecify their order. We will also work on collaborative planning in tier 2. We intend to add some generic, resource-bounded versions of forward search. We will partition tier 3 based upon learned weights, and then prioritize the allocation of resources accordingly. Finally, we expect to explore weight-learning algorithms that are more domain-specific. In that context, we expect to consider learning about the application of pairs of Advisors, as in WINNOW [23].

5. Conclusion

The combination of constraint programming and Advisor-based collaborative learning, is an innovative approach toward making constraint software more effective and more widely available. Our Adaptive Constraint Engine will provide a comprehensive architecture for acquiring and controlling collaborative and adaptive constraint solving methods.

The FORR architecture supports this frontier CSP research by transforming amorphous objectives ("reinforce success") into concrete ones ("reward Advisors"). The CSP research will in turn motivate major extensions of FORR facilities. A case study has demonstrated the potential of our project; it constructed different algorithms for different classes of graphs, "rediscovered" some constraint solving insights, and outperformed the Brelaz heuristic.

Acknowledgements

This work was supported in part by NSF grant IIS-9907385 and by NASA. We thank Richard Wallace for his assistance in generating test problems.

References

1. Borrett, J., E.P.K. Tsang, and N.R. Walsh. Adaptive constraint satisfaction: the quickest first principle. in *12th European Conference on AI*. 1996. Budapest, Hungary.

2. Caseau, Y., F. Laburthe, and G. Silverstein, A Meta-Heuristic Factory for Vehicle Routing Problems, in *Principles and Practice of Constraint Programming - CP'99*. 1999, Springer: Berlin.

3. Minton, S., Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, 1996. 1(1).

4. Smith, D.R., KIDS: A Knowledge-based Software Development System, in *Automating Software Design*, M.R. Lowry and R.D. McCartney, Editors. 1991, AAAI Press.

5. Saraswat, V.J. and P. Van Hentenryck, Constraint Programming. *ACM Computing Surveys*, Special Issue on Strategic Directions in Computing Research, 1996. 28(4).

6. Freuder, E. and M. Wallace, eds. Special Issue on Constraints. *IEEE Intelligent Systems*. 2000: 15:1.

7. Freuder, E. and A. Mackworth, eds. *Constraint-Based Reasoning*. 1992, MIT Press: Cambridge, MA.

8. Tsang, E.P.K., *Foundations of Constraint Satisfaction*. 1993, London: Academic Press.

9. Chatterjee, S. and S. Chatterjee, On Combining Expert Opinions. *American Journal of Mathematical and Management Sciences*, 1987. 7(3-4): p. 271-295.

10. Jacobs, R.A., Methods for Combining Experts' Probability Assessments. *Neural Computation*, 1995. 7: p. 867-888.

11. Biswas, G., S. Goldman, D. Fisher, B. Bhuva, and G. Glewwe, Assessing Design Activity in Complex CMOS Circuit Design, in *Cognitively Diagnostic Assessment*, P. Nichols, S. Chipman, and R. Brennan, Editors. 1995, Lawrence Erlbaum: Hillsdale, NJ.

12. Crowley, K. and R.S. Siegler, Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science*, 1993. 17(4): p. 531-561.

13. Ratterman, M.J. and S.L. Epstein. Skilled like a Person: A Comparison of Human and Computer Game Playing, in *Seventeenth Annual Conference of the Cognitive Science Society*. 1995. Pittsburgh: Lawrence Erlbaum Associates.

14. Epstein, S.L. On Heuristic Reasoning, Reactivity, and Search, in *Fourteenth International Joint Conference on Artificial Intelligence*. 1995. Montreal: Morgan Kaufmann.

Epstein, S. L. (1992). Prior Knowledge Strengthens Learning to Control Search in Weak Theory Domains. *International Journal of Intelligent Systems*, *7*, 547-586.

16. Simon, H.A., *The Sciences of the Artificial*. second ed. 1981, Cambridge, MA: MIT Press.

17. Keim, G.A., et al. PROVERB: The Probabilistic Cruciverbalist, in *Sixteenth National Conference on Artificial Intelligence.* 1999. Orlando: AAAI Press.

18. Smith, B.M., The Brélaz Heuristic and Optimal Static Orderings, in *Principles and Practice of Constraint Programming - CP'99*. 1999, Springer: Berlin. p. 405-418.

19. Bessiere, C. and J.-C. Regin, MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems, in *Principles and Practice of Constraint Programming - CP96*, LNCS 1118, E.C. Freuder, Editor.

1996, Springer-Verlag. p. 61-75.

20. Freuder, E.C., A sufficient condition for backtrack-free search. *Journal of the ACM*, 1982. 29(1): p. 24-32.

21. Epstein, S.L., Perceptually-Supported Learning. Submitted for publication.

22. Epstein, S.L., J. Gelfand, and E.T. Lock, Learning Game-Specific Spatially-Oriented Heuristics. *Constraints*, 1998. 3(2-3): p. 239-253.

23. Littlestone, N. and M.K. Warmuth, The Weighted Majority Algorithm. *Information and Computation*, 1994. 108(2): p. 212-261.

24. Nadel, B., Consistent labeling problems and their algorithms: expected complexities and theory-based heuristics. *Artificial Intelligence*, 1983. 21: p. 135-178.

25. Gent, I., E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem, in *CP-96*. 1996.