Appears in the Proceedings of the Tenth International Conference on Machine Learning, Amherst, MA, June 1993, pp. 135–142.

Learning Search Control Knowledge for Deep Space Network Scheduling

Jonathan Gratch*, Steve Chien+, and Gerald DeJong*

*Beckman Institute
University of Illinois
405 N. Mathews Av., Urbana, IL 61801
{gratch, dejong}@cs.uiuc.edu

+Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109-8099
chien@aig.jpl.nasa.gov

Abstract

While the general class of most scheduling problems is NP–hard in worst–case complexity, in practice, for specific distributions of problems and constraints, domain–specific solutions have been shown to perform in much better than exponential time. Unfortunately, constructing such techniques is a knowledge–intensive and time–consuming process that requires a deep understanding of the domain and the scheduler. The goal of our work is to develop techniques to allow for automated learning of an effective domain–specific search strategy given a general problem solver with a flexible control architecture. In this approach, a learning system searches a space of possible control strategies, using statistics to evaluate performance over the expected problem distribution. We discuss an application of the approach to scheduling satellite communications. Using problem distributions based on actual mission requirements, our approach identified strategies that both decrease the amount of CPU time required to produce schedules, and increase the percentage of problems that are solvable within computational resource limitations.

1 INTRODUCTION

General problem solving tasks like planning and scheduling are inherently complex. Nevertheless, in many practical situations these complex problems have reasonable solutions (e.g. traveling salesman problem [Held70]). Often we can take advantage of the structure of a domain or the distribution of problems to formulate effective solutions to complex problems. Unfortunately, a system designer must devote considerable expense to the performance aspects of an algorithm.

In this article we investigate the use of a machine learning approach to automatically improve a performance element with respect to a specific domain and distribution of problems. The performance element must be flexible, meaning there are control decisions that effect search that may be resolved in alternative ways. The overall learning problem we are addressing can be specified as follows. Given a flexible performance element *PE* with control points $CP_1 ... CP_n$, where each control point CP_i corresponds to a particular control decision and for which there is a set of alternative decision methods $M_{i,1} ... M_{i,k}$,[1] a *control strategy* is a selection of a specific method for every control point (e.g., $STRAT = \langle M_{1,3}, M_{2,6}, M_{3,1}, ... \rangle$). A control strategy determines the overall behavior of the scheduler. It may effect properties like computational efficiency or the quality of its solutions. Let $PE(STRAT)$ be the problem solver operating under a particular control strategy. The function $U(PE(STRAT), d)$ is a real valued *utility function* that is a measure of the goodness of the behavior of the scheduler over problem d. The goal of learning can be expressed as: given a problem distribution D, find $STRAT$ so as to maximize the *expected utility* of PE. Expected utility is defined formally as:

$$\sum_{d \in D} U(PE(STRAT), d) \times probability(d)$$

For example, in a planning system such as PRODIGY [Minton88], when planning to achieve a goal, control points would be: how to select an operator to use to achieve the goal; how to select variable bindings to instantiate the operator; etc. A method for the operator choice control point might be a set of control rules to determine which operators to use to achieve various goals plus a default operator choice method. A strategy would be a set of control rules and default methods for every control point (e.g., one for operator choice, one for binding choice, etc.). Utility might be defined as a function of the time to construct a plan.

Our proposed solution to this learning problem, sometimes called the *utility problem* [Minton88], is embodied in the COMPOSER system. COMPOSER can be characterized as a hill-climbing search in the space of possible strategies. The learning system alternately conjectures changes to the current control strategy and statistically evaluates them to determine how well they enhance expected utility.

1. Note that a method may consist of smaller elements so that a method may be a set of control rules or a combination of heuristics.

In this paper we describe an application of the COMPOSER framework to learning search strategies for an automated scheduling technique in a NASA domain of space-craft communication scheduling. Two important aspects of this evaluation are that the task and problem distribution are based on a real-world situation, and the scheduling approach was developed independently of our learning work [Bell92]. The implementation includes a novel approach for improving learning efficiency. The performance of the system, along with previous results in artificial planning domains [Gratch91, Gratch92], demonstrates COMPOSER's flexibility and its potential to identify beneficial knowledge in practical learning problems.

2 COMPOSER

COMPOSER is a statistical approach to improving the expected utility of problem solving. The overall approach is one of generate and test hill-climbing. Given an initial problem solver, a *transformation generator* constructs a set of possible transformations to the control strategy. Each of these changes is evaluated statistically over the expected distribution of problems. A transformation is adopted if it increases the expected utility of solving problems over that distribution. The generator then constructs a set of transformations to this new strategy and so on. For a complete description of the method see [Gratch92]. The algorithm is summarized in the Appendix.

COMPOSER's solution is applicable in cases where the following conditions apply.

1. The control strategy space can be structured to facilitate hill-climbing search. In general, the space of such strategies is so large as to make exhaustive search intractable. COMPOSER requires a transformation generator that structures this space into a sequence search steps, with relatively few transformations at each step. In Section 3.4 we discuss some techniques for incorporating domain specific information into the structuring of the control strategy space.

2. There is a large supply of representative training problems so that an adequate sampling of problems can be used to estimate expected utility for various control strategies.

3. Problems can be solved with a sufficiently low cost in resources so that estimating expected utility is feasible.

4. There is sufficient regularity in the domain such that the cost of learning a good strategy can be amortized over the gains in solving many problems.

COMPOSER can be seen as one of a class of statistical approaches to improving the expected utility of problem solving (see also [Greiner92, Laird92, Subramanian92]. The principle drawback of these techniques is that they find only local maxima, they may require many examples, and examples can be expensive to process. Furthermore, their statistical properties rest upon assumptions that may not hold in practice. Greiner and Jurisica demonstrate that, under very weak assumptions, the number of examples, or *sample complexity,* can be bounded by a polynomial function of the allowable statistical error [Greiner92]. Weak assumptions are a way of ensuring wide applicability. Unfortunately, the resulting bounds prove too large for most practical applications (see [Buntine89, Gratch92]). COMPOSER embodies stronger statistical assumptions suggested by the Central Limit Theorem [Hogg78 pp. 192–195]. These stronger assumptions drastically reduce the number of examples required to make statistical decisions, but they limit the applicability of the approach. In previous evaluations these stronger assumptions have proved reasonable: COMPOSER's assumptions produced sample complexities two to three order of magnitudes less than the weak assumptions adopted by Greiner and Jurisica, *without* compromising the statistical error [Gratch92]. However these promising results have the drawback that they are based on artificial domains and problem distributions. The ultimate usefulness of our approach depends on its behavior on real-world domains, which this paper addresses.

3 THE DEEP SPACE NETWORK

We applied the COMPOSER approach to improving the performance of a scheduler operating in the domain of spacecraft communication scheduling. The resulting system is called DSN-COMPOSER. The scheduling problem is a complicated real world task that has proved challenging to state-of-the-art scheduling techniques. The problem is to allocate communication requests between earth-orbiting satellites and the three 26-meter antennas at Goldstone, Canberra, and Madrid. These antennas make up part of the Deep Space Network (DSN) that is responsible for communication with earth-orbiting and inter-planetary spacecraft. Each satellite has a set of constraints, called project requirements, that define its communication needs. For example, the Nimbus-7 satellite must have at least four 15-minute communication slots per day, and these slots cannot be greater than five hours apart. Two factors complicate the problem. First, antennas are a limited resource – two satellites cannot communicate with the same antenna at the same time. Second, satellites can only communicate with certain antennas at certain times, depending on their orbits.

Scheduling is done on a weekly basis. A weekly scheduling problem is defined by three elements: (1) the set of satellites to be scheduled, (2) the constraints associated with each satellite, and (3) a set of time periods specifying all temporal intervals when a satellite can legally communicate with an antenna. Two time periods conflict if they use the same antenna and overlap in temporal extent. A valid schedule specifies a non-conflicting subset of all possible time periods where each project's requirements are satisfied.

3.1 THE SCHEDULER SYSTEM

LR-26 is a heuristic approach to the scheduling problem [Bell92] developed at the Jet Propulsion Laboratory. It provides a good platform for learning as it can be modified easi-

ly to incorporate alternative heuristic strategies. Furthermore, it uses an expert crafted control strategy that provide a challenging base-line to judge learned knowledge. There is also significant motivation to improve the effectiveness of the default control strategy. If LR-26 is chosen to replace the human scheduler, it will serve as one module in a larger interactive system. This system must make many repeated calls to the scheduler to compare variants of each weekly schedule. For this reason, LR-26 must provided solutions in a timely fashion.

Scheduling is formulated as a 0-1 integer programming problem [Taha82]. This is a methodology for finding an assignment to integer variables that maximizes the value of an *objective function*, subject to a set of linear constraints. The objective function characterizes the "value" of the solution. Many constraint satisfaction problems (CSP) are easily cast as integer programming problems [Mackworth92]. In the DSN domain, time periods are treated as 0-1 integer variables (0 if the time period is excluded from the schedule or 1 if it is included), the objective is to maximize the number of time periods in the schedule subject to the project requirements and temporal conflict constraints which are expressed as sets of linear inequalities. Integer programming is NP-hard, and the size of our scheduling problems makes the conventional approach impractical: a typical problem has approximately 650 variables and 1300 constraints. LR-26 embodies a heuristic approach called *lagrangian relaxation* [Fisher81]. Lagrangian relaxation requires identifying a set of constraints that, if removed, make the problem computationally easy. These constraints are "relaxed," meaning they no longer act as constraints but instead modify the objective function. A relaxed objective function is automatically generated such that satisfying relaxed constraints increases the value of the relaxed solution. The relaxed problem is by definition easy to solve and often finding the highest value relaxed solution solves the original problem. Furthermore, each relaxed constraint has a weight associated with it when it is added to the objective function. By systematically adjusting these weights and re-solving the relaxed problem, a solution to the unrelaxed problem is often efficiently discovered. Even if the unrelaxed problem cannot be solved in this manner, this weight adjustment cycle can move the scheduler closer to a solution, allowing the unrelaxed solution to be discovered with less search. LR-26 relaxes inter-antenna constraints. This representation facilitates an efficient implicit representation of temporal conflict constraints, which make up more than half of all constraints in a typical problem.

LR-26 combines lagrangian relaxation with standard constraint satisfaction search techniques. The scheduler performs depth-first search through a space of partial schedules. A variable is assigned a value of *in* if the associated time period is included in the partial schedule, *out* if it is excluded from the partial schedule. The scheduler constructs a complete schedule by incrementally extending the partial schedule. First it attempts to completely extend the schedule using the lagrangian relaxation method. If the relaxed solution satisfies all constraints it is returned. Otherwise, a set of possible extensions to the partial schedule is created and these are recursive explored. Extensions are created by choosing an unsatisfied constraint, identifying a set of uncommitted variables in the constraint, and assigning possible values to these variables. The set of extensions are placed on a stack to implement the depth-first search. The search continues until a solution is uncovered or a time-bound is reached. Currently the scheduler implements a time-bound of five CPU minutes. Any problem not solved within this bound is deemed unsolvable.

LR-26 can be viewed as a recursive application of four control decisions: (1) decide on a lagrangian weight adjustment scheme, terminating if a viable solution is found; otherwise (2) choose an unsatisfied constraint, (3) determine a set of extensions to the partial schedule that satisfy the constraint, and (4) determine an order to explore these extensions. The scheduler embodies a heuristic method for each of these control operations. These operations provide natural points at which to insert alternative learned methods.

3.2 PROBLEM DISTRIBUTION

We constructed a distribution of scheduling problems using the requirements and time periods of satellites using the deep space network. Ideally, we would use the identical problem distribution faced by the human experts in this domain. Unfortunately, not all of this information is in electronic form and thus is difficult to present to the LR-26 scheduler. There does, however, exist a large electronic database of information for many of the projects in the deep space network. We used this database to construct a large body of scheduling problems that are representative of, if not identical to, the type of problems faced by the human schedulers. Problems are generated by randomly choosing combinations of projects from the available data. The requirements and time periods represent the requirements and time periods of actual projects. The primary difference between these and actual problems lies in the particular combinations of projects that appear in the schedule.

We performed some initial evaluations of the LR-26 scheduler on these generated problems. We observed that some problems could not be solved by the scheduler using any of several search control strategies even with large resource bounds. This is consistent with the observation that the scheduling problem is inherently NP-Hard – there will be some problems that cannot be efficiently solved, even with good heuristics. These problems tend to dilute any performance improvement that we might gain through learning. For our experiments we eliminated this complicating factor by constructing our problem distribution without these unsolved problems. These problems were identified by solving each randomly generated problem multiple times using about twenty different search strategies (the strategies were identified during our pilot investigations). If a problem

could not be solved by any of the strategies within the time bound, it was not added to the experimental distribution. For comparative purposes we include a secondary set of experiments that incorporate these unsolved problems. For a complete description of how training examples are generated, see [Gratch93].

3.3 EXPECTED UTILITY

In the DSN application a chief concern is with the computational efficiency of the scheduler. There is a strong need that the scheduler return quickly on average. This behavioral preference can be expressed by a utility function related to the computational effort required to solve a problem. As the effort to solve a problem increases, the utility of the problem solver on that problem should decrease. In this paper we characterize this preference by making utility the negative of the CPU time required by the scheduler on a problem.

3.4 HEURISTICS FOR LR-26

LR-26 combines lagrangian relaxation and constraint satisfaction search techniques to increase scheduling efficiency. Nevertheless, scheduling is still quite expensive. While the problems are of sufficient complexity that some search is unavoidable, alternate search control methods can drastically impact the amount of search required, especially as there is substantial repetition in this domain. Many projects use the antennas for many years, and their project requirements vary little across weeks. Because possible slots for specific antennas communicating with spacecraft are dictated by spacecraft orbits, the space of potential communication slots also contains significant regularity. This suggests that heuristics can be crafted to exploit this regularity to improve performance.

Many heuristics have been suggested to improve scheduling efficiency. Often these heuristics are stated as general principles (e.g. “first instantiate variables that maximally constraint the rest of the search space” [Dechter92]) and there may be many ways to realize them in a particular scheduler and domain. Furthermore, there are almost certainly interactions between methods used at different control points that makes it difficult to construct a good overall strategy. These factors conspire to make manual developing and evaluation of heuristics a tedious, time consuming task that requires significant knowledge about the domain and scheduler. As a result, only a limited set of alternate heuristics were considered in LR–26’s development. We formalized a much larger set for automatic consideration.

As described previously, there are four basic control points in LR–26: a weight adjustment method, constraint selection method, a method for constructing alternative solutions for the constraint, and method for ordering these alternatives. We allow for both a primary and secondary sort function to order candidate constraints, so there are effectively five control points. A control strategy consists of a particular heuristic method for each of the five control points. For LR–26 there are 4 alternative methods for the lagrangian weight adjustment control point, 9 alternative methods for each constraint selection sort (9 for the primary sort and 9 for the secondary sort – excluding the primary sort but including the possibility of no secondary sort), 2 alternative methods for constructing alternative solutions to a constraint, and 4 alternative methods for ordering alternative solutions to a constraint. Exhaustively searching the space of possible control strategies is in general computationally infeasible. As there are roughly 4×9×9×2×4=2592 possible strategies in the LR-26 control strategy search space, to exhaustively search the control strategy space, taking a significant number of examples per strategy (fifty) at a cost of 5 CPU minutes per problem would require approximately 450 CPU days.

Some example methods are as follows. For the weight adjustment, methods perform adjustments at every search node, perform only at root node, never perform adjustments. Methods for selecting constraints involve features of the constraints and constraint graph. Prefer the shortest constraint (e.g. the one that mentions the fewest unbound variables) and prefer the constraint that mentions the variables that temporally conflicts with the most other variables are examples of constraint selection methods (recall variables correspond to time-periods). For constructing alternative solutions, there were 2 methods: force the first variable in the constraint into the schedule for one child and out for the second child; and construct a child for each variable in the constraint, forcing that variable into the schedule. Examples of alternative ordering methods include preferring high conflict variables and preferring low conflict variables[2]. These heuristics can be viewed as variable ordering heuristics from the CSP literature [Dechter92].

COMPOSER requires this search to be structured for hill-climbing search. A simple way to organize the search through this strategy space would be to treat all control points a equal and consider all single method changes to a given control strategy. This was the strategy in our PRODIGY implementation of COMPOSER [Gratch92]. Here, we used our knowledge of the scheduler to take advantage of interactions (or lack thereof) between control points to help structure the search. The intent of this organization is to reduce the branching factor in the control strategy search and improve the expected utility of locally optimal solutions. This approach led to a transformation generator that implements a layered search through the strategy space. Each control point is assigned to a level. The control strategy space is search by evaluating all combinations of methods at a single level, adopting the best combinations, and then

2. This is a prime example of the difficulty of determining good heuristics. Selecting high conflict variable has the benefit of rapidly forcing many time-periods to be in or out of the schedule (e.g. reducing the number of steps to solution). Selecting low conflict intervals advocates adding those time-periods which cost little, and thus may satisfy constraints without causing conflicts [Dechter92].

moving onto the next level. The organization is shown below:

Level 0: weight adjustment
Level 1: constructing alternatives
Level 2: secondary constraint sort, child sort
Level 3: primary constraint sort

The weight adjustment and alternative construction control points were separated because they are relatively independent from the other control points. While there is clearly some interaction between weight search, alternative construction, and the other control points, a good selection of methods for pricing and alternative construction should perform well across all constraint and child sorts. The primary constraint sort was separated into another level because it was the sole control point that the implementor of LR–26 had spent time experimenting and optimizing. Thus, we believed that it was unlikely the default strategy could be improved upon, and hence relegated to a separate level.

Given this transformation generator, DSN-COMPOSER hill-climbs across levels. It first entertains weight adjustment methods, then alternative construction methods, then combinations of secondary constraint sort and child sort methods, and finally primary constraint sort methods. Each choice is made given the previously adopted methods. Searching the structured space involves evaluating at most 4+2+9×4+9=51 strategies.

This leveled search can be viewed as the consequence of asserting certain types of relations between control points. Independence relations indicate cases in which the utility of methods for one control point is roughly independent of the methods used at other control points. Dominance relations indicate that the changes in utility from changing methods for one control point are much larger than the changes in utility for another control point. Finally, inconsistency relations indicate when a method M_1 for control point X is inconsistent with method M_2 for control point Y. This means that any strategy using these methods for these control points need not be considered.

3.5 EXTRACTING UTILITY INFORMATION

To perform its evaluations, COMPOSER must be able to determine, given a current control strategy, a transformation, and a problem, what improvement the transformations provides over the current strategy on that problem. How we can extract this information depends intimately on the utility function, the form of the transformations, and the extent to which we can model the behavior of the problem solver. In the best case we possess a detailed cost model of the problem solver that efficiently derives the ramification of proposed modifications without actually solving the problem (e.g. [Greiner89, Subramanian90]). In the worst case we can resort to brute-force simulation: solve the problem with and without the proposed modification and observe the difference in utility between the two solution attempts. In the former case the cost of processing an example is tied to the efficiency of manipulating the model. In the later case the cost is tied to the efficiency of the problem solver and grows linearly with the number of transformations we consider.

In the current context we found it necessary to use the later, more costly, alternative. Given m candidate transformations, DSN-COMPOSER solves each problem m+1 times; once with the current control strategy and once using each of the m transformations. This allows us to generate the m incremental utility values. There are several issues that lead us to this particular solution. In particular we found that other more efficient proposals for gathering statistics (see [Gratch92, Greiner92]) were not appropriate to this problem. We elaborate on this issue in Section 5.

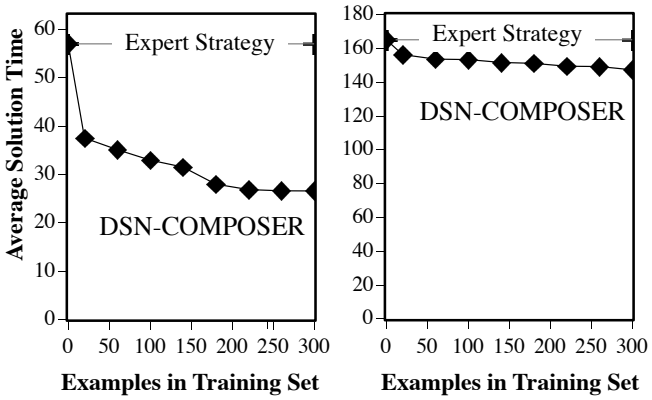
4 EXPERIMENT AND RESULTS

DSN-COMPOSER should, with high probability, improve the expected utility of the scheduler over the distribution of problems. This can be seen as two basic claims that can be tested empirically. First, DSN-COMPOSER should identify transformations that improve the expert strategy. Second, it should identify these transformations with the confidence predicted by the statistical theory. Besides testing these claims, we are also interested in two secondary questions. How quickly does the technique improves expected utility (e.g., how many examples are required to make statistical inferences?). Also, many problems are unsolvable within five minutes with the expert strategy. Can DSN-COMPOSER improve the number of solvable problems?

When DSN-COMPOSER learns a strategy, its behavior is guided by a random selection of training examples according to the problem distribution. As a result of this random factor, the system will exhibit different behavior on different runs of the system. On some runs the system may learn high utility strategies. On other runs the random examples may poorly represent the distribution and the system may adopt transformations with negative utility. The typical behavior can be estimated from several runs of the system.

For the experiments a learning run consists of 300 randomly selected training examples. The expected utility of all learned strategies is assessed on an independent test set of 1000 test examples. A measurement of learning rate is determined by recording the strategy learned by DSN-COMPOSER after every 20 examples. Thus we can see the result of learning with only twenty examples, only forty examples, etc. To assess statistical error, we perform twenty runs of the system on twenty distinct training sets.

COMPOSER has two parameters. The parameter δ specifies the acceptable level of statistical error. This is the chance that the technique will adopt a bad transformation or reject a good one. In DSN-COMPOSER this is set to a standard value of 5%. COMPOSER bases each statistical inferences on at least n_0 examples. In DSN-COMPOSER n_0 is set to the empirically determined value of fifteen.



Summary of Results			Dist. 1	Dist. 2
Avg. Solution Time seconds per prob.	Expert Strategy		57	165
	Learned Strategies	avg. across trials	27	147
		best strategy	12	140
		worst strategy	36	160
Statistical Error Rate		predicted	5%	5%
		observed	3%	6%
Solution Rate % of solvable probs.	Expert Strategy		93%	51%
	Learned Strategies	avg. across trials	95%	54%
		best strategy	97%	59%
		worst strategy	92%	50%

Figure 1. Learning curves showing performance as a function of the number of training examples and table of experimental results. Results are provided for original distribution (Distribution 1) and the distribution including unsolved problems (Distribution 2).

Recall that we purposely excluded several inherently difficult scheduling problems from the problem distribution. These problems, if added to the problem distribution, should make learning more difficult as no strategy is likely to provide a noticeable improvement within the five minute resource bound. Nevertheless, it is important to show that DSN-COMPOSER does not require the elimination of these problems. To test this we created a new distribution by incorporating these problems into the original distribution and repeated the experiments. Results for both sets of experiments are shown in Figure 1. The original distribution is called Distribution 1 while the second is referred to as Distribution 2.

The results support the two primary claims. For Distribution 1, the system learned search control strategies that yielded a significant improvement in performance. It reduced the average time to solve a problem from 57 to 27 seconds (a 53% improvement). There is modest variance in the expected utility of the strategies learned in the twenty trials. The best of these strategies required only 12 seconds on average to solve a problem (an improvement of 78%). The observed statistical accuracy remained well within the theoretically predicted bound: of 93 transformations adopted across the twenty trials, only 3% decreased expected utility. It took an average of 58 examples to adopt each transformation. The human expert strategy was unable to solve 7% of the scheduling problems within the resource bound. One strategy learned by the system reduced this number to 3%.

For Distribution 2, learned strategies reduced the average solution time from 165 to 147 seconds (an 11% improvement). The best learned strategies required 160 seconds on average to solve a problem (an improvement of 15%). The observed statistical accuracy did not significantly differ from the theoretically predicted bound: of 107 transformations were adopted across the trials, only 6% decreased expected utility. The introduction of the difficult problems resulted in higher variance in the distribution of incremental utility values and this is reflected in a higher sample complexity: an average of 108 examples to adopt each transformation. Some improvement was noted on the supposedly unsolvable problems. One strategy learned by DSN-COMPOSER increased the number of solvable problems from 51% to 59% (a 16% improvement).

Most of the learned improvement came from improved methods for lagrangian weight adjustment, secondary constraint sorting, and child ordering. The best learned strategy performed no weight adjustment, preferred constraints that contained time periods with high levels of temporal conflicts, and, among child nodes, prefer alternatives with few temporal conflicts. By avoiding the weight adjustment, DSN-COMPOSER implies that there is no utility in using lagrangian weights to improve the first relaxed solution. One interpretation of the constraint and child methods is that search should proceed by first identifying a highly constrained constraint, and then choose the least constrained way of satisfying it. It is interesting that this is consistent with the general recommendations in the CSP literature [Dechter92].

5 DISCUSSION

This paper evaluates COMPOSER on a real-world domain. COMPOSER is grounded in a mathematical framework. While the framework embodies statistical assumptions, there is a theoretical support for these (the Central Limit Theorem) and they enjoy wide acceptance in the statistical community. Admittedly, these statistical approaches have not seen wide use within machine learning systems, so there is some reason to be cautious about their applicability. However, the current scheduling results and previous demonstrations in artificial planning domains [Gratch92] pro-

vide growing support for the effectiveness and generality of the COMPOSER framework.

An important aspect of statistical frameworks like COMPOSER is their flexibility. In our research we have applied the technique to scheduling and planning tasks, in both cases improving the average time to produce solutions. Cohen and Greiner illustrate how such statistical frameworks can apply to a wide variety of utility functions. For example, by choosing another utility function we could guide DSN-COMPOSER towards influencing other aspects of LR-26's behavior such as increasing the amount of flexibility in the generated schedules.

Our experience in the scheduling domain uncovered several aspects of COMPOSER that can be improved. Its performance is tied to the transformations it is given and the expense of processing examples. Just as an inductive learning technique relies on good attributes, if COMPOSER is to be effective, there must exist good methods for the control points that make up a strategy. Because of the nature of hill-climbing, even if a good strategy exists, there is no guarantee that COMPOSER can find it. One may have to consider carefully how to explore the transformation space.

When it is available, knowledge such as dominance and independence can improve learning efficiency and mitigate the effects of local maxima. An important question is to what extent this information influenced the expected utility of the learned knowledge in the DSN domain. We are currently performing a series of experiments to address this question. We are also interested in whether this type of information could assist learning in our earlier PRODIGY implementation. An obvious question is if such information can be acquired automatically. Another could help overcome the problem of local maxima. One strategy we are investigating is to perform several learning trials, starting the system at a differing random locations in the strategy space. This is similar to the training strategy for neural network systems.

In the LR-26 domain the cost of processing each training example grows linearly with the number of candidates at each hill-climbing step. While this is not bad from a complexity standpoint, it is a pragmatic concern. There have been a few proposals to reduce the expense in gathering statistics. In [Gratch92] we exploited properties of the transformations to gather statistics from a single solution attempt. That system relied on so-called "rejection rules" [Minton88] that only avoid backtracking. The same technique could not be applied to "preference rules" that suggest novel search directions. Greiner and Jurisica [Greiner92] propose one method for evaluating preference rules from a single solution attempt by maintaining upper and lower bounds on the utility of the novel search paths. In the LR-26 domain these bounds are too weak to discriminate between alternative search strategies because there is very little overlap between the search spaces explored by alternative control strategies. Presumably this could also occur in other domains.

Finally, an important issue is the notion of a shifting problem distribution. COMPOSER assumes a stable distribution of problems. In many domains this may not be an appropriate assumption. For example, there are properties in the deep space network that produce distribution shifts: old satellites are deactivated, new satellites are launched, and orbits change in a predictable pattern. We ignored these factors in our experiments by combining all problems into a single batch and choosing a random ordering. One solution to shifting distributions is to use a moving window of problems, periodically retraining the system with new problems added to the front of the window and old problems removed from the end. Alternatively, it might be possible to predict and exploit predictable shifts to guide its behavior.

Acknowledgements

Portions of research was performed at the University of Illinois under National Science Foundation grant NSF-IRI-92-09394 and portions at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration. We thank Colin Bell and Eric Biefeld for many helpful comments and discussions.

References

[Bell92] C. E. Bell, "Scheduling Deep Space Network Data Transmissions: A Lagrangian Relaxation Approach," ***Internal JPL report***, July 1992.

[Buntine89] W. Buntine, "A Critique of the Valiant Model," ***Proceedings of the Eleventh International Joint Conference on Artificial Intelligence***, Detroit, MI, August 1989, pp. 837–842.

[Dechter92] R. Dechter, "Constraint Networks," in ***Encyclopedia of Artificial Intelligence***, Stuart C Shapiro (ed.), John Wiley and Sons, Inc., 1992.

[Fisher81] M. Fisher, "The Lagrangian Relaxation Method for Solving Integer Programming Problems," ***Management Science 27***, 1 (1981), pp. 1–18.

[Gratch91] J. Gratch and G. DeJong, "A Hybrid Approach to Guaranteed Effective Control Strategies," ***Proceedings of the Eighth International Workshop on Machine Learning***, Evanston, IL, June 1991.

[Gratch92] J. Gratch and G. DeJong, "COMPOSER: A Probabilistic Solution to the Utility Problem in Speed–up Learning," ***Proceedings of the National Conference on Artificial Intelligence***, San Jose, CA, July 1992, pp. 235–240.

[Gratch93] J. Gratch and S. Chien, "Learning Search Control Knowledge for the Deep Space Network Scheduling Problem: Extended Report and Guide to Software," Technical Report UIUCDCS–R–93–1789, Department of Computer Science, University of Illinois, Urbana, IL, January 1993.

[Greiner89] R. Greiner and J. Likuski, "Incorporating Redundant Learned Rules: A Preliminary Formal Analysis of EBL," ***Proceedings of the Eleventh International Joint Conference on Artificial Intelligence***, Detroit, MI, August 1989, pp. 744–749.

[Greiner92] R. Greiner and I. Jurisica, "A Statistical Approach to Solving the EBL Utility Problem," ***Proceedings of the Na-***

tional Conference on Artificial Intelligence, San Jose, CA, July 1992, pp. 241–248.

[Held70] M. Held and R. M. Karp," "The Traveling Salesman Problem and Minimum Spanning Trees"," ***Operations Research 18***, (1970), pp. 1138–1162.

[Hogg78] R. V. Hogg and A. T. Craig, ***Introduction to Mathematical Statistics***, Macmillan Publishing Co., Inc., London, 1978.

[Laird92] P. Laird, "Dynamic Optimization," ***Proceedings of the Ninth International Conference on Machine Learning***, Aberdeen, Scotland, July 1992, pp. 263–272.

[Mackworth92] A. K. Mackworth, "Constraint Satisfaction,"in ***Encyclopedia of Artificial Intelligence***, Stuart C Shapiro (ed.), John Wiley and Sons, Inc., 1992.

[Minton88] S. Minton,in ***Learning Search Control Knowledge: An Explanation–Based Approach***, Kluwer Academic Publishers, Norwell, MA, 1988.

[Subramanian90] D. Subramanian and R. Feldman, "The Utility of EBL in Recursive Domain Theories," ***Proceedings of the National Conference on Artificial Intelligence***, Boston, MA, August 1990, pp. 942–949.

[Subramanian92] D. Subramanian and S. Hunter, "Measuring Utility and the Design of Provably Good EBL Algorithms," ***Proceedings of the Ninth International Conference on Machine Learning***, Aberdeen, Scotland, July 1992, pp. 426–435.

[Taha82] H. A. Taha, ***Operations Research, an Introduction***, Macmillan Publishing Co., INC., 1982.

Appendix: COMPOSER Algorithm

Let *PE* denote a performance element. COMPOSER takes an initial element, PE_0, and identifies a sequence, $PE_0, PE_1, \ldots$ where each subsequent *PE* has, with probability 1 - δ, higher expected utility. *TRANSFORMS(PE)* is a function that takes a *PE* and returns a set of candidate changes. *APPLY(t, PE)* is a function that takes a transformation, $t \in$ *TRANSFORMS(PE)*, and a *PE* and returns a new *PE'* that is the result of transforming *PE* with *t*. Let $U_j(PE)$ denote the utility of *PE* on problem *j*. The change in utility that a transformation provides for the *j*th problem, called the *incremental utility* of a transformation, is denoted by $\Delta U_j(t|PE)$. This is the difference in utility between solving the problem with and without the transformation: $\Delta U_j(t|PE) = U_j(APPLY(t, PE)) - U_j(PE)$. COMPOSER finds a *PE* with high expected utility by identifying transformations with positive expected incremental utility. The expected incremental utility is estimated by averaging a sample of randomly drawn incremental utility values. Given a sample of *n* values, the average of that sample is denoted by $\overline{\Delta U}_n(t|PE)$. The likely difference between the average and the true expected incremental utility depends on the variance of the distribution, estimated from a sample by the *sample variance* $S_n^2(t|PE)$, and the size of the sample, *n*. COMPOSER provides a statistical technique for determining when sufficient examples have been gathered to decide, with error δ, that the expected incremental utility of a transformation is positive or negative. The algorithm is summarized in Figure 2.[3]

3. This reflects three differences from [Gratch92]. The first is the correction of a typo in the definition of $\Phi(a)$ that appeared in that paper. The second is superficial – TRANSFORM introduces all transformations at the beginning of a step, instead of incrementally. While we still allow the later, the former reflects the current implementation. Finally, we adopt the δ^* term, recommended by [Greiner92]. The error at each step is dependent on the cardinality of T. While δ^* is overly conservative for most applications, we feel this is more reasonable than the overly liberal previous approach of ignoring the cardinality of T.

Let $PE = PE_0$ T = TRANSFORMS(PE) $j = 0$ $\delta^* = \delta/(2|T|)$

While more examples and $T \neq \varnothing$ do

$j = j + 1$

$\forall t \in T$: Get $\Delta U_j(t|PE)$ /* Gather statistics and find transformations that have reached significance */

$$significant = \left\{ t \in T : j \geq n_0 \text{ and } \frac{S_j^2(t|PE)}{(\overline{\Delta U}_j(t|PE))^2} < \frac{j}{a^2} \right\} \quad \text{where} \quad \Phi(a) = \int_{-\infty}^{a} \left(1/\sqrt{2\pi}\right) exp\{-0.5y^2\} dy = \delta^*$$

$T = T - \{t \in significant : \overline{\Delta U}_j(t|PE) < 0\}$ /* Discard transformations that decrease expeced utility */

If $\exists t \in stopped : \overline{\Delta U}_j(t|PE) > 0$ Then /* Adopt transformation that most increases expected utility */

$PE = Apply(x \in significant : \forall y \in significant \left[\overline{\Delta U}_j(x|PE) > \overline{\Delta U}_j(y|PE)\right], PE)$

T = TRANSFORMS(PE) $j = 0$ $\delta^* = \delta/(2|T|)$

Return (PE)

Figure 2: The COMPOSER algorithm