Learned Subproblem Selection Techniques for Combinatorial Optimization

Robert Moll, Theodore J. Perkins, Andrew G. Barto
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003

Computer Science Technical Report 99-67

November 30, 1999

Abstract

Subproblem generation, solution, and recombination is a standard solution method for combinatorial optimization problems. In many settings identifying suitable subproblems is itself a significant component of the method. Such subproblems are often identified using a heuristic rule. Here we show how to use machine learning to make this identification. In particular we use a learned objective function to direct search in an appropriate space of subproblem decompositions. We demonstrate the efficacy of our technique for problem decomposition on two examples: graph coloring for geometric graphs, a deterministic optimization problem, and the Multiple Uninhabited Air Vehicle Surveillance Control Problem, or MUAV, a stochastic optimization problem that is related to vehicle routing problems in operations research.

1 Introduction

Divide and conquer — that is, subproblem generation, solution, and recombination — is a standard technique in combinatorial optimization. In many divide and conquer settings the search for suitable subproblems is itself a significant component of the technique, and overall results can depend critically on the component subproblems that are actually considered. For example, in a classical vehicle routing problem in Operations Research, a fleet of vans, each with a fixed carrying capacity and each starting from a common depot, is given the job of making deliveries to a set of sites and then returning to the depot. The deliveries assigned to any van must not exceed that van's carrying capacity. The problem objective is to assign deliveries to the vans so as to respect the capacity constraints, and to route vans so as to minimize the total van travel distance. Clearly, how sites are assigned to vans, i.e., which single van subproblems are formed, is at least as important as the routing plan for each van.

How a problem is factored into subproblems, e.g., which sites are assigned to which vans, is often done heuristically. Here we describe an alternative approach to subproblem selection, one that uses machine learning to direct search in the space of problem decompositions. While in principle we do not restrict the form of this search, in both of the examples presented here we use local search over suitably formulated spaces of subproblems.

Suppose an optimization problem P, such as the vehicle routing problem described above, can be solved by first decomposing it into subproblems from some class C, and then by solving those subproblems. In the example above the class C consists of single van routing problems. Now suppose algorithm A solves instances from C. Instead of factoring P into subproblems according to some heuristic rule, as is common practice, we view the task of choosing a decomposition as a search problem in the space of subproblem decompositions. The effectiveness of this search depends on being able to estimate algorithm A's performance quickly on example decompositions. To accomplish this we first identify a set of quickly computable features that capture, with some precision, the expected behavior of A on any instance. Then, using supervised learning in an off-line training procedure, we create $\tilde{A}(\cdot)$, an estimate of A's performance. This is done by running A on a large corpus of examples, and then using regression to fit the feature values of the examples to A's corresponding output values. Once constructed, we use $\tilde{A}$ — a fast, inexpensive substitute for A itself — as a cost function for hillclimbing in the space of subproblem decompositions. When we arrive at a local optimum in "subproblem decomposition" space, we apply A to the subproblem or subproblems associated with the optimum, and use the results obtained to form an overall problem solution.

A number of investigators have studied the use of a learned cost function to direct local search. Zhang and Dietterich [5] consider instances of a NASA space shuttle mission scheduling problem. Using reinforcement learning, they develop a local search-like technique for removing constraint violations from infeasible schedules. In [2], Boyan and Moore develop a technique for acquiring a hill-climbing cost function on-line. They simultaneously learn this function and use it to search for good initial solutions from which to begin conventional local search. Their STAGE algorithm works on specific instances of deterministic optimization problems. Our work in [6] shows how to use a learned value function directly as a generalized cost function within the framework of conventional local search. That work is concerned with demonstrating a methodology for acquiring and then using a learned cost function for standard local search across all instances of a problem class.

All three studies cited above investigate the efficacy of learning a cost function for searching spaces of feasible (or, in the case of [5], infeasible) solutions. Our approach here operates at a different level. We employ local search over a space of viable subproblems in an attempt to find subproblems that are most promising with respect to the estimated behavior of a known algorithm.

We illustrate our technique with two examples, graph coloring for a class of geometric graphs, a deterministic problem, and the Multiple Uninhabited Air Vehicle Surveillance Problem, MUAV, a stochastic, traveling salesman-like problem.

2 Graph Coloring

Graph coloring is one of the most widely studied NP-complete combinatorial optimization problems, and has many important applications, including timetable scheduling, register allocation in compilers, VLSI layout, and channel assignment for mobile communications. We first illustrate our subproblem selection technique by examining graph coloring for a particular subclass of graphs, namely geometric graphs.

A coloring of an undirected graph is an assignment of colors to vertices, with the property that adjacent vertices (vertices connected by an edge) have different colors. Thus a coloring partitions vertices into classes, such that members of the same color class are not adjacent. An optimal graph coloring partitions the vertices of a graph into the fewest possible color classes.

For the class of random geometric graphs we consider, vertices correspond to uniformly randomly generated points in the unit square. Two vertices are adjacent in the graph if the distance between their corresponding points falls below a certain threshold. $U_{N,t}$ denotes the class of geometric graphs on N points with adjacency determined by distance threshold t. A widely studied threshold is t =.5. Geometric graphs model the channel assignment problem for mobile communications systems.

In the case of graph coloring for geometric graphs we proceed as follows. Let C be a subclass of geometric graphs, determined, for example, by parameter t, and let A be any coloring algorithm. We first construct estimator function $\tilde{A}$, which can quickly estimate the behavior of A on a graph in C, i.e., the number of colors A uses to color that graph. (We describe this construction shortly).

Once $\tilde{A}$ has been constructed, we use it to augment our original algorithm A. Let G be an arbitrary geometric graph. First we color G using A. Then we identify a set of color classes where we believe the coloring has been inefficient. Here we choose small classes, because it is more likely that by combining their members and then recoloring, we will be able to reduce the number of colors required. Thus we identify the recoloring of the members of some selected small color classes as the subproblem we want to attack. However there may be other subproblems, obtained by exchanging vertices in this set with other vertices, that could be more promising for recoloring using A. The essence of our method is to use $\tilde{A}$ to search through these subproblems until we arrive at a (locally) most promising one, and then apply A to it. More specifically, our method, applied to (geometric) graph coloring, can be described as follows.

Given geometric graph G, first color G using A. Then:

1. Select a subset of color classes that are appropriate for reorganization into fewer classes (e.g., some of the sparsely populated color classes). Collect the vertices in these classes into a "recolor" set. The collection of vertex sets that are the same size as this set constitutes the space of subproblems to be considered.
2. Using standard first-improvement local search flow of control [1], hill-climb in the space of recolor sets by repeatedly considering pairwise exchanges between a member of the current recolor set and a member of some still-intact color class S. Exchanges are accepted if the swap preserves the integrity of S (here: the alteration of S should introduce no adjacencies to S, i.e., no vertices in S should be linked by an edge), and if $\tilde{A}$'s estimate of A's performance on the recolor set is improved by the exchange.
3. When a local optimum is reached in 2), solve the subproblem by applying A to the newly constituted recolor set.
4. Once this subproblem has been solved, combine its coloring scheme with the other color classes (which may be slightly altered as a result of the exchange process, but which are still legal classes) to get a complete, optimized solution to the problem.

Notice that the above formulation — selecting a subproblem by hill-climbing in subproblem space using a learned estimator for a known algorithm — does not depend on the kind of graph under consideration, and indeed does not depend on the objects being graphs at all. In fact our methodology applies to any constrained partitioning problem and any algorithm A that does a poor job on some of the partitions. For example, the transformation of this methodology to classical one-dimensional bin-packing is quite direct. One-dimensional bin-packing is the problem of packing items of size between 0 and 1 in as few unit-sized bins as possible, such that the items assigned to any bin sum to a value ≤ 1. To lift our graph coloring methodology, start with an approximation algorithm A such as First-Fit-Decreasing [1], and build an estimator, $\tilde{A}$, for it. Apply A to pack the bins. Collect some subset of poorly packed bins and empty their contents, forming a repacking set. Now repeatedly exchange elements of this set for items that still reside in bins. An exchange is accepted if it maintains the integrity of the still intact bin (the sum of that bin's contents remains ≤ 1 after the exchange), and if $\tilde{A}$'s estimate of the repacking set improves as a result of the exchange. When this process reaches a local optimum, apply A to this final repacking set, and combine these bins with the previously packed bins to obtain a complete solution. In section 4 we briefly consider a multi-knapsack packing problem, which illustrates some complications that arise with our model.

2.1 Graph Coloring Results

We considered geometric graphs from the classes $U_{N,t}$ for N = 250, 500, and 750 vertices, and for threshold t = .5. We use the DSATUR algorithm as our base graph coloring algorithm [8]. This algorithm is reported to be the best, or at least competitive with the best algorithms known for geometric graphs [9]. It works as follows:

```
Begin with a random permutation of the graph's vertices.
While vertices remain uncolored:
1. Choose the vertex adjacent to the highest number of distinct colors among
the already colored vertices. (Break ties by taking the lowest-indexed vertex.)
2. Assign the lowest-indexed legal color to that vertex.
```

For our estimation function $\tilde{A}$ for DSATUR we use a linear function approximator. It is constructed using regression from normalized versions of two features and their squares. These features are: the number of edges in the graph (E), and the variance of the degrees of the vertices about the average graph degree (V). Our function approximator is:

$$\mathrm{D}\widetilde{\mathrm{SATUR}}(E,V) = (-1.78 * E) + (2.18 * E^2) + (-.36 * V) + (.47 * V^2) + .77$$

Feature normalization is necessary to account for the size variation across problem instances in the class of recolor vertex sets that are to be searched. Here, too, we used regression: we constructed several thousand example geometric

Algorithm / Problem Size	N = 250	N = 500	N = 750
no swapping	14%	22%	23%
full optimization, RecolorClassCt = 10	56%	69%	75%
full optimization, RecolorClassCt = 15	57%	77%	80%

Figure 1: Frequency of Optimization Improvement over DSATUR

graphs of various sizes, and then did a least-squares fit to determine values for α, β, and γ such that, for example in the case of the variance feature, V satisfies

$$V/(\alpha + \beta * N + \gamma * N^2) = .5$$

We then constructed $\widetilde{\text{DSATUR}}$ by applying linear regression to DSATUR's performance on 10,000 example geometric graphs from sizes 10 to 80 (the approximate size range of recolor vertex sets). For each example graph G we calculated the normalized feature values E, E^2, V, and V^2. We also calculated a target value, the result of applying DSATUR to the graph in question (number of colors used), divided by the number of vertices in the graph. We then used linear regression to fit the feature values to the target values.

The approximate size of the recolor set is controlled by a parameter called the RecolorClassCt, which determines the number of sparse color classes that participate in recolor set formation. Given this value, all singleton classes are included in the recolor set, and as many doubleton classes are added as are needed to achieve the intended size. (If there are too few singleton and doubleton classes, then the algorithm just proceeds with a smaller recolor set).

Figure 1 gives our results on 100 randomly generated geometric graphs of 250, 500 and 750 vertices and for two settings of the RecolorClassCt parameter. In each case we ran DSATUR and recorded the number of colors used. We then formed the recolor set, optimized it according to $\widetilde{\text{DSATUR}}$, and reapplied DSATUR to the optimized recolor set. In this way, we improved on the DSATUR results by at least one color on as many as 57%, 77%, and 80% of the graphs at the three sizes.

For comparison purposes, we also report improvement percentages in the case where we apply DSATUR directly to the recolor vertices, without optimizing with $\widetilde{\text{DSATUR}}$ to improve their potential for recoloring (the row labeled "no swapping" in Figure 1).

Figure 2 shows the running time of the algorithm at various graph sizes and different settings of RecolorClassCt. Results are given in terms of a base running time of 1.0, at each size, for DSATUR.

We also compared the performance of our enhanced DSATUR algorithm against

Algorithm / Problem Size	N = 250	N = 500	N = 750
full optimization, RecolorClassCt = 10	1.3	1.7	1.8
full optimization, RecolorClassCt = 15	2.6	3.6	7.1

Figure 2: Optimization Routine Running Times, DSATUR = 1.0

	N = 250	N = 500	N = 750
full optimization, RecolorClassCt = 10	22%	47%	50%

Figure 3: Enhanced DSATUR improvement frequency over pure DSATUR, equalized for time

pure DSATUR on a time-equalized basis. For this comparison we set RecolorClassCt = 10 since our optimization scheme is much faster at this setting, while performance is only slightly diminished. We compared these algorithms on 100 randomly generated examples at three sizes. For each example we ran DSATUR from 10 random starting orders. We recorded the best coloring found among the 10, and we also recorded the running time for the 10 runs. We then allowed our enhanced DSATUR algorithm to run for an equivalent amount of time. Figure 3 gives the percentage of example instances on which our optimized DSATUR algorithm colored graphs using fewer colors.

While our results for $U_{N,.5}$ are encouraging, especially in light of DSATUR's apparent success on this class compared with other algorithms, we should point out that to be effective, the algorithm requires a certain amount of time-consuming elaboration, which first involves choosing appropriate features, and then requires solving several regression problems, namely feature normalization and learning $\tilde{A}$ off-line. Thus even a class as closely related to $U_{N,.5}$ as $U_{N,.9}$ would require at least several additional hours in order to reestimate $\widetilde{\text{DSATUR}}$. On the other hand, DSATUR's status here is unremarkable: if the XRLF algorithm of [9] proves to be the champion coloring algorithm on a particular class of graphs, then we could just as easily apply our method with it as the base graph coloring algorithm. Furthermore there is no intrinsic reason for using the same algorithm for the base coloring and for the enhancement phase. Indeed, with RecolorClassCt set to 10 and perhaps even 15, the resulting subproblems are small enough that an optimal coloring algorithm, O, and its estimation function, $\tilde{O}$, could be used in the enhancement phase.

3 The Multiple Uninhabited Air Vehicle Surveillance Problem (MUAV)

In this section we study a simplified air surveillance task, in which a fleet of planes is dispatched and controlled to observe (fly over) target sites, subject to uncertain weather conditions. We use linear regression to build a feature-based estimator function that is intended to capture the behavior of a single plane as it flies over a set of sites under the direction of a heuristic controller. In a multiple-plane task, this learned prediction function and a local search procedure are used to partition the set of target sites – each partition being assigned to one plane. This produces single-plane surveillance subproblems, which are then solved using the heuristic controller.

We also experimented with continually optimizing the site partitioning while the planes are in flight and observing. The subproblem divisions are thus adapted on-line to the stochastic progression of the system. This adaptation results in significant performance gains compared to assigning each site to some plane at the start and never changing assignments.

3.1 MUAV Problem Description

Figure 4 depicts an instance of the MUAV problem. The target sites for observation are points in the unit square. Each site has an associated reward — the value for observing the site — and a local weather condition that varies with time. The weather at each site can be cloudy or clear. Only if the weather is clear will an overflying plane be able to observe the site and receive the observation reward. Once observed, there is no further reward if a plane flies over the site again. The weather at a site toggles between cloudy and clear randomly according to a Poisson waiting time distribution.

The fleet of planes departs from a home base at time zero. Control is centralized and there is no hidden information. The central controller knows the location and current headings of all the planes and the locations of, rewards for, and current weather at each site. The goal of the problem is to accumulate as much total observation reward as possible across the whole fleet of planes.

The planes carry limited fuel, which is expended at a unit rate per time. The planes also travel at unit distance per time, and are required to return to base before fuel runs out. The fuel limit can thus be thought of equivalently as a total distance limit. We experiment with fuel allottments ranging from 1.0 (not enough to reach some sites, let alone return) to 5.0 (enough to circumnavigative the unit-area surveillance region with fuel to spare).

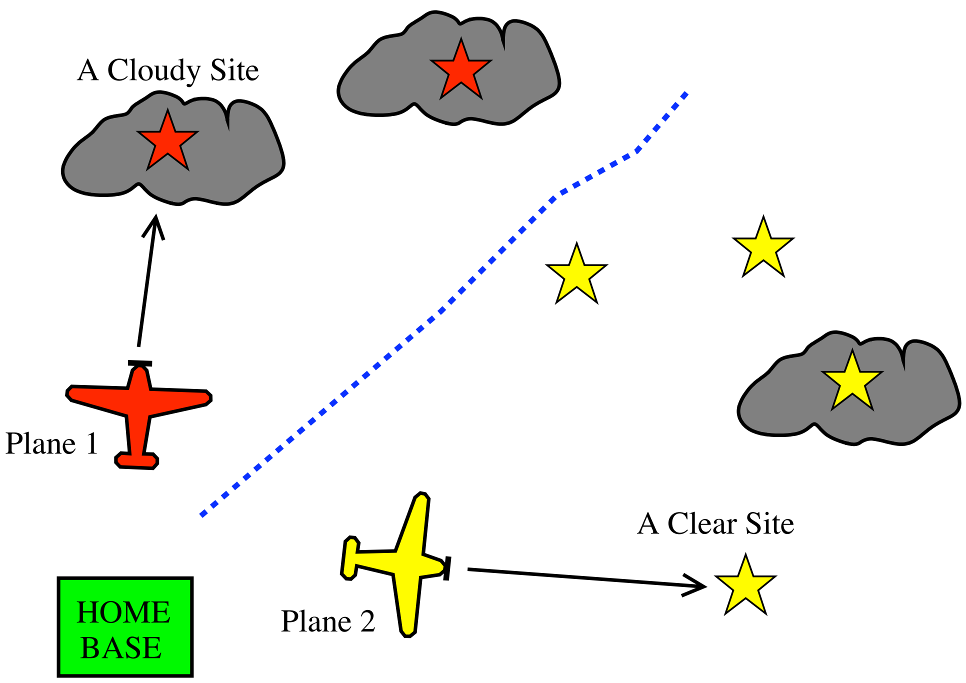

Figure 4: Depiction of an MUAV instance

3.2 The Partitioning Solution

We approach the MUAV task by splitting the sites among the planes, creating a set of single-plane surveillance tasks. These single-plane subtasks are solved by a simple heuristic controller, which we found to be the most effective of several obvious contenders. If possible, a plane flies towards the nearest clear-weather site in its partition. If all the sites are cloudy, the plane simply flies to the nearest of those. And finally, when a plane has no assigned sites, it heads back to base. We call this control rule "Nearest Site, Visible Preferred", or just NVP for short.

The question, then, is how to divide the sites in a multi-plane task among the planes for solution by NVP . To do this, we first learn $\widetilde{\text{NVP}}$, an estimate of the performance of NVP on arbitrary, hypothesized single plane instances. A local search procedure in the space of partitions, using performance estimator $\widetilde{\text{NVP}}$ to evaluate components of a candidate partition, then identifies a promising division of the MUAV instance into a collection of single-plane problems.

3.2.1 Estimating performance of NVP

To learn $\widetilde{\text{NVP}}$, we generated 500 random, single-plane surveillance problem instances, with 6-10 sites. The number of sites, their placement in the unit square, and initial weather conditions were all chosen uniformly randomly. The plane was given 3.0 fuel, and was allowed to reevaluate and possibly choose a new heading every 0.025 time units. The home base for the plane was at the origin, (0,0). The changes in weather at each site occurred at discrete times according to Poisson process with parameter $\lambda = 2.0$.

On each instance, we simulated one run of NVP , recording at each decision point 18 features and the total observation reward achieved from that point forward in the run. We did a linear least-squares fit to predict the fraction of remaining reward that will be achieved. Multiplying by the amount of remaining reward thus predicts the total observation reward. It is this linear approximation that we use to estimate the performance of NVP on potential single-plane subproblems resulting from the decomposition of an MUAV instance. See figure 5 for a description of features and the fitted weights.

3.2.2 Optimizing subproblem decomposition

To divide a MUAV task into single-plane subproblems, we begin by randomly assigning each site to some plane. From this initial partitioning, we run a best-improvement local search procedure. For any partitioning, the "neighboring"

Feature	**Description**	**Weight**
1	Sum of site distances to centroid of unobserved sites	0.1896
2	Mean of site distances to centroid of unobserved sites	-0.2791
3	Plane distance to centroid of unobserved sites	-0.0373
4	Home base distance to centroid of unobserved sites	-0.2721
5-8	Like 1-4 but base included in computation of centroid	-0.2145
		0.2528
		0.0137
		0.3088
9	Plane distance to nearest clear site	-0.0507
10	Indicator (0/1) of no clear sites	-0.0382
11	Plane distance to nearest site, cloudy or clear	0.0184
12	Indicator (0/1) of no unobserved sites at all	0.0
13	NNTSP: Reward if plane were to visit sites in closest-first order until fuel is low, assuming weather always clear (as a percentage of total unobserved reward remaining)	0.2997
14	Like 13 (NNTSP) but visitation reward weighted by probability of clear weather when plane gets there.	0.6651
15	Like 14, but only first site is so weighted	-0.2836
16	Fuel left after NNTSP tour	0.0619
17	Fuel left now	0.0820
18	Total unobserved reward out there	-0.0001
19	Bias (+1.0)	0.1034

Figure 5: Features and linear weights for predicting NVP performance.

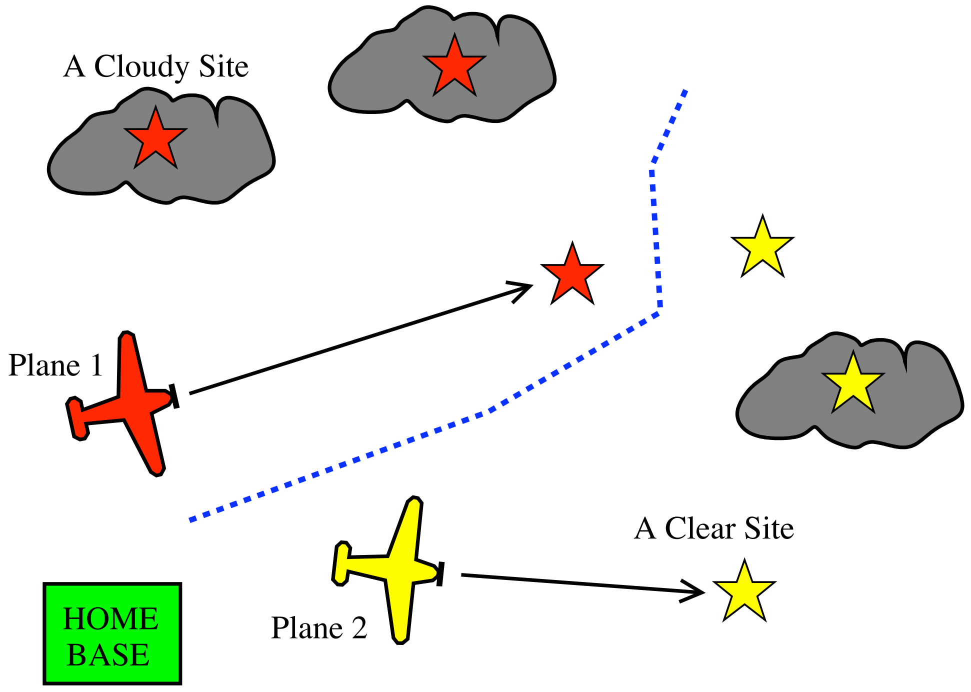

Figure 6: Result of a single local search step.

partitionings are those obtained by reassigning a single site to a different plane. Figure 6 gives an example of a search step. Compared with figure 4, one of the sites has been moved to the partition of plane 1, and that plane is now heading toward the site in accordance with the NVP rule.

The local search examines the neighboring partitionings, finding the one with best estimated value — just the sum of the NVP estimates for each partition. If the best neighbor has better estimated value than the "current" partitioning, that neighbor becomes current and search proceeds from there. When arriving at a local optimum of estimated value in this space of partitionings, search terminates. Each plane is then controlled by NVP as if the sites in its partitions are the only ones that exist.

We also experimented with repeated, or "continual" optimization of the partitions. Under this scheme, each time the controller considered new headings for the planes the partitioning was also reoptimized. We used the same local search procedure, starting from the current partitioning. This allowed the controller of the fleet of planes to react to variable performance caused by weather. For example, if one plane had made little progress because of cloudy weather at its sites while another had made much progress in its partition, then sites might be reassigned to balance current loads among planes. The continual opti-

Fuel Allotted	NVP estimate continuously optimized	NVP estimate optimized. No repartitioning after trial starts	Compactness continuously optimzed	Compactness optimized. No repartitioning after trial starts
1.0	**12.5%**	11.7%	9.5%	9.6 %
2.0	**42.5%**	39.7%	37.5%	38.8%
3.0	**74.2%**	69.6%	70.7%	69.5%
4.0	**91.0%**	89.9%	**90.8%**	87.9%
5.0	97.6%	97.1%	**98.3%**	96.6%

Figure 7: Total observation reward achieved by different partitioning schemes at different fuel levels, as percentage of reward available.

mization even allows the system to respond to changes not modelled as part of the dynamics of the system. For instance, if we were to remove or add planes, or change the sites, the system would seamlessly redivide the problem for the individual planes.

3.3 Results

We compared four different partitioning schemes on 100 random 24-site, 3-plane MUAV instances, with different amounts of fuel. We tried partitioning based on the linear NVP performance estimates and also partitioning based on compactness – feature 1 in figure 5. Partitioning based exclusively on compactness means that we accepted a single move site reassignment when by doing so we improved (reduced) the sum of the compactness measures for the three groups of sites. Compactness-based partition was the best heuristic method we tested, outperforming sector-based partitioning, a standard approach in multiple-vehicle routing problems. For both, we ran trials in which the initial partitioning was held fixed throughout the run, and trials in which the partitioning was continually optimized.

Figure 7 summarizes the results. Within each row, the boldface numbers are statistically significantly larger than the other numbers, by a t-test at p=0.05. Through much of the fuel range studied, optimizing NVP led to better partitioning than did compactness-based partitioning. At the highest levels of fuel, compactness performs a little better. In the single-plane trials that provided the data for the NVP estimate, the plane always started with 3.0 fuel. It is possible that adjusting this would change the range where partitioning based on NVP performs best.

Note also that the continual optimization of partitions yielded significant improvements over sticking with initial partitionings. In this domain, redividing the full problem into different subproblems is easy to do: there is little cost in moving a site to another plane's partition. In other stochastic systems where subproblem divisions may be revised with little cost, such an approach may prove more effective than algorithms that commit to decisions once made. For example, multiprocessor scheduling algorithms usually maintain a queue for each processor, and when a program is submitted, that program is placed on a particular processor queue. In the case where processes have unpredictable running times, our approach suggests a way in which computational loads might might be balanced more effectively.

4 Discussion

We have demonstrated a general learning-based method for subproblem exploration which, we believe, holds great promise for algorithm development for both deterministic and stochastic combinatorial optimization. For example, the subproblem conditioning technique used for geometric graphs should be applicable to a wide variety of scheduling, packing, and layout problems, where regions that are poorly scheduled, packed, or layed-out play the role of the recolor set in our graph study. As our MUAV example shows, our method can also be effective for developing stochastic optimization algorithms. A general weakness of the method is the degree to which the nature of the particular problem class can be captured using rapidly computable features. Thus, we showed good results for graphs in the class $U_{N,.5}$, but the method's effectiveness for other classes of graphs, which might require completely different features, is unproven.

Our MUAV example illustrates two other novel aspects of the method. First, our estimator is fast enough (and the time demands of the problem are sufficently undemanding) that it is feasible to reoptimize repeatedly on a very fine time scale. This process not only yields an apparently superior algorithm, but also demonstrates a way in which our method can be used to simplify the construction of an effective algorithm for a complex stochastic problem. That is, the factorization style we employed meant that we needed to construct an estimator function via simulations for the single plane case only. The overall stochastic characteristics of the full MUAV are handled by the resulting single plane estimator, combined with a nonstochastic parititioning algorithm, applied repeatedly to respond to uncertainites as they arise in the simulated world. We believe this methodology holds promise for other, similarly factorable stochastic optimization problems, e.g., elevator scheduling and multiprocessor scheduling.

Next we identify an issue that will greatly affect the breadth of application of the method. Imagine a multiple knapsack packing problem: given a collection of N identical knapsacks, and a large collection of objects of varying sizes and values, pack pieces in the N knapsacks to achieve the highest total value possible, subject to the capacity constraints of the knapsacks. Suppose we proceed exactly as we have for graph coloring: given algorithm A for multiple knapsack packing, we form $\tilde{A}$ off-line, as before. Next we apply A, and then collect and empty out the poorly packed knapsacks, forming an "unpacked" set from the objects in these knapsacks and any other as yet unused objects. We wish to repack the emptied knapsacks, again using A. Proceeding as we did with graph coloring, we attempt to make exchanges between objects still in knapsacks, and elements of the unpacked set. But now we encounter a difficulty that did not arise in the graph coloring example. Suppose we come to a proposed exchange between an unpacked set element p and an element q that is still in a knapsack, say K, with the following consequences: 1) suppose exchanging p for q leaves K legal; 2) the value of K's contents falls because of the exchange (this is bad, since total value across all knapsacks is our primary objective); but 3) $\tilde{A}$'s estimate of the value of the unpacked set improves due to the exchange, because q has replaced p in the unpacked set (this is good). Do we accept the exchange? The crux of the matter is the need to understand the trade-off between falling real values (changes in K), and estimated improvement ($\tilde{A}$'s assessment of the unpacked set).

This last issue may well be affected by the level of precision of the function approximator we employ. In the two examples we have considered so far, we have used only simple linear function approximators. This is an asset, in that we seem capable of achieving good results with comparatively weak estimator functions, which only need to supply qualitative information about estimated algorithmic performance. But as the knapsack example shows, some quantitative information about estimator accuracy may be necessary in order for the method to operate more broadly. Thus a more sophisticated, statistically qualified, function representation may be needed in order for hill-climbing of the kind we have described to be broadly effective.

This research was supported by a grant from the Air Force Office of Scientific Research, Bolling AFB (AFOSR F49620-96-1-0254).

References

[1] C. H. Papadimitriou, and K. Steiglitz, K.(1982). *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ.

[2] J.A. Boyan, and A.W. Moore (1997). Using Prediction to Improve Combinatorial Optimization Search, Proceedings of AI-STATS-97.

[3] P. Healy and R. Moll (1995). A New Extension to Local Search Applied to the Dial-A-Ride Problem. *EJOR*, 8: 83-104.

[4] P.Healy (1991). *Sacrificing: An Augmentation of Local Search.* Ph.D. thesis, University of Massachusetts, Amherst.

[5] W. Zhang, and T.G. Dietterich (1995). A Reinforcement Learning Approach to Job-Shop Scheduling, Proceedings of the 14th IJCAI, pp. 1114-1120. Morgan Kaufmann, San Francisco.

[6] R. Moll, A.G. Barto, T.J. Perkins, R. S. Sutton (1999). Learning Instance-Independent Value Functions to Enhance Local Search, *Advances in Neural Information Processing Systems 11*, M. Kearns, S. Solla, D. Cohn, eds., pp. 1017-1023, MIT Press, Cambridge, MA.

[7] B.W. Kernigham and S.Lin (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49: 291-307.

[8] D. Brelaz, New Methods to Color Vertices of a Graph, *Comm. ACM* 22: 251-256.

[9] D. Johnson, C. Aragon, L. McGeoch, C. Schevon, Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning (1991). *OR*, 39: 378-406