Selecting the Right Heuristic Algorithm: Runtime Performance Predictors

John A. Allen[1] and Steven Minton[2]

[1] Caelum Research Corporation, Mail-Stop 269-2, NASA Ames Research Center, Moffett Field, CA 94035-1000, allen@ptolemy.arc.nasa.gov

[2] USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, minton@isi.edu

Abstract. It is obvious that, given a problem instance, some heuristic algorithms can perform vastly better than others; however, in most cases the existing literature provides little guidance for choosing the best heuristic algorithm. This paper describes how runtime performance predictors can be used to identify a good algorithm for a particular problem instance. The approach is demonstrated on two families of heuristic algorithms.

1 Introduction

Using a good heuristic algorithm can make a tremendous difference in the efficiency of solving a constraint-satisfaction problem (CSP). Without a good algorithm, solving even a moderate-sized CSP (or any combinatorial problem) may be extremely time consuming. This is why there are so many papers written each year about new heuristic CSP methods. A great variety of heuristic algorithms have been described in the literature, each purported to perform well on some example problems. Unfortunately, it is rarely clear which method will perform best for a given problem, and the literature provides almost no guidance on this topic.

In this paper, we illustrate the difference selecting the "right" heuristic algorithm can make by citing several cases in the current literature. We then argue that selecting a suitable algorithm can be surprisingly difficult, and certainly non-obvious. To begin to address this problem, we present evidence that, for certain families of heuristic algorithms, we can predict which algorithm will do best after running them each for a very short period of time. We do this by identifying *secondary performance characteristics* that tell us whether the heuristic is having a positive effect. We demonstrate this idea for two very different families of heuristic algorithms: backtracking constraint-propagation and iterative repair tabu search.

2 Finding the Right Heuristic Algorithm is Important

Recently, researchers have been interested in identifying "hard" constraint satisfaction problems (e.g., [1, 10]). Informally, an problem is *hard* if no algorithm can solve "most" of its instances "quickly", for suitable definitions of "most" and "quickly". Similarly, a CSP problem is *trivial* if most CSP algorithms can solve most of its instances quickly. In our view, many problems we are likely to encounter fall into a middle class. Most of the instances of these *challenging* problems can be solved quickly, but only if the right CSP method is applied.

Some recent studies provide examples of challenging problems — problems that were considered hard until the right method was found:

- Smith [18] describes a DARPA scheduling problem for which he synthesized an algorithm that was orders of magnitude faster than previous methods. By making heavy use of constraint propagation techniques, Smith's algorithm completely eliminated search in some cases.
- Minton et al. [9] have demonstrated that an iterative repair method solves instances of the N-queens problem easily, even if $n = 10^6$, and Kale [5] found a backtrack version that works well as well. While N-queens is not intrinsically interesting, it is notable because it has long served as a benchmark for AI search methods.
- Researchers [11, 4, 16] have recently identified heuristic techniques that quickly solve a job shop problem originally proposed by Sadeh [15]. For instance, Johnston found that a combination of a dispatch heuristic and iterative repair search solved most of the instances almost immediately.

In each case, benchmark problems that were meant to be difficult were eventually "cracked" by heuristic methods; the problems were solved so rapidly that they became unsuitable as benchmarks! In other words, with the appropriate heuristics, the problems were not, in fact, very hard at all despite their initial appearance.

While there has been recent work analyzing the nature of hard problems and proposing that such problems are "relatively rare", we know of no studies examining the distinction between challenging and trivial problems. Yet this distinction is extremely important to the practitioner, since challenging problems are, in fact, the ones that *seem* difficult to solve, but are easy if the right algorithm is used.

3 Finding the Right Algorithm can be Difficult

If we could identify a few CSP algorithms that were better than all the rest, then we wouldn't need to worry about which algorithm to select for a given problem, we could simply run them all in parallel. Unfortunately, as we pointed out, it seems there is a very large variety of CSP algorithms to choose from, perhaps thousands. For instance, in the IJCAI-93 conference there were at least eighteen new algorithms described. While one can more or less group the algorithms into different approaches (such as backtracking, branch-and-bound, iterative repair, genetic search and divide-and-conquer), there are many different heuristics that can be utilized by these approaches [3] and many different ways to combine the different techniques. Some authors have tried to characterize the types of problems for which their algorithms are most appropriate. However, there is often no general way to do this except with respect to very gross characteristics, such as whether the problem is known to be solvable or not. The trouble with using gross characteristics is that in many cases we have found that seemingly small differences in an algorithm, or the representation of a problem, can make a big difference in performance.

As an example, we describe our recent experience comparing min-conflicts [9] and GSAT [17], two iterative-repair algorithms. Min-conflicts begins with an initial assignment. On each iteration, min-conflicts randomly chooses a variable that is in conflict and assigns it the value that minimizes the number of remaining conflicts. GSAT is a descendant of min-conflicts designed for boolean satisfiability problems. GSAT differs from min-conflicts in that it selects the variable that eliminates the most conflicts when flipped rather than choosing randomly among conflicted variables.[4]

Selman et al. [17] claims that GSAT performs well on graph-coloring problems and describes a 17-color problem where GSAT significantly outperforms other algorithms. This result surprised us, since Minton et al. [9] reports that min-conflicts performs relatively poorly on graph-coloring problems! We recently took a look at this discrepancy and were surprised to see that the differences in the algorithms, which are seemingly minor, result in significant differences in performance.

In a graph-coloring problem, each node must be assigned a color, and adjacent nodes must not have the same color. On most of the graphs that we tried, we found that GSAT did in fact outperform min-conflicts. Initially we hypothesized that this was due to the representation difference. (Min-conflicts uses the standard CSP representation scheme, where each node is a variable whose value is a color. GSAT uses a propositional representation scheme of the form "NodeX is ColorY", allowing nodes to have multiple, or no, colors.) However, this hypothesis was insufficient to explain our results, since min-conflicts does not do significantly better using the propositional representation. In fact, in some cases the representation difference actually benefits min-conflicts. After taking a closer look at GSAT's performance on the 17-color problem, using code and data supplied by Selman, we found that:

[3] For instance, some heuristic variations of "intelligent backtracking" mentioned in AAAI-94 [?] articles include: dynamic backtracking, backjumping, conflict directed-backtracking, ATMS-based search schemes, learning with backjumping, value-based shallow learning, graph-based shallow learning, jump-back learning, backmarking, backmarking and sticking values, Min-conflict backtracking, weak-commitment search.

[4] In their original paper introducing GSAT, the authors note that a distinguishing difference between the two algorithms is that GSAT makes sideways moves and min-conflicts does not. This claim is incorrect as both algorithms make sideways moves.

1. GSAT only does well if one leaves out the constraint that each node must be colored a single color. (If GSAT finds a satisfying assignment where a node is colored two colors, then selecting either color will give a solution.)
2. GSAT requires a tabu list. (Selman used a tabu list of length 20. A tabu list of length k keeps track of the last k moves, so they are not repeated by the repair process[2, 3]. This can help the algorithm cope with local minima and plateaus.)
3. GSAT's variable-ordering strategy helps significantly.

Armed with this knowledge, we were able to create a variation of min-conflicts that does almost as well as GSAT on the graph-coloring instances we tried. The algorithm uses GSAT's variable-ordering strategy *and* a tabu list, but not GSAT's propositional representations. It took several weeks to complete this study.

While this example might be extreme, we believe that the difficulty of finding an appropriate algorithm for a problem is often under-appreciated. The difficulty is aggravated when an algorithm incorporates a variety of heuristic mechanisms, as is often the case, since these can interact in unexpected ways. For example, Minton [8] found that different backtracking algorithms were appropriate for two distributions of a CSP called Minimal Maximum Matching. The two backtracking algorithms used different variable and value ordering strategies. Because, in each algorithm, the strategies interacted synergistically, finding the right algorithm for a given distribution required simultaneously trying both the right value-ordering strategy and the right variable-ordering strategy.

4 Predicting which Algorithm is Best

In the previous sections we argued that, when presented with a problem instance, it may not be obvious which algorithm to choose, even though one algorithm might end up performing much better than others on that instance. One possible solution is to run a variety of algorithms in parallel. This can lead to a significant improvement in performance, e.g., super-linear speedup [13], but typically resource limitations prevent us from running more than a few algorithms in parallel. Here we consider an approach, *runtime performance prediction* (*RPP*), whereby we monitor an algorithm as it runs for a short period and predict its performance. We can use this approach to test a variety of algorithms quickly.

A *secondary performance characteristic* is a property of the search process that we can monitor in order to predict an algorithm's final performance. Which properties we choose to monitor depends on the heuristics incorporated in the algorithm. If we understand the effect that a heuristic is supposed to have on the search, we can choose a property that provides evidence as to whether or not the heuristic is having its intended effect. We demonstrate two different properties that enable us to predict how well an algorithm will do. For illustrative purposes, we chose two very different types of algorithms for our demonstration: constraint-propagation methods in a backtracking framework and tabu search in an iterative repair framework.

4.1 Estimated Total Constraint Checks

Constraint-propagation (CP) [12, 7] is frequently used in conjunction with backtracking search. The idea is simple: whenever a variable is instantiated with a new value, we can prune the domains (the possible values) of the uninstantiated variables using consistency maintenance techniques as follows. Given two variables, V and U, we say that $arc(V, U)$ is consistent if for every value v in the domain of V there exists a value u in the domain of U that is consistent with v. The procedure "revise" is used to enforce arc consistency: applying $revise(V, U)$ removes values from the domain of V until $arc(V, U)$ is consistent. Nadel [12] describes the following CP heuristics, where U is the variable just instantiated, and $V_1, \ldots, V_n$ are the uninstantiated variables:

1. Forward checking (FC): After instantiating variable U, $revise(V_k, U)$, for $k = 1, \ldots, n$.
2. Partial Lookahead (PL): Do forward checking, and in addition, for each pair of uninstantiated variables, V_i, V_j, such that $i > j$, $revise(V_i, V_j)$.
3. Full Lookahead (FL): Do forward checking, and in addition, for each for each pair of uninstantiated variables, V_i, V_j, such that $i \neq j$, $revise(V_i, V_j)$.
4. Really Full Lookahead (RFL): Do forward checking, and then revise arcs until the constraint graph is fully arc-consistent (i.e., until quiescence).

Each heuristic implements a different level of consistency maintenance. Notice that they can be implemented so that in a given state the values pruned by each successive level will include the values pruned by the preceding level. In this sense, each successive CP heuristic is more powerful than the preceding one. However, more powerful does not necessarily result in better overall performance. In fact, recently Sabin and Freuder [14] have shown that the relative performance of these techniques is problem-dependent.

There is a simple model that allows us to understand the basic effects of CP. Simply put, CP techniques increase efficiency by reducing the amount of backtracking required to find a solution. The reason they do not always lead to improved performance is because they also tend to raise the number of constraint checks performed at each node. These two effects compete against each other.

We can monitor the degree to which CP is increasing efficiency by estimating the size of the tree and the number of constraint checks per node[5]. This gives us an estimate of the total number of constraint checks that would be required if the entire tree were to be searched, which we refer to as $EstTCC$. The procedure is based on an iterative sampling approach described by Knuth [6] for estimating the size of the search tree, which we adapt for use with depth-first search, as described below.

Let $CC(n)$ be the number of constraint checks that occur at node n, and $k(n)$ be the branching factor at node n. Let $P = \{n_1, \ldots, n_j\}$ be the path of

[5] We use the number of constraint checks in our estimates since they correlate well with time, and they are implementation independent, but the estimates could be done directly using time as our basic unit of cost.

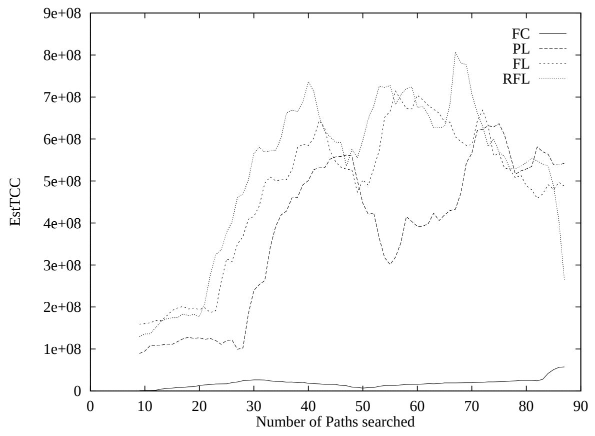

Fig. 1. The behavior of the secondary characteristic *EstTCC* during problem solving. (The data in the graph has been smoothed to make the graph more readable)

instantiated variables from the root, n_1, to a leaf, n_j, where n_{i+1} is the child of n_i, and a leaf is a node whose children cannot be expanded. Given P, we calculate $EstTCC(P)$, the estimated number of constraint checks required to search the tree, as follows:

$$EstTCC(P) = \sum_{i=1}^{j}[CC(n_i)\prod_{l=1}^{i-1}k(n_l)]$$

Using essentially the same argument as presented by Knuth, it is straightforward to show that the expected value of $EstTCC$ is the number of constraint checks in the tree, provided the variable domains are randomized. (Space limitations do not permit its inclusion.) The difference is that we calculate this sum while the CP algorithms are conducting backtracking search, and average over the last 10 $EstTCC$ values. Knuth uses an iterative sampling approach, which gives estimates with a lower variance after exploring the same number of nodes, and so iterative sampling is an alternative method that can be considered. (There are a variety of tradeoffs which we cannot address here.)

$EstTCC$ gives us a basis for comparing the CP algorithms, because the algorithm that performs the fewest constraint checks is the most efficient. However, $EstTCC$ is the estimated number of constraint checks for the *entire* tree, but our CP search procedures will terminate as soon as the first solution is found. Nevertheless, if we run two or more algorithms side by side, we can predict which will do best based on a running comparison of $EstTCC$ for each search tree. This prediction is based on the fact that the trees contain the same number of solutions and on the assumption that the solutions are distributed in the same way throughout both trees.

Figure 1 shows how the value of $EstTCC$ changes as the different constraint propagation techniques solve a problem. The problem instance is a randomly generated CSP of 40 variables with domains of 7 values, a density of .5, and a tightness of .15. The independent variable is the number of paths, from the root to a leaf, examined. The dependent variable is $EstTCC$. For this instance, FC does 108,319 constraint checks to solve the problem; PL, FL, and RFL require 795,029, 954,538, and 879,521 constraint checks respectively.

While the example in Figure 1 shows that $EstTCC$ correctly predicts that FC will perform the fewest constraint checks, the intent of using a secondary characteristic is to allow one to determine the best algorithm without having to run each to completion. This requires that one know how long to let the CP systems run before comparing their $EstTCC$ values. Since $EstTCC$ is an estimate based on a single path through the tree, and since it is unlikely that any one path will be representative of the tree, we average the $EstTCC$ values of first ten paths explored by the algorithm.

Table 1 summarizes this information for 11 problem instances (10 randomly generated problems and one hand-coded problem) for the four CP algorithms. The first column specifies the problem instance being examined. The second column presents the number of constraint checks required by the algorithm chosen by comparing $EstTCC$ values. The third column is the number of constraint checks performed by the best CP algorithm. The fourth column, degradation over best, presents the percent increase in the number of constraint checks incurred by the chosen algorithm over the best algorithm. The final column show the percent increase over choosing an algorithm at random.

Table 1 show that $EstTCC$ works well with random problems. In each case the best algorithm was selected, resulting in an improvement over random by a factor of 2 to almost 10. The hand coded instance (HC) shows an interesting difficulty with using $EstTCC$. Each path, P, evaluated by $EstTCC(P)$ is assumed to be representative of the tree as a whole. If the tree has a section that is shallow and sparse, then samples taken from that part of the tree will make it look very small. Conversely, if a part of the tree is very bushy and deep, then samples taken from that section can make the tree look much larger than it really is[6]. The domain HC was specifically designed to be conducive to the more powerful CP techniques, and in the case, a great deal of pruning was done by RFL near the root of the tree. As a result, RFL appeared to be searching the tree with the fewest constraint checks. (In this instance, PL performed best.)

Table 2 summarizes the cost of testing and running the chosen algorithm. The first column labels the problem. The second column shows the number of constraint checks needed to calculate $EstTCC$ for each of the four algorithms, FC, PL, FL, and RFL. The third column shows the sum of the constraint checks used during testing, and the number of constraint checks needed by the chosen algorithm to solve the instance. The fourth column shows the expected reduction in the number of constraint checks of using runtime performance prediction (RPP) versus interleaving the execution of the four algorithms. The expected number of constraint checks for interleaving four algorithms is calculated as four times the number of constraint checks of the best algorithm.

[6] Knuth notes this as well[6].

Prob.	Inst.	CC's of Chosen Algorithm	CCs of Best Algorithm	Degradation over Best	Improvement over Average
Rand1	1	601,105	601,105	0%	370%
	2	765,664	765,664	0%	493%
	3	108,319	108,319	0%	532%
	4	1,411,902	1,411,902	0%	432%
	5	592,427	592,427	0%	527%
Rand2	1	126,152	126,152	0%	255%
	2	340,814	340,814	0%	547%
	3	125,569	125,569	0%	273%
	4	30,741	30,741	0%	964%
	5	55,556	55,556	0%	348%
HC	1	480,464	171,465	180%	-2%

Table 1. A comparison of number of constraint checks performed by the CP algorithm chosen by *EstTCC* against the constraint checks performed by the best algorithm and the average number of constraint checks performed by all four algorithms. Rand1 and Rand2 are randomly generated binary CSPs, HC is hand-crafted.

Prob.	Inst.	Test CCs	Total CCs	Improvement over Interleaved
Rand1	1	201,577	802,682	199%
	2	209,956	975,620	214%
	3	233,157	341,476	27%
	4	200,061	1,611,963	250%
	5	163,446	755,873	213%
Rand2	1	202,360	328,512	53%
	2	180,804	521,618	161%
	3	188,152	313,721	60%
	4	203,061	233,802	-47%
	5	209,541	265,097	-16%
HC	1	185,439	665,903	3%

Table 2. The cost of using *EstTCC* for testing CP algorithms versus interleaving the algorithms.

Our results show that *RPP* using *EstTCC* can result in fewer constraint checks than interleaving. There are a few instances where the number of constraint checks was large enough relative to the number of constraint checks performed by the best algorithms as to overwhelm the benefit. However, the penalties in these two examples are relatively small, and we are currently investigating ways of reducing the testing time using decision theory.

In general, one of the advantages of the *EstTCC* predictor is that it can be adapted for use with a variety of backtracking heuristics. For example, variable-ordering strategies and intelligent backtracking strategies are both similar to CP in that they work by reducing the size of the search tree, and thus *EstTCC* should work similarly with them. Of course, not all backtracking heuristics operate by reducing tree size. For instance, value ordering heuristics do not work this way, and therefore another predictor must be used to gauge whether a value-ordering heuristic is having a positive effect.

4.2 Minimum Conflict Count

For our second family of heuristic methods, we consider the use of tabu [2, 3] lists by GSAT. As explained in the previous section, tabu lists are used to record the last k moves, so that they will not be repeated. Tabu lists are one of many heuristic techniques designed to help with the problems of plateaus and local minima encountered by non-systematic local search methods. Other such techniques include stochastic moves (as in simulated annealing) and randomized descent (as in Min-conflict's variable selection method).

Of course, there is the problem of determining if a tabu list will help (or hurt) and how long the list should be. Based on an informal model of what a tabu list is supposed to do – escape large plateaus and local minima, we can attempt to monitor the degree to which this is succeeding. However, since it is often expensive to determine when a plateau or local minima is encountered, we measure this indirectly. Specifically, we execute a series of relatively short sample runs recording the number of conflicted, or unsatisfied, clauses after each flip during the run. Presumably, if the tabu list is having a positive effect, this should be reflected by a decrease in the number of conflicted clauses.

The *Minimum Conflict Count*, *MCC*, is calculated by making several short runs with the heuristic algorithm, taking the minimum conflict count that occured in each run and averaging them. The minimum number of conflicts is used because it gives some indication of how well the system is navigating plateaus and local minima. An iterative repair algorithm with a poorly chosen tabu list length will have difficulty escaping these obstacles, and will not have as good an opportunity to reduce the conflict count as a system with a good tabu list length. We average the minimum conflict count because of the strong stochastic nature of GSAT and other non-systematic techniques.

Figure 2 shows how *MCC* behaves as the number of flips (the function GSAT uses to assign a variable a value) increases. In this example, GSAT is run on a graph 3 coloring problem with 200 nodes and 600 links, translating into 600 variables and 2600 clauses for GSAT. The data was collected after every 100 flips and is an averaged over 1000 tries. Notice that for the first 200 flips on this problem, the 6 different tabu list lengths behave identically. This indicates that during this time there are no significant obstacles that needs to be overcome in the early part of the search. However, after the first 200 flips the *MCC* of the 6 systems start to diverge. In this example, the best tabu length is 25, in which one can expect GSAT to find a solution in 11,740 flips. Tabu lengths of 0, 5, 15,

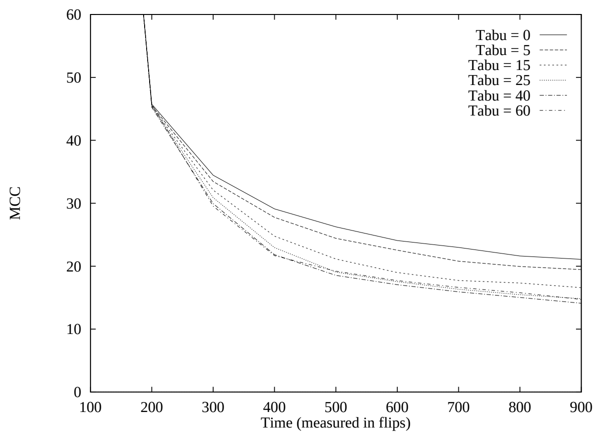

Fig. 2. The behavior of the secondary characteristic *MCC* during problem solving.

Prob.	Inst.	Ave. Flips of Chosen Alg.	Ave. Flips of Best Alg.	Degradation over Best	Improvement over Average
GC	1	2,873,056	1,255,913	129%	15%
	2	12,409	6,529	90%	2,474%
	3	123,537	99,632	24%	-7%
	4	94,730	29,678	219%	-10%
	5	48,957	10,635	360%	-22%
Plan	1	2,945	2,945	0%	2,468%
	2	19,167	11,177	71%	510%
	3	27,078	19,763	37%	540%
	4	12,055	12,055	0%	1,392%
	5	225,120	14,496	1,453%	-16%
Sat	1	62,288	53,099	17%	245%
	2	299,293	34,432	769%	-49%
	3	64,009	13,458	376%	33%
	4	170,636	54,610	212%	58%
	5	425,584	219,962	93%	-3%

Table 3. A comparison of the average number of flips performed by the tabu algorithm selected by *MCC* against the average flips performed by the best algorithm and the average flips performed by all eight algorithms. GC is a set of randomly generated graph-coloring instances. Plan is a set of four move blocks world instances, and Sat is a set of randomly generated satisfiability instances.

40, and 60 require 117,967, 104,253, 23,351, 21,943 and 70,680 flips respectively to find a solution.

Our method for determining when to compare *MCC* is to let GSAT perform n flips, where n is the number of variables in the problem, and to average them over ten trails. Our aim is to give the system enough time to reach and escape the first few plateaus or local minima it finds during the search. Notice in Figure 2 that the tabu lengths of 25, 40 and 60 are closely clustered together. Our experiments indicate that the MCC values will eventually diverge, but it takes many more flips before it happens. When there is a cluster of algorithms that have approximately the same MCC, choosing the algorithm with the shortest tabu length seems to be a good choice. Thus, we use the following procedure: Collect all the algorithms whose MCC value is within 1 of the minimum MCC value. Of these algorithms, select the one with the shortest tabu list.

Table 3 shows the results of using MCC on several instances.[7] The format is similar to that of Table 1. The first column presents the problem and instance tested; the second shows the average number of flips needed by the algorithm selected to solve the instance. The third column is the average number of flips needed by the best algorithm to solve the instance. The fourth column shows the percent increase in the number of flips of the chosen over the best, and the last column is the percent improvement over selecting an algorithm at random.

The first thing to notice about the data in Table 3 is that it is not as consistent as that for the CP programs. While good improvements over average selection can be seen, there are also examples where the performance of the chosen algorithm is worse than average. A closer look at the data shows that our method of determining the best tabu length tends to underestimate. For example, the best tabu length for instance *GC-1* is 15, while the tabu length chosen by RPP is 10. This is due to our choosing the smallest tabu length within 1 of the minimum MCC. Our method for dealing with clusters of algorithms was chosen solely because it gave good results on preliminary data. A better method would be to use statistical techniques to determine the clustered values. We are currently exploring this possibility.

While most of the time RPP with MCC is underestimating, there are are a few cases where RPP did not chose a tabu length that was close to the optimal length. An example of this is *Plan-5*, where the best tabu length is 10 and the chosen tabu length is 25. In these cases, the MCC values do not behave like those in Figure 2. Instead of having a group of contiguous tabu lengths diverging and a group of clustered contiguous tabu lengths, they are jumbled together. Further experimentation shows that this is due to noise in the sample. If one averages over a larger set of sample runs the MCC values behave like those shown in Figure 2. To correct this, one should continue to collect sample runs until the variance of the samples reduces to a desired level. We are currently exploring this technique.

Table 4 shows how using MCC as a secondary predictor compares against interleaving the execution of the algorithms. The first column shows the number of flips needed to test each of the eight different tabu lengths. The second column is the sum of the flips needed for testing and the average number of flips needed

[7] The results are averages of 10 runs where each run consists of 10 tries and each try consists of nx100 flips. The task is to predict among eight tabu lengths, 0, 5, 10, 15, 20, 25, 30, and 35.

Prob.	Inst.	Test flips	Total flips	Improvement over Interleaved
GC	1	170,000	3,043,056	230%
	2	48,000	60,409	-13%
	3	24,000	147,537	440%
	4	12,000	106,730	122%
	5	12,000	60,957	40%
Plan	1	6,480	9,425	150%
	2	12,480	31,647	182%
	3	17,440	44,518	255%
	4	17,440	29,495	227%
	5	17,440	242,560	-52%
Sat	1	16,000	78,288	443%
	2	12,000	311,293	-11%
	3	16,000	80,009	34%
	4	20,000	190,636	129%
	5	20,000	445,584	295%

Table 4. The cost of using *MCC* by testing tabu algorithms versus interleaving the algorithms.

by the chosen algorithm to solve the problem. The final column is the percent improvement over interleaving the algorithms. This number is eight times the average number of flips of the best algorithm.

Similar to the results in Table 2, using *RPP* with *MCC* most often provides a approach of finding a solution using on average fewer flips than interleaving the execution of the algorithms. The exceptions fall into two categories. Instances *Plan-5* and *Sat-2* do not compare favorably against interleaving primarily due to choosing a poor algorithm. This problem will be alleviated as we develop better methods of calculating and comparing *MCC* values. Instance *GC-2* is an example of the number of flips needed for testing overwhelming the number of flips needed by the best algorithm. Again we are investigating decision theoretic methods as a way of reducing the number of flips needed for testing.

5 Conclusions

In this paper we have argued the importance and difficulty of selecting a good heuristic algorithm for a given problem. There has been very little work on this subject. We have outlined an empirical approach that can be used to help predict which member of an algorithm family is best, so that we can quickly search for an appropriate algorithm. Obviously, this is only a beginning. Much more is possible, including empirical work extending and validating our *RPP* predictors and theoretical work to establish a sound foundation.

Acknowledgments: We would like to thank Bart Selman for his quick response to e-mailed questions and for providing GSAT.

References

1. P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of the 12 th IJCAI*, pages 331–337, Sydney, Australia, 1991. Morgan Kaufmann.
2. F. Glover. Tabu search - part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.
3. F. Glover. Tabu search - part II. *ORSA Journal on Computing*, 2:4–32, 1990.
4. M. Johnston and S. Minton. Analyzing a heuristic strategy for constraint-satisfaction and scheduling. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, pages 257–290. Morgan Kaufmann, 1994.
5. L. Kale. An almost perfect heuristic for the n nonattacking queens problem. *Inf. Process. Lett.*, 34:173–178, 1990.
6. D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.
7. V. Kumar. Algorithms for constraint-satisfaction problems: a survey. *AI Magazine*, 13(1):32–44, 1992.
8. S. Minton. Integrating heuristics for constraint satisfaction problems: A case study. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, San Jose, CA, 1993. AAAI Press.
9. S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conficts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161 – 205, 1992.
10. D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of AAAI-92*, pages 459–465, San Jose, CA, 1991. AAAI Press.
11. N. Muscettola. HSTS: Integrating planning and scheduling. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, pages 169–212. Morgan Kaufmann, 1994.
12. B. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.
13. V. Rao and V. Kumar. Superlinear speedup in state-space search. In *Conference on foundations of softwar technology and theoretical computer science*, 1988.
14. D. Sabin and E. Freuder. Constradicting conventional wisdom in constraint satisfaction. In A. Borning, editor, *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, Orcas Island, Washington, 1994. Springer-Verlag.
15. N. Sadeh. Look-ahead techniques for micro-opportunist job shop scheduling. Technical Report CMU-CS-91-102, School of Computer Science, Carnegie Mellon, 1991.
16. N. Sadeh. Micro-opportunistic scheduling: the micro-boss factory scheduler. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, pages 99–135. Morgan Kaufmann, 1994.
17. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pages 440–446, San Jose, CA, 1992. AAAI Press.
18. D. Smith. Transformational approach to scheduling. Technical Report KES.U.92.2, Kestrel Institute, 1992.