Plateaus and Plateau Search in Boolean Satisfiability Problems: When to Give Up Searching and Start Again

Steven Hampson and Dennis Kibler

Dept. of Information and Computer Science
University of California, Irvine, USA

Abstract: We empirically investigate the properties of the search space and the behavior of hill-climbing search for solving hard, random Boolean satisfiability problems. In these experiments it was frequently observed that rather than attempting to escape from plateaus by extensive search, it was better to completely restart from a new random initial state. The optimum point to terminate search and restart was determined empirically over a range of problem sizes and complexities. The growth rate of the optimum cutoff is faster than linear with the number of features, although the exact growth rate was not determined. Based on these empirical results, a simple run-time heuristic is proposed to determine when to give up searching a plateau and restart. This heuristic closely approximates the empirically determined optimum values over a range of problem sizes and complexities, and consequently allows the search algorithm to automatically adjust its strategy for each particular problem without prior knowledge of the problem's complexity.

1 Introduction

1.1 Generating Boolean Satisfiability Problems

Boolean Satisfiability (BS) problems arise in different forms and in a number of circumstances (e.g., theorem proving, planning, constraint satisfaction), but for empirical testing of satisfiability algorithms, randomly generated BS problems are usually created and represented in Conjunctive Normal Form (CNF). Specifically, given v Boolean variables, a CNF expression can be generated by creating c clauses, each with k randomly chosen variables or their negation. For fixed v and k, increasing c makes the CNF expression harder to satisfy, in the sense that there are fewer variable bindings that satisfy the expression. Likewise, for fixed v and c, decreasing k makes the expression harder to satisfy. However, the special case of k = 2 is decidable in polynomial time, so the hardest to satisfy CNF expressions that are not polynomially decidable are created with k = 3 (3-sat). Consequently 3-sat is used for most reported experiments, and that is the case for all results reported here.

Mitchell et al. (1992), empirically found that the most difficult CNF expressions were created when c/v = 4.30, and conjectured that the ratio approached 4.25 for large v. Below 4.25, the expression has increasingly many solutions and is correspondingly easier to satisfy. Above 4.25, the expression is increasingly likely to have no solutions and a contradiction can be found more easily. At 4.25, about half the randomly generated CNF expressions

are satisfiable, and few are sufficiently under or over-constrained to permit an easy answer. This property has been observed in a number of AI problems (Cheesman et al., 1991). All of the results presented here are for c/v = 4.25, although limited tests were performed by varying c around that value. Most results are for v = 64, since that was the largest value that could be extensively explored in reasonable time. However, some tests were made at v = 128, 256 and 512 in order to selectively test specific hypotheses. In each set of tests, a set of 1000 random functions are generated and the number of solved problems is reported for different algorithms or parameter settings. Reporting the average number of clauses satisfied would be a somewhat more sensitive measure of algorithm performance, but BS is generally concerned with complete satisfiability, so only those results will be reported here.

1.2 Hill Climbing and Optimization

Boolean satisfiability can be viewed as a maximization/optimization problem: given a CNF expression, find a variable setting that maximizes the number of clauses satisfied. However, unlike most optimization problems where the maximum possible value is not known beforehand, for BS, the maximum number of clauses that can be satisfied is simply the number of clauses in the CNF expression. Consequently the algorithm can immediately terminate if that maximum value is reached.

Function maximization is a topic of long-standing interest, and a number of techniques for finding function maxima have been developed over the years, many of which are discussed in *Numerical Recipes in C* (Press et al., 1988). As summarized there (pg. 290):

> An extremum (maximum or minimum point) can be either *global* (truly the highest or lowest function value) or *local* (the highest or lowest in a finite neighborhood...) Virtually nothing is know about finding global extrema in general. There are two standard heuristics that everyone uses: (i) find local extrema starting from widely varying starting values of the independent variables, and then pick the most extreme of these (if they are not all the same) or (ii) perturb a local extremum by taking a finite amplitude step away from it, and then see if your routine returns you to a better point, or "always" to the same one. There are tantalizing hints that so-called "annealing methods" may lead to important progress in the global extremization problem.

In all cases, hill-climbing from multiple start points forms the core of these methods. For BS, we have tried both straight hill-climbing using multiple random starts, and simulated annealing with multiple temperature fluctuations. Both work surprisingly well. However, our discussion in this paper will be limited to straight hill-climbing plus multiple restarts.

2 The Basic Algorithm: Greedy Hill-Climbing

The hill-climbing algorithm we have used for BS is as follows. It uses the basic generate-and-climb approach to optimization and is similar to algorithms proposed by Selman and Gu and Walsh, with two major differences. First, hill-climbing is done in a greedy manner, where any move that improves the state is accepted, without checking to see if a larger increase could be achieved. Second, search is limited by the number of sideways moves at a plateau, rather than the total moves in a hill-climbing episode. The search process simply cycles through the variables, always moving up when possible and moving sideways with 50:50 probability. (Probabilities other than 50:50 will be considered briefly in section 7.) Empirically, this simple algorithm has proved surprisingly effective, and in terms of problems solved to states generated, performs as well methods which consider all possible transitions from a state. However, our goal is not to compare this particular algorithm to other algorithms, but to develop a general mechanism for automatically deciding when to give up searching a plateau and restart, a process that should be applicable to a range of algorithms.

The algorithm uses a single, global current-state description. Cover_count is the number of clauses in the CNF expression that are satisfied by the current setting of the variables. Clause_num is the total number of clauses in the CNF expression. Max_start bounds the number of restarts. Max_moves limits the number of moves on a plateau.

```
SOLVE_BS
for start = 1 to max_start
    begin
    set random start state
    call hill_climb
    if hill_climb was successful, return(success)
    end
return(fail)
HILL_CLIMB
moves = 0
for cycles = 1 to forever
    for i = 1 to v
        begin
        if flip(i) decreases cover_count, then do nothing
        if flip(i) increases cover_count, then
            begin
            flip(i)
            moves = 0
            if (cover_count = clause_num) return(success)
            end
        if flip(i) wouldn't change cover_count, then with 50:50 probability
            begin
            flip(i)
            moves = moves + 1
```

```
        if moves> max_moves, return(fail)
        end
    end
```

If Hill_climb reaches a state where no improvement or sideways moves are possible, it is detected and failure is returned.

To summarize the process, the function Solve_bs generates random start states and calls the function Hill_climb on each of them until either the problem is solved or the maximum number of restarts is reached. The function Hill_climb approaches the over-all hill-climbing problem as a series of plateau searches. To do this, it continually cycles through the features, checking the resulting change in cover_count if the value of that feature were flipped. If it finds a feature flip that improves cover_count, it flips the feature and resets moves to zero. If cover_count ever equals the number of clauses in the CNF expression, Hill_climb returns with success. If flipping the value of a feature leaves cover_count unchanged, then with 50:50 probability the feature is flipped and moves is increased by one. If Hill_climb fails to find an improvement in cover_count within the alloted number of sideways moves per plateau, or is stuck at a single-state local maximum, it returns with failure.

The two adjustable parameters in the algorithm are the maximum number of starts (max_start) and the maximum number of sideways moves per plateau (max_moves). For brevity, these will be refered to as s (starts) and n (n referring to the maximum number of nodes expanded on a plateau). The number of nodes expanded is counted as the number of moves made.

Note however that the product s*n is not an exact bound on the *total* number of nodes expanded to solve a problem, since a number of plateaus have to be searched for each start. However, as a general rule, only plateaus with near maximum values of cover_count are searched to anywhere near the node limit. Consequently, since the emphasis of the work presented here is on methods of plateau search, results will be presented in terms of the algorithm parameters s and n, with the added assumption that it is desirable to minimize their product.

3 Search Space Characteristics

The most obvious problem of hill-climbing techniques is that they can get stuck at local maxima. Various techniques are available to ameliorate this problem but, as previously observed, one standard method is simply to restart the process at a new, random location whenever a local maximum is reached. This is straightforward if local maxima are easily detectable, e.g., when all one-step transitions from the current state reduce the evaluation function. However, in many discretely-valued optimization problems, plateaus exist, where a number of states of equal evaluation are interconnected by one-step transitions. If none of the states can be improved by single-step transitions, the entire plateau is a local maximum. However, some of the states on the plateau may permit improvement. For example, in the

n-queens problem for $n > 100$, Minton et al. (1990) observed that all plateaus contain improvable states, so that restarts are unnecessary. However, for other problems such as BS, plateaus exist such that no state on it can be improved by a single-feature flip. Thus, unlike the n-queens problem, perseverance in plateau search is not guaranteed to yield an improvement and the question of when to give up on a plateau and restart must be addressed.

In order to better understand the observed behavior of BS hill-climbing, the general nature of the search space was empirically investigated, and a number of regularities were observed. Most of these results relate to the size of plateaus and the density of improvable states on them (refered to as *solutions* to the plateau). These values were determined by exhaustive plateau search techniques (e.g., breadth first). Specifically, plateaus were exhaustively searched until all states on the plateau had been explored, or a node limit of 100,000 was reached. In all cases c/v = 4.25.

First, it was observed that the plateaus increase in size with increasing depth (decreasing cover_count), down to the average coverage created by random variable settings, which is c*7/8 for 3-sat. However, the density of improvable states in the plateaus also increases rapidly with depth, so that very little search is needed to find an improvement except for the top few plateaus. Consequently, overall problem complexity for a single hill-climbing episode can be expected to be dominated by the complexity of searching the top few plateaus. This is consistent with the observation that hill-climbing is initially quite rapid, but then spends long periods moving sideways on the final plateau(s).

A second, and more significant result was that the size of the top few plateaus appears to grow exponentially with the number of features while the density of solutions on them decreases inversely with the number of features. The inverse relationship of solution density with number of features means that the expected search time for the "average" top plateaus would increase approximately linearly with the number of features if states on the plateaus could be chosen at random. Depending of the distribution and density of the solutions on the plateaus that contributed to the overall average density, and the bias in the search order, actual plateau search results could be quite different, but as will be seen, for $v < 100$, setting the cutoff, n, to the number of features is a reasonable heuristic.

Finally, the average number of restarts needed to solve problems also grows with the number of features. This was tested by choosing a sufficiently large n so that increasing it did not increase the number of problems solved, and performing a *single* hill-climb (s = 1) for each problem in the set. Under these conditions, the probability that a solvable problem will be solved on a single start appears to decrease no faster than inversely with v. Based on this observation, it might be expected that the number of restarts necessary to solve the average problem would increase no more than linearly with the number of features, but by other measures (e.g., the number of starts necessary to solve half the solvable problems), the rate appears to be faster than linear. The ratio of global to local maxima, as modified by the size of their basins of attraction, is clearly important in the

behavior of the algorithm, but the best method for measuring or reporting this property is unclear.

Thus the expected time to solve a problem by hill-climbing is affected by a number of separable factors. First, if states on a plateau could be chosen at random, the expected time to search a plateau would go up roughly linearly with the number of features. Second, the number of plateaus to be searched in a single hill-climbing episode (those between (7/8)c and c) also grows linearly. Multi-step improvements sometimes occur, but not frequently enough to effect the linear growth in the number of plateaus searched. As long as a non-decreasing fraction of the plateaus are searched to near the limit, this suggests that the expected search time of a successful hill-climbing episode is roughly quadratic. In fact, empirical results suggest that the growth rate is somewhat greater than quadratic, so these assumptions may not be completely justified. Finally, the number of hill-climbs necessary to solve a problem also increase at least linearly with the number of features, although the actual growth rate is unknown.

These results are at best rough approximation, but give a qualitative feel for the properties of the search space and the resulting behavior of the algorithm. Over the tested range (32-256 features) the reported growth rates are reasonable approximations, but for arbitrarily large v, the true growth rates might be quite different (e.g., exponential vs. linear or quadratic).

4 Restarts versus Plateau Search

For reasonably sized BS problems ($v > 50$), the plateaus are generally too big to be searched exhaustively, so, in general, it is impossible to determine if a plateau is actually a true local maximum, or simply has a very low density of solutions. Consequently, the question becomes when to give up searching an empirically unproductive plateau and restart. Without a detailed knowledge of the properties of the search space, this would be a difficult question to answer analytically. Fortunately, the empirical results are relatively straightforward. This can be seen in Table 1, which considers a range of values of starts (s) and nodes per plateau (n) in powers of 2. In most cases more problems can be solved by doubling s than by doubling n. The only exceptions to this are for small search bounds that are less than the number of variables. Since a goal is to solve BS problems in the shortest amount of time, this translates to solving them with the smallest s*n product. For example, the maximum number of problems solved (532) is achieved with a minimum s*n product of 16,384. This is achieved at (s = 64, n = 256), and again at (256, 64), (512, 32) and (1024, 16). Extending the s*n = 16,383 diagonal from (64, 256) to the lower left (decreasing s, increasing n), the number of problems solved are 521, 507, 473, 419, 325, and 233, reflecting an increasingly inefficient partitioning of resources between plateau search and restarts.

	Starts											
Nodes	1	2	4	8	16	32	64	128	256	512	1024	2048
8	15	25	48	80	132	211	296	398	453	494	524	530
16	35*	59	95	163	251*	361	440	491	517	526*	532*	532
32	58	101*	169*	253*	343	447*	502	520	527*	532	532	532
64	96	146	236	324	419	488	524*	530*	532	532	532	532
128	131	195	303	388	465	509	527	531	532	532	532	532
256	155	232	343	423	487	517	532	532	532	532	532	
512	185	273	384	458	502	521	532	532	532	532		
1024	207	299	399	472	507	524	532	532	532			
2048	218	314	414	473	508	525	532	532				
4096	223	322	419	475	512	526	532					
8192	228	325	421	476	512	526						
16384	233	330	422	476	512	526						
32768	237	332	423	476	512							
65536	237	332	423									

Table 1. Number of problems solved with increasing restarts and search limits. 64 features, 272 clauses, 1000 problems. * = smallest n for a given s for which doubling s is as good or better than doubling n.

Over a wide range of restarts, the optimum amount of plateau search (the point at which doubling s becomes better than doubling n) is relatively constant at about n = 32, or half the number of features. Alternatively, for diagonals of equal s*n product, the maximum number of problems solved occurs at around n = 32. For larger numbers of features the optimum value increases; at v = 128, the optimum is approximately equal to v, and by v = 256, the optimum is two to four times the number of features. However, for problems of up to about one hundred features, an easy and reasonable rule of thumb is simply to fix n at the number of variables, and devote all further computational resources to restarts. Using a slightly different algorithm and a different testing technique, Gent and Walsh (1993) obtained similar values for the optimum search cutoff, and speculate that the growth rate is quadratic with the number of features. Various experiments we have run suggest that the growth rate of the optimum n is greater than quadratic, but at present the data is insufficient to conclude the true growth rate.

5 Adaptive Cutoff

Given that the empirical results for the optimum search cutoff values are reasonably robust across a range of algorithm variations and testing techniques, an interesting question is *why* the particular relationship holds. It is possible to construct a simplified model that helps explain this restart/search tradeoff, and suggests a method of automatically adjusting n for each problem. First, assume there are an infinite number of plateaus with differing densities of solutions and with an over-all average solution density of D. The simplified problem is to choose plateaus and states at random until a solution is found. One strategy would be to choose a plateau, and then continue to choose states on it until $(1-D)^x < .5$, where x is the number of states chosen on that particular plateau. That is, when empirically the plateau appears to be worse than average, choose a new plateau. This is a reasonable strategy, but unfortunately, the average density of solutions is not apt to be known. However, an even simpler solution is possible: choose a new plateau after *every* state selection. This is certainly better than the other extreme of choosing a single plateau and sticking with it, and is in fact the optimal strategy for this simplified problem. Roughly speaking, once a number of states have been expanded on a plateau without improvement, its expected density of solutions is somewhat lower than other plateaus that have not been explored at all. Consequently, once *any* states have been explored, it is marginally better to switch to a new plateau.

This simplified model gives some intuition for the surprisingly low optimum cutoff values observed, but is obviously not an exact model of BS problems. In actuality, there is some cost in "choosing a plateau" since the algorithm must hill-climb back to the previously achieved level from a random restart state, and in fact may fail to reach the same level. This cost in choosing a plateau means that there is some value in sticking with a particular plateau for a while before restarting: the greater the expected cost of reclimbing to a plateau at the current level, the longer you should search the current plateau before giving up. The measured cost to reach the current level provides a reasonable estimation of the expected cost to reclimb a comparable plateau. Just what the exact function of the expected cost should be remains to be determined, but this perspective suggests a method of dynamically adjusting the algorithm to the problem at hand. In the absence of any compelling argument for a particular function, we tested the simplest rule:

Search Restart Heuristic:

> *Quit searching a plateau when the number of nodes expanded on the plateau becomes greater than the number of nodes expanded in the search from the initial state to the plateau.*

A lower bound on the cutoff value was also necessary to get things started, since the cost to the initial plateau is zero, and a lucky one feature flip on the initial plateau would produce a one-node limit for the next. An intentionally small lower bound of v/4 was

used during testing, but a somewhat larger lower bound of v is probably a better choice to optimize performance. If every plateau were explored to its limit before an improvement was found, this would produce an exponentially growing series of search bounds during each hill-climbing episode, so that the final cutoff value would be exponential with the number of features. However, in practice, very few plateaus are searched to anywhere near the limit, and the resulting cutoff for the final plateau was generally near the optimum value indicated by Table 1. In fact, this self-tuning technique gave results comparable to the empirically determined optimum values of n at v = 64, 128, and 256. For 512 features, this heuristic suggests that the optimum cutoff is around 9000, although we have not yet verified that value by exhaustive empirical testing. (To be precise, the optimum value of n is best approximated if only the total *sideways* moves are counted, but the difference is only noticeable for small v, and for practical application the difference is unimportant since there is little penalty in slightly overestimating n.)

In limited testing, the heuristic also gave empirically correct results for c values above and below 4.25*v. In general, the optimum n does not change greatly with c when empirically measured, and neither does the automatic adjustment, although it does appear to increase and decrease slightly with c. In addition, the required s was not noticeably affected. That is, the point at which increasing s no longer increased the number of solved problems stayed the same, although the maximum number of solved problems changed. (Obviously as c approaches 0, both n and s approach 1, but in the range 4.0 to 4.5, they were relatively constant.) So, in general, although the percentage of solvable problems is strongly affected by relatively small changes in c, the difficulty of solving the resulting problems, as reflected in s and n, was not greatly affected.

In another short set of experiments, the individual clauses were assigned random integer weights which determined their relative importance. (In "normal" testing, they are all of the same importance.) This was done to more closely approximate the optimization tasks we had previously worked on, which had varying levels of importance attached to different constrains. For BS, this was found to make the problems much harder to solve, although it does not change the number of solvable problems. Again the optimum n was not significantly changed as determined either by empirical measurement or by automatic adjustment. However, in this case, the necessary s did increase with the size of the random weights.

Thus, it appears that the algorithm parameter n can be made appropriately self-adjusting to a range of circumstances by a very simple mechanism. The self-adjusting mechanism changes n when empirically, the optimum n changes, and leaves n unchanged under conditions which impact the algorithm in various ways, but empirically do not effect the optimum n. Automatic adjustment should be of some value when the complexity of individual problems is not known *a priori.*

A slightly more complex technique for adjusting n was also tried, in which the cutoff for a plateau at level m was the average number of nodes expanded over all climbs to level

m for prior restarts on the current problem. This kept the average amount of search the same, but reduced the variance from start to start. The general idea was based on the observation that successful hill-climbs (those that successfully solve the entire problem) are generally comparatively quick, so that a start that is still making progress, but is taking more than average time to reach each plateau, will probably not be ultimately successful. However, the results of this averaging method did not appear significantly different than the original, non-averaging method.

As a point of comparison for other algorithms, the adaptive cutoff version of the algorithm solved all of the 8-feature compressed parity problems (around 70 features in CNF) in the Dimacs benchmark collection using between 4,000 and 60,000 node expansions per problem. However, it failed on the first of the 16-feature compressed parity problems.

6 Variants and Extensions

The "adaptive clause weighting" technique of Selman and Kautz (93) was implemented and tested. With that technique, clauses are weighted by the number of times they remain unsolved after each hill-climbing episode. For certain hard problems it was quite effective, but it did not improve performance on randomly generated problems.

The program was also found to be rather insensitive to the 50:50 probability of sideways moves, and could be varied between 30:70 and 70:30 with little effect of performance. Consequently, no attempt was made to "tune" that value between the extremes (100:0 = never take a sideways move, 0:100 = always take a sideways move). At 100:0, performance is very poor, while at 0:100 performance is quite good, although a small amount of probabilistic choice (e.g., 10:90) appears to be slightly better. With 0:100, the algorithm is reasonably close to the HSAT algorithm of Gent and Walsh (1993), in which of all the potential flips, the feature which was flipped longest ago is chosen. They experimented with a number of related algorithms and found HSAT to be the most effective.

As previously observed, the most significant difference between this algorithm other hill-climbing BS algorithms is that all of the children of a node are not generated before a move is chosen. Thus it gains in efficiency in the number of states that must be considered before a move is made, but does not guarantee steepest ascent at each point. We tested both approaches, and generally we found the "feature cycling" algorithm to be more efficient in terms of problems solved per nodes generated. Its extreme simplicity also has a certain appeal. Consequently that is the approach we have used in all of our studies. In future work we plan to try the adaptive cutoff mechanism in conjunction with other hill-climbing algorithms.

7 Discussion

Many optimization problems can be effectively addressed by hill-climbing from multiple start states. However, if the function to be optimized is limited to discrete values, plateaus often exist, and most of the total run time can be spent searching those plateaus. This is the case for Boolean satisfiability. In this paper, we consider a simple feature-cycling search of plateaus. The feature-cycling algorithm was chosen primarily for its simplicity and as a comparison point for more sophisticated search techniques, but surprisingly, turned out to be the most efficient algorithm we tested. This point is discussed in more detail in a companion paper (Hampson and Kibler, submitted).

The first set of experiments was designed to roughly characterize the search space properties and the corresponding behavior of the search algorithm. Three points were considered: the expected time to successfully search a plateau, the expected number of plateaus searched per start, and the expected number of starts required. The density of solutions on the top few plateaus goes down as the inverse of the number of features, so search time per plateau might be expected to go up linearly. The number of plateaus search per start is roughly linear, so expected search time per start should be roughly quadratic. Empirically, this prediction appears to be reasonable, but may be on the low side. The number of starts required grows at least linearly, but was not adequately characterized.

The primary set of experiments was designed to explore the tradeoff between the amount of plateau search and the number of restarts. That is, given a limited amount of processing time, what is the best allocation of time between completeness of individual hill-climbing episodes and the number of restarts. The results were straightforward: the optimum amount of plateau search increases with the number of features. Empirically, the growth rate is at least quadratic, but up to about 100 hundred features the plateau search limit can be reasonably approximated as the number of features. This was determined empirically by measuring a wide range of restarts, search cutoffs, and number of variables.

The optimum amount of time to spend exploring a plateau is a function of the amount of work needed to reach a comparable plateau after a restart. Thisamount is not known before hand, but can be reasonably estimated as the cost to reach the *current* plateau. This raises the possibility that the plateau search cutoff can be adjusted dynamically during individual problem-solving episodes rather than empirically pre-determined by exhaustive testing. As an initial attempt at automatic tuning of the cutoff parameter, a simple plateau search heuristic was tested: perseverance in plateau search was made exactly equal the amount of work expended in reaching the plateau. That is, for any plateau, n should be equal to the total number of nodes expanded while climbing to that level. This particular strategy was implemented more for simplicity than any other reason, but gave surprisingly good results, closely approximating the empirical optimum n values measured at 64, 128 and 256 features.

In limited testing the heuristic also approximated the empirical optimum as c was

varied above and below 4.25 * v, although somewhat surprisingly neither changed very much while the percent of solvable problems was strongly affected. The required s was also not noticeably affected. That is, while small changes in c significantly affected the percent of solvable problems, it did not greatly affect the difficulty of solving the solvable problems. Conversely, random weighting of clauses does not affect the percent of solvable problems, but greatly increased the difficulty of solving them. Again, this did not noticeably effect the optimum n. However, the required s did increase with the size of the random weights. It is not surprising that the optimum n is affected by the number of features, but it was unexpected that it was so insensitive to other factors that strongly affected the overall "difficulty" of the problems.

The general result of the adaptive cutoff experiments is that while it is possible to premeasure a reasonable cutoff for particular classes of problems, it is also possible to make the cutoff more sensitive to the specific problem at hand, and in fact to the current state of problem solving for that problem. In a sense, the algorithm can be viewed as a best-first search, where the cost to be minimized in the remaining cost to solution, not the total cost of solution. Consequently, a restart is justified when the expected cost to solution (which may be infinite) for the current state becomes greater than the expected cost to solution from a random restart state. The expected cost to solution is always problematic in any heuristic search technique, but the expected cost of restarting and climbing to an equivalent plateau can be reasonably estimated as the cost to the current plateau.

References

Cheesman, P., Kanefsky, B., Taylor, W. M. (1991): Where the really hard problems are. In: Proceedings IJCAI-91 163–169.

Gent, I. P., Walsh, T, (1993): Towards an understanding of hill-climbing procedures. In: Proceedings of Eleventh NCAI, 28–34.

Gu, J. (1992): Efficient local search for very large-scale satisfiability problems. Sigart Bulletin 3, 8–12

Minton, S., Johnston, M. D., Philips, A. B., Laird, P. (1990): Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In: Proceedings AAAI-90, 17–24.

Mitchell, D., Selman, B., Levesque, H. (1992): Hard and easy distributions of SAT problems. In: Proceedings AAAI-92, 459–465.

Press, W. H., Flannery, B. P., Teukulsky, S. A., Vetterling, W. T. (1988): Numerical Recipes in C. Cambridge Univ. Press, New York.

Selman, B., Kautz, H.(1993) An Empirical Study of Greedy Local Search for Satisfiability Testing. In: Proceedings NCAI-93, 46–51.

Selman, B., Levesque, H., Mitchell, D. (1992): A new method for solving hard satisfiability problems. In: Proceedings AAAI-92, 440–446.