The Breakout Method For Escaping From Local Minima

Paul Morris
IntelliCorp
1975 El Camino Real West
Mountain View, CA 94040
morris@intellicorp.com

Abstract

A number of algorithms have recently been proposed that use *iterative improvement* (a form of hill-climbing) to solve constraint satisfaction problems. These techniques have had dramatic success on certain problems. However, one factor limiting their wider application is the possibility of getting stuck at non-solution local minima. In this paper we describe an iterative improvement algorithm, called *Breakout*, that can escape from local minima. We present empirical evidence that this method is very effective in cases where previous approaches have difficulty. Although Breakout is not theoretically complete, in practice it appears to almost always find solutions for solvable problems. We prove that an idealized (but less efficient) version of the algorithm is complete.

Introduction

Several recent papers have studied iterative improvement methods for solving constraint satisfaction and optimization problems. (See [Minton *et al.* 1990], [Zweben 1990], [Sosic & Gu 1991], [Minton *et al.* 1992], [Selman, Levesque, & Mitchell 1992].) These methods work by first generating an initial, flawed "solution" (i.e., containing constraint violations) to a problem. They then try to eliminate the flaws by making local changes that reduce the total number of constraint violations. Thus, they perform hill-climbing in a space where goodness is measured in terms of how few constraints are violated, in the hope that eventually a point will be reached that provides an acceptable solution to the problem. The papers provide empirical and analytical evidence that such methods can lead to rapid solutions for important classes of problems.

One drawback of such methods, however, is the possibility of becoming stuck at locally optimal points that are not acceptable as solutions. (We will henceforth call these "local minima," viewing the local changes as movements on a *cost surface* where the height reflects the current number of constraint violations.) While the above approaches incorporate some techniques to mitigate this problem, these are at best only moderately successful. For example, the Minton *et al.* and Selman *et al.* algorithms can escape from plateaus on the cost surface, since they allow random "sideways" local changes. However, they still get caught in other local minima. This causes them to miss solutions in many difficult SAT and K-coloring problems. While the random walk character of the Zweben algorithm would seem to ensure almost certain eventual movement to a solution, this kind of probabilistic guarantee may not be very useful.[1] The Sosic and Gu approach formulates the search space in a way that avoids local minima. However, the method is specific to N-queens, and has no obvious generalization to other problems.

Another remedy for the local minimum problem is to repeatedly restart the iterative improvement process from new random starting points until an acceptable solution is reached, as is done in the Selman *et al.* algorithm. This amounts to randomly searching the local minima for a solution. Figure 1 illustrates why this is computationally impractical in many cases. Iterative improvement methods derive their power from an assumption that the number of constraint violations is a rough indicator of the closeness to a solution. In general, we might expect some noise in the estimate, suggesting a cost surface with a cross-section something like that shown in the upper part of the figure. (The lowest point in the surface represents a solution.) Now consider an algorithm that gets stuck at each of the local minima shown. Notice that repeated restarting will perform no better on the upper surface than it would on the lower surface shown in the figure. That is, it fails to take advantage of the overall *trend* of the surface. Looking only at the lower surface, it is easy to see that the average time to a solution depends on the number of local minima in a region around the solution (and thus indirectly on the "volume" of the

[1] As an analogy, if two flasks are connected by a tube, the air molecules will, with probability 1, eventually all pile up in one flask. However, the mean time before this happens is enormous.

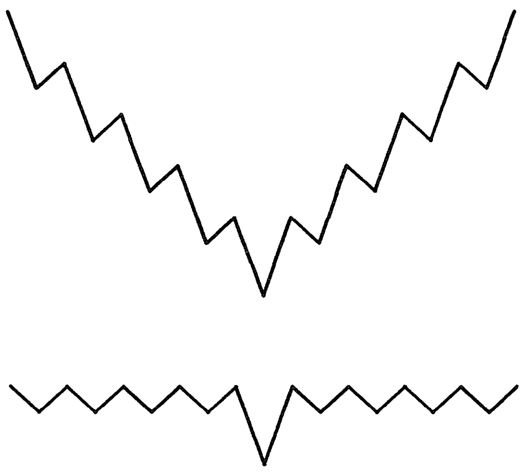

Figure 1: Why escaping is better than restarting.

region). For CSPs, the dimension of the cost surface (i.e., the number of states adjacent to a given state) increases linearly with the size of the problem. This makes the "volume" (and hence, presumably, the expense of a restart search) increase rapidly with the size. Now consider an algorithm that can escape from the local minima shown. This should perform better on the upper surface, since it takes advantage of the general trend towards a solution. For this type of surface, the time required to find a solution should be largely independent of the dimensionality.

It is known that for certain problems, like N-queens, almost all the local minima are narrow plateaus (Morris [1992]). The above analysis suggests that plateau-escaping algorithms (like that of Minton *et al.*) would solve such problems very efficiently. Indeed, this does appear to be the case.

In this paper we present a deterministic algorithm for solving finite constraint satisfaction problems using an iterative improvement method. The algorithm includes a technique called *breakout* for escaping from local minima. In the following sections, we will define the algorithm, and compare its performance to that of other methods. Finally, we will prove a theoretical result that helps explain the success of the algorithm.

The Breakout Algorithm

We now discuss the Breakout algorithm in more detail and consider how it applies to a constraint satisfaction problem (CSP). The essential features of this algorithm were first introduced in Morris [1990], where it was applied to the Zebra problem (see Dechter [1990]).[2]

[2]Selman and Kautz [1993] have independently developed a closely related method.

Informally, a CSP consists of a set of variables, each of which is assigned a value from a set called the *domain* of the variable. A *state* is a complete set of assignments for the variables. The solution states must satisfy a set of *constraints*, which mandate relationships between the values of different variables. We refer the reader to Dechter [1990] for the formal definition of a CSP. In this paper we will consider only *finite* CSPs, i.e., where there is a finite set of variables, and the domain of each variable is finite. Constraint satisfaction problems are generally expressed in terms of sets of tuples that are allowed. For our purposes, it is convenient to instead focus attention on the *nogoods*, i.e., the tuples that are prohibited.

The intuition for the breakout algorithm comes from a physical force metaphor. We think of the variables as repelling each other from values that conflict. The variables move (i.e., are reassigned) under the influence of these forces until they reach a position of equilibrium. This corresponds to a state where each variable is at a value that is repelled the least by the current values of the other variables. In physical terms, an equilibrium state is surrounded by a potential barrier that prevents further movement. If this is not a solution, we need some way of breaking through that barrier to reach a state of lower energy.

In an equilibrium state, the variables that are still in conflict are stable because they are repelled from alternative values at least as much as from their current values. Suppose, however, the repulsive force associated with the current nogoods is boosted relative to the other nogoods. As the repulsive force on current values increases, at some point, for some variable, it will exceed that applied against the alternative values. Then the variable will no longer be stable, and the iterative improvement procedure can continue. The boosting process has effectively changed the topography of the cost surface so that the current state is no longer a local minimum. We refer to this as a *breakout* from the equilibrium position.

In iterative improvement, the cost of a state is measured as the number of constraints that it violates, i.e., the number of nogoods that are matched by the state. For the Breakout algorithm, we associate a weight with each nogood, and measure the cost as the sum of the weights of the matched nogoods. The weights have 1 as their initial value. Iterative improvement proceeds as usual until an equilibrium state (i.e., a local minimum) is reached.[3] At that point, the weight of each current nogood is increased (by unit increments) until breakout occurs. Then iterative improvement resumes. Figure 2 summarizes the algorithm.

We note that the algorithm does not specify the initial state. In our experiments we used random start-

[3]Plateau points are treated just like any other local minima. Thus, the algorithm relies on breakouts to move it along plateaus.

```
UNTIL current state is solution DO
IF current state is not a local minimum
THEN make any local change that reduces the total
    cost
ELSE increase weights of all current nogoods
END
```

Figure 2: The Breakout Algorithm.

ing points. The results of Minton *et al.* suggest that it may be worthwhile to use a greedy preproccessing algorithm to produce an initial point with few violations. This will generally shorten the time required to reach the first local minimum.

The algorithm also does not specify which local change to make in a non-equilibrium state. In our implementation, we simply use the first one found in a left-to-right search.

Experimental Results

We tested the breakout algorithm on several types of CSP, including Boolean 3-Satisfiability, and Graph K-Coloring. We describe the results here.

Boolean Satisfiability

For Boolean 3-satisfiability, we generated random *solvable* formulas. The results we report here are for 3-SAT problems where the *clause density*[4] has the critical value of 4.3 that has been identified as particularly difficult by Mitchell *et al.* [1992]. We use the same method of generating random problems as theirs except for the following: to ensure the problems are solvable, we select a desired solution[5] in advance and modify the generation process so that at least one literal of every clause matches it. Specifically, we reject (and replace) any clauses that would exclude the desired solution.[6] The variables are then initialized to random values before starting the breakout solution procedure. Note that for Boolean satisfiability problems, the nogoods are just the clauses expressed negatively.

The results are shown in table 1 for a range of problem sizes, averaged over 100 trials in each case. We wish to emphasize that the algorithm never failed to find a solution in any of the trials. Note that the growth of total hill-climbing (HC) steps appears to be a little faster than linear, but less than quadratic. The timing figures shown are the average elapsed time for

Vars	Breakouts	HC steps	Time (sec.)
100	60	168	3.2
300	180	795	17.7
500	320	1678	46.9
700	325	2165	68.0
900	475	3215	122.2
1100	575	4508	182.9

Table 1: Breakout on 3-SAT problems with prearranged solution.

Vars	Tries	Total Flips	Time (sec.)
100	3.6	1505	10.4
150	4.2	4093	38.2
200	6.5	11,901	138.7
250	4.3	11,789	166.6
300	6.9	28,782	473.9

Table 2: GSAT on similar problems.

the (combined) creation and solution of 3-SAT problems, running in Lisp on a Sun 4/110.

A recent paper [Williams & Hogg 1993] has noted that using a prespecified solution when generating random problems introduces a subtle bias in favor of problems with a greater number of solutions, and thus is likely to produce easier problems. (On the other hand, it is a convenient way of producing large known-solvable problems, which are otherwise difficult to obtain.) Selman *et al.* [1992] use a different random generation process for testing their GSAT algorithm. First they generate formulas that may or may not be satisfiable. Unsolvable formulas are then filtered out by an exhaustive search (using a variant of the Davis-Putnam algorithm). Thus, the results in table 1 cannot be directly compared to those reported for GSAT. To obtain a better comparison, we reimplemented GSAT and tested it on problems generated in the same way as those used for testing Breakout. The results are shown in table 2 (averaged over 100 trials). The Tries parameter here is the number of restarts (including the final successful one). The Total Flips parameter is the number of local changes needed to reach a solution (summed over all the Tries). This figure is roughly comparable to the number of hill-climbing steps for the Breakout algorithm. In GSAT, each Try phase is limited to *Max-Flips* steps; for these experiments, Max-Flips was set to $n^2/20$, where n is the number of variables.[7]

In terms of the machine-independent parameters,[8] the problems we use are clearly easier for GSAT than those on which it was originally tested. In particular, the Tries figure appears to stay roughly constant

[4]The number of clauses divided by the number of variables.

[5]By symmetry, it doesn't matter which one.

[6]In an earlier version of this paper, instead of using a rejection method, the negated/unnegated status of one literal was directly chosen to match the solution. That method produced very easy problems.

[7]An $O(n^2)$ setting was suggested by Bart Selman (personal communication).

[8]The sun 4/110 is slower than the MIPS machine used by Selman *et al.*

Density	3	4	5	6	7
Breakouts	6.4	36	74	23	8.1
HC steps	41	115	248	143	99

Table 3: Breakout for different clause densities

as the problem size increases. Nevertheless, the performance of Breakout seems significantly better than that of GSAT, and avoids the inconvenience of having to set the Max-Flips parameter. We remark that testing with a variant of the Davis-Putnam algorithm shows exponential growth for these problems.

We also ran Breakout for different values of the clause density, keeping the number of variables fixed at 100. Instead of a peak centered at 4.3, we found one in the neighbourhood of 5. (Testing at higher resolution indicates a range from 4.8 to 5.2 as the region of greatest difficulty.) The results shown in table 3 are each averaged over 1,000 trials.

In preliminary testing of Breakout on problems generated in the same way as those in Selman *et al.* [1992], average performance appears to degrade to exponential, like that of GSAT, and the peak difficulty occurs at the 4.3 value of the density. Remarkably, the average appears dominated by a small number of very difficult problems. The average over 100 trials has been observed to fluctuate by an order of magnitude, depending on how many of the very difficult problems are encountered. These problems may have cost surfaces that more closely resemble the lower surface in figure 1.

Graph Coloring

For graph K-coloring, we generated random solvable K-coloring problems with n nodes and m arcs in essentially the same way as described in Minton *et al.* [1992] (and attributed there to Adorph and Johnson [1990]). That is, we choose a coloring in advance that divides the K colors as equally as possible between the nodes. Then we generate random arcs, rejecting any that violate the desired coloring, until m arcs have been accepted. The entire process is repeated until a connected graph is obtained.

We used two sets of test data, one with $K = 3$ and $m = 2n$, and the other with $K = 4$ and $m = 4.7n$. The first set are the "sparse" problems for which Minton *et al.* report poor performance of their Min-Conflicts hill-climbing (MCHC) algorithm. The second represents a critical value of the arc density identified by Cheeseman *et al.* [1991] as producing particularly difficult problems.

Table 4 shows the results on the first set of test data. Each figure is averaged over 100 trials. The algorithm never failed to find a solution on any of the trials. We note the number of breakouts seems to increase roughly linearly (with some fluctuation). The number of transitions per breakout also appears to be slowly

Nodes	30	60	90	120	150	180
Breakout Algorithm:						
Breakouts	12	24	48	46	54	80
HC steps	35	77	159	191	245	363
MCHC:						
% solved	46.5	12.13	1.75	0.5	0.25	0.12

Table 4: Breakout and MCHC on 3-coloring

Nodes	30	60	90	120	150
Breakouts	8	47	189	655	1390
HC steps	49	308	1257	3959	8873

Table 5: Breakout on 4-coloring

growing. This performance can be contrasted with that of MCHC, which shows an apparent exponential decline in the frequency with which solvable sparse 3-coloring problems are solved (within a bound of $9n$ steps), as the number of nodes increases.[9]

Table 5 shows the results on the second set of test data. In this case, each figure is averaged over 100 trials, except for N = 120 and 150, which were averaged over only 99 trials each. This really does seem to be a more difficult task for Breakout. For the omitted trials, the algorithm failed to reach a solution even after 100,000 breakouts, and was terminated. This may be a further instance of the phenomenon of a small number of very difficult problems sprinkled among the majority of easier problems.

A Complete Algorithm

The experimental results show that Breakout has remarkable success on important classes of CSPs. This is partially explained by the discussion regarding figure 1. However, one point has not yet been answered. Since Breakout modifies the cost function, it appears plausible that it could often get trapped in infinite loops; yet the experimental data shows this almost never occurs (at least, for randomly generated problems). In this section, we provide some insight into this by showing that a closely related algorithm is theoretically *complete*; that is, it is guaranteed to eventually find a solution if one exists.

Consider the effect of a breakout on the cost surface: the cost of the current state, and perhaps several neighbouring states (that share nogoods with the current state), is increased. However, all that is really needed to escape the local minimum is that the cost of the *current* state itself increase. We are thus led to consider an idealized version of Breakout where that is the only state whose cost changes. To facilitate this, we assume every state has a stored cost associated with it that can be modified directly. (The initial costs would be the same as before.) This ideal-

[9] We thank Andy Philips for providing this data.

```
UNTIL current state is solution DO
IF current state is not a local minimum
THEN make any local change that reduces the cost
ELSE increase stored cost of current state
END
```

Figure 3: The Fill Algorithm.

ized algorithm is summarized in figure 3. We will call this the *Fill* algorithm because it tends to smoothly fill depressions in the cost surface.

It turns out that this idealized version of Breakout is complete, as we prove here.[10] In the following, we say two states are *adjacent* if they differ in the value of a single variable, a state is *visited* when it occurs as the current state during the course of the algorithm, and a state is *lifted* when its stored cost is incremented as a result of the action of the algorithm. Note that lifting only occurs at a local minimum.

Theorem 1 *Given a finite CSP, the Fill algorithm eventually reaches a solution, if one exists.*

Proof: Suppose the algorithm does not find a solution. Then we can divide the state space into states that are lifted infinitely often, and states that are lifted at most a finite number of times. Let $\mathcal{S}$ be the set of states that are lifted infinitely often. A *boundary state* of $\mathcal{S}$ is one that is adjacent to a state not in $\mathcal{S}$. To see that $\mathcal{S}$ must have a boundary state, consider a path that connects any state in $\mathcal{S}$ to a solution. Let s be the last state that belongs to $\mathcal{S}$ on this path. Clearly s is a boundary state of $\mathcal{S}$.

As the algorithm proceeds, there must eventually come a time when all the following conditions hold.

1. The states outside $\mathcal{S}$ will never again be lifted.
2. The cost of each state in $\mathcal{S}$ exceeds the cost of every state not in $\mathcal{S}$.
3. A boundary state of $\mathcal{S}$ is lifted.

Notice that at the moment the last event occurs, the boundary state involved must be a local minimum. But this contradicts the fact that the state is adjacent to a state not in $\mathcal{S}$, which (by the second condition) has a lower cost. Thus, the assumption that a solution is not found must be false. ▮

The Breakout algorithm may be regarded as a "sloppy," or approximate version of the Fill algorithm, where some of the increase in cost spills onto neighbouring states. Note that Breakout is much more efficient because of the compact storage of the increased costs.

[10]The reader may wonder whether a simpler algorithm that just marked local minima, and never visited them again, would be complete. It turns out this is not the case because of the possibility of "painting oneself into a corner." Note that Fill may revisit states.

The Fill algorithm is itself related to the LRTA* algorithm of Korf [1990], which has also been proved complete. The latter algorithm has been studied in the context of shortest path problems, rather than CSPs. In a path problem, the goal state is usually known ahead of time. Note, however, that this is not essential as long as a suitable heuristic distance function is available. Iterative improvement implicitly treats a CSP as a path problem by seeking a path that transforms an initial state into a solution state, thereby obtaining the solution state as a side product. From this viewpoint, the number of conflicts serves as a heuristic distance function. (However, this heuristic is not *admissible* in the sense of the A* algorithm, because it may occasionally overestimate the distance to a solution.)

Both Fill and LRTA* have the effect of increasing the stored cost of a state when at a local minimum. We note the following technical differences between the two algorithms.

1. LRTA* transitions to the neighbour of *least* cost, whereas the Fill algorithm is satisfied with any lower cost neighbour.
2. LRTA* may modify costs at states that are not local minima, and may decrease costs as well as increasing them.

Item 1 suggests Fill/Breakout is more suited for CSPs, where the number of states adjacent to a given state is generally very large.

One might consider using the Fill algorithm directly to solve CSPs. However, the only obvious advantage of this over Breakout is the theoretical guarantee of completeness. It appears that, in practice, Breakout almost always finds a solution anyway, and has a much lower overhead with regard to storage and retrieval costs. The Fill algorithm requires storage space proportional to $n \times l$, where n is the number of variables, and l is the number of local minima encountered on the way to a solution. By contrast, Breakout only requires storage proportional to the fixed set of nogoods derived from the specification of the problem. Moreover, preliminary experiments suggest that Fill requires many more steps than Breakout to reach a solution. This may be due to a beneficial effect of the cost increase spillovers in Breakout—presumably depressions get filled more rapidly.

It is known that Breakout itself is not complete. As a counterexample, consider a Boolean Satisfiability problem with four variables, x, y, z, w, and the clause

$$x \vee y \vee z \vee w$$

together with the 12 clauses

$$\begin{array}{cccc} & \neg x \vee y & \neg x \vee z & \neg x \vee w \\ \neg y \vee x & & \neg y \vee z & \neg y \vee w \\ \neg z \vee x & \neg z \vee y & & \neg z \vee w \\ \neg w \vee x & \neg w \vee y & \neg w \vee z & \end{array}$$

Note that these clauses have a single solution, in which all the variables are *true*.

Suppose the initial state sets all the variables to *false*. It is not hard to see that the Breakout algorithm will produce oscillations here, where each variable in turn moves to *true*, and then back to *false*.

To understand this better, consider the three states S_1, S_2, and S_3, such that x is *true* in S_1, y is *true* in S_2, and both x and y are *true* in S_3. All of the other variables are *false* in each case.

Each time S_1 occurs as an local minimum, the weight of each of its nogoods is incremented. Thus, the total cost of S_1 increases by 3. Since S_1 shares two nogoods with S_3, the cost of the latter increases by 2 at the same time. Similarly, when state S_2 becomes a local minimum, the cost of S_3 increases by 2. This means that S_3 undergoes a combined increase of 4 during each cycle, which exceeds the increase for each of S_1 and S_2. Thus, S_3 is never visited, and this path to a solution is blocked.

Thus, the basic reason for incompleteness is that the cost increase spillovers from several local minima can conspire to block potential paths to a solution. However, this kind of blockage requires nogoods to interact locally in a specific "unlucky" manner. For large random CSPs, the number of possible exits from a region of the state space tends to be very large, and the probability that all the exits get blocked in this way would appear to be vanishingly small. This may explain why we did not observe infinite oscillations in our experiments.

Conclusions

The class of Boolean 3-Satisfiability problems is of importance because of its central position in the family of NP-complete problems. We have seen that the Breakout algorithm performs very successfully on 3-SAT problems with prearranged solutions, including those at the critical clause density. Breakout also performs quite well on K-coloring problems, and appears superior to previous approaches for both of these classes.

We have provided analyses that explain both the efficiency of the algorithm, and its apparent avoidance of infinite cycles in practice. In particular, an idealized version of the algorithm has been proved to be complete.

Several possibilities for future work suggest themselves. The relationship to LRTA* ought to be explored in greater detail, particularly in view of the attractive learning capabilities of LRTA*. One might also consider applying some form of Breakout to other classes of search problems where a cost measure can be distributed over individual "flaws" in a draft solution. More generally, the metaphor of competing forces that inspired Breakout may encourage novel architectures for other computational systems.

Acknowledgements The author is grateful to Rina Dechter, Bob Filman, Dennis Kibler, Rich Korf, Steve Minton and Bart Selman for beneficial discussions, and would also like to thank the anonymous referees for their useful comments.

References

Adorph, H. M., and Johnson, M. D. A discrete stochastic neural network for constraint satisfaction problems. In *Proceedings of IJCNN-90*, San Diego, 1990.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. M. Where the *really* hard problems are. In *Proceedings of IJCAI-91*, Sydney, Australia, 1991.

Dechter, R. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3), 1990.

Korf, R. E. Real-time heuristic search. *Artificial Intelligence*, 42(2-3), 1990.

Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI-90*, Boston, 1990.

Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3), 1992.

Mitchell, D.; Selman, B.; and Levesque, H. Hard and Easy Distribution of SAT Problems. In *Proceedings of AAAI-92*, San Jose, California, 1992.

Morris, P. Solutions Without Exhaustive Search: An Iterative Descent Method for Solving Binary Constraint Satisfaction Problems. In *Proceedings of AAAI-90 Workshop on Constraint-Directed Reasoning*, Boston, 1990.

Morris, P. On the Density of Solutions in Equilibrium Points for the Queens Problem. In *Proceedings of AAAI-92*, San Jose, California, 1992.

Selman, B.; Levesque, H.; and Mitchell, D. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of AAAI-92*, San Jose, California, 1992.

Selman, B., and Kautz, H. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of IJCAI-93*, Chambéry, France, 1993.

Sosic, R., and Gu, J. 3,000,000 Queens in Less Than One Minute. *Sigart Bulletin*, 2(2), 1991.

Williams, C.P., and Hogg, T. Exploiting the Deep Structure of Constraint Problems. Preprint, Xerox PARC, 1993.

Zweben, M. A Framework for Iterative Improvement Search Algorithms Suited for Constraint Satisfaction Problems. In *Proceedings of AAAI-90 Workshop on Constraint-Directed Reasoning*, Boston, 1990.