# Adaptive Methods for Netlist Partitioning

Wray L. Buntine, Lixin Su, A. Richard Newton and Andrew Mayer
Dept. of Electrical Engineering and Computer Sciences
Room 520 Cory Hall, Univ. of Cal., Berkeley, CA, 94720-1770

## Abstract

*An algorithm that remains in use at the core of many partitioning systems is the Kernighan-Lin algorithm and a variant the Fidducia-Matheysses (FM) algorithm. To understand the FM algorithm we applied principles of data engineering where visualization and statistical analysis are used to analyze the run-time behavior. We identified two improvements to the algorithm which, without clustering or an improved heuristic function, bring the performance of the algorithm near that of more sophisticated algorithms. One improvement is based on the observation, explored empirically, that the full passes in the FM algorithm appear comparable to a stochastic local restart in the search. We motivate this observation with a discussion of recent improvements in Monte Carlo Markov Chain methods in statistics. The other improvement is based on the observation that when an FM-like algorithm is run 20 times and the best run chosen, the performance trace of the algorithm on earlier runs is useful data for learning when to abort a later run. These improvements, implemented with a simple adaptive scheme, are orthogonal to techniques used in state-of-the-art implementations, and therefore should be applicable to other VLSI optimization algorithms.*

## 1 Introduction

In discrete optimization, there are several broad classes of techniques for developing an algorithm. In *local search* [1] a neighborhood space is created and algorithms update a current solution by local moves around that space. Some variations of this are gradient descent which makes the most locally promising move at each step, simulated annealing [16] which makes a move in a stochastic direction based on the theory of Gibbs sampling [8], and tabu search which enforces restrictions on the ordering in which variables can be updated [10]. Another approach, *problem decomposition* breaks the problem down into smaller components, which are then solved separately, and their solutions spliced back together to offer a solution to the original problem.

Netlist partitioning [3] is one approach to the general idea of problem decomposition. In the two-way min-cut netlist partition problem, one is given a hypergraph whose nodes are referred to as cells, and whose hyper-edges are referred to as nets. The cells are to be split into two partitions such that the minimum number of nets have cells in both partitions. These cut nets are referred to as the *cut-set*. This is the hypergraph version of the graph partitioning problem. Typically, a balance constraint is also enforced whereby the area of each partition must lie in a given interval.

Hypergraph partitioning is generally useful in constraint satisfaction problems for the following reason. For an arbitrary constraint satisfaction problem, $(V,C)$, where there are variables $V$ and constraints $C$, suppose the variables are split into two partitions $V_1$ and $V_2$ and constraints $C_1$ and $C_2$ are wholly contained within the partitions $V_1$ and $V_2$ respectively, and constraints $C_{1+2}$ are cut (so $V_1 \cup V_2 = V$ and $C_1 \cup C_2 \cup C_{1+2} = C$). Then under reasonable conditions on the nature of the constraints, one can first solve the problem of $(V_1,C_1)$, express the solutions for $V_1$ as partial evaluations of the cut constraints $C_{1+2}$ taking values $V_{1+2}$, and finally solve the remaining induced problem $\left(V_2+V_{1+2},C_2+C'_{1+2}\right)$. If the cutset size $|C_{1+2}|$ is small then one solves two smaller problems instead of one large problem. While we restrict ourselves here to a two-way partitioning, in general a good k-way partitioning can be similarly invaluable for simplifying a problem. Related graphical constructions are used to decompose probability distributions, linear equations, and constraints in design [15].

Common local search techniques for netlist partitioning are the Kernighan-Lin (KL) algorithm and the Fidducia-Matheysses (FM) algorithm [3]. While these have some similarities with their cousin, the Lin-Kernighan (LK) algorithm for the traveling salesman problem [12], the algorithms are in fact very different. The LK algorithm is frequently within 2% of optimal on large problems, whereas the FM must be run repeatedly, perhaps 50 times, before a solution would be found that is near optimal, and on large problems (10,000 or more nodes), many times more might be needed. Of course, by current theory and algorithms one rarely knows what is truly optimal. One run of the FM algorithm really corresponds to the inner loop of the LK algorithm. Recent state of the art algorithms for partitioning

0-89791-993-9/97 $10.00 © 1997 IEEE

such as PROP [5], PANZA [13], and STRAWMAN [11] apply more clever heuristics for choosing the next node, techniques for finding an initial partition, or they preprocess the problem via clustering. However, two out of three of these algorithms still retain the FM or KL algorithm at their core, and usually run it repeatedly.

We apply principles of *data engineering* where visualization and statistical analysis are used to analyze data gathered from a process, in this case the run-time performance of the algorithm. Whereas Hauck and Borriello [11] systematically explored a number of variations around the standard FM algorithm, in this research, we sought to understand and thereby improve specific details of the existing algorithms. We have identified two statistically motivated improvements to the algorithm. Interestingly enough, these two improvements lead us to develop a new algorithm which is comparable to PROP in performance on some standard test problems. We see the general maxim "speed over smarts" apply because our approach merely places an adaptive wrapper around the simple "local search with tabu list" core of FM without requiring sophisticated approximation techniques or clustering.

For background on the netlist partitioning problem the KL and FM algorithms, and many other algorithms of interest, we refer the reader to the excellent review article by Alpert and Kahng [3]. In our Section 3, experiment details and protocol are discussed. Comparison of algorithms is fraught with difficulty, so we take some care in this section to outline the precautions we took. In the following sections we introduce the analyses we did that motivated our work. We gathered some data on the runs, and performed a number of basic statistical tests, from which we formulated two hypotheses about how the algorithms were really operating. These are also explained in Section 4 and later in Section 6. From the understanding we gleaned from the experiments, we devised a modified control for a FM-style algorithm which is introduced in Section 7.

## 2 Basic Algorithm

The basic KL algorithm is outlined below, and for more detail we refer the reader to [3] or many of the textbooks in VLSI design. Some initial partition is used, in our case a random partition. The algorithm uses a data structure (usually buckets, linked lists or heaps) that gives for each cell the immediate "gain" (measured in terms of reduction of cutset size) obtained if that cell were to be moved to the other partition. The data structure handles operations such as "fetch the cell with the best gain," "remove the cell from the available list," "update the cell's gain," etc. The choice of this structure and its method for tie-breaking (LIFO, FIFO) can impact the performance significantly. More advanced algorithms might have more advanced move heuristics such as 2-step look-ahead. The algorithm proceeds in *passes*. In each pass, all cells are initially *unlocked* and available to be moved. The unlocked cell with the highest gain is drawn from the data structure, moved to the other partition, and all other cell's gains are updated (in the simplest case this just involves looking at the neighbors). Then this drawn cell is locked (or placed on a tabu list) until the end of the pass. This proceeds until all cells are locked, and no more moves are possible. So all cells are swapped, and we now have a mirror of the partition we started with (in the binary case). The algorithm has just finished one pass through all the cells, and the least cutset-size partition in the pass is taken as the starting partition for the next pass. The KL algorithm typically has 5-10 passes for the smaller circuits, and 10-30 for the larger circuits. The algorithm terminates when no improvement in cutset size is obtained for a pass. A good summary of modifications to the basic algorithm, and further pointers to literature can be found at [11, 3].

In our implementation in C, netlist partitions are constrained to contain between 45% and 55% of the area (the 45-55% balance criteria), and each cell and pin is assumed to have unit area for this purpose. The algorithms are run from random initial placements, and cells are stored in a bucket structure, an array of doubly-linked lists indexed by the gain from making a swap. Tie breaking (accessing the doubly-linked list) is done in a LIFO fashion [11, 3].

## 3 Experimental details

All experiments are run on a DEC Alpha 5000/400MHz with 128Mb of memory. The larger data sets used for experimentation were obtained from Alpert's website (`http://vlsicad.cs.ucla.edu/~cheese/`) which in turn are netlist versions of some standards from TU Munich from their copy of the ACM/SIGDA Layout Benchmark Suite from MCNC. Some smaller test problems were from the Berkeley OCTTOOLS environment. The details are listed in Table 1. Notice in some cases the pins are 0 due to the earlier processing of the netlist (in our experiments pins and cells are treated identically). Trials were also run with Dutt and Deng's recently released C++ code for FM partitioning, lookahead, and their PROP algorithm [5]. Use of Dutt and Deng's code is discussed in the appendix. We have compared our FM implementation in both timing and resultant cutset size to Dutt and Deng's C++ code, with the timing differences explainable by our use of C.

Finally, in comparing results from algorithms with unequal times, we use the *best-of-X wrapper*. This wrapper runs the faster algorithm multiple times so that the algorithms being compared have approximately equal run time (ignoring discretization effects). Results are then reported for the best of the X ran. For instance if comparing PROP against FM, typically a best-of-5 wrapper would be used.

| Dataset | Nets | Cells | Pads |
|---|---|---|---|
| C1355 | 245 | 73 | 204 |
| C1908 | 282 | 249 | 58 |
| C2670 | 529 | 348 | 296 |
| C3540 | 811 | 761 | 72 |
| C5315 | 907 | 729 | 296 |
| C6288 | 1925 | 1893 | 64 |
| biomed | 5742 | 6514 | 0 |
| struct | 1920 | 1952 | 0 |
| industry2 | 13419 | 12637 | 0 |
| industry3 | 21923 | 15406 | 0 |
| avq-small | 22124 | 21853 | 65 |
| avq-large | 25384 | 25113 | 65 |
| s9234 | 5844 | 5866 | 0 |
| s13207 | 8651 | 8772 | 0 |
| s15850 | 10383 | 10470 | 0 |
| s35932 | 17828 | 18148 | 0 |
| s38417 | 23843 | 23949 | 0 |
| s38584 | 20717 | 20995 | 0 |
| golem | 144949 | 103048 | 0 |

Table 1: Summary of the Data sets

Our full algorithm corresponds to about 40-60 FM runs, depending on the data set, so for each data set X is chosen accordingly to make a time-equalized comparison.

## 4 Why full passes in the FM algorithm?

The basis of our research is the following observation. Shown in Figure 1 is the microscopic behavior of the cost function during a typical run of the algorithm. The X-axis corresponds to swaps, and the Y-axis is the evaluated cost function. The lines drawn vertically from the cost curve down to the X-axis indicate the start/end of a pass. Note that the first three passes have not been included in the figure since these would dwarf the above curves as the algorithm is mostly performing rapid gradient descent initially.

What is most interesting about the figure is the inverted “U” shape of the costs during each later pass. The first pass or two (not shown) are “U” shaped as the algorithm does rapid gradient descent. Note that the end point of each pass is identical in cost to the starting point, due to the symmetry of swapping every cell. The inverted “U” shape means that the algorithm invariably chooses the best partition from near the beginning or the end of the pass. This makes sense if one considers that the algorithm is merely making local changes towards the end of a run, and the local changes therefore lie near the beginning or end of the pass. This raises the question, what does the effort for the middle 95% of the pass achieve? We refer to a *full pass* as one that eventually locks every node during its course. In view of this we made the following hypothesis:

**Hypothesis 1:** The use of full passes in the later passes of the FM algorithm is serving the purpose of performing a local restart for the algorithm, and this goal can also be achieved by stochastic local moves.

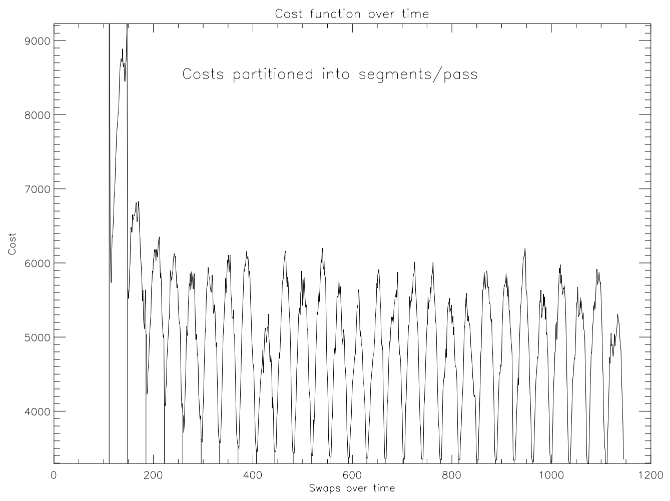

Figure 1: Typical behavior of cost

To test out the hypothesis, we designed and ran a number of experiments. We modified FM so that no pass would run for more than half way through. We call this *Half-pass FM*. Second, we implemented a stochastic variation of FM. The stochastic variation has two complementary aspects: we would like to introduce shorter passes, but with shorter passes alone, the performance of the algorithm is poorer (results of an experiment not reported here). Therefore, the introduction of a means of escaping local minima is also introduced using stochasticity. Components of this stochastic FM (SFM) are as follows:

**Initial FM passes:** Full passes are done for the first 4 passes, and thereafter, passes are stopped short, and occasional stochastic passes are used to escape local minima. The initial passes in FM are mostly running down-hill, with the cutset size decreasing. The full passes initially seemed to be important to settle the search in a good local region, perhaps due to the nature of tabu search used in FM. Significant stochasticity is injected into the algorithm by the starting point for search so there seemed no benefit in introducing additional stochasticity.

**Short Passes:** These passes are constrained to be less than 15% in size of a full pass, and as the algorithm converges, the pass length is decreased yet more using a simple adaptive scheme. The current pass length is set to 200% of the length of the successful prefix of the previous pass, but is constrained to not go below 2% of a full pass. We refer to these passes as *short passes*. Note we view this implementation of a short pass is being a straw-man for the purposes of our experimentation, and we make no claim about the parameters (15%,200%,2%) being carefully tuned, or theoretically justified. The justification for this scheme is as follows: passes cannot be too short (hence, 2%) or no neighborhood search is done; once a reasonable state is found (in the initial 4 FM passes) passes do not need to be long on the basis of evidence shown in Figure 1 (hence, 15%); finally, shortening passes on the basis of the length of previous passes seems to be an inherently dangerous practice (hence 200% of the previous pass).

**Stochastic Passes:** the second kind of pass we use is to escape local minima. The stochastic pass does not choose the top or best cell to move (according to gain), but instead draws the "next best cell" repeatedly from the gain bucket structure and then rejects it with probability $\delta$, where $\delta$ is set so that the probability of drawing a down-hill (favorable) move if one exists is 0.8. That is, prefer swaps at the top of the gain bucket array, and with probability about 0.8, move down-hill. Moreover, the stochastic pass will move $\rho$ percent of a full pass where $\rho \in [0.02, 0.10]$ and is selected uniformly. Again, we make no claim about the parameters $(0.8, 0.02, 0.10)$ being carefully tuned, or theoretically justified. On one particular problem the stochastic pass performed poorly in comparison with the full pass, and in this case we played with the parameter $\rho$ with some success. The justification for this choice is as follows: the stochastic pass is implemented to generally go down-hill at each move, and the stochastic pass is allowed variable length. We found the length of the stochastic pass is a determiner of the algorithm's performance, but in simple studies we could not find a way to judge an appropriate length so randomization of the length seemed a fair middle ground. Clearly, the stochastic pass could not be too long or too short. A long stochastic pass is no different to a full restart of the algorithm.

**Stochastic FM:** Our stochastic FM goes as follows: When the algorithm reaches a local optimum (the most recent pass chose its first state as the best new state), it then runs one stochastic pass to attempt to escape followed by a sequence of regular short passes until another local optimum is reached. If the new local optimum is superior, then this becomes the current state, otherwise the algorithm restarts from the previous local optimum. In this way, the algorithm repeatedly does gradient descent, and on reaching a local minimum attempts to jump out with a stochastic pass. As it stands this algorithm never terminates, so we terminate if no improvement is gained after 100 passes, or if a time-bound is exceeded. We call this *stochastic FM*, as

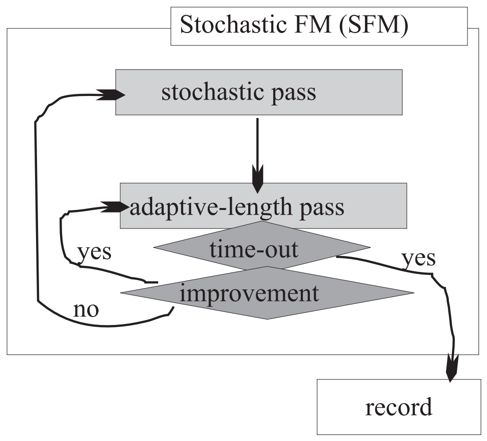

Figure 2: Stochastic FM

given in Figure 2. Depending on the problem used and the time bound, stochastic FM may do 2 to 20 times as many passes as regular FM. While this approach might remind one of simulated annealing, the algorithm is quite different, as discussed in Section 5.

These two FM variations, half-pass and stochastic, are compared empirically against standard FM in Table 2. Stochastic FM labelled SFM is run for about the same time as regular FM by setting the time-bound appropriately. Half-pass FM is labelled 1/2-FM. It is clear that the half-

| Dataset | FM | SFM | 1/2-FM |
|---|---|---|---|
| C1355 | 22.6± 4.1 | 21.4± 3.2 | 28.5± 6.9 |
| C1908 | 38.7± 3.3 | 36.6± 2.6 | 43.5± 6.9 |
| C2670 | 38.2± 6.8 | 35.5± 4.4 | 40.7± 6.1 |
| C3540 | 84.1± 14.6 | 80.7± 11.7 | 105± 21.1 |
| C5315 | 47.2± 10.4 | 48.7± 3.6 | 68.3± 13.7 |
| C6288 | 86.9± 24.3 | 94.3± 15.7 | 121± 24.6 |
| biomed | 214± 39.5 | 240± 18.8 | 256± 31.4 |
| struct | 55.1± 10.5 | 48.3± 5.5 | 92.8± 10.8 |
| industry2 | 769± 193 | 923± 103 | 1269± 175 |
| industry3 | 568± 185 | 516± 223 | 999± 347 |
| avq-small | 609± 132 | 698± 95.5 | 1196± 154 |
| avq-large | 774± 175 | 795± 78.8 | 1332± 185 |
| s9234 | 93.4± 27.3 | 113± 24.4 | 201± 35.3 |
| s13207 | 131± 22.7 | 144± 14.8 | 187± 25.1 |
| s15850 | 186± 37.3 | 200± 36.7 | 301± 36.0 |
| s35932 | 191± 47.4 | 305± 57.4 | 531± 75.1 |
| s38417 | 521± 70.8 | 484± 38.6 | 560± 89.9 |
| s38584 | 289± 105 | 481± 134 | 887± 139 |
| golem | 3156± 318 | 2990± 201 | 6228± 836 |

Table 2: Results for FM vs. Stochastic (SFM) vs. Half-pass (1/2-FM)

pass FM is a terrible algorithm. It seems the full passes, despite all their overhead, are necessary for the performance of the FM algorithm[1]. Note we believe this is an artifact of the property that for binary partitioning, the state at the end of a pass is a mirror of the state at the start of the pass, and we expect this would not hold for other related algorithms. Stochastic FM is a reasonable algorithm. In some larger problems it is inferior, however. Given that we have done little to tune the performance of the stochastic component of the algorithm, we believe there is considerable room for improvement. Playing with the parameters on some hard problems (for instance the *s38584* problem) showed we could get considerably better performance. While it appears that one of our more efficient stochastic passes is not comparable to one full pass in terms of contribution to the final result, this is not in contradiction with our hypothesis, and the stochastic FM algorithm is yielding reasonable performance all up. We concluded that Hypothesis 1 has good evidence in its favor.

One obvious question that arises from all this is: What happens if the stochastic FM is allowed to run for longer? We conducted a variety of experiments to explore this question. It seems that in general, the longer the stochastic algorithm is run, the better the performance over time equalized best-of-X FM, yet more support for our hypothesis. The longer stochastic FM algorithm routinely yields improvements of up to 25% over time-equalized best-of-X FM. The shorter (middle) version of stochastic FM in these experiments runs for about 6 regular FM runs. The longer version runs for the equivalent of 50 regular FM runs but usually terminates far shorter since we have the algorithm terminate if there is no improvement of cost after 100 passes. Surprisingly we found that best-of-X shorter SFM was comparable with longer SFM in minimum cutset size after being time-equalized, so there seemed little reason to distinguish between restarting the algorithm or letting it run for longer.

## 5 Notes on Stochastic Search Algorithms

The stochastic technique we have introduced in the previous section could be compared with simulated annealing, for instance reviewed in [16]. We note that by most accounts, standard simulated annealing is not competitive with best-of-X FM so we did not implement it. There are several differences between our stochastic FM and standard implementations of simulated annealing. First, stochastic passes are only introduced into our SFM algorithm at the point of a local minima. All other times, standard gradient descent via a variation of FM is used. Thus our approach is sampling the space of local minima stochastically, not the full search space. This would seem to be a more appropriate technique for optimization. Second, our approach has a very different stochastic component. The FM core maintains a list of all possible moves at each point, and the stochastic element samples efficiently from that, with a high probability of an upward move. In contrast, standard simulated annealing only samples one move at a time, so is not able to sample high quality moves as efficiently as our variation.

Simulated annealing in its popular form is based on the theory of Gibbs sampling. While Gibbs samplers are seeing a resurgence in statistics, it is well known that they are inefficient. Some basic problems are discussed in [14], and a simple explanation is as follows: Gibbs samplers generally take $O(N^2)$ to move $N$ steps, more or less because of the fact that the standard deviation of a sample is $O(1/\sqrt{N})$, and hence the sum of the sample is $O(\sqrt{N})$ away from the mean. One might argue that the use of tabu lists in local search could be justified on the same grounds—the full space is more slowly traversed if moves are allowed to be entirely local instead of enforcing some discrete analogue to line-search.

More recent methods for Monte Carlo Markov Chain sampling use better proposal distributions or more sophisticated methods for taking multiple steps. To use stochastic methods in search, one clearly needs these more efficient samplers. We view our approach as an *ad hoc* technique for introducing stochasticity in search without having to suffer the known inefficiencies of the Gibbs samplers inside standard simulated annealing.

## 6 Making multiple runs more efficient

One can see from Figure 1 that the last half of the run is mostly just polishing the final solution. Moreover, the standard deviations in the final cutset size for multiple FM runs are quite large in comparison with the improvements made in the last half or two-thirds of the run. We gathered data over many runs of many different problems, and determined that the correlation coefficient between the intermediate and final cutset sizes approaches 1.0 towards the end of the run, and is already quite high half-way through the run.

Perhaps one could estimate part way through a run that the current run could not, with high probability, improve sufficiently fast to produce a new lower cost winner. The plot of different costs in Figure 3 shows the progress of the cutset size after each pass for 20 different runs on the "avq-large" dataset. Lines are shaded grey depending on the size of the final cutset. From the plot, the potential of cutting-off runs at about pass 7 is clear. A good time to perform the corresponding test on the "golem" dataset is at pass 13. Since we cannot determine "7" or "13" ahead of time for different datasets, we need to do so adaptively as the algorithm gains empirical experience on the particular

[1]This is a fact we are sure many researchers and Kernighan and Lin are aware of.

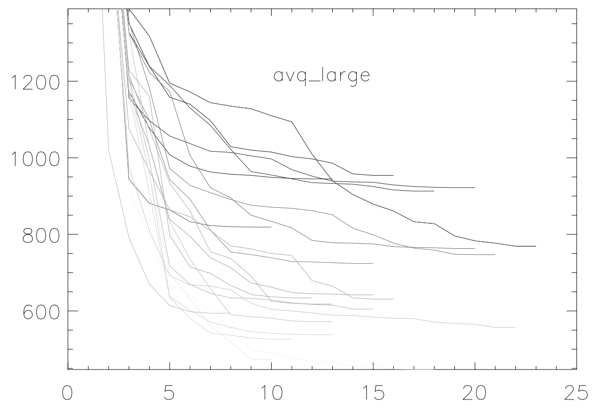

Figure 3: Multiple FM runs from the avq-large domain

dataset. Such a strategy could be considered to be a serial, on-line version of the "go with the winners" algorithm for tree search suggested by Aldous [2].

We have implemented this algorithm in the following simple way, shown in Figure 4. We place a "cutoff best-of X" wrapper around the FM variant we are running. It records data about runs and passes during operation. After each pass, it performs a simple quality test to see if this run is converging fast enough. The quality test is an adaptive component of the algorithm that improves in rigor as the algorithm proceeds. It passes the first 5 runs without trouble, and thereafter checks to see if the current pass has cutset size as good as the same numbered pass in previous runs that were in the lower 20-th percentile of final cutset size. The lower 20-th percentile of $N$ runs is estimated conservatively: take the run with the $Max(4, N/5)$-th smallest final cutset size. The wrapper runs X runs in all. Again, this quality test is a simple, *ad hoc* adaptive scheme we have used as a straw-man for the purposes of testing out the basic concept.

We applied this wrapper to standard FM. This makes the best-of-X algorithm run about 20% faster, and appears to have little detrimental effect on performance. When running best-of-X FM and cutoff b-of-20 FM for the same amount of time, cutset size reductions where 0-15%, for instance on golem mean cutset size went from 2618 down to 2435 when using cutoff runs. We believe there is considerable room for improvement in our implementation. For instance, we could reduce running time further with a smarter adaptive core.

## 7 A proposed new algorithm: ASFM

We have combined the above two affects—stochastic passes to escape local minima, and cutoff runs to abort potentially poor runs—in the one algorithm we call *ASFM*, where the "AS" stands for "adaptive stochastic". Since the code for these two modifications discussed in Sections 4

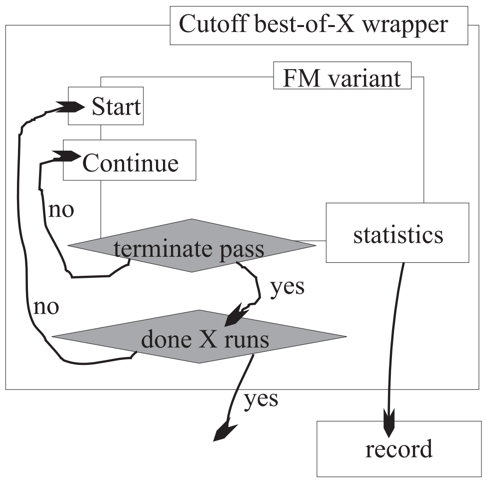

Figure 4: Cutoff best-of-X wrapper

and 6 do not interact, to implement this we merely set both sets of switches in our implementation. We explored two variations of this basic implementation, where the stochastic element was allowed to run for a shorter or longer period of time, corresponding in time to roughly 6 or 10-15 times a full FM run. As before, all results are time equalized using the best-of-X wrapper. The results for these experiments are given in Table 3.

The longer stochastic version appeared superior so we chose that as our winning algorithm. This is our final ASFM. As a comparison, we compare our algorithm with Dutt and Deng's PROP in Tables 4 and 5. Table 4 gives the average performance for each algorithm over 10 repeated trials. For the trials reported in Table 4, X for PROP is about 5-7 runs, and X for FM is about 25-40. Because of the differences in implementation, these comparisons should be taken with a grain of salt. The problem where key differences arise between ASFM and PROP is *s38417*, and indicates the clear superiority of the heuristic rules in PROP. However, with some tuning of our stochastic parameters, we were able to get closer to the performance of PROP.

Results in Table 4 are the average of 10 runs. The best results from these runs are given in Table 5, giving the best performance achieved, and the total time taken to achieve that for both ASFM (for ten trials) and best-of-X PROP (which has its real times given), and the value X used for PROP. For instance, for one run of *38584*, ASFM takes approximately 2 minutes, and 30 seconds on *biomed*. When compared with the results of Hauck and Borriello [11] run on a much slower machine, the results in Table 4 are comparable, not the best results in Table 5. This difference between ASFM and the quality of solutions in [11] can be largely attributed to the effects of clustering. The final col-

| Dataset | b-of-X FM | b-of-X cutoff b-of-20 short SFM | cutoff b-of-20 long SFM |
|---|---|---|---|
| C1355 | 17.6± 0.5 | 17.3± 0.5 | 17.3± 0.5 |
| C1908 | 32.7± 1.6 | 30.9± 1.6 | 30.3± 2.2 |
| C2670 | 25.4± 1.2 | 19.3± 1.7 | 21.9± 2.9 |
| C3540 | 63.1± 1.1 | 60.2± 0.4 | 61.3± 1.3 |
| C5315 | 36.8± 6.9 | 32.2± 3.2 | 32.3± 3.2 |
| C6288 | 50.3± 0.7 | 50.0± 0.0 | 50.2± 0.6 |
| biomed | 135± 5.2 | 94.2± 4.1 | 83.0± 0.0 |
| struct | 42.0± 1.4 | 39.6± 1.7 | 38.8± 0.7 |
| industry2 | 403± 64.4 | 293± 10.3 | 285± 27.9 |
| industry3 | 262± 19.7 | 248± 6.7 | 247± 7.7 |
| avq-small | 300± 30.7 | 244± 25.9 | 252± 31.0 |
| avq-large | 391± 30.5 | 339± 37.2 | 304± 33.8 |
| s9234 | 52.0± 2.3 | 47.5± 1.4 | 46.2± 1.9 |
| s13207 | 87.9± 9.4 | 82.9± 5.6 | 78.1± 5.3 |
| s15850 | 113± 12.8 | 85.9± 8.0 | 77.0± 7.9 |
| s35932 | 117± 10.6 | 87.6± 13.8 | 83.5± 19.1 |
| s38417 | 372± 22.6 | 310± 33.5 | 282± 28.7 |
| s38584 | 136± 8.7 | 114± 12.9 | 84.1± 12.0 |
| golem | 2403± 119 | 2123± 137 | 1834± 247 |

Table 3: Results for best-of-X FM vs. short and long ASFM

| Dataset | b-of-X FM | b-of-X PROP | ASFM |
|---|---|---|---|
| C1355 | 17.6± 0.5 | 18.1± 1.0 | 17.3± 0.5 |
| C1908 | 32.7± 1.6 | 30.8± 1.6 | 30.3± 2.2 |
| C2670 | 25.4± 1.2 | 18.1± 0.9 | 21.9± 2.9 |
| C3540 | 63.1± 1.1 | 64.3± 2.5 | 61.3± 1.3 |
| C5315 | 36.8± 6.9 | 30.1± 2.3 | 32.3± 3.2 |
| C6288 | 50.3± 0.7 | 51.0± 2.7 | 50.2± 0.6 |
| biomed | 135± 5.2 | 88.4± 6.0 | 83.0± 0.0 |
| struct | 42.0± 1.4 | 37.5± 3.3 | 38.8± 0.7 |
| industry2 | 403± 64.4 | 246± 35.4 | 285± 27.9 |
| industry3 | 261± 19.7 | 292± 33.0 | 247± 7.7 |
| avq-small | 300± 30.7 | 307± 53.6 | 252± 31.0 |
| avq-large | 391± 30.5 | na | 304± 33.8 |
| s9234 | 52.0± 2.3 | 50.9± 5.5 | 46.2± 1.9 |
| s13207 | 87.9± 9.4 | 88.6± 13.2 | 78.1± 5.3 |
| s15850 | 112± 12.8 | 83.5± 12.0 | 77.0± 7.9 |
| s35932 | 117± 10.6 | 72.2± 3.7 | 83.5± 19.1 |
| s38417 | 372± 22.6 | 97.2± 18.0 | 282± 28.7 |
| s38584 | 136± 8.7 | 70.2± 10.3 | 84.1± 12.0 |
| golem | 2403± 119 | 1785± 190 | 1834± 247 |

Table 4: Results for FM, PROP and ASFM

| Dataset | bof-10 ASFM | time in secs. | bof-X PROP | time in secs. | X | best publ. |
|---|---|---|---|---|---|---|
| C1355 | 17 | 4.8 | 17 | 6.8 | 73 | * |
| C1908 | 29 | 5.05 | 29 | 7.25 | 58 | * |
| C2670 | 18 | 10.20 | 17 | 14.84 | 73 | * |
| C3540 | 60 | 19.71 | 61 | 28.35 | 47 | * |
| C5315 | 29 | 24.5 | 28 | 35.3 | 62 | * |
| C6288 | 50 | 55.6 | 50 | 81.3 | 33 | * |
| biomed | 83 | 312.1 | 84 | 451.3 | 51 | 83 |
| struct | 37 | 49.4 | 35 | 70.8 | 61 | 33 |
| industry2 | 240 | 698.7 | 199 | 1007 | 38 | 174 |
| industry3 | 241 | 811.1 | 243 | 1171 | 50 | 241 |
| avq-small | 204 | 1229 | 214 | 1762 | 77 | 129 |
| avq-large | 241 | 1583 | na | na | 79 | 127 |
| s9234 | 42 | 185.0 | 45 | 266 | 71 | 40 |
| s13207 | 71 | 257.8 | 73 | 374 | 59 | 57 |
| s15850 | 66 | 313.5 | 54 | 454 | 50 | 44 |
| s35932 | 61 | 727.0 | 73 | 1052 | 60 | 42 |
| s38417 | 231 | 995.3 | 57 | 1436 | 51 | 49 |
| s38584 | 65 | 1102 | 56 | 1582 | 56 | 47 |
| golem | 1429 | 13197 | 1625 | 18952 | 113 | * |

Table 5: Best results for PROP and ASFM

umn in Table 5 lists the best known result from [11, 6, 13].

## 8 Discussion

First, we believe the interpretation of full passes we give in Section 4 is a significant insight into FM and KL-style algorithms. For instance, this explains why k-way partitioning with FM and KL-style algorithms does not perform well. For k-way partitioning with *k* greater than 2, the final state in the pass is not a mirror of the initial state, so as the pass proceeds, the states just drift away from the initial state and a second local move is not created at the end of the pass. The subsequent comparison between our modified stochastic search algorithm and simulated annealing (discussed in Section 5) also indicates there is considerable potential for improved stochastic search algorithms of a very different flavor to standard simulated annealing. Our methods and results throws additional light on the role of stochasticity in search, and alternatives was of introducing it; compare with the empirical investigations by Gent and Walsh on satisfiability [9].

The results of the experiments in Section 7, when compared with results from state-of-the-art algorithms recorded in the literature such as PROP [5] STRAWMAN [11] and PANZA [13] are competitive. Since PROP is itself an iterated algorithm with full passes, and STRAWMAN uses FM as its basis, it is realistic that our approach can also be used to speed up these algorithms even more. We do not advocate the use of our approach to replace sophisticated algorithms such as these others, but rather for fine-tuning of these approaches, or as a rapid prototyping approach for other optimization problems where one cannot afford the research effort required to devise more sophisticated techniques.

Because the stochastic and adaptive approach we use has not undergone much fine tuning, we believe there is considerable room for our ASFM algorithm to improve. For instance, the stochastic pass in SFM could be further explored, and the means of adapting the cutoff in ASFM could be put on a more solid statistical foundation. Local stochastic perturbation (similar to our stochastic pass) has been explored, for instance, in satisfiability [9], and it appears that well-motivated perturbations are superior to random ones. Moreover, our algorithm has an embedded adaptive component that illustrates, albiet in a simple context, the use of learning within constraint satisfaction. Simple adaptive schemes are beginning to see more use in search [7], as authors attempt to fine-tune key algorithm parameters dynamically.

**Acknowledgments**

Many thanks to John Lillis and Jeremy Frank for their guidance. This work was supported in part by the Semiconductor Research Corporation under Contract 1997-DC324 and by the Digital Equipment Corporation.

## References

[1] E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley and Sons, 1995.

[2] D. Aldous and U. Vazirani. “Go with the winners” algorithms. In *Proc. 35th Symp. Foundations of Computer Sci.*, pages 492–501, 1994.

[3] C. J. Alpert and A. B. Kahng. Recent developments in netlist partitioning: A survey. *Integration: the VLSI Journal*, 19(1–2):1–81, 1995.

[4] G.L. Bilbro, W.E. Snyder, and R.C. Mann. Mean-field approximation minimizes relative entropy. *Jnl. Opt. Soc. of America A*, 8(2):290–295, 1991.

[5] S. Dutt and W. Deng. A probability-based approach to VLSI circuit partitioning. In *Proc. Design Automation Conference*, 1996.

[6] S. Dutt and W. Deng. VLSI circuit partitioning by cluster-removal using iterative improvement techniques. In *IEEE/ACM Int. Conf. on CAD*, pages 194–200, 1996.

[7] J. Frank. Learning short term weights for GSAT. In *IJCAI-97*, Nagoya, Japan, 1997.

[8] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian relation of images. *IEEE Trans. on PAMI*, 6:721–741, 1984.

[9] I.P. Gent and T. Walsh. Towards an Understanding of Hill-climbing for Procedures for SAT. In *11th Nat. Conf. on AI*, MIT Press, 1935.

[10] F. Glover. Tabu search – part I. *ORSA Journal on Computing*, 1989.

[11] S. Hauck and G. Borriello. An evaluation of bipartitioning techniques. 1996. Submitted.

[12] D.S. Johnson and L.A. McGeoch. The travelling salesman problem: A case study in local optimization. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*. John Wiley and Sons, 1995.

[13] J. Li, J. Lillis, and C.-K. Cheng. Linear decomposition algorithm for VLSI design applications. In *IEEE Int. Conf. on CAD*, pages 223–228, 1995.

[14] R.M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. TR CRG-TR-93-1, Dept. of Computer Science, University of Toronto, 1993.

[15] S.Y. Reddy, K.W. Fertig, and D.E. Smith. Constraint management methodology for conceptual design tradeoff studies. In *ASME Design Eng. Tech. Conf. and Computers in Eng. Conf.*, Irvine, California, 1996.

[16] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. D. Reidel, Dordrecht, 1987.

## Appendix: Compilation and parameter details

Unfortunately our compiler (g++ version 2.7.2) would not optimize Dutt and Deng’s code without producing memory errors so we ran their code unoptimized. To be fair, we therefore shrank their times by a factor of 0.7 when making timing comparisons. Our own code is written in C and was optimized with “gcc -O2”. PROP was run with the following parameters: algorithm=0 iter=1 ratio=0.45 initProb=0.98 upper=1.75 lower=-1.75 min=0.4, recommended by Dutt.