Simulated Annealing Algorithms: An Overview

Rob A. Rutenbar

Introduction

Simulated annealing is a technique for combinatorial optimization problems, such as minimizing functions of very many variables. Because many real-world design problems can be cast in the form of such optimization problems, there is intense interest in general techniques for their solution. Simulated annealing is one such technique of rather recent vintage (it was introduced in 1982 by Kirkpatrick et al. [1]) with an unusual pedigree: it is motivated by an analogy to the statistical mechanics of annealing in solids. To understand why such a physics problem is of interest, consider how to coerce a solid into a low energy state. A low energy state usually means a highly ordered state, such as a crystal lattice; a relevant example here is the need to grow silicon in the form of highly ordered, defect-free crystals for use in semiconductor manufacturing. To accomplish this, the material is *annealed*: heated to a temperature that permits many atomic rearrangements, then cooled carefully, slowly, until the material freezes into a good crystal. Simulated annealing techniques use an analogous set of "controlled cooling" operations for nonphysical optimization problems, in effect transforming a poor, unordered solution into a highly optimized, desirable solution. Thus, simulated annealing offers an appealing physical analogy for the solution of optimization problems, and more importantly, the potential to reshape mathematical insights from the domain of physics into insights for real optimization problems.

Interest in such solution techniques is intense because few important combinatorial optimization problems can be solved exactly in a reasonable time. Many optimization problems arising in practice are *NP-complete* [2]: all known techniques for obtaining an exact solution require an exponentially increasing number of steps as the problems become larger. Hence, emphasis has been directed toward heuristic techniques for solving these important problems. The difference between a heuristic and an algorithm is that a heuristic is not guaranteed to get the optimum answer; heuristics are designed to give an acceptable answer for typical problems in a reasonable time. In practice, however, the terms *algorithm* and *heuristic* are often used interchangeably. Moreover, simulated annealing is not an algorithm in the sense that it prescribes a mechanical sequence of computations to solve a specific problem, e.g., in the sense that Gaussian elimination is an algorithm for matrix inversion. Rather, annealing is a *strategy* or *style* for solving combinatorial optimization problems. Specifically, simulated annealing is a heuristic solution strategy applicable to a wide variety of optimization problems. Nevertheless, we will still speak of "annealing algorithms," knowing that this simply means a solution technique implemented in the style of annealing.

Since its introduction, annealing has diffused rapidly into many application areas. It is impossible to survey all of them here. What we seek to provide is a brief overview of simulated annealing ideas and applications in the area of integrated circuit (IC) design and to survey some current research problems and current perspectives on the technique. We will begin this article with a brief introduction to the actual mechanics of simulated annealing and employ a simple example from an IC layout to illustrate how these ideas may be applied. Next, we will illustrate the complexities and trade-offs involved in attacking a realistically complex design problem by dissecting two very different annealing algorithms for VLSI chip floorplanning. We will then briefly survey several current research problems aimed at telling us more precisely how and why annealing algorithms work. With this technical background, we will then discuss some philosophical issues raised by the introduction of annealing. It is certainly unusual for an optimization technique to arouse the passions of engineers and algorithm designers, but remarkably enough, such has been the case with annealing. These flames continue to burn, although, now, less brightly, fanned originally by some remarkable (and, unfortunately, unattainable) expectations surrounding the introduction of the technique. We survey these issues and offer some opinions here. Finally, we present some concluding remarks.

Basic Mechanics and a Simple Example

For our purposes, a *combinatorial optimization problem* is one in which we seek to find some configuration of parameters $\overline{X} = (X_1, X_2, \ldots, X_N)$ that minimizes some function $f(\overline{X})$. This function is usually referred to as the *cost* or *objective* function; it is a measure of goodness of a particular configuration of parameters. Realistic design problems may require many parameters and a complex cost function. Consider, for example, deciding the placement of components on the surface of an IC in an optimal way. We may seek to maximize the ability to route wires to interconnect these components, minimize the overall chip area, maximize the manufacturing yield of the chip, minimize the deviation from specified timing constraints, and so forth. The cost function may be very sophisticated, and the number of parameters large: perhaps 10^3 to 10^5 variables to specify positions for each component.

Heuristic strategies for solving such problems come in several styles. Sometimes *constructive* heuristics can be found, which build up a good answer directly, piece by piece. Of more interest to us are *iterative improvement* strategies, which attempt to perturb some existing, suboptimal solution in the direction of a better, lower-cost solution. The idea can be neatly illustrated with a "balls and hills" diagram, as shown in Fig. 1. We regard all the values of $f(\overline{X})$, taken over

8755-3996/89/0100-0019$1.00 © 1989 IEEE

all legal configurations (the *configuration space*) of the N parameters $\bar{X}$, as defining a cost surface. In the figure, we plot this schematically for $N = 1$, i.e., a single parameter, as a set of hills and valleys in the cost surface. The ball represents the current configuration we plan to perturb. In practice, iterative improvement algorithms often start with a random initial configuration, or, when possible, with a heuristically constructed initial configuration that is not as costly as a random solution.

To find a good solution, we try to perturb the known solution to improve it. From the diagram, an obvious approach is to explore easily reached neighboring configurations and to select the one with least cost, i.e., the one giving the most improvement. In practice, we attempt some small random perturbation to the configuration that yields a nearby solution. This process can continue starting from the new configuration until no further improvements are obtained, at which point the process terminates. This strategy seems reasonable, but it has a serious problem: it is easily trapped in local minima, solutions that look good in some small neighborhood of the cost surface but are not necessarily the global optimum. Standard iterative improvement is a *downhill-only* style; in Fig. 1, each new perturbation moves to a configuration *downhill* from the previous one, thus becoming trapped in local minima. In practice, one scheme to overcome this is simply to try many random initial configurations, improve each, and use the best answer found. However, for very large problems, the computational expense is great here, the number of random starts needed to adequately sample the cost surface is unreasonable, and we still have no guarantees of finding a good answer.

Simulated annealing offers a strategy very similar to iterative improvement, with one major difference: annealing allows perturbations to move uphill in a controlled fashion. We now refer to individual perturbations as *moves*. Because each move can now transform one configuration into a *worse* configuration, it is possible to jump out of local minima and potentially fall into a more promising downhill path. However, because the uphill moves are carefully controlled, we need not worry about getting close to a good, final solution, only to randomly jump uphill to some far worse one.

The relevant analogy here is physical annealing of a solid. To coerce some material into a low energy state, we heat it, then cool it very slowly, allowing it to come to thermal equilibrium at each temperature. Simulating this process is (in hindsight) very similar to a combinatorial optimization task. For this physical system, the goal is to find some arrangement of atomic particles (a configuration) that minimizes the energy (cost) of the system. The basic requirement for simulating this process is the ability to simulate how the system reaches thermodynamic equilibrium at each fixed temperature in the *schedule* of decreasing temperatures used to anneal it. Toward this end, the Metropolis algorithm, developed in 1953 (see [1]), can be employed. The algorithm is shown in Fig. 2.

The idea, as in iterative improvement, is to propose some random perturbation, such as moving a particle to a new location, then evaluate the resulting change in energy ΔE. If the energy is reduced, $\Delta E < 0$, the new configuration has lower energy and is accepted as the starting point for the next move. However, if the energy is increased, $\Delta E > 0$, the move *may* still happen: the new, higher energy configuration may be acceptable. In physical systems, jumps to higher energy actually do happen, but they are moderated by the current temperature T. At higher temperatures, the probability of large *uphill* moves in energy is large; at low temperatures the probability is small. The Metropolis algorithm models this with a Boltzmann distribution: the probability of an uphill move of size ΔE at temperature T is $Pr[\text{accept}] = e^{-\Delta E/T}$. In practice, this probabilistic acceptance is achieved by generating a uniform random number R in [0,1] and comparing it against the threshold Pr[accept]. Only if $R < Pr$[accept] is the move accepted. Thus, very probable moves can be rejected, and very improbable moves can be accepted—at least occasionally. By successively lowering the temperature and running this algorithm, we can simulate the material coming into equilibrium at each newly reduced temperature, and thus effectively simulate the physical annealing.

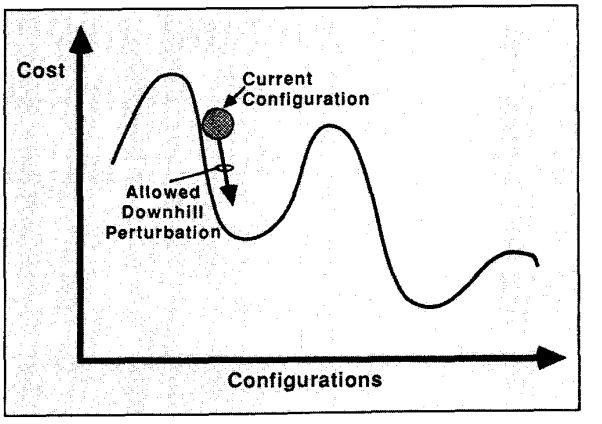

Fig 1 Configuration space: balls and hills.

We can readily apply this simulated annealing procedure to arbitrary combinatorial optimization problems. With respect to standard iterative improvement, the only addition is the notion of a temperature parameter. In physical systems, temperature has a physical meaning; in arbitrary nonphysical optimization tasks, the temperature is simply a control parameter. The idea is to employ a *cooling schedule*, a sequence of decreasing temperatures, to moderate the acceptance of uphill moves over the course of the solution.

```
M = number of moves to attempt;
T = current temperature;
for m=1 to M {

    Generate a random move, e.g., move a particle;
    Evaluate the change in energy, ΔE;
    if ( ΔE  <  0 ) {
        /* downhill move: accept it */
        accept this move, and update configuration;
    }
    else {
        /* uphill move: accept maybe */
        accept with probability P  =  e^(-ΔE/T);
        update configuration if accepted;
    }
} /* end for loop */
```

Fig 2 Metropolis algorithm.

Initially, this *effective* temperature parameter is high enough to permit an aggressive, essentially random search of the configuration space. Most uphill moves are allowed. As the temperature cools, fewer uphill moves are allowed; we tend to improve the value of the cost function here, but some local minima can also be avoided. At the coldest temperatures, the solution is close to freezing into its final form, and very few disruptive uphill moves are permitted. In this temperature regime, annealing closely resembles standard downhill-only iterative improvement.

We can illustrate these ideas concretely with a simple example drawn from an IC layout. The task is to place components on the surface of an IC so as to optimize subsequent wirability (see Fig. 3). The IC itself is modeled as a grid, where each grid point can hold one module. The input circuit to be "placed" is a set of interconnected modules, in this case, each with a maximum of four electrical terminals. This abstraction is a reasonable, though simple, model of placing gates on a gate array chip. Each set of module terminals to be wired together forms a *net*. We must place the modules and optimize the wirability of all the required nets.

An annealing algorithm for this task needs four basic components:

1. **Configurations:** a model of what a legal placement (configuration) is. These represent the possible problem solutions over which we will search for a good answer.
2. **Move set:** a set of allowable moves that will permit us to reach all feasible configurations and one that is easy to compute. These moves are the computations we must perform to move from configuration to configuration as annealing proceeds.
3. **Cost function:** to measure how good any given placement configuration is.
4. **Cooling schedule:** to anneal the problem from a random solution to a good, frozen, placement. Specifically, we need a starting hot temperature (or a heuristic for determining a starting temperature for the current problem) and rules to determine when the current temperature should be lowered, by how much the temperature should be lowered, and when annealing should be terminated.

For this task, a legal configuration is just an assignment of modules to grid locations, one module per grid site. A simple move set can be composed solely of pairwise module swaps. It is easy to see that any target configuration can be reached from any starting configuration using only pair swaps. The cost function we choose is simply the estimated total wirelength. This is simplistic but workable, i.e., a real placement algorithm might include more effects like wiring congestion. We approximate the length of each net as one-half the perimeter of the smallest rectangle that encloses all the modules connected to that net. This metric is an optimistic lower bound on the amount of wire actually required for each net. It has the advantage that it is easy to compute incrementally, after any pair swap. The cooling schedule will be the simplest possible: $T_{new} = \alpha T_{old}$, $\alpha < 1$, where the initial starting temperature and cooling rate α are determined empirically to give good results. In particular, the starting temperature is chosen to give an *accept rate* (the fraction of accepted moves at the current temperature) of 0.95. At each temperature, we perform $100 \cdot M$ moves, for M modules being placed. The stopping criterion is to terminate annealing when the cost improvement seen across three successive temperatures is sufficiently small, e.g., less than 1 percent.

The above design decisions fully characterize this annealing algorithm. However, one additional note of sophistication is worth including. This is the idea of *range limiting*. Recall that at colder temperatures, large uphill moves are unlikely to be accepted. Nevertheless, their evaluation takes time, and it is worthwhile attempting to bias the generation of random moves in favor of those more likely to be accepted and thus advance the placement toward its final configuration. We accomplish this by empirically restricting the distance between modules in a proposed pair swap as the temperature cools. Thus, near freezing, we simply do not attempt across-the-chip moves that are likely to be rejected.

The algorithm described above has been implemented to illustrate its execution on a typical problem. For a placement problem with approximately 800 gates, a plot of cost versus temperature for a typical annealing run appears in Fig. 4. (Note that the plot is read right to left; annealing proceeds from hot to cold temperatures.) At hot temperatures, the placement is essentially randomized. As cooling proceeds, cost begins to fall rapidly over a sequence of temperatures. Finally, improvement levels off as we near a good, final solution. The algorithm terminates when the cost value is essentially "flat enough" for a few temperatures. Note that the wirelength is reduced dramatically, from around 17,000 for the initial solution to around 2,300 at freezing.

Note also that annealing algorithms are *not* deterministic and will produce different answers each time they are run,

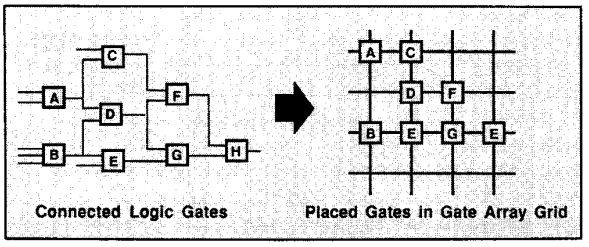

Fig 3 Gate array placement model.

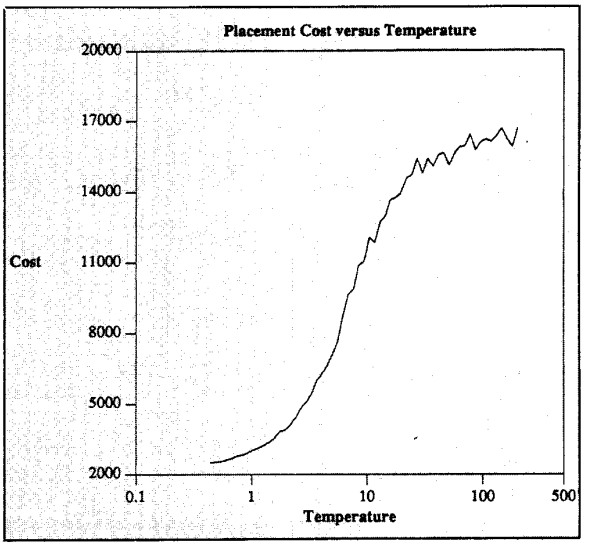

Fig 4 Placement: cost versus temperature.

even on the same problem. This is because of the probabilistic nature of choosing moves and accepting uphill moves. In particular, there is *no* guarantee of getting precisely the optimum answer in any annealing algorithm or even of getting the same answer on multiple runs. This is illustrated for our placement problem in Fig. 5. For about 20 placement runs, the figure plots the final placement cost attained versus the final temperature at which each run terminated. The mean and variance for these data are also shown. (Although some tuning of the cooling schedule may affect the magnitude of the spread of these data, it will never eliminate this spread entirely.) What annealing really offers here is *some* probability of getting out of *some* local minima; this is not the same as a guarantee of finding the optimum. The data in Fig. 5 clearly show that some placements froze earlier than others, some at a more costly local minimum than others.

A Real Application: Chip Floorplanning

The placement algorithm presented in the previous section is simplistic but not all that different from placement algorithms in actual use; it just lacks some extensions to handle nuances of real technologies. However, this simplicity may be misleading, since it may suggest that the actual design of the components of an annealing algorithm—the configuration space, move set, cost function, cooling schedule—is also simple, straightforward, and perhaps even inevitable. The placement algorithm is a good first example *because* it can be annealed simply, but it is an inappropriate generalization here to regard all annealing algorithms as simple. Not all applications admit annealing solutions, and not all annealing solutions are obvious. The purpose of this section is to make apparent the fact that the design of annealing algorithms, like that of all good algorithms, requires insight and judgment. We illustrate this by comparing two very different annealing-based solutions to a related IC layout task: the floorplanning problem.

The floorplanning task is also a placement task, but now the modules to be placed on the chip may have different shapes and can no longer be modeled as dimensionless points. Let us regard them as arbitrary rectangular shapes (this is itself a simplification, but a reasonable one). The overall goal is to minimize chip size and total wirelength.

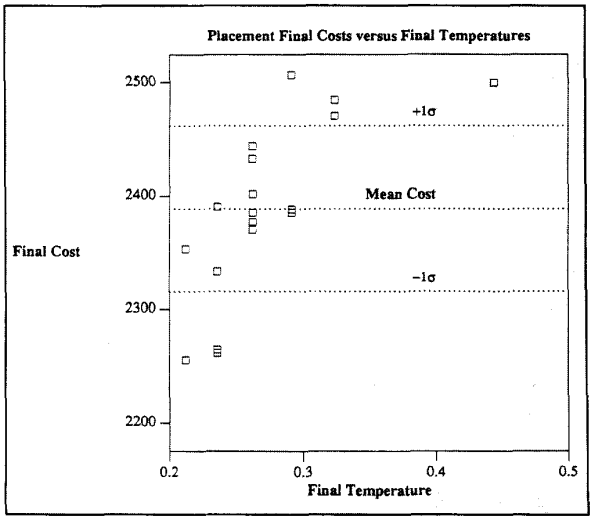

Fig 5 Placement: nondeterminism in results.

It should be apparent that many of the design decisions for the simple placement algorithm are not workable here. For example, we cannot assume modules will simply reside on a fixed grid, since they now have different shapes. Moreover, we cannot just swap pairs of modules to perturb the placement: if a small module packed tightly among its neighbors is swapped with a much larger module, the swap will result in an illegal placement with the larger module overlapping its new neighbors. The key problems are to design configurations, moves, and cost functions cleverly so that we ultimately arrive at a compact, realizable floorplan. Here are two very different solutions to these problems. One approach is similar to our own previous placement algorithm; we refer to it as a *direct* solution because it has the same flavor of rearranging the physical location of modules on the silicon surface and measuring the quality of the result. The other approach, which we refer to as an *indirect* style, builds first a graph-based, topological representation of the floorplan and anneals this *abstract* representation; a subsequent mapping process is required to turn this abstract form into a real layout. Each style has its advantages and disadvantages.

The direct style appears to have originated with Jepsen and Gelatt of IBM [3], where it was originally formulated for gate array macro cell placement. It was extended to more general placement problems by Sechen and Sangiovanni-Vincentelli of U.C. Berkeley [4]. The essential problem is that relocating or swapping modules of varying shapes cannot be guaranteed to produce a layout without overlaps. The key idea to solve this is simply to *allow* these overlaps, but *penalize* them in the overall cost function. Thus, the move set can consist of moving modules directly, without regard for overlap. The layout configurations thus visited during annealing are not necessarily realizable as real chips. Such configurations are often referred to as *erroneous* or *incorrect intermediate states*. The idea is that by allowing such configurations during annealing, we greatly simplify the design of the move set. The cost function must now account for "real" objectives such as wirelength and chip area, which should be minimized, as well as module-to-module overlap, which must be driven to zero in the final floorplan.

An example of a floorplan evolving during a direct-style annealing run appears in Fig. 6. This floorplan was produced by PASHA, a floorplan tool developed by R. Jayaraman at Carnegie-Mellon University (CMU) as part of a study of parallel annealing algorithms [5]. Note that, for this run, the overall topology of the floorplan evolves first, and overlap rather quickly evaporates. As cooling proceeds, the final packing of this topology is found.

The second, *indirect*, style of solution relies on the idea of building an abstract topological representation of the floorplan. Topological in this sense means knowing simply which modules are above, below, left, or right of each module in a specific floorplan. Graph structures are amenable to this task. The key idea here is to *avoid* incorrect intermediate states. The trick is to design the layout representation and move set so that each move applied to a legal floorplan produces a new legal floorplan. These requirements are the reason the graph-based topological representation is attractive: layout modifications that appear complex when viewed as individual module movements may re-

quire only a relatively simple modification to the abstract graph.

To be concrete, we consider the indirect-style algorithm of Wong and Liu of the University of Illinois [6]. This algorithm relies on a widely used topological representation for layouts called a *slicing tree*. An important point is that not all legal floorplans can be represented with a slicing tree; only those with a *slicing structure* can be so represented. Informally, a slicing structure is a layout that can be arrived at by repeatedly bisecting a rectangle, as illustrated in Fig. 7. The constraint is that each slicing cut attempted must completely bisect one of the rectangles produced by an earlier cut. A slicing tree is a compact representation for this class of layouts. The nodes at the bottom of the tree (the leaf nodes) represent modules themselves; the intermediate nodes represent slicing cuts.

The algorithm of Wong and Liu uses a slightly specialized form of slicing tree to define the configuration space for their floorplanner. The advantage of this design choice is that a good move set can be built around a few types of moves that simply rearrange the tree, not the (x,y) locations of individual modules. Recall that by moving nodes in the tree, we change the order and location of slicing cuts, thus changing the floorplan. An example of such a move, and the resulting floorplans (from [6]), is illustrated in Fig. 8. After each move, the tree can be quickly manipulated to determine the total size of the resulting layout, the new location of each module, and, hence, the total estimated wirelength. Indeed, the cost function for this floorplanner includes only wirelength and area terms but no overlap term since only legal floorplans are shown.

Each of these floorplanning algorithms required a clever, elegant insight to construct an annealing solution, and yet these two solution styles represent very different annealing design choices, each with their own advantages and disadvantages. However, our goal is not to determine the best style but to argue that a difficult problem may require extreme cleverness to recast it in a form suitable for an annealing algorithm. Annealing techniques are not a panacea for combinatorial optimization problems: hillclimbing does not obviate the need for careful thought in algorithm design.

Research Issues

In this section, we present a brief survey of issues at the forefront of research on simulated annealing techniques and their applications in IC design. Given the very recent vintage of the ideas underlying this solution technique, there remain a great many unanswered questions with both practical and theoretical importance; we merely touch on some of these. Although the range of research in annealing is broad, we informally partition it into three components: applications of annealing, acceleration of annealing, and foundations of annealing.

Research concerning applications of annealing strives to use annealing to improve solutions to existing problems and also to apply the technique to new, unsolved problems. As with any general combinatorial optimization technique, interest at first was intense as practitioners sought

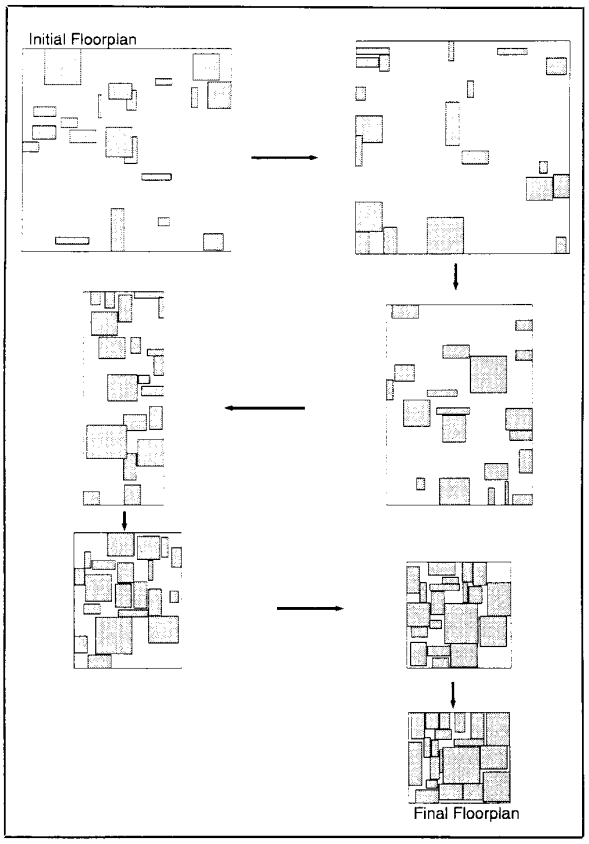

Fig 6 Evolution of direct-style floorplan in PASHA.

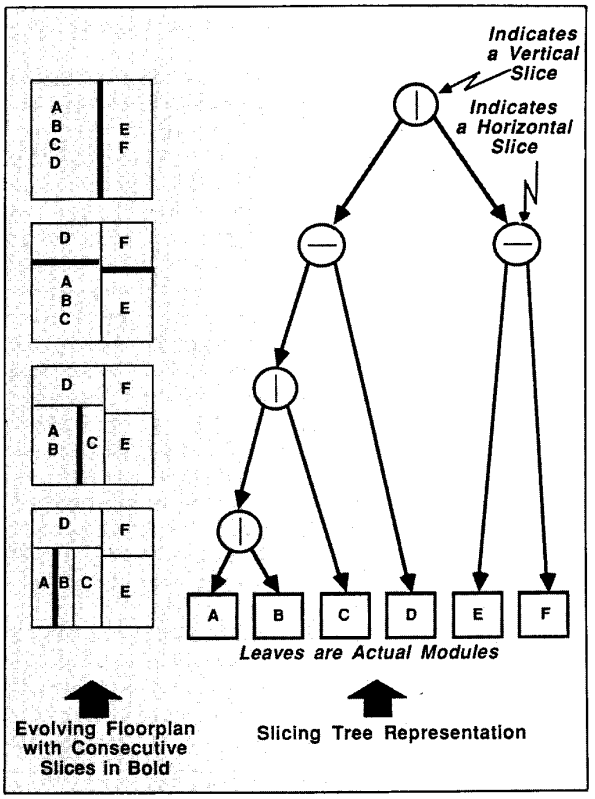

Fig 7 Slicing floorplans and slicing trees.

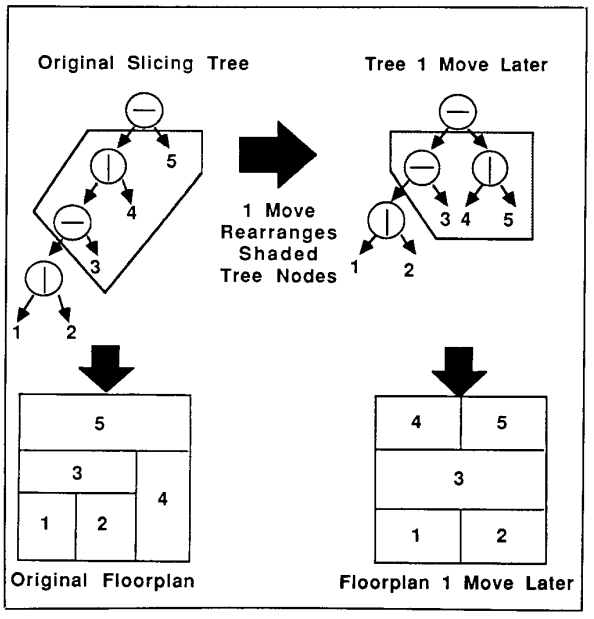

Fig 8 Evolution of indirect-style slicing floorplan.

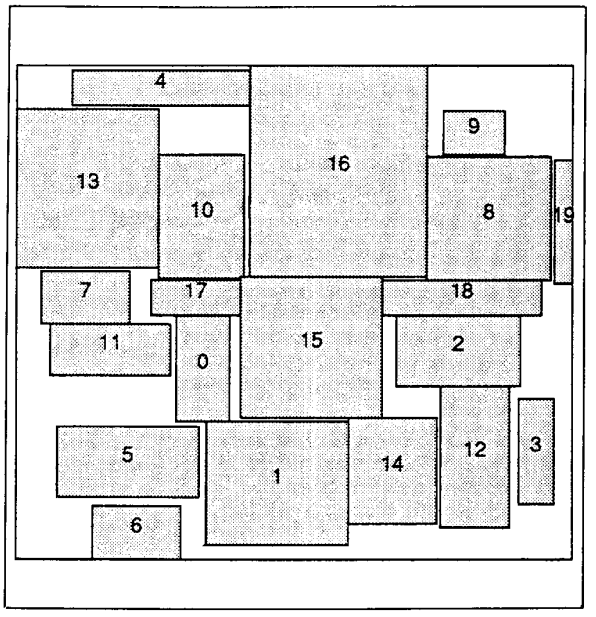

Fig 9 Parallel floorplan results from PASHA.

to understand for which types of problems the technique was most well suited. Applications in IC design are too numerous to survey here; [7] has a fairly complete survey of this field. We do note, however, that in IC design the dominant applications for annealing currently are layout tasks. Interestingly enough, it turns out that these problems simply have the right *character* to anneal well; we shall describe this in more detail later. A general survey of layout problems and algorithms, including annealing-based approaches, appears in [8].

Research concerning acceleration of annealing focuses on this central fact: annealing algorithms are often slow algorithms. This is because of their iterative improvement nature—many configurations need to be visited at many temperatures to reach a good solution. Although each move represents a small amount of computation, the need to compute 10^6 to 10^8 moves can still consume hours or days of CPU time. One approach to this problem is the engineering approach: improve and tune the annealing algorithms we have now. Better data structures, more clever program implementations of the computations underlying move generation and evaluation, etc., all have led to great reductions in run time. Another approach is to concentrate on *inherently* faster annealing algorithms, e.g., the indirect-style floorplanner discussed previously tends to be faster than the direct style, since it visits only legal floorplan configurations. Another approach is to make fundamental improvements in our understanding of the foundations of annealing algorithms, for example, improvements in cooling schedules or in understanding what forms of cost functions are more easily annealable.

This leads us naturally to the third focus of annealing research, foundations of annealing. Broadly, this work asks these questions: what kinds of problems can be annealed, and for these problems, what is the optimal strategy for annealing them? A major, open question is to characterize the sorts of problems amenable to annealing. Not all combinatorial optimization problems can be annealed to give satisfactory solutions (e.g., the time taken to get a decent answer may prove to be unreasonable). It is widely believed that something about the basic *structure* of a particular configuration space—its landscape of multidimensional hills and valleys—determines whether annealing is viable. A simple example clarifies the problem. A configuration space that is mostly smooth, with gradually flowing hills and valleys, is relatively easy to anneal: once we find a good global downhill track, we can proceed on it, climbing over the occasional hill or bump as we approach a good solution. In contrast, consider a mostly flat landscape with numerous, densely packed gopher holes, each of widely varying depth, but some of which lead to the very best solutions. Such a problem is probably impossible to anneal, since moves will keep falling into these dead-end holes, and as the temperature cools, it will become impossible to climb out of them, thus trapping the solution in a bad local minimum. The question here is how to characterize in formal, rigorous mathematical language these prosaic notions of landscapes that are "too irregular." Approaches to this problem—both for narrowly defined, specific classes of problems or for more general problems—are the subject of active investigation.

Another fundamental question concerns the convergence of annealing algorithms and asks whether it is possible formally to prove that an annealing algorithm will converge to an optimal answer. It turns out that, by making certain simplifications, annealing algorithms can be modeled probabilistically (actually, as Markov chains); in fact, convergence can be proven. However, these technical proofs typically show that annealing converges *asymptotically, in probability*. In other words, if we perform enough (maybe infinitely many) moves, the *probability* that we have found a global minimum can be made as close to 1 as we like. These often misunderstood results suggest that annealing is on a firm footing in a fundamental sense, but they do *not* provide any practical guarantee of convergence in a real problem.

A related area of research is optimal cooling schedules. In our simple gate array placement example, we used the

crudest possible schedule: $T_{new} = \alpha T_{old}$, for some constant factor $\alpha < 1$. This is workable but almost never optimal. Better schemes rely on statistics observable as annealing proceeds (e.g., the sampled mean and variance of cost values seen at the current temperature) to adapt the schedule to the annealing. In this case, optimal means *fast*: we want the fastest possible cooling schedule so that we will perform as few moves as possible across the sequence of cooling temperatures. The problem here is to balance the desire for speed with the need to ensure a solution of good quality. As with the convergence results, work on schedules tends to rely on many technical ideas derived from probabilistic (again, often Markov) models of annealing, with occasional insights from statistical mechanics. A good reference surveying theoretical work on both convergence and cooling problems is [7].

Another area of very recent, intense work cuts across all three basic areas—applications, acceleration, foundations—and focuses on *parallel* annealing. The growing availability of commercial multiprocessor machines suggests that, perhaps, we can mitigate some of the execution time penalties associated with annealing approaches. Approaches vary, but the basic idea is to perform many moves in parallel, thus exploring more of the configuration space in less time. Unfortunately, parallel moves may interact; e.g., the wirelength being computed by one processor is affected by the fact that another processor is moving a module connected to this wire. The problem is that it is usually too expensive in interprocessor communication to have each move broadcast minute details of its progress to all other active moves. Thus, work has focused on managing the errors that occur when parallel moves interact with each other. Remarkably enough, experiments with parallel annealing algorithms show that they are exceedingly tolerant of such errors. Reasonable results have been obtained for a variety of VLSI layout problems run on a variety of multiprocessors, all with promising speedups over their counterpart sequential algorithms. Figure 9 shows results from a parallel version of the PASHA floorplanner described in the previous section. This version runs on a 16-processor message-passing hypercube multiprocessor [5]. This result is aggressively parallel, since it uses 8 processors to floorplan only 20 modules. Work is under way on the many new problems—analogous to those for serial annealing—posed by these parallel algorithms: convergence in the face of errors, parresist simulated annealing (recall the previous discussion about badly irregular configuration landscapes). Moreover, there are even applications where annealing gives reasonable results but is outperformed, either in execution time or solution quality, by more conventional heuristic approaches. Even in the applications where it works well, there is no substitute for cleverness on the part of the annealing algorithm designer (recall our comparison between direct and indirect floorplanning algorithms). Coupled with misunderstandings about the (nonexistent) practical guarantees provided by theoretical convergence results, these early beliefs attributed many desirable, but unattainable, properties to annealing. In fact, annealing is simply another technique for solving combinatorial optimization problems, superior to some approaches for some problems, inferior to others. We regard the major consequence of annealing as its introduction of an exceedingly novel—and practically useful—framework for studying and solving a wide spectrum of optimization problems.

Perhaps the more interesting philosophical issue concerns the early tensions between proponents of annealing and proponents of more traditional, "tuned" heuristic approaches. More traditionally, solving an optimization problem required uncovering some essential, subtle structure that underlies the problem, then using this knowledge to craft a heuristic solution tuned to the nuances of this structure. Proponents of this approach have argued that, by this standard, annealing techniques are essentially inelegant, brute-force searches through configuration space. Moreover, they argue, annealing exposes no such fundamental structural insights about the problems to which it is applied. Proponents of annealing (and this author freely admits his biases in this direction) argue that, although heuristics are certainly desirable, they are not available for all problems of interest to solve. When both clever heuristics and annealing techniques work for a specific problem, the choice of which to use can be made based on objective measurements, e.g., execution time and solution quality. However, for many hard problems, we have no completely successful heuristics. Indeed, there is growing consensus that annealing is especially well suited to tackling problems best described as "dirty," i.e., problems with either numerous, contradictory constraints, or complex, baroque cost functions. The best tuned heuristics are usually formulated for clean, rather abstract forms of problems. In contrast, when move sets and cost functions can be adequately constructed, annealing formulations can attack some of these messy problems directly.

As annealing techniques have matured, however, some of this tension seems to be evaporating. It is now widely known that annealing is not a universal solution to all problems, but it is well established as a successful solution method for many important classes of problems, e.g., for layout problems in IC design. Moreover, as annealing becomes yet another tool in the algorithm designer's toolbox, it has begun to appear in concert with other algorithms. We expect to see more of these kinds of mixed annealing/heuristic solutions as annealing continues to mature.

Philosophical Issues

It is interesting for the introduction of an algorithm to generate much controversy, but such was the case with simulated annealing. Much of this controversy seems to have died down of late as the technique has matured and proven its utility in many applications. But it is still worthwhile to examine some causes and consequences of this controversy. We briefly survey such philosophical issues and offer some opinions in this section.

One major issue was the (incorrect) belief on the part of some early practitioners that annealing was a panacea for all optimization problems. Specifically, this view held that all problems were annealable and that all annealing algorithms were straightforward, even mechanical, to design. Unfortunately, experimental evidence proved otherwise. Just as some materials resist physical annealing, some problems

Conclusions

Since its introduction in 1982, simulated annealing has diffused widely into many diverse applications. In IC de-

sign, it has been especially successful in layout tasks. Research on both new applications of annealing and on the theoretical foundations of annealing is progressing on many fronts. Although not a panacea for all combinatorial optimization problems, annealing has already established itself as a mature, very useful tool to have in any algorithm designer's toolbox.

Acknowledgments

I am grateful to Ron Rohrer and Rajeev Jayaraman of CMU, and Jonathon Rose of Stanford for helpful comments on earlier drafts of this paper.

References

[1] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[2] E. M. Reingold, J. Nevergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, 1977.

[3] D. W. Jepsen and C. D. Gelatt, Jr., "Macro Placement by Monte Carlo Annealing," *Proc. IEEE Intl. Conf. on Computer Design*, pp. 495–498, Nov. 1984.

[4] C. Sechen and A. L. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE J. Solid-State Circ.*, vol. 20, pp. 510–522, 1985.

[5] R. Jayaraman and R. A. Rutenbar, "Floorplanning by Annealing on a Hypercube Multiprocessor," *Proc. IEEE Intl. Conf. on CAD*, pp. 346–349, Nov. 1987.

[6] D. F. Wong and C. L. Liu, "A New Algorithm for Floorplan Design," *Proc. 23rd ACM/IEEE Design Automation Conf.*, pp. 101–107, June 1986.

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

[8] A. Sangiovanni-Vincentelli, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, Eds., Martinus Nijthoff Publishers, pp. 113–196, "Automatic Layout of Integrated Circuits," 1987.

Rob A. Rutenbar received the B.S. degree in electrical and computer engineering from Wayne State University, Detroit, in 1978, and the M.S. and Ph.D. degrees in computer engineering (CICE) from the University of Michigan, Ann Arbor, in 1979 and 1984, respectively.

In 1984, he joined the faculty of Carnegie-Mellon University, Pittsburgh, Pennsylvania, where he is currently an Assistant Professor of Electrical and Computer Engineering, and of Computer Science. His research interests include VLSI layout algorithms, parallel architectures and algorithms for CAD, and knowledge-based approaches to VLSI design, in particular, automatic synthesis of CAD software and automatic synthesis of analog circuits.

In 1987, Dr. Rutenbar received a Presidential Young Investigator Award from the National Science Foundation. At the 1987 IEEE-ACM Design Automation Conference, he received a Best Paper Award. In 1987, he also received the George Tallman Ladd Award for Excellence in Research from the College of Engineering at Carnegie-Mellon. Dr. Rutenbar is a member of ACM, Eta Kappa Nu, Sigma Xi, and AAAS.