Nested annealing: a provable improvement to simulated annealing

Sanguthevar Rajasekaran *
Aiken Computing Lab., Harvard University, Cambridge, MA 02138, USA

John H. Reif *
Dept. of Computer Science, Duke University, USA

Abstract

Simulated annealing is a family of randomized algorithms for solving multivariate global optimization problems. Empirical results from the application of simulated annealing algorithms to certain *hard* problems including certain types of NP-complete problems demonstrate that these algorithms yield better results than known heuristic algorithms. But for the worst case input, the time bound can be exponential. In this paper, for the first time, we show how to improve the performance of simulated annealing algorithms by exploiting some special properties of the cost function to be optimized. In particular, the cost functions we consider are *small-separable*, with parameter $s(n)$. We develop an algorithm we call "Nested Annealing" which is a simple modification of simulated annealing where we assign different temperatures to different regions. Simulated annealing can be shown to have expected run time $2^{\Omega(n)}$ whereas our improved algorithm has expected performance $2^{O(s(n))}$. Thus for example, in many vision and VLSI layout problems, for which $s(n) = O(\sqrt{n})$, our time bound is $2^{O(\sqrt{n})}$ rather than $2^{\Omega(n)}$.

1. Introduction

1.1. Heuristic algorithms

Heuristic algorithms play a vital role in solving many presumably difficult computational problems. A heuristic program uses as much information as the programmer can provide about the problem to be solved, together with some logical rules for

* Supported by NSF-DCR-85-03251 and ONR contract N00014-87-K-0310. First author's current address: Dept. of CIS, Univ. of Pennsylvania, PA 19104.

Preprint of paper appearing in Journal of Theoretical Computer Science, 99(1):157-176, 1 June 1992.

optimizing the chance of success at intermediate points in the computation. Heuristic program writing is an attempt to endow a computer with the information an intelligent human being would use as he gropes his way though a particular problem.

Numerous heuristic techniques are in use in the following areas of computer science: (1) learning, (2) vision, (3) VLSI layout and routing, (4) combinatorial optimization etc. Even through solutions yielded by a heuristic technique may not be exact, they are acceptable approximations to the actual solutions. A heuristic algorithm may not work with the same efficiency on all the inputs and in general is problem dependent.

Some of the well-known heuristics in use are (1) analogy, (2) random search, (3) best first search, (4) hill climbing, (5) alpha-beta pruning, etc. These heuristics are best explained with an example. The graph partitioning problem is defined as follows: Given a graph $G(V, E)$. The problem is to partition V into two equal sized sets V_1 and V_2 such that the total number of edges from any node in V_1 to any node in V_2 is a minimum. This problem is known to be NP-complete. Three heuristics widely used in graph partitioning are (1) Kernighan-Lin, (2) clustering, and (3) λ-opting. All these three fall under the category of random search heuristic. Kernighan-Lin's algorithm proceeds as follows. It starts with an arbitrary partition A, B of V. A single step of the algorithm consists of interchanging a node in A with a node in B. The interchange is accepted if it results in a lower "cost". This basic step is repeated until no further improvement in the cost is possible. As one can readily see, the solution so obtained will be a local optimum.

λ-opting is a generalization of Kernighan-Lin's approach. Instead of interchanging a single element in the sets A and B at a time, λ elements are interchanged. Clustering technique works by identifying "normal clusters" in the cost matrix.

All these heuristics have the following structure in common: (1) start with an initial solution, (2) "perturb" this solution, (3) test if the perturbed solution is acceptable, (4) repeat steps (2) and (3) until no further improvement in the cost is possible.

1.2. Two kinds of analysis

Early approaches to analyzing heuristic algorithms presupposed a distribution on the space of all possible inputs. An example is Hoare's Quick-sort algorithm. Quick-sort may run for an indefinitely long time on certain inputs. But all such "bad" input permutations are only a small fraction. If we assume (which indeed Hoare does) that each input permutation is equally likely to occur, then Quick-sort is very well practical because with very high probability, the given input will not be a bad one and hence the algorithm will terminate quickly.

But the assumption on the input distribution is questionable, since the input distribution may vary quite unpredictably. This problem was rectified by Rabin [18], and Solovay and Strassen [19] by introducing randomness into the algorithm itself. An algorithm of this type is called a randomized algorithm.

Informally, a randomized algorithm (in the sense of [18, 19]) is one which bases some of its decisions on the outcomes of coin flips. We can think of the algorithm with one possible sequence of outcomes for the coin flips to be different from the same algorithm with a different sequence of outcomes for the coin flips. Therefore, a randomized algorithm is really a family of algorithms. For a given input, some of the algorithms in this family might run for an indefinitely long time. The objective in the design of a randomized algorithm is to ensure that the number of such bad algorithms in the family is only a small fraction of the total number of algorithms. If for *any* input we can find at leat $(1-\varepsilon)$ (ε being very close to 0) portion of algorithms in the family that will run quickly on that input, then clearly, a random algorithm in the family will run quickly on any input with probability $\geqslant(1-\varepsilon)$. In this case we say that this family of algorithms (or this randomized algorithm) runs quickly with probability at least $(1-\varepsilon)$. ε is called the error probability. Observe that this probability is independent of the input and the input distribution.

1.3. Simulated annealing

Simulated annealing is a general heuristic technique used to solve a wide range of problems. Simulated annealing (abbreviated as SA hereafter) is a family of randomized algorithms derived by analogy with the statistical physics of random systems. Kirkpatrick et al., [11] observed that a multivariate optimization problem can be identified with a fluid. The parameters of the optimization problem will correspond to the atoms of the fluid, and an optimal solution to the problem will correspond to a low energy state of the fluid. A procedure for bringing the fluid to a low energy state will then serve as a procedure for finding an optimal solution for the optimization problem.

One of the popular methods for bringing a fluid to a low energy state is annealing. The substance is melted first. Then, it is cooled slowly spending more time at temperatures near the freezing point to ensure that the system continues to be in equilibrium. Metropolis [11] has given a probabilistic algorithm for the computer simulation of the motion of atoms in equilibrium at any given temperature. In each step of this algorithm, the position of an atom is changed by a random amount. If the new configuration (or state) of the fluid has a lower energy, it is accepted with probability 1; otherwise, it is accepted with probability $\exp(-\Delta E/T)$, (ΔE being the difference in energy levels of the old and new configurations of the fluid). An SA algorithm is the repeated application of the Metropolis' simulation step over a sequence of decreasing temperatures starting from the melting point. An SA algorithm can readily be used to solve a combinatoric optimization problem once we define the notions of *configuration*, *displacement*, *temperature*, and *energy level* for the optimization problem.

SA algorithms perform better than known heuristic algorithms [11] on certain problem instances. One of the reasons for this is that heuristic algorithms do not allow hill climbing from a local optimal configuration whereas SA algorithms allow hill climbing with certain probability. It should be noted here that SA algorithms

need not necessarily produce a ground state (i.e., a global optimal configuration). In general, they only produce metastable states.

SA approach for optimization has been used to solve a myriad of computational problems. One of the first applications was to solve the graph partitioning and the travelling salesman problems [11]. The travelling salesman problem is defined as follows: We are given a list of N cities and a means of calculating the cost of travelling between any two cities. We must plan the salesman's route, which will pass through each city once and return finally to the starting point minimizing the total cost.

Kirkpatrick et al., [11] claim that SA gave better results than some of the previously existing heuristics on both these problems. But, recently, a more extensive experiment done by [7] reveals that SA performs much better than other heuristics on certain problem instances and does not do well on other instances. In any case, SA took considerably more time for convergence than the other heuristics that SA was compared with. Johnson et al., [7] compare the performance of SA with a simple local optimization algorithm and the Kernighan-Lin heuristic [10] for the problem of graph partitioning. Their experiments show that for certain types of random graphs, SA performs much better than the other two in terms of the quality of output produced. But SA is outclassed by the other two algorithms on other types of graphs.

Another interesting application of SA has been to VLSI layout [11]. The physical design of computer involves the problem of determining which circuits should go into which chip and assigning physical locations to the chips on a board and wiring the chips. In the implementation of [11], they solve a problem with 5000 logic gates and 200 external gates. The results obtained by SA is much better than the following simple strategy: "Randomly assign gates to the chips."

More recently, SA has been applied to learning. As one can easily see, the process of annealing naturally corresponds to a simple learning model. We can think of each atom in a fluid to be equivalent to an individual learning element (or a neuron). Each processor modifies its concept on the basis of the concepts of its neighbors and on the basis of a global control function C. After each element is given a sufficiently long time to modify its concept, this concept will converge to the correct one with high probability. SA also very much resembles the Boltzman learning model.

Some more applications where SA has been used are (1) coloring the vertices of a graph [2], (2) placement of VLSI circuits [4], (3) design of good codes [5], (4) minimum weighted perfect Euclidean matching problem [1], (5) routing and location problems [6], (6) global wiring [21] etc. In summary, SA is a general heuristic that performs well for many problems and not so well for others.

1.4. Nested annealing

We now give a simple modification of simulated annealing which is provably more efficient than simulated annealing. Simulated annealing algorithm, recall, is the repeated application of Metropolis' simulation step over a sequence of decreasing temperatures, starting from the melting point. A single step of Metropolis' simulation

imitates the behavior of a fluid (or an optimization problem) at a fixed temperature. At any time unit, the position of an atom is changed (or the value of a parameter is changed). The new configuration is accepted if it results in a lower energy else it is accepted with certain probability. When the fluid attains steady state at this temperature, the temperature of the system is changed and the Metropolis' simulation step is repeated at this new temperature.

The above algorithm when implemented as such can be shown to have a worst case input convergence time of $2^{\Omega(n)}$ (if the number of possible states of the fluid is 2^n). The reason for this large convergence time is that all the atoms of the fluid are at the same temperature at any time unit. The requirement that all the atoms should be at the same temperature at any time, presupposes that the fluid is equally likely to be at any possible state at a given time. Since SA is a random walk (of some type) on the solution space, the walk should go through each possible state at least once and hence the run time is very large.

Our idea of Nested Annealing (abbreviated as NA) is to partition the fluid into different *zones* and anneal these different zones at different temperatures. *zone* in the case of an optimization problem means a subset of the parameter set. If the temperatures of the zones are set right, the convergence time of NA will be dominated by a single zone. Since the convergence time of SA is an increasing function of the number of states it can assume, NA will indeed result in faster convergence.

What follows is an informal description of NA as applied to Optimization Problems (abbreviated as OP). We will exploit the interdependency of the parameters of the OP in a clever way. The parameters are partitioned into a tree of zones (this tree is nothing but the *separator tree* of the OP as will be explained later). Each zone uses the SA algorithm to find an optimal value for its parameters. The key idea is a displacement (i.e., change in a parameter) in any zone (e.g., zone A say) occurs only after both the children zones of zone A have converged (corresponding to the current configuration of zone A). The size of the zones decreases exponentially with increasing depth. This is what makes the run time of NA to be dominated by the root zone (details in Section 3).

How long should we run the algorithm before the system *freezes*? If there are n parameters in the OP and each parameter can take only binary values then there are 2^n possible states. We can find the globally optimal state by an exhaustive search in time 2^n. Theoretically we can prove that the above NA process will see a globally optimal state within $2^{ks(n)}$ time, with probability 1 for some fixed constant k for $s(n)$-*separable* cost functions. This is as opposed to the $2^{\Omega(n)}$ run time of SA. Even $2^{ks(n)}$ time is not practical. So in practice we stop the above NA process after a long time (that we can afford) and hope that at that time the system has obtained an optimal state. How good the state at that time will be is dependent on how good the solution space is and how long we run the process. After all, that is what heuristic algorithms are all about. For the same run time NA will provably produce a better result than SA. Even though deterministic algorithms with the same asymptotic complexity as that of NA exist [9], these deterministic algorithms are impractical

due to the large run time. In NA, since we only look for a metastable state, we can terminate the algorithm after a reasonably long time (much less than $2^{ks(n)}$), and hence NA is very well practical.

All the applications discussed for SA have small separable cost functions. Thus application of NA will result in faster convergence for these problems. The advantages of NA over SA are (1) NA algorithm has a very simple control structure, (2) NA has all the merits of SA; in particular NA is applicable to physical systems as well, (3) NA has faster convergence than SA. NA is also better than the analogous deterministic algorithms [9] for the reasons stated above.

1.5. Some definitions and notations

Consider an optimization problem on n parameters $\vec{p} = [p_1, p_2, \ldots, p_n]$. Assume each parameter can only take either 0 or 1 as its value. A *Markov chain on* $\vec{p}$ is defined as follows: It is a Markov chain with 2^n states. Each state is denoted by an n-binary vector. There is a transition arc from a node to another if the hamming distance between the two nodes (when treated as n-binary vectors) is 1. In other words, the graph of the Markov chain is a hypercube of dimension n.

If $\vec{q} = [q_1, q_2, \ldots, q_m]$, and $\vec{r} = [r_1, r_2, \ldots, r_n]$, then let $\vec{q} \circ \vec{r}$ stand for $[q_1, q_2, \ldots, q_m, r_1, r_2, \ldots, r_n]$.

Let M be a Markov chain on $\vec{p}$ $(=[p_1, \ldots, p_n])$, and let $\vec{q} = [q_1, q_2, \ldots, q_m]$ be such that $\{q_1, \ldots, q_m\} \subseteq \{p_1, \ldots, p_n\}$. We define the *restriction of* M *on* $\vec{q}$ (denoted by $M_{\vec{q}}$) as follows: Fix the values of parameters of $\vec{p}$ not in $\vec{q}$. $M_{\vec{q}}$ has 2^m states, corresponding to the 2^m different values that $\vec{p}$ can take (after fixing $(n-m)$ parameters). A transition arc from a node of M to another node of M is retained in $M_{\vec{q}}$ only if both the nodes are states of $M_{\vec{q}}$. In other words, the graph of $M_{\vec{q}}$ is a subcube of dimension m in the hypercube corresponding to M. Which subcube the graph of $M_{\vec{q}}$ corresponds to is determined by what (fixed) values that the parameters of $\vec{p}$ not in $\vec{q}$ take.

We say a Markov chain M on $\vec{p}$ has *converged* with respect to a cost function C defined on the states of M, if M has been in a state q such that $C(q)$ is optimal.

Intuitively, *small separable* graphs are those that can be split into two roughly equal parts by removing only a few vertices of the graph. More rigorously, let S be a class of graphs. The class has $f(n)$ *separator theorem* if S is closed under the subgraph relation, and there exists constants $\alpha < 1$, $\beta > 0$ such that for any n-vertex graph G in S having nonnegative vertex costs summing to no more than 1, the vertices of G can be partitioned into three sets A, B, C such that no vertex in A is adjacent to any vertex in B, neither A nor B has total cost exceeding α, and C contains no more than $\beta f(n)$ vertices.

We say a class of graphs is *small separable* if it has a small separator (i.e., $f(n)$) theorem.

If C is a function on $\vec{p}$ to be optimized, we can rewrite C as $C_1 + C_2 + \cdots + C_m$ where C_i, $i = 1, \ldots, m$ is a product of functions of the parameters. Call each C_i as

a *clause*. We can define a graph $G_C(V, E)$ corresponding to the cost function C, where V comprises of the parameters p_i, $i = 1, \ldots, n$ and the clauses C_j, $j = 1, \ldots, m$. There is an edge between a clause node and a parameter node, if the parameter occurs in that clause. We say G_C is *the graph of* C. We say C is $s(n)$ separable if G_C is $s(n)$ separable.

1.6. Organization of this paper

We describe the structure of an SA algorithm in Section 2 and also give a mathematical model for the SA process. In Section 3 we present the idea of NA in more detail. In Section 3 we also prove the worst case convergence time of NA for OP. Applications of NA are discussed in Section 4. In Section 5 we give a divide and conquer algorithm for solving any OP. NA as the reader will see is nothing but this divide and conquer algorithm with SA embodied in it. This algorithm is not only an algorithm for solving OP in its own right but also the proof of convergence for this divide and conquer algorithm serves as an alternative (and more rigorous) proof for the convergence of NA. Finally in Section 6 we discuss some extensions of NA. In particular we deduce the convergence time of NA for OP with solution spaces having some special properties.

2. Details of an SA algorithm

2.1. The algorithm

In order to apply the Metropolis simulation algorithm to solve a combinatoric OP, we need to define the notions of *configuration*, *displacement*, *temperature*, and *energy level* for the OP. Each possible assignment of values to the parameters of the OP is a configuration of the OP. Displacement in the context of an OP means changing the value of one of its parameters. Cost function of an OP corresponds to the energy function of a fluid. And finally, the temperature of an OP is simply a control parameter in the same units as the cost function.

The next step in the design of an SA algorithm is to come out with an *annealing schedule*, i.e., to determine the sequence of temperatures on which to run the Metropolis simulation, and to decide the time to be spent at each temperature. An annealing schedule may be developed by trial and error for a given problem, or may consist of heating the system enough to obviously melt it and cooling it slowly until no further changes in the system configuration occur.

Next, we present a typical SA algorithm that encaptures the above given ideas.

procedure SAnneal(X_0, T_0);
(* X_0 is the initial configuration and T_0 is the initial temperature *)
  $X := X_0$; $s := 0$;
  while (*not* frozen) *do*

```
begin
  while (not steadystate) do
  begin
    XN = generateconfiguration(X);
    if (accept(c(XN), c(X), T_s)) then X := XN
    (* c(X) is the cost corresponding to the configuration X *)
  end;
  T_{s+1} := update(T_s); s := s+1
end;

function accept(c(i), c(j), T): boolean;
  accept := false;
  if c(i) < c(j) then accept := true
  else
  begin
    r := random(0, 1);
    (* r is a pseudo-random number uniform in [0, 1] *)
    p := exp(-(c(i) - c(j))/T);
    if r ≤ p then accept := true
  end;
```

In the above algorithm, "frozen" is a boolean variable which is set *true* when no further changes in the configuration of the system occur. "steadystate" is another boolean variable that is set true (at state s) after an amount of time the system needs to attain steady state at temperature T_s. The function "generateconfiguration(X)" chooses a random configuration from out of those reachable from X. In other words, this function gives a random displacement for a parameter. "update(T_s)" produces a new temperature. This function has to be decided by trial and error. Some of the popular choices for "update" are

(1) $T_{s+1} = cT_s$, for some constant $c < 1$ [11],

(2) $T_s = \gamma/\log(s + s_0 + 1)$, $s = 0, 1, \ldots$ where γ is a constant times the radius of the underlying Markov chain [16] and s_0 is a constant.

2.2. *A mathematical model for* SA

It is easy to see that the process of simulated annealing, at a fixed temperature, can be modelled by a Markov chain whose connectivity is fully defined by the *accept* and *generateconfiguration* functions. The set of states of the Markov chain is the solution space of the OP we are interested in solving.

If our problem has n parameters $[p_1, p_2, \ldots, p_n]$, and each parameter can only take a value of either 0 or 1, then the graph of the Markov chain is a hypercube H with dimension n (as was shown in Section 1.5). Throughout this paper we will be considering only this case for our OP. But the results to be stated are general. The

only specifications of the Markov chain that remain to be given are the state transition probabilities.

The one step transition probabilities of the Markov chain are represented as weights on the edges of the graph H, and are determined by the product of the probability of generating a given state and the probability of accepting it. If we assume that when the Markov chain is at state i, each one of the $N(i)$ neighbors of i is equally likely to be generated next by *generateconfiguration*, then, $P_{ij}(T)$, the transition probability from state i to state j, is given by

$$P_{ij}(T) = \begin{cases} 0 & \text{if } j \notin N(i) \,\&\, j \neq i, \\ 1/|N(i)|\ min(1, e^{(c(j)-c(i))/T}) & \text{if } j \in N(i), \end{cases}$$

and

$$P_{ii}(T) = 1 - \sum_{j \in N(i)} P_{ij}(T).$$

Since the transition probabilities $P_{ij}(T)$ remain constant for a fixed temperature, SA can be modelled by a time-homogeneous Markov chain at any fixed temperature. But SA is the execution of the Metropolis simulation algorithm over a sequence of temperatures. Therefore, the exact model for SA is a time-inhomogeneous Markov chain.

A time-inhomogeneous Markov chain model for SA is used successfully by [16] to prove the convergence of SA algorithms. Their proof is for a special case of annealing schedule (see Section 2.1). In this paper we prove better convergence times when the cost function to be optimized has certain special properties.

3. Nested annealing

The algorithm

Let C be the given cost function on n parameters $\vec{p} = [p_1, p_2, \ldots, p_n]$. Define the state (i.e., configuration) of the NA process to be the n-vector of values that the parameters have assumed. If $\vec{p}_0$ is the starting state of NA, then we say NA has *frozen* (or converged) if it has been in a state $\vec{p}_1$ such that $C(\vec{p}_1)$ is globally optimal. Next we show if G_C is $s(n)$ separable (see Section 1.5 for definition), then NA freezes in $2^{ks(n)}$ time (for some k) with probability 1.

The property of small separability of graphs allows us to divide the given problem into successively smaller and independent subproblems involving smaller and smaller number of variables. The results of these subproblems can be combined to obtain the result of the original problem. This divide and conquer strategy has been used to improve the performance of many graph problems.

It was proven by [14] that planar graphs are $O(\sqrt{n})$ separable. Some of the algorithms that exploit this separability theorem and the divide and conquer strategy are (1) [15]'s approximation algorithm for many NP-complete problems including

the maximum independent set in a graph, (2) [17]'s inversion algorithm for sparse matrices, and (3) [9]'s algorithm for planar 3-SAT.

In this paper we show how to use NA technique to small separable cost functions to obtain better convergence (over SA). Let C be the function to be optimized and $G_C(V, E)$ be its graph. If G_C is $s(n)$ separable, then, either $|V| \leq n_0$ for some constant n_0 or G_C may be partitioned into two disjoint subgraphs with the vertex sets V_1 and V_2 such that $|V_i| \leq \alpha |V|$, $i = 1, 2$ (for some constant $\alpha < 1$), by deleting some separator set S of vertices such that $|S| \leq s(|V|)$ and furthermore, each of the two subgraphs of G_C defined by $S \cup V_i$, $i = 1, 2$ also will be $s(n)$ separable.

Define the separator tree T_{G_C} of G_C as follows. If $|V| \leq n_0$ let T_{G_C} be the trivial tree with no edges and with the single leaf S. If $|V| \geq n_0$, we know an $s(n)$ separator for G_C, so we can find a partition V_1, V_2, and S of V such that $|V_1| \leq \alpha n$, $|V_2| \leq \alpha n$ (for some $\alpha < 1$) and $|S| \leq s(n)$. Then T_{G_C} is defined to be the binary tree with the root S having exactly two children that are the roots of two subtrees T_{G_1} and T_{G_2} of T_{G_C}, where G_j is the subgraph of G_C induced by the vertex set V_j $(j = 1, 2)$. Define the depth of a node in T_{G_C} to be its distance to the root. Each node in the tree constitutes a zone (a zone is nothing but some separator set). There are 2^k zones at depth k. Number the nodes in any depth in increasing order from left to right. Every zone runs the SA algorithm SAnneal of Section 2. Zones in the same depth take the same time between displacements. This time is set such that within this time both its children zones would have converged. Detailed algorithm follows.

```
algorithm NAnneal(Z, X0, T0);
(* Z is a node (zone) in the separator tree. V is the parameter set in
this node. Z1 and Z2 are the two children of Z with parameter sets
V1 and V2. X0 is the initial configuration and T0 is the initial tem-
perature. *)
  if |V| ≤ n0 then solve the problem exhaustively else
  begin
    X := X0; s := 0;
    while (not frozen) do
      while (not steadystate) do
        XN := generateconfiguration(X);
        if (accept(c(XN), c(X), Ts))) then
          X := XN;
          NAnneal(Z1, X0', T0');
          NAnneal(Z2, X0'', T0'')
        Ts+1 := update(Ts);
        s := s + 1
  end;
```

In the above algorithm frozen and steadystate are boolean variables defined exactly as in Section 2.1. generateconfiguration(X) is also the same as was defined in Section 2.1; the only difference is the transition time (i.e., time between successive

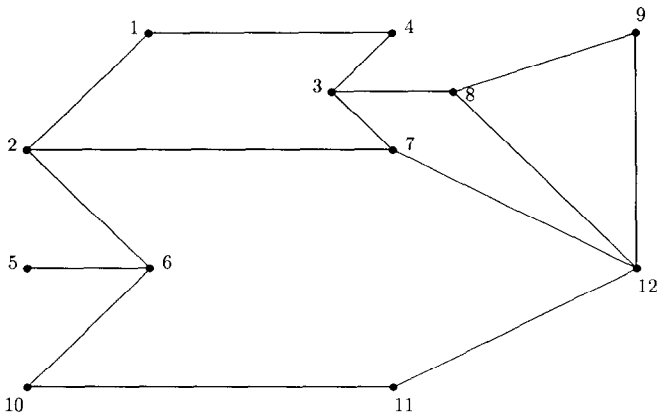

Fig. 1. Graph of an OP.

calls) is $2^{k's(\alpha^{d+1}n)}$ (for some constant k') if Z was at depth d of the separator tree. This time will be shown to be such that a displacement in zone Z occurs only after the two children zones of Z (viz., Z_1 and Z_2) have frozen. X_0', T_0', X_0'', T_0'' are initial configurations and initial temperatures for the zones Z_1 and Z_2. The parameters to the initial call of NAnneal will be the root zone and the corresponding initial configuration and initial temperature.

The construction of the separator tree can be explained with an example. The separator tree of the graph in Fig. 1 appears in Fig. 2. A displacement of the zone Z_1^1 (for example) occurs only after Z_1^2 and Z_2^2 have converged corresponding to the previous configuration of Z_1^1.

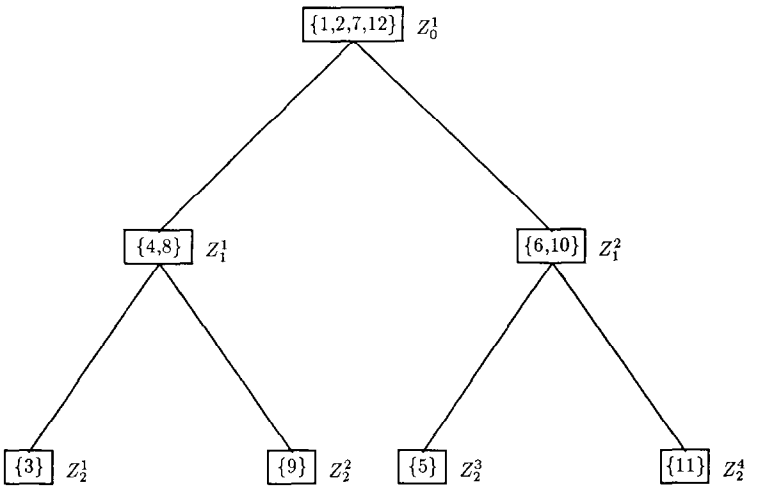

Fig. 2. Zones corresponding to Fig. 1.

Theorem 3.1. *The above algorithm freezes within time* $2^{ks(n)}$ *with probability* 1 *for some fixed constant k.*

We need the following lemma in our proof.

Lemma 3.1 [16, 22]. *The worst case convergence time of the algorithm SAnneal for an OP with n parameters is* $O(2^{cn})$ *with probability* 1 *for some constant c.*

Proof of Theorem 3.1. At level d of the separator tree T_{G_c} there are 2^d nodes (zones). Each one of these zones consists of $s(\alpha^d n)$ parameters. Let Z_d^l be any zone in depth d.

The worst case number of displacements any zone at depth d will have to make (before convergence) is $O(2^{cs(\alpha^d n)})$ with probability 1 (Lemma 3.1). Therefore if T is the convergence time for Z_d^l, then the convergence time for its parent is $O(2 \cdot T \cdot 2^{cs(\alpha^{d+1}n)})$ (notice there are two children for this parent). If T' is the convergence time for any of Z_d^l's children, then since Z_d^l has to make at the most $O(2^{cs(\alpha^d n)})$ moves (i.e., displacements) the run time of Z_d^l's parent is $O(2 \cdot 2T' \cdot 2^{cs(\alpha^d n)} \cdot 2^{cs(\alpha^{d+1}n)})$ with probability 1.

Similarly proceeding we see that the convergence time for Z_0^1 (and hence for NAnneal) is $O(c' 2^{c\{s(n)+s(\alpha n)+s(\alpha^2 n)+\cdots+s(\alpha^{d'}n)\}})$ where c' is the convergence time of leaves of the separator tree (and therefore a constant) and d' is the depth of the separator tree (equal to $O(\log n)$). If $s(n) = O(n^\sigma)$ for $\sigma < 1$, then the convergence time of NAnneal is $O(2^{cs(n)})$ (for some constant c) with probability 1. (Notice that the time between two displacements of Z_d^l has been set right.) □

The algorithm NAnneal inherently implements a recursive algorithm that will be described in Section 5. In a physical system, it is not possible to impose such a recursive algorithm explicitly. But NAnneal applies to physical systems as well. The proof of convergence of the recursive algorithm in Section 5 also serves as an alternative proof for the convergence of NAnneal.

4. Applications of NA

NA is applicable to all the problems that SA can solve. The relative advantage of NA over SA in terms of convergence speed will depend on how small-separable the cost functions are. The expected worst case convergence time of NA will equal that of SA only when the cost function is $\Omega(n)$ separable.

In this section we discuss four interesting applications of NA viz. (1) planar-SAT, (2) a vision problem, (3) planar travelling salesman (planar-TS) and (4) learning. All these four problems have graphs that are planar and hence $O(\sqrt{n})$ separable. So NAnneal will run in time $O(2^{c\sqrt{n}})$ with probability 1 for all these problems.

4.1. Planar-satisfiability

SAT is the problem of testing the satisfiability of boolean formulae in Conjunctive Normal Form. Let F be a boolean formula over the variables U. Define a bipartite graph G_F as follows: (1) nodes of G_F are variables and clauses of F, (2) there is an edge from a variable node to a clause node iff that variable appears in that clause. As an example, if $F=(x_1+x_2+x_5)(x_3+\bar{x}_2)(x_6+\bar{x}_1+x_3)$, then the graph G_F is shown in Fig. 3.

Planar-SAT is SAT restricted to formulae whose graphs are planar. Planar-SAT is known to be NP-complete. Since planar graphs are $O(\sqrt{n})$ separable, our NA runs in worst case expected time $O(2^{c\sqrt{n}})$ (for any $c>0$).

4.2. A vision problem

The vision problem we are interested in is: Given a drawing of straight lines on the plane, to decide if it is the projection of the visible part of a set of opaque polyhedra. This is a typical problem that a robot faces in motion planning. This vision problem is also known to be NP-complete. Basic steps of any heuristic used to solve this problem are [12] (1) label the edges as *concave*, *convex*, or *contour*

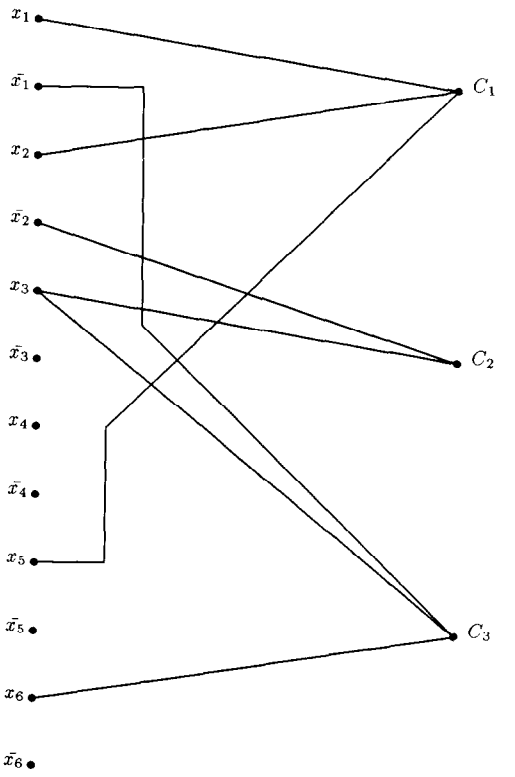

Fig. 3. G_F for $F=(x_1+x_2+x_5)(x_3+\bar{x}_2)(x_6+\bar{x}_1+x_3)$.

depending on their slopes on the projection plane, and (2) using the information from (1), calculate the coordinates of points and slopes of planes.

Nested annealing when used to solve this vision problem yields a worst case input expected run time of $O(2^{c\sqrt{n}})$, $c > 0$ where n is the number of connected components in the projection.

4.3. *Planar travelling salesman problem* (PTS)

PTS is to construct a polygon of minimum perimeter through a set of given points. [3] presented an $O(n^3)$ algorithm for this problem that gives a tour less than 50% longer than the optimal tour. [8] assumed the input was random. His probabilistic algorithm runs in time $C(\varepsilon) + O(n \log n)$ and yields a tour $\leq \varepsilon$ times longer than the optimal tour. [8]'s algorithm consisted of two phases: (1) The given regions on the plane (a unit square w.l.o.g.) is divided into rectangular subregions such that each subregion consists of nearly the same number of points. An optimal tour is obtained for each one of these subregions by exhaustive search. (2) The subtours so obtained are connected appropriately to get an optimal tour for the original problem.

Instead of solving the subproblems exhaustively, NA can be used. Or, NA by itself can be used to solve the whole problem. In either case, the run time and the quality of approximation improves.

4.4. *Learning*

A simple and popular paradigm for machine learning is this: The machine is required to learn a concept (or predicate). An oracle teaches the machine this concept by giving a series of examples for which the predicate is true. The question is what are all the possible concepts the machine can learn and how fast will the machine learn them?

We say a machine has learnt a concept Q if a program for recognizing Q has been deduced by the machine, by some methods other than the acquisition from the outside of the explicit program.

Valiant [20] shows that the following classes of concepts are *polynomial time learnable*: (1) CNF expressions with a bounded number of literals in each clause, (2) monotone DNF expressions, (3) arbitrary expressions in which each variable occurs just once.

The above classes of concepts are learnable in polynomial time. The concept learnt will be good only relative to the distribution of examples supplied by the oracle. In other words, there is a nonzero probability that the concept learnt will be wrong for an example that the oracle rarely supplies.

NA can be applied for learning not only the above concepts but in general any boolean expression.

Some other applications of NA are (1) graph partitioning, (2) circuit partitioning, (3) placement of VLSI circuits, (4) routing and location, (5) global wiring, (6)

coloring of the vertices of a graph and so on. All these problems have cost functions whose graphs are planar. So, the worst case input expected run time for these problems is no more than $O(2^{c\sqrt{n}})$, $c > 0$.

5. A divide and conquer simulated annealing algorithm

In this section we present a divide and conquer algorithm for solving any OP that has a small separable graph. As the reader can see algorithm NAnneal is nothing but an implementation of this divide and conquer algorithm that embodies SA also. This divide and conquer algorithm is not only an algorithm in its own right (for solving OP) but also the proof of convergence time for this algorithm serves as an alternative (and more rigorous) proof for the convergence time of NAnneal.

5.1. Summary of the algorithm

Let C be the function on n parameters to be optimized. Let $G_C(V, E)$ be the graph of C. If G_C is $s(n)$ separable, then, either $|V| \leq n_0$ for some constant n_0 or G_C may be partitioned into two disjoint subgraphs with the vertex sets V_1 and V_2 such that $|V_i| \leq \alpha |V|$, $i = 1, 2$ (for some constant $\alpha < 1$), by deleting some separator set S of vertices such that $|S| \leq s(|V|)$ and furthermore, each of the two subgraphs of G_C defined by $S \cup V_i$, $i = 1, 2$ also will be $s(n)$ separable.

Let $\vec{S}_p$ stand for the parameters of the optimization problem in the separator set S. Also let $\vec{V}_{1p}$ and $\vec{V}_{2p}$ stand for the parameters in V_1 and V_2 respectively. The optimum value of C occurs for some values of the parameters $\vec{S}_p$, $\vec{V}_{1p}$, and $\vec{V}_{2p}$. One way of finding optimum values for these parameters will be the following.

repeat until no improvement in C occurs
begin
step1: Fix the value of $\vec{S}_p$.
step2: Find an optimal value of $\vec{V}_{1p} \circ \vec{V}_{2p}$ corresponding to the above value of $\vec{S}_p$.
step3: Change the value of $\vec{S}_p$.
end;

The above procedure defines two different processes. The main process is to find an optimal value for $\vec{S}_p$. The other process works to find an optimal value for $\vec{V}_{1p} \circ \vec{V}_{2p}$. If each one of these processes performs an exhaustive search in its corresponding solution space, then the above procedure corresponds to an exhaustive search in the solution space of $\vec{p}$. We do not want to do this exhaustive search. Instead we will replace each one of these processes by an SA process. If we do so, then we get two Markov chains M' and M'' corresponding to the main process and the secondary process. These two Markov chains are such that a transition of M' occurs only after M'' has converged with respect to the previous state of M'. In

other words, the value of $\vec{S}_p$ is changed only after M'' has found an optimal value for $\vec{V}_{1p} \circ \vec{V}_{2p}$ corresponding to the previous value of $\vec{S}_p$.

We can also use the SA algorithm of Section 2 to find an optimal value for $\vec{p}$ without partitioning $\vec{p}$ into three parts as suggested by the above procedure. If M is the Markov chain corresponding to this process, then it can easily be seen that M'' is nothing but $M_{\vec{V}_{1p} \circ \vec{V}_{2p}}$, the restriction of M on $\vec{V}_{1p} \circ \vec{V}_{2p}$. The number of states of M'' is $2^{n-|\vec{S}_p|}$. As stated in the following lemma, M'' can be decomposed into two independent Markov chains with roughly $2^{(n-|\vec{S}_p|)/2}$ number of states in each. Since the convergence time of a Markov chain is an increasing function of the number of states in the Markov chain, such a decomposition indeed will result in an improved run time of the whole algorithm. Using Lemma 3.1, M'' without decomposition converges in time $2^{c(n-|\vec{S}_p|)}$ whereas M'' with decomposition converges in time $2 \cdot 2^{c((n-|\vec{S}_p|)/2)}$.

Lemma 5.1. *If $\vec{q}_1$ is an optimal state of $M_{\vec{V}_{1p}}$ and $\vec{q}_2$ is an optimal state of $M_{\vec{V}_{2p}}$, then $\vec{q}_1 \circ \vec{q}_2$ is an optimal state of $M_{\vec{V}_{1p} \circ \vec{V}_{2p}}$. Furthermore, $M_{\vec{V}_{1p}}$ and $M_{\vec{V}_{2p}}$ are independent.*

Proof. Under some substitution for the parameters in the separator set, we want to know what value of $\vec{V}_{1p} \circ \vec{V}_{2p}$ will make the function $C = C_1 + C_2 + \cdots + C_m$ optimal. Under some substitution for $\vec{S}_p$, C can be written down as $C = C' + C''$ where only parameters in $\vec{V}_{1p}$ appear in C' and only parameters in $\vec{V}_{2p}$ appear in C'', since S is a separator set. The optimum value of C occurs when both C' and C'' are optimum and the optimum value of C' does not affect the optimum value of C'' and vice-versa. Therefore, the problem of finding an optimal vector $\vec{V}_{1p} \circ \vec{V}_{2p}$ reduces to the problems of finding an optimal value of $\vec{V}_{1p}$ that optimizes C' and finding an optimal value of $\vec{V}_{2p}$ that optimizes C''. Clearly, the process that finds an optimal value of $\vec{V}_{1p}$ is the Markov chain $M_{\vec{V}_{1p}}$ and the process that finds an optimal value of $\vec{V}_{2p}$ is $M_{\vec{V}_{2p}}$. This proves the first statement of the lemma.

$M_{\vec{V}_{1p}}$ and $M_{\vec{V}_{2p}}$ are independent because all the transition arcs from states of $\vec{V}_{1p}$ to the states of $\vec{V}_{2p}$ and vice-versa vanish due to the substitution for the separator set. Thus the Markov chain $M_{\vec{V}_{1p} \circ \vec{V}_{2p}}$ decomposes into two disjoint sub-Markov chains $M_{\vec{V}_{1p}}$ and $M_{\vec{V}_{2p}}$. □

Lemma 5.2. *If T' is the maximum of the convergence times of $M_{\vec{V}_{1p}}$ and $M_{\vec{V}_{2p}}$, and T is the convergence time of $M_{\vec{V}_{1p} \circ \vec{V}_{2p}}$, then $T \leq 2T'$.*

5.2. The algorithm

Lemmas 5.1 and 5.2 can be exploited to obtain an efficient SA algorithm as will be shown in this section. The main structure of our algorithm will be the one given in Section 5.1. The only change will be to modify step 2 in accordance with Lemma 5.1. The problem of computing an optimal value for $\vec{V}_{1p} \circ \vec{V}_{2p}$ will be reduced to the problem of finding optimal values for $\vec{V}_{1p}$ and $\vec{V}_{2p}$. Optimal values for $\vec{V}_{1p}$ and

$\vec{V}_{2p}$ will be found using the algorithm recursively. Details of the algorithm follow.

algorithm DC(G_C(V, E));
if $|V| \leq n_0$ *then* solve the problem exhaustively *else*
begin
Find a separator set S for G_C; Let V_1 and V_2 be the vertex sets for the two disjoint graphs G'_C and G''_C obtained from G_C by deleting S.
repeat until no improvement in C occurs
begin
step1: Fix the value of $\vec{S}_p$ (parameters in S);
step2: Let $\vec{V}_{1p} = \mathrm{DC}(G'_C(V_1, E_1))$; Let $\vec{V}_{2p} = \mathrm{DC}(G''_C(V_2, E_2))$; compute $\mathrm{C}(\vec{V}_{1p} \circ \vec{S}_p \circ \vec{V}_{2p})$;
step3: *generateconfiguration*($\vec{S}_p$) (i.e., change the value of $\vec{S}_p$)
end;
end;

Theorem 5.1. *The above algorithm runs in time* $2^{k\cdot\beta\cdot s(n)}$ *with probability* $\geq(1-n^{-\beta})$, *for some fixed constant k.*

Proof. The proof will be by induction on the level of recursion. Let the separator tree of G_C be T_{G_C}. Each level of T_{G_C} corresponds to one level of recursion of our SA algorithm. Depth d of T_{G_C} corresponds to level d of recursion (assuming that the recursion starts at level 0). At depth d of T_{G_C}, there are 2^d nodes, each node having a separator set and two subgraphs of G_C as its children. In our algorithm, correspondingly, at level d there are 2^d sub-Markov chains each solving a subproblem on $\alpha^d n$ parameters.

We first prove the expected convergence time by induction and then apply Markov's inequality to assert the worst case run time.

Inductive proof of the expected run time.

Inductive hypothesis. Each sub-Markov chain at level d converges within $2^{k\beta s(\alpha^d n)}$ expected time.

Base case. Let d be such that $\alpha^d n = \mathrm{O}(1)$. At this level d, there are $c'n$ (for constant c') sub-Markov chains each solving an optimization problem on O(1) parameters. Thus the induction hypothesis holds trivially.

Induction step. Assume that induction hypothesis holds for all levels $\geq(d+1)$. We prove the hypothesis for level d.

Consider a node N at level d. It has a separator set of size $s(\alpha^d n)$ and each one of its two children has subgraphs with $\leq\alpha^d n$ vertices in each. Corresponding to each fixed value of the separator set, both the sub-Markov chains of N will converge within expected time $2^{k\beta s(\alpha^{d+1}n)}$. M_S, the Markov chain corresponding to the separator set makes a move only after both its children have converged. According to Lemma 3.1, M_S has to make at the most $\mathrm{O}(2^{cs(\alpha^d n)})$ moves (for some constant c). The transition time between any two moves of M_S is $\leq 2\times 2^{k\beta s(\alpha^{d+1}n)}$ (Lemma 5.2).

Therefore, T_d, the time needed for the convergence of M_S is $\leq 2^{cs(\alpha^d n)} \times 2^{k\beta s(\alpha^{d+1} n)}$. T_d will be $\leq 2^{k\beta s(\alpha^d n)}$ if

$$cs(\alpha^d n) + k\beta s(\alpha^{d+1} n) \leq k\beta s(\alpha^d n),$$

i.e., if

$$k \geq \frac{cs(\alpha^d n)}{\beta[s(\alpha^d n) - s(\alpha^{d+1} n)]} \approx \frac{c}{\beta}\left[1 + \frac{s(\alpha^{d+1} n)}{s(\alpha^d n)}\right].$$

If $s(n) = O(n^\sigma)$ for some $\sigma > 0$, then k will be a constant.

It remains to show that the convergence time of the whole process is $\leq 2^{k\beta s(n)}$ with overwhelming probability. This can be shown using Markov's inequality. Let X be the random variable corresponding to the convergence time. The above proof for expected convergence time implies that the expected value of X, $E(X) = 2^{k\beta s(n)}$. Applying Markov's inequality, probability that X is $\geq 2^{k\beta s(n) + \alpha \log n}$ is $\leq n^{-\alpha}$. Thus the claim follows. □

Even though the convergence time for our algorithm is exponential in the separator size, in practice, the run time will be much smaller since each one of the Markov chains in our algorithm is only looking for a quasi-optimal solution.

6. Extensions and conclusions

The success of any heuristic technique depends on how good the solution space is. There can not be a single heuristic that is guaranteed to work on all problem instances of a problem domain, since the existence of such a heuristic will imply that the heuristic closely characterizes the problem domain, which in turn will prove the existence of fast algorithms for exact solutions. SA is no exception to this fact. In this section we explore how good NAnneal will perform for problem domains with certain special properties.

Case 1: The solution space (of cardinality 2^n), consists of l optimal solutions distributed uniformly in a hypercube with 2^n nodes.

Lemma 6.1. *If $2^{ks(n)}$ is the worst case run time of our NAnneal for a general solution space, then the run time of NAnneal for case* 1 *is $2^{ks(n - \log l)}$.*

Proof. If case 1 is true, then any adjacent $2^n/l$ nodes of the hypercube is guaranteed to have an optimal solution with high probability. In particular, any subcube of dimension $(n - \log l)$ contains an optimal solution with high probability. Therefore, if we fix arbitrarily the values of any $\log l$ parameters and use NAnneal to find an optimal value for the other $(n - \log l)$ parameters, then the solution we obtain will be an optimal one with high probability. Now use Theorem 5.1 to prove the lemma. □

Case 2: Any subcube of dimension εn ($\varepsilon \leq 1$) consists of a solution that is at the most $1/\varepsilon$ times the global optimal value.

Lemma 6.2. *For case* 2, *NAnneal needs* $\leq 2^{ks(\varepsilon n)}$ *time to find an optimal solution that is at the most* $1/\varepsilon$ *times the global optimal value.*

Proof. Similar to the proof of Lemma 6.1. □

In summary, SA has proven to be a competitive algorithm for solving *hard* combinatorial optimization problems. In comparison with other heuristic algorithms, it performs better on many instances. We have explained in this paper how the NA technique can be exploited to improve the performance of SA algorithms in cases where the cost functions are small-separable.

References

[1] E. Bonomi and J.L. Lutton, Simulated annealing algorithm for the minimum weighted perfect euclidean matching problem, R.A.I.R.O. Operations Research, to appear.

[2] D. Brelaz, New methods to color the vertices of a graph, *Comm. ACM* **22** (1979) 251-256.

[3] N. Christofides, Worst case analysis of a new heuristic for the travelling salesman problem, Abstract in: J. Traub, ed., *Algorithms and Complexity* (Academic Press, New York, 1976) 441.

[4] A.E. Dunlop and B.W. Kernighan, A procedure for placement of standard-cell VLSI circuits, *IEEE Trans. Comput. Aided Design* **4** (1985) 92-98.

[5] A. ElGamal and I. Shperling, Design of good codes via simulated annealing, List of Abstracts, Workshop on Statistical Physics in Engineering and Biology, Yorktown Heights, NY, April 1984.

[6] B.L. Golden and C.C. Skiscim, Using simulated annealing to solve routing and location problems, *Naval Res. Logistics Quart.* **33** (1986) 261-279.

[7] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, Optimization by simulated annealing: an experimental evaluation (Part I), Preliminary Draft, AT&T Bell Labs., Murray Hill, NJ, 1987.

[8] R.M. Karp, Probabilistic analysis of partitioning algorithms for the travelling salesman problem in the plane, *Math. Oper. Res.* **2** (3) (1977) 209-224.

[9] S. Kasif, J.H. Reif and D.D. Sherlekar, Formula dissection: a divide and conquer algorithm for satisfiability, Technical Report, Johns Hopkins University, 1985.

[10] B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *The Bell System Tech. J.* Feb. (1970) 291-307.

[11] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, Optimization by simulated annealing, *Science* May (1983) 671-680.

[12] L.M. Kirousis and C.H. Papadimitriou, The complexity of recognizing polyhedral scenes, in: *Proc. IEEE Symp. on Foundations of Computer Science* (1985) 175-185.

[13] D. Lichtenstein, Planar formulae and their uses, *SIAM J. Comput.* **11** (2) (1982) 329-343.

[14] R.J. Lipton and R.E. Tarjan, Applications of a planar separator theorem, *SIAM J. Comput.* **9** (3) (1980) 615-627.

[15] R.J. Lipton and R.E. Tarjan, A planar separator theorem, *SIAM J. Appl. Math.* **36** (2) (1979) 177-189.

[16] D. Mitra, F. Romeo and A.S. Vincentelli, Convergence and finite-time behavior of simulated annealing, *J. Advances Appl. Probability* Sept. (1986) 747-771.

[17] V. Pan and J.H. Reif, Fast and efficient solutions of linear systems, in: *Proc. 17th Ann. Symp. on Theory of Computing* (1985) 143-152.

[18] M.O. Rabin, Probabilistic algorithms, in: J.F. Traub, ed., *Algorithms and Complexity* (Academic Press, New York, 1976) 21–36.
[19] R. Solovay and V. Strassen, A fast Monte-Carlo test for primality, *SIAM J. Comput.* **6** (1977) 84–85.
[20] L.G. Valiant, A theory of the learnable, *Comm. ACM* **27** (11) (1984) 1134–1142.
[21] M.P. Vecchi and S. Kirkpatrick, Global wiring by simulated annealing, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* **CAD-2** (1983) 215–222.
[22] S. Rajasekaran, Separability of a random graph and applications, presented in the International Seminar on Random Graphs, Poznań, Poland, August 1991.