Simulated Annealing:
a Fast Heuristic for Some Generic Layout Problems

Jimmy Lam[†], Jean-Marc Delosme[††]
[†] Department of Computer Science
[††] Department of Electrical Engineering
Yale University
New Haven, CT 06520

Abstract

There are two major criticisms about simulated annealing as a general method for solving combinatorial optimization problems: its effectiveness when compared with other well-designed heuristics and its excessive computation time. In this paper, we show that simulated annealing, with properly designed annealing schedule and move generation strategy, achieves significant *speedups* for high quality solutions when compared with tailored heuristics on two well-studied problems: the traveling salesman problem and the graph partition problem. Efficient heuristics for the traveling salesman problem can be applied to power and ground routing and, for the graph partition problem, to min-cut placement and logic partitioning.

1. Introduction

Most of the practical applications of simulated annealing (S–A) have been in complicated problem domains, where algorithms either did not exist or performed poorly. To assess the performance of S–A as a general method for solving combinatorial optimization problems, we have to compare the method with efficient heuristics on well-studied problems. The traveling salesman problem (TSP) and the graph partition problem (GPP) are chosen as the "arena" for our comparison because of two reasons. First, quite a few problems in the layout domain can be formulated as TSP and GPP. For example, power and ground routing in PI, a placement and interconnect system [11, p.429], is formulated as a TSP while min-cut placement [2] and logic partitioning [4] are formulated as GPP. Second, both the TSP and the GPP have been studied extensively for two decades resulting in a number of efficient heuristics, such as the ones by Lin and Kernighan (L–K–H) [9], and Karp (Ka–H) [6] for the TSP, and the one by Fiduccia and Mattheyses (F–M–H) [3] for the GPP. By comparing S–A with these efficient heuristics, we can assess how well S–A performs as a general method for solving combinatorial optimization problems. If S–A can compete favorably with these well-designed heuristics on both the qualities of solutions and the computation time, imagine what the method can do on problems in the layout domain where few efficient heuristics are known. TimberWolfSC [10], a standard cell placement and global routing package, is an example of successful application of S–A in an area where few good algorithms exist.

In the next two sections, we shall show that S–A, with efficient annealing schedule and move generation strategy and for high quality solutions, achieves a speedup of up to 47 when compared with L–K–H for the TSP and a speedup of up to 5 when compared with multiple executions of F–M–H for the GPP.

The test results given in this paper were measured on a Sun 3/280-s8 with MC68881 floating point option. Unless stated otherwise, they are average results of *at least eight* executions. All programs were implemented in the C programming language.

2. Traveling salesman problem

One of the classical combinatorial optimization problems is the TSP. The goal of this problem is to minimize the length of a tour starting from any city, visiting each of the N given cities once and only once, and returning to the original place of departure. We restrict our attention to problems where the cities lie in the plane, and the distances between cities are symmetric and Euclidean. We first cover our S–A implementation, followed by a discussion of the test results. The simulated annealing schedule and the move generation control method described in [7] and [8] are used in our implementation.

2.1. Implementation details

The cost to be minimized in a TSP is the length of a tour. In our S–A implementation, each city maintains a list of up to M neighboring cities sorted in ascending order according to their distances from that city. A move is proposed by first picking two cities A and B, then modifying the tour as shown in Fig. 1. City A is always picked randomly with equal probability from the N given cities while city B is the kth entry on the neighbor list of city A with

$$k = -r * \log(\theta),$$

where θ is a random number uniformly distributed between 0 and 1, and r is a control parameter between 2 and N. Because θ is uniformly distributed, k will be exponentially distributed,

$$p(k) = \frac{1}{r} e^{-k/r},$$

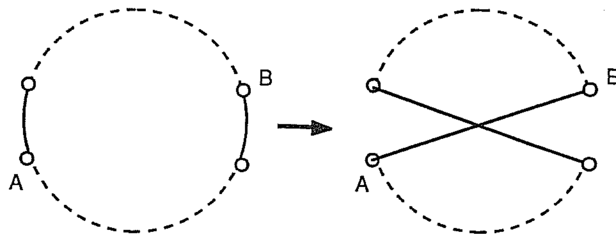

Figure 1: How a tour is modified in the TSP.

with an average value of r. Thus, the closer is a city to city A, the higher is its probability of being picked. If the value of k exceeds the value of M, a condition that happens less frequently as r gets smaller, k will be set to $\theta * N$. The control parameter r is adjusted after every 100 moves according to the formula

$$r \leftarrow r + 100*(\hat{\alpha} - 0.44),$$

where $\hat{\alpha}$ is the measured acceptance ratio for the last 100 moves. If $\hat{\alpha}$ is less than 0.44, the value of r is lowered. If $\hat{\alpha}$ is greater than 0.44, the value of r is raised. This updating procedure enables us to keep the acceptance ratio close to the *desired* acceptance ratio of 44%. (Refer to [7] for an explanation to why an acceptance ratio of 44% is desirable.) The value of M is set to the minimum of N-1 and 250.

2.2. Test results

We compare S–A with L–K–H, and Ka–H for the TSP. The comparison with L–K–H covers four types of city distributions: uniformly distributed cities, super-clustered cities, cities on a lattice, and cities on a lattice with perturbations. In the last case, the maximum distance by which a city can be perturbed is 50% of the normal inter-neighbor distance. All results tabulated are the average of *at least eight* executions. In the case of L–K–H, each execution consists of 3 iterations and within each iteration, the local optima found in the preceding iterations are used to speed up the search at the current iteration.

The speedups, defined as the ratio of the CPU time used by L–K–H, t', to the CPU time used by S–A, t, are shown in Table 1 to Table 4. In Table 1 and Table 4, where the optimal costs are not known, δ' and δ are, respectively, the percentages above the estimated best costs for L–K–H and for S–A while in Table 2 and Table 3, where the optimal costs are known, δ' and δ are the percentages above the optimal costs. The number of cities (size) ranges from 64 to 512.

Size	CPU time (sec.)			Tour length	
	t'	t	t'/t	$\delta'(\%)$	$\delta(\%)$
100	69.8	52.9	1.32	1.01	1.01
200	296.0	101.6	2.91	1.25	1.21
300	686.1	197.5	3.47	1.32	1.32
400	1492.7	330.8	4.51	1.46	1.40

Table 1: Comparison with L–K–H on the TSP with uniformly distributed cities.

Size	CPU time (sec.)			Tour length	
	t'	t	t'/t	$\delta'(\%)$	$\delta(\%)$
64	57.9	9.3	6.23	0.58	0.58
128	428.2	17.4	24.61	1.13	1.06
256	2113.6	61.5	34.37	2.52	2.58
512	13883.1	296.3	46.85	4.11	4.03

Table 2: Comparison with L–K–H on the TSP with super-clustered cities.

Size	CPU time (sec.)			Tour length	
	t'	t	t'/t	$\delta'(\%)$	$\delta(\%)$
100	13.0	78.2	0.17	0.20	0.31
200	70.7	492.0	0.14	0.15	0.67
300	190.3	1338.4	0.14	0.10	1.09
400	415.6	3125.5	0.13	0.25	1.09

Table 3: Comparison with L–K–H on the TSP with cities on a lattice.

Size	CPU time (sec.)			Tour length	
	t'	t	t'/t	$\delta'(\%)$	$\delta(\%)$
100	32.7	104.9	0.31	0.70	0.70
200	182.3	205.6	0.89	0.73	0.70
300	510.6	270.8	1.89	1.42	1.38
400	978.8	417.5	2.34	1.43	1.42

Table 4: Comparison with L–K–H on the TSP with cities on a lattice with up to 50% perturbations.

From these tables, we observe that L–K–H performs very poorly on super-clustered cities and does a very good job on cities on a lattice. This comes as no surprise since L–K–H produces 3-optimal solutions—an n-optimal solution is a local optimum with respect to moves that exchange n sections of a tour instead of only two sections as in Fig. 1. In the case of super-clustered cities, the number of locally optimal solutions that are 2-optimal is large. Therefore, L–K–H spent lots of time going from one 2-optimal solution to another until it reached a 3-optimal solution. In the case of cities on a lattice, the number of globally optimal solutions is large: any tour formed by connecting cities either vertically or horizontally is a globally optimal solution. Consequently, the probability of finding one of these optimal solutions is high and the heuristic terminates quickly. To check that the number of globally optimal solutions influences the speed of L–K–H, we perturbed the positions of the cities on a lattice so that the number of globally optimal solutions is small. The CPU times for the perturbed problems shown in Table 4 fall in between those for the uniformly distributed cities and the cities on a lattice. This observation confirms our conjecture that the abundance of globally optimal solutions speeds up L–K–H.

On the contrary, S–A is relatively insensitive to the structure and regularity of an optimization problem; the CPU times used by S–A for different city distributions stay fairly constant for similar qualities of solutions. This combination of the sensitivity of L–K–H towards the number of global optimums and the insensitivity of S–A towards the structure of a problem explains the drastic speedup (up to 47) for problems with super-clustered cities as well as the superiority of L–K–H over S–A on problems with cities on a lattice.

Except for the case of cities on a lattice, S–A provides a substantial speedup over L–K–H for high quality solutions. This observation indicates that for problems where globally optimal solutions are sparse and high quality solutions are desired, S–A coupled with a good move generation strategy is an efficient and highly competitive heuristic.

The comparison with Ka–H coupled with L–K–H was performed on a TSP with 10,000 cities whose coordinates are uniformly distributed in the interval of 0 to 1,000,000. Ka–H consists of three stages: partition a problem into smaller size sub-problems, solve the sub-problems using another heuristic, and patch the resulting solutions to form a tour. In this experiment, we used L–K–H as the heuristic that solves the partitioned sub-problems.

The result of the comparison between S–A and Ka–H is shown in Table 5. The estimated best cost for this problem is based on the formula [1]

$$\lim_{N \to \infty} \frac{C_{opt}}{\sqrt{N*I^2}} = 0.749,$$

where C_{opt} is the optimal tour length and I = 1,000,000 is the interval from which the coordinates of the cities are picked. Table 5 is constructed by varying the number of partitions used in Ka–H and comparing the resulting solutions with simulated annealing. We observe that S–A out-performs Ka–H for high quality solutions. This evidence strongly supports our belief that an efficient annealing schedule coupled with a good move generation strategy makes S–A an extremely effective heuristic.

3. Graph partition problem

The goal of the GPP is to partition the vertices of a graph into two equal size vertex sets, V_A and V_B, so as to minimize the number of edges with end-points in both sets. Our experimental results in this problem are similar to those obtained by Johnson *et al.* [5] with one essential difference. By

using an efficient annealing schedule [7] and a good move generation strategy, we are able to speed up S–A significantly. Thus, for the same high quality solutions, the CPU time spent by S–A compares favorably with the CPU time spent by F–M–H. We first describe our S–A implementation, followed by a discussion of the test results.

Partition	CPU time (sec.)			Tour length	
	t'	t	t' / t	$\delta'(\%)$	$\delta(\%)$
128	4850	11050	0.44	6.98	6.66
64	10330	16070	0.64	3.52	3.41
32	26460	21610	1.22	2.03	1.97

Table 5: Comparison with Ka–H on the TSP with 10,000 uniformly distributed cities.

3.1. Implementation details

The cost to be minimized in a GPP is the size of the cutset—the number of edges with end-points in both vertex sets. In order to increase flexibility in move generation, we, following the work of Johnson *et al*, introduce an additional imbalance factor into the cost function:

$$Cost = cutsize + \xi*(\mid V_A \mid - \mid V_B \mid)^2,$$

where *cutsize* is the size of the cutset, ξ is the imbalance factor, and $\mid V_A \mid$ and $\mid V_B \mid$ are, respectively, the number of vertices in sets V_A and V_B. The imbalance factor serves two purposes. The first one is to increase flexibility in move generation by allowing moves that result in a difference between $\mid V_A \mid$ and $\mid V_B \mid$. This increase in flexibility increases the number of available moves and, hence, the chance of climbing out from a local optimum. The second one is to control the difference in size (the number of vertices) between V_A and V_B. A big difference in size is unlikely to occur at low temperature because moves that reduce this difference would likely be accepted. Since these two purposes are contradictory, we must choose the imbalance factor carefully: too small a value will unduly favor the first purpose while too large a value will unduly favor the second purpose. The value of the imbalance factor used in our experiments is either 0.005 or 0.02. The former value is used for problems in which the average edge degree is low while the latter value is used for problems in which the average edge degree is high.

A move is proposed in two steps involving two vertices. First, a vertex is picked from an array of double link lists and moved to the other vertex set. Then, a second vertex is picked independently from the array and moved to the other vertex set. Since the two vertices are picked independently, we may end up moving two vertices from the same set to the other, a perfectly acceptable operation.

In order to understand how move generation is controlled, we must first know how the double link lists are organized. Associated with each vertex, there is a variable called *gain* that keeps track of the change in cutsize if this vertex is moved from its current vertex set to the other. All vertices with the same *absolute* gain are grouped to form a double link list. These lists can then be indexed by an array in which the nth element of the array points to the double link list of vertices with absolute gain n. We shall call the vertices to which the array points the *head vertices* of the lists.

Move generation is controlled by picking two vertices independently from the head vertices of two double link lists selected using the formula

$$k = -r*\log(\theta).$$

In this formula, θ is a random number uniformly distributed between 0 and 1, r is a control parameter between 1.5 and M (the maximum absolute gain possible) and k denotes the kth list in the array. If the value of k exceeds the value of M, a condition that happens less frequently as r gets smaller, k will be set to $\theta*M$. After the head vertex of a list is picked, it loses its head position to the next vertex in the list even if the proposed move is rejected. The control parameter r is adjusted after every 100 moves according to the formula

$$r \leftarrow r + 5*(\hat{\alpha} - 0.44),$$

where $\hat{\alpha}$ is the measured acceptance ratio for the last 100 moves. If $\hat{\alpha}$ is less than 0.44, the value of r is lowered. If $\hat{\alpha}$ is greater than 0.44, the value of r is raised. Thus, the acceptance ratio is regulated so that it stays close to the desired acceptance ratio of 44%.

In the implementation of F–M–H, we considered two procedures in which the size of the two vertex sets was allowed to differ either by at most 1 or by at most $0.1*N$, where N is the total number of vertices. The resulting size difference is eliminated at the end of an execution by moving vertices that lead to the smallest increase in cutsize from the larger vertex set to the smaller one. Preliminary experimentation did not indicate any significant difference between the final results of the two procedures. We, therefore, restrict the maximum size difference of the two vertex sets to 1.

3.2. Test results

We compare S–A with the best of multiple executions of F–M–H (a linear time heuristic) for the GPP. Three types of graphs were used in the comparison: random graph, geometric random graph, and hierarchical graph. To construct a random graph with N vertices and an average edge degree d, we add an edge between any pair of vertices with a probability of $d / (N - 1)$. Since for any given vertex, there are $N - 1$ such pairs, the average edge degree is d. To construct a geometric random graph with N vertices and an average edge degree d as described in [5], we first pick N pairs of random numbers uniformly and independently distributed in the interval of 0 to 1. These N pairs of numbers are the coordinates of the N vertices. Then, we add an edge between any two vertices whose distance apart is less than r with $r = \sqrt{d / N\pi}$. This formula for r is obtained by noting that all N vertices are contained in an area of 1. Hence, the average area per vertex is $1 / N$. To find an average of d vertices whose distances are less than r apart from a given vertex (i.e., in an area of πr^2), we let $r = \sqrt{d / N\pi}$. To construct a hierarchical graph with 4^k vertices where $k = 1, 2, \ldots$, we apply the following algorithm:

1. let $i = k$.
2. partition the 4^i vertices into 4^{i-1} groups with four vertices in each group.
3. add four edges to each group to make a cycle.
4. select one vertex from each of these groups to obtain a total of 4^{i-1} vertices.
5. let $i = i - 1$ and repeat step 2 to 4 on these 4^i vertices until the graph is connected.

It is easy to verify that the minimum cutsize for these graphs is always 2.

The test results for the three types of graphs are displayed in Table 6 to Table 8 in which C_0 is the best cutsize found and γ' and γ are the cutsizes in excess of C_0 for F–M–H and S–A, respectively. The speedups, t' / t, are defined as the ratio of the CPU time spent by F–M–H, t', to the CPU time spent by S–A, t. All results tabulated for F–M–H are the average results of *at least eight* groups, where the CPU time of a group is the cumulative CPU time for 100 executions and the cutsize of a group is the best cutsize found in those 100 executions. All results tabulated for S–A are the average results of *at least eight* executions.

From Table 6 we observe a speedup of up to 5 for a random graph with 1,000 vertices and an average degree of 20. To better understand the relationship between the speedup and the quality of the final solution, we picked a random

N	d	CPU time (sec.)			Cutsize		
		t'	t	t'/t	γ'	γ	C_0
500	5	64.7	41.5	1.56	10	10	247
1000	5	154.0	42.1	3.66	31	32	463
500	20	209.7	49.4	4.24	11	12	1706
1000	20	487.4	91.7	5.32	45	47	3374

Table 6: Comparison with F–M–H on random graphs.

N	d	CPU time (sec.)			Cutsize		
		t'	t	t'/t	γ'	γ	C_0
500	5	66.4	1050.1	0.06	6	11	5
1000	5	204.2	905.7	0.23	14	14	20
500	20	174.5	2823.5	0.06	0	49	100
1000	20	388.5	1315.3	0.30	2	369	181

Table 7: Comparison with F–M–H on geometric random graphs.

N	CPU time (sec.)			Cutsize	
	t'	t	t'/t	γ'	γ
256	20.1	248.3	0.08	0	0
1024	128.6	126.3	1.02	7	7
4096	488.7	161.4	3.03	43	43

Table 8: Comparison with F–M–H on hierarchical graphs.

Exec.	CPU time (sec.)			Cutsize	
	t'	t	t'/t	γ'	γ
10	8.6	24.0	0.36	13	13
100	64.7	41.5	1.56	10	10
500	308.4	76.4	4.04	7	6
2000	1226.4	82.0	14.96	5	5

Table 9: Detailed comparison with F–M–H on a random graph with N=500 and d=5 (C_0=247).

N	CPU time (sec.)		Cutsize	
	t'	t	γ'	γ
256	20.1	248.3	0	0
1024	2574.2	1745.7	5	1
4096	9890.5	2755.5	42	18

Table 10: The best average results obtained (with reasonable amount of CPU time) by S–A and F–M–H on hierarchical graphs.

graph with $N = 500$ and $d = 5$ and vary the execution time. The test result on such a graph is shown in Table 9 in which the number of executions within a group is varied from 10 to 2,000. We observe a speedup of up to 15 for high quality solutions. In addition, the result indicates that using 100 executions per group seems to be a reasonable compromise between the computation time and the quality of the final solution.

In contrast to the good result for random graph, S–A performs relatively poorly on geometric random graph. We observe from Table 7 that S–A runs up to 17 times slower and gives worse results than F–M–H. The superiority of F–M–H over S–A on geometric graphs may be related to the special structure of these graphs. All edges of a geometric random graph are between pairs of vertices that are close by. S–A, being insensitive to any structure, performs poorly on this type of graph when compared to F–M–H which takes advantage of this structure.

Another kind of graph for which S–A does not perform well is hierarchical graphs. The optimal cutsize for these graphs is always 2. The test result for hierarchical graphs is displayed in Table 8. Although S–A performs poorly on this kind of graph, F–M–H does even worse. We observe a speedup of up to 3 on a hierarchical graph with 4,096 vertices. Furthermore, by comparing Table 8 with Table 10, we notice that the quality of the solutions obtainable by S–A is much higher than that obtainable by F–M–H if we increase the allotted CPU time. Both algorithms find the optimal cutsize for $N = 256$. However, as the number of vertices increases, the problem becomes increasingly difficult. Except for specially designed heuristics that exploit this kind of structure, we believe finding high quality solutions for hierarchical graphs is a very difficult problem because of the small optimal cutsize.

4. Conclusions and acknowledgements

We have shown that for problems that have little structure and for high quality solutions, S–A achieves significant speedups over specially designed heuristics for the TSP and the GPP. Although these two problems are not your typical day-to-day layout problems, quite a few problems in the layout area can be formulated as TSP and GPP. Since S–A competes favorably with well-designed heuristics on these well-studied problems, we believe the method can be extremely competitive in the layout domain where few good algorithms exist.

This research was supported by the Army Research Office under contract DAAL03-86-K-0158 and by the Office of Naval Research under contract N00014-85-K-0461.

References

[1] J. Beardwood, J. Halton, and J. Hammersley, "The Shortest Path Through Many Points," *Proc. of the Cambridge Philosophical Society,* Vol. 55, 299-327, 1959.

[2] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits," *IEEE Trans. on CAD,* Vol. 4, No. 1, 92-98, 1985.

[3] C. Fiduccia and R. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. of the 19th DAC,* 175-181, 1982.

[4] J. Greene and K. Supowit, "Simulated Annealing Without Rejected Moves," *IEEE Trans. on CAD,* Vol. 5, No. 1, 221-228, 1986.

[5] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: an Experimental Evaluation (Part I)," *Draft,* 1987.

[6] R. Karp, "Probabilistic Analysis of Partitioning Algorithms for the Traveling Salesman Problem in the Plane," *Math. of Oper. Res.,* Vol. 2, No. 3, 209-224, 1977.

[7] J. Lam, "An Efficient Simulated Annealing Schedule," Ph.D. Dissertation, Dept. of CS, Yale University, Fall 1988.

[8] J. Lam and J.-M. Delosme, "Performance of A New Annealing Schedule," *Proc. of the 25th DAC,* 1988.

[9] S. Lin and B. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Oper. Res.,* Vol. 21, 498-516, 1973.

[10] C. Sechen and K.-W. Lee, "An Improved Simulated Annealing Algorithm for Row-Based Placement," *Proc. of the IEEE ICCAD,* 478-481, 1987.

[11] J. Ullman, *Computational Aspects of VLSI,* Computer Science Press, 1984.