Annals of Operations Research, 21 (1989) 59–84

SERIAL AND PARALLEL SIMULATED ANNEALING AND TABU SEARCH ALGORITHMS FOR THE TRAVELING SALESMAN PROBLEM

Miroslaw MALEK, Mohan GURUSWAMY, Mihir PANDYA

Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas 78712, U.S.A.

and

Howard OWENS

Microcomputer Division, Motorola Inc.

Abstract

This paper describes serial and parallel implementations of two different search techniques applied to the traveling salesman problem. A novel approach has been taken to parallelize simulated annealing and the results are compared with the traditional annealing algorithm. This approach uses abbreviated cooling schedule and achieves a superlinear speedup. Also a new search technique, called tabu search, has been adapted to execute in a parallel computing environment. Comparison between simulated annealing and tabu search indicate that tabu search consistently outperforms simulated annealing with respect to computation time while giving comparable solutions. Examples include 25, 33, 42, 50, 57, 75 and 100 city problems.

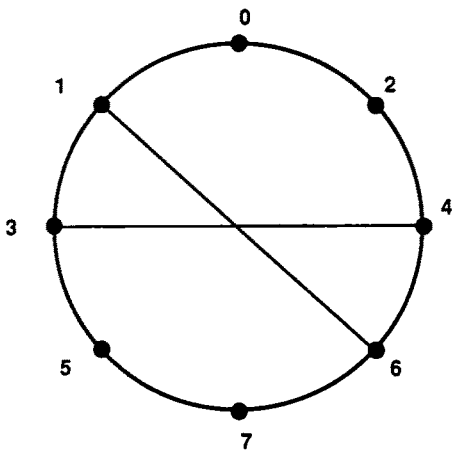
1. Introduction

The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization research. This is because it is easily defined, and improved algorithms to find optimized solutions to this problem can be adapted to an entire class of NP-complete problems.

Our objective is to implement combinatorial optimization algorithms such that they may execute in parallel and exchange data periodically. The goal is to study the potential performance improvements as compared to a single processor implementation of the traveling salesman problem.

In particular, we are interested in comparing the time efficiency and cost of simulated annealing [1] and tabu search [2] algorithms. Each of these algorithms have the ability to avoid or escape local minima in searching for the global optimal solution. However, the method of reaching the optimal solution is very different as will be described later. These universal search algorithms are tested on a specific heuristic used for the TSP problem.

The traveling salesman problem consists of finding the shortest Hamiltonian circuit in a complete graph where the nodes represent cities. The weights on the



COMPUTER REPRESENTATION OF TOURS:

[0 2 4 6 7 5 3 1]

[0 2 4 3 5 7 6 1]

Fig. 1. A 2-opt move.

edges represent the distance between cities. The cost of the tour is the total distance covered in traversing all the cities. Although we did not consider the possibility of two cities not having a direct route this can be accounted for by assigning the distance as some arbitrarily large value. Experiments were conducted on seven well-known problems from the literature [3], namely, the 25 city [2], 33 city [4], 42 city [5], 50 city [6], 57 city [4], 75 city [6] and 100 city [7] problems. Other algorithms and problems for the TSP are described in [9] and [10].

Each of the algorithms we will present takes an arbitrary starting solution and makes incremental changes to it to search for a better solution. We use moves of the 2-opt heuristic [8] to make these incremental changes in the tours. Basically, 2-opt move (exchange) is the swap of two non-adjacent edges.

In fig. 1 we have a tour on eight nodes. As an example we delete the edges between nodes 4 and 6 and between nodes 3 and 1. We replace them with edges between nodes 6 and 1 and between nodes 3 and 4. These two edges are the only ones which will maintain a Hamiltonian circuit. Our computer algorithms represent a tour as an array of cities such that the index into the array corresponds to the position in the tour. The 2-opt swap can then be performed simply by reversing the order of all the cities in the tour from node 6 to node 3. The result of the 2-opt swap is subtracting the cost of two edges from the original tour cost and adding the cost of the two new edges to the tour cost. This edge exchange is very economical in terms of computation time.

2. Description of algorithms

2.1. SIMULATED ANNEALING

2.1.1. Physical analogy

Simulated annealing uses the analogy of annealing to guide the use of moves which increase cost. Annealing is a process which finds a low energy state of a metal by melting it and then cooling it slowly. Temperature is the controlling variable in the annealing process and determines how random the energy state is. Let's take a simple intuitive account of why simulated annealing works. Consider an energy landscape as shown in fig. 2 [11]. The energy barrier equally divides the energy landscape into the two local minima A and B. A ball placed at random on the landscape and allowed to only travel downhill has an equal chance of ending up in A as in B.

To increase the probability of the ball ending in B let us use shaking to simulate temperature in the annealing process. Due to a smaller energy barrier shaking is more likely to cause the ball to cross the energy barrier from A to B. Also, the harder we shake the more likely the ball will cross the energy barrier in either direction. We should therefore shake hard at first and then slowly decrease the intensity of shaking. As we decrease the intensity of shaking, we also decrease the probability of crossing the barrier in any direction. However, the ratio of probabilities of crossing from A to B versus from B to A is increased. Finally, the result is the ball stays on the side of the barrier which offers the lowest energy state.

2.1.2. Simulated annealing algorithm

The main body of the algorithm consists of two loops, where the inner loop is nested within an outer loop. The **inner loop**

```
WHILE (equilibrium not reached)
  Generate-next-move( )
  IF (Accept(Temperature [1], change-in-cost))
    Update-tour( )
ENDWHILE
```

runs till an equilibrium is reached. In this loop, a possible move is generated using the 2-opt exchange and the decision of accepting the chosen move is made using an *accept* function. If the move is accepted, it is applied to the current tour to generate the next tour state. Equilibrium is reached when large swings in average energy (miles) no longer occur at this temperature.

[1] Temperature, by analogy to physical annealing process, represents the control variable for accepting uphill moves.

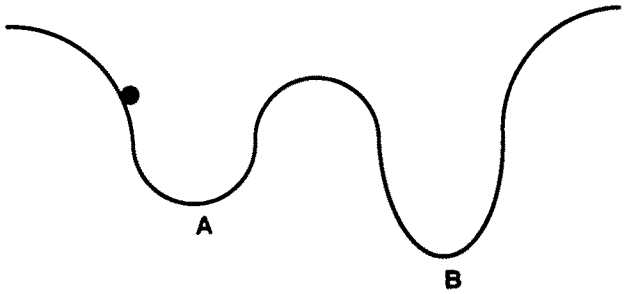


Fig. 2. Energy landscape.

The outer loop

```
WHILE (stopping criterion not met)
  INNER LOOP
  Calculate-new-temperature( )
ENDWHILE
```

checks for the stopping condition to be met. Each time the inner loop is completed, the temperature (T) is updated using an *update temperature* function and the stopping criterion is checked again. This continues until the stopping criterion is met.

The accept function

```
IF (ΔC < 0) RETURN (TRUE)
ELSEIF (e^(-ΔC/T) > random(0, 1)) RETURN(TRUE)
ELSE RETURN(FALSE)
```

assigns a probability of accepting a move based on the current temperature and the change in cost (ΔC) which would result in the tour if the move is accepted. If ΔC is negative, meaning the cost would go down, a probability of one is assigned to acceptance. Otherwise the probability of acceptance is assigned the value

$$\textit{probability of acceptance} = \exp(-\Delta C/T).$$

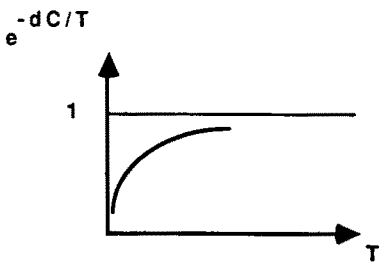


Fig. 3. Probability function for annealing process.

A randomly generated number is used to test whether the move is accepted. The probability function takes the shape shown in fig. 3. Notice at high temperatures most uphill moves are accepted with a high probability regardless of the increase in cost. Eventually, as temperature decreases, only downhill moves will be accepted.

2.2. TABU SEARCH

Tabu search is an optimization technique for solving permutation problems [2]. In this technique, we start with an arbitrary permutation and make a succession of moves to transform this permutation into an optimal one (or as close to the optimum as possible). In this particular setting, it is equivalent to starting with a randomly generated tour and making a succession of edge swaps trying to reduce the cost of the tour until we can find the minimum cost. As in simulated annealing, we use the 2-opt exchange as the basic move.

2.2.1. The algorithm

The hill climbing heuristic is greedy and gets stuck at local optima. But, this can be avoided by forming a simple tabu search procedure. Now, in order to guide this hill climbing process out of the local optimum and to continue exploration, we have to identify the *swap attributes* that will be used to create a tabu classification. The attributes could be one of the following:

- the cities involved in the swap, or
- the *positions* they occupy before/after the swap, or
- the direction in which the cities move in the swap.

The tour of the cities is represented in a one-dimensional array format, with the array index denoting the *position* of the city in the tour. If the city moves from a lower index to a higher index during a swap, then it is said to move right. Conversely, if it moves from a higher index to a lower one, then it is said to move left.

We also need to identify the tabu classifications based on the attributes so that we can specify a set of moves as tabu. These attributes are discussed in detail in the next section. Figure 4 shows the tabu strategy superimposed on the hill climbing heuristic.

The main distinction comes in the decision making stage of the inner loop. The algorithm examines all the swaps of the current tour and keeps track of the *best-swap-value*, however, those that are classified as tabu are rejected if they do not satisfy the *aspiration criteria* [2]. In other words, we restrict the set of available swaps. The tabu status of the move is overridden and the move is accepted if the swap-value satisfies the aspiration level. The *best-swap* among all the available

[2] This is a concept based on the observation that it may be advantageous to override a tabu restriction if it promises a better solution or a new search space.

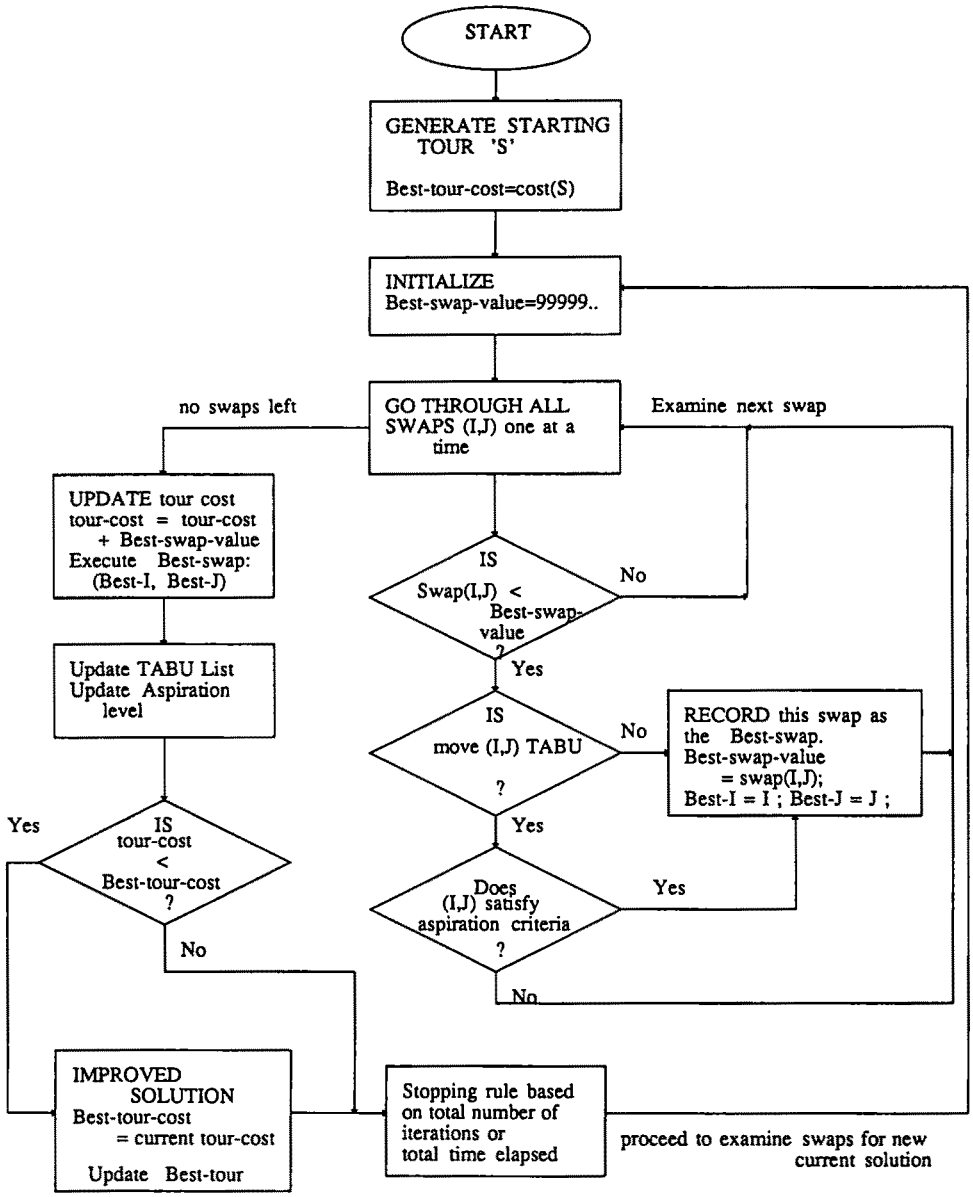


Fig. 4. Tabu search.

swaps for the current tour is obtained at the exit of the inner loop. In the hill climbing method, the best-swap-value is usually negative indicating a reduction of the current tour cost. When it becomes positive, the process has reached its terminating condition.

In tabu search, the best-swap is executed regardless of the sign of the best-swap-value. The best swap from the inner loop is accepted even if it results

in a higher tour cost. This helps the process to climb out of local optima. The outer loop keeps track of the *best-tour* and its cost. The tabu list is also suitably updated by including the current move made. The stopping criteria is usually a fixed set of iterations or a fixed computation time specified in the input.

2.2.2. Tabu conditions

This section presents examples of the move attributes and the tabu restrictions based on these attributes. In our implementation we select only one tabu condition for a specific tabu search process.

1. **Vector (*I*, *J*, POSITION(*I*), POSITION(*J*))–**
   this vector is maintained to prevent any swap in the future from resulting in a tour with city *I* and city *J* occupying POSITION(*I*) **and** POSITION(*J*) respectively.
2. **Vector (*I*, *J*, POSITION(*I*), POSITION(*J*))–**
   the same vector to prevent a swap resulting in city *I* occupying POSITION(*I*) **or** city *J* occupying POSITION(*J*).
3. **Vector (*I*, POSITION(*I*))–**
   to prevent city *I* from returning to POSITION(*I*).
4. **City *I*–**
   to prevent city *I* from moving left of current position.
5. **City *I*–**
   to prevent *I* from moving in any direction.
6. **Vector (*J*, POSITION(*J*))–**
   to prevent city *J* from returning to POSITION(*J*).
7. **City *J*–**
   to prevent city *J* from moving right of current position.
8. **City *J*–**
   to prevent *J* from moving in any direction.
9. **Cities *I* and *J*–**
   to prevent both from moving.

Conditions 3 through 9 have been established by assuming that cities *I* and *J* are identified such that POSITION(*I*) < POSITION(*J*). It is obvious that condition 1 is the least restrictive and 9 is the most restrictive. Conditions 3, 4 and 5 have increasing restrictiveness.

3. Serial and parallel implementation of simulated annealing

3.1. IMPLEMENTATION OF SIMULATED ANNEALING

To implement the simulated annealing algorithm, as described earlier, requires specifying the stopping and equilibrium criteria, and the update temperature rule. The stopping criteria chosen for the algorithm is for the temperature to reach a

specified value. This stopping temperature is chosen such that the probability of accepting an uphill move is very close to 0. We make the typical assumption that the equilibrium is reached after a fixed number of iterations. The update temperature rule is

new-temperature = α * temperature,

where α is a constant less than one.

The consequence of choosing these parameters to be simple constants is some increase in computation time. The simulated annealing algorithm implemented has as inputs the initial temperature, the number of iterations to simulate equilibrium, and α.

3.2. PARALLEL IMPLEMENTATION OF SIMULATED ANNEALING

There are several approaches to parallelization of algorithms. The most common one is based on dividing (partitioning) the problem such that several partitions could be run in parallel and then merged. This method has good potential if the traveling salesman problem is based on real cartographical map. Then the probabilistic technique, as suggested by Held and Karp [12] could be implemented in parallel. Since this approach is not general for the traveling salesman problem and is efficient only for TSP problems with at least hundreds of cities due to interprocessor communication overhead, we still need another approach to solve each partition or the entire problem. Our basic idea is to run several processes, for the entire problem, in parallel and periodically compare the results. The search is then continued by all the processes from a common good solution.

First, a main process is generated which reads in the problem definition. Then it creates a set of child processes on separate processors, each running an annealing algorithm with different parameters (starting temperature, α, number of iterations to equilibrium). After specified time intervals, the child processes are halted and the main process compares their results. It selects a *good* solution for the child process to continue with. A *good* solution might be the one with the least cost. To prevent cycling, an alternative tour is passed back if the tour is repetitive.

During the parallel search, a process may start off with an entirely different tour after the previous results have been compared. This tour might not correspond to the current annealing temperature. The following strategies for updating the temperature have been considered:

1. Continue with the same temperature after swapping.
2. Update temperatures in proportion to change in cost.
3. Reset temperature to starting value.

After some experimentation, it was found that the first two strategies did not lead to good solutions, while a modification of the third strategy turned out to be very effective. In this case, each process is annealed at a faster rate by keeping α small

and reducing the number of iterations to reach equilibrium. This in turn allows each process to complete its annealing schedule before swapping the results. The temperature is reset to the starting value after the swap and this causes the annealing process to begin all over again from the new tour. A memory function assures this new tour is not repetitive.

4. Serial and parallel implementation of tabu search

4.1. IMPLEMENTATION OF TABU SEARCH

4.1.1. Data structures

In order to determine the tabu status of a move and update the tabu list efficiently, we need well-designed data structures. As an example, the tabu identification and tabu-list update for one of the tabu conditions (condition-4) is described below.

There are two lists, *tabu-left* and *tabu-list*. The first list, *tabu-left*, indicates which cities are prevented from moving left of their current position. The second list, *tabu-list*, contains a fixed number of cities that had been moved to the right in the last k iterations (k is the *tabu-list size* which is an input parameter).

Updating the tabu-list with a new city i which was moved right is done by incrementing an index (*ring-index*) to the tabu-list and overwriting the city i at this new index position. This automatically removes the tabu status of the city which was residing in the new position of the index. In order for the index to stay in range of the list, the incrementing is done using a mod operator: new-ring index = (ring-index + 1) mod tabu-size. Similar data structures have been implemented for other tabu conditions.

4.1.2. Aspiration criterion

The aspiration criterion we have used is straightforward. Any tabu move that reduces the current tour-cost to a value below the best-tour-cost obtained by the process so far is accepted. When the move results in a tour-cost lower than the best-tour-cost, it indicates a new path not previously visited and so the move can no longer be termed as tabu. This simple aspiration criterion is:

tour-cost + swap-value(I, J) < best-tour-cost

4.1.3. Tabu list size

This parameter needs experimental tuning. It can be observed that for highly restrictive tabu conditions the tabu list size has to be smaller than for lesser restrictive conditions. If the tabu list size is small, cycling phenomena will be evident, whereas, if it is large, the process might be driven away from the vicinity of global optimum. The optimum tabu list size will be the one which is long enough to prevent cycling but small enough to explore a continuum of solution space. Experimental results with the 42 and 57 city problems have shown that for

tabu conditions 1, 2, 3 and 6, list sizes of the order of the size of the problem is good. For conditions 4, 5, 7 and 8, the list size ranging from 7 to 30 gave us good results.

4.1.4. Long term memory

This is a function designed to enable the process to explore new areas of the solution state space. Applying tabu search from several different starting points is more likely to perform better than exploring from one starting point. Instead of starting at randomly chosen starting points, if we can provide a purposeful alternate starting point for search, we might be able to reach the optimum faster. In our approach for an alternate starting point, we use a long term history function that maintains the edges visited by the process and generates a starting point consisting of edges that have been visited the least. We maintain a two dimensional array of occurrence of each edge. After each 2-opt move, the entries corresponding to all the edges in the new tour are incremented. After a specified number of iterations, a new starting tour is generated based on the edges that occur least frequently. The results obtained using this memory function are very encouraging. We were able to reach the optimum results for the 42 and 57 city problems consistently from different random starting points.

4.2. PARALLEL IMPLEMENTATION OF TABU SEARCH

As in parallel simulated annealing, each processor executes a process which is a tabu search algorithm with different tabu conditions and parameters. Here also, the child processes are stopped after a specified interval of time, the results compared and then restarted with a *good* solution.

Each tabu process takes a different path because of the different parameter set it is given causing a wider area to be searched. After a swap, bad areas of the solution space are eliminated and exploration of promising regions is carried out. The child processes are restarted with empty tabu lists as it is pointless to apply previous restrictions and penalize a different tour.

The long term memory function was removed as it interfered with swapped data. Any call to long term memory function would nullify the new swapped tour, whereas, any swap would nullify the tour obtained by calling long term memory function. Long term memory might be useful for very large problems whose time interval between swaps is large.

5. TSP experiments

5.1. PARALLEL ENVIRONMENT

Our parallel computation environment consists of a Sequent Balance 8000 computer running the DYNIX [3] operating system, a version of UNIX 4.2 bsd [4].

[3] DYNIX is a trademark of Sequent Computer, Inc.

[4] UNIX 4.2bsd is a Berkely Software Distribution version of UNIX, UNIX is a trademark of AT&T.

The Sequent Balance 8000 is a multiprocessor computer consisting of ten homogeneous 32-bit processors. All of our programs are written in PPL (Parallel Program Language) [13], which is a set of extensions to the C programming language [14]. PPL is used for the ease of access to the parallel processing features of the Sequent Balance 8000.

The objective in our experimentation is a comparison of the execution time and cost of simulated annealing and tabu search algorithms. The experiments are presented in four parts: simulated annealing, parallel simulated annealing, tabu search, and parallel tabu search. In each part we are interested in achieving reasonable execution time and cost, not necessarily optimal solutions. From the results of these experiments we make comparisons and draw conclusions.

5.2. SIMULATED ANNEALING

Each execution of the simulated annealing algorithm takes as input parameters the number of iterations to approximate equilibrium, the starting temperature, and the cooling rate α. These input parameters allow the algorithm to be tuned for a specific problem. Experiments were performed for the 25, 33, 42, 57 and 100 city problems with proven optimum solution and for 50 and 75 city problems where only best solutions to date are known [3].

For each problem the first step in the experimentation was to execute the program several times, changing the input parameters in a tuning process in order to obtain a set of input parameters which give the most reasonable execution time and cost. Initially, each execution was performed on the same starting tour. These parameters were then tested against several starting tours. The results were used to determine if more tuning was necessary.

During tuning of the algorithm we varied the input parameters throughout a wide range. The number of iterations to simulate equilibrium ranged from 4 to 20. The starting temperatures varied from 10 all the way up to 10000. α values of 0.98 down to 0.50 were tried. For each of these problems a different input parameter set was chosen from the tuning process.

The 25 city problem was easily solved with our simulated annealing algorithm. The published optimum solution of 1711 miles was reached in every case from each of our randomly generated starting tours. The best input parameters obtained from tuning the algorithm resulted in execution time of 55 seconds. Results from applying our simulated annealing algorithm on the 33 city problem was nearly as good. Only once did the algorithm fail to find the published optimum of 10861 miles. The time required for execution of the problem was 115 seconds.

The process of tuning the algorithm for the 42 city problem resulted in several input parameter sets reaching the same minimum cost solution. The parameter set which found the minimum cost in shortest time took 185 seconds to execute. Using this input parameter set, resulting tour costs ranged from 699 miles to 705

miles. Roughly half of the results had a tour cost of 699 miles for our set of randomly generated starting tours.

The optimal solution to the 42 city problem is published to be 699 miles [5]. We observed this tour to be (0 1 2...40 41). Ordinarily the simulated annealing algorithm is expected to result in the best tour at the end of the algorithm's execution. But we noticed, by using a monitor to record the best tour during a given run, that sometimes a better solution is found earlier than the final one. Exploring this phenomenon further we observed a good example of what was happening. As the algorithm reached a low temperature the optimal solution of 699 miles was found. However, the algorithm then accepted a move which resulted in a cost increase of 10 miles even though the probability of acceptance was only 0.0007. The resulting tour was (0 1 2...11 12 14 13 15 16...40 41). The next move found which decreased the cost resulted in the tour (0 1 2...11 12 16 15 13 14 17 18...40 41) with a cost of 701 miles. After this set of moves, it requires the reverse set of moves to return to the optimal solution. But the uphill step needed had only a 0.07% chance of being accepted. In fact the algorithm remained stuck with a final tour cost of 701 miles. Also note the resulting tour had three edges which are not a part of the optimal solution. We therefore feel the edge exchange of the 3-opt heuristic may have overcome these problems.

The 57 city problem proved to be more difficult. The best result achieved during tuning of the algorithm for this problem had a tour cost of 12955 miles. The shortest time to find this cost was 673 seconds. Using these parameters and applying the algorithm to random starting tours resulted in the same best tour about 15% of the time. The published optimum tour cost for this problem is in

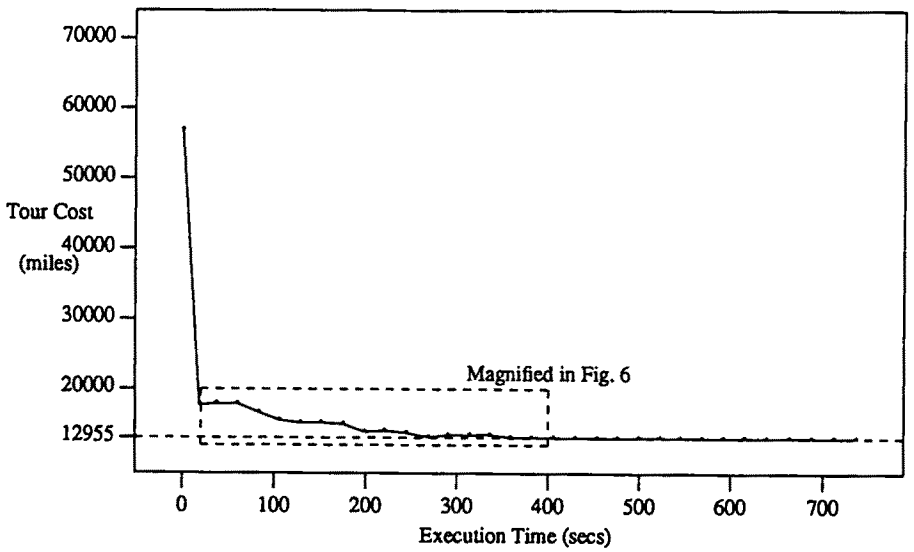


Fig. 5. Tour cost versus execution time for 57 city problem using simulated annealing.

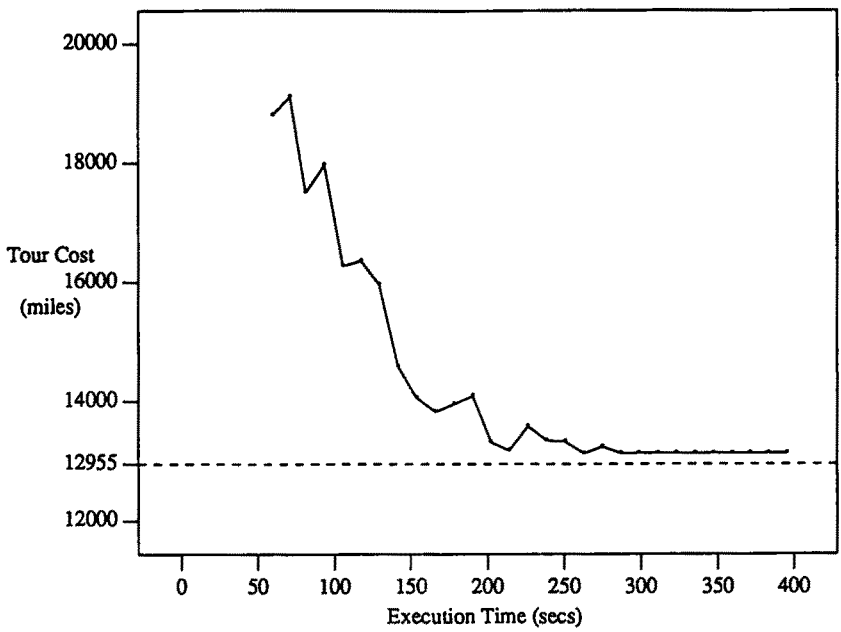


Fig. 6. Magnification of a section of fig. 5.

fact 12955 miles [4]. Although reaching the optimum 15% of the time may seem small, attempting to reach it more often would require large increases in time. Besides, all non-optimal results were within less than one percent of optimum. Resulting tour costs found ranged from 12955 miles to 13042 miles.

Figure 5 shows graphically how the simulated annealing algorithm progressed on the 57 city problem for an example starting tour. Each point on the graph represents the tour cost at an equilibrium point and the time it was reached. Notice that even equilibrium points later in time (smaller temperature) can have higher cost. Figure 6 contains an enlargement of a portion of fig. 5 to provide better detail.

The 100 city problem was the most difficult. The published optimal solution is 21282 miles when the problem is executed using floating point numbers. This corresponds to a tour cost of 21247 miles when the elements of the cost matrix are truncated, as we did in our experiments for the 100 city problem, for the same tour. While tuning our algorithm for this problem we chose cooling schedules which took up to 10 hours to complete. However, we never reached the optimal solution. The cooling schedule we finally choose required 4900 seconds to execute. Using this schedule our results ranged from 21267 miles to 21356 miles on our randomly generated set of starting tours.

The final two problems we considered have no proven optimal solutions. The best published solution to the 50 city problem is 430 miles and 553 miles for the 75 city problem. Executing our simulated annealing algorithm on the 50 city

problem resulted in two solutions better than the best published for our randomly generated starting tours. Our results ranged from 425 miles to 438 miles, including solutions of 425 miles and 429 miles. The time to execute our algorithm on this problem was 280 seconds.

For the 75 city problem our algorithm resulted in 4 out of 6 solutions of better quality than the best published on our set of randomly generated starting tours. The two remaining solutions included one which equalled the best published, and one two miles longer. The solutions beating the best published had tour costs of 541, 542, 542, and 550 miles. The time to execute our algorithm on this problem was 315 seconds.

5.3. PARALLEL SIMULATED ANNEALING

The next part of our experiments dealt with parallel implementation of simulated annealing. Recall the basic algorithm for our implementation was to execute several simulated annealing processes in parallel on multiple processors and periodically compare the results. Each time the results are compared all processes continue with a common tour. The period at which results are compared is based on time. However, all processes must be at equilibrium. Each process executes a simulated annealing algorithm with different input parameters. In our experiments we attempted three strategies for implementing the cooling schedule for our parallel simulated annealing algorithm.

The first strategy for the cooling schedule was simply to continue with the schedule from the point from which it was interrupted for tour comparisons. Each process had its own cooling schedule which corresponded to a reasonable schedule for a serial implementation. Experimentation with this strategy provided unsatisfactory results. Our first impression was that the temperature of each process when applied to the new tour, was not representative of the state of the new tour.

Our next strategy was to update the current temperature each time tours were compared by some proportional amount based on the new tour cost. In other words, each time a process stopped for the tour comparisons, not only did the process get a new tour to continue with, but it also updated its current temperature in proportion to the change in cost of the two tours. This added a new variable to the algorithm which set the proportion at which the temperature was updated. Unfortunately, our experiments failed to find results which were satisfactory. Not surprisingly, we observed at this point that the simulated annealing cooling strategy was very sensitive to being disturbed.

Finally, we decided to let the cooling schedule complete before comparing results. The cooling schedule in this strategy was chosen differently than the previous two. In this case, the cooling rate of each schedule was selected such that it could be repeated several times in the same time period that a reasonable serial implementation required. The strategy was to allow each process to complete a

different abbreviated schedule, compare results, and repeat the annealing process on the resulting new tour. Although abbreviating the cooling schedule is believed to be contrary to the theory of simulated annealing, experimentation with this strategy demonstrated promising results. This departure from the framework of simulated annealing might be viewed as a variation of probabilistic tabu search [2] in a relaxed form.

Our experiments were conducted by giving one process a very high starting temperature, a small number of iterations, and a small value for α. Each additional process was given a progressively smaller starting temperature, larger number of iterations, and larger α. The input parameters for each process were chosen so that all processes required about the same amount of time to complete their cooling schedule. The intuitive idea was to have processes choosing the lowest energy side of different size energy barriers.

Our implementation of parallel simulated annealing gave optimal results for the 25 city problem for every randomly generated starting point. Not only was the optimal tour found, but it was found on the first iteration, before any tour exchange was performed. This implies that even though the algorithm took 58 seconds to execute five iterations, only one 12 second iteration was needed. The 33 city problem proved to be very similar. For this problem the optimal solution of 10861 miles was found for each of our starting tours. This optimum solution was reached on the first or second iteration for each execution. The time required to execute the algorithms' five iterations was 72 seconds, while the optimum was always reached within 27 seconds.

Our parallel simulated annealing algorithm gave optimum results for the 42 city problem for all but one randomly generated starting points. The one execution which failed to produce the optimal solution found a tour of 701 miles. The time required to execute the algorithm was 210 seconds, a little longer than required by the serial algorithm. However, confidence of reaching the optimal solution has increased significantly for the parallel implementation.

For the 57 city problem our parallel simulated annealing algorithm executed in 225 seconds while the serial implementation required 673 seconds, about one third of the time. In fact the optimal solution was reached in as little as 94 seconds, eight times faster than the serial version with only four processors!

Figure 7 shows graphically how our parallel simulated annealing algorithm progressed on the 57 city problem for an example starting tour. As in the serial version, each point represents the tour cost at an equilibrium point and the time it was reached. Figure 8 contains an enlargement of a portion of fig. 7 to provide better detail.

As in our serial simulated annealing algorithm, the 100 city problem was the most difficult for parallel simulated annealing algorithm. For this problem we allowed our algorithm to execute 15 iterations. This still required only 900 seconds, compared to 4900 seconds for the serial version. The quality of solution is nearly the same for both techniques.

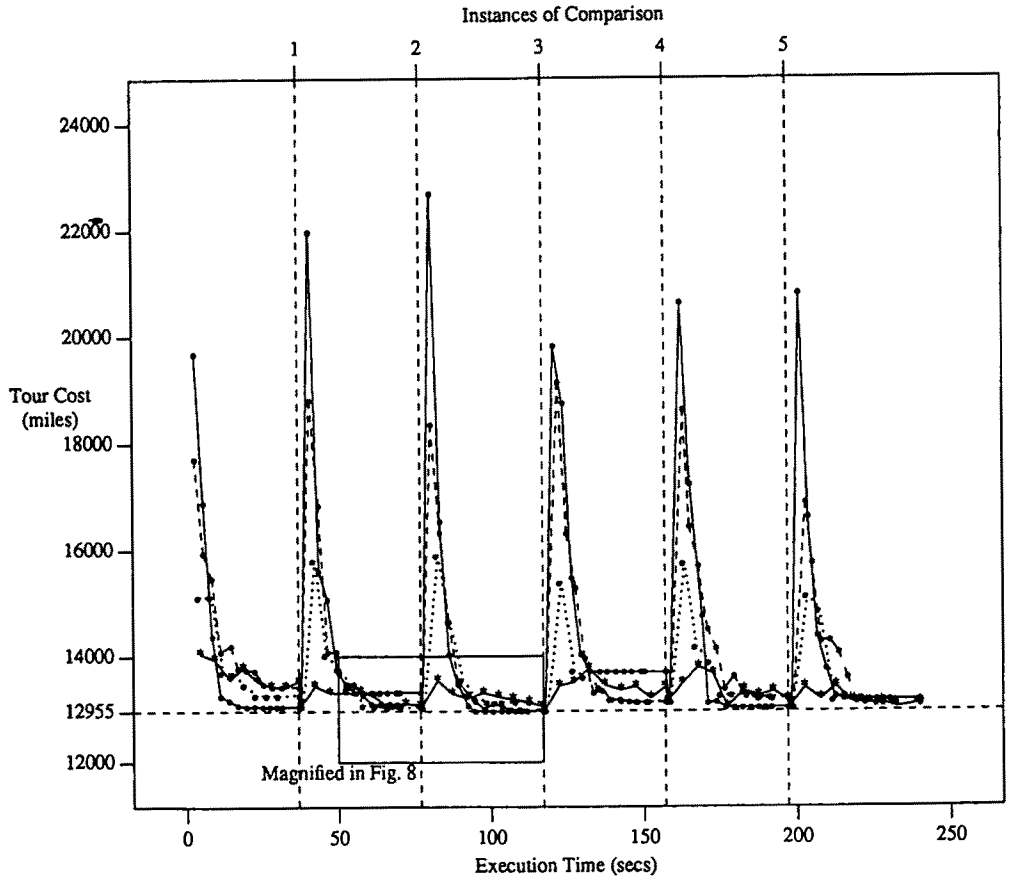


Fig. 7. Tour cost versus execution time for 57 city problem using parallel simulated annealing.

The difficulty of the 100 city problem brings out some very interesting ideas of comparison. It has become obvious that the harder the problem, the more advantageous the parallel simulated annealing algorithm is. In fig. 9 we have plotted the speed up of our parallel simulated annealing algorithm versus the number of processors. In this graph, speedup is defined to be the ratio of time for the serial simulated annealing algorithm execution to the time for the parallel simulated annealing algorithm execution. Notice that the speedup for two processor system was 11 times. Intuitively this phenomenal speedup could be attributed to the fact that each processor uses an abbreviated and different cooling schedule. Figure 10 shows a family of curves giving solution quality versus time the parallel simulated annealing algorithm is allowed to execute. Each curve is for a given number of processors. Each point represents the cost of the best solution the algorithm has considered at a given time. These points are taken at the tour exchange times for the algorithm.

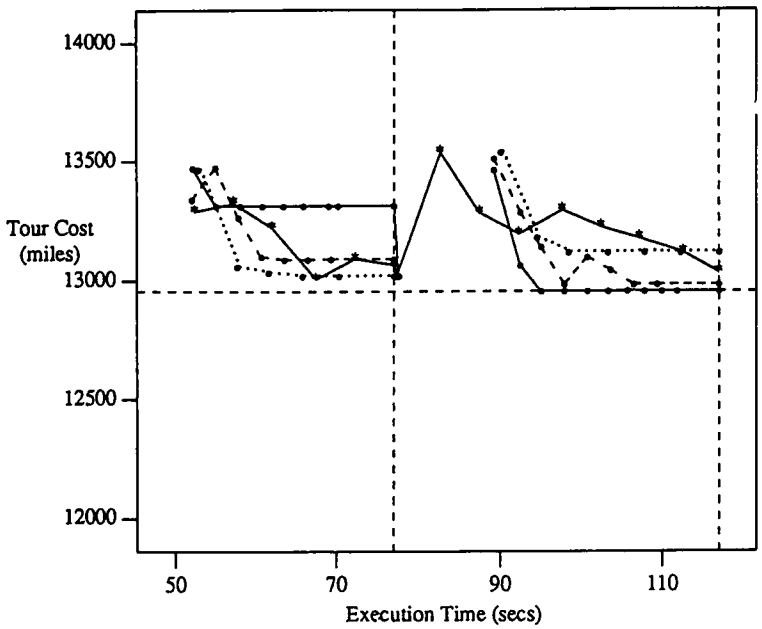


Fig. 8. Magnification of a section of fig. 7.

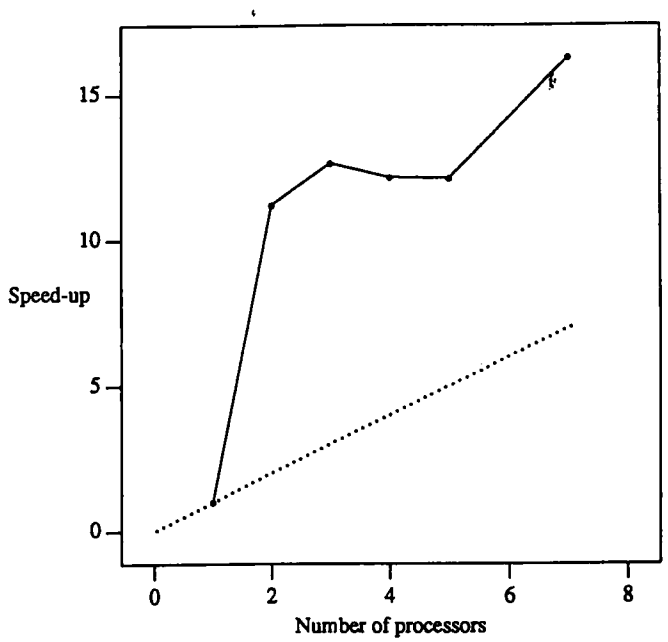


Fig. 9. Speed-up vs. number of processors.

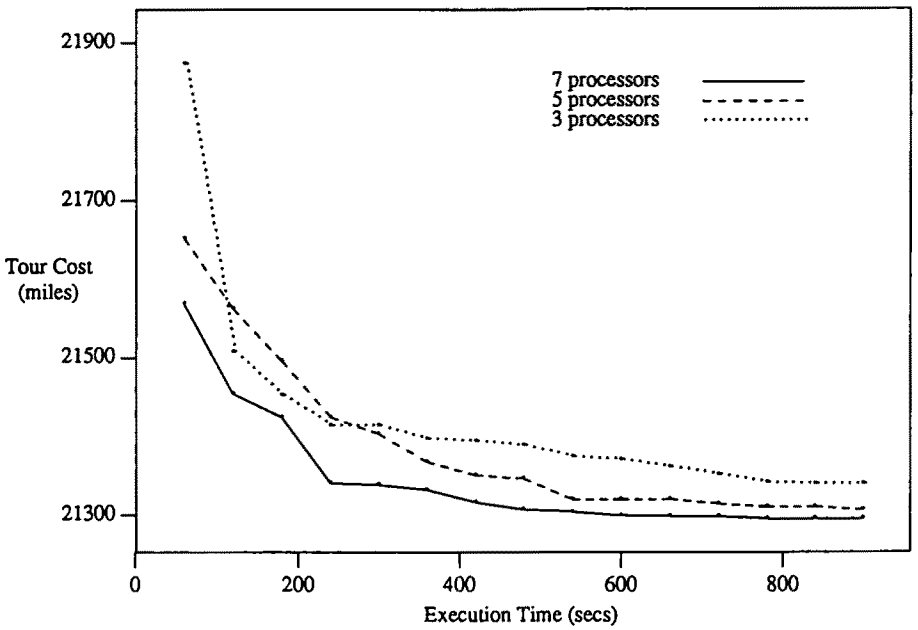


Fig. 10. Quality of solutions as a function of number of processors.

Solutions from execution of our parallel simulated annealing algorithm on the 50 city problem ranged from 427 miles to 435 miles. The best published solution of 430 miles was surpassed in 50% of our set of starting points. The time for execution of 5 iterations was 258 seconds, slightly better than the serial version. The average quality of solution is also a little bit better for the parallel version.

Execution of the 75 city problem resulted in the solutions better than the best published in all but one of our set of starting tours. The best tour cost obtained was 540 miles compared to the previously published best tour cost of 553 miles. As in the 50 city problem, the time to execute the parallel algorithm was slightly less than the serial algorithm while improving the average quality of result.

5.4. TABU SEARCH

The first stage in developing the tabu search algorithm was the implementation of the hill climbing heuristic. The hill climbing algorithm was then transformed into the tabu search algorithm using the nine different tabu conditions discussed earlier. The tabu search process has the following input parameters:

- tabu condition,
- tabu list size and
- total number of iterations.

The tabu condition and the tabu list size are two interdependant parameters and the algorithm is very sensitive to both of them. A smaller tabu list size for a

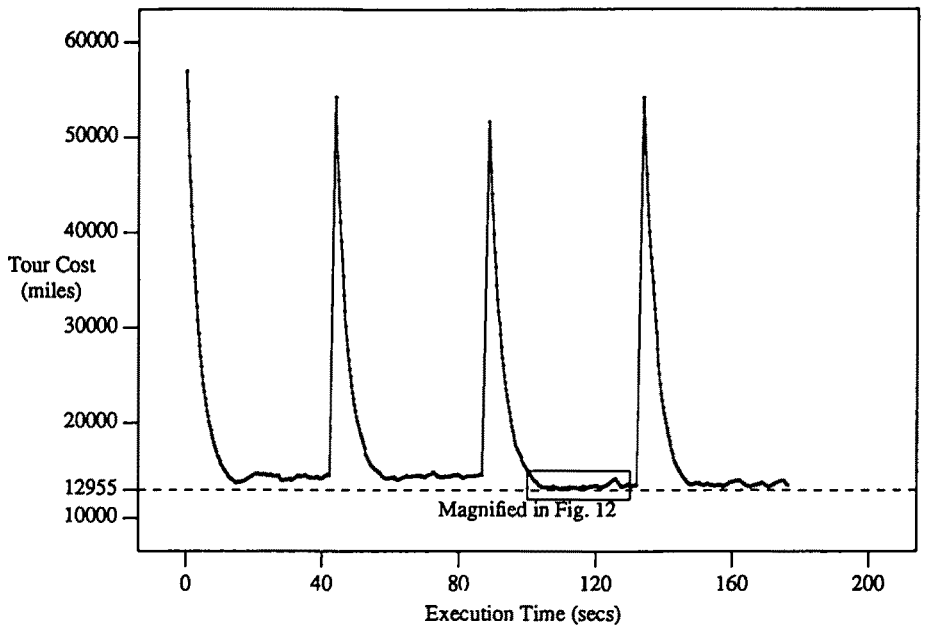


Fig. 11. Tour cost versus execution time for 57 city problem using tabu search algorithm.

weaker tabu condition will result in cycling, whereas, a larger tabu list size for a stronger tabu condition could drive the search process away from the global optimum. A compromise had to be reached and experiments were conducted for all the nine tabu conditions to find out a reasonable range of tabu list sizes for each of these conditions. Generally, tabu list sizes from one-fourth to one-third the number of cities for conditions 4 and 7, and about one-fifth for conditions 5, 8 and 9 gave the best results for the problems tested. Conditions 1, 2, 3 and 6 required tabu list sizes in the vicinity of the problem size. On an average, tabu conditions 4 and 7 produced better results in a shorter time than the other conditions.

The next step was to include the long term memory function. This function requires the tuning of an additional input parameter–the number of iterations in tabu search before generating a new starting tour. The algorithm must be given sufficient search time in its current path before generating a totally different starting tour. The performance of the algorithm with long term memory function was compared with the original tabu search algorithm by making the total number of iterations and the starting points identical for both versions. For every run, the algorithm with the memory function outperformed the simpler version in both computation time and the quality of the solution.

Figure 11 shows the search process of the tabu algorithm for a particular starting instance of the 57 city problem. Each point in the graph indicates the tour cost after the best swap was made by the algorithm. The peaks in the graph

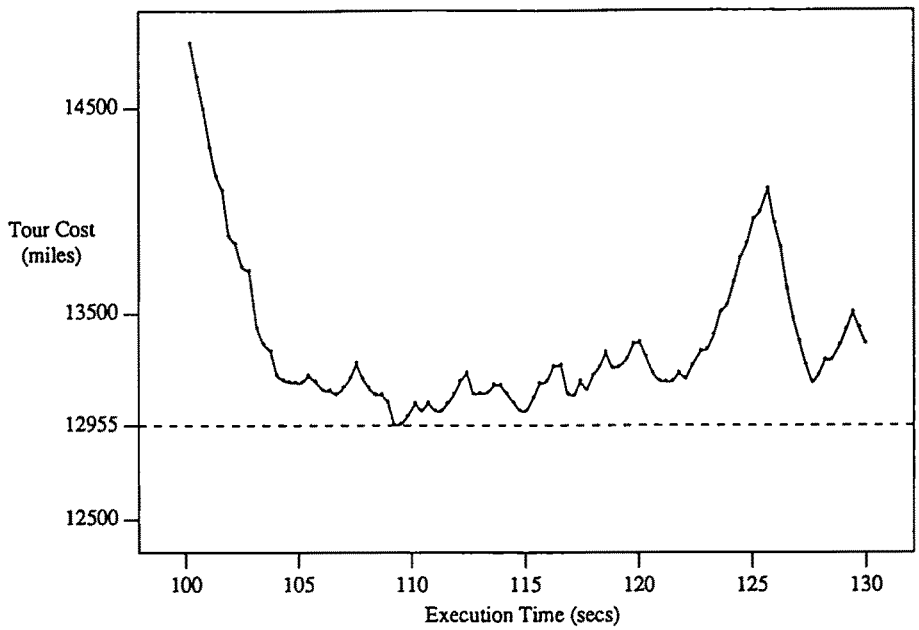


Fig. 12. Magnified graph of a section of fig. 11.

indicate the new starting points in the search process generated by the long term memory function. Figure 12, an enlargement of a portion of fig. 11, clearly shows the tabu search process escaping the local optima by accepting uphill moves and as shown in fig. 11, finding the global optimum.

On the 25, 33 and 42 city problems, the tabu search algorithm produced the published optimal results of 1711, 10861 and 699 miles respectively from each of the randomly selected starting tours. The number of iterations required to produce the optimum result for the 25 city problem ranged from 18 to 38 for a total run time of less than 12 seconds in each case. For the 33 city problem, the range was from 29 to 271 iterations taking about 30 seconds. For the 42 city problem, the optimum was attained in less than 30 seconds in each run with the number of iterations ranging from 33 to 117.

The 57 city problem proved to be more difficult with the optimum of 12955 miles attained twice with six randomly selected starting tours. But, the worst tour cost in these runs was only 13067 miles. The number of iterations for these random starting tours to reach the best tour varied from 73 to 897 iterations requiring a maximum of 262 seconds. The best solution found by tabu search for the 100 city problem was 21317 miles taking 1193 iterations (about 16 minutes).

For the 50 and 75 city problems whose optimal solutions are unknown, tabu search easily beat the best previously published results. The best solution for the 50 city problem had a tour length of 426 miles and tabu search equalled or bettered the best previously known solution of 431 miles in 7 out of 12 runs

conducted. The average time at which the best solution was found in each run was 54 seconds. Tabu search superceded the previous best result of 553 miles for the 75 city problem in 11 out of 12 runs. The best solution had a tour length of 537 miles and the average time taken to reach the best solution in each run was 112 seconds.

5.5. PARALLEL TABU SEARCH

In the parallel implementation, the parameters of each tabu search process were chosen differently to ensure that their search space did not overlap. In addition, the long term memory function was removed because of its interference with the swapped data. Since the main process compares the results of the child processes and decides on a good starting tour, the call to the long term memory function would nullify the decision made by the main process.

All the experiments were conducted with one main process and four child processes running different tabu conditions in parallel. For the 25, 33 and 42 city problems, the speedup due to parallelism was not very noticeable because of the small run times and the relative ease with which the serial algorithm produces optimum results. For the 57 city problem, the optimal result was obtained more consistently and in a shorter time period. Eight out of twelve runs produced the optimum taking an average of about 56 seconds. The parallel search process produced a better tour length of 425 miles for the 50 city problem. The speedup was not very noticeable in this case also. On an average the quality of the solutions were better than the serial algorithm. The parallel algorithm also obtained a superior solution in the 75 city problem with a tour length of 535

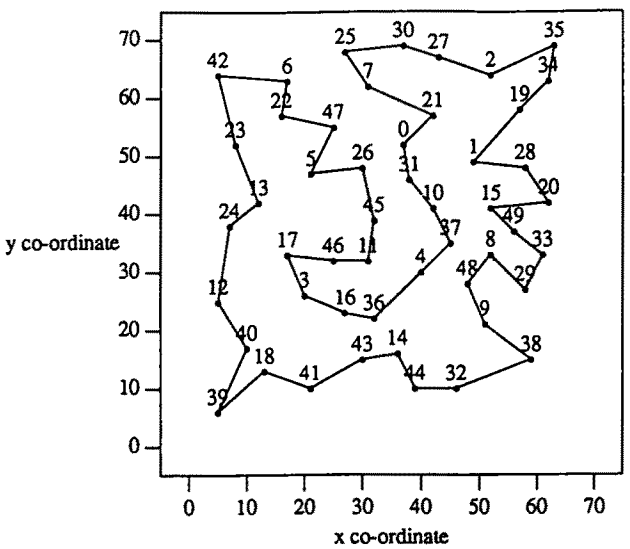


Fig. 13. Plot of the best 50 city tour, tour cost = 425 miles.

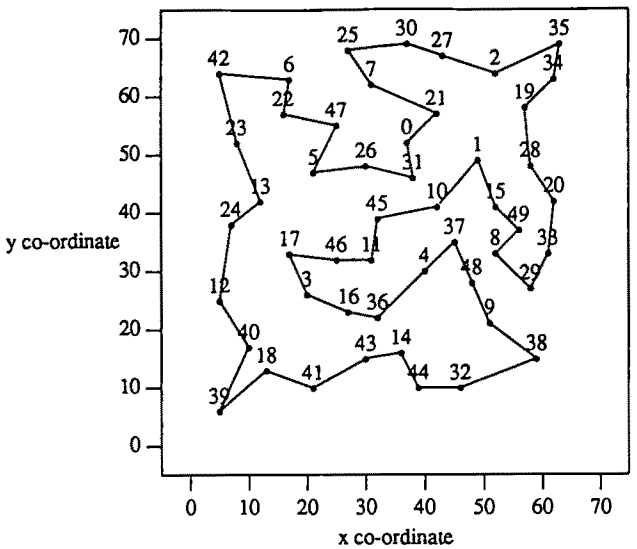


Fig. 14. Plot of the best 50 city tour, tour cost = 426 miles.

miles in 132 seconds. Every run with a different random starting tour beat the best result published with an average run time of 140 seconds. The best tour for the 50 city problem obtained during our experimentation is plotted in fig. 13. Figure 14 shows the tour that is only one mile longer. Observe that there is a substantial difference between these tours despite only one mile difference in the tour cost. Furthermore, in fig. 15 the best tour for a 75 city problem is depicted.

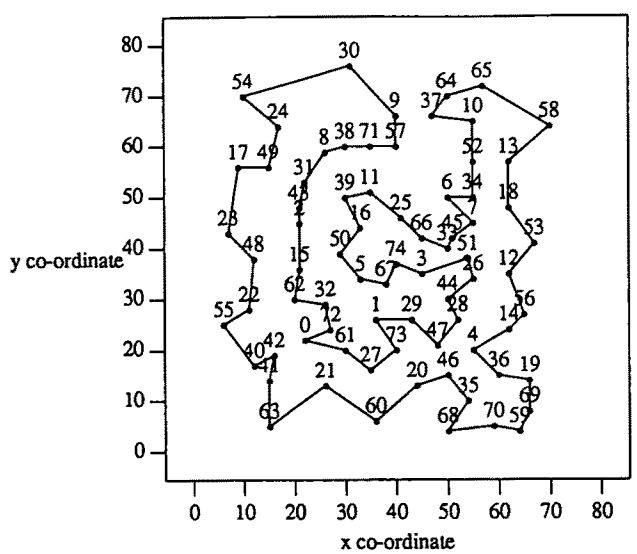


Fig. 15. Plot of the best 75 city tour, tour cost = 535 miles.

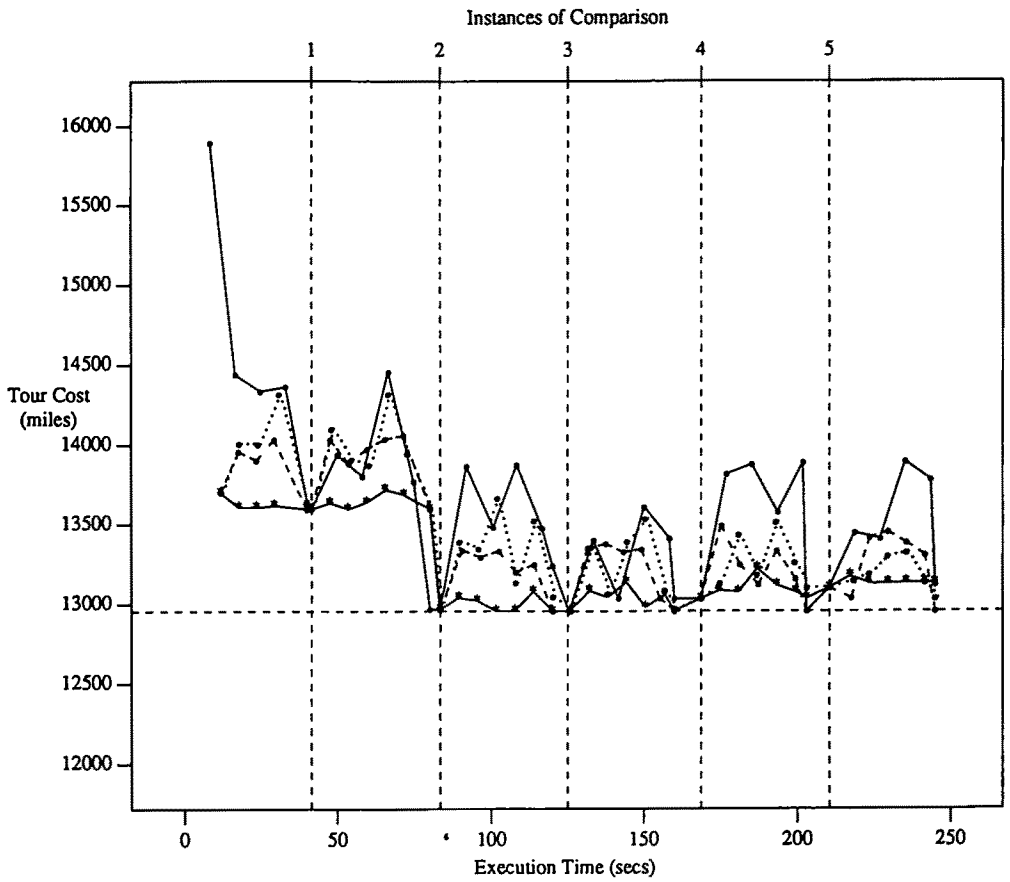


Fig. 16. Tour cost versus execution time for 57 city problem using parallel tabu search.

Figure 16 shows graphically how the parallel tabu algorithm progressed on the 57 city problem for an example starting tour. The vertical dashed lines represent the instances at which the main process compares the intermediate results of the child processes and supplies them with the common starting tour.

6. Comparative analysis and conclusions

We have implemented simulated annealing and tabu search algorithms for the traveling salesman problem to compare their performance with respect to time and tour cost. Both serial and parallel implementations produced many interesting results that can be compared in four ways as follows.

1. First, we observed significant improvement, in both execution time and tour cost, of parallel implementation of simulated annealing over serial one. In fact, surprisingly, the speedup (ratio of serial execution time to parallel

execution time to obtain best result) ranged from two to sixteen times on a two to seven processor implementation for a given set of randomly generated initial tours. In fact for a 100 city problem we have achieved an incredible speedup of 11 times on a two processor system! Intuitively, this phenomenon could be attributed to the abbreviated and different cooling schedules. Furthermore, tour cost was consistently better for a parallel implementation. Figure 17 shows the comparison of execution times for the serial and parallel implementations of simulated annealing for six randomly generated starting tours of the 57 city problem.

2. Parallel tabu search implementation not only outperformed the serial one with respect to both time and cost, but also consistently produced results comparable and in some instances better than the parallel implementation of simulated annealing. For the 50 and 75 city problems, the parallel tabu search produced better than the best published to date tour lengths.
3. In our experimentation with simulated annealing and tabu search, we also observe that tabu search outperformed simulated annealing in terms of execution time and cost in serial implementation of these algorithms. Figure 18 provides a graphical comparison for the performance of the two algorithms. For the 42 city problem our data indicates that in general, tabu search finds better results in about one-ninth of the execution time. Tabu search also gave optimum solutions consistently. Similar comparisons of our data for the 57 city problem indicate that the time taken by tabu search to reach comparable results had a speed-up ranging from 5 to 26 over that of simulated annealing.

   As shown in the above examples, the tabu search pretty consistently outperforms simulated annealing with respect to execution time and cost. On the other hand, simulated annealing parameters, once determined, are pretty stable for a wide range of applications while tabu search parameters might need to be adjusted depending on problem size and application. Tabu search is still in its infancy, leaving many research issues unexplored.
4. While serial implementation of tabu search consistently outperforms serial simulated annealing in terms of time and cost, the performance of parallel incarnations of these algorithms appears to be comparable with respect to cost, but, tabu search is much faster. Further research is needed to reach definite conclusions on this matter.

Parallel computing has created a new environment in which better results can be obtained in shorter time for many algorithms. The traveling salesman problem example is shown in this study. A lot of research is needed to explore different ways of executing algorithms in parallel and many experiments are required to confirm effectiveness of parallelization methods. One of the ways is a new method of combining algorithms, which we call a hybrid algorithm technique.

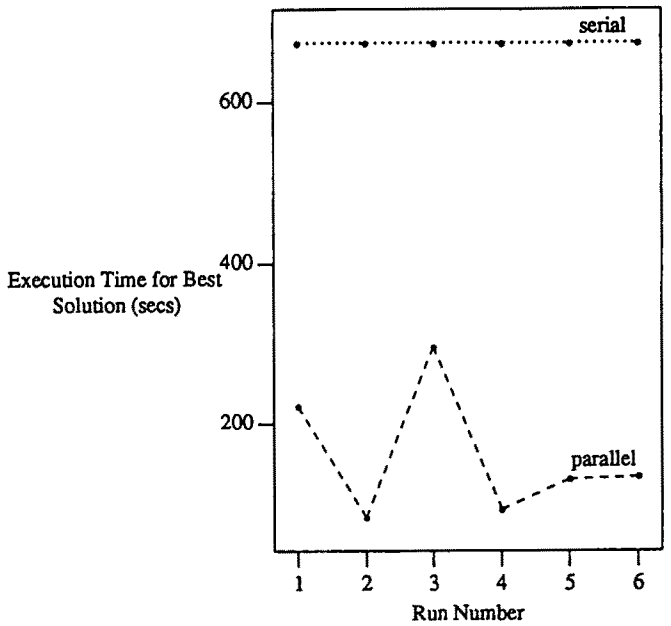


Fig. 17. Comparison of serial and parallel simulated annealing.

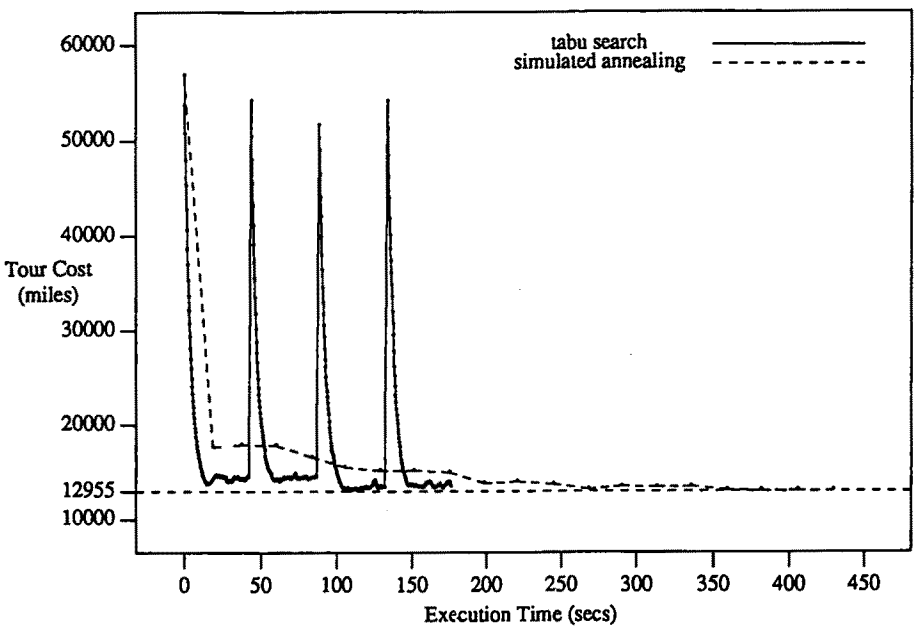


Fig. 18. Comparison of serial simulated annealing and tabu search.

Initial results of applying this technique to the traveling salesman problem are reported in [15].

Acknowledgement

We would like to express our appreciation to Dr. Fred Glover for helpful discussions and insightful comments on the earlier versions of this paper.

This research was supported in part by the Office of Naval Research under Grants Nos. N00014-86-K-0554 and N00014-88-K-0543.

References

[1] S. Kirkpatrick, C.D. Gelatt and M.P. Vechi, Optimization by simulated annealing, Science 220 (May 13, 1983) Number 4598.

[2] F. Glover, Tabu search, Center for Applied Artificial Intelligence, Graduate School of Business, University of Colorado, Boulder, 1988.

[3] E.L. Lawler, J.K. Lenstra and A.H.G. Rimnooy Kan, eds., *The Traveling Salesman Problem* (North-Holland, 1985).

[4] R.L. Karg and G.L. Thompson, A heuristic approach to solving travelling-salesman problems, Management Science 10 (1964) 225–247.

[5] G.B. Dantzig, D.R. Fulkerson and S.M. Johnson, Solution of a large-scale travelling-salesman problem, Operations Research 2 (1954) 393–410.

[6] N. Christofides and S. Eilon, An algorithm for vehicle dispatching problem, Operational Res. Q. 20 (1969) 309–318.

[7] P. Krolak, W. Felts and G. Marble, A man-machine approach towards solving the travelling salesman problem, Communications of the Association for Computing Machinery 14 (1971) 327–334.

[8] S. Lin and B.W. Kernighan, An effective heuristic algorithm for the traveling salesman problem, Oper. Res. 21 (1973) 495–516.

[9] R.G. Parker and R.L. Rardin, The traveling salesman problem: an update of research, Naval Research Logistics Quarterly 30 (1983) 69–96.

[10] M. Padberg and G. Rinaldi, Optimization of a 532-city symmetric traveling salesman problem by branch and cut, Operations Research Letters 6, no. 1 (1987) 1–7.

[11] G.E. Hinton and T.J. Sejnowski, Learning and relearning in Boltzmann machines, in: *Parallel Distributed Processing*, Vol. 1 (MIT Press, 1986).

[12] M. Held and R.M. Karp, The travelling salesman problem and minimum spanning trees, part I, Operations Research 18 (1970) 1138–1162; Part II, Mathematical Programming 1 (1971) 6–26.

[13] H. Schwetman, *PPL Reference Manual*, version 1.1 (Microelectronics and Computer Technology Corporation, 1985).

[14] B.W. Kernighan and D.M. Ritchie, *The C Programming Language* (Prentice-Hall, 1978).

[15] M. Malek, M. Guruswamy, H. Owens and M. Pandya, A hybrid algorithm technique, Technical Report, Dept. of Computer Sciences, The University of Texas at Austin, TR-89-06, 1989.