**INDIAN INSTITUTE OF MANAGEMENT CALCUTTA**

**WORKING PAPER SERIES**

**WPS No. 668/ December 2010**

# A Fast Tabu Search Implementation for Large Asymmetric Traveling Salesman Problems Defined on Sparse Graphs

**by**

**Sumanta Basu**
Assistant Professor, IIM Calcutta, Diamond Harbour Road, Joka P.O., Kolkata 700 104
India

**Ravindra S. Gajulapalli**
HCL Technologies, Bangalore

**&**

**Diptesh Ghosh**
Associate Professor, IIM Ahmedabad, Vastrapur, Ahmedabad Pin-380015, India

# A Fast Tabu Search Implementation for Large Asymmetric Traveling Salesman Problems Defined on Sparse Graphs

Sumanta Basu[*] Ravindra S. Gajulapalli[†] Diptesh Ghosh[‡]

**Abstract**

Real life traveling salesman problem (TSP) instances are often defined on large, sparse, and asymmetric graphs. Tabu search implementations for the TSP that have been reported in the literature almost always deal with small, dense, and symmetric instances. In this paper, we outline a tabu search implementation which can solve TSP instances much faster than conventional implementations if the graph defining the instance is sparse. We present results from computational experiments with this implementation which validate our claim.

**Keywords:** Tabu Search, Asymmetric Traveling Salesman Problem, Data Structures

## 1 Introduction

Given a graph $G = (V, A, C)$, where V is a set of nodes, $A$ is a set of arcs of the form $(i, j)$, $i, j \in V$, and $C = (c(i, j))$ is a vector of costs where $c : A \to \mathbb{Z}_+$, the traveling salesman problem (TSP) is one of finding a simple cycle in $G$ covering all nodes in $V$, such that the sum of the costs of the arcs in the cycle is the minimum possible. Cycles covering all nodes in $G$ are called tours and the sum of the costs of the arcs in a tour is called the cost of the tour. If the existence of an arc $(i, j) \in A$ implies that the arc $(j, i) \in A$ and that $c(i, j) = c(j, i)$, then the TSP defined on such a graph is called a symmetric traveling salesman problem (STSP), otherwise the TSP is called an asymmetric traveling salesman problem (ATSP).

The TSP is one of the most well-studied combinatorial problems and has a large number of practical applications (see e.g., Lawler et al. (1985)). It is also known to be NP-hard (Karp, 1972). Consequently, a lot of research effort has focused on optimal and heuristic algorithms to solve the TSP. In particular, in recent years, metaheuristics have been extensively used to solve this problem.

Most metaheuristics proposed in the literature are modifications of the local search algorithm. The local search algorithm is an improvement heuristic. For the TSP, it starts with a given tour, calls it the current tour, and iteratively tries to generate better tours. To do so, in each iteration, it searches a pre-defined neighborhood of tours of the current tour for the iteration, and finds the best tour in the neighborhood. If the cost of the best neighboring tour is less than that of the current tour, then the best neighboring tour is designated as the current tour for the next iteration. Otherwise, the current tour is output and the algorithm terminates. As with all local search algorithms, the local search algorithm for the TSP terminates at the first locally optimal tour it encounters. Tabu search is a metaheuristic proposed by Glover (see Glover (1989, 1990)) which extends the conventional philosophy of local search by allowing non-improving solutions to be chosen as the current solution and by discouraging the search from returning to a solution already visited in previous few iterations.

One concern that motivates this paper is that Basu and Ghosh (2008), providing a review of the literature on tabu search applied to the TSP in the last twenty years, report only nine papers

[*]Corresponding Author. OM Group, Indian Institute of Management, Calcutta, Joka, Diamond Harbour Road, Kolkata 700104, India. Phone +91 33 2467 8300–06 Extension 543. Email: sumanta@iimcal.ac.in.
[†]HCL Technologies, Bangalore, India.
[‡]P&QM Area, Indian Institute of Management, Ahmedabad, Vastrapur, Ahmedabad 380015, India.

among more than fifty that address problems with more than 400 nodes. Among these, Cordeau et al. (2001) report their experiemce with implementing tabu search on real life TSPs with between 819 and 1025 nodes, where their implementation required more than 16 hours to perform 50000 tabu search iterations. Homberger and Gehring (2005); Cordeau et al. (1998); Montane and Galvao (2006); Tarantilis (2005) etc. report similar experiences. The lack of literature on tabu search applied to large sized TSPs is surprising, since tabu search is a heuristic, and is designed to solve problems with sizes which make them inaccessible to exact algorithms.

Another concern is about the nature of the graphs used to test TSP implementations in the literature. An overwhelming majority of the papers reviewed in Basu and Ghosh (2008) implement tabu search on complete graphs. The only exceptions are Toth and Vigo (1998) and Glover and Laguna (1998). In Toth and Vigo (1998), a restricted neighborhood is created for tabu search by choosing only those arcs with costs less than a threshold value. In Glover and Laguna (1998), there is a suggestion to use a candidate list strategy, although it is not backed by computational experiments. TSP instances arising out of real-life situations, especially in logistics are defined on sparse graphs. The highway network of India for example, passes through 174 major cities, but has only 212 direct intercity connections, thus forming a graph with a density of 1.4%. If we consider a square grid with $n$ nodes on each side, and join each node on the grid with the next node to the left, right, above and below it, the density of the graph formed is $\Theta(1/n)$. These types of graphs are good representations of road networks in cities. So there is a need to design tabu search implementations that exploit the sparsity of graphs defining TSPs.

A third concern in the tabu search literature on TSPs is that almost all papers study symmetric TSPs. This fact is either made explicit or is implicit from the discussion on neighborhood generation. Some authors (see Brando and Mercer, 1997; Cordeau et al., 2001) select non-integer distances in their problem context but still assume symmetric distances. Basu and Ghosh (2008) found only one paper (Tarantilis, 2005) which addressed asymmetric problem sets developed in Golden et al. (1998) to implement tabu search for TSPs. Most TSPs arising out of real life situations, especially in logistics are defined on asymmetric graphs.

We address the concerns raised above by presenting a tabu search implementation designed for ATSPs defined on large sparse graphs. Developing such an implementation introduces interesting challenges. Conventionally, sparse graphs are represented as complete graphs in which the costs of non-existent arcs are taken as infinite (see, e.g., representations of the `code198` and `code253` instances in Johnson et al. (2002)). While this representation is correct, it does not allow tabu search implementations to take advantage of the fact that on a sparse graph, in each iteration, the algorithm searches a much smaller neighborhood than in a complete graph with the same number of nodes. Secondly, since the sizes of neighborhoods typically grow exponentially with the size of the TSPs, for large TSPs it is impractical to implement neighborhoods larger than $O(n^2)$. For these neighborhoods, the complexity of searching a neighborhood in an ATSP is one order of complexity higher than searching an equivalent neighborhood in a STSP. This is because in such neighborhoods in ATSPs, the directions of a large number of arcs in two neighboring tours are reversed. There are specialized ATSP neighborhoods reported in the literature (e.g., the irreversible 3-opt neighborhood (Kanellakis and Papadimitriou, 1980) and the double bridge neighborhood (Glover, 1996) that create neighbors in which the directions of the arcs are maintained. However, there are no $O(n^2)$ neighborhoods with these properties.

In this paper, we present a tabu search implementation that is superior to conventional implementations for large sparse ATSPs. In Section 2, we describe the basic tabu search algorithm. In Section 3, we present our implementation to speed up tabu search on large sparse ATSPs. We report a computational comparison between our implementation and conventional implementations on randomly generated ATSP instances in Section 4. We conclude this paper in Section 5 by summarizing our results and highlighting our key contributions.

## 2 Conventional Tabu Search

Tabu search starts by designating a tour (say $\tau_0$) input to it as the current tour whose neighborhood is to be searched. The best tour found by tabu search is initialized to $\tau_0$, and a tabu list $L$ is initialized to an empty list. At the beginning of each iteration, tabu search checks if pre-specified termination conditions have been met. If such conditions are met, then it terminates after printing the best tour it has found during its execution. If the conditions are not met then it executes three steps. In the first step, it searches the neighborhood of the current tour $\tau$ to find the least cost neighboring tour $\tau'$ which can be reached through a move by adding arcs not in the tabu list. In the second step, $\tau'$ is designated as the tour whose neighborhood is to be searched in the next iteration. In the third step, the tabu list $L$ is updated. The update involves removing arcs which have been in $L$ for a pre-specified number of iterations, and adding the arcs that were removed from $\tau$ in the current iteration in the process of generating $\tau'$. In addition, a check is performed to see if the cost of $\tau'$ is less than that of $\tau^*$. If that is the case, then $\tau'$ is copied into $\tau^*$. Several conditions are used either individually or collectively as termination conditions of tabu search. Some widely used ones are fixed execution time, fixed number of iterations, number of iterations without any improvement in the value of $\tau^*$ etc.

In this paper, we used a 2-opt neighborhood since it is a commonly used neighborhood structure for tabu search on TSPs (see Basu and Ghosh, 2008). In a 2-opt neighborhood, a tour $\tau'$ is said to be a neighbor of a tour $\tau$ if $\tau'$ can be obtained by deleting two arcs from $\tau$ and recombining the two paths thus formed into a tour different from $\tau$. Obviously, in a TSP defined on a graph with $n$ nodes, there are $O(n^2)$ 2-opt neighbors of any given tour. For a STSP, searching the 2-opt neighborhood of a tour takes $O(n^2)$ time, since the cost of a neighboring tour can be obtained from the cost of a tour in constant time. In an ATSP, a 2-opt move requires deleting two arcs from a tour, reversing the directions of each of the arcs in one of the paths formed as a result of the deletions, and then adding two arcs to create a neighboring tour. Hence finding the cost of a 2-opt neighbor in an ATSP requires $O(n)$ time. Since the size of a 2-opt neighborhood is $O(n^2)$, the complexity of searching the 2-opt neighborhood of a tour is $O(n^3)$ for an ATSP. Our implementation scheme tries to reduce this complexity by making use of the sparsity of graphs defining ATSPs.

In this regard, a special irreversible 3-opt neighborhood is reported in the literature for ATSPs (see Kanellakis and Papadimitriou, 1980). This neighborhood is a restriction on the conventional 3-opt neighborhood. In this neighborhood, a neighboring tour is generated as follows. Three arcs, say $(i, j)$, $(k, l)$, and $(p, q)$, not all adjacent, are deleted from current tour, and arcs $(i, l)$, $(p, j)$, and $(k, q)$ are added back to form a neighboring tour. This operation does not require the directions of any arc to be reversed in the current tour to create a neighboring tour, and hence the cost of the neighboring tour can be computed in constant time given the cost of the current tour. We use tabu search using this neighborhood in our computational experiments in Section 4 to compare with our implementation. Irreversible 2-opt neighborhoods are impossible to construct.

## 3 Our Implementation for Sparse ATSPs

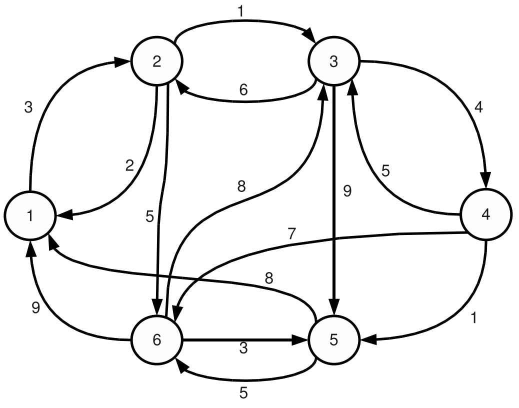
Two main data structures used in any tabu search implementation for TSPs are structures to store the graph describing the problem and structures to store information about tours. In this section we first describe the data structures in our implementation to store graphs in Subsection 3.1. We then describe the data structures to store tours in Subsection 3.2. We finally describe how we use these data structures to increase the speed of tabu search iterations in Subsection 3.3. In this section, we will assume that the graph defining the ATSP has $n$ nodes, and each node is directly connected to $k$ other nodes in the graph on average.

## 3.1 Data structures to store graphs

Complete graphs are predominantly used in tabu search literature and therefore the most efficient data structure for storing the costs of edges or arcs is the adjacency matrix. Given a TSP with n nodes, this is a $n \times n$ matrix A $= [a_{ij}]$, in which $a_{ij}$ stores the cost of the arc from node $i$ to node $j$. In case the TSP is symmetric, it is sufficient to store $a_{ij}$ values only when $i < j$. If an arc $(p, q)$ does not exist in the graph then $a_{pq}$ is set to $\infty$.

If the graph is sparse, storing costs of infeasible arcs is inefficient. Papers like Saad (1994) and Eijkhout (1992) elaborate on different data structures used to store sparse matrices, of which the compressed row format is especially effective. In our tabu search implementation, we use a minor modification of this format to store the arc costs for a given problem.

Given an asymmetric graph $G = (V, A, C)$, we define an array `arcs` and a vector `arc_position_ptr`. Without loss of generality, let us assume that the nodes in the graph are numbered 1 through $n$ and $|A| = m$. `arcs` is a $m \times 3$ matrix, in which each row represents one arc in $A$. The first column stores the tail of an arc, the second column stores its head, and the third column its cost. The arcs are ordered in non-decreasing order of their tail nodes, and then according to the increasing order of their head nodes. `arc_position_ptr` is a vector of size n. The $i$-th element of `arc_position_ptr` stores the smallest row number in `arcs` which represents an arc with $i$ as its tail. Figure 1 presents an example of the data structures that we use to store graphs in our implementation.



| tail | head | cost |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 1 | 2 |
| 2 | 3 | 1 |
| 2 | 6 | 5 |
| 3 | 2 | 6 |
| 3 | 4 | 4 |
| 3 | 5 | 9 |
| 4 | 3 | 5 |
| 4 | 5 | 1 |
| 4 | 6 | 7 |
| 5 | 1 | 8 |
| 5 | 6 | 5 |
| 6 | 1 | 9 |
| 6 | 3 | 8 |
| 6 | 5 | 3 |

`arcs`

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 5 | 8 | 11 | 13 |

`arc_position_ptr`

Figure 1: A graph for an ATSP instance and its representation in our implementation

With our data structures, the time required to search whether a particular node is directly connected to another node is $O(\log(k))$ on average, and the time required to list all neighbors of a node is $O(k)$ on average. In an adjacency matrix representation, the first operation is constant time, while the second requires $O(n)$ time.

## 3.2 Data structures to store tours

In conventional implementation, tours are either stored as a permutation of nodes in $V$ or as a linked list of arcs. Storing tours in the latter format has obvious advantages when one is dealing with symmetric TSPs on complete graphs; in such cases, converting a tour to neighboring tours is a constant time operation of modifying four pointers. However, this is not true when one is dealing with ATSPs, since a 2-opt move in an ATSP requires the sequence of intermediate nodes to be reversed. In addition, when dealing with ATSPs defined on sparse graphs, the existence of a chain of nodes in one direction is no guarantee that the chain will remain feasible when the arc directions are reversed. Hence, in our implementation, we store the tour as a $n \times 6$ array `tour_arcs` where each row corresponds to data about one arc in the tour. Consider a row in the array which corresponds to arc $(i, j)$. Columns 1 and 2 store $i$ and $j$, and column 3 stores $c(i, j)$. Column 4 stores a value of 1 if arc $(j, i)$ exists in the graph, and 0 otherwise. Column 5 stores the value of $c(j, i)$ if it exists, and column 6 stores a value of 1 or 0 depending on whether the arc $(j, i)$, if it exists, is in the tabu list or otherwise. Notice that the information being stored in the fourth and fifth columns can be obtained by looking up the graph data structure, but by storing them in the tour data structure, we can obtain these in constant time rather than spend $O(\log(k))$ time to look it up from the graph data structure. We also store a $n \times 2$ array called `tour_node_ptr`. The first column of the $i$-th row of this array stores that row in the `tour_arcs` matrix which has node $i$ as its tail. The second column stores that row in the `tour_arcs` matrix which has node $i$ as its head.

**Example**: Consider the tour $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 1$ in the graph in Figure 1. Assume that

| tail | head | arc cost | rever-sible? | rev. arc cost | rev. arc tabu? |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 0 |
| 2 | 3 | 1 | 1 | 6 | 1 |
| 3 | 4 | 4 | 1 | 5 | 1 |
| 4 | 5 | 1 | 0 | * | * |
| 5 | 6 | 5 | 1 | 3 | 0 |
| 6 | 1 | 9 | 0 | * | * |

`tour_arcs`

| tail | head |
|---|---|
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |
| 5 | 4 |
| 6 | 5 |

`tour_node_ptr`

Figure 2: Representation of a tour in our implementation

the tabu list is $\{(2, 6),\ (3, 2),\ (3, 5),\ (4, 3)\}$. Figure 2 represents the tour structure used to store the tour in our implementation. In the representation a '*' at any position implies that the value at that position is not important for storing tour information. For example, for arc (4,5) represented in the fourth row of `tour-arcs`, columns 5 and 6 have a '*' since the arc (5,4) does not exist in the graph, as noted in column 4 of the array. □

## 3.3 Our neighborhood search method

We make tabu search efficient for problems defined on sparse graphs by devising a method to quickly find legitimate candidate arcs for 2-opt moves given a tour and a tabu list, by utilizing the sparseness of the graph defining the ATSP. Given a graph $G = (V, A, C)$, a tour $\tau$ and a tabu list $L$, Algorithm 1 presents our method of finding the best neighboring tour which does not include any arc present in $L$. This algorithm therefore defines the main part of a tabu search iteration for our implementation. For notational convenience, a tour $\tau$ is a vector of arcs $(a_1, a_2, \ldots, a_n)$ where the head of arc $a_i$ is the tail of arc $a_{i+1}$ for $i = 1, \ldots, n-1$, and the head of arc $a_n$ is the tail of arc $a_1$, and $[\tau \circledast (a_i, a_j)]$ is the tour obtained from tour $\tau$ through a 2-opt operation involving the deletion of arcs $a_i$ and $a_j$.

Our implementation is fast because we are able to, without explicit computation, eliminate infeasible 2-opt neighbors of a given tour. We do this by manipulating a flag value (see steps 6 through 15 in Algorithm 1). To understand how the flag value is manipulated, note that two arcs $a_i$ and $a_j$ will yield a feasible 2-opt neighbor only if the following three conditions are met.

1. For each arc $(p, q)$ between $a_i$ and $a_j$ in the tour, the arc $(q, p)$ must exist in the graph, and not be in the tabu list;
2. there must exist an arc from the tail of $a_i$ to the tail of $a_j$ in the graph and it should not be in the tabu list; and
3. there must exist an arc from the head of $a_i$ to the head of $a_j$ in the graph and it should not be in the tabu list.

**Algorithm 1** Finding the best neighbor of a tour in our implementation

```
Require: $G = (V, A, C)$, A tour $\tau$, tabu list $L$
Ensure: The best neighbor $\tau_{best}$ of $\tau$
 1: {INITIALIZATION}
 2: set $b \leftarrow \infty$
 3: set $\tau' \leftarrow \phi$

 4: {ITERATION}
 5: for all $a_i = (p, q) \in \tau$ do
 6:     set $flag(a_j) \leftarrow 0$ for each $a_j \in \tau$
 7:     for all $a_j = (r, s) \in \tau$ such that $(s, r) \in A \setminus L$ do
 8:         set $flag(a_{j+1}) \leftarrow 1$
 9:     end for
10:     for all $a_j = (r, s) \in \tau$ such that $flag(a_{j-1}) = 1$ and $(p, r) \in A \setminus L$ do
11:         set $flag(a_j) \leftarrow 2$
12:     end for
13:     for all $a_j = (r, s) \in \tau$ such that $flag(a_j) = 2$ and $(q, s) \in A \setminus L$ do
14:         set $flag(a_j) \leftarrow 3$
15:     end for
16:     for all $a_j (\neq a_i) \in \tau$ such that $a_i$ and $a_j$ are not adjacent
        and $flag(a_j) = 3$ do
17:         if $c([\tau \circledast (a_i, a_j)]) < b$ then
18:             set $a_1^* \leftarrow a_i$, $a_2^* \leftarrow a_j$, and $b \leftarrow c([\tau \circledast (a_1, a_2)])$
19:         end if
20:     end for
21: end for
22: set $\tau_{best} \leftarrow [\tau \circledast (a_1^*, a_2^*)]$
23: return $\tau_{best}$
```

Now suppose, without loss of generality, that we want to find those arcs with which arc $a_1$ can pair to yield feasible 2-opt neighbors. We start by initializing the flag value for each arc to 0, and incrementing the flag value for each arc by 1 for each one of the three conditions that it satisfies. Step 6 in Algorithm 1 performs this operation. Steps 7 through 9 implement condition 1, steps 10 through 12 implement condition 2, and steps 13 through 15 implement condition 3. At the end of the incrementing, only those arcs which have a flag value of 3 satisfy all the conditions to be eligible to participate in a 2-opt operation with $a_1$. Of course $a_2$ and $a_n$, being adjacent to $a_1$ cannot participate in a 2-opt operation with it and can be ignored. Also, if we set flag values starting from $a_3$ and going up to to $a_{n-1}$, and if we find an arc $a_i$ which does not satisfy condition 1, then we can conclude that none of the arcs from $a_{i+1}$ through $a_{n-1}$ will satisfy condition 1. So we do not need to check those arcs for conditions 2 and 3. This observation speeds up the update of flag values significantly.

We now compute the complexity of Algorithm 1. Consider the flag value computation to identify all arcs which can participate in a 2-opt operation with a particular arc $a_i$ in the tour. Given our tour data structure, checking condition 1 for all arcs in a tour requires $O(n)$ time. Finding the list of nodes that are directly connected to a node in the graph requires $O(k)$ time on average. Checking if an arc connecting two nodes is in the tabu list requires $O(\log(|L|))$ time. So performing steps 2 and 3 require $O(k \log(|L|))$ time on average. Therefore the flag value computation requires $\max\{O(n), O(k \log(|L|))\}$ time on average. If $k$ and $|L|$ are small compared to $n$, as is the case in large sparse ATSPs, the computation of flag values require $O(n)$ time.

At the end of the flag value computation, let us assume that we have a list of $K$ ($\ll n-3$) arcs which yield feasible tours through a 2-opt operation with $a_i$. Finding the best among these $K$ tours takes $O(nK)$ time, which can be reduced through intelligent book-keeping. Since there are $n$ arcs in a tour Algorithm 1 requires $O(n^2 K_{avg})$ time, where $K_{avg}$ denotes the average number of arcs that can participate in a 2-opt operation with any given arc in a tour. Note that in a conventional implementation, the equivalent time complexity is $O(n^3)$. The effect of this decrease in time required to perform tabu search iterations is evident from the results presented in Section 4.

## 4 Computational Experience

In this section we describe our computational experience with the tabu search implementation that we have described in Section 3. We compared our implementation with conventional tabu search implementations incorporating the same tabu search features. We chose to solve ATSP instances with between 1000 and 3000 nodes and densities varying between 1% and 5%. Since we did not find such benchmark ATSP instances in the literature, we chose to generate our own problems for the comparison. In Section 4.1 we describe the procedure we used to generate the problems. In Section 4.2 we discuss our findings.

### 4.1 Generating problem instances

The instances that we generated are "node clustered instances" in which nodes form well connected groups which are interconnected. The rationale behind considering these instances is that in many real life problems, especially arising in logistics networks, nodes exist in well connected clusters, while connections between clusters are not so abundant. The method of constructing these instances is as follows. Let us suppose that we want to create a graph on $n$ nodes with density $\rho$. To do this, we try to divide the nodes into $l$ completely connected clusters named $C_1$ through $C_l$ of approximately equal size, and then ensure that there is a connection between clusters $C_i$ and $C_{i+1}$ for $i = 1, \ldots l-1$ and a connection between clusters $C_1$ and $C_l$. Ignoring the requirement that $l$ must have an integer value, $n$, $l$ and $\rho$ are related as per the equation

$$l\frac{n}{l}(\frac{n}{l}-1) + 2l = \rho n(n-1). \tag{1}$$

The first term on left-hand side is the number of intra-cluster arcs in the graph and the second term is the inter-cluster arcs. We solve equation (1) to obtain $l$ and then partition the nodes into $\lceil l \rceil$ clusters in which ($n \mod \lceil l \rceil$) clusters have $\lceil n/l \rceil$ nodes and the remaining have $\lfloor n/l \rfloor$ nodes. We populate the arc set of $D$ by first creating arcs between each ordered pair of nodes in every cluster. Then for each pair of clusters that we need to connect, we choose two nodes from each of the clusters. Suppose we choose nodes $p$ and $q$ from the first cluster and nodes $r$ and $s$ from the second cluster. We add arcs $(p, r)$ and $(s, q)$ to $D$. If the density of $D$ after this operation is less than $\rho$, we add an adequate number of inter-cluster arcs at random to $D$ to raise its density to $\rho$. Clearly, from the way the clusters are connected, and the fact that each of the clusters are completely connected, $D$ contains tours. Also, since each of the clusters is completely connected, any tour in $D$ has neighboring tours. Hence these problems are also adequate for performing experiments using tabu search.

In our experiments, the cost of each arc in each graph is an integer chosen at random from a uniform distribution on [10000, 50000].

### 4.2 Results from computational experiments

In our computational experiments we compared the implementation that we propose here with two conventional tabu search implementations. We call our tabu search implementation for sparse asymmetric graphs TS-SAG, and an implementation of conventional tabu search using the 2-opt neighborhood TS-CI. We coded a third implementation of conventional tabu search using the irreversible 3-opt neighborhood structure and called it TS-3OPT. All the three implementations only used tabu lists, which had a capacity to store 50 arcs, and no intermediate or long term memory structures. Our tabu list is longer than the usual size of such lists reported in the literature. This is because, from preliminary experiments we have observed that shorter tabu lists lead to cycling in tabu search implementation for the problem sizes that we consider in our experiments. The algorithms were coded in C and the experiments were performed on a computer with 3 GB RAM and a 2.4GHz Quad Core processor.

The experiments were divided into two parts. In the first part, we allowed the implementations to run for a pre-specified number of tabu search iterations, and in the second part all the implementations were allowed to run for a pre-specified amount of execution time. We describe the experiments and results in detail in the remainder of this section.

**Experiments in which the number of tabu search iterations were fixed**
For these experiments we generated instances on graphs with 1000, 2000, and 3000 nodes with densities of 0.01, 0.02, and 0.05. We proposed to allow each implementation to run for 1000 tabu search iterations. However from preliminary experiments, we observed that TS-3OPT took extremely long times for these problems, and so we decided not to include TS-3OPT in our detailed experiments for this part. Since TS-SAG and TS-CI only differ in the way they search the same neighborhood, the costs of the best tours that they find in each iteration are identical; so in our comparison of the two implementations, we compare them only on the time they required to complete 1000 tabu search iterations.

For each problem size and density combination, we generated five instances. We ran tabu search on each of them starting from four initial tours. The execution time for each instance is taken as the average of the execution times obtained from all four runs. The execution time for a problem size-density combination is the average of the execution times for all the five instances with that problem size-density combination.

Table 1 presents the execution times required by TS-SAG and TS-CI on different problem size-density combinations for node clustered problems.

Table 1: Execution time required by TS-SAG and TS-CI for 1000 iterations (in seconds).

| Density | 1000 | | 2000 | | 3000 | |
|---|---|---|---|---|---|---|
| $\rho$ | TS-SAG | TS-CI | TS-SAG | TS-CI | TS-SAG | TS-CI |
| 0.01 | 11.75 | 29.55 | 47.60 | 140.30 | 110.60 | 355.55 |
| 0.02 | 17.00 | 44.35 | 64.50 | 259.00 | 165.20 | 736.05 |
| 0.05 | 34.10 | 142.75 | 151.45 | 1020.05 | 352.75 | 3303.05 |

Several observations are immediate from Table 1. First, for ATSPs defined on graphs with low densities, TS-SAG is significantly faster than TS-CI. This time advantage reduces as the densities increase, possibly due to the larger overheads involved in the data structures used in TS-SAG. Secondly, the critical density at which the time required by TS-SAG becomes lower than the time required by TS-CI reduces with increasing problem size. So TS-SAG is the implementation of choice only for large ATSPs defined on sparse graphs.

We ran TS-3OPT on node clustered ATSP instances with 1000 nodes and different density levels. The execution times required by TS-3OPT for the different densities were approximately 1610 seconds for $\rho = 0.01$, 1620 seconds for $\rho = 0.02$, and 2461 seconds for $\rho = 0.05$. The cost of tours obtained were of course much less. Compared to corresponding execution time values in Table 1 it is clear that at least on the time dimension, TS-3OPT cannot compete with either TS-CI or TS-SAG. A fairer comparison of the three implementations is achieved when they are allowed to run for identical execution times. This was done in the second part of our experiments.

**Experiments in which the execution times were fixed**
For these experiments we chose the same instances that we used for the first part of the experiments. In this part, we allowed each run of each of the three implementations to execute for one hour, i.e., 3600 seconds. Obviously, this means that the number of tabu search iterations allowed by each of the implementations are different. The measure of performance for this part of experiments was the relative quality of the tours that the implementations output after one hour of execution.

Since the cost of tours output by the same implementation for different runs are different for the same instance, our method of evaluating the performance of the three implementations is the following. Let us suppose that for a given ATSP instance and a given starting tour, `TS-CI`, `TS-SAG`, and `TS-3OPT` output tours with cost $c_c$, $c_s$, and $c_3$ respectively. We define the relative quality of the tour output by `TS-CI` as $q_c = 1$, that of the tours output by `TS-SAG` and `TS-3OPT` as $q_s = c_s/c_c$ and $q_3 = c_3/c_c$ respectively. For the given instance, the relative quality of the tours output by `TS-CI` is taken as 1, that of tours output by `TS-SAG` is the average of the $q_s$ values over all four runs, and that of tours output by `TS-3OPT` is the average of the $q_3$ values over all four runs. The relative quality of tours output by `TS-CI` for a particular problem size-density combination is taken as 1, and the relative quality of tours output by `TS-CI` and `TS-3OPT` are the averages of the relative quality values over all ten instances with the same problem size-density combination. A lower relative quality value thus implies a better quality of tours.

Table 2 presents the relative quality of tours output by `TS-SAG` on different problem size-density combinations of node clustered problems. Note that the relative quality values for tours generated by `TS-CI` is 1 by construction. From Table 2 we observe that under time constraints, both `TS-CI` and `TS-SAG` perform better than `TS-3OPT`. On average, `TS-3OPT` generates better tours than the other two variants at every iteration, but since the time it requires to execute an iteration is much longer than the times required by `TS-CI` and `TS-SAG`, the number of iterations that it can execute within a specified execution time limit is much smaller than the other two variants, with a result that, given an execution time limit, the quality of the tour it outputs is worse than those output by the other variants. `TS-CI` and `TS-SAG` output identical tours after each iteration, but since `TS-SAG` is faster than `TS-CI` it outputs better quality tours than `TS-CI` after a specfied execution time limit. The dominance of `TS-SAG` and `TS-CI` over `TS-3OPT` becomes more prominent as problem sizes increase.

Table 2: Relative quality of tours generated by `TS-SAG` and `TS-3OPT`.

| Density | 1000 | | 2000 | | 3000 | |
|---|---|---|---|---|---|---|
| $\rho$ | `TS-SAG` | `TS-3OPT` | `TS-SAG` | `TS-3OPT` | `TS-SAG` | `TS-3OPT` |
| 0.01 | 1.000 | 1.076 | 0.998 | 1.745 | 1.000 | 1.886 |
| 0.02 | 0.999 | 0.848 | 1.000 | 1.821 | 0.999 | 1.971 |
| 0.05 | 1.000 | 0.660 | 0.999 | 1.894 | 0.999 | 2.037 |

## 5 Conclusion

In this paper, we present an implementation of tabu search on sparse ATSPs. To the best of our knowledge, this is the only tabu search implementation available in the public domain that exploits the sparsity of graphs defining ATSPs. In Section 3 we show that each tabu search iteration in our implementation requires $O(|A||V|)$ time, while iterations in conventional implementations require $O(|V|^3)$ time. This makes our implementation attractive for solving asymmetric TSPs with sparse graphs, especially when problem sizes are large. In Section 4 we experimentally verify the assertion above on ATSPs with up to 3000 nodes, and densities up to 5%. Our experiments show that our implementation, called `TS-SAG` in this paper, is superior for ATSPs defined on large asymmetric graphs.

## References

Basu S and Ghosh D. (2008). *A review of the tabu search literature on traveling salesman problems. Working Paper Series, Indian Institute of Management Ahmedabad*, W.P. No. 2008-10-01.

Brando J and Mercer A (1997). A tabu search algorithm for the multi-trip vehicle routing and scheduling problem. *European Journal of Operational Research* **100**:180–191.

Cordeau J, Gendreau M and Laporte G (1998). A tabu search heuristic for periodic and multi-depot vehicle routing problems. *Networks* **30**:105–119.

Cordeau J, Laporte G and Mercier A (2001). A unified tabu search heuristic for vehicle routing problems with time windows. *The Journal of the Operational Research Society* **52**:928–936.

Eijkhout V (1992). *Distributed sparse data structures for linear algebra operations.* Technical report, Computer Science Department, University of Tennessee, Knoxville, TN.

Glover F (1989). Tabu search– part I. *ORSA Journal on Computing* **1**:190–206.

Glover F (1990). Tabu search– part II. *ORSA Journal on Computing* **2**:4–32.

Glover F (1996). Finding a best traveling salesman 4-opt move in the same time as a best 2-opt move. *Journal of Heuristics* **2**:169–179.

Glover F and Laguna M (1998). *Tabu Search.* Kluwer Academic Publisher.

Golden B, Wasil E, Kelly J and Chao I (1998). Fleet management and logistics. In: *The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets, and computational results.* pp 33–56. Kluwer Academic Publishers.

Homberger J and Gehring H (2005). A two-phase hybrid metaheuristic for the vehicle routing problem with time windows. *European Journal of Operational Research* **162**:220–238.

Johnson D, Gutin G, McGeoch L, Yeo A, Zhang W and Zverovich A (2002). Experimental analysis of heuristics for the ATSP. In: Gutin G and Punnen A (eds.) *The Traveling Salesman Problem and Its Variations.* Kluwer Academic Publishers.

Kanellakis P and Papadimitriou C (1980). Local search for the asymmetric traveling salesman problem. *Operations Research* **28**: 5:1066 – 1099.

Karp R (1972). *Reducibility among Combinatorial Problems*, pp 85–103. Plenum Press.

Lawler E, Lenstra J, Rinnooy K, and Shmoys D (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley-Interscience Publication.

Montane F and Galvao R (2006). A tabu search algorithm for the vehicle routing problem with simultaneous pick-up and delivery service. *Computers & Operations Research* **33**:595–619.

Saad Y (1994). Sparskit: A Basic Tool-kit for Sparse Matrix Computation. Version 2.

Tarantilis C (2005). Solving the vehicle routing problem with adaptive memory programming methodology. *Computers & Operations Research* **32**:2309–2327.

Toth P and Vigo D (1998). *The granular tabu search and its application to the vrp.* Technical report, University of Bologna.