Implementing Tabu Search to Exploit Sparsity in ATSP Instances

Sumanta Basu
Ravindra S. Gajulapalli
Diptesh Ghosh

W.P. No. 2008-10-02
October 2008

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD – 380015
INDIA

Implementing Tabu Search to Exploit Sparsity in ATSP Instances

Sumanta Basu[1]
Ravindra S. Gajulapalli[2]
Diptesh Ghosh[3]

Abstract

Real life traveling salesman problem (TSP) instances are often large, sparse, and asymmetric. Conventional tabu search implementations for the TSP that have been reported in the literature, almost always deals with small, dense and symmetric instances. In this paper, we outline data structures and a tabu search implementation that takes advantage of such data structures, which can exploit sparsity of a TSP instances, and hence can solve relatively large TSP instances (with up to 3000 nodes) much faster than conventional implementations. We also provide computational experiences with this implementation.

Keywords: Tabu Search, Asymmetric Traveling Salesman Problems, Data Structures

1 Introduction

Given a weighted graph $G = (V, E, C)$, where V refers to the set of nodes in G, E refers to the set of edges, and $C = (c_e)$ for each $e \in E$, the traveling salesman problem (TSP) is one of finding a cycle in G covering all nodes in V, such that the sum of the weights of the edges in the cycle is the minimum possible. Cycles covering all nodes in G are called *tours*, the weights c_e are called *edge costs*, and the sum of the weights of the edges in a tour is called the *cost of the tour*. In case the weighted graph is asymmetric, the corresponding TSP is called the 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., [8]). It was also known to be $\mathscr{NP}$-hard since 1972 ([7]). Consequently, a lot of research effort has focused on finding exact and approximate solutions to the TSP. In particular, in recent years, metaheuristics have extensively been used to solve this problem.

In this paper, we concentrate on tabu search. Tabu search (see e.g., [5]) is a metaheuristic proposed by Glover ([3], [4]) that extends local search by allowing the search to explore solutions that worsen the solution value, and discouraging the search from returning to a solution that has already been evaluated. Reference [1] provides a review of the tabu search literature on TSPs from the early 1990s to the present. Three interesting features are brought out in that paper.

1. Although tabu search and other metaheuristics are intended to solve large problems, most tabu search implementations for the TSP report computational results for TSP instances with only up to 400 nodes. Three recent papers deal with TSP instances with up to 1000 nodes, but none have been found to deal with larger problems.

2. Almost all the literature on tabu search applied to the TSP deal with complete graphs,

[1]Wipro Technologies, Bangalore, India; Email: sumanta.basu2@wipro.com
[2]IIM Ahmedabad, India; Email: gs@iimahd.ernet.in
[3]IIM Ahmedabad, India; Email: diptesh@iimahd.ernet.in

3. All but one of the papers in the literature deal with symmetric problems. This is interesting, since specialized solution codes for solving large symmetric TSPs are already available, (see e.g., CONCORDE at `http://neos.mcs.anl.gov/neos/solvers/co:concorde/TSP.html`) However, no such efficient code exists in the public domain for solving large asymmetric TSPs.

An interesting feature of TSPs, especially those arising in physical transportation networks, is that the graph describing the TSP instance is sparse. For example, the highway network in India connecting 174 cities has only 212 direct connections, which means that the density of the network is 1.4%. Larger networks are likely to be even sparser. Additionally, such networks are often asymmetric, given that road segments are sometimes one-way. Hence for solving practical TSPs, one needs algorithms designed for large sparse asymmetric TSPs.

We think that the reason that the tabu search literature on TSPs deal with relatively small TSP instances is that the sizes of neighborhoods for large problems become unmanageable. In conventional implementations, asymmetry is modeled by storing the costs of all possible arcs between nodes in the graph. Sparsity is modeled by assigning infinite distances to non-existent arcs (see, e.g. representations of the `code198` and `code253` instances in [6]). While this representation is correct, it does not allow implementations to take advantage of the sparsity of the graph while solving TSPs. In this paper, we present data structures and a tabu search implementation that uses these data structures to exploit sparsity and solve large sparse ATSPs. We implement a simple tabu search in which we make use of a tabu list with fixed tabu tenure. No intermediate or long term memory structure has been used, nor is any aspiration criteria set. This is because our aim in this work is to demonstrate how tabu search implementations can exploit sparsity in TSPs, and any advanced feature of tabu search used in conventional implementations can be added to our implementation without difficulty. We use the 2-opt neighborhood structure in our implementation. In this neighborhood structure, a neighbor of a tour is generated by removing two non-adjacent arcs in the tour, reversing the direction of traversal of one of the two chains thus formed, and then reconnecting the two chains into a tour.

In Section 2 we describe the basic tabu search algorithm. In Section 3 we describe the data structures that we use in our implementation. We also explain how these data structures can be used in our implementation to drastically reduce the neighborhood search time for sparse graphs. We report our computational experience with our new implementation on randomly generated problems in Section 4. Here we show that our implementation is capable of solving large sparse ATSPs with up to 3000 nodes in times that are significantly lower than the times required by conventional implementations. We also show that given limited execution times, our implementation performs much better than conventional tabu search implementations. Section 5 summarizes the contributions of this paper and provides some research directions in this area.

2 The Tabu Search Algorithm

The basic tabu search algorithm was proposed in [3]. Several modifications have since been suggested, but in our work we use the basic algorithm. Algorithm 1 provides a pseudocode for tabu search for the TSP. The initial tour input to Algorithm 1 is denoted by τ_0 and the best tour that it finds by τ^*. The tabu list in Algorithm 1 is denoted by L. $\mathscr{N}(\cdot)$ is a function which returns a set of tours neighboring the one input to it. $[\tau \circledast (a_1, a_2)]$ represents a tour obtained from tour τ by performing a 2-opt move in which arcs a_1 and a_2 are removed from τ. $\mathscr{C}(\tau)$ returns the cost of the tour τ.

In the initialization step, the tour whose neighborhood is to be searched and the best tour found by Algorithm 1 are both set to the tour τ_0 input to Algorithm 1. The tabu list is empty at this stage.

Algorithm 1 Tabu Search

Input: $G = (V, A, C)$, a feasible tour τ_0, duration that an arc remains tabu
Output: A tour τ^* and its cost

```
1: /* INITIALIZATION */
2: $\tau = \tau^* \leftarrow \tau_0$, $L \leftarrow \emptyset$
3: /* ITERATION */
4: if termination criteria are not satisfied then
5:     /* ITERATION-STEP 1 */
6:     find $a_1$ and $a_2$ such that $a_1,\ a_2 \notin L$ and
       $\mathscr{C}([\tau \circledast (a_1, a_2)]) = \min\{\mathscr{C}(t) : t \in \mathscr{N}(\tau)\}$
7:     /* ITERATION-STEP 2 */
8:     if such $a_1$ and $a_2$ exist then
9:         $\tau \leftarrow \tau' = [\tau \circledast (a_1, a_2)]$
10:    end if
11:    /* ITERATION-STEP 3 */
12:    update $L$ by removing arcs which have been in $L$ for more than a pre-specified number of
       iterations, and adding $a_1$ and $a_2$ if they have been found in ITERATION-STEP 1
13:    if $\mathscr{C}(\tau') < \mathscr{C}(\tau^*)$ then
14:        $\tau^* \leftarrow \tau'$
15:    end if
16: else
17:    return $\tau^*$ and $\mathscr{C}(\tau^*)$
18: end if
```

At the beginning of each iteration, Algorithm 1 checks if pre-specified termination conditions have been met. If such conditions are met, then Algorithm 1 terminates after printing the best solution it has found and the cost of the best solution. If the conditions are not met then Algorithm 1 performs a tabu search iteration. A tabu search iteration consists of three steps. In the first step, Algorithm 1 searches the neighborhood of the current tour to find a least cost neighboring tour τ' which can be reached through a move involving arcs that are not in the tabu list. In the second step, τ' is chosen as the tour whose neighborhood is to be searched in the next iteration. In the third step, the tabu list L is updated. This involves removing arcs which have been in L for the pre-specified number of iterations, and adding the two arcs that were removed from τ in the current iteration to yield τ'. In addition, a check is performed to see if τ' is less costly than τ^*. If that is the case, then τ' is copied into τ^*.

Several conditions are used either individually or collectively as termination conditions. One of the most widely used is that of a time limit. This stops Algorithm 1 after a certain pre-specified execution time is spent. A closely related termination condition stops Algorithm 1 after a certain number of tabu search iterations. Another widely used termination condition depends on the solutions that Algorithm 1 encounters. It can be stopped once $\mathscr{C}(\tau^*)$ is below a certain threshold. Alternatively, Algorithm 1 can be stopped when τ^* is not overwritten for a pre-specified number of iterations.

3 Our Tabu Search Implementation

There are two main data structures used in any tabu search implementation for TSPs. They are structures to store the graph describing the problem, and structures to store information about tours. In this section we first describe these structures as they are defined in our implementation. We then describe how we exploit them to make each neighborhood search efficient.

3.1 Storing Graphs

Recall that tabu search implementations described in the literature deal with complete graphs. Hence 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 $a_{pq} = \infty$ in this matrix then we can conclude that arc (p, q) does not exist. In this representation, finding whether or not an arc exists, as well as finding the cost of an arc takes constant time; however, finding the list of neighbors of a node takes $O(n)$ time.

If the graph is sparse, the adjacency matrix stores a large number of infinite cost entries, and it is possible to store the costs in a more efficient manner. References [2] and [9] describe various methods of storing sparse matrices, of which the compressed row format is especially effective for storing sparse matrices without any special structure. In our tabu search implementation, we use a minor modification of this format to store the arc costs for a given problem.

Given a weighted asymmetric graph $G = (V, A, C)$, where V denotes the node set, A the arc set, and C the cost vector, we define an array `arcs` and a vector `graph_node_ptr`. Without loss of generality, assume that the nodes in the graph are numbered 1 through $|V|$. Let $|V| = n$ 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 each arc, the second column stores the head, and the third column the cost. The arcs are ordered in `arcs` according to non-decreasing order of their tail nodes, and then according to the increasing order of their head nodes. `graph_node_ptr` is a vector of size n. The ith element of `graph_node_ptr` stores the smallest row number in `arcs` which represents an arc with i as its tail.

Example: Consider the graph in Figure 1. Its representation in our implementation is shown in Figure 2. ■

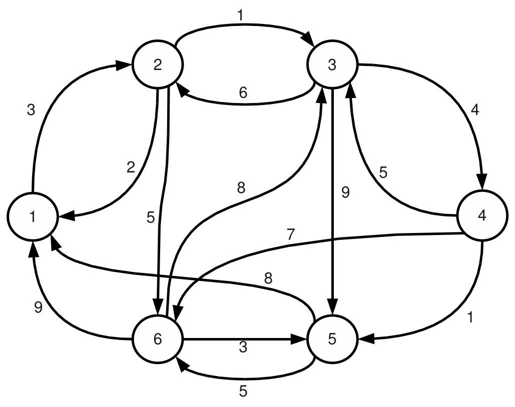

Figure 1: A graph for a TSP instance

With this representation, the time required to find whether a particular arc exists or not, and to find the cost of an arc both are $\Omega(\log(m/n))$, and the time required to find a list of all neighbors of a node is $\Omega(m/n)$.

`arcs`

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

`graph_node_ptr`

1	2	5	8	11	13

Figure 2: Representation of the graph in Figure 1 in our implementation

3.2 Storing Tours

In conventional implementations, 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 tours to neighboring tours is a constant time operation of deleting two links in the list and adding two new links. However, this is not true when one is dealing with asymmetric TSPs, since a 2-opt move requires re-orienting a chain of nodes in the tour, and the cost of traversing the chain in the forward direction is different from traversing it in the reverse direction. In addition, when one is dealing with TSPs defined on sparse graphs, the existence of a chain of nodes in one direction is no guarantee of the chain existing in the reverse direction. 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. The first three columns in this array store the tail, the head, and the cost of the arc represented by each row. The fourth column stores a 1 if the reverse of the arc exists in the graph, and 0 otherwise. (The reverse arc for the arc (i, j) is the arc (j, i).) If the fourth column for an arc has a value of 1, then the fifth column stores the cost of the reverse arc, and the sixth column stores a 1 if the reverse arc is tabu, and 0 otherwise. Notice that the information being stored in the fourth, fifth, and sixth 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 spending $\Omega(\log(m/n))$ time. We also store a $n \times 2$ array called `tour_node_ptr`. The first column of the ith 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→2→3→4→5→6→1 in the graph in Figure 1. Also assume that the tabu list comprises of the set $\{(3, 2), (4, 3), (2, 6), (3, 5)\}$. Figure 3 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. ■

3.3 Making Neighborhood Search Efficient

We can make tabu search can be made efficient for problems defined on sparse graphs if we can devise a method to quickly find candidates for 2-opt moves once we are given a tour and a tabu list.

`tour_arcs`

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_node_ptr`

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

Figure 3: Representation of a tour in the new implementation

(This is Step 6 in Algorithm 1.) In this section we present one such method. For this, we need an additional vector called `flag` of size n. In our search for the best cost neigboring tour for any given tour, we consider 2-opt moves with only those arcs which correspond to a particular flag value. We first present Algorithm 2 for finding the best neighbor of a tour, and follow it up by the procedure to set flag values.

In this discussion we use $\rho = |A|/|V|$ to denote the density of the graph describing the TSP, and $k = \rho\,|V|$ to denote the average number of arcs in the graph with their tail at one particular node.

Algorithm 2 Find Best Neighbor

Input: $G = (V, A, C)$, a tour τ, tabu list L
Output: The best neighbor τ' of τ

```
 1: /* INITIALIZATION */
 2: $b \leftarrow \infty$
 3: for all nodes $n$ in the graph do
 4:     $N(i) \leftarrow \{j : (i,j) \in A\}$
 5:     arrange nodes in $N(i)$ in the order in which they appear in $\tau$
 6: end for

 7: /* ITERATION */
 8: for all $a_1 = (v_{[i]}, v_{[i+1]})) \in \tau$ do
 9:     update the flag value for each arc in $\tau$
10:     for all $v \in N$ such that $\exists a_2 \in \tau$ with $t(a) = v_{[i]}$ and flag$(a_2) = 3$ do
11:         if $[\tau \circledast (a_1, a_2) \in \mathscr{N}(\tau)\}$ and $c([\tau \circledast (a_1, a_2)]) < b$ then
12:             $a_1^* \leftarrow a_1$, $a_2^* \leftarrow a_2$
13:             $b \leftarrow c([\tau \circledast (a_1, a_2)])$
14:         end if
15:     end for
16: end for
17: $\tau' \leftarrow [\tau \circledast (a_1^*, a_2^*)]$

18: return $\tau'$
```

Algorithm 2 is self explanatory, Notice that in Step 4, the vector $N(i)$ is populated with the neighbors of node i in the graph. Because of the data structures that we use to represent graphs, this operation takes $\Omega(k)$ time for each node instead of $O(n)$ time it would have taken in an adjacency matrix representation.

The `flag` vector is updated in Step 9 in Algorithm 2. The ith element of this vector is associated with the ith arc in the tour. Given an arc a_1 in the tour, updating the `flag` vector is the process of assigning values to the vector elements which will allow us to pick only those arcs which can be removed with a_1 to yield feasible 2-opt moves. Notice that arcs which are positioned just before or just after a_1 cannot participate in a 2-opt move with a_1.

A 2-opt move involving arcs a_1 and a_2 is possible when three conditions are met:

Condition 1: the arcs in the tour between a_1 and a_2 are reversible, and the reverse arcs for those arcs are not tabu;

Condition 2: there is a non-tabu arc from $h(a_1)$ to $h(a_2)$; and

Condition 3: there is a non-tabu arc from $t(a_1)$ to $t(a_2)$.

The update process proceeds in three stages. Initially the flag values corresponding to all the arcs in the tour except a_1 and its two neighbors in the tour are set to 0. The flag values of a_1 and its two neighbors are set to 1. In the first stage we look at each arc in τ in order and check if they are reversible and the reverse of the arc is not tabu. If the arc is reversible, the reverse arc is not tabu, and the flag of the arc is set to a value 1, then the flag value of the *next* arc in τ is incremented by 1. Given that information about the reversibility of each arc in a tour and the tabu status of the reverse arc is stored in our data structure for the tour, this stage requires $O(n)$ time. At the end of this stage, all arcs which satisfy condition 1 above have a flag value of 1. In the second stage, we find all arcs $a \in \tau$ such that the arc from the head of a_1 to the head of a exists in G and is not tabu. We increment the flag values of such arcs. Finding the existence of each of these arcs takes $\Omega(\log k)$ time, and checking its tabu status takes $O(\log |L|)$ time. Hence this step requires $\Omega(n \min(\log k,\ \log |L|))$ time. At the end of this stage all arcs which satisfy conditions 1 and 2 above have a flag value of 2. The third stage is similar to the second stage. In it we find all arcs $a \in \tau$ such that the arc from the tail of a_1 to the tail of a exists in G and is not tabu. We increment their flag values by 1. Clearly this step also requires $\Omega(n \min(\log k,\ \log |L|))$ time. At the end of the third stage, the flag values of a_1 and its two neighbors in τ are reset to 0. This completes the update process. At the end of the process, only those arcs in τ which have a flag value of 3 are capable of being removed from τ in addition to a_1 in a valid 2-opt move.

The overall time complexity of Algorithm 2 is $O(mn)$ while that of finding the best neighbor of a tour in conventional implementations is $O(n^3)$. If the graph describing the TSP is dense, then $m \to n^2$ and the time complexities of both implementations are identical. However, for TSPs defined on sparse graphs $m << n^2$, and the new implementation should be significantly faster. We test this conclusion experimentally in the next section.

4 Computational Experience

In this section, we report our computational experience with the new implementation on large sparse ATSPs. We call our implementation for TSPs on sparse asymmetric graphs `TS-SAG`. In order to benchmark its performance, we also implemented a "conventional" tabu search implementation, and called it `TS-CI`. `TS-CI`uses an adjacency matrix to store graph data, and stores tours as vectors of arcs. Both implementations were done in C and the computer experiments were run on a quad core 2.4 GHz processor and 3GB of RAM. We tested the performance of the two implementations on problems defined on graphs with 500, 1000, 2000, and 3000 nodes and with densities varying from 5% to 30%. The tabu list in both implementations were designed to store 16 moves. We used the following two measures to compare the performance of the two implementations:

1. the time required to execute 800 iterations, and
2. the time required to reach the best solution obtained at the end of 800 iterations.

For each problem size-density combination, we generated ten random problems, and for each problem we ran tabu search from four initial tours. The results obtained for each problem is the average of the results from the four initial tours. The results presented for each problem size-density combination is summarized from the results for all ten problems for that combination. Subsection 4.1 provides details of how the problems were generated, and Subsection 4.2 describes the results obtained.

4.1 Generating Random Problems

Our method of generating random TSP instances is as follows. We initially generate $\rho\ n/2$ initial tours at random and add the arcs in the tours to the graph. For each of the tours, we randomly decide on the number of 2-opt neighbors of the tour that we want to generate, and generate those neighboring tours. The arcs in these tours are also added to the graph. For each initial tour we then choose one of the neighbors thus generated and repeat the neighborhood generation step for these chosen tours. Algorithm 3 describes this process.

Algorithm 3 is observed to produce many clusters of tours that are neighbors of each other, and hopefully are interconnected.

4.2 Results from Computer Experiments

The results of our experiments are presented in Tables 1 and 2.

Table 1 presents the time taken by the two implementations to execute 800 iterations for each problem size-density combination. For each combination, the numbers not in parenthesis represents the average of the execution times on all ten instances for the combination while the numbers in parentheses refers to the standard deviations of the execution times. For example, the average of the times taken by `TS-CI` to perform 800 iterations on each of the ten instances of size 500 and density 10% was 19.83 seconds, and the standard deviation of these times was 14.07 seconds.

From the table we observe that `TS-SAG` is consistently faster than `TS-CI`. For each density level, the ratio of execution times taken by `TS-SAG` and `TS-CI` reduces with problem size. This means that the larger the problem, the more attractive is `TS-SAG` compared to `TS-CI` for the same density level. We also note that for each problem size `TS-SAG` performs relatively better with respect to `TS-CI` for smaller density levels. This validates our conjecture that `TS-CI` should be used only when the graph describing the TSP is sparse.

Since tabu search allows worsening moves, the best solution that is obtained after 800 tabu search iterations are possibly achieved after fewer than 800 iterations. Table 2 presents details about the time taken by both the implementations to encounter the solution that they found to be the best at the end of 800 iterations. The reporting convention in Table 2 is the same as that in Table 1.

From the table we observe that `TS-SAG` reaches the best solution earlier than `TS-CI`. However, there is another interesting observation to be made here. Since `TS-SAG` and `TS-CI` are different implementations of the same algorithm (Algorithm 1, the solutions reached by both at the end of the same iteration are identical. However, the ratio of the times taken by `TS-SAG` and `TS-CI` to

Algorithm 3 Data Generation Process

Input: Problem size n, density ρ
Output: A TSP instance $G = (V, A)$ of size n and density ρ, $c(a)$ for all arcs $a \in A$

```
1: /* INITIALIZATION */
2: generate $C'$ as a $n \times n$ matrix of random numbers
3: $V \leftarrow \{1, 2, \ldots, n\}$, $A \leftarrow \emptyset$
4: $f_i \leftarrow \frac{\rho}{k} n(n-1)$

5: /* ITERATION */
6: for $i = 1$ to $\rho\, n/2$ do
7:     $j \leftarrow 0$, $A_i \leftarrow \emptyset$
8:     generate a random tour $\tau_i$
9:     for all arcs $a \in \tau_i$ do
10:        $A_i \leftarrow A_i \cup \{a\}$
11:    end for
12:    if $|A_i| \geq f_i$ then
13:        $A \leftarrow A \cup A_i$
14:    else
15:        while $|A_i| < f_i$ do
16:            generate a random number $n_i$ between 6 and 12
17:            for $j = 1$ to $n_i$ do
18:                generate $\tau_{ij}$, a 2-opt neighbor of $\tau_i$
19:                for all arcs $a \in \tau_{ij}$ do
20:                    $A_i \leftarrow A_i \cup \{a\}$
21:                    if $|A_i| \geq f_i$ break
22:                end for
23:                if $|A_i| \geq f_i$ break
24:            end for
25:            $\tau_i \leftarrow \arg\min\{c(\tau_{ij} : j = 1, \ldots, n_i)\}$ where tour costs are based on the matrix $C'$
26:        end while
27:    end if
28: end for
29: for all arcs $(i, j) \in A$ do
30:    $c((i, j)) = C'[i, j]$
31: end for

32: return $G = (V, A)$, $c(a)$ for all $a \in A$
```

reach the best solution is consistently lower than the ratio of times taken by the two algorithms to complete 800 iterations. This suggests that the `TS-SAG` algorithm is faster in the initial stages of the algorithm than in the later stages. At present, we have no explanation for this behavior. We suspect that it is because of the way the problems have been generated, but need further experimentation to either validate or refute this suspicion.

In summary, the computational experiments validate our assertion in Section 3 that the new tabu search implementation (`TS-SAG`) is more efficient than conventional tabu search implementations (`TS-CI`) for asymmetric TSPs defined on sparse graphs. The computations however show that the dominance of `TS-SAG` over `TS-CI` on sparse graphs becomes more comprehensive when problem sizes increase.

Table 1: Details of the time taken for 800 iterations

Density	**500**		**1000**		**2000**		**3000**	
(ρ)	`TS-CI`	`TS-SAG`	`TS-CI`	`TS-SAG`	`TS-CI`	`TS-SAG`	`TS-CI`	`TS-SAG`
5%	9.88 (2.95)	4.26 (1.16)	82.87 (34.73)	22.27 (7.58)	413.29 (248.49)	84.63 (32.43)	996.84 (787.84)	180.88 (80.72)
10%	19.83 (14.07)	5.68 (1.94)	76.19 (48.27)	20.17 (4.17)	378.23 (607.75)	83.77 (47.10)	751.52 (383.79)	174.82 (23.93)
15%	15.65 (11.54)	5.89 (1.00)	65.97 (41.08)	22.52 (3.32)	295.38 (135.41)	90.31 (9.40)	786.65 (283.00)	208.53 (18.47)
20%	18.51 (12.61)	6.97 (1.13)	54.03 (24.88)	25.06 (1.90)	324.05 (123.16)	106.37 (8.10)	1043.13 (400.23)	257.67 (25.24)
25%	11.96 (5.93)	7.37 (0.52)	60.55 (31.56)	29.21 (2.27)	337.29 (163.00)	121.53 (10.60)	1283.21 (520.35)	301.70 (32.16)
30%	11.62 (5.23)	8.42 (0.50)	68.68 (29.50)	33.58 (2.06)	428.36 (140.23)	141.83 (8.89)	1654.91 (780.79)	355.41 (47.33)

Table 2: Time taken to obtain the best solution

Density	**500**		**1000**		**2000**		**3000**	
(ρ)	`TS-CI`	`TS-SAG`	`TS-CI`	`TS-SAG`	`TS-CI`	`TS-SAG`	`TS-CI`	`TS-SAG`
5%	0.32 (0.32)	0.14 (0.20)	3.59 (0.98)	0.94 (0.36)	44.35 (15.93)	7.92 (2.19)	185.82 (87.71)	26.95 (9.53)
10%	1.28 (0.58)	0.35 (0.31)	14.49 (5.65)	2.26 (0.64)	134.02 (94.79)	17.68 (7.37)	471.19 (208.16)	57.04 (14.37)
15%	2.40 (1.18)	0.55 (0.28)	24.43 (11.95)	3.74 (0.99)	212.28 (109.70)	28.56 (7.89)	676.69 (277.09)	88.95 (18.45)
20%	4.31 (1.83)	0.77 (0.30)	30.51 (15.23)	5.03 (1.13)	276.93 (116.40)	40.28 (8.08)	978.59 (393.56)	134.47 (25.83)
25%	4.93 (2.57)	0.99 (0.25)	43.22 (27.38)	7.16 (1.86)	307.14 (160.83)	51.44 (10.26)	1242.81 (520.21)	178.75 (33.04)
30%	5.87 (3.00)	1.17 (0.35)	55.51 (28.25)	9.33 (2.03)	405.06 (141.00)	67.62 (9.56)	1627.14 (781.43)	238.48 (47.21)

5 Summary

In this paper, we present a novel implementation of tabu search on sparse asymmetric TSPs. 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 asymmetric TSPs. 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 asymmetric TSPs with sizes up to 3000 nodes, and densities up to 30%. It is important to note here that no published paper in the literature only deal with tabu search for problems with more than 1000 nodes. Our experiments show that our implementation is up to five times faster than conventional implementations when solving problems with 3000 nodes and having a graph density of 5%.

There are several interesting ways in which the research reported here can be carried forward. First, our computational experiments have been performed with randomly generated instances of only one type. It may be interesting to see whether the results are similar for other types of randomly generated sparse graphs, and to design implementations that take advantage of particular structures of graphs. Another important direction would be to generate benchmark sparse problems arising in the real world. Second, it would also be interesting to see how other tabu search components such as medium term and long term memory structures can be efficiently utilized in developing more advanced implementations. Third, it would be interesting to see if for other problems, one can develop more efficient implementations than conventional implementations by exploiting problem structures.

References

[1] Basu, S., Ghosh, D.: A review of the tabu search literature on traveling salesman problems, WP-2008-10-01, Working Paper Series, Indian Institute of Management Ahmedabad, India (2008)

[2] Eijkhout, V.: Distributed sparse data structures for linear algebra operations, Technical Report CS 92-169, Computer Science Department, University of Tennessee, Knoxville, TN (1992)

[3] Glover, F.: Tabu search Part-I, ORSA Journal on Computing 1, 190–206 (1989)

[4] Glover, F.: Tabu search Part-II, ORSA Journal on Computing 2, 4–32 (1990)

[5] Glover, F., Laguna, M.: Tabu Search, Kluwer Academic Publisher (1998)

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

[7] Karp, R.M.: Reducibility among combinatorial problems. Miller, R.E. and Thatcher, J.W. (eds.) Complexity of Computer Computations, Plenum Press, 85–103 (1972)

[8] Lawler, E.L., Lenstra, J.K., Rinnooy, A.H.K., Shmoys, D.B.: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley-Interscience Publication (1985)

[9] Saad, Y.: SPARSKIT: A basic tool-kit for sparse matrix computation, version 2 (1994)