Submitted to *INFORMS Journal on Computing*
manuscript (Please, provide the manuscript number!)

An Adaptive Heuristic Approach to Compute Upper and Lower Bounds for The Close-Enough Traveling Salesman Problem

Francesco Carrabs
Department of Mathematics, University of Salerno, Fisciano, Italy, fcarrabs@unisa.it

Carmine Cerrone
Department of Economics & Business Studies, University of Genova, Italy, carmine.cerrone@unimol.it

Raffaele Cerulli
Department of Mathematics, University of Salerno, Fisciano, Italy, raffaele@unisa.it

Bruce Golden
Robert H. Smith School of Business, University of Maryland, College Park, MD, United States, bgolden@rhsmith.umd.edu

This paper addresses the Close-Enough Traveling Salesman Problem, a variant of the Euclidean traveling salesman problem, in which the traveler visits a node if it passes through the neighborhood set of that node. We apply an effective strategy to discretize the neighborhoods of the nodes and the Carousel Greedy algorithm to appropriately select the neighborhoods that, step by step, are added to the partial solution until a feasible solution is generated. Our heuristic, based on these ingredients, is able to compute tight upper and lower bounds on the optimal solution relatively quickly. The computational results, carried out on benchmark instances, show that our heuristic often finds the optimal solution, on the instances where it is known, and, in general, the upper bounds are more accurate than those from other algorithms available in the literature.

Key words: Close-Enough, Traveling Salesman Problem, Neighborhoods, Carousel Greedy, Drones

1. Introduction

In this paper, we study the Close-Enough Traveling Salesman Problem (CETSP), a generalization of the classical TSP. In the CETSP, rather than visit the vertices of a graph, the traveler must visit a specific neighborhood of each vertex. We assume that the neighborhood of a vertex is represented by a circle that has this vertex as a center. Therefore, a vertex of the graph is visited if the traveler passes within this circle or on its border.

Formally, given a set of target points (the vertices of a graph) in Euclidean space, the CETSP seeks to find the shortest tour that starts and ends at the depot and intersects each circle (which is associated with a unique vertex of the graph) once.

The CETSP has a number of practical applications. For instance, consider the task of meter reading for utility companies. Homes and businesses have meters that measure the usage of gas, water, and electricity. Each meter transmits signals which can be read by a meter reader vehicle via radio-frequency identification (RFID) technology if the distance between the meter and the reader is less than r units. Each meter plays the role of a target point and the neighborhood is a disc of radius r centered at each target point. Now, suppose the meter reader vehicle is a drone and the goal is to visit each disc while minimizing the amount of energy expended by the drone. Other applications include military surveillance and forest fire detection by drones (Poikonen et al. (2017)), robot monitoring of wireless sensor networks (Yuan et al. (2007)), and coastal surveillance by submarines.

Variants of the CETSP have been studied for several decades under different names. Some authors assume that travel distances are Euclidean; others assume an underlying street network. The covering salesman problem was introduced by Current (1981) and Current and Schilling (1989). The geometric covering salesman problem was introduced by Arkin and Hassin (1994). See also Mata and Mitchell (1995). This problem was re-named the covering tour problem by Gendreau et al. (1997). The authors provided a new formulation and the first exact algorithm for its solution. The Euclidean TSP with neighborhoods was studied by Dumitrescu and Mitchell (2003). In this paper, new approximation algorithm results were presented. This same problem was given the name CETSP in about 2005 or 2006 and it is the name most commonly used today. The CETSP was introduced into the literature by Gulczynski et al. (2006). Early work on the CETSP includes papers by Dong et al. (2007), Yuan et al. (2007), Shuttleworth et al. (2008), Mennell et al. (2011), and Mennell (2009). See Silberholz and Golden (2007) for related work on the generalized traveling salesman problem (GTSP). More recent and sophisticated approaches to the CETSP have been presented by Behdani and Smith (2014), Carrabs et al. (2017a,b), Coutinho et al. (2016), Yang et al. (2018), and Wang et al. (2019).

Now we discuss several of the CETSP algorithms in some detail. Mennell (2009) and Mennell et al. (2011) propose a heuristic algorithm based on Steiner zones, that is, the nonempty zones obtained by the intersections of the neighborhood sets, consisting of the following three phases: (a) identifying a collection of Steiner zones that cover every neighborhood set; (b) representing each Steiner zone with one of its points and finding a TSP tour over these representative points; and (c) improving upon the feasible CETSP tour found in the previous step by possibly modifying the location of each Steiner zone's representative point. Computational results show that the heuristic is able to efficiently find a good feasible CETSP tour. Moreover, the authors provide a second order cone programming model to solve the Touring Steiner Zones Problem when the sequence of visits is given. Finally, Mennell et al. (2011) highlight that developing lower bounds for the CETSP is a non-trivial task.

Behdani and Smith (2014) were the first to try to solve the CETSP exactly. They applied mixed integer programming, Benders decomposition, and an iterative algorithm. They generated reasonably good upper and lower bounds on numerous instances. Carrabs et al. (2017b) improved upon the work of Behdani and Smith (2014) and obtained tighten upper and lower bounds for the instances given by Behdani and Smith (2014). Their results were obtained by introducing a new (internal) discretization schema for the neighborhoods and a graph reduction algorithm that significantly reduces the problem size. The results of Carrabs et al. (2017b) are further improved by Carrabs et al. (2017a) with a heuristic that embeds an improved version of the internal discretization scheme proposed by Carrabs et al. (2017b) with a second-order cone programming algorithm proposed by Mennell et al. (2011). Coutinho et al. (2016) were the first authors to propose an exact branch-and-bound algorithm for the CETSP. The algorithm was able to solve all the instances of Behdani and Smith (2014) to optimality. For larger instances, as expected, this exact algorithm is less successful. Faigl (2018) proposed an unsupervised learning based approach, named Growing Self-Organizing Array (GSOA), to solve routing problems and, in particular, the CETSP. The author tested the performance of GSOA on the instances from Mennell (2009). The results show that this approach is very fast, but the gap from the best known solutions is non-trivial. In related work, Faigl et al. (2019) extended GSOA to solve a three-dimensional variant of the CETSP. They also propose a second heuristic. Both heuristics are very fast. In future work, it might be possible to apply sophisticated optimization post-processors to further improve these solutions. Yang et al. (2018) developed a hybrid heuristic that combines particle swarm optimization and a genetic algorithm. The heuristic performed as well as the best heuristic from Mennell (2009) on the instances from Mennell

(2009). Wang et al. (2019) extended the work of Mennell et al. (2011) and developed a Steiner Zone Variable Neighborhood Search heuristic (SZVNS). The authors conducted computational experiments on 842 instances and demonstrate that overall performance with respect to solution quality and running time is high.

In this paper, we apply the Carousel Greedy algorithm (a type of generalized greedy algorithm) to select the neighborhoods that, step by step, are added to the partial solution, until a feasible solution is generated. We develop a heuristic approach, based on these concepts, that is able to compute tight upper and lower bounds on the optimal solution relatively quickly. The computational results, carried out on benchmark instances, demonstrate that our heuristic often finds the optimal solution, for the instances where it is known. In general, our heuristic performs well with respect to the best of the existing algorithms. Furthermore, we point out that our algorithm is capable of generating reasonably tight lower bounds quickly.

The remainder of this paper is organized as follows. In Section 2, we introduce terminology and notation to be used throughout the paper. In Section 3, we survey previous research on the CETSP. In Sections 4 and 5, we introduce two procedures for the calculation of effective lower and upper bounds. In Section 6, we present our new heuristic for the CETSP, which we call the *(lb/ub)Alg*. Computational results as reported in Section 7. Finally, conclusions are provided in Section 8.

2. Definitions and notation

Given a two-dimensional plane, let N be a set of *target points* placed in the plane, with $|N| = n$, and let $v_0 \notin N$ be the *depot*. The edge length between two points, v_i and v_j, is given by the Euclidean distance between v_i and v_j and it is denoted by $\ell(v_i, v_j)$. A circumference C_v, with center v and radius r_v, is associated with each target point $v \in N$ (Figure 1(a)).

Given two points a and b on the boundary of C_v, we denote by $\overline{a,b}$ the *chord* between these points and by $\widehat{a,b}$ the *circular arc* from a to b in the clockwise direction (Figure 1(b)). The set of points within and on C_v compose the *neighborhood* $N(v)$ of v. Figure 1(c) shows the neighborhoods associated with the target points. We define $C = \bigcup_{v \in N} C_v$ and, without loss of generality, we suppose that $v_0 \notin N(v)$, $\forall v \in N$. A feasible tour T for the CETSP is a cycle starting and ending at the depot v_0 and intersecting every neighborhood $N(v)$. In Figure 1(d) we depict a feasible tour T intersecting the neighborhoods $N(v_1), \ldots, N(v_6)$.

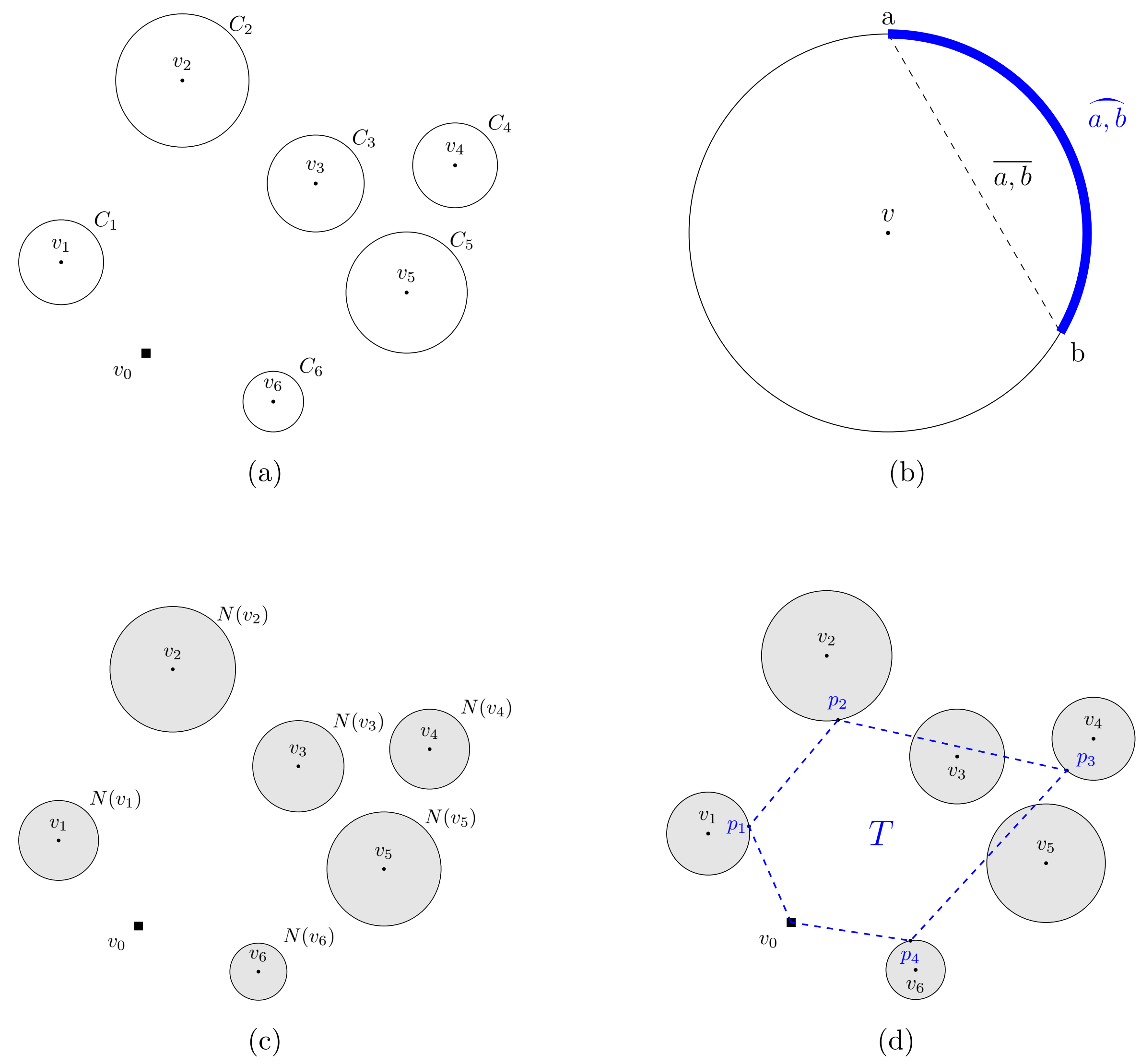

Figure 1 **(a) Set $N = \{v_1, \dots, v_6\}$ of target points and the circumferences $C = \{C_1, \dots, C_6\}$ associated with them. (b) The chord $\overline{a,b}$ and the circular arc $\widehat{a,b}$. (c) The neighborhoods $N(v_1), \dots, N(v_6)$ associated with the target points $\{v_1, \dots, v_6\}$. (d) A feasible tour T defined by turn points p_1, p_2, p_3, p_4, and v_0.**

The total cost of T is denoted by $w(T)$ and it is equal to the sum of the edge lengths in T. The CETSP consists of finding the shortest tour T^* intersecting every neighborhood $N(v)$. Finally, let us define the *turn points* as the points of a tour where a direction change occurs. Any tour can be uniquely identified through its turn points. For instance, the tour T in Figure 1(d) is identified by turn points p_1, p_2, p_3, p_4, and v_0. For the convenience of the reader, all notation used in this paper is presented in Table 9.

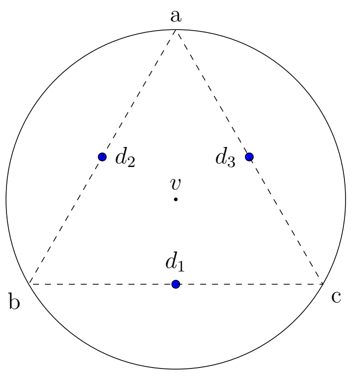

Figure 2 IPD scheme for $k = 3$.

3. The previous approaches

The computation of upper and lower bounds for the CETSP was recently addressed in Behdani and Smith (2014), Carrabs et al. (2017a,b), Yang et al. (2018). The main idea behind these papers is the computation of these bounds through the discretization of the neighborhoods. More specifically, for each neighborhood $N(v)$, we define a new discretized neighborhood $\hat{N}(v)$ that is composed of some points of $N(v)$, appropriately selected. We refer to these points as *discretization points* and their selection is carried out by using a *discretization scheme*. In this paper, we use the *internal point discretization* (IPD) scheme proposed by Carrabs et al. (2017a,b) to discretize the neighborhoods. This scheme works as follows. Given k discretization points, the IPD scheme divides C_v into k equal circular arcs and, for each arc $\widehat{a,b}$, places a discretization point in the middle of the chord $\overline{a,b}$. In Figure 2 the IPD scheme, with three discretization points, is shown. Here $\hat{N}(v) = \{d_1, d_2, d_3\}$. For $k = 1$, the IPD schema places the discretization point at the center of the circle.

Let $\hat{G} = (\hat{V}, \hat{E})$ be the complete graph induced by discretization points, that is $\hat{V} = \bigcup_{v \in N} \hat{N}(v) \cup \{v_0\}$. It is easy to see that the weight of any tour $\hat{T}$, that starts and ends at the depot and that visits a discretization point in each neighborhood, is an upper bound of $w(T^*)$. Figure 3 shows target points $\{v_1, \ldots, v_6\}$, the optimal tour T^* (dotted lines), and the discretized optimal tour $\hat{T}^*$ (dashed lines). It is easy to see from this figure that $w(\hat{T}^*) \geq w(T^*)$.

From now on, we will use the terms T and $\hat{T}$ to denote feasible tours of the CETSP computed by using the points of $N(v)$ and of $\hat{N}(v)$, $\forall v \in N$, respectively. To find an upper bound of $w(T^*)$, as tight as possible, we compute the shortest tour $\hat{T}^*$ by solving the generalized traveling salesman problem (GTSP) on $\hat{G}$. However, since the construction of

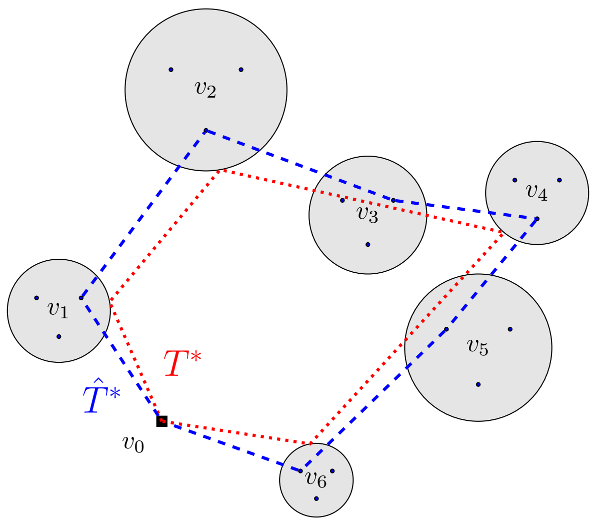

Figure 3 **The discretized optimal tour $\hat{T}^*$ and the optimal tour T^*.**

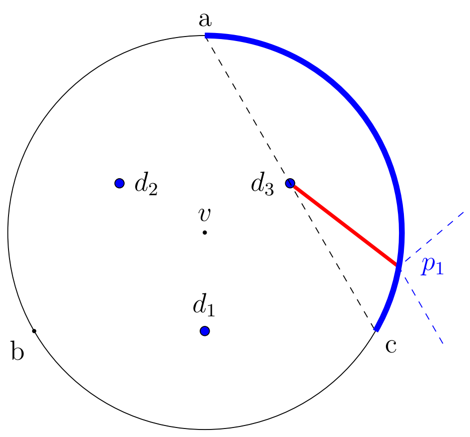

Figure 4 **In red the distance between the turn point p_1 and the discretization point closest to it (d_3). This distance, multiplied by 2, is the discretization error $\xi(v)$ that occurs on $\hat{N}(v)$ due to the use of the discretization points.**

$\hat{T}^*$ is carried out by using only the discretization points, a *discretization error* $\xi(v)$ occurs, in each neighborhood $\hat{N}(v)$, with respect to the turn point p_i of T^* in $N(v)$. If d_i is the discretization point of $\hat{N}(v)$ closest to the turn point p_i, then $\xi(v)$ is equal to two times the length of $\overline{p_i, d_i}$. For instance, in Figure 4 the neighborhood $N(v)$ is discretized by using three points d_1, d_2 and d_3. Since the tour T^* intersects $N(v)$ at the turn point p_1, then $\xi(v)$ is equal to two times the length of $\overline{p_1, d_3}$, one time to come from p_1 to d_3 and another one to come back.

Note that the maximum distance between p_1 and d_3, on the circular arc $\widehat{a, c}$, occurs when d_3 coincides with the points a or c. From trigonometry, the length of $\overline{a, d_3}$ is equal to

$2r_v sin(\frac{\pi}{k})$ (see Carrabs et al. (2017a) for more details). Now, since the turn points of T^* are unknown, we always set $\xi(v)$ to this maximum distance.

In the following, to denote the discretization error carried out on a neighborhood $\hat{N}(v)$ by using k discretization points, we use the notation $\xi_k(v)$.

From the information reported above, we derive that, by subtracting from $w(\hat{T}^*)$ the discretization errors associated with any discretized neighborhood, the resulting value is a lower bound of $w(T^*)$:

$$lb_1 = w(\hat{T}^*) - \sum_{v \in N} \xi(v). \tag{1}$$

In Carrabs et al. (2017a), we provide a detailed description of how this lower bound is derived. For the convenience of the reader, this description is included in the Appendix.

4. Lower bound computation

From equation 1, we know that the lower bound value depends on the values of $w(\hat{T}^*)$ and $\xi(v)$, $\forall v \in N$. In particular, as $w(\hat{T}^*)$ increases, the quality of the lower bound increases. On the other hand, as $\sum_{v \in N} \xi(v)$ increases, the quality of the lower bound decreases. In this paper, we propose a procedure, named *lbc*, that tries to maximize the lower bound value by computing it on a subset of target points appropriately selected for this aim. Our approach is based on the following observation.

Observation 1. Let $\hat{G}$ and $\hat{G}_s$ be the graphs induced by discretization points used to discretize the neighborhoods of N and $N_s \subset N$ (we define N_s below), respectively. The following properties hold.

1. Any feasible tour of CETSP in $\hat{G}$ is a feasible tour for $\hat{G}_s$ also.
2. The cost of the optimal tour $\hat{T}^*$ of $\hat{G}$ will always be greater than or equal to the cost of the optimal solution $\hat{T}_s^*$ for $\hat{G}_s$, i.e., $w(\hat{T}^*) \geq w(\hat{T}_s^*)$.
3. Any lower bound of $w(\hat{T}_s^*)$ is a lower bound for $w(\hat{T}^*)$ also.

To compute the lower bound, procedure *lbc* selects a subset of target points $N_s \subset N$ and it discretizes the neighborhoods associated with these target points. Subsequently, it computes a tour $\hat{T}_s$ by solving the GTSP on $\hat{G}_s = (\hat{V}_s, \hat{E}_s)$, where $\hat{V}_s = \bigcup_{v \in N_s} \hat{N}(v)$, and finally it obtains a lower bound of $w(\hat{T}^*)$ via the equation:

$$lb_2 = w(\hat{T}_s) - \sum_{v \in N_s} \xi(v). \tag{2}$$

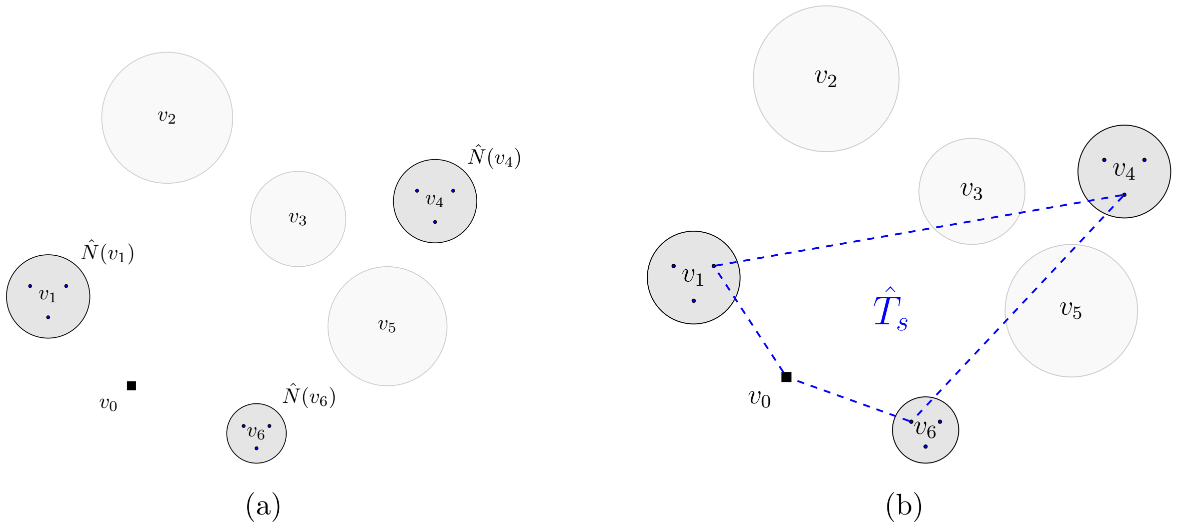

Figure 5 **(a) From the set of target points $\{v_1,\dots,v_6\}$ we select v_1, v_4, and v_6, and we discretize their neighborhoods. (b) Tour $\hat{T}_s$ is computed by solving the GTSP on the graph induced by points $\{\hat{N}(v_1)\cup\hat{N}(v_4)\cup\hat{N}(v_6)\cup\{v_0\}\}$.**

Notice that this equation is a version of equation 1, but applied on a subset of target points. Our aim here is to select this subset of target points so that $lb_2 > lb_1$. Figure 5 shows how the computation of the lower bound is carried out.

Given the six target points $\{v_1,\dots,v_6\}$ shown in Figure 5(a), *lbc* selects the target points v_1, v_4, and v_6 and it discretizes the neighborhoods associated with these target points. Let $\hat{G}_s$ be the graph induced by discretization points of $\hat{N}(v_1)$, $\hat{N}(v_4)$, $\hat{N}(v_6)$, and v_0. By solving the GTSP on $\hat{G}_s$, *lbc* finds the tour $\hat{T}_s^*$ shown in Figure 5(b). Finally, the lower bound of $w(\hat{T}^*)$ is obtained by subtracting from $w(\hat{T}_s^*)$ the discretization errors $\xi(v_1)$, $\xi(v_4)$, and $\xi(v_6)$.

The lower bound computation using procedure *lbc*, rather than the approaches described in Section 3, offers two main advantages. First of all, this computation is faster because the GTSP problem is solved on graph $\hat{G}_s$ which is usually much smaller than $\hat{G}$. The second advantage is the ability to produce tight lower bounds due to an appropriate selection of the target points to use. Indeed, if on one hand $w(\hat{T}_s^*) \le w(\hat{T}^*)$, on the other hand the number of discretization errors subtracted from $w(\hat{T}^*)$ is usually much larger than the number subtracted from $w(\hat{T}_s^*)$. For instance, let us suppose that, in the previous example, $w(\hat{T}^*) = 20$, $w(\hat{T}_s^*) = 17$ and the discretization errors associated with the neighborhoods of $\hat{G}$ are: $\xi(v_1) = 0.4$, $\xi(v_2) = 1.7$, $\xi(v_3) = 0.9$, $\xi(v_4) = 0.3$, $\xi(v_5) = 1.2$, $\xi(v_6) = 0.1$. By computing the lower bounds with the old and the new method, we obtain: $w(\hat{T}^*) - \sum_{i=1}^{6}\xi(v_i) = 15.4$ and

$w(\hat{T}^*_s) - \xi(v_1) - \xi(v_4) - \xi(v_6) = 16.2$. Notice that, even if the cost of $w(\hat{T}^*)$ is 17% greater than the cost of $w(\hat{T}^*_s)$, the subtraction of all the discretization error values from $w(\hat{T}^*)$ significantly reduces the final lower bound. Obviously, this is not always true; it depends on the selected target points N_s and the discretization error values. However, we will show in Section 7 that, with the appropriate selection of the target points in N_s, the lower bound computed with this new approach is often tighter than the lower bound computed by other approaches proposed in the literature.

4.1. Selection of Target Points

The key point of the algorithm, described in the previous section, is the definition of the subset $N_s \subset N$. We implemented a Carousel Greedy (CG) algorithm (denoted by CarouselGreedy) to carry out this selection. This is an enhanced greedy algorithm which has been shown to be both effective and computationally efficient when applied to a wide variety of subset selection problems (see Cerrone et al. (2017)). It is evident that, based on equation (2), to obtain a tight lower bound we seek to maximize $w(\hat{T}_s)$ and minimize $\sum_{v \in N_s} \xi(v)$. Our CG algorithm is developed taking into account both of these goals and its pseudocode is shown in Algorithm 1.

Lines 1-7 present a greedy algorithm used to generate a solution that is, subsequently, improved by CarouselGreedy (line 8-16). A detailed description of the *FindNextPoint* and *NumDiscPoints* procedures is given afterwards. The number of discretization points k to use is computed by invoking the *NumDiscPoints* procedure in line 2. At each iteration of the while loop (line 3), the procedure *FindNextPoint* selects the next target point to be added to N_s. If *FindNextPoint* fails to find a target point, i.e., $v = NULL$, the while loop is stopped (line 5); otherwise a new iteration is carried out. The while loop is repeated until either the threshold $nMax$ is reached or the condition on line 5 is satisfied.

The first step of CarouselGreedy consists of invoking a greedy procedure to obtain the set N_s (line 9). The cardinality of N_s is saved by the variable *dim* (line 10). On line 11, CarouselGreedy removes from N_s the β oldest target points where β is equal to 5% of *dim*. In the next while loop (line 12-15), the algorithm removes the oldest target point of N_s (line 13) and invokes the procedure *FindNextPoint* to choose the next point to add to N_s. This operation is repeated $\alpha \cdot dim$ times with $\alpha = 30$. Note that in this while loop the cardinality of N_s is always equal to 95% of *dim*, if *FindNextPoint* never returns NULL.

Algorithm 1: *Carousel Greedy*

```
1  Greedy ($N, N_s, v_0, \Delta, nMax$)
2      $k \leftarrow NumDiscPoints(N, N_s, v_0, \Delta, nMax)$;
3      while $|N_s| \cdot k \leq nMax$ do
4          $v \leftarrow FindNextPoint(N, N_s, v_0, \Delta, k)$;
5          if $v = NULL$ then break;
6          else $N_s \leftarrow N_s \cup \{v\}$;
7      return $N_s$;

8  CarouselGreedy ($N, v_0, \alpha, \beta, \Delta, nMax$)
9      $N_s \leftarrow$ Greedy$(N, \emptyset, v_0, \Delta, nMax)$;
10     $dim \leftarrow |N_s|$;
11     Remove the $\beta$ oldest target points from $N_s$;
12     while $iter \leq \alpha \cdot dim$ do
13         Remove the oldest target point from $N_s$;
14         $N_s \leftarrow N_s \cup FindNextPoint(N, N_s, v_0, \Delta, k)$;
15         iter++;
16     return Greedy$(N, N_s, v_0, \Delta, nMax)$;
```

After the end of the while loop, CarouselGreedy invokes again Greedy on N_s (line 16) to restore the original cardinality dim, if possible. The final subset N_s is returned.

The selection of more promising target points is carried out by *FindNextPoint*; the pseudocode is shown in Algorithm 2. In the foreach loop (line 1-5), the procedure assigns to each target point v, not yet selected, a “score” $val(v)$. That is, for each point $v_i \in N_s \cup \{v_0\}$, we compute its distance w_i from v. This distance is computed as the distance $\ell(v, v_i)$ minus the radii of the circumferences, C_v and C_{v_i}. The higher this distance w_i is, the greater will be the score assigned to v. Here, we used w_i instead of $\ell(v, v_i)$ to allow negative values when C_v and C_{v_i} intersect each other. Indeed, in this case, we do not want to classify vertex v as promising because it is too close to v_i. The next step consists of selecting the four vertices of $N_s \cup \{v_0\}$ with the lowest w_i values (line 3). If $|N_s| < 4$, then the procedure selects all the vertices in N_s. A weighted average μ is computed using the w_i values (line

Algorithm 2: *FindNextPoint*(N, N_s, v_0, Δ, k)

```
1 foreach $v \in N \setminus N_s$ do
2     Compute $w_i \leftarrow \ell(v, v_i) - r_v - r_{v_i} \qquad \forall v_i \in N_s \cup \{v_0\}$;
3     W.l.o.g. let $v_1, v_2, v_3$, and $v_4$ be the four vertices of $N_s \cup \{v_0\}$ having the
        minimum $w_i$ value, respectively;
4     $\mu \leftarrow \frac{8w_1+4w_2+2w_3+w4}{15}$;
5     $val(v) \leftarrow \Gamma_v \Delta \mu - \xi_k(v)$;
6 $v' \leftarrow \arg\max_{v \in N \setminus N_s} \{val(v)\}$;
7 if $val(v') > 0$ then return $v'$;
8 else return $NULL$;
```

4). More specifically, we assign a priority to the selected vertices according to their w_i values (smaller is better). In particular, the priority assigned to w_i has to be double the priority assigned to w_{i+1}. For this reason, w_1 is multiplied by eight, w_2 by four, and so on. To normalize the value of μ, this sum is divided by the sum of these priorities. Finally, the score $val(v)$ is computed by multiplying μ for the two parameters Δ and Γ_v and by subtracting from the obtained value the discretization error $\xi_k(v)$. The most promising target point v' is the one with the maximum score (line 6). If the score $val(v')$ is positive then v' is returned (line 7), otherwise the procedure returns NULL (line 8) because v' is not considered promising.

There are three key parameters in this procedure: Δ, Γ_v, and k. Δ is used to increase or decrease the value of μ in the computation of $val(v)$. In particular, Δ is increased as the probability of finding new promising points is increased. Notice that this parameter does not depend on the vertex v considered. The parameter Γ_v is a "recommendation value" associated with each vertex v and it is used to influence the selection of vertices carried out by *FindNextPoint*. Indeed, the higher Γ_v is, the higher is the probability of selecting the vertex v. In Section 6, we describe in detail how the Δ and Γ_v parameters are dynamically updated. Let us suppose, in this section, that these parameters are equal to 1 so that the computation of $val(v)$ is not affected by them.

The third parameter of the procedure is k which affects the discretization error value $\xi_k(v)$. In particular, as k increases, $\xi_k(v)$ decreases. Also, as $|N_s|$ increases, $\xi_k(v)$ decreases.

Algorithm 3: *NumDiscPoints*$(N, N_s, v_0, \Delta, nMax)$

```
1 $k = 1$;
2 while $TRUE$ do
3     $N_s \leftarrow \emptyset$;
4     while $|N_s| \cdot k \le nMax$ do
5         $v \leftarrow FindNextPoint(N, N_s, v_0, \Delta, k)$;
6         if $v = NULL$ then break;
7         else $N_s \leftarrow N_s \cup \{v\}$;
8     if $\left\lfloor \frac{nMax}{|N_s|} \right\rfloor > k$ then $k \leftarrow k + 1$;
9     else return $k$ ;
```

Therefore, k and $|N_s|$ are closely related values. Now, if $nMax$ is the total number of available discretization points, then the ratio $\frac{nMax}{|N_s|}$ is the number of discretization points associated with each neighborhood $N(v)$, $v \in N_s$. The idea is to assign to k a value as close as possible to $\frac{nMax}{|N_s|}$. However, since $|N_s|$ depends on k, we compute this value with an iterative procedure, named *NumDiscPoints*; the pseudocode is shown in Algorithm 3.

We start with $k = 1$ and we use this value to build N_s through a greedy algorithm (lines 3-7) whose while loop is the same as in the greedy algorithm used in Algorithm 1 (lines 3-6). After the construction of N_s, we check to see if k is less than $\frac{nMax}{|N_s|}$(line 8). If this is the case, then we have to increase the value of k so that $|N_s|$ increases and $\frac{nMax}{|N_s|}$ decreases. Otherwise, we obtained the value we were looking for and we return k (line 9).

At this point, we are ready to describe how we compute the lower bound of $w(T^*)$. Given the set of target points N_s selected by CG, for each target point $v \in N_s$, we discretize its neighborhood $N(v)$ by using the internal discretization scheme proposed by Carrabs et al. (2017a,b) with $\left\lceil nMax/|N_s| \right\rceil$ discretization points. Note that, by construction, $|N_s| \le nMax$. If CG selects $nMax$ target points, then just one discretization point is used for each neighborhood and this point coincides with the center of the circumference.

Let $G_s = (V_s, E_s)$ be the subgraph formed by the depot and the discretization points associated with the neighborhoods of target points in N_s, that is, $V_s = \{\bigcup_{v \in N_s} \hat{N}(v) \cup v_0\}$ and $E_s = \{(v_i, v_j) : v_i, v_j \in V_s\}$. By solving the GTSP on G_s, we obtain the discretized tour $\hat{T}_s^*$. By subtracting from $w(\hat{T}_s^*)$ the discretization error associated with the neighborhoods

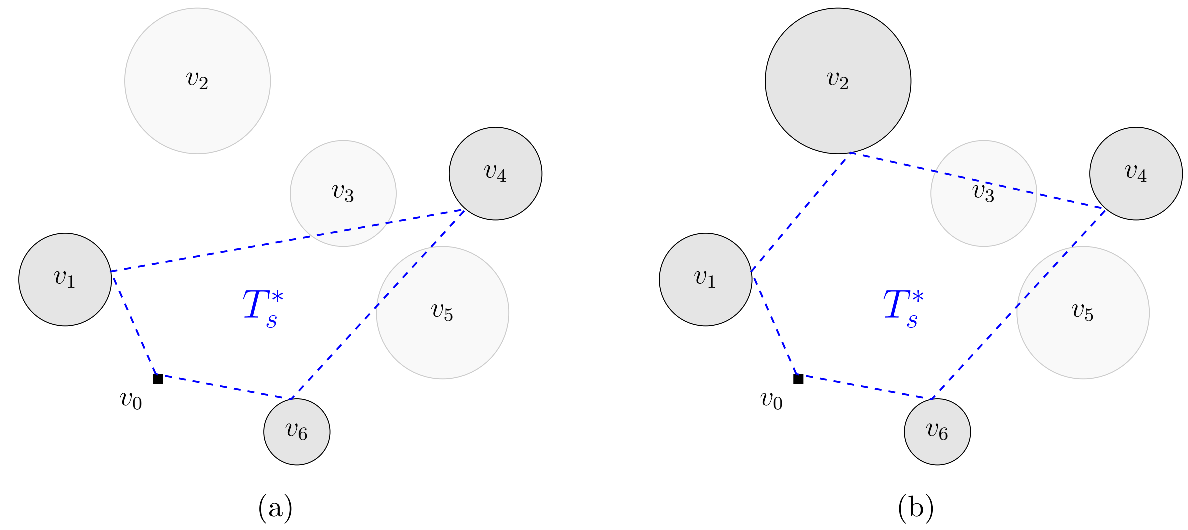

Figure 6 **(a) Optimal Tour T_s^* of CETSP when $N_s = \{v_1, v_4, v_6\}$. (b) Optimal Tour T_s^* of CETSP when $N_s = \{v_1, v_2, v_4, v_6\}$.**

of target points in N_s, we obtain the lower bound lb_2 we were looking for (see equation (2)).

5. Upper bound computation

In this section, we describe how to compute an upper bound of $w(T^*)$ by using the tour $\hat{T}_s^*$ computed in the previous section. Our strategy, named *ubc*, is based on the following observation.

OBSERVATION 2. When the visiting sequence of neighborhoods is fixed a priori, the CETSP corresponds to the Touring Steiner Zones Problem that can be formulated as a second-order cone program (Mennell (2009)) and can be solved in polynomial time (Andersen et al. (2003)).

Note that the tour $\hat{T}_s^*$ defines a visiting sequence of the target points in N_s. For instance, in Figure 5(b) the visiting order, defined by $\hat{T}_s$, is v_1, v_4, and v_6.

We use the second-order cone programming model (SOCP), described by Mennell (2009) and by Carrabs et al. (2017a), to find the optimal tour T_s^* of CETSP, according to the visiting sequence defined by $\hat{T}_s^*$. In Figure 6(a), we depict the optimal tour T_s^* obtained with $N_s = \{v_1, v_4, v_6\}$. The determination of an upper bound of $w(T^*)$ is carried out according to the following two cases:

Case 1. T_s^* does not intersect the neighborhoods of all target points in N.

This is the situation shown in Figure 6(a). Since the neighborhood $N(v_2)$ is not covered by T_s^*, this tour is not feasible for the CETSP on N. We say a target point v is *uncovered*, with respect to a tour T, if and only if T does not intersect $N(v)$. To produce a feasible tour, at each iteration, *ubc* adds to N_s one uncovered target point and it builds a new tour T_s^* by using the SOCP model on G_s. The algorithm stops when T_s^* intersects all the neighborhoods. Since our aim is to produce an upper bound as tight as possible, at each iteration *ubc* adds to N_s the uncovered target point v having the minimum distance from an edge of the tour. In this way, we try to minimize the cost of the new tour built on N_s.

Figure 6 shows how *ubc* works. Tour T_s^*, depicted in Figure 6(a), is not feasible because it does not intersect $N(v_2)$. Therefore, the algorithm adds v_2 to N_s and it builds a new tour by using the SOCP model on the new graph composed of neighborhoods $N(v_1)$, $N(v_2)$, $N(v_4)$, and $N(v_6)$. In Figure 6(b), we depict this new tour T_s^*. Since this tour intersects all the neighborhoods, it is a feasible solution and, thus, *ubc* stops.

Case 2. T_s^* intersects the neighborhoods of all target points in N.

In this case T_s^* is a feasible tour of CETSP on N and $w(T_s^*)$ is an upper bound on $w(T^*)$. Since there are no uncovered targets, the algorithm stops.

6. *(lb/ub)Alg* algorithm

In the previous two sections, we described how to compute an upper and lower bound for the CETSP. These two procedures are fast, but it is clear that the quality of the bounds computed depends on several choices that we made during the computations. For instance, the quality of the lower bounds depends on the vertices selected by *FindNextPoint* while the quality of the upper bounds depends on the uncovered vertices, added to the tour to obtain a feasible solution.

Our idea here is to develop a new algorithm (*(lb/ub)Alg*) that, at each iteration, *i)* invokes both *lbc* and *ubc* procedures, *ii)* updates the values of Δ and Γ_v, and *iii)* uses the new values of these parameters to impact the behavior of *lbc* and *ubc* during the next iteration. In particular, *(lb/ub)Alg* focuses the attention on the parameters Δ and Γ_v, used in the *FindNextPoint* procedure, because the values of these parameters affect the construction of tour T_s^* and, then, the quality of the lower bound computed by *lbc*. On the other hand, by providing several starting tours T_s^* to *ubc*, there are more chances to improve the final upper bound.

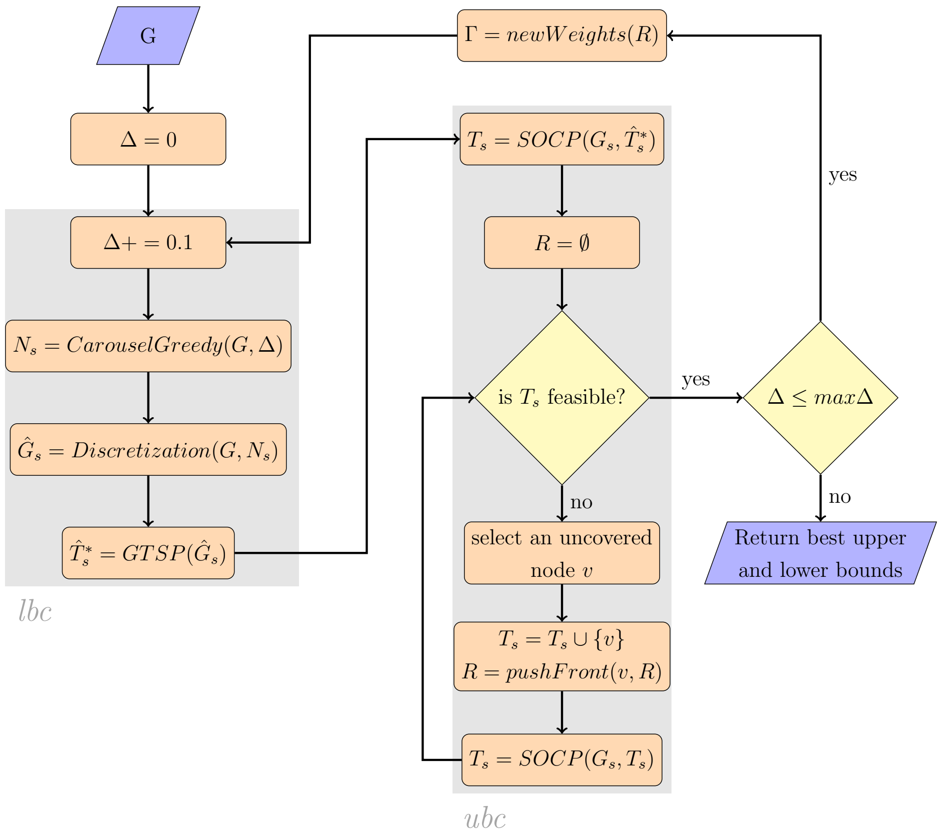
G
$\Delta = 0$
$\Delta += 0.1$
$N_s = CarouselGreedy(G, \Delta)$
$\hat{G}_s = Discretization(G, N_s)$
$\hat{T}_s^* = GTSP(\hat{G}_s)$
lbc
$\Gamma = newWeights(R)$
$T_s = SOCP(G_s, \hat{T}_s^*)$
$R = \emptyset$
is T_s feasible?
yes
no
select an uncovered node v
$T_s = T_s \cup \{v\}$
$R = pushFront(v, R)$
$T_s = SOCP(G_s, T_s)$
ubc
$\Delta \leq max\Delta$
yes
no
Return best upper and lower bounds

Figure 7 **(lb/ub)Alg flowchart**

Figure 7 displays the flowchart of *(lb/ub)Alg*. The algorithm takes as input the graph G and it initializes Δ to 0. The following four steps describe how the *lbc* procedure works. First of all, the value of Δ is increased by 0.1. Then, the CarouselGreedy procedure builds the set of nodes N_s that is used to create the discretized graph $\hat{G}_s$. Finally, the tour $\hat{T}_s^*$ is computed by solving the GTSP on $\hat{G}_s$. As described in Section 4.1, the value of Δ affects the score *val(v)* assigned to any vertex $v \in N \setminus N_s$ by *FindNextPoint*. In particular, when Δ is lower than 1, $val(v)$ is decreased, and then fewer vertices are selected by *FindNextPoint*. On the other hand, when Δ is greater than 1, $val(v)$ is increased and more vertices can be selected by *FindNextPoint*. Since it is not possible to know a priori what is the best set N_s of vertices to select to obtain the best lower bound, we iteratively modify the Δ value to compute different lower bounds, among which the best one will be chosen. Since the incremental step of Δ is equal to 0.1, during the first 10 iterations of the algorithm fewer points will be selected by *FindNextPoint* while, in later iterations, this number increases.

In the middle of the flowchart the steps carried out to compute the upper bound are described. Based on the sequence defined by $\hat{T}_s^*$, the new tour T_s is computed by using the SOCP model on G_s. The sequence of "*recommended*" vertices R is initialized to an empty sequence. At this point, *(lb/ub)Alg* checks if T_s is a feasible tour of G. There are two possible cases that *(lb/ub)Alg* has to take into account.

- If T_s is infeasible, it means that there is at least one target point whose neighborhood is not intersected by T_s. In this case, *(lb/ub)Alg* selects one of the uncovered vertices, let us say v, and it adds v to the tour T_s as described in Section 5. Moreover, v is added at the beginning of the sequence R too. Finally, the SOCP model is invoked according to the new visitation sequence defined by T_s and, again, *(lb/ub)Alg* checks to see if the new tour obtained is feasible. This loop is repeated until a feasible tour is built.
- If T_s is feasible, *(lb/ub)Alg* checks to see if the stop criterion ($\Delta \leq max\Delta$) is satisfied. If this is the case, *(lb/ub)Alg* stops by returning the best upper and lower bounds found during the procedure. Otherwise, the Γ values, associated with the target points, are updated by *newWeights* procedure according to R and a new iteration of *(lb/ub)Alg* starts by increasing the Δ value by 0.1.

In our implementation, $max\Delta$ is set to 3 and, then, with an increment step of Δ equal to 0.1, *(lb/ub)Alg* carries out a total of 30 iterations. We have to describe now why we introduced the sequence of recommended vertices R and how the procedure *newWeights* updates the weights Γ_v. During the computation of the upper bound, carried out by *ubc*, some uncovered vertices are added to the starting tour T_s to obtain a feasible solution for CETSP on G. Let us denote by R the sequence of the uncovered vertices selected. Since the points in R are necessary to produce a feasible solution through the SOCP model, we use the CarouselGreedy procedure to select these points for the computation of the next lower bound. To this end, we have to create a mechanism that increases the chances of introducing these points in N_s. We know that the construction of N_s depends on the *FindNextPoint* procedure and, in particular, on the score assigned to each point in line 5 of this procedure. Obviously, if the discretization error $\xi_k(v)$, with $v \in R$, is high, there are few chances to introduce the vertex v in N_s. For this reason, we add the parameter Γ_v into the computation of the score $val(v)$. With this parameter, even a target point v with a high discretization error value could be inserted in N_s if its Γ_v value is sufficiently high, as described in the following. The setting of Γ_v values is carried out by the *newWeight* procedure as follows. First, *newWeight* sets $\Gamma_v = 1$ for all $v \in N$. Subsequently, to each Γ_v, $v \in R$, we add a weight equal to $\frac{1}{pos_v}$ where pos_v is the position of the vertex v inside the sequence R. For instance, if $R = \langle v_3, v_7, v_5, v_{12} \rangle$ then we have $\Gamma_{v_3} = 1 + \frac{1}{1}$, $\Gamma_{v_7} = 1 + \frac{1}{2}$, $\Gamma_{v_5} = 1 + \frac{1}{3}$, $\Gamma_{v_{12}} = 1 + \frac{1}{4}$. Notice that the lower the position of a point v in R, the higher is its value Γ_v and the higher is its probability of being inserted into N_s. We made this choice because the first point in R corresponds to the last uncovered point whose insertion in T_s has produced a feasible solution. However, according to the selection criterion of uncovered vertices, used by *ubc*, v is the point of R furthest from the tour T_s generated at the first step of the *ubc* procedure by invoking $SOCP(G_s, \hat{T}_s^*)$. As a consequence, by using this point to compute $\hat{T}_s^*$, in *lbc*, the increment of $w(\hat{T}_s^*)$ should be higher than the increment obtained with the other points in R. Obviously, the same reasoning holds between the point in the second position of R, and the points in the next position, and so on.

In conclusion, the Γ_v parameter allows CarouselGreedy to recommend some vertices that might be useful to improve the quality of the lower bound.

7. Computational Tests

In this section, we describe the results of *(lb/ub)Alg* algorithm obtained during our computational test phase. Our algorithm was coded in Java while the mathematical formulations were implemented and solved using the IBM ILOG CPLEX 12.6.1 solver. All tests were performed in single thread mode on a machine with an Intel i5 processor running at 2.3 GHz and 8 GB of RAM.

The computational tests are carried out on 842 instances divided into three sets. The first set of 720 instances has 6, 8, 10, 12, 14, 16, 18, or 20 customers and three customer radius values: 0.25, 0.50, and 1.00. There are 30 instances for each combination of the number of customers and the radius of customers. This set was proposed by Behdani and Smith (2014). The second set of 60 instances has 25 or 30 customers and the same three customer radius values. There are 10 instances for each combination of the number of customers and the radius of customers. This last set was proposed by Carrabs et al. (2017b). In the following, we will refer to these two sets of instances as the "CETSP" instances. All the CETSP instances have been solved optimally by Coutinho et al. (2016). The third set has

62 instances with 100 to 1001 customers and it was proposed by Mennell (2009)[1]. These instances are divided into the following three groups:

- To the first group belong the 27 instances whose names start with *team*, *rotatingDiamonds*, *bubbles*, *concentricCircles* plus the instances *bonus1000* and *chaoSigleDep*. In these instances, all the target points have the same radius size while the overlap ratios are different but they are not specified.
- The second group contains the 21 instances named *d493*, *dsj1000*, *kroD100*, *lin318*, *pcb442*, *rat195*, and *rd400* that are derived from the TSPLIB. These instances have three fixed-radius scenarios, low overlap ratio (0.02), medium overlap ratio (0.1), and high overlap ratio (0.3).
- The last group of 14 instances is divided in two subgroups named *Team Random Radius Problems* and *TSPLIB Random Radius Problems*. In these instances, the size of the radii is chosen in a random way but assuring that $r_i \neq r_j$, $\forall i, j \in N$. To distinguish these instances from the other two groups, the suffix "*rdmRad*" is introduced in their names.

7.1. Upper Bound Comparisons

In this section, we verify the effectiveness and the performance of *(lb/ub)Alg* algorithm by comparing it with the best heuristics proposed by Mennell (2009), the Steiner Zone Variable Neighborhood Search heuristic (SZVNS) proposed by Wang et al. (2019) and the branch and bound algorithm introduced by Coutinho et al. (2016). We start by comparing the solutions of *(lb/ub)Alg* with the optimal solutions provided by branch and bound. This last algorithm runs for at most 4 hours and, within this time limit, it finds the optimal solution for all the instances of the first two sets.

In Table 1 we evaluate the effectiveness of *(lb/ub)Alg* and SZVNS algorithms by comparing their solutions with the optimal ones. Each row of the table contains average values computed on a test scenario, composed by 30 instances for CETSP-06,...,CETSP-20 and 10 instances for CETSP-25 and CETSP-30. Under the *Instance* heading, we report the scenario name while the next eight columns report the number of optimal solution found (*#Opt*), the average gap from the optimal solution (*AvgGap*), the maximum gap from the optimal solution (*MaxGap*), and the average computational time, in seconds (*AvgTime*),

[1] This dataset and the solutions found by *(lb/ub)Alg* are available here: https://github.com/CerroneCarmine/CETSP or, alternatively, upon request of the authors.

	(lb/ub)Alg				**SZVNS**			
Instance	**#Opt**	**AvgGap(%)**	**MaxGap(%)**	**AvgTime**	**#Opt**	**AvgGap(%)**	**MaxGap(%)**	**AvgTime**
				r=0.25				
CETSP-06	**30**	0.00	0.00	0.12	**30**	0.00	0.00	0.04
CETSP-08	**30**	0.00	0.00	0.17	**30**	0.00	0.00	0.04
CETSP-10	**30**	0.00	0.00	0.38	**30**	0.00	0.00	0.05
CETSP-12	**30**	0.00	0.00	0.65	**30**	0.00	0.00	0.07
CETSP-14	**29**	0.00	0.04	0.71	27	0.02	0.34	0.07
CETSP-16	**30**	0.00	0.00	1.31	27	0.07	0.86	0.09
CETSP-18	**30**	0.00	0.00	1.54	29	0.01	0.20	0.10
CETSP-20	**30**	0.00	0.00	2.54	29	0.00	0.07	0.14
CETSP-25	**10**	0.00	0.00	7.27	7	0.18	0.84	0.24
CETSP-30	**9**	0.01	0.06	17.05	6	0.41	1.74	0.41
				r=0.50				
CETSP-06	**29**	0.01	0.19	0.16	**29**	0.00	0.02	0.04
CETSP-08	**30**	0.00	0.00	0.35	29	0.00	0.01	0.04
CETSP-10	**30**	0.00	0.00	0.86	28	0.01	0.18	0.06
CETSP-12	**30**	0.00	0.00	1.67	29	0.00	0.08	0.07
CETSP-14	**30**	0.00	0.00	2.03	28	0.03	0.90	0.07
CETSP-16	**30**	0.00	0.00	3.43	27	0.08	1.10	0.10
CETSP-18	**29**	0.00	0.01	4.63	**29**	0.00	0.08	0.11
CETSP-20	**30**	0.00	0.00	6.07	28	0.01	0.32	0.15
CETSP-25	**9**	0.00	0.00	11.49	8	0.23	1.54	0.25
CETSP-30	**9**	0.06	0.57	20.30	8	0.02	0.19	0.33
				r=1.00				
CETSP-06	**30**	0.00	0.00	0.25	29	0.00	0.12	0.04
CETSP-08	**30**	0.00	0.00	0.73	**30**	0.00	0.00	0.05
CETSP-10	**30**	0.00	0.00	1.79	28	0.00	0.08	0.06
CETSP-12	**30**	0.00	0.00	3.03	28	0.03	0.89	0.07
CETSP-14	**30**	0.00	0.00	5.16	27	0.01	0.24	0.08
CETSP-16	**30**	0.00	0.00	5.67	28	0.01	0.15	0.10
CETSP-18	**29**	0.00	0.01	8.22	28	0.02	0.65	0.11
CETSP-20	**29**	0.00	0.05	8.33	23	0.03	0.32	0.14
CETSP-25	**10**	0.00	0.00	20.48	9	0.00	0.01	0.23
CETSP-30	**10**	0.00	0.00	18.36	9	0.04	0.44	0.32

Table 1 Comparison between the optimal solutions and the solutions found by (lb/ub)Alg on the CETSP instances.

of the *(lb/ub)Alg* and SZVNS algorithms, respectively. The gap values are computed by using the formula: $100 \times \frac{Alg-Opt}{Opt}$. In each row, the best #Opt value is reported in bold. The results of Table 1 highlight the effectiveness of *(lb/ub)Alg* on the CETSP instances. Indeed, by adding the #Opt values, we find out that *(lb/ub)Alg* finds the optimal solution on 772 out of 780 instances with an average gap that is always lower than 0.07%. On the contrary, SZVNS finds the optimal solution on 727 out of 780 instances and its average gap value is always lower than 0.42%. Moreover, the MaxGap values show that the solutions found by *(lb/ub)Alg* are very close to the optimal ones; excluding the worst case with 0.57%, on the remaining cases, this gap is always lower than 0.2%. In contrast, the MaxGap values of SZVNS are higher than 1.0% three times (1.74% in the worst case) and higher than 0.2% thirteen times. Regarding the computational time, SZVNS is extremely fast; it requires no more than a second. *(lb/ub)Alg* is always slower than SZVNS, but it never requires more than 21 seconds and its computational time is around 5 seconds, on average. Summarizing, on the CETSP instances, *(lb/ub)Alg* finds the optimal solution in 98.9% of the instances and, on the remaining instances, the average gap from the optimal solution is always lower than 0.07%.

To further investigate the effectiveness of *(lb/ub)Alg*, we compare its solutions with the optimal solutions provided by Coutinho et al. (2016) using branch and bound on the third set of instances. However, due to the size of these instances, the branch and bound approach finds the optimal solution only on the subset of 23 instances included in Table 2. Under the *Instance* and *Size* headings, we report the instance name and the instance size, respectively, while the next two columns report the optimal solution value (*Opt*) and the solution value (*(lb/ub)Alg*) found by our algorithm, respectively. Finally, under the *GAP* heading, we report the percentage gap between the solution values. This gap is computed by using the formula: $100 \times \frac{(lb/ub)Alg\ -Opt}{Opt}$. Finally, the last line of the table reports the average values of the GAP column. The solution values of *(lb/ub)Alg* are in bold whenever they coincide with the optimal solution values.

The results under the GAP heading show that *(lb/ub)Alg* is very effective because it finds the optimal solution on 19 out of 23 instances with an average percentage gap equal to 0.05%. On the remaining four instances, the gap from the optimal solution is at most equal to 0.75%.

The computational time of *(lb/ub)Alg* will be analyzed, in detail, in later tables. However, we want to highlight here that all the instances mentioned in Table 2 are solved in less than 200 seconds and, in the 80% of the cases, the time is less than 120 seconds. These results show the capacity of *(lb/ub)Alg* to quickly find the optimal solution on most instances of this subset.

Instance	**Size**	**Opt**	***(lb/ub)Alg***	**Gap(%)**
		Varied overlap ratios		
bubbles1	37	349.14	**349.14**	0.00
bubbles2	77	428.28	**428.28**	0.00
bubbles3	127	529.96	**529.96**	0.00
concentricCircles1	17	53.16	**53.16**	0.00
rotatingDiamonds1	21	32.39	**32.39**	0.00
rotatingDiamonds2	61	140.48	**140.48**	0.00
Team1_100	101	307.34	**307.34**	0.00
Team2_200	201	246.68	**246.68**	0.00
Team6_500	501	225.22	**225.22**	0.00
		Overlap ratios (0.1)		
d493	493	100.72	**100.72**	0.00
kroD100	100	89.67	**89.67**	0.00
lin318	318	1394.63	1405.07	0.75
rat195	195	67.99	68.14	0.22
		Overlap ratios (0.3)		
d493	493	69.76	69.79	0.05
dsj1000	1000	199.95	**199.95**	0.00
kroD100	100	58.54	**58.54**	0.00
lin318	318	765.96	**765.96**	0.00
pcb442	442	83.54	**83.54**	0.00
rat195	195	45.70	**45.70**	0.00
rd400	400	224.84	**224.84**	0.00
		Arbitrary radius		
rat195rdmRad	195	68.22	**68.22**	0.00
team1_100rdmRad	101	388.54	**388.54**	0.00
team3_300rdmRad	301	378.09	378.51	0.11
Avg				**0.05**

Table 2 Comparison between the optimal solutions and the solutions found by (lb/ub)Alg.

In Table 3 we compare the solutions found by *(lb/ub)Alg* with the ones found by SZVNS (Wang et al. (2019)) and by the best heuristics proposed by Mennell (2009), on the instances with a fixed radius. In the first column the identification number (*id*) associated to each instance is shown while in the second column (*Instance*) the name of the instance is provided. The next eight columns report the solution values of *(lb/ub)Alg*, $SZVNS$, HA, $SZ3_{0-360}$, $SZ3_{50-59}$, $SZ3$, $SZ2_{0-360}$, and $GTSP2$ heuristics, respectively. At the bottom, *#best* shows how many times each algorithm finds the best solution while *#U.best* reports how many times the algorithm is the only one to find the best solution. GAP reports the

id	Instance	Size	*(lb/ub)Alg*	SZVNS	HA	$SZ3_{0-360}$	$SZ3_{50-59}$	SZ3	$SZ2_{0-360}$	GTSP2
					Varied overlap ratios					
1	bonus1000	1001	**387.13**	403.06	408.44	404.01	408.39	414.42	416.82	578.87
2	bubbles1	37	**349.13**	349.14	349.14	**349.13**	**349.13**	**349.13**	**349.13**	**349.13**
3	bubbles2	77	**428.28**	**428.28**	**428.28**	**428.28**	**428.28**	**428.28**	**428.28**	432.38
4	bubbles3	127	**529.96**	532.21	532.28	536.62	536.62	548.89	564.04	546.73
5	bubbles4	185	**805.56**	825.33	832.27	836.54	842.73	836.54	844.35	877.47
6	bubbles5	251	**1061.64**	1073.43	1067.96	1073.71	1073.71	1095.24	1106.41	1131.09
7	bubbles6	325	1313.02	**1263.68**	1394.14	1382.41	1383.14	1404.16	1446.39	1446.26
8	bubbles7	407	1650.04	**1639.33**	1735.38	1720.21	1736.75	1720.21	1806.89	1775.92
9	bubbles8	497	2021.27	**1972.99**	2120.98	2117.49	2129.52	2158.78	2197.88	2236.40
10	bubbles9	595	2413.31	**2330.31**	2456.27	2481.23	2481.23	2550.53	2586.72	2748.93
11	chaoSingleDep	201	1039.61	1039.63	1042.82	**1022.88**	**1022.88**	**1022.88**	**1022.88**	1039.61
12	concentricCircles1	17	**53.16**	**53.16**	**53.16**	**53.16**	53.40	53.40	**53.16**	**53.16**
13	concentricCircles2	37	154.81	154.88	**153.13**	159.49	160.00	160.00	159.99	**153.13**
14	concentricCircles3	61	272.69	272.49	**271.08**	272.89	272.89	272.89	273.41	**271.08**
15	concentricCircles4	105	466.52	461.36	455.62	472.79	475.96	475.96	472.88	**454.46**
16	concentricCircles5	149	659.36	647.84	647.64	654.94	664.09	664.09	656.38	**645.38**
17	rotatingDiamonds1	21	**32.39**	**32.39**	**32.39**	**32.39**	**32.39**	33.15	**32.39**	**32.39**
18	rotatingDiamonds2	61	**140.48**	**140.48**	**140.48**	**140.48**	**140.48**	**140.48**	**140.48**	**140.48**
19	rotatingDiamonds3	181	**380.89**	**380.89**	382.50	381.27	381.48	381.48	382.05	382.17
20	rotatingDiamonds4	321	772.00	**770.68**	777.05	770.76	770.76	771.36	771.31	773.21
21	rotatingDiamonds5	681	1531.74	**1510.88**	1530.31	1511.44	1511.50	1511.50	1511.44	1517.70
22	Team1_100	101	**307.34**	**307.34**	**307.34**	**307.34**	**307.34**	308.73	309.18	307.79
23	Team2_200	201	**246.68**	246.69	247.48	246.74	246.84	250.55	247.57	249.92
24	Team3_300	301	476.43	**465.80**	466.12	466.24	478.28	468.65	483.26	484.70
25	Team4_400	401	702.69	698.05	686.76	688.63	698.42	688.63	711.93	**680.21**
26	Team5_499	500	708.45	703.38	711.14	**702.82**	704.48	710.02	715.99	703.20
27	Team6_500	501	**225.22**	226.18	227.50	**225.22**	225.23	225.92	225.88	355.02
					Overlap ratios (0.02)					
28	d493	493	205.39	205.74	203.84	**203.47**	205.66	207.44	207.82	204.71
29	dsj1000	1000	955.57	943.83	949.37	943.44	943.44	965.52	977.73	**935.74**
30	kroD100	100	160.09	**159.04**	159.05	160.81	160.81	160.96	162.17	159.05
31	lin318	318	2902.53	**2842.32**	2883.17	2863.37	2873.10	2872.37	2905.53	2867.46
32	pcb442	442	337.51	325.02	325.05	325.63	331.87	328.08	330.27	**323.03**
33	rat195	195	166.51	160.06	**158.79**	164.47	164.47	167.57	166.41	**158.79**
34	rd400	400	1085.75	1039.77	1041.77	1039.73	1043.58	1051.70	1050.42	**1033.42**
					Overlap ratios (0.1)					
35	d493	493	**100.72**	102.92	101.75	101.73	101.85	104.16	103.42	112.55
36	dsj1000	1000	**374.06**	393.06	380.59	377.10	376.35	379.97	385.74	482.85
37	kroD100	100	**89.67**	89.92	**89.67**	**89.67**	89.90	90.88	90.97	89.94
38	lin318	318	**1405.07**	1414.66	1410.25	1414.44	1425.05	1444.53	1441.99	1467.02
39	pcb442	442	**146.03**	152.73	148.74	148.88	151.26	152.98	153.55	147.24
40	rat195	195	**68.14**	68.32	68.24	68.26	68.85	69.48	69.77	68.26
41	rd400	400	**460.21**	474.78	469.19	469.75	482.28	500.42	498.52	473.70
					Overlap ratios (0.3)					
42	d493	493	69.79	69.90	70.40	**69.76**	**69.76**	**69.76**	**69.76**	82.83
43	dsj1000	1000	**199.95**	203.07	202.94	**199.95**	**199.95**	**199.95**	**199.95**	459.22
44	kroD100	100	**58.54**	**58.54**	59.03	**58.54**	**58.54**	**58.54**	**58.54**	62.15
45	lin318	318	**765.96**	766.16	770.08	**765.96**	**765.96**	**765.96**	**765.96**	946.67
46	pcb442	442	**83.54**	83.80	84.02	**83.54**	**83.54**	**83.54**	**83.54**	126.21
47	rat195	195	**45.70**	**45.70**	45.80	**45.70**	**45.70**	**45.70**	**45.70**	48.19
48	rd400	400	**224.84**	224.98	226.09	**224.84**	**224.84**	**224.84**	**224.84**	348.99
#best			**26**	**17**	**9**	**18**	**13**	**11**	**13**	**13**
#U.best			**11**	**9**	–	**2**	–	–	–	**6**
GAP			**0.98%**	**0.81%**	**1.34%**	**1.35%**	**1.78%**	**2.39%**	**2.89%**	**11.04%**
S.DEV			**1.48%**	**1.26%**	**2.13%**	**2.17%**	**2.37%**	**2.85%**	**3.51%**	**23.21%**

Table 3 Comparison of the upper bound values on the instances with a fixed radius.

average percentage gap from the best solution value while $S.DEV$ reports the standard deviation value. For each row, the best value is marked in bold.

The results of line #best show that *(lb/ub)Alg* is the most effective algorithm because it finds the best solution on 26 out of 48 instances. The #best value for the other heuristics is much lower; $SZVNS$ and $SZ3_{0-360}$ have values of 17 and 18, respectively. Moreover, the highest #U.best value is, again, associated with *(lb/ub)Alg*. In eleven instances, *(lb/ub)Alg* is the only algorithm to find the best solution. $SZVNS$ and $GTSP2$ have values of 9 and 6, respectively, in this row of Table 3. The other algorithms are not competitive.

With respect to the GAP row, $SZVNS$ and *(lb/ub)Alg* are the only algorithms to have average gaps of less than 1%. In the final row, we see that $SZVNS$ and *(lb/ub)Alg* are the only algorithms to have standard deviations of less than 1.5%. Clearly, $SZVNS$ and *(lb/ub)Alg* are the best performing of the eight algorithms.

It is worth noting that the overlap ratio of the instances has a major impact on the effectiveness of *(lb/ub)Alg* and $SZVNS$. Based on the logic of *(lb/ub)Alg*, we expect that, as the overlap ratio increases, the effectiveness of *(lb/ub)Alg* will improve, and this is exactly what happens. When we compare *(lb/ub)Alg* and $SZVNS$ on the seven instances with an overlap ratio of 0.02, we observe that $SZVNS$ obtains a better solution in six of these cases. However, on the instances with overlap ratios of 0.1 and 0.3, *(lb/ub)Alg* obtains better solutions than $SZVNS$ in 12 of 14 instances (with two ties). It is interesting to observe that the SZ heuristics of Mennell (2009) are particularly effective on the instances with overlap ratio equal to 0.3 where they always find the best solution. Over the 27 instances with varied overlap ratios, *(lb/ub)Alg* outperforms $SZVNS$ in 15 cases. These results nicely demonstrate that the choice of which algorithm to use (*(lb/ub)Alg* or $SZVNS$) strongly depends on the overlap ratio of the specific instance being studied.

The computational times of the algorithms are reported in Table 4. For each row, the computational time of the fastest algorithm is marked in bold. The bottom row displays the average computation time of each heuristic on the instances with a fixed radius. For the two best performing heuristics from Table 3, we observe that $SZVNS$ has an average time of about 90 seconds, whereas *(lb/ub)Alg* has an average time of 200 seconds. $SZ3_{50-59}$ and $SZ3$ are faster than *(lb/ub)Alg* but have percentage gaps (from optimality) greater than that of *(lb/ub)Alg*. The remaining heuristics have average running times at least

id	Instance	Size	*(lb/ub)Alg*	SZVNS	HA	$SZ3_{0-360}$	$SZ3_{50-59}$	SZ3	$SZ2_{0-360}$	GTSP2
					Varied overlap ratios					
1	bonus1000	1001	116.63	1109.87	1508.10	29067.30	807.43	**82.58**	2885.86	2648762.00
2	bubbles1	37	11.42	**0.24**	55.40	226.03	6.28	0.45	44.93	95.00
3	bubbles2	77	22.68	**1.09**	97.50	733.22	20.37	4.30	136.24	64.00
4	bubbles3	127	130.17	**2.51**	146.20	2121.30	58.93	20.41	359.55	471.00
5	bubbles4	185	130.11	**9.53**	202.90	2885.31	80.15	28.16	462.21	23414.00
6	bubbles5	251	116.63	37.37	234.90	4435.84	123.22	**21.24**	486.77	6085.00
7	bubbles6	325	217.20	**14.00**	284.90	5439.70	151.10	24.96	617.77	24657.00
8	bubbles7	407	304.46	50.29	521.00	8814.55	244.85	**36.61**	1452.30	34295.00
9	bubbles8	497	416.30	110.30	412.50	12151.05	337.53	**55.41**	2009.94	100462.00
10	bubbles9	595	296.14	168.18	415.50	17111.39	475.32	**69.10**	2788.71	304348.00
11	chaoSingleDep	201	89.69	12.67	201.40	2503.34	69.54	**5.42**	290.08	43743.00
12	concentricCircles1	17	6.43	**0.10**	18.50	130.08	3.61	0.36	24.29	123.00
13	concentricCircles2	37	51.96	**0.27**	40.10	235.16	6.53	0.65	51.72	169.00
14	concentricCircles3	61	362.24	**1.28**	57.60	451.66	12.55	2.16	71.44	372.00
15	concentricCircles4	105	96.86	5.82	105.70	752.92	20.92	**2.05**	122.30	4468.00
16	concentricCircles5	149	351.21	10.06	141.20	1163.94	32.33	**3.20**	218.90	22031.00
17	rotatingDiamonds1	21	7.06	**0.12**	16.60	166.98	4.64	0.49	17.65	24.00
18	rotatingDiamonds2	61	176.13	**0.54**	58.90	508.92	14.14	2.31	75.51	368.00
19	rotatingDiamonds3	181	615.47	8.81	192.10	1568.31	43.56	**5.02**	234.89	16407.00
20	rotatingDiamonds4	321	271.20	18.49	257.10	3064.52	85.13	**9.22**	570.13	49045.00
21	rotatingDiamonds5	681	210.99	53.62	765.40	10841.58	301.16	**27.56**	2058.39	210899.00
22	Team1_100	101	76.86	**3.20**	122.60	1292.77	35.91	5.83	178.45	913.00
23	Team2_200	201	36.06	16.21	336.20	2117.86	58.83	**9.17**	286.69	34768.00
24	Team3_300	301	56.88	31.48	366.20	3389.92	94.17	**14.95**	553.45	69823.00
25	Team4_400	401	120.25	43.47	474.80	5335.48	148.21	**32.24**	737.30	276264.00
26	Team5_499	500	828.33	88.17	447.20	7537.02	209.36	**42.69**	1426.78	396631.00
27	Team6_500	501	**8.15**	719.32	819.90	9355.44	259.87	19.44	873.53	196027.00
					Overlap ratios (0.02)					
28	d493	493	247.04	69.72	435.90	10061.84	279.50	**35.73**	1533.77	309071.00
29	dsj1000	1000	998.99	151.77	841.50	26666.84	740.75	**71.27**	3987.42	12682545.00
30	kroD100	100	167.14	9.34	85.10	351.25	9.76	**2.22**	153.41	666.00
31	lin318	318	292.90	31.17	245.00	3744.98	104.03	**18.02**	707.70	48545.00
32	pcb442	442	805.44	52.88	381.40	3932.38	109.23	**23.72**	1214.61	222047.00
33	rat195	195	569.55	11.16	162.30	742.17	20.62	**4.22**	304.14	7781.00
34	rd400	400	662.60	27.70	308.00	2669.81	74.16	**15.25**	927.03	76010.00
					Overlap ratios (0.1)					
35	d493	493	33.87	142.94	689.00	8217.20	228.26	**22.88**	992.23	61715.00
36	dsj1000	1000	83.00	356.79	1372.90	24126.11	670.17	**65.02**	2939.34	7807747.00
37	kroD100	100	193.44	**1.99**	125.30	817.63	22.71	3.20	114.58	1483.00
38	lin318	318	26.50	65.98	444.50	3895.64	108.21	**17.06**	474.13	60252.00
39	pcb442	442	224.95	54.99	653.00	5903.30	163.98	**17.83**	746.81	250485.00
40	rat195	195	95.19	9.47	269.30	1967.56	54.66	**6.81**	220.39	5951.00
41	rd400	400	101.27	96.98	560.80	4938.53	137.18	**20.27**	626.05	91003.00
					Overlap ratios (0.3)					
42	d493	493	**2.72**	58.07	983.40	9295.19	258.20	26.02	654.09	59109.00
43	dsj1000	1000	**6.89**	316.41	1766.60	25575.72	710.44	35.28	2042.81	2623498.00
44	kroD100	100	**0.37**	4.58	189.80	681.45	18.93	1.39	101.39	865.00
45	lin318	318	**1.21**	49.01	589.80	4334.06	120.39	6.48	349.50	40166.00
46	pcb442	442	**0.58**	196.54	846.70	8661.81	240.61	37.63	631.64	76137.00
47	rat195	195	**0.33**	19.01	371.20	5872.27	163.12	11.44	236.50	6323.00
48	rd400	400	**4.19**	74.86	743.90	6563.88	182.33	14.19	602.11	61120.00
AVG			**200.95**	**89.97**	**424.29**	**6092.11**	**169.23**	**20.46**	**804.07**	**603275.98**

Table 4 Computational time (in seconds) on the instances with fixed radius.

twice as large as for *(lb/ub)Alg*. $SZ3_{0-360}$ and $GTSP2$ have average running times that are excessive.

It is interesting to observe how the running times of *(lb/ub)Alg* change as a function of the overlap ratio. In general, as this ratio increases, running times decrease. On instance 28-34, with an overlap ratio of 0.02, the running times range from 167 to 998 seconds. On instances 35-41, with an overlap ratio of 0.1, running times range from 33 to 224 seconds. On average, the instances with an overlap ratio of 0.02 take nearly five times as long to solve as the instances with an overlap ratio of 0.1. Finally, on the instances with an overlap ratio of 0.3, the running times of *(lb/ub)Alg* are very small – no more than 7 seconds. When we look at the varied overlap ratio instances, *(lb/ub)Alg*'s running times range from a few seconds to 828 seconds. In only two of the 27 instances do running times exceed 420 seconds. In comparing the running times of *(lb/ub)Alg* and $SZVNS$, we observe that *(lb/ub)Alg* is always faster than $SZVNS$ on the instances with overlap ratio 0.3 and it is always slower on the instances with overlap ratio 0.02. It, therefore, seems that *(lb/ub)Alg* is the preferred heuristic on instances with a higher overlap ratio, whereas $SZVNS$ is better for a lower overlap ratio.

The computational results on the instances with arbitrary radius are reported in Table 5. The comparison is carried out between *(lb/ub)Alg*, $SZVNS$, and the best heuristics proposed by Mennell (2009). Some of these are different from the algorithms compared in Table 3 because we selected the heuristics of Mennell (2009) having the best overall gap according to the instances reported in each table. The last four rows of this table and Table 3 are similar. For each row, the best value is marked in bold.

Table 5 reveals that both *(lb/ub)Alg* and $SZVNS$ are very effective. They find the best solutions in 7 and 8 out of 14 instances. The number of unique solutions found by *(lb/ub)Alg* and $SZVNS$ is 5 and 6, respectively. The best average gaps are achieved by $SZVNS$ (0.30%) and *(lb/ub)Alg* (0.66%). The lowest standard deviations are also achieved by $SZVNS$ (0.52%) and *(lb/ub)Alg* (1.23%). The other algorithms represented in this table are not competitive.

The computational times of the heuristics from Table 5 are shown in Table 6. For each row, the computational time of the fastest algorithm is marked in bold. The fastest algorithm is $SZVNS$ with an average running time of 12 seconds, followed by $SZ2$ with an average of 28 seconds. The average time of *(lb/ub)Alg* is 270 seconds, while HA requires

id	Instance	Size	*(lb/ub)Alg*	SZVNS	HA	HYBRID2	HYBRID1	SZ2	$SZ3_{50-59}$	GTSP1
				Arbitrary radius						
49	bonus1000rdmRad	1001	955.41	**938.27**	1001.1	996.31	996.97	992.61	1015.52	1087.47
50	d493rdmRad	493	**134.74**	135.02	141.34	140.12	140.69	142.08	153.29	175.95
51	dsj1000rdmRad	1000	625.92	**625.25**	659.64	655.95	660.39	653.37	715.97	901.09
52	kroD100rdmRad	100	142.36	**141.83**	141.84	144.27	151.52	144.39	145	142.2
53	lin318rdmRad	318	**2055.77**	2082.25	2079.49	2160.85	2164.45	2175.72	2136.45	2165.26
54	pcb442rdmRad	442	**220.44**	221.16	234.15	235.19	237.32	243.48	237.83	267.03
55	rat195rdmRad	195	**68.22**	**68.22**	**68.22**	68.81	68.84	68.66	68.27	129.71
56	rd400rdmRad	400	1305.46	1257.73	**1252.22**	1252.38	1260.77	1276.08	1270.04	1258.15
57	team1_100rdmRad	101	**388.54**	**388.54**	390.23	390.34	390.78	390.95	389.1	392.55
58	team2_200rdmRad	201	**616.82**	626.9	624.79	642.81	643.12	655.81	644.57	627.45
59	team3_300rdmRad	301	**378.51**	379.84	382.16	399.39	400.02	396.61	381.83	496.78
60	team4_400rdmRad	401	1025.76	**1006.71**	1020.16	1026.32	1028.89	1025.83	1011.77	1016
61	team5_499rdmRad	500	446.51	**446.19**	458.35	456.39	457.88	455.61	476.19	631.27
62	team6_500rdmRad	501	626.18	**621.99**	672.36	666.15	666.64	678.03	679.67	755.79
#best			**7**	**8**	**2**	–	–	–	–	–
#U.best			**5**	**6**	**1**	–	–	–	–	–
GAP			**0.66%**	**0.30%**	**2.81%**	**3.64%**	**4.30%**	**4.37%**	**5.29%**	**21.85%**
S.DEV			**1.23%**	**0.52%**	**2.86%**	**2.40%**	**2.48%**	**3.03%**	**4.91%**	**25.14%**

Table 5 Comparison of the upper bound values on the instances with an arbitrary radius.

577 seconds, on average. The remaining heuristics require more than 1000 seconds, on average.

7.2. Lower Bound Comparisons

In this section, we evaluate the quality of the lower bounds computed by *(lb/ub)Alg*. Since no additional computational effort is required by *(lb/ub)Alg*, the running times are the same as shown in Tables 4 and 6.

In Table 7, we report the lower bounds computed by *(lb/ub)Alg*, and the branch and bound (BB) method proposed by Coutinho et al. (2016). The first two columns show the id and instance name. The next four columns show the lower bound (LB) and running time (in seconds) for *(lb/ub)Alg* and BB, respectively. For each row, the best lower bound is marked in bold. In the last row of the table, the average gap percentage (GAP) between the best lower bound and the lower bound computed by the algorithm is presented. It is quite clear that the best lower bounds are, almost always, computed by BB. However, it is often the case that BB requires 14400 seconds (its time limit) to obtain these results. The GAP value of *(lb/ub)Alg* is significantly higher that for BB (12.53% vs 0.15%), but the

id	Instance	Size	*(lb/ub)Alg*	SZVNS	HA	HYBRID2	HYBRID1	SZ2	$SZ3_{50-59}$	GTSP1
				Arbitrary radius						
49	bonus1000rdmRad	1001	728.55	**3.93**	1043.70	2080.28	1759.89	41.20	113452.34	10101278.00
50	d493rdmRad	493	95.34	**0.26**	770.70	46.22	20.88	7.42	25737.78	484757.00
51	dsj1000rdmRad	1000	286.23	**2.73**	1594.60	234.14	150.89	24.58	196446.44	8819121.00
52	kroD100rdmRad	100	46.25	87.16	110.70	86.61	83.89	**4.27**	295.91	1006.00
53	lin318rdmRad	318	59.52	**3.18**	483.00	204.72	191.41	8.81	9896.97	193723.00
54	pcb442rdmRad	442	153.38	**1.36**	620.80	1160.75	770.03	73.28	13067.88	492575.00
55	rat195rdmRad	195	9.97	10.57	427.10	44.59	38.63	**8.00**	1655.23	9213.00
56	rd400rdmRad	400	1110.35	**0.67**	277.30	12795.41	9499.78	57.25	2573.50	350351.00
57	team1_100rdmRad	101	20.80	**3.89**	129.70	51.00	43.36	9.30	403.59	413.00
58	team2_200rdmRad	201	209.44	**0.93**	211.80	350.84	341.20	6.92	1300.39	7448.00
59	team3_300rdmRad	301	16.65	34.51	488.00	56.39	48.06	**5.00**	7949.72	27571.00
60	team4_400rdmRad	401	658.03	**2.62**	348.50	1794.55	1514.84	22.20	5175.19	574193.00
61	team5_499rdmRad	500	51.43	**5.05**	874.90	52.19	35.91	124.42	38920.25	811557.00
62	team6_500rdmRad	501	347.08	19.77	701.20	244.80	212.84	**11.47**	20166.17	535527.00
AVG			**270.93**	**12.62**	**577.29**	**1371.61**	**1050.83**	**28.87**	**31217.24**	**1600623.79**

Table 6 Computational time (in seconds) on the instances with arbitrary radius.

required running times for *(lb/ub)Alg* are much smaller. For this reason, the lower bound computed by *(lb/ub)Alg*, even if less effective, could be used within a branch-and-bound approach to reduce the size of the search tree; BB requires too much computational effort to be used in this context.

Along these lines, we carried out some additional experiments to examine the effectiveness of *(lb/ub)Alg* as a function of the number of iterations (this number was set to 30 in Section 6). If we use a single iteration rather than 30, *(lb/ub)Alg* produces an overall GAP value of 14.82% with running times that are always less than 3 seconds. This represents a large reduction in running time for a slight deterioration in lower bound quality. This observation makes the idea of applying *(lb/ub)Alg* within a branch-and-bound algorithm very attractive.

In analyzing the results of Table 7, we observe that the worst results for *(lb/ub)Alg* are obtained on the instances with varied overlap ratio (see GAP1). We observe a reduction in the average gap as the overlap ratio increases, but this is not strict (e.g., see GAP3). When the overlap ratio is equal to 0.3, the average gap is only 2.83%. The situation changes dramatically on the instances with an arbitrary radius, as illustrated in Table 8.

There are 14 instances compared in Table 8. Here, the lower bounds obtained by *(lb/ub)Alg* are better, on average, than the lower bounds found by BB (4.12% vs. 7.24%).

			(lb/ub) Alg		**BB**	
id	**Instance**	**Size**	**LB**	**Time**	**LB**	**Time**
			Varied overlap ratios			
1	bonus1000	1001	285.32	116.63	**359.38**	14400.02
2	bubbles1	37	323.55	11.42	**349.14**	0.1
3	bubbles2	77	391.64	22.68	**428.28**	0.22
4	bubbles3	127	476.75	130.17	**529.96**	193.12
5	bubbles4	185	561.98	130.11	**690.58**	14400.01
6	bubbles5	251	658.52	116.63	**851.82**	14400.28
7	bubbles6	325	766.44	217.2	**993.98**	14400.15
8	bubbles7	407	866.14	304.46	**1123.52**	14400.19
9	bubbles8	497	965.7	416.3	**1252.72**	14400.28
10	bubbles9	595	1073.1	296.14	**1374.41**	14400.33
11	chaoSingleDep	201	831.06	89.69	**1000.15**	14400.09
12	concentricCircles1	17	45.84	6.43	**53.16**	5.18
13	concentricCircles2	37	114.56	51.96	**149.87**	14400.03
14	concentricCircles3	61	191.5	362.24	**247.62**	14400.18
15	concentricCircles4	105	289.36	96.86	**358.89**	14400.06
16	concentricCircles5	149	392.89	351.21	**459.41**	14400.16
17	rotatingDiamonds1	21	30.73	7.06	**32.39**	0.09
18	rotatingDiamonds2	61	111.44	176.13	**140.48**	730.37
19	rotatingDiamonds3	181	272.76	615.47	**348.61**	14400.07
20	rotatingDiamonds4	321	527.85	271.2	**593.35**	14400.03
21	rotatingDiamonds5	681	1098.37	210.99	**1106.58**	14400.23
22	Team1_100	101	270.49	76.86	**307.34**	9.61
23	Team2_200	201	232.87	36.06	**246.68**	0.72
24	Team3_300	301	330.92	56.88	**447.53**	14400.03
25	Team4_400	401	391.45	120.25	**507.3**	14400.1
26	Team5_499	500	481.49	828.33	**524.59**	14400.05
27	Team6_500	501	217.74	8.15	**225.22**	0.43
GAP1			*15.80%*		*0.00%*	
			Overlap ratios (0.02)			
28	d493	493	129.05	247.04	**146.33**	14400.17
29	dsj1000	1000	521.52	998.99	**559.11**	14400.24
30	kroD100	100	118.64	167.14	**142.87**	14400.16
31	lin318	318	1830.97	292.9	**1990.9**	14400.08
32	pcb442	442	177.36	805.44	**185.85**	14400.03
33	rat195	195	93.72	569.55	**108.1**	14400.04
34	rd400	400	**609.79**	662.6	567.19	14440.27
GAP2			*8.77%*		*1.00%*	
			Overlap ratios (0.1)			
35	d493	493	91.9	33.87	**100.72**	53.28
36	dsj1000	1000	317.33	83	**373.73**	14400.51
37	kroD100	100	85.39	193.44	**89.67**	1.86
38	lin318	318	1139.23	26.5	**1394.63**	8541.19
39	pcb442	442	110.99	224.95	**137.45**	14400.09
40	rat195	195	65.41	95.19	**67.99**	17.32
41	rd400	400	329.82	101.27	**432.8**	14400.03
GAP3			*13.40%*		*0.00%*	
			Overlap ratios (0.3)			
42	d493	493	68.64	2.72	**69.76**	0.32
43	dsj1000	1000	193.5	6.89	**199.95**	0.75
44	kroD100	100	56.89	0.37	**58.54**	0.07
45	lin318	318	754.21	1.21	**765.96**	0.24
46	pcb442	442	81.88	0.58	**83.54**	0.31
47	rat195	195	44.51	0.33	**45.7**	0.13
48	rd400	400	219.29	4.19	**224.84**	0.33
GAP4			*2.83%*		*0.00%*	
Overall GAP			**12.53%**		**0.15%**	

Table 7 Comparison of the lower bound values on the instances with a fixed radius.

			(lb/ub)Alg		BB	
id	**Instance**	**Size**	**LB**	**Time**	**LB**	**Time**
		Arbitrary radius				
49	bonus1000rdmRad	1001	**700.54**	728.55	506.13	14400.22
50	d493rdmRad	493	116.57	95.34	**125.31**	14400.14
51	dsj1000rdmRad	1000	**545.38**	286.23	509.74	14400.3
52	kroD100rdmRad	100	119.46	46.25	**136.62**	14400.12
53	lin318rdmRad	318	1719.54	59.52	**1807.68**	14400
54	pcb442rdmRad	442	**181.21**	153.38	175.83	14400.23
55	rat195rdmRad	195	65.12	9.97	**68.22**	5.16
56	rd400rdmRad	400	**880.91**	1110.35	571.48	14400.29
57	team1_100rdmRad	101	350.78	20.8	**388.54**	269.31
58	team2_200rdmRad	201	482.1	209.44	**488.18**	14400.12
59	team3_300rdmRad	301	342.65	16.65	**378.09**	682.39
60	team4_400rdmRad	401	**737.16**	658.03	549.91	14400.32
61	team5_499rdmRad	500	405.46	51.43	**442.64**	14400.36
62	team6_500rdmRad	501	**507.43**	347.08	489.61	14400.08
GAP			**4.12%**		**7.24%**	

Table 8 **Comparison of the lower bound values on the instances with an arbitrary radius.**

On some instances, BB performs very poorly. In addition, *(lb/ub)Alg* is much faster. These key observations are probably connected. We assume that the disappointing results of BB are due to the fact that instances with arbitrary radius are more difficult for the BB algorithm to solve, as evidenced by the running times (which often reach the time limit of 14400 seconds). On the other hand, *(lb/ub)Alg* never requires more than 1110 seconds. In 11 of the 14 instances, running times for *(lb/ub)Alg* are under 350 seconds.

8. Conclusions

In this paper, we developed a new metaheuristic approach, *(lb/ub)Alg*, to solve the Close-Enough Traveling Salesman Problem; the method computes both upper and lower bounds for the CETSP. It works by discretizing neighborhoods around target points in order to minimize the total discretization error. The Carousel Greedy algorithm is used to select and to add, one by one, neighborhoods to the current partial solution until a feasible solution is obtained. We tested the performance of *(lb/ub)Alg* on the key benchmark instances with respect to accuracy and running time. The computational results show that *(lb/ub)Alg* finds the optimal solution in 98.9% of the CETSP instances and, on the remaining instances, the

$\ell(v_i, v_j)$	Euclidean distance between v_i and v_j
C_v	Circumference with center v and radius r_v associated with target point v
$\overline{a,b}$	Chord between the points a and b of circumference
$\widehat{a,b}$	Circular arc from a to b in the clockwise direction
N_v	Set of point within and on C_v
$\hat{N}_v$	Discretized neighborhood of $N(v)$
T^*	Shortest tour intersecting every neighborhood $N(v)$
$\hat{T}^*$	shortest tour intersecting every neighborhood $\hat{N}(v)$
$w(T)$	cost of tour T. It is given by the sum of the edge lengths in T
$\xi(v)$	Discretization error occurring on $\hat{N}(v)$
$\xi_k(v)$	Discretization error occurring on $\hat{N}(v)$ by using k discretization points
lbc	Lower bound value computed by *(lb/ub)Alg*
ubc	Upper bound value computed by *(lb/ub)Alg*
N_s	Subset of target points selected by the lbc procedure
$\hat{G}_s$	Subgraphs induced by discretization points used to discretize the neighborhoods of N_s
$\hat{T}^*_s$	Shortest tour of $\hat{G}_s$
$nMax$	total number of available discretization points
$val(v)$	Score assigned to the vertex v by the FindNextPoint procedure
μ	Weighted average computed by FindNextPoint and used to compute $val(v)$
Δ	Parameter used to increase or decrease the μ in the computation of $val(v)$
Γ_v	"Recommendation value" associated with each vertex v. It influences the selection of vertices carried out by FindNextPoint
k	estimate of the number of discretization points used to discretize a neighborhood.
R	The sequence of "recommended" vertices used by *(lb/ub)Alg* denoted by $\langle v_1, v_2, \ldots, v_m \rangle$
$max\Delta$	*(lb/ub)Alg* stops when $\Delta > max\Delta$
pos_v	position of the vertex v in the sequence R

Table 9 Glossary of the terms used in the paper.

average gap from the optimal solution is almost always lower than 0.07%. On Mennell's instances, where the optimal solution is known, *(lb/ub)Alg* obtains this solution in 83% of the cases and, in the remaining cases, the deviation from optimality is 0.28%, on average. On the instances with the highest overlap ratio, *(lb/ub)Alg* is the fastest algorithm and, in all cases except one, it finds the best solution. When the overlap ratio is small, $SZVNS$ is better and faster than *(lb/ub)Alg*. Finally, on the instances with arbitrary radius, *(lb/ub)Alg* finds better lower bounds than the branch and bound approach in much less time.

References

Andersen ED, Roos C, Terlaky T (2003) On implementing a primal-dual interior-point method for conic quadratic optimization. *Mathematical Programming* 95(2):249–277.

Arkin EM, Hassin R (1994) Approximation algorithms for the geometric covering salesman problem. *Discrete Applied Mathematics* 55(3):197–218.

Behdani B, Smith J (2014) An integer-programming-based approach to the close-enough traveling salesman problem. *INFORMS Journal on Computing* 26(3):415–432.

Carrabs F, Cerrone C, Cerulli R, D'Ambrosio C (2017a) Improved upper and lower bounds for the close enough traveling salesman problem. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10232 LNCS:165–177.

Carrabs F, Cerrone C, Cerulli R, Gaudioso M (2017b) A novel discretization scheme for the close enough traveling salesman problem. *Computers & Operations Research* 78:163–171.

Cerrone C, Cerulli R, Golden B (2017) Carousel greedy: A generalized greedy algorithm with applications in optimization. *Computers & Operations Research* 85:97–112.

Coutinho W, Do Nascimento R, Pessoa A, Subramanian A (2016) A branch-and-bound algorithm for the close-enough traveling salesman problem. *INFORMS Journal on Computing* 28(4):752–765.

Current JR (1981) *Multiobjective Design of Transportation Networks.* Ph.D. thesis, Department of Geography and Environmental Engineering. The Johns Hopkins University.

Current JR, Schilling DA (1989) The covering salesman problem. *Transportation Science* 23(3):208–213.

Dong J, Yang N, Chen M (2007) Heuristic approaches for a tsp variant: The automatic meter reading shortest tour problem. *Operations Research/Computer Science Interfaces Series* 37:145–163.

Dumitrescu A, Mitchell J (2003) Approximation algorithms for tsp with neighborhoods in the plane. *Journal of Algorithms* 48(1):135–159.

Faigl J (2018) GSOA: Growing self-organizing array - unsupervised learning for the close-enough traveling salesman problem and other routing problems. *Neurocomputing* 312:120–134.

Faigl J, Váňa P, Deckerová J (2019) Fast heuristics for the 3-d multi-goal path planning based on the generalized traveling salesman problem with neighborhoods. *IEEE Robotics and Automation Letters* 4(3):2439–2446.

Gendreau M, Laporte G, Semet F (1997) The covering tour problem. *Operations Research* 45:568–576.

Gulczynski D, Heath J, Price C (2006) Close enough traveling salesman problem: A discussion of several heuristics. *Perspectives in Operations Research*, volume 36 of *Operations Research/Computer Science Interfaces Series*, 271–283 (Springer US).

Mata CS, Mitchell JSB (1995) Approximation algorithms for geometric tour and network design problems (extended abstract). *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, 360–369.

Mennell W (2009) *Heuristics for solving three routing problems: Close-enough traveling salesman problem, close-enough vehicle routing problem, sequence-dependent team orienteering problem.* Ph.D. thesis, The Robert H. Smith School of Business, University of Maryland, College Park.

Mennell W, Golden B, Wasil E (2011) A steiner-zone heuristic for solving the close-enough traveling salesman problem. *2th INFORMS Computing Society Conference: Operations Research, Computing, and Homeland Defense.*

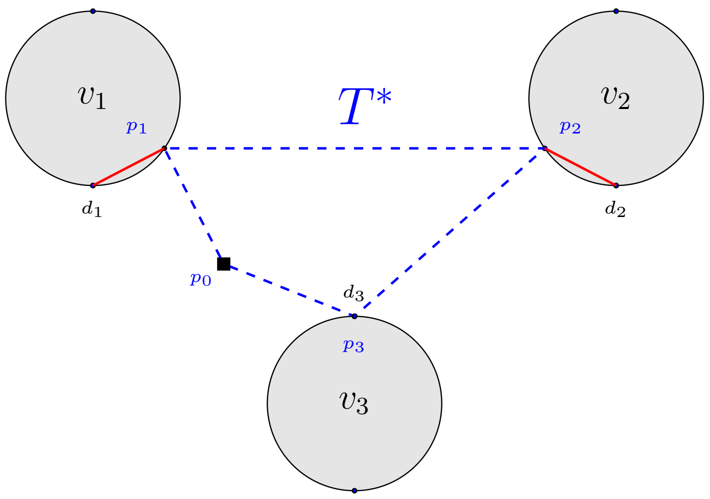

Figure 8 The computation of lb_1.

Poikonen S, Wang X, Golden B (2017) The vehicle routing problem with drones: Extended models and connections. *Networks* 70(1):34–43.

Shuttleworth R, Golden B, Smith S, Wasil E (2008) Advances in meter reading: Heuristic solution of the close enough traveling salesman problem over a street network. *Operations Research/ Computer Science Interfaces Series* 43:487–501.

Silberholz J, Golden B (2007) The generalized traveling salesman problem: a new genetic algorithm approach. *In "Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies"* 37:165 – 181.

Wang X, Golden B, Wasil E (2019) A Steiner zone variable neighborhood search heuristic for the close-enough traveling salesman problem. *Computers & Operations Research* 101(1):200–219.

Yang Z, Xiao MQ, Ge YW, Feng DL, Zhang L, Song HF, Tang XL (2018) A double-loop hybrid algorithm for the traveling salesman problem with arbitrary neighbourhoods. *European Journal of Operational Research* 265(1):65–80.

Yuan B, Orlowska M, Sadiq S (2007) On the optimal robot routing problem in wireless sensor networks. *IEEE Transactions on Knowledge and Data Engineering* 19(9):1252–1261.

Appendix

In this section, we describe how the lower bound lb_1 is derived. Let us consider the example in Figure 8 where $N = \{v_1, v_2, v_3\}$ and T^* is the optimal tour for the CETSP, identified by the turn points p_1, p_2, p_3 and the depot p_0. Note that the turn points of T^* are always on the boundary of the spheres (see Proposition 1 in Behdani and Smith (2014)). Each neighborhood is discretized by using only $k = 2$ discretization points placed on the corresponding circumference.

Let us build now the *walk* $Q = \{p_0, p_1, d_1, p_1, p_2, d_2, p_2, p_3, p_0\}$. In practice, Q is built by following the edges of T^* and, for each turn point $p_i \in C_{v_i}$, the closest discretization point $d_i \in \hat{N}(v_i)$ is detected and the chord $\overline{p_i, d_i}$ is crossed twice. The discretization error $\xi(v_i)$ is equal to two times the length of $\overline{p_i, d_i}$. Thus $\xi(v_i)$ represents the error in $\hat{N}(v_i)$ with respect to T^*, due to the choice of the discretization points. It is easy to see that:

$$w(Q) = w(T^*) + \sum_{v \in N} \xi(v).$$

Since Q starts and ends at the depot p_0 and visits one discretization point for each neighborhood, then $w(\hat{T}^*) \leq w(Q)$. This means that $w(\hat{T}^*) \leq w(T^*) + \sum_{v \in N} \xi(v)$ and then $lb_1 = w(\hat{T}^*) - \sum_{v \in N} \xi(v) \leq w(T^*)$.