# A Selector Operator-Based Adaptive Large Neighborhood Search for the Covering Tour Problem

Leticia Vargas, Nicolas Jozefowiez, Sandra Ulrich Ngueveu

# A Selector Operator-Based Adaptive Large Neighborhood Search for the Covering Tour Problem

Leticia Vargas[1,2], Nicolas Jozefowiez[1,2], and Sandra Ulrich Ngueveu[1,3]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
[3] Univ de Toulouse, INP, LAAS, F-31400 Toulouse, France

**Abstract.** The Covering Tour Problem finds application in distribution network design. It includes two types of vertices: the covering ones and the ones to be covered. This problem is about identifying a lowest-cost Hamiltonian cycle over a subset of the covering vertices in such a way that every element not of this type is covered. In this case, a vertex is considered covered when it is located within a given distance from a vertex in the tour. This paper presents a solution procedure based on a *Selector* operator that allows to convert a giant tour into an optimal CTP solution. This operator is embedded in an adaptive large neighborhood search. The method is competitive as shown by the quality of results evaluated using the output of a state-of-the-art exact algorithm.

**Keywords:** Covering Tour Problem, *Split* procedure, ALNS algorithm

## 1 Introduction

This study aims at solving a tour location problem (TLP), namely the *Covering Tour Problem* (CTP) through a new splitting operator, *Selector*, embedded into an *Adaptive Large Neighborhood Search* (ALNS) metaheuristic. The overall goal of TLPs is to construct an optimal tour through a *subset* of the vertices of a network, subject to a set of constraints. They differ from classical vehicle routing problems since the assumption shared by problems of the TSP and VRP family is that *all* vertices of the network should be served, something which is not valid in many real applications. In TLPs, the visits are optional. The new *Selector* operator finds a minimum-cost tour (subject to a given sequence) which passes through a subset of vertices and meets side constraints. In the case of the CTP, every vertex not in the elementary cycle must lie within a prespecified radius from at least one vertex in the cycle.

The CTP is a generalization of the *Traveling Salesman Problem* (TSP) and it can be formally described as follows. Let $G = (N, E)$ be an undirected graph, where $N = V \cup W$ represents the vertex set and $E = \{(v_i, v_j) | v_i, v_j \in N, i < j\}$ is the edge set. $V$ is the subset of $n$ vertices that *can* be visited at most once, $T \subseteq V$ is the subset of vertices that *must* be visited exactly once, while $W$ is

the subset of vertices that must be *covered*. Vertex $v_0 \in T$ is the depot. Let $d_{ij}$ be the distance associated with edge $(i, j) \in E$, and $D = (d_{ij})$ the distance matrix that satisfies the triangle inequality. The solution of the CTP is to find a minimum-length elementary cycle in $V$ such that each vertex $w_i \in W$ is covered by the cycle, and also all vertices in T are found in the cycle. A vertex $w_i \in W$ is covered if there exists at least one vertex $v_j \in V$ in the cycle for which $d_{ij} \leqslant c$, where $c$ is known as the *covering distance*. Figure 1 shows a feasible CTP tour for an instance where $|V| = 8$, $|T| = 2$ and $|W| = 17$, and exemplifies how vertex $v_A \in V$ covers vertices $\{w_1, w_2, w_3\}$.

A very closely related problem is the *Covering Salesman Problem* (CSP) where the aim is to identify a minimum-length tour visiting a subset of the vertices in $N$ and covering all the vertices not on the tour. When the subset of vertices that must be on the tour is empty, $T = \emptyset$, the CTP reduces to a CSP, and when $T$ consists of the entire vertex subset, $T = V$, the CTP reduces to the TSP. Therefore, it is NP-hard. The CTP can be formulated, with suitable definitions, as a *Generalized Traveling Salesman Problem* (GTSP) where vertices are clustered and the aim is to identify a minimum-length cycle which visits at least a vertex of each cluster, as explained by Fischetti et al. (1997).
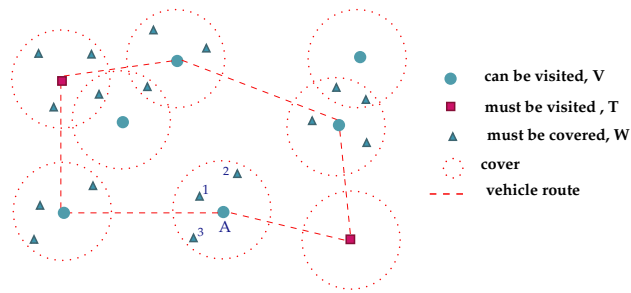


**Fig. 1.** Example of CTP tour

Despite its practical importance, the CTP has not been widely studied. It is introduced and formulated in Current and Schilling (1989) as the *Covering Salesman Problem* (CSP), where they also describe some real world routing problems that can be modelled by the CTP, such as the design of bimodal distribution systems. For instance, cities in the tour are served by air, while the ones not in the tour are served by trucks which originate at their nearest city on the air route. Another application explained is the routing of rural health care delivery teams in developing countries where medical services are only delivered to a subset of villages, but individuals living in villages not in the route are able to reach the medical team at the nearest stop. In the formerly cited work, and in Current and Schilling (1994) the problem is treated as a bicriterion routing problem: the length of the tour and the number of vertices included in it.

In the literature, only one exact method, a branch-and-cut algorithm by Gendreau et al. (1997), has been presented so far to solve the CTP. Hodgson et al. (1998) successfully applied this exact method to the routing of a mobile medical facility in Ghana. Jozefowiez et al. (2007) also propose a bi-objective treatment of the problem: minimization of the tour length and minimization of the covering distance, and develop a two-phase cooperative strategy that combines a multi-objective evolutionary algorithm with the branch-and-cut algorithm of Gendreau et al. (1997). Salari and Naji-Azimi (2012) combine heuristic search and integer linear programming techniques to solve the CSP.

Other heuristic algorithms have also been studied. The ones proposed by Current and Schilling (1989) and Gendreau et al. (1997) are based upon solution procedures for the *Set Covering Problem* (SCP) and the TSP. Motta et al. (2001) have proposed a GRASP metaheuristic to solve a generalized version of the CTP where the tour may also include vertices of set $W$, while Baldacci et al. (2005) have presented three scatter-search heuristic algorithms for the CTP. In addition, Golden et al. (2012) developed a generalized version of the CSP, which they named the *Generalized Covering Salesman Problem* (GCSP), and defined three variants of it for which they proposed two local search heuristics.

The contribution of this study is the development of a new operator that optimally splits a giant tour into visited and not-visited vertices. In other words, it selects the vertices of a given giant tour that comprise the optimal solution for the CTP. A second contribution is the proposal of a state-of-the-art metaheuristic for solving the CTP.

The remainder of this paper is structured as follows. Section 2 presents the Selector operator, while in Section 3 we describe our implementation of an ALNS-based metaheuristic that incorporates the Selector operator. Computational results are presented in Section 4, and conclusions are reported in Section 5.

## 2 Selector Operator

Our solution method is based on the *route first–cluster second* approach proposed by Beasley (1983). The first phase, routing also called ordering, is handled by the ALNS metaheuristic, while the second phase, clustering also called splitting, is handled by our new operator. When solving the CTP, the *Selector* operator splits a giant tour (GT), which is a permutation of all $n$ vertices that can be in the tour, into subsequences of visited and not-visited vertices in a similar way as the *Split* operator segments a GT into feasible vehicle routes when applied to solve the *Capacitated Vehicle Routing Problem* (CVRP) as proposed by Prins (2004). Splitting the GT entails solving a shortest-path problem. However, in the case of the CTP, the covered vertices act as a constraining resource and the problem to be solved then becomes an *Elementary Shortest Path Problem with Resource Constraints* (ESPPRC) (see Feillet et al. 2004).

In Section 2.1, Beasley's approach is explained. Section 2.2 presents the method used by our *Selector* operator to solve the ESPPRC in general terms, whereas Section 2.3 demonstrates its specific algorithmic implementation.

## 2.1 Split Method

The *split* method was firstly presented by Beasley (1983) as the second phase of a route first–cluster second heuristic to solve the CVRP. Relaxing vehicle capacity and maximum route length, the first phase solves a TSP to form a GT that determines the order in which the customers are to be visited. The second phase constructs an auxiliary cost network and then applies a shortest-path algorithm to obtain an optimal partition of the GT into least-cost, capacity-feasible, vehicle routes. This shortest path can be computed using Bellman-Ford's algorithm for directed, acyclic graphs. Beasley provided no computational results for his proposal, and the method neither outperformed more traditional CVRP heuristics nor was it given adequate recognition (see Laporte and Semet 2002). However, when Beasley's seminal method was efficiently implemented within a genetic algorithm (GA) (Prins 2004), it proved to be the first GA able to compete with the best methods available at that time for the solution of the CVRP, i.e. tabu search heuristics. It is since known as the basic *Split* procedure, and other versions of it have been developed to tackle additional constraints as presented in Prins et al. (2009). In the last decade, the route first–cluster second approach has led to successful constructive heuristics and metaheuristics for routing problems as explained in Prins et al. (2014) where a more general name, order–first split–second, is given to the methodology, and an analysis of 70 articles involving splitting procedures is made.

## 2.2 Resource-Constrained Shortest Path Problem

The ESPPRC requires the computation of an elementary shortest path in a network such that the overall resource usage does not exceed the limits; resources are used when visiting vertices or traversing arcs. Hence, record of used resources should be kept. Such problems are NP-hard (Feillet et al. 2004). The standard approach to solve an ESPPRC is dynamic programming (DP) and has pseudopolynomial complexity.

The ESPPRC for the CTP can be solved quickly enough in practice by adapting Desrochers' algorithm (1988), a multi-label version taking resource constraints into consideration of the Bellman-Ford algorithm, which is a label-correcting approach where labels on a vertex are repeatedly extended to its successors. The basic principle of Desrochers' algorithm is to associate with each partial path a label indicating the cost of the path and its consumption of resources, and to eliminate unnecessary labels as the search progresses. Throughout the search, then, every vertex receives labels, and these labels are iteratively extended toward every possible successor vertex until no new labels are created.

Every label representing a feasible path can be understood as a vector $V = (\zeta|r_1, r_2, \ldots, r_k)$ that memorizes the path cost $\zeta$ and the resource consumptions $r_i$ that enable to know if a partial path can still be extended. The effectiveness of the DP algorithm outlined relies upon the feasibility of pruning labels that cannot lead to an optimal solution. For this purpose, suitable dominance tests are always performed when labels are extended, so that the algorithm records only non-dominated labels.

### 2.3 Selector Algorithm Proposed

**Auxiliary Graph.** *Selector* works on a directed, acyclic graph $M = (V', A)$, where $V'$ represents the position in the giant tour of the $n$ vertices that can be visited in the original graph, i.e., the representation of $GT = (GT_0, GT_1, \ldots, GT_{n-1})$ in $V'$ implies $v_i = GT_i$. Thus, in the following, when refering to vertex $v_i$ we imply the vertex located in position $i$ in $GT$. Figure 2 shows an example for $n = 6$ and depicts only some of the possible arcs. An arc $(i, j) \in A \mid j \geqslant i + 2$ models a subpath that visits only vertices $v_i$ and $v_j$. Such arc exists only if it is feasible to skip the points located between vertices $v_i$ and $v_j$. For instance, arc $(v_1, v_4)$ in Fig. 2 indicates a subpath that visits vertex $v_1$, skips $v_2$ and $v_3$, and ends at vertex $v_4$. This arc will be kept if no $w_i \in W$ remains uncovered despite skipping vertices $v_2$ and $v_3$. This means that the subset $\{v_1, v_4, v_5\}$ covers all $w_i \in W$. The algorithm creates a vector of size $n$ to represent $M$, and this vector maintains the labels (subpaths) generated to reach each vertex $v_i, i \in \{1, 2, \ldots, n-1\}$. The weight of an arc is equal to the Euclidean distance, $d_{ij}$, between the vertex located in position $i$ and the one in position $j$ in $GT$.

An optimal solution for a given $GT$ indicates a minimum-cost path $\sigma$ from $0$ to $n - 1$ in $M$. This result can be seen as the splitting of the giant tour into visited and not-visited vertices. Finding $\sigma$ has pseudopolynomial complexity.
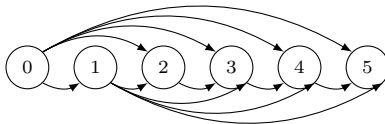


**Fig. 2.** Auxiliary graph $M$ representing optional visits for vertices 0 and 1.

**Labels and Their Control.** Starting from $v_0$, a vertex in $M$ may be reached through different permutations of visited and not-visited predecessors and each with a different cost and a different coverage of the vertices in $W$. In practice, this information is stored in labels. As a result, several labels might exist at each vertex. A label $\lambda_j$ stored in vertex $v_i \in M \backslash \{v_0\}$ represents a path that starts at the depot and ends at vertex $v_i$. It contains five fields $\lambda_j^i = (\zeta, \omega[k], v_i, \pi, \nu)$ which are useful for the decision making at different stages of the algorithm. $\zeta$ memorizes the cost (sum of the arc weights) of the path represented, $\omega[k]$ is a vector in which location $i$ either stores the number of vertices in $V$ that can still cover vertex $w_i \in W$ or stores a flag indicating vertex $w_i$ is covered, $v_i$ keeps the last visited vertex (site where label is stored), $\pi$ stores the path from the depot to vertex $v_i$, and $\nu$ keeps the number of vertices already covered.

The dominance rule applied to control label proliferation is as follows. Let $\Lambda^i = \{\lambda_1^i, \lambda_2^i, \ldots, \lambda_k^i\}$ be the set of labels associated with vertex $v_i \in M \backslash \{v_0\}$,

and $\Omega_j$ the set of vertices covered by label $\lambda_j$. A label $\lambda_1 \in \Lambda^i$ dominates $\lambda_2 \in \Lambda^i$, with $\lambda_1 \neq \lambda_2$, if $\zeta_1 \leq \zeta_2$ and $\Omega_2 \subseteq \Omega_1$.

A look ahead mechanism also allows to reduce the number of labels created. If when extending a label it is found that a vertex $v_i \in V \backslash T$ must be visited in the future because it is the only one that can cover a set $\Gamma \subset W$, then mark all the vertices $w_i \in \Gamma$ as covered. The result of this look ahead is that it is known then there is no need to visit any vertex $v_j \in V \backslash T$ that only usefully covers vertices in $\Gamma \subset W$, and as a consequence, less labels are produced.

On the other hand, the following feasibility rule is used. Let $\bar{\Omega}_j$ denote the subset of vertices of $W$ that are not covered by the subpath represented by label $\lambda_j$. A vertex $v_i \in V \backslash T$ can only be skipped if for each $w_i \in \bar{\Omega}_j$, there still remain vertices ahead that can cover it. The number of such vertices is kept through field $\omega[k]$. Thus, when the decision to skip a node is evaluated, for each $w_i \in \bar{\Omega}_j$, feasible labels yield $w[i] > 0$. Such value indicates that no vertex is left uncovered, so it is feasible to skip vertex $v_i \in V \backslash T$. For computational efficiency, a matrix relating the coverage of the vertices $w_i \in W$ by the vertices $v_i \in V$ is precomputed and kept at hand.

Other way to control label proliferation in this algorithm is the computation and updating of an upper bound as it will be explained in the ensuing section.

---

**Algorithm 1** : Selector

---

**Input:** giant tour $GT$, distance matrix $D$, set $T$
**Output:** optimal tour of visited vertices, $S$, and cost value of tour, $c(S)$
 1: $L^* \leftarrow$ search upper bound {See Algorithm 2}
    {build an initial set of labels}
 2: **while** ( $\exists$ arc$(v_0, v_i)$ ) **do**
 3:    $\Lambda^i \leftarrow \Lambda^i \cup \{L\}$ {$L$ is the label being treated}
 4:    Extend Horizontally$(L)$ {see Algorithm 3}
 5:    $i \leftarrow i + 1$
 6: **end while**
    {extend labels created}
 7: **while** ( $\exists$ an $\Lambda^i$ ) **do**
 8:    $L \leftarrow \min_{i \in N}\{\lambda_j\}$ {find label of lowest cost}
 9:    Extend Skipping$(L)$ {see Algorithm 4}
10: **end while**

---

**Finding the Shortest Path.** Algorithm 1 illustrates the core procedure of *Selector*. It executes three main steps: (i) search for an initial feasible solution or upper bound (UB), (ii) build an initial set of labels, and (iii) extend the created labels. In the explanations that follow the term *horizontal extension* means to iteratively visit in $M$ the adjacent successor vertex until a complete feasible solution is built or any other of the stopping criteria is met (see Algorithm 3). The worst-case time complexity of the horizontal extension process is $O(n)$.

As mentioned, besides the feasibility and dominance rules, the upper bound is helpful to limit the creation of labels, and it is computed as follows (refer to Algorithm 2). For every vertex $v_i \in M$ for which arc $(v_0, v_i)$ exists, it builds arc $(v_i, v_{i+1})$ and from this point continues the construction with a horizontal extension. Next, it constructs the arc to the next successor, arc $(v_i, v_{i+2})$, and proceeds with the same horizontal extension, and so on. The process stops when arc $(v_i, v_{i+k})$ can no longer be constructed, and restarts with the creation for the next vertex of arc $(v_0, v_{i+1})$. In this first step, every built path is compared and the best one is kept, no labels are stored in order to execute it fast. The worst-case time complexity of the search is $O(n^3)$.

The second step, the generation of an initial set of labels, iteratively constructs arc $(v_0, v_i)$ followed by a horizontal extension. However, at each step (at every vertex) a non-dominated label documenting the subpath is stored. The process repeats as long as it is possible to construct arc $(v_0, v_i)$. The worst-case time complexity of this step is $O(n^2)$.

Finally, the created labels are extended (see Algorithm 4). This means that from the last visited vertex stored in the label, $v_{\text{last}}$, it tries to reach successor $v_{\text{last}+2}$ and from this point does a horizontal extension storing non-dominated labels at every step. The process repeats as long as arc $(v_{\text{last}}, v_{\text{last+k}})$ exists. The label chosen for extension is always the one that documents the shortest path and the execution of Algorithm 4 continues until there are no labels to extend. The worst-case time complexity of this extension is $O(n^2)$. Nevertheless, it might be executed for several thousands of labels in large instances.

As can be observed in Algorithm 3, at every step of the label extension the following conditions are verified: (a) the vertex to be included is not redundant, (b) $\zeta_{\text{subpath}} < \zeta_{\text{best}}$, (c) $UB_{\text{current}} < UB_{\text{best}}$, and (d) label is not dominated. Any vertex that turns out to be redundant is simply skipped and the construction continues, no labels are kept for not-visited vertices. If the cost of the path being built is worse than the cost of the best known solution, the search in that trajectory is abandoned. The initial upper bound built at the onset is updated throughout the search to improve the limits for the creation of labels.

A distinctive and important characteristic of our operator is that aside from the constraints mentioned in the definition of the problem, it does not impose any further restrictions on the selected vertices of $V \backslash T$ such as adjacency, for example. This operator is capable of discarding any vertex $v_i \in V \backslash T$ at any point in the tour.

## 3 Adaptive Large Neighborhood Search (ALNS)

In our methodology, the overall task of the ALNS metaheuristic is to build suboptimal giant tours from which efficient CTP solutions are extracted. This is, according to Beasley's method (1983), the ordering phase, and the splitting phase is performed by the *Selector* operator embedded into this heuristic. ALNS, a local search framework which uses several competing destroy and repair methods and chooses amongst them using statistics gathered during the search, com-

**Algorithm 2** : Search Upper Bound
___
**Input:** giant tour $GT$, distance matrix $D$, set $T$
**Output:** feasible tour of visited vertices, $S$, and cost value of tour, $c(S)$
1: $i \leftarrow 1$
2: **while** ( $\exists \, \mathrm{arc}(v_0, v_i)$ ) **do**
3: $\quad k \leftarrow i$
4: $\quad$ **while** ( $\exists \, \mathrm{arc} \, (v_i, v_{k+1})$ ) **do**
5: $\quad\quad$ Extend Horizontally($L$) {see Algorithm 3}
6: $\quad\quad k \leftarrow k + 1$
7: $\quad$ **end while**
8: $\quad i \leftarrow i + 1$
9: **end while**
___

**Algorithm 3** : Extend Horizontally($L$)
___
**Input:** label to be extended, $L$
**Output:** labels derived from $L$
$\quad$ {only nondominated labels that can be extended are kept}
1: **for** ($j = lastVisitedNode + 1$ **to** $j \leqslant n$) **do**
2: $\quad clientsCovered \leftarrow clientsCovered + coverage \, of \, j$
3: $\quad$ **if** ( node is not redundant ) **then**
4: $\quad\quad cost \, of \, L \leftarrow cost \, of \, L + cost \, of \, visiting \, j$
5: $\quad\quad$ **if** ( $cost \, of \, L < cost \, of \, L^*$ ) **then**
6: $\quad\quad\quad$ **if** ( $clientsCovered \neq clients$ ) **then**
7: $\quad\quad\quad\quad$ **if** ( $L$ not dominated ) **then**
8: $\quad\quad\quad\quad\quad \Lambda^i \leftarrow \Lambda^i \cup \{L\}$
9: $\quad\quad\quad\quad$ **end if**
10: $\quad\quad\quad$ **else**
11: $\quad\quad\quad\quad L^* \leftarrow L$
12: $\quad\quad\quad\quad$ **return**
13: $\quad\quad\quad$ **end if**
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ **return**
16: $\quad\quad$ **end if**
17: $\quad$ **end if**
18: **end for**
___

**Algorithm 4** : Extend Skipping($L$)
___
**Input:** label to be extended, $L$
**Output:** labels derived from $L$
1: $i \leftarrow lastVisitedNode$
2: $k \leftarrow 2$
3: **while** ( $\exists \, \mathrm{arc} \, (v_i, v_{i+k})$ ) **do**
4: $\quad$ Extend Horizontally($L$)
5: $\quad k \leftarrow k + 1$
6: **end while**
___

petes strongly with genetic algorithms (GA) in vehicle routing. However, the efficiency of GAs relies on sophisticated local search methods and population management techniques, while in ALNS neighborhoods are searched by simple and fast heuristics. ALNS has provided good solutions for a wide variety of VRPs as shown in Pisinger and Ropke (2010) and Ribeiro and Laporte (2011).

The three backbones of our implementation are (i) removal operators, (ii) insertion operators and (iii) metaheuristic that defines the criteria to accept a new solution. Three removal and three insertion heuristics were implemented. In the following, the word facility indicates a vertex that belongs to the giant tour, and lower-case Greek letters indicate user-controlled parameters.

### 3.1 Removal Operators

**Shaw Removal Heuristic (SRH).** Originally proposed by Shaw (1997), its general idea is to remove facilities that exhibit similitude, characteristic computed by a *relatedness measure* $R(i,j)$. For this implementation, the similarity between two facilities is measured by $R(i,j) = d_{ij}$, where $d_{ij}$ is the Euclidian distance between facilities $i$ and $j$. This relatedness measure is used to remove facilities in the same way as described by Shaw (1998). In order to avoid the sorting of facilities required at each iteration, a nearest facility matrix is precomputed and kept at hand. The worst-case time complexity of the SRH is $O(n^2)$.

**Worst Removal Heuristic (WRH).** Ropke and Pisinger (2006) propose a heuristic that randomly removes facilities with a high cost in the current solution $X$ and tries to insert them in better positions. It iterates recalculating the costs until it has removed the indicated number of facilities. The removal, though random, is user-controlled by parameter $\rho$. The worst-case time complexity of the WRH is $O(n^2)$.

**Random Removal Heuristic (RRH).** This procedure simply selects $\gamma$ facilities at random and removes them from the current solution $X$. Though it tends to generate a poor set of removed members, it is useful to diversify the search. The worst-case time complexity of the RRH is $O(n)$.

**How Many to Remove.** The number of facilities removed, $\gamma$, from the current solution $X$ is key to the ALNS performance. When few elements are removed, the heuristic has a higher probability of being trapped in one suboptimal area of the search space. On the other hand, when too many are removed, it is almost like starting from scratch and the insertion heuristics cannot build a good solution from such situation. In addition, the larger the number removed, the larger the execution time of both insertion and removing heuristics. We choose $\gamma$ randomly between a lower and upper limit. The lower limit is fixed at a value given according to the number of vertices in set $V$, 20% to 25% of its size, while the upper limit is fine-tuned with parameter $\epsilon$. This parameter indicates the maximum percentage of elements removed from the complete solution size.

### 3.2 Insertion Operators

**Best Greedy Heuristic (BGH).** This simple construction heuristic performs at most $\gamma$ iterations as it inserts one facility into solution $X$ in each iteration. The minimum cost position value is computed for all facilities waiting insertion, set $F$, and the one with the minimum global cost position is chosen. This process is repeated until $F = \emptyset$. The worst-case time complexity of the BGH is $O(n^2)$.

**First Greedy Heuristic (FGH).** This heuristic works similarly to the previous one. However, instead of inserting the facility having the minimum global cost position, it inserts the one sitting in the first position. That is to say, it respects the order of the facilities in $F$. After the first facility has been inserted, the minimum cost position for each is recalculated and the process repeats until all facilities in set $F$ have been inserted.

Ropke and Pisinger(2006) add a noise term to the objective function during the insertion phase of the BGH and regret-$k$ heuristics in order to randomize them and avoid always making the move that seems best locally. In our implementation, the FGH is used mainly to introduce this noise into the insertion process as done by Ribeiro and Laporte(2011). This heuristic obviously runs faster than the BGH.

**Regret-$k$ Heuristic (RKH).** This heuristic tries to improve the myopic behaviour of the greedy heuristics by incorporating a kind of look ahead information when selecting the facility to insert, as done by Ropke and Pisinger(2006) and Pisinger and Ropke(2007). Let $\Delta f_i^1$ denote the change in objective value incurred by inserting facility $i$ at its minimum cost position, and $\Delta f_i^2$ denote the change by inserting it at its second best position. The regret value is defined in terms of the former values as $c_i^* = \Delta f_i^2 - \Delta f_i^1$. In each iteration, the regret heuristic chooses to insert the facility $i$ that maximizes $max_{i \in F}\{c_i^*\}$, and such facility is inserted at its minimum cost position. Ties are broken by selecting the facility with lowest cost insertion. This is a time-consuming operator but unnecessary computations were avoided when computing $\Delta f_i^n$. The worst-case time complexity of the RKH is $O(n^3)$.

**Choosing a Removal and an Insertion Heuristic.** In order to select a heuristic, weights are assigned to them and a *roulette wheel selection principle* is applied. The removal heuristic is selected independently of the insertion heuristic and vice versa. Initially, all heuristics are equally likely.

**Adaptive Weight Adjustment.** Based on its performance, the probability of choosing a heuristic changes. To enable this change, a score is kept for each and it is updated at each iteration. Our implementation keeps track of visited solutions using a hash table. A hash key is assigned to every solution and this key is stored in the table. We followed the scheme of scores and updating procedure of probability weights proposed by Ropke and Pisinger (2006).

**Table 1.** Values of the ALNS parameters after experimental tuning.

| Parameter | Meaning | Value |
|---|---|---|
| $\gamma$ | number of facilities removed at each ALNS iteration (instance size dependent) | [5, 30] |
| $\varsigma$ | segment size for updating probabilities in number of ALNS iterations | 50 |
| $\tau$ | reaction factor that controls the rate of change of the weight adjustment | 0.3 |
| $\delta$ | avoids determinism in the SRH | 6 |
| $\rho$ | avoids determinism in the WRH | 2 |
| $\sigma_1$ | score for finding a new global best solution | 50 |
| $\sigma_2$ | score for finding a new solution that is better than the current one | 20 |
| $\sigma_3$ | score for finding a new non-improving solution that is accepted | 5 |
| $\beta$ | cooling factor used by simulated annealing | 0.99999 |
| $\epsilon$ | fixes the upper limit of facilities removed at each iteration | 0.3 |

### 3.3 General Framework with Simulated Annealing

Algorithm 5 depicts the ALNS process implemented with simulated annealing (SA) as the outer metaheuristic that guides the search. We followed the SA scheme suggested by Pisinger and Ropke (2007). The algorithm works on two items: the giant tour, $GT$, constructed by the ALNS heuristics, and $S$, the solution covering tour computed by *Selector* when applied to $GT$. The cost of $GT$ is labelled $l$, while the cost value for solution $S$ is identified as $c$. The variable $GT_{\mathrm{current}}$ indicates the solution obtained at the beginning of an iteration, and the variable $GT_{\mathrm{new}}$ is the temporary solution obtained during the iteration. For the sake of clarity, the updating processes for scores, hash keys and probabilities are not shown.

The 2-*opt* procedure is used to rapidly improve the length of the starting GT and avoid a long, random, initial walk. In addition, the acceptance of $GT_{\mathrm{new}}$ is controlled by the cost value of solution $S$. The latter is because experimentation showed that improvements on the length of the giant tour do not necessarily lead to improvements on the length of the tour computed by *Selector*. However, in the long run, the length of the CTP tour benefits from improvements on the length of the GT. An important consequence of this finding is that execution of *Selector* to optimality at each ALNS iteration is of low benefit. The upper bound computed by Algorithm 2 can serve as a probe to determine if the complete process is worth executing. This derives in important time savings.

## 4 Computational Results

The results obtained by our metaheuristic are compared against the optimal solutions computed by the branch-and-cut algorithm of Gendreau et al. (1997). The exact algorithm is written in Python 2.7 and uses 5.6 Gurobi callbacks. Library Python-Igraph 0.7.0 helps to solve graph problems occurring in the valid cut separation. The heuristic algorithms are coded in C++ and the benchmark

**Algorithm 5** : The General Framework of the ALNS with Simulated Annealing

**Input:** Giant tour $GT$, distance matrix $D$

**Output:** $S_{\text{best}}$ and $c(S_{\text{best}})$

1: 2-opt$(GT_0)$
2: compute $l_0(GT_0)$
3: initialize, to the same value, probability $P_r^t$ for each removal operator $r \in R$, and likewise probability $P_i^t$ for each insertion operator $i \in I$.
4: $t \leftarrow l_0$, {set initial temperature, variable used in probability function}
5: $l_{\text{current}} \leftarrow l_0$
6: $GT_{\text{current}} \leftarrow GT_0$
7: $UB_{\text{best}} \leftarrow$ Search Upper Bound$(GT_0)$ {see Algorithm 2}
8: $c(S_{\text{best}}) \leftarrow c(S_{\text{current}}) \leftarrow$ Selector$(GT_0)$ {see Algorithm 1}
9: $i \leftarrow 1$ {iteration counter}
10: **repeat**
11:   select a removal operator $r \in R$ with probability $P_r^t$ {roulette wheel}
12:   obtain $GT_{\text{new}}^-$ by applying $r$ to $GT_{\text{current}}$
13:   select an insertion operator $i \in I$ with probability $P_i^t$
14:   obtain $GT_{\text{new}}$ by applying $i$ to $GT_{\text{new}}^-$
15:   $UB_{\text{current}} \leftarrow$ Search Upper Bound$(GT_{\text{new}})$
16:   **if** ( $UB_{\text{current}} < UB_{\text{best}}$ ) **then**
17:     $c(S_{\text{new}}) \leftarrow$ Selector$(GT_{\text{new}})$
18:     $UB_{\text{best}} \leftarrow UB_{\text{current}}$
19:   **else**
20:     $c(S_{\text{new}}) \leftarrow UB_{\text{current}}$
21:   **end if**
      {decide acceptance of new solution}
22:   **if** ( $c(S_{\text{new}}) < c(S_{\text{current}})$ ) **then**
23:     $c(S_{\text{current}}) \leftarrow c(S_{\text{new}})$
24:     $GT_{\text{current}} \leftarrow GT_{\text{new}}$
25:   **else**
26:     $p \leftarrow e^{-\frac{c(S_{\text{new}}) - c(S_{\text{current}})}{t}}$
27:     generate a random number $n \in [0,1]$
      {new solution might be accepted with a computed probability even it is worse}
28:     **if** ( $n < p$ ) **then**
29:       $c(S_{\text{current}}) \leftarrow c(S_{\text{new}})$
30:       $GT_{\text{current}} \leftarrow GT_{\text{new}}$
31:     **end if**
32:   **end if**
33:   **if** ( $c(S_{\text{new}}) < c(S_{\text{best}})$ ) **then**
34:     $c(S_{\text{best}}) \leftarrow c(S_{\text{new}})$
35:     $S_{\text{best}} \leftarrow S_{\text{new}}$
36:   **end if**
37:   $t \leftarrow \beta \cdot t$ {cooling rate set to be very slow}
38:   **if** ( segment size $= \varsigma$ ) **then**
39:     update probabilities using the adaptive weight adjustment procedure
40:   **end if**
41:   $i \leftarrow i + 1$
42: **until** ( defined number of iterations is met )

was done on a computer with 8 GiB of memory, processor Intel Core i7-4770 CPU@3.40GHz, and Linux OS type 64 bits.

Since test problems for the CTP are not found in the literature, we created data sets based on 9 Euclidean TSPLIB instances (Reinelt 1991) whose sizes range from 100 to 200 vertices. Sets of vertices of $|V \cup W| \in \{100, 150, 200\}$ were created using kroX100 (X $\in$ {A,B, ... ,E}), kroX150 and kroX200 (X $\in$ {A,B}) respectively. $T$ and $V$ are defined by taking the first $|T|$ and $|V| - |T|$ points, respectively, while $W$ is defined by the remaining points. Tests were run for $|V| \in \{25, 50, 75, 100\}$. $|T| = 1$, only the depot is compulsory, which is the worst case regarding the number of labels created.

The costs $\{\zeta_{ij}\}$ are treated as integer values equal to $\lfloor d_{ij} + .5 \rfloor$, where $d_{ij}$ is the Euclidean distance between points $i$ and $j$ (Reinelt 1991). The value of $c$ is resolved using $c = \max \left( \max_{v_k \in V \setminus T} \min_{w_l \in W} \{\zeta_{l,k}\}, \max_{w_l \in W} \{\zeta_{l,k(l)}\} \right)$, where $k(l)$ indicates the second nearest vertex $v_k \in V \setminus T$. Computing this value in such way ensures that each vertex $v_i \in V \setminus T$ covers at least one vertex $w_i \in W$, and each vertex $w_i \in W$ is covered by at least two different vertices $v_i \in V \setminus T$ as explained in Gendreau et al. (1997).

Several independent executions were done to test our randomized heuristic. Each instance was run 30 times with a different seed each time and for 30,000 iterations. To define an efficient parameter set, we used a *ceteris paribus* approach based on sets of three or four values for each parameter. The resulting set is listed in Table 1. On the other hand, both the optimal values and the quality of the solutions computed by the heuristic can be observed in Table 2 where the first three columns document the instance information, the next three report the findings of the exact method, and the last five those of the approximate approach. Columns UB and Opt show the time in seconds needed to reach an upper bound and the optimal value respectively. Column $\bar{\theta}$ indicates the deviation of the heuristic solution from the optimum value in percentage, and $\bar{t}$ corresponds to the total run time in seconds. These two figures are average values over the 30 runs. Column Found indicates how many times the heuristic found the optimum value in the set of runs. Column Best Gap shows how close (in percentage) the heuristic came to the optimum value, and the last one, labeled $S_{N-1}$, exhibits the corrected sample standard deviation.

On the whole, the heuristic is very accurate and its performance is highly satisfactory, since for 96% of the instances it was capable of finding the optimum value rapidly. In the few cases where the optimum was not reached, the minimum value computed was less than 1% away from the optimal solution value. In addition, the average deviation is typically within 1% of optimality. Also, it repeatedly found the optimum value for 63% of the instances. In general, given an instance, this number worsens as $|V|$ increases. Results are reported for 30,000 iterations. However, observing the evolution of the search, we could see that optimal solutions were identified for approximately 75% of the instances as early as in the first 1,000 iterations. Furthermore, in general, the spread around the optimum of the values computed is very moderate. We can, thus, state that it is a heuristic capable of identifying very good solutions quite quickly.

**Table 2.** Performance of heuristic compared to the branch-and-cut algorithm.

| Instance Based on | $|V|$ | $|W|$ | Optimum | UB(s) | Opt(s) | $\overline{\theta}(\%)$ | $\overline{t}(s)$ | Found | Best Gap(%) | $S_{N-1}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| kroA100 | 25 | 75 | 7985 | 0.17 | 0.17 | 2.36 | 0.42 | 14 | 0 | 272.51 |
| kroA100 | 50 | 50 | 8608 | 22.20 | 44.95 | 0.21 | 0.95 | 11 | 0 | 23.06 |
| kroB100 | 25 | 75 | 6449 | 0.21 | 0.27 | 0.16 | 0.50 | 24 | 0 | 22.79 |
| kroB100 | 50 | 50 | 8043 | 1.18 | 21.54 | 0.70 | 1.25 | 1 | 0 | 60.20 |
| kroC100 | 25 | 75 | 6161 | 0.01 | 0.01 | 0 | 0.81 | 30 | 0 | 0 |
| kroC100 | 50 | 50 | 7942 | 0.81 | 0.81 | 0 | 2.27 | 30 | 0 | 0 |
| kroD100 | 25 | 75 | 6651 | 0.24 | 0.38 | 0 | 0.31 | 30 | 0 | 0 |
| kroD100 | 50 | 50 | 8411 | 3.75 | 4.33 | 0.02 | 1.13 | 27 | 0 | 4.64 |
| kroE100 | 25 | 75 | 7417 | 0.26 | 0.27 | 0.02 | 0.42 | 29 | 0 | 8.71 |
| kroE100 | 50 | 50 | 8493 | 1.10 | 1.11 | 0 | 1.00 | 30 | 0 | 0 |
| kroA150 | 25 | 125 | 8050 | 0.13 | 0.13 | 1.43 | 0.54 | 3 | 0 | 131.72 |
| kroA150 | 50 | 100 | 9623 | 118.80 | 121.58 | 0.37 | 1.16 | 2 | 0 | 38.56 |
| kroA150 | 75 | 75 | 9971 | 1569.38 | 2884.34 | 0.60 | 2.93 | 0 | 0.59 | 59.81 |
| kroB150 | 25 | 125 | 6165 | 0.01 | 0.01 | 0 | 1.50 | 30 | 0 | 0 |
| kroB150 | 50 | 100 | 7818 | 1.16 | 1.16 | 0.02 | 2.32 | 29 | 0 | 7.23 |
| kroB150 | 75 | 75 | 7434 | 13.34 | 38.24 | 0.01 | 4.32 | 26 | 0 | 2.16 |
| kroA200 | 25 | 175 | 6165 | 0.01 | 0.01 | 0 | 1.63 | 30 | 0 | 0 |
| kroA200 | 50 | 150 | 8273 | 0.46 | 0.49 | 0 | 5.09 | 30 | 0 | 0 |
| kroA200 | 75 | 125 | 8499 | 141.97 | 266.19 | 0 | 6.31 | 30 | 0 | 0 |
| kroA200 | 100 | 100 | 8355 | 4110.87 | 4789.22 | 0 | 14.21 | 30 | 0 | 0 |
| kroB200 | 25 | 175 | 6450 | 0.15 | 0.15 | 0.18 | 1.17 | 23 | 0 | 24.62 |
| kroB200 | 50 | 150 | 8171 | 2.69 | 3.46 | 0.78 | 2.6 | 3 | 0 | 77.29 |
| kroB200 | 75 | 125 | 10007 | 0.65 | 0.65 | 1.42 | 4.5 | 5 | 0 | 166.52 |
| kroB200 | 100 | 100 | 9988 | 17.20 | 17.68 | 1.73 | 11.60 | 5 | 0 | 202.63 |

## 5 Conclusions

This paper presents a study on a novel resolution method for a difficult combinatorial optimization problem which finds application in network design and vehicle routing. Its key feature is the *Selector* operator that optimally splits an initial sequence of facilities into subsequences of visited and not-visited vertices. We have proposed an approximate method capable of obtaining very high quality solutions in very short periods of time. It is a simple, easy to implement heuristic and its core, the *Selector* operator, is new and creative in its own right. Given the practical relevance of the CTP, we will look into other heuristic mechanisms to solve large-scale instances. We believe the approach developed in this work can be translated or adapted to solve related TLPs like the *Orienteering Problem*, also known as the *Selective TSP*, and the *Prize Collecting TSP*.

## References

Baldacci, R., Boschetti, M.A., Maniezzo, V., Zamboni, M.: Scatter search methods for the covering tour problem. In: Rego, C., Alidaee, B. (eds.) Metaheuristic Optimization Via Memory and Evolution, pp. 59–91. Kluwer, Boston (2005)

Beasley, J.: Route first–cluster second methods for vehicle routing. OMEGA The International Journal of Management Science 11, 403–408 (1983)

Current, J.R., Schilling, D.A.: The covering salesman problem. Transportation Science 23, 208–213 (1989)

Current, J.R., Schilling, D.A.: The median tour and maximal covering problems. European Journal of Operational Research 73, 114–126 (1994)

Desrochers, M.: An algorithm for the shortest path problem with resource constraints. Technical Report G-88-27, GERAD, Montréal (1988)

Feillet, D., Dejax, P., Gendreau, M., Gueguen, C.: An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems. Networks 44, 216–229 (2004)

Fischetti, M., Salazar, J.J., Toth, P.: A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. Operations Research 45, 378–394 (1997)

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

Golden, B., Naji-Azimi, Z., Raghavan, S., Salari, M., Toth, P.: The generalized covering salesman problem. INFORMS Journal on Computing 24, 534–553 (2012)

Hodgson, J., Laporte, G., Semet, F.: A covering tour model for planning mobile health care facilities in Suhum district, Ghana. Jl of Regional Sci. 38, 621–638 (1998)

Jozefowiez, N., Semet, F., Talbi, E.G.: The bi-objective covering tour problem. Computers & Operations Research 34, 1929–1942 (2007)

Laporte, G., Semet, F.: Classical and new heuristics for the vehicle routing problem. In: Toth, P., Vigo, D. (eds.) The Vehicle Routing Problem, pp. 109–128. SIAM, Philadelphia (2002)

Motta, L., Ochi, L.S., Martinhon, C.: GRASP metaheuristics to the generalized covering tour problem. In: MIC'2001–4th Metaheuristics International Conference, pp. 387–391. Porto, Portugal (2001)

Pisinger, D., Ropke, S.: A general heuristic for vehicle routing problems. Computers & Operations Research 34, 2403–2435 (2007)

Pisinger D., Ropke, S.: Large neighborhood search. In: Gendreau, M., Potvin, J.-Y. (eds.) Handbook of Metaheuristics, pp. 399–419. Springer, New York (2010)

Prins, C.: A simple and effective evolutionary algorithm for the vehicle routing problem. Computers & Operations Research 31, 1985–2002 (2004)

Prins, C., Labadi, N., Reghioui, M.: Tour splitting algorithms for vehicle routing problems. International Journal of Production Research 47, 507–535 (2009)

Prins, C., Lacomme, P., Prodhon, C.: Order-first split-second methods for vehicle routing problems: A review. Transportation Research 40, 179–200 (2014)

Reinelt, G.: TSPLIB - a traveling salesman problem library. ORSA Journal on Computing 3, 376–384 (1991)

Ribeiro, G., Laporte, G.: An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem. CIRRELT, (2011)

Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transport. Sci. 40, 455–472 (2006)

Salari, M., Naji-Azimi, Z.: An integer-programming-based local search for the covering salesman problem. Computers & Operations Research 39, 2594–2602 (2012)

Shaw P.: A new local search algorithm providing high quality solutions to vehicle routing problems. Technical Report, University of Strathclyde, Scotland (1997)

Shaw P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming, pp. 417–431. Springer, New York (1998)