Computers and Operations Research 87 (2017) 1–19

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor

GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem☆

Stephen L. Smith*, Frank Imeson

Department of Electrical and Computer Engineering, University of Waterloo, Waterloo ON, N2L 3G1 Canada

ARTICLE INFO

Article history:
Received 21 March 2016
Revised 9 May 2017
Accepted 16 May 2017
Available online 17 May 2017

Keywords:
Generalized Traveling Salesman Problem
Traveling Salesman Problem
Adaptive Large Neighborhood Search
Clustered Traveling Salesman

ABSTRACT

This paper presents a new solver for the exactly one-in-a-set Generalized Traveling Salesman Problem (GTSP). In the GTSP, we are given as input a complete directed graph with edge weights, along with a partition of the vertices into disjoint sets. The objective is to find a cycle (or tour) in the graph that visits each set exactly once and has minimum length. In this paper we present an effective algorithm for the GTSP based on adaptive large neighborhood search. The algorithm operates by repeatedly removing from, and inserting vertices in, the tour. We propose a general insertion mechanism that contains as special cases the well-known nearest, farthest and random insertion mechanisms. We provide extensive benchmarking results for our solver in comparison to the state-of-the-art on a wide range of existing and new problem libraries. We show that on the GTSP-LIB library, the proposed algorithm is competitive with the best known algorithms. On several other libraries, we show that given the same amount of time, the proposed solver finds higher quality solutions than existing approaches, particularly on harder instances that are non-metric and/or whose sets are not clustered.

1. Introduction

This paper presents a new solver called GLNS for the exactly-one-in-a-set Generalized Traveling Salesman Problem (GTSP). In the GTSP, we are given as input a complete directed graph with edge weights (that is, the edges weights may be asymmetric), along with a partition of the vertices into disjoint sets. The goal is to compute a minimum length cycle in the graph that contains exactly one vertex from each set in the partition. A common variant of this problem is to find the minimum cycle that visits each set at least once. These two problems are equivalent if the edge weights are metric. A further generalization of the GTSP allows the vertex sets to be non-disjoint (i.e., overlapping). In this paper we do not directly address either of these variants. However, the authors of Noon and Bean (1993) provide reductions from each variant to the exactly-one-in-a-set GTSP, and thus the results in this paper can also be applied to these variants.

The GTSP has a variety of applications in operations research (Laporte et al., 1996), including material flow design, vehicle routing, and post-box collection. In robotics, a common problem is to plan tours through a set of points in a robot's workspace (Saha et al., 2006). However, due to the number of degrees of freedom of the robot, there may be several robot configurations that reach a desired workspace position. A common solution technique is to convert the problem into a GTSP, in which each set contains a sample of different robot configurations for the given location (Ny et al., 2012; Obermeyer et al., 2012; Saha et al., 2006). The GTSP also arises in complex motion planning problems in which the goal is to compute a tour over a set of locations, but with additional constraints on which combinations of locations are or are not allowed in the tour (Imeson and Smith, 2015; Mathew et al., 2015; Wolff et al., 2013). In such problems the GTSP instances are typically not metric or symmetric, and thus pose significant challenges for generating high-quality solutions.

Related Work: Many approaches have been proposed for solving the GTSP. The GTSP is a direct generalization of the TSP, and thus no approximation algorithm exists for the general problem. If the edge weights are metric, then the best known approximation algorithm yields an approximation factor of $O(\log^2 n \log\log n \log m)$ (Garg et al., 2000), where n is the number of vertices and m is the number of sets.

A commonly used solution approach is to reduce the GTSP to an asymmetric TSP instance using the Noon–Bean reduction (Ben-Arieh et al., 2003; Noon and Bean, 1993) and then solve the TSPinstance with a standard TSPsolver (Helsgaun, 2000; Lin and Kernighan, 1973). Other approaches include Lagrangian

☆ This research is partially supported by an Early Researcher Award from the Ontario Ministry of Research and Innovation.

* Corresponding author.
E-mail addresses: stephen.smith@uwaterloo.ca (S.L. Smith), stephen.smith@uwaterloo.ca (F. Imeson).

http://dx.doi.org/10.1016/j.cor.2017.05.010
0305-0548/

relaxations (Noon and Bean, 1991) and branch-and-bound techniques (Fischetti et al., 1997) based on properties of the integer linear program representation (Fischetti et al., 1995). In Fischetti et al. (1997), the authors also developed a library of GTSPinstances, called GTSP-Lib, by taking TSP-Lib instances and performing clustering on the vertices. This library was subsequently extended to add larger problem instances in Silberholz and Golden (2007).

Several genetic and memetic algorithms have also been proposed for the GTSP. Early algorithms included Snyder and Daskin (2006) and Silberholz and Golden (2007). In benchmarks on the GTSP-Lib, performance was subsequently improved in Bontoux et al. (2010) and then in Gutin and Karapetyan (2010). The memetic algorithm in Gutin and Karapetyan (2010), called GK, yields impressive performance on GTSP-Lib, with runtimes under 60 s and tours consistently within 0.1% of the best known. However, it was shown in Drexl (2013) that the memetic algorithm (Gutin and Karapetyan, 2010) solver's performance degraded significantly with problem size on rural postman problems, in particular for problems consisting of more than 200 sets.

In Karapetyan and Gutin (2011), a generalization of the successful Lin–Kernighan TSPheuristic (Lin and Kernighan, 1973) was proposed for the GTSP, and while runtimes were impressive, the heuristic did not provide better performance than the memetic algorithm in Gutin and Karapetyan (2010). A particle swarm based approach was also proposed in Tasgetiren et al. (2007).

Most recently, Helsgaun (2015) combined the Noon–Bean reduction and the powerful TSPsolver LKH (Helsgaun, 2000) to produce the GLKH solver. This solver improved solution quality on GTSP-Lib instances over the GK solver in Gutin and Karapetyan (2010). The improved quality does however, come at the expense of runtime, with some instances requiring more than 50 times the computation time. GLKH is also tested on several other problem libraries, including Large-Lib (proposed in Helsgaun (2015)), MOM-Lib (Mestria et al., 2013), BAF-Lib (Bontoux, 2008), and ARC-Lib[1], which include directed rural postman problems, with GLKH showing a very strong performance on these libraries.

Adaptive large neighborhood search: The GLNS solver proposed in this paper operates under the general framework of adaptive large neighborhood search (ALNS). Work in Ropke and Pisinger (2006) and Pisinger and Ropke (2007) initially proposed the idea of ALNS for pickup and delivery problems. At a high level, the idea is simple. One begins with an initial solution, and then iteratively destroys and repairs the solution. If a better solution is found, then the solution is accepted and the destroy/repair procedure is repeated until a termination condition is met. The ALNS framework has since been successfully applied to several different problems, including in two-echelon vehicle routing (Hemmelmayr et al., 2012), in capacitated vehicle routing (Ribeiro and Laporte, 2012) and recently in continuous berth allocation (Mauri et al., 2016). The two key operations in an ALNS approach to the GTSP are removing vertices from a tour and inserting new vertices back into the tour such that each set is visited. For insertions we build upon the classic TSP insertion heuristics (Rosenkrantz et al., 1977) and their extensions to the GTSP proposed initially in Fischetti et al. (1997). Our approach is related to the early work in Renaud and Boctor (1998), which proposes a GTSP solver that constructs an initial tour, followed by cheapest insertion of the remaining sets and a local optimization. We build on this idea by generalizing the insertion mechanisms to increase randomization, and operate within an ALNS framework that repeatedly re-optimizes the tour.

In the adaptive large neighborhood search framework, there are two sets (or banks) of heuristics: one for insertions and one for removals. Weights are associated with each insertion and removal heuristic, and the heuristics are selected at each destroy-repair iteration according to their weights. Weights are then updated online, such that more successful heuristics will be selected more frequently in the future. Thus, the word *adaptive* is used to refer to the mechanism of altering the heuristic weights. Adaptive large neighborhood search, and the GLNS solver, fall into a class of algorithms called hyper-heuristics (Burke et al., 2003), as they use online learning for heuristic selection (Burke et al., 2010).

Contributions: The main contribution of this paper is to present an effective algorithm for the GTSP called GLNS. We propose a novel insertion mechanism that contains as special cases nearest, farthest and random insertions from Fischetti et al. (1997). The mechanism allows for greater randomization when exploring neighbors of a given GTSP tour. Based on our benchmarking, this appears to result in a more effective search of the solution space. Similarly, we propose several new removal methods for the GTSP that generalize the idea of Shaw removal (Shaw, 1997) and worst removal (Ropke and Pisinger, 2006).

We present extensive benchmarking results, comparing the solution quality obtained by GLNS against both GLKH and GK when all solvers are given the same amount of computation time. On the classic GTSP-Lib library, where vertex sets are clustered, the GLNS shows very similar performance to that of GLKH and GK. On the existing MOM-Lib and BAF-Lib libraries, GLNS shows significant performance improvements over both solvers, improving several of the best known solutions found by GLKH. We provide two new challenging problem libraries motivated by our work in robot motion planning, called SAT-Lib and GTSP$^+$-Lib, for which the performance of GLNS is considerably better than both GK and GLKH, at times finding solutions that have an order of magnitude lower cost than the other two solvers. One of the key strengths of the GLNS solver is that it performs consistently well across the different problem libraries, and excels relative to other approaches on harder instances that are neither symmetric nor metric and do not contain clustered vertex sets. The code for the GLNS solver and the new problem libraries are available at https://ece.uwaterloo.ca/~sl2smith/GLNS.

Organization of paper: The remainder of the paper is organized as follows. In Section 2 we formally define the GTSP and the framework for the GLNS solver. In Section 3 we present the insertion heuristics used in GLNS and show how they are captured as a single unified insertion heuristic. In Section 4 we present three main removal heuristics: worst removal, distance removal, and segment removal. In Section 5.4 we give details on how the insertion and removal heuristic methods are chosen throughout the algorithm's execution, along with the local tour optimization methods implemented. In Section 6 we present independent tests of the different components in the GLNS solver on a small tuning library to show the effectiveness of each component and to justify the default GLNS settings. We provide benchmarking results using the default GLNS settings on six problem libraries: four existing libraries from the literature and two new libraries that stem from our recent work on robot motion planning languages (Imeson and Smith, 2015). We compare the performance of GLNS to the state-of-the-art solvers GLKH (Helsgaun, 2015) and GK (Gutin and Karapetyan, 2010). Finally, in Section 7 we provide conclusions and avenues for future research.

2. Problem statement and solver approach

In this section we formally define the Generalized Traveling Salesman Problem considered in this paper and provide an overview of the GLNS solver.

[1] ARC-Lib instances come from http://www.uv.es/corberan/instancias.htm.

2.1. The Generalized Traveling Salesman Problem

The exactly-one-in-a-set generalized traveling salesman problem, which we refer to simply as GTSP, can be stated as follows.

Problem 2.1 (The Generalized Traveling Salesman Problem)**.** Given a complete weighted graph $G = (V, E, w)$ on n vertices and a partition of V into m sets $P_V = \{V_1, \ldots, V_m\}$, where $V_i \cap V_j = \emptyset$ for all $i \neq j$ and $\cup_{i=1}^{m} V_i = V$, find a cycle in G that contains exactly one vertex from each set V_i, $i \in \{1, \ldots, m\}$ and has minimum length.

The GTSP is NP-hard (Noon and Bean, 1993) and contains the TSP as a special case (i.e., where $|V_i| = 1$ for each set $i \in \{1, \ldots, m\}$). There are several variations to the GTSP, including the case where the cycle must contain *at least* one vertex in each set and the case in which the sets V_i are not disjoint (Noon and Bean, 1993). There are well known reductions from both of these problems to the GTSP as stated above (Noon and Bean, 1993).

2.2. GLNS solver framework

The GLNS algorithm for the GTSP, shown in Algorithm 1, is based on the adaptive large neighborhood search framework (line comments in the algorithm indicate the section of the paper where details can be found).

Algorithm 1: GLNS(G, P_V).

```
Input: A GTSP instance (G, P_V).
Output: A GTSP tour on G.
1  for i = 1 to num_trials do
2      T ← initial_tour(G, P_V)                          // Sec.5.1
3      T_best,i ← T
4      repeat
5          Select a removal heuristic R and insertion heuristic I
           using the selection weights                    // Sec. 5.2
6          Select the number of vertices to remove, N_r, uniformly
           randomly from {1, ..., N_max}
7          Create a copy of T called T_new
8          Remove N_r vertices from T_new using R         // Sec. 4
9          For each of the N_r sets not visited by T_new, insert a
           vertex into T_new using I                      // Sec. 3
10         Locally re-optimize T_new                      // Sec. 5.5
11         if w(T_new) < w(T_best,i) then
12             T_best,i ← T_new
13         if accept(T_new, T) then                       // Sec. 5.3
14             T ← T_new
15             Record improvement made by R and I
16     until stop criterion is met                        // Sec.5.3
17     Update selection weights based on improvements of each
       heuristic over trial                               // Sec. 5.4
18 return tour T_best,i that attains min_i w(T_best,i)
```

As in adaptive large neighborhood search, we start with an initial random tour (described in Section 5.1). We repeatedly perform removals (described in Section 4) and insertions (described in Section 3), updating the scores of each removal and insertion heuristic based on their success. At each iteration we uniformly randomly select an integer number of removals N_r between 1 and $N_{\max}$, where $N_{\max}$ is a parameter of the algorithm. We keep track of the best known solution at each iteration. In line 13 of Algorithm 1 we accept or decline the modified tour T_{new} based on a standard simulated annealing criterion, as in Ropke and Pisinger (2006) (details are in Section 5.3). The stopping criteria in 16 operates in two phases (detailed in Section 5.3). The first phase is an initial descent, which stops after a fixed number of non-improving iterations. The second phase consists of several warm restarts, each beginning with the best solution found, but with a lower simulated annealing temperature. Each warm restart again ends after a fixed number of non-improving iterations. The warm restarts serve to refine the tour found after the initial descent.

Two key components of GLNS are the bank of methods used for insertions and deletions of vertices in the tour. Other unique aspects are the presence of multiple trials, which are used to adapt the weights, and the use of local optimization on each tour T_{new} at line 10 prior to comparing the tour length to the best tour of the iteration $T_{\text{best},\,i}$ and the previous tour T. The mechanism for adapting weights is described in Section 5.4 and the local optimizations are detailed in Section 5.5. At the completion of the algorithm, the best tour found is returned.

We provide three default settings of our solver. Under each setting, $O(n^2)$ computation time is required to parse the input instance. Given the parsed instance, the three settings have runtimes scaling approximately as $O(nm)$ for the fastest setting and $O(\max\{mn^2,\ m^3 n \log m\})$ for the slowest (see Section 6.3 for more details). The parameter values for the three settings are detailed in Section 6.2. Since the removal methods are based on the same general concepts as the insertion methods, we begin by presenting insertions (Section 3), followed by removals (Section 4).

3. GTSP insertion heuristics

In this section we begin by summarizing four insertion heuristics that are proposed in Fischetti et al. (1997) as extensions of the well-known TSP insertions (Rosenkrantz et al., 1977). We then provide a unified insertion mechanism that contains three of the four insertions as special cases, and give an analysis of the space and time complexity of these insertion heuristics. The unified insertion allows for a large bank of insertion heuristics that can be used in line 9 of Algorithm 1.

3.1. Nearest, farthest, random, and cheapest insertion

Given a graph $G = (V, E, w)$ along with the partition $P_V := \{V_1, \ldots, V_m\}$ of V, we define a partial GTSP tour to be a cycle in G such that each set in the partition P_V is visited at most once (in a complete tour, each set is visited exactly once). Given a partial tour $T = (V_T, E_T)$, the sets of the partition that are visited by T are denoted $P_T \subseteq P_V$. The framework of an insertion heuristic is given in Algorithm 2 .

Algorithm 2: Framework of insertion heuristics.

```
Input: A GTSP instance (G, P_V) and a partial tour T = (V_T, E_T)
       of G
Output: An updated partial tour T that visits one additional
        set.
1 Pick a set V_i in P_V \ P_T.
2 Find an edge (x, y) ∈ E_T and vertex v ∈ V_i thatminimizes
  w(x, v) + w(v, y) − w(x, y).
3 Delete the edge (x, y) from E_T, add the edges (x, v)and (v, y)
  to E_T, and add v to V_T.
4 return T
```

What remains is to specify the mechanism for choosing the set to insert (V_i in $P_V \backslash P_T$). For every set V_i, for $i \in \{1, \ldots, m\}$ and each vertex $u \in V \backslash V_i$ we compute the distance

$$\text{dist}(V_i, u) = \min_{v \in V_i} \left\{ \min \left\{ w(v, u), w(u, v) \right\} \right\}. \tag{1}$$

In Rosenkrantz et al. (1977), four insertion heuristics were proposed for iteratively constructing a TSP tour: nearest, farthest, random, and cheapest insertion. In Fischetti et al. (1997, Section 4), they proposed extensions of the TSP insertion heuristics (Rosenkrantz et al., 1977) to the GTSP as follows:

(i) **Nearest insertion** picks the set V_i that contains a vertex v at minimum distance to a vertex on the partial tour T. That is, we choose the set V_i

$$\operatorname*{argmin}_{V_i \in P_V \setminus P_T} \min_{u \in V_T} \operatorname{dist}(V_i, u).$$

(ii) **Farthest insertion** picks the set V_i whose closest vertex to a vertex on the partial tour T is maximum. That is, we choose the set V_i

$$\operatorname*{argmax}_{V_i \in P_V \setminus P_T} \min_{u \in V_T} \operatorname{dist}(V_i, u).$$

(iii) **Random insertion** picks a set V_i uniformly randomly from $P_V \setminus P_T$.

(iv) **Cheapest insertion** picks the set V_i that contains the vertex v that minimizes the insertion cost. That is, we choose the set

$$\operatorname*{argmin}_{V_i \in P_V \setminus P_T} \min_{v \in V_i,\ (x,y) \in E_T} \big\{ w(x, v) + w(v, y) - w(x, y) \big\}.$$

Remark 3.1 (Distance from Set to Tour)**.** In Fischetti et al. (1997), the distance was proposed as $\operatorname{dist}(V_i, u) = \min_{v \in V_i} w(v, u)$ rather than the form in (1). For symmetric instances, the distance in (1) is equivalent. For asymmetric instances, by minimizing $\min\{w(v, u), w(u, v)\}$, we find the set containing the vertex at minimum distance either "to" or "from" the tour. In our benchmarks of asymmetric instances, we have observed better performance using the distance in (1). •

Remark 3.2 (Relation to TSP insertions)**.** For TSP instances, each V_i contains exactly one vertex, and the four insertion heuristics are exactly those proposed in Rosenkrantz et al. (1977). Starting with an initial vertex, each heuristic can be used to create a complete tour by performing $n - 1$ insertions. For metric instances, it is shown in Rosenkrantz et al. (1977) that nearest and cheapest insertions provide 2-factor approximations to the optimal tour, while farthest and random insertions provide $\lceil \log_2 |V| \rceil + 1$-factor approximations. The authors also note that farthest insertion often outperforms cheapest and nearest insertion. In addition, random insertion runs more quickly in practice than farthest or nearest insertion (by a constant factor) since no computation is needed to select the next vertex to insert. •

3.2. Unified insertion heuristic

The nearest, farthest, and random GTSP insertion heuristics can be unified into a single insertion heuristic as follows. Given a partial tour T, there are $\ell = |P_V \setminus P_T|$ sets that can be inserted next. From line 9 of Algorithm 1 we have that $\ell \in \{1, \dots, N_r\}$. For each set $V_i \in P_V \setminus P_T$ we define the minimum distance between V_i and T as

$$d_i = \operatorname{dist}(V_i, T) = \min_{u \in V_T} \operatorname{dist}(V_i, u), \tag{2}$$

where $\operatorname{dist}(V_i, u)$ is defined in (1). Given a parameter $\lambda \in [0, \infty)$ and a partial tour, we select a new set (Line 1 of Algorithm 2) according to the procedure in Algorithm 3 . Note that in line 1 of Algorithm 3, we use the common convention that $0^0 = 1$.

Once a set is selected, the unified insertion heuristic proceeds as in Algorithm 2. Notice that,

(i) random insertion is obtained by setting $\lambda = 1$
(ii) nearest insertion is obtained setting $\lambda = 0$;

Algorithm 3: Set selection for the unified insertion heuristic.

Input: A GTSP instance (G, P_V), a partial tour $T = (V_T, E_T)$ of G, and $\lambda \in [0, \infty)$
Output: A set $V_i \in P_V \setminus P_T$.
1 Randomly select $k \in \{1, \dots, \ell\}$ according to the unnormalized probability mass function$[\lambda^0, \lambda^1, \dots, \lambda^{\ell-1}]$.
2 Pick the set $V_i \in P_V \setminus P_T$ with the kth smallest distance d_i to the tour (where d_i is computed in (2)).
3 **return** V_i

(iii) farthest insertion is obtained setting $\lambda = +\infty$ (or sufficiently large).

Moreover, by selecting intermediate values for λ, we can obtain randomized versions of nearest, farthest, and random insertion, allowing for greater exploration of solutions during large neighborhood search. In Section 6.2 we show the performance improvements that result from using the unified insertion mechanism with several different λ values over that of using just nearest, farthest, and random insertion.

Remark 3.3 (Similarities/differences with `softmax`)**.** The mechanism for choosing a set is similar to softmax (Sutton and Barto, 1998), in which λ is fixed, and the set V_i is chosen with probability proportional to $\exp(\lambda d_i)$. The main issue with softmax is that the resulting probability mass function over sets is dependent on both the absolute and relative magnitudes of each d_i. This means that the value of λ should be scaled for each GTSP problem instance, based on some criterion of the distribution and values of edge lengths in the graphs. The proposed heuristic avoids this issue by making the selection purely on the ordering of the distances (i.e., choosing the kth smallest), rather than their specific values. •

3.3. Implementation details, runtime, and Sspace complexity of insertions

The following list provides details of how the unified insertion heuristic is implemented in GLNS, and allows us to characterize the complexity of the approach.

(i) The normalizing constant for the distribution in line 1 of Algorithm 3 has a closed form expression of $(1 - \lambda^\ell)/(1 - \lambda)$, since $\sum_{k=0}^{\ell-1} \lambda^k$ is a geometric series. As such, k can be drawn from the distribution in $O(\ell)$ time.
(ii) Given an unsorted array of ℓ numbers, selection of the kth smallest value can be performed in expected time of $O(\ell)$ using a simple randomized algorithm (Cormen et al., 2009, Chapter 9).
(iii) At the beginning of an execution of the GLNS solver, we precompute the distances in (1) between each set-vertex pair. This computation requires $O(n^2)$ time and $O(nm)$ space, where n is the number of vertices and m is the number of sets. This computation is performed while parsing the input instance, which does not alter the $O(n^2)$ parsing runtime.
(iv) When performing multiple insertions, we can efficiently update the values d_i in (2) after each insertion. If we have the distance d_i for vertex set $V_i \in P_V \setminus P_T$ to a partial tour T and a vertex $v \in V_j$ is inserted into T, then d_i can be updated as

$$d_i \leftarrow \min\{d_i, \operatorname{dist}(V_i, v)\}.$$

Thus, the initial computation of $d_1, \dots, d_\ell$ requires $O(\ell(m - \ell))$ time, and after subsequent insertions, each distance can be updated in constant time, giving $O(\ell)$ time to update all distances.

Table 1
The runtime of each GTSP insertion heuristic for inserting N_r sets into a partial tour T for a GTSP instance with n vertices and m sets. The second set of columns is for the case that each set $|V_i| \in O(n/m)$. * is expected runtime.

Insertion heuristic	General GTSP input		Each $\|V_i\| \in O(n/m)$	
	Runtime	Space	Runtime	Space
Unified*	$O(nm)$	$O(N_r)$	$O(nN_r)$	$O(N_r)$
Nearest	$O(nm)$	$O(N_r)$	$O(nN_r)$	$O(N_r)$
Farthest	$O(nm)$	$O(N_r)$	$O(nN_r)$	$O(N_r)$
Random	$O(nm)$	$O(1)$	$O(nN_r)$	$O(1)$
Cheapest	$O(\max\{nm, nN_r \log m\})$	$O(nm)$	$O\big(\max\{nN_r, \frac{N_r^2 n}{m} \log m\}\big)$	$O(nN_r)$

Given a parsed GTSP instance, the following proposition characterizes the runtime of the unified insertion heuristic for any $\lambda \in [0, \infty)$. In this proposition we consider both general GTSP instances, and instances in which each set contains $O(n/m)$ vertices. The latter instances capture the case where vertices are distributed "approximately evenly" among the sets (i.e., their sizes are equal up to a constant factor).

Proposition 3.4 (Runtime of insertion mechanisms)**.** *Given a GTSP instance (G, P_V) and a partial tour T, the insertion of the $N_r = |P_V \setminus P_T|$ sets into T has an expected runtime of $O(mn)$ using the unified insertion mechanism. Moreover, if $|V_i| \in O(n/m)$ for each set $V_i \in P_V$, then the expected runtime is $O(N_r n)$. The space complexity of the insertion heuristic is $O(N_r)$.*

Proof. Given N_r sets to insert, the initial computation of the distances d_i in (2) requires $O(N_r(m - N_r)) = O(N_r m)$ computation time (this is performed only for the first insertion). Given these distances, Algorithm 3 runs in expected time of $O(N_r)$ using the randomized selection in Cormen et al. (2009, Chapter 9), and the distances d_i can be updated in $O(N_r)$ time after each subsequent insertion (by the implementation details above). Thus, the total expected runtime for N_r set selections is $O(N_r m)$.

Once the first set V_i is selected, Line 2 of Algorithm 2 requires $O(|V_i|(m - N_r)) \in O(|V_i|m)$ computation time to evaluate the insertion of each vertex in V_i in each of the $m - N_r$ positions on T. For ease of notation, we renumber the sets in $P_V \setminus P_T$ such that they are inserted in the order $V_1, \ldots, V_{N_r}$. Then, the total computation time in Line 2 over the N_r insertions is

$$\sum_{k=1}^{N_r} O\big(|V_k|m\big) = O\Big(m \sum_{k=1}^{N_r} |V_k|\Big). \tag{3}$$

What remains is to bound the quantity $\sum_{k=1}^{N_r} |V_k|$. In the worst-case we have $\sum_{k=1}^{N_r} |V_k| = O(n)$ and the total runtime is $O(nm)$. If each set in $V_i \in P_V \setminus P_T$ has size in $O(n/m)$, then $\sum_{k=1}^{N_r} |V_k| = O(nN_r/m)$, and the total runtime from (3) is $O(N_r n)$. □

Table 1 summarizes the runtime for the unified insertion, as well as the runtime for each of the four insertion mechanisms in Section 3.1. The runtimes for nearest, farthest, and random insertion follow from the analysis in the proof of Proposition 3.4, and are simple extensions of the runtimes in Rosenkrantz et al. (1977). The key point of this table is that the unified insertion maintains the same (expected) runtime as nearest, farthest, and random insertion, with only moderate space requirements.

Remark 3.5 (TSP insertion runtimes Rosenkrantz et al., 1977)**.** For a TSP instance $|V| = n = m$. A complete tour can be computed in $O(n^2)$ using nearest, farthest, or random insertion, and $O(n^2 \log n)$ using cheapest insertion. •

The implementation details of cheapest insertion are outlined in the following remark.

Remark 3.6 (Implementing cheapest insertion)**.** Given N_r sets to insert, cheapest insertion can be naïvely implemented to run in $O(nmN_r)$ time, with $O(1)$ space. A more efficient implementation of cheapest insertion requires significantly more space (Rosenkrantz et al., 1977). Given a partial tour T, for each vertex v in a set in $P_V \setminus P_T$ we maintain a min-priority queue (implemented as a min binary heap), giving the insertion cost for each edge $e \in E_T$. The runtime needed to construct these queues, and the space complexity, is $O(nm)$. When a new vertex is added to the tour T, we add two new edges and remove one edge, along with their insertion costs, from each queue. Each insertion/removal can be performed in $O(\log m)$ time for each priority queue, since each queue contains $O(m)$ edges. There are $O(n)$ vertices contained in the sets in $P_V \setminus P_T$, and the total time to update all priority queues is $O(n \log m)$.

The vertex to insert is found in $O(n)$ taking the minimum of the min elements in each priority queue. The time to insert the next vertex into T is $O(n \log m)$, and the overall complexity to create the priority queues and insert N_r sets is $O(\max\{nm, nN_r \log m\})$. The space complexity is dominated by the space to store the priority queues, which is $O(nm)$. Finally, the results for $|V_i| \in O(n/m)$ follow by noting that there are $O(N_r n/m)$ vertices contained in the sets in $P_V \setminus P_T$ instead of $O(n)$. •

3.4. Bounding insertions before evaluation

A simple and effective mechanism we have found for speeding up insertions is to use the precomputed distances $\text{dist}(V_i, u)$ between each $V_i \in P_V \setminus P_T$ and each vertex $u \in V_T$ to lower bound insertion costs. In particular, in Line 2 of Algorithm 2, prior to checking the insertion cost for each vertex $v \in V_i$ in each edge $(x, y) \in E_T$, we compute

$$\texttt{lb} = \text{dist}(V_i, x) + \text{dist}(V_i, y) - w(x, y).$$

If `lb` is at least as large the minimum insertion cost found for a vertex $v \in V_i$ so far, then the insertion costs for the edge (x, y) do not need to be computed, since they cannot be smaller than `lb`. Experimental results in Section 6.2 show that this simple lower-bounding technique reduces the runtime of GLNS by 25% on average.

3.5. Randomizing insertions through noise and subsetting

The performance of large neighborhood search algorithms are improved by increasing randomness during removals and insertions (see for example Ropke and Pisinger (2006) or Pisinger and Ropke (2007)), which helps in exploring a larger portion of the solution space and in avoiding repetition of locally optimal choices. The λ parameter in the unified insertion mechanism allows us to introduce randomness into the set selection. To add randomness to the choice of vertex from the selected set in Line 2 of Algorithm 2, we use the following two mechanisms.

Additive noise. Motivated by Ropke and Pisinger (2006), given a maximum noise level $\eta \geq 0$, we perform the following: For each vertex $v \in V_i$ and edge $(x, y) \in E_T$, we generate a uniform random number $\texttt{rand} \in [0, \eta]$ and compute the insertion cost as

$$(1 + \texttt{rand})\big(w(x, v) + w(v, y) - w(x, y)\big).$$

We then perform the insertion with minimum "noisy" cost. Note, in Ropke and Pisinger (2006), the magnitude of the additive term was based on the maximum edge cost in the problem instance. We have modified this notion by scaling each additive noise term by the corresponding insertion cost, rather than the maximum edge. The reason for this is that many of our GTSP instances contain both "infinite" (i.e., large cost) and low-cost edges. These large variations make it difficult to use a single scaling constant for the noise, and we have found that the proposed scaling is more robust to these large variations in edge costs.

Insertion subsetting. A second method for adding randomness is to consider only a random subset of the possible insertion combinations. Given a parameter $f \in (0, 1)$, we uniformly randomly select a subset $\bar{E}_T \subset E_T$ of edges of cardinality $\lceil f|E_T|\rceil$ and evaluate the insertion cost $w(x, v) + w(v, y) - w(x, y)$ for all $v \in V_i$ and all $(x, y) \in \bar{E}_T$. This allows for a vertex to be inserted in a position that is not locally optimal, widening the search.

Our experimental results in Section 6.2 show that while the additive noise improves performance, the insertion subsetting does not on average improve solution quality of the GLNS solver (it does however, moderately improve the runtime). Thus, the default GLNS settings include only additive noise.

4. Removal heuristics

In each iteration of GLNS, we use a removal heuristic to remove N_r vertices from a tour $T = (V_T, E_T)$ (Line 8 of Algorithm 1). Two of the three removal methods operate in a similar manner to the unified insertion, incrementally removing the N_r sets. Consider a partial tour T containing $\ell \in \{m - N_r + 1, \ldots, m\}$ vertices, where for simplicity of notation the vertices are numbered such that $V_T = \{v_1, \ldots, v_\ell\}$ and $E_T = \{(v_1, v_2), (v_2, v_3), \ldots, (v_\ell, v_1)\}$. Then, for a fixed parameter λ and a set of distances r_j for each $v_j \in V_T$ (the computation of these distances is specified for each removal heuristic in the following two subsections), the general framework is specified in Algorithm 4 .

Algorithm 4: Removal heuristic framework for a given λ and distance metric r_j.

Input: A partial tour $T = (V_T, E_T)$, $\lambda \in [0, \infty)$, and values r_j for each $v_j \in V_T$
Output: A new tour with one vertex removed from V_T.
1 Randomly select $k \in \{1, \ldots, \ell\}$ according to the unnormalized probability mass function$[\lambda^0, \lambda^1, \ldots, \lambda^{\ell-1}]$, where $\ell = |V_T|$.
2 Pick the vertex $v_j \in V_T$ with the kth smallest value r_j.
3 Remove v_j from V_T
4 Remove (v_{j-1}, v_j) and (v_j, v_{j+1}) from E_T and add (v_{j-1}, v_{j+1}) to E_T.
5 **return** T

In line 4, the indices $j + 1$ and $j - 1$ are evaluated with the understanding that $\ell + 1 \equiv 1$ and $0 \equiv \ell$. That is, for $j = 1$, we have $(v_{j-1}, v_j) = (v_\ell, v_1)$, and with $j = \ell$, we have $(v_j, v_{j+1}) = (v_\ell, v_1)$.

4.1. Unified worst removal

In worst removal (see Ropke and Pisinger (2006) for an example), given a partial tour T with $V_T = \{v_1, \ldots, v_\ell\}$ we remove the vertex v_j that maximizes the removal cost

$$r_j = w(v_{j-1}, v_j) + w(v_j, v_{j+1}) - w(v_{j-1}, v_{j+1}),$$

where the indices $j + 1$ and $j - 1$ are evaluated using $\ell + 1 \equiv 1$ and $0 \equiv \ell$. That is, we remove the vertex that results in the greatest reduction in tour length, since it is likely to be misplaced in the tour. In the unified worst removal heuristic, we extend this idea by using the removal costs r_j in the framework of Algorithm 4. This creates a suite of removal heuristics, parametrized by λ. In particular,

(i) $\lambda = 1$ corresponds to random removal, where N_r vertices are uniformly randomly chosen from V_T for removal; and
(ii) $\lambda = \infty$ corresponds to worst removal.

For values of $\lambda \in (1, \infty)$ we obtain randomized versions of worst removal with varying degrees of randomization, similar to the randomization method proposed in Ropke and Pisinger (2006). Each removal can be performed in $O(m)$ time, and thus N_r removals require $O(N_r m)$ time.

4.2. Distance removal

The idea behind distance removal is to remove vertices from the tour that are "close" to one another. This is similar to Shaw removal for the pickup and delivery problem Shaw (1997). Starting with a complete tour $T = (V_T, E_T)$, we begin by randomly removing a vertex from T and adding it to a set V_{removed}. Then, at each iteration of the removal, the distances r_j are computed as follows:

(i) uniformly randomly select a seed vertex v_{seed} from V_{removed}, and
(ii) for each $v_j \in V_T$, compute r_j as

$$r_j = \min\{w(v_{\text{seed}}, v_j), w(v_j, v_{\text{seed}})\}.$$

We then perform the removal as in Algorithm 4 with the distances $r_1, r_2, \ldots, r_\ell$, where $\ell = |V_T|$. This process is repeated until the desired number N_r of vertices have been removed from T. With $\lambda = 0$ we always pick the closest vertex v_j to the random seed. With $\lambda = 1$, the algorithm becomes a simple random removal. As with the unified worst removal, the runtime to perform N_r removals is $O(N_r m)$.

4.3. Segment removal

In segment removal, we simply remove a continuous segment of the tour of length N_r. Given a complete tour T with vertices $V_T = \{v_1, \ldots, v_m\}$, we uniformly randomly select a vertex v_j and then remove the vertices $v_j, v_{j+1}, \ldots, v_{j+N_r-1}$ from the tour (where the indices wrap around, or more formally, each index is replaced with the unique value in $\{1, \ldots, m\}$ that is congruent modulo m). The motivation for this removal is to try to escape deep local minima by completely destroying large segments of the tour.

5. Adaptive weights, local optimization, and acceptance criteria

In this section we provide details on the remaining aspects of GLNS (Algorithm 1). In particular, we describe the initial tour construction (Line 2), the mechanism for selecting removal and insertion heuristics (Line 5), the method for adapting weights (Line 17), the acceptance and stopping criteria (Lines 13 and 16) and the local tour optimizations (Line 10).

5.1. Initial tour construction

We use the following two methods for initial tour construction.

Table 2
GLNS parameter values for the three solver settings: fast, medium, and slow.

Parameter	Symbol	Solver setting		
		Fast	Medium	Slow
Stopping criteria				
Number of trials	`num_trials`	3	5	10
Number of warm restarts	`num_warm`	2	3	5
Number of iterations	`num_iterations`	$60m$	$60m$	$150\,m$
Warm iterations: first improvement	`first_improve`	$10m$	$15m$	$25m$
Warm iterations: last improvement	`last_improve`	$15m$	$30m$	$50\,m$
Acceptance criteria				
Initial acceptance (%)	p_1	5.00	5.00	5.00
Warm restart acceptance (%)	p_3	0.50	0.50	0.50
Final acceptance (%)	p_2	0.05	0.05	0.05
Reaction factor	ϵ	0.5	0.5	0.5
Tour optimizations				
Initial tour construction	T_{init}	Rand insert	Random	Random
Maximum removals	$N_{\max}$	$\min\{20,\ 0.1m\}$	$\min\{100,\ 0.3m\}$	$0.4m$
Re-Opt	–	No	Yes	Yes
Iterations of move-opt	N_{move}	$N_{\max}$	$N_{\max}$	$N_{\max}$
Insertion/Deletion				
Cheapest insertion	–	No	Yes	Yes
Insertion λ values	–	$(0,\ 1/2,\ 1/\sqrt{2},\ 1,\ \sqrt{2},\ 2,\ \infty)$		
Distance removal λ values	–	$(1/\sqrt{2}, 1,\ \sqrt{2},\ 2,\ \infty)$		
Worst removal λ values	–	$(1/\sqrt{2},\ 1,\ \sqrt{2},\ 2,\ \infty)$		
Additive noise levels	η	$(0, 0.25, 0.75)$		

Random insertion tour. In random insertion we start by choosing a random vertex $v \in V$, and add it to an empty tour T. We then insert the remaining $m-1$ vertex sets using the unified insertion heuristic with $\lambda = 1$ (i.e., where each set is chosen at random, followed by insertion as in Algorithm 2), and with additive noise using the largest additive noise level η from Table 2. This construction takes $O(nm)$ time.

Random tour. In the random tour construction we create an initial tour completely randomly. We shuffle the m vertex sets and then uniformly randomly select one vertex from each set in the shuffled order to create a tour. This construction requires $O(m)$ time.

When a sufficient number of GLNS iterations are used, we have found little difference in performance between these two initial tour construction methods (see Section 6.2 for more details). However, when time (and thus the number of iterations) are constrained, it can be helpful to start at the higher quality initial tour created through the random insertion heuristic.

5.2. Choosing removal and insertion heuristics

We maintain a bank of insertion and removal heuristics. For the insertion bank, we use the unified insertion with several different configurations of λ and η (λ is the set selection parameter and η is the additive noise parameter). Each configuration pair of (λ_i, η_i) is considered as a separate insertion mechanism in the bank of insertion heuristics. We also use cheapest insertion in the insertion the bank (there are no configuration parameters for this method, so it appears only once in the bank).

Removals are handled in a similar manner. We use several configurations (λ) for both unified worst removal and distance removal. Segment removal is also used in the bank (there are no configuration parameters for this method, so it appears only once in the bank).

For selection purposes, we maintain a weight for each insertion and removal heuristic, with all weights initialized to 1. At each iteration, we use a standard roulette wheel selection mechanism (Ropke and Pisinger, 2006) to select a removal and an insertion heuristic from the corresponding banks according to their weights. The adaptive mechanism for updating weights is described in Section 5.4.

5.3. Acceptance and stopping criteria

The acceptance criteria and the stopping criteria used in GLNS in Algorithm 1 are as follows.

Acceptance criteria. We use a standard simulated annealing acceptance criterion, where given a temperature $\mathcal{T}$, the new tour T_{new} is accepted with probability $\min\left\{\exp\left(\frac{w(T)-w(T_{\text{new}})}{\mathcal{T}}\right), 1\right\}$. Note, if $w(T_{\text{new}}) \leq w(T)$ then the tour T_{new} is accepted with probability one (thus, improving tours are always accepted). The temperature $\mathcal{T}$ is initialized to some value $\mathcal{T}_{\text{init}}$ and is decreased at every iteration as $\mathcal{T} \leftarrow c\mathcal{T}$ for some cooling rate $c < 1$. The implementation of our simulated annealing procedure along with the stopping criteria for each trial includes several parameters, described as follows (their values are described in Section 6.2):

(i) Following the strategy in Ropke and Pisinger (2006), we initialize the temperature such that in the first trial, a tour with p_1% higher cost than the initial tour (p_1 is an solver parameter) is accepted with a probability of 1/2.
(ii) For subsequent trials, the initial temperature is set in the same way, using the best tour cost from the previous trial. Since in the first trial the initial tour is constructed as in Section 5.1, the initial temperature in the first trial is typically much larger than in subsequent trials.
(iii) The cooling rate is chosen such that after `num_iterations` iterations, a tour with p_2% higher cost is accepted with a probability of 1/2.

Stopping criteria. Each of the `num_trials` trials has two phases: an initial descent followed by several warm restarts. The stopping criteria for these phases are as follows:

(i) The initial descent ends when the best tour has not improved for `last_improve` (an input parameter) consecutive iterations.
(ii) Each warm restart begins with the best tour found during the trial, but the temperature is raised to allow for a tour with p_3% (a solver parameter) higher cost to be accepted with a probability of 1/2. The warm restart ends if no improvement is made to the best solution after the first `first_improve` iterations.

(iii) If there is an initial improvement in a warm restart, then as with the initial descent, the warm restart terminates when no further improvement is made for `last_improve` consecutive iterations.
(iv) In each trial, warm restarts continue until no improvement has been made to the best tour in `num_warm` consecutive warm restarts.

Alternate stopping criteria. In the GLNS solver we provide two alternate stopping criteria: 1) the solver can terminate if a maximum time is reached, or 2) the solver can terminate if a tour less than a given bound is found.

5.4. Adaptive weights

The weight of each insertion and removal heuristic is adapted at the end of each trial based on its score for the trial. When recording the performance of each heuristic (used to calculate the scores), we split a trial into three phases:

(i) **early**, consisting of the first `num_iterations`/2 iterations;
(ii) **mid**, consisting of the remaining iterations of the initial descent, and
(iii) **late**, consisting of the warm restarts.

In each iteration, a removal and insertion heuristic is used to destroy and create the tour T_{new} from the tour T. As in Ropke and Pisinger (2006), we record the same score for the insertion and the removal, calculating the score for the iteration as

$$\texttt{score} = \max\left\{\frac{w(T) - w(T_{\text{new}})}{w(T)}, 0\right\}.$$

This score gives the fractional improvement in tour cost, where the max ensures that we do not penalize a heuristic when it increases the tour cost. At the end of a trial, the overall score for a heuristic is given by the sum of its scores, divided by the number of times it was used (i.e., the average score). We then update the weight of each heuristic as ϵ times the previous weight plus $1-\epsilon$ times the average score on the trial.

Remark 5.1 (Comparison with adaptive weights in Ropke and Pisinger (2006))**.** For the GLNS solver, the advantage to this method over the adaptation proposed in Ropke and Pisinger (2006) is two-fold. First, in Ropke and Pisinger (2006) the weights are updated every 100 iterations, where 100 is referred to as the segment_length. This works well when there are a small number of insertion and deletion heuristics, but in our case the unified insertion and removal mechanisms allow us to create a large bank of insertion and removal heuristics. Thus, to gather a statistically significant score for each insertion and removal, we require a sufficiently large segment length. Moreover, as the weights of some heuristics are reduced, their sample size over a segment is also decreased, increasing the variance of subsequent scores. To avoid these issues we simplify the procedure to update after a complete trial is performed instead of after each segment. Second, our implementation depends on just one parameter ϵ and the definitions of early, mid, and late, rather than the five parameters (σ_1, σ_2, σ_3, r, segment_length) in Ropke and Pisinger (2006). •

5.5. Local tour optimizations

After each iteration of GLNS we locally optimize the new tour prior to evaluating the acceptance condition. We use two standard local optimization techniques, which are applied cyclically until there is no further improvement in tour cost.

Re-optimize vertex in each set (Re-Opt). This method re-optimizes the vertex in each set, keeping the ordering of the sets fixed. It has been proposed in Fischetti et al. (1997, Section 4) and Renaud and Boctor (1998) and is used as a subroutine in several GTSP solvers (Gutin and Karapetyan, 2010; Helsgaun, 2015). Given an ordering of the vertex sets, a directed acyclic graph (DAG) is created containing all vertices in the GTSP graph, plus a copy of each vertex in the first set of the ordering. Each vertex in a set is connected to all vertices in the next set of the ordering. Vertices in the last set in the ordering are connected to those in the copy of the first set. A tour satisfying the ordering is then a path in the DAG from a vertex in the first set to its copy. The graph contains $O(n)$ vertices and $O(n^2)$ edges. Given a start and end vertex, a shortest path in a DAG can be computed in $O(n+n^2) = O(n^2)$ time (Cormen et al., 2009). For our DAG, we must calculate a shortest path for each vertex in the first set of the ordering to its copy. The tour can be rotated such that the first set contains at most n/m vertices, and thus the total runtime is $O(n^3/m)$. If there is a vertex set with $O(1)$ vertices, then the total runtime is $O(n^2)$, and if in addition each vertex set contains $O(n/m)$ vertices, then the runtime is $O(n^2/m)$.

Move-opt. This method attempts to optimize the ordering of the sets. Given a complete tour $T = (V_T, E_T)$, the second local optimization technique randomly selects a vertex v in the tour T, removes it, and then reinserts a vertex from the same set V_i with minimum insertion cost. That is, it performs the insertion that minimizes $w(x,u) + w(u,y) - w(x,y)$ for all $u \in V_i$, and all $(x, y) \in E_T$. The procedure is repeated N_{move} times. Since the expected number of vertices in a randomly chosen vertex set is n/m, the expected runtime of this method is $O(N_{move}n)$.

Move-opt can be thought of as a special case of GLNS in which only one set is removed from the tour. This method is called "Inserts" in Gutin and Karapetyan (2010) and the "Move" operator in Bontoux et al. (2010). For this method, we implement the lower bounding technique in Section 3.4, which substantially improves the runtime.

6. Experimental results

The GLNS solver has been implemented in the language Julia[2] and is freely available at https://ece.uwaterloo.ca/~sl2smith/GLNS. The core solver is implemented in approximately 1000 lines of code, which is less than 1/4 of the code in the GK solver and less than 1/15 of the code in the GLKH solver. We also provide a parser that handles instances in the GTSP-Lib format Fischetti et al. (1997). The solver runs on a single core and does not use any parallelization. The solver can be run both through the Julia REPL or via the command line. Several flags are available to set the key parameters shown in Table 2 and to provide a solver timeout or a desired tour cost at which the solver should terminate.

In this section we summarize the tuning procedure for GLNS and our extensive benchmarking results. The tuning results also demonstrate the performance improvements obtained by several of the novel components of the GLNS solver. We use our benchmarking results to compare performance with the GLKH solver from Helsgaun (2015) and the GK solver from Gutin and Karapetyan (2010) on six problem libraries as described in the following section.

6.1. Problem libraries

We benchmark the performance of the GLNS solver on six GTSP problem libraries: four existing GTSP libraries (GTSP-Lib, MOM-Lib,

[2] Julia is a high-level, high-performance programming language available at http://julialang.org/.

Table 3
Tuning results. Performance of GLNS solver under 13 different configurations, each including/excluding components of solver. Performance is given by the number of best known solutions (# B) and the average gap to best known (Δ), or for SAT-Lib, the number of best known solutions (# B) and the average number of infinite cost edges (Avg.). Entries show the rank of the configuration on the corresponding library, from 1 (best) to 13 (worst), including ties.

Rank	Description	GTSP-Lib		BAF-Lib		GTSP$^+$-Lib		SAT-Lib		Overall	
		Δ	# B	Δ	# B	Δ	# B	Avg.	# B	Δ (Avg.)	# B
1	Default GLNS Configuration	4	2	2	2	4	3	1	2	1	2
1	Random insertion T_{init}	1	1	1	1	10	5	2	1	2	1
3	Nearest, farthest, random	2	7	4	5	7	7	4	3	3	5
4	No adaptive weights	7	3	3	6	2	2	12	8	6	3
4	No cheapest insertion	5	4	5	8	3	1	8	7	5	4
6	No noise	3	6	13	10	1	9	3	3	4	8
7	No local optimizations	8	8	6	6	5	7	5	6	6	7
8	Subset and additive noise	9	5	9	9	11	6	7	5	9	6
9	Only subset noise	6	9	8	4	6	10	11	8	8	9
10	No restarts	10	10	7	3	9	11	10	10	9	10
11	Worst and random removal	12	11	11	12	12	4	6	10	11	11
12	One trial	13	13	10	11	8	13	13	10	12	13
12	Settings 3 and 11	11	12	12	13	13	11	9	10	13	12

BAF-Lib and Large-Lib) and two new GTSP libraries (GTSP$^+$-Lib and SAT-Lib) that provide instances on which solvers show larger performance gaps relative to the best known solutions. The instances in SAT-Lib and GTSP$^+$-Lib are subsets of the instances proposed in Imeson and Smith (2015) for the SAT-TSP planning language.

GTSP-Lib (Fischetti et al., 1997; Silberholz and Golden, 2007) consists of instances created from TSP-Lib, where vertex sets are created using a clustering procedure from Fischetti et al. (1997). Instances contain up to 1000 vertices.

MOM-Lib (Mestria et al., 2013) consists of clustered Euclidean traveling salesman problem instances with up to 2000 vertices.

BAF-Lib (Bontoux, 2008) consists of the same underlying instances as GTSP-Lib, but where the vertex sets are created psuedorandomly.

Large-Lib (Helsgaun, 2015) consists of very large GTSP instances created primarily from TSP-Lib. Sets are created using the clustering procedure from Fischetti et al. (1997).

GTSP$^+$-Lib consists of instances created as follows. A Euclidean instance from GTSP-Lib is chosen. Then, additional constraints are placed between randomly chosen pairs of vertices in separate sets: each constraint takes either the form $v_1 \Rightarrow v_2$ or "not both v_1 and v_2." Thus, if v_1 is visited in the tour, then the two constraints express that either v_2 must be in the tour, or v_2 cannot be in the tour. The constrained GTSP instance is then reduced to a standard GTSP instance using the reduction in Imeson and Smith (2015). This reduction uses infinite cost edges (set to a cost of 999,999) to encode the extra constraints between pairs of vertices. These instances are motivated by path planning problems (Imeson and Smith, 2015), where a robot (vehicle) must visit a set of locations, with additional constraints on the compatibility between locations. For this library, we also score the solvers on percentage of time that a feasible solution is found (i.e., a solution with no infinite cost edges). Instance names in this library indicate the original GTSP-Lib instance and the number of additional constraints x expressed as a percentage of the number of vertices in the instance. For example, the instance 53gil262_× 35 contains $0.35 \times 262 = 92$ additional constraints. Moreover, the constraints are added incrementally, so that 53gil262_× 30 contains a subset of the constraints in 53gil262_× 35, and so on.

SAT-Lib consists of instances produced from SAT-Lib (Hoos and Stiitzle, 2000) by performing a reduction from SAT to GTSP (only satisfiable SAT instances are used). For each instance in this library, the graph contains only two types of edges – zero cost and infinite cost (set to a cost of 999,999). Each solution has an optimal tour with cost of zero. If the solver does not find an optimal solution, then it is using at least one infinite cost edge. In this case the percent error will be infinite. Instead, our metrics for performance are the number of times an optimal solution is found (out of the ten 10 runs), and, when optimal is not found, the number of infinite cost edges in the solution. Instance names in this library indicate the SAT instance, the number of sets, and the number of vertices. Note, the reduction from SAT to GTSP allows us to generate a set of challenging GTSP instances for which optimal solution costs are known. We are not proposing this as an effective means for solving SAT instances.

6.2. Tuning instances and results

We performed several experiments to determine the effectiveness of the components of the GLNS algorithm and to tune the default parameters for GLNS. To do this, we used a similar method to that used in prior ALNS solvers (Mauri et al., 2016; Ropke and Pisinger, 2006). We created a small tuning library consisting of 15 problem instances randomly sampled from the libraries GTSP-Lib, GTSP$^+$-Lib, BAF-Lib, and SAT-Lib. We then created several different solver configurations, each turning on or off some components of the GLNS solver. For each configuration we solved each problem in the tuning library 50 times. We recorded the number of times the best known solution was found for each problem and the average gap to best known Δ (for a given run, the percentage gap is calculated as $\Delta = 100 \times (\texttt{Run\_Cost} - \texttt{Best\_Known})/\texttt{Best\_Known}$). We then selected the best performing configuration, created several variations of this configuration, and repeated the experiments. Table 2 summarizes the final settings chosen for the GLNS solver. We created three different built-in solver settings – slow, medium, and fast. The value of each parameter is shown for each of the three solver settings.

Table 3 summarizes the final round of tuning experiments for the medium solver setting. Details of each solver setting are given in Table B.13 of Appendix B. The table ranks the performance of each solver configuration from 1 to 13 (there are 13 configurations, and ties are allowed) on each library in both average percent gap to best known (Δ) and number of best known solutions (# B). For SAT-Lib, where Δ is not defined, the configurations are ranked using the average number of infinite cost edges (Avg.) for each instance. The table highlights the benefits of the different components of the GLNS solver. For example, local optimizations, additive noise, adaptive weights, multiple trials, and restarts all provide significant performance improvements. The worst performing con-

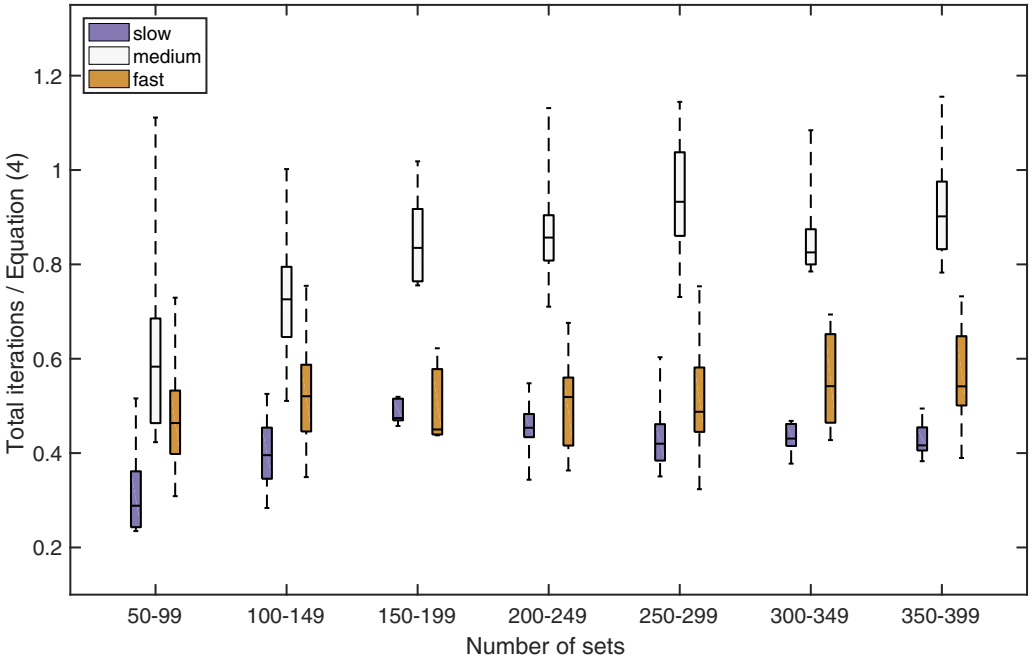

Fig. 1. A boxplot of the ratio of total number of iterations performed by GLNS and the approximate value given in the expression (4) on all instances of GTSP-Lib with 50 or more sets and on the instances of Large-Lib with 200 to 400 sets. The horizontal axis groups instances by the number of sets, and the vertical axis plots the ratio of actual to predicted iterations. The total number of iterations ranges from 6000 for the smallest instance on the fast setting to over 1.2 million for the largest instance on the slow setting.

figuration, ranked 13th, is a configuration in which the insertions are limited to nearest, farthest, and random insertion, while the removals are limited to just worst and random removal. For this configuration, the number of best known solutions was approximately 20% of that found using the final GLNS configurations. This highlights the benefits of the unified insertion and removal methods and the λ-selection mechanism in Algorithm 3. The top two configurations in Table 3 differ only in the initial tour construction. In the default configurations, the tour is constructed using a completely random tour, while in the 2nd ranked configuration, the tour is constructed using the random insertion method. The performance of these two configurations is very similar, and either can be chosen when running the solver. The default configurations was chosen for its higher consistency.

We also ran the tuning library 50 times both with and without the lower-bounding technique described in Section 3.4. By applying the lower-bounding mechanism, the average runtime was reduced by 25% and as expected, this mechanism has no impact on solution quality, since it prunes only unnecessary insertion calculations.

6.3. Runtimes of GLNS solver settings

In this section we discuss the runtime of GLNS for its three settings (slow, medium, and fast). First, for any solver setting, $O(n^2)$ computation time is needed to parse the input, since a complete $n \times n$ distance matrix is formed and the distances in (1) are computed. After parsing, the solver runtime depends on the number of iterations and the computation time per iteration. The exact number of iterations is not deterministic due to the simulated annealing procedure, which is described in Section 5.3. However, for each setting, the same values for p_1, p_2, and p_3 (acceptance parameters defined in Table 2) are used, and thus we have found empirically that the total number of iterations is approximately proportional to

$$\texttt{num\_trials} \times \texttt{num\_warm} \times \texttt{num\_iterations}. \tag{4}$$

Fig. 1 compares the total number of iterations to the expression in (4) for all GTSP-Lib instances with at least 50 sets and for Large-Lib instances with between 200 and 400 sets. Each instance was solved five times for each GLNS setting (slow, medium, and fast). Notice that for each setting, the ratio flattens out as the instances get larger, and the two quantities (predicted and actual) scale proportionately. In an iteration, $N_r \leq N_{\max}$ vertices are removed and inserted, and another $N_{\max}$ removals and insertions are performed during move-opt. Thus, the number of insertions per iteration is at most $2N_{\max}$. In the medium and fast settings, the maximum number of removals $N_{\max}$ per iteration is a constant: at most 20 for fast and at most 100 for medium. This saturation is similar to that proposed in Ropke and Pisinger (2006). Using (4) and the number of insertions per iteration, estimates for the total number of insertions performed in GLNS under each setting are shown in Table 4. The table is broken into three categories since $N_{\max}$ saturates at $m = 200$ for fast and at $m = 333$ for medium.

Finally, for the runtime of each setting, recall that each removal algorithm runs in $O(N_r m)$ time, which is negligible compared to the runtime of insertions and tour optimizations. Using Table 1, Unified insertion and Move-Opt run in $O(N_{\max} n)$ under the assumption that all sets contain $O(n/m)$ vertices. When $N_{\max}$ is constant, the runtime of cheapest insertion is also $O(N_{\max} n)$. However, for the slow setting, the maximum number of removals per iteration $N_{\max}$ is 0.4m, and the runtime of cheapest insertion becomes $O(mn \log m)$. Assuming that there is a set with $O(1)$ vertices, then the tour optimization Re-Opt runs in $O(n^2/m)$ per iteration — Re-Opt is not used in the fast setting. Putting these results together we obtain the results in the rightmost column of Table 4.

A key observation is that for $m \leq 200$, fast performs approximately 10 times fewer insertions than medium, which performs approximately 10 times fewer insertions than slow. The runtime of GLNS follows the same general trend, as summarized Table 5. This table summarizes the results of the GLNS solver on the six problem libraries. Each instance was solved 10 times on each of the three solver settings: slow, medium and fast. The solver was given a timeout of 1200 s and if the timeout was reached on an instance, then the best solution found is reported. The top table in Table 5 shows the average solution quality and runtime for

Table 4
Runtime and approximate number of iterations for the three GLNS settings for a GTSP instance with n vertices and m sets under the assumption that $|V_i| \in O(n/m)$ for each set V_i and there exists a set with $O(1)$ vertices.

GLNS setting	Approx # of Insertions			Runtime after parsing
	$m \leq 200$	$200 < m \leq 333$	$m > 333$	
Slow	$6\,000m^2$	$6\,000m^2$	$6\,000m^2$	$O(\max\{mn^2, m^3 n \log m\})$
Medium	$540m^2$	$540m^2$	$180,000m$	$O(n^2)$
Fast	$72m^2$	$14,400m$	$14,400m$	$O(nm)$

the three different settings of GLNS; fast, medium, and slow. A more detailed comparison is shown in the bottom table of Table 5, where eight sample instances are shown from four different libraries. We see that when all runtimes are below the 1200 s timeout, fast is approximately 10 to 20 times faster than medium, which is approximately 10 times faster than slow. Moreover, we have found in our experiments that the runtime of GLNS is quite stable across multiple runs on a given instance. For the default GLNS settings, on the 750 runs summarized in the first row of Table 3, the maximum percentage difference in runtimes on an instance was 55.4% (calculated as (max − min)/mean over the 50 trials for each instance). It is also worth noting that as the solver begins to time out on the slower setting(s), then the faster setting(s) start to perform better in solution quality (percent gap from best) (as in LARGE-LIB of Table 5), which gives some insight into the benefits of each of the three default settings.

6.4. Library results and discussion

To compare solvers, we performed 300 s and 1200 s persistent tests with the three solvers, GLKH, GK, and GLNS on the six libraries described in Section 6.1. For the persistent tests, each solver is given a 300 or 1200 s timeout, and the settings of the solvers are altered such that they will always use the full amount of time. For GLKH this was done by setting the `MAX_TRIALS` parameter to a sufficiently large number. Note that at the end of each run GLKH performs a post-optimization procedure on the GTSP tour, which also has a `MAX_TRIALS` parameter. This parameter was kept at its default setting (as were all other parameters in the post-optimization). For GK we set the number of `generations` to a large value, and for GLNS we used the medium setting, but with `num_trials` set to a large number. These tests allow for a fair comparison of the performance of each solver over a fixed amount of time. These experiments were performed for all three solvers (GLNS, GLKH, and GK) on an Intel Core i7-6700, 3.40 GHz with 16 GB of RAM. Each instance was run on a single core.

In each experiment we solved each instance 10 times. When the timeout is reached, the best solution found during that time is reported. We do not stop when the optimum is reached as we are interested in applications in which the optimal cost is not known prior to solving. On each instance we report the number of times the best known solution is found and the average percentage gap from the best known solution.

Remark 6.1 (GLKH and GK solver settings)**.** In Helsgaun (2015), three different sets of parameter values are provided: one for GTSP-LIB, one for LARGE-LIB, and one for the arc routing instances. In our benchmarking we ran each library for each of these three settings and took the best performance among them. On GTSP-LIB, the GTSP-LIB settings performed best. On all other libraries, the LARGE-LIB settings performed best in both the 300 and 1200 s persistent tests. It is also worth noting that on the largest instances, the GLKH solver exceeded its configured timeout of 1200 s (for example, in Table A.9). In these instances we let the solver complete, and simply report 1200 s.

For GK, the solver settings are hard-coded into the program, and these settings were not modified, other than increasing `generations` to achieve persistent behavior. •

Detailed tables of results. Detailed tables for each of the six libraries are given in Appendix A. In these tables, only the largest 45 instances are shown for each library in order to limit the size of the tables. For each instance, the best entry (i.e., percent gap from best known, or number of best known solutions) is shown in bold. For each library, we display results for the GLNS medium setting. The GLKH and GK settings are chosen as described in Remark 6.1.

For the libraries GTSP-LIB, MOM-LIB, BAF-LIB, and GTSP$^+$-LIB, which contain problems with fewer than 1500 vertices, we display results for the 300 s persistent tests. For LARGE-LIB and SAT-LIB, where most instances are significantly larger (some exceeding 10,000 vertices), we display results from the 1200 s persistent tests. The persistent results are helpful in removing the substantial differences in runtimes for each solver. In particular, GK in its default setting typically terminates very quickly on instances with fewer than 1000 vertices (Gutin and Karapetyan, 2010, Table 3), even when results are still suboptimal. This results in low runtimes but larger error gap from the best known. On the converse, GLKH often continues running far beyond the time at which the best known solution is found, resulting in high runtimes but low error gap (Helsgaun, 2000).

Summary of results. The results for each of the six tables are summarized in the two bar charts in Fig. 2. These bar charts compare the three solvers on their average percent gap Δ (%) for each instance in each library. For SAT-LIB, where Δ is not defined, the charts compare each solver on the average number of infinite cost edges (Avg.) for each instance. The left chart shows the percentage of instances in each library where a given solver outperformed one or both of the other solvers (thus, the black bars show how often one solver dominates the other two). The right chart is complementary, showing the percentage of instances in each library where a given solver was outperformed by either one or both of the other solvers (thus, the white bars show how often a solver was dominated by the other two solvers). In each chart, the libraries are arranged so that as we move from left to right, the instances go from primarily Euclidean instances with clustered vertex sets to non-metric and highly constrained instances. These constrained non-metric problems typically arise when a related combinatorial problem is reduced to GTSP, and thus are relevant to researchers looking to leverage a GTSP solver for related combinatorial problems. The figures show that GLNS is competitive with the state-of-the-art solvers on all of the libraries. In the libraries MOM-LIB, BAF-LIB, GTSP$^+$-LIB, and SAT-LIB the GLNS solver has the best performance, both in terms of having the largest number of instances where it outperforms both other solvers and having the fewest instances where it is outperformed by the other two solvers. Moreover, as the problems become non-metric and more highly constrained, the performance differences between GLNS and the next best solver widen. For the SAT-LIB library, in which each solver shows the poorest performance relative to optimal, the GLNS

Table 5
Summary of GLNS performance for its three settings, slow, medium, and fast. Top table shows the average results for each library. The bottom table shows eight sample instances in four of the libraries. The column Δ(%) reports the average percentage gap from best known, the column `Time(s)` records the average solver time, and the column `Avg.` records the average number of infinite edges. In the bottom table averages are taken over the ten runs of each instance and in the top table they are taken over the entire library.

Library	Slow		Medium		Fast	
	Δ(%)	Time (s)	Δ(%)	Time (s)	Δ(%)	Time (s)
GTSP-Lib	**0.01**	169	0.04	18	0.99	1
MOM-Lib	**0.01**	485	0.03	159	0.49	12
BAF-Lib	**0.08**	114	0.27	13	2.58	1
Large-Lib	2.66	1140	**1.88**	745	3.43	202
GTSP⁺-Lib	3.33	893	**1.16**	393	13.53	10
	Avg.	Time (s)	Avg.	Time (s)	Avg.	Time (s)
SAT-Lib	8.29	1127	**2.81**	969	8.67	92

Library	Slow		Medium		Fast	
	Δ(%)	Time (s)	Δ(%)	Time (s)	Δ(%)	Time (s)
GTSP-Lib						
134gr666	**0.09**	182	0.38	22	1.17	1
145u724	**0.02**	321	0.06	34	0.96	1
157rat783	**0.08**	471	0.17	43	1.60	2
200dsj1000	**0.01**	1091	0.10	102	2.20	5
201pr1002	**0.00**	846	0.04	78	1.73	5
207si1032	**0.06**	963	0.11	98	0.94	5
212u1060	**0.07**	1048	0.19	120	2.24	5
217vm1084	**0.04**	1167	0.09	126	1.72	5
GTSP⁺-Lib						
64lin318_x15	**2.19**	671	2.43	71	5.61	3
64lin318_x20	**1.20**	1157	1.74	120	7.09	5
64lin318_x25	**1.36**	1200	1.58	224	11.05	10
64lin318_x30	**1.22**	1200	1.63	353	16.41	10
64lin318_x35	**1.05**	1200	1.12	601	19.31	14
64lin318_x40	0.97	1200	**0.80**	649	28.85	15
64lin318_x45	4.51	1200	**1.02**	656	23.60	20
64lin318_x50	6.85	1200	**0.68**	1051	33.79	26
BAF-Lib						
baf131p654	**0.04**	203	0.06	21	0.17	1
baf132d657	**0.00**	152	0.05	20	1.98	2
baf145u724	**0.06**	218	0.35	25	5.18	2
baf157rat783	**0.00**	380	0.79	39	2.82	2
baf201pr1002	**1.99**	741	2.31	80	3.88	5
baf207si1032	**0.00**	877	0.03	91	1.36	5
baf212u1060	**0.10**	811	0.39	101	7.98	7
baf217vm1084	**0.00**	784	0.19	87	0.12	7
Large-Lib						
10C1k	0.00	1074	0.00	134	0.00	5
200C1k	**0.00**	814	0.00	87	1.03	5
261rl1304	**0.19**	1200	0.26	279	2.60	10
287u1432	**0.52**	1200	0.82	281	3.61	12
364u1817	0.71	1200	**0.63**	640	3.49	17
464u2319	4.46	1200	**1.11**	1072	5.18	31
633E3k	5.42	1200	**1.70**	1200	5.28	93
1187rl5934	6.77	1200	6.97	1200	**5.63**	322

solver produces higher quality solutions on every instance, and its solution quality is often an order of magnitude better than GLKH and GK.

The GTSP⁺-Lib library was designed specifically to explore the degradation of solution quality of each solver as the GTSP instance becomes more constrained. In this library, the value of x captures the number of additional pairwise constraints added between vertices. From Table A.11, we can see that for a given GTSP-Lib instance, as we increase x, the performance of GLKH and GK degrades substantially relative to GLNS. For low values of x, where the problem still has much of the structure of the underlying Euclidean instance, the GK solver often outperforms GLNS. However, when x becomes sufficiently large, neither GLKH nor GK are able to find a single feasible solution across the 10 runs (for example 53gil262_x65 and 53gil262_x70), while GLNS finds a feasible solution in all ten runs. This trend is illustrated in Fig. 3 for the series of instances produced from the GTSP-Lib instance 53gil262.

Strengths and weaknesses of each solver. For the six libraries tested, the GLKH solver's biggest strength is on the clustered Euclidean instances in GTSP-Lib (Table A.7) and Large-Lib (Table A.9). On the largest instances within Large-Lib (Table A.9), its average solution error is several percent better than that of GLNS or GK. The solver struggles when vertex sets are not the result of the clustering procedure from Fischetti et al. (1995), such as MOM-Lib in Table A.8 and BAF-Lib in Table A.10. In BAF-Lib there are over 10 instances where the GLKH gap from the best known solution exceeds 5% and five where the gap exceeds 20%. The sets in BAF-Lib are created using pseudorandom clustering, and the GLKH solver struggles with such instances when compared with

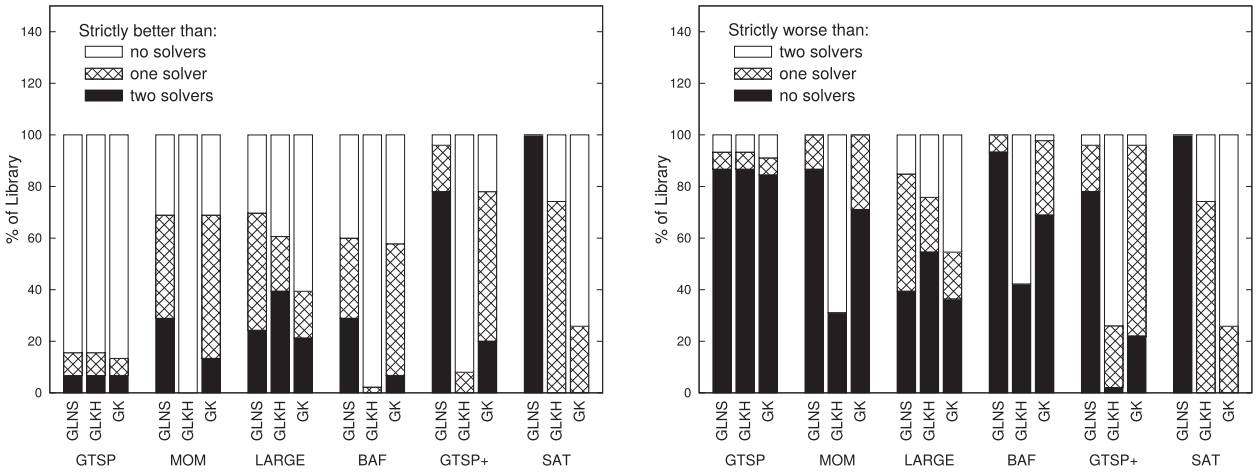

Fig. 2. Performance comparison between solvers for each library. Left: percent of instances in each library where a solver performed strictly better than two, one or no other solvers. Right: percent of instances in each library where a solver performed strictly worse that two, one or no other solvers. For all libraries except SAT-Lib, solver performance is measured by the average error gap $\Delta(\%)$. For SAT-Lib, performance is measured by the average number of infinite edges (Avg.).

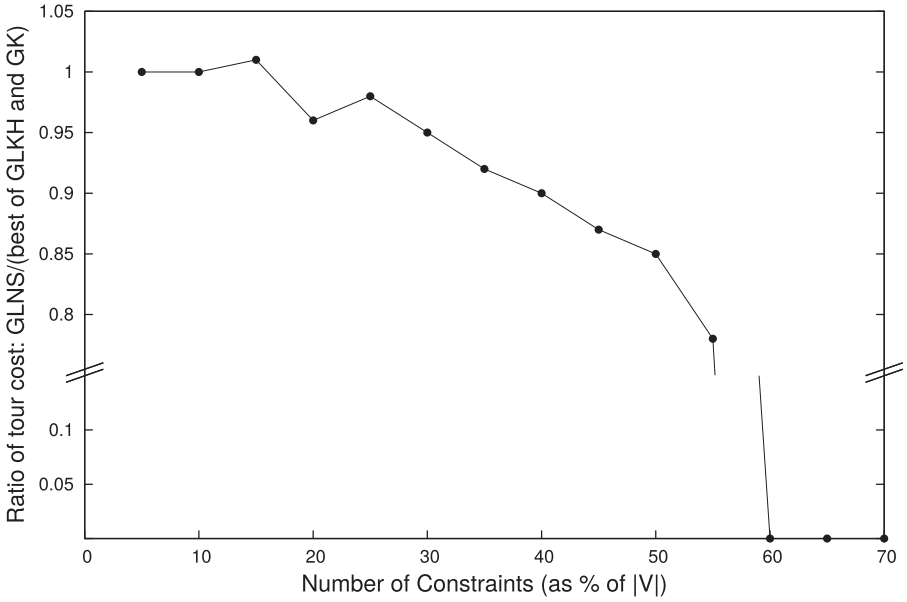

Fig. 3. Performance of GLNS relative to the best result obtained from GLKH and GK with respect to the number of constraints added to the original instance as a percentage of $|V|$ (as indicated by the postfix **_x##** in the instance name(s)), for the series of 53gil262 GTSP$^+$-Lib instances.

instances containing closely grouped vertices in each set. For the more constrained problems in GTSP$^+$-Lib (Table A.11) and SAT-Lib in Table A.12, its performance is not competitive with GLNS.

The GK solver has strong performance on metric instances, independent of the mechanism in which the sets were constructed. Its gap from best known does not exceed 4% on any instance in GTSP-Lib, MOM-Lib, and BAF-Lib, and for the vast majority of instances, its gap is under 0.2%. The solver also found several new best solutions on existing libraries, including one that was not found by GLNS, as shown in Table 6. The solver, however, does not perform well on the highly constrained instances in GTSP$^+$-Lib and SAT-Lib, with the solver at times producing solutions with over 50 times more infinite cost edges than the GLNS solver (for example, the instance jnh7_s953_v4363).

The GLNS solver performs consistently well across all libraries. From Fig. 2, we see that GLNS is competitive with the state-of-the-art solvers on GTSP-Lib instances and on the smaller instances of Large-Lib. It outperforms existing solvers on MOM-Lib and BAF-Lib, finding several new best solutions as shown in Table 6. Its average gap from the best known on these libraries is less than 0.1%, with the gap exceeding 0.5% on only two instances. Moreover, it finds the best known solution more frequently than GK or GLKH. Its main advantage over existing solvers becomes more apparent on highly constrained non-metric problems in GTSP$^+$-Lib and SAT-Lib, where it is able to find feasible solutions when the other solvers are not. On SAT-Lib, the solution quality for GLNS is consistently an order of magnitude better than GK or GLKH. This makes the GLNS solver a particularly good choice for solving GTSP problems that have non-native GTSP constraints encoded into the problem instance.

A weakness of the GLNS solver is its performance on some instances with Euclidean structure. In particular, GLKH shows better performance than GLNS on the largest instances in Large-Lib, and GK shows better performance on the low-x instances in GTSP$^+$-Lib that still maintain much of the structure of the underlying Euclidean instance. An explanation for this is that GLNS lacks any optimization routines that exploit the structure of Euclidean (or metric or symmetric) instances. A key component of the GLKH solver is its ability to identify candidate edges (via α-nearness Helsgaun (2000)) that are likely to be in optimal tours, and this

Table 6
New best solutions found by GLNS and GK. Previous best solutions obtained by GLKH and are available at http://www.akira.ruc.dk/~keld/research/GLKH/.

Problem instance	Previous best	New best	% Change	Solver(s)
BAF-Lib				
baf113pa561	442	431	−2.49	GK
baf115rat575	1346	1330	−1.19	GLNS
baf145u724	7934	7354	−7.32	GLNS
baf201pr1002	48,807	48,400	−0.83	GLNS
baf107att532	3891	3880	−0.28	GLNS, GK
baf131p654	5827	5824	−0.05	GLNS, GK
baf132d657	8160	8132	−0.34	GLNS, GK
baf157rat783	1841	1700	−7.66	GLNS, GK
baf207si1032	18,936	18,836	−0.53	GLNS, GK
baf212u1060	44,488	38,639	−13.15	GLNS, GK
MOM-Lib				
144pcb1173-12x12	16,418	16,412	−0.04	GLNS
150i2000-605	5942	5940	−0.03	GLNS, GK
200i3000-805	6913	6902	−0.16	GLNS, GK

appears to be very effective on Euclidean instances. The GK solver utilizes a wide array of local improvement methods, including 2-opt local search, which are effective for Euclidean and symmetric instances.

7. Conclusions and future work

In this paper we presented GLNS, a new solver for the GTSP based on adaptive large neighborhood search. At the core of the solver is a unified insertion mechanism that contains as special cases the well-known nearest, farthest, and random insertions. This mechanism is also used to provide a unified removal method. In our extensive benchmarking, we have found that the GLNS solver outperforms the state-of-the-art GLKH and GK solvers on several problem libraries. In particular, we found that when compared to existing approaches, the GLNS solver performs particularly well on libraries that are non-metric and/or contain non-clustered vertex sets. These types of problems frequently arise when reducing a related vehicle routing problem to a GTSP instance.

The GLNS solver was designed with the goal of achieving good performance on a diverse set of problem types, and as such it does not have any special optimization mechanisms for Euclidean instances. In particular, even if the input is given as a list of vertex coordinates, it is converted to a complete distance matrix prior to solving. For problems consisting of more than 10,000 vertices, the size of this distance matrix becomes very large, which is why we do not test on instances significantly larger than this size. This is an area of future improvement. In addition, the SAT-Lib and GTSP$^+$-Lib libraries provide two sets of moderately sized problems (compared to those in Large-Lib) that are challenging for current GTSP solvers, and for which significant future performance improvements are possible. Another important avenue for future work is to extend GLNS to handle overlapping sets and the "at least one in a set" GTSP variant. Many problems, including the pairwise constraints in GTSP$^+$-Lib, have a more succinct reduction to the GTSP with overlapping sets, and thus this could provide a promising method to handle these challenging types of constraints.

Appendix A. Detailed library results

In each table, each instance was run ten times. The column Best Val. records the cost of the best known solution. For each run we calculate the gap from the best known as $100 \times$ (`Run_Cost` − `Best_Known`)/`Best_Known`, and $\Delta(\%)$ is the average percentage gap over the 10 runs. The column # B gives the number of times the best known solution was found in the ten runs. Instances were run as either 300 s or 1200 s persistent tests, as described in Section 6.4. The settings are detailed in each table caption.

In this table, an E is used to mark instances in which the GK solver faulted (i.e., it ran out of memory).

In Table A.11, for the column B/F, the first number gives the number of runs (out of 10) in which the solver found the best known solution. The second number gives the number of times a feasible solution was found, where feasible is defined as a tour with no infinite edges. For example, an entry 5/6 would indicate that the solver found a feasible solution (one without infinite cost edges) in 6 of the 10 trials, and 5 of those 6 feasible solutions were the best known. The average percentage gap from the best known $\Delta(\%)$ is calculated on the subset of runs that found feasible solutions. When no feasible solution is found in all 10 runs, a dash "–" is used to populate the corresponding entries.

In Table A.12, column Avg. records the average number of infinite cost edges in the solution. An E is used to mark instances in which the GK solver faulted (it ran out of memory).

Table A7
GTSP-Lib, GLNS (mode=medium) vs GLKH (parameters set using GTSP-Lib mode from [23]) and GK. All experiments were run for 300 s.

Name	Best Val.	GLNS		GLKH		GK	
		# B	Δ(%)	# B	Δ(%)	# B	Δ(%)
31pr152	51,576	10	0.00	10	0.00	10	0.00
32u159	22,664	10	0.00	10	0.00	10	0.00
35ftv170	1,205	10	0.00	10	0.00	10	0.00
35si175	5,564	10	0.00	10	0.00	10	0.00
36brg180	4,420	10	0.00	10	0.00	10	0.00
39rat195	854	10	0.00	10	0.00	10	0.00
40d198	10,557	10	0.00	10	0.00	10	0.00
40kroa200	13,406	10	0.00	10	0.00	10	0.00
40krob200	13,111	10	0.00	10	0.00	10	0.00
41gr202	23,301	10	0.00	10	0.00	10	0.00
45ts225	68,340	10	0.00	10	0.00	10	0.00
45tsp225	1,612	10	0.00	10	0.00	10	0.00
46gr229	71,972	10	0.00	10	0.00	10	0.00
46pr226	64,007	10	0.00	10	0.00	10	0.00
53gil262	1,013	10	0.00	10	0.00	10	0.00
53pr264	29,549	10	0.00	10	0.00	10	0.00
56a280	1,079	10	0.00	10	0.00	10	0.00
60pr299	22,615	10	0.00	10	0.00	10	0.00
64lin318	20,765	10	0.00	10	0.00	10	0.00
65rbg323	471	10	0.00	10	0.00	10	0.00
72rbg358	693	10	0.00	10	0.00	10	0.00
80rd400	6,361	10	0.00	10	0.00	10	0.00
81rbg403	1,170	10	0.00	10	0.00	10	0.00
84 417	9,651	10	0.00	10	0.00	10	0.00
87gr431	101,946	10	0.00	10	0.00	10	0.00
88pr439	60,099	10	0.00	10	0.00	10	0.00
89pcb442	21,657	10	0.00	10	0.00	10	0.00
89rbg443	632	10	0.00	10	0.00	8	0.05
99d493	20,023	10	0.00	10	0.00	10	0.00
107ali535	128,639	10	0.00	10	0.00	10	0.00
107att532	13,464	10	0.00	10	0.00	10	0.00
107si535	13,502	10	0.00	10	0.00	10	0.00
113pa561	1,038	10	0.00	10	0.00	10	0.00
115rat575	2,388	10	0.00	10	0.00	10	0.00
115u574	16,689	10	0.00	10	0.00	10	0.00
131p654	27,428	10	0.00	10	0.00	10	0.00
132d657	22,498	**10**	**0.00**	7	0.02	7	0.00
134gr666	163,028	**10**	**0.00**	9	0.00	7	0.14
145u724	17,272	5	0.02	9	0.02	**10**	**0.00**
157rat783	3,262	2	0.07	**5**	**0.04**	2	0.09
200dsj1000	9,187,884	1	0.06	**7**	**0.02**	7	0.05
201pr1002	114,311	6	0.02	**10**	**0.00**	6	0.02
207si1032	22,306	0	0.07	2	0.07	2	**0.01**
212u1060	106,007	1	**0.07**	0	0.07	**3**	0.12
217vm1084	130,704	4	0.11	9	0.02	**10**	**0.00**
Average		8.87	0.01	**9.29**	**0.01**	9.16	0.01

Table A8
MOM-Lib, GLNS (mode=medium) vs GLKH (parameters set using Large-Lib mode from [23]) and GK. All experiments were run for 300 s.

Name	Best Val.	GLNS		GLKH		GK	
		# B	Δ(%)	# B	Δ(%)	# B	Δ(%)
50i2000-603	4325	10	0.00	8	0.10	10	0.00
50i2500-707	3961	4	0.05	4	0.65	**10**	**0.00**
50i3000-802	4070	10	0.00	4	1.38	10	0.00
50kroA100	15,944	10	0.00	10	0.00	10	0.00
50kroB100	15,842	10	0.00	10	0.00	10	0.00
50lin105	11,294	10	0.00	10	0.00	10	0.00
50lin318	18,163	10	0.00	10	0.00	10	0.00
50nrw1379	7449	10	0.00	9	0.09	10	0.00
50pcb1173	9385	10	0.00	10	0.00	10	0.00
50pcb442	14,430	10	0.00	10	0.00	10	0.00
50pr1002	54,583	10	0.00	9	0.02	10	0.00
50pr439	45,253	10	0.00	10	0.00	10	0.00
50rat783	1626	10	0.00	10	0.00	10	0.00
50rat99	814	10	0.00	10	0.00	10	0.00
50vm1084	54,156	10	0.00	10	0.00	10	0.00
72vm1084-8x9	64,647	10	0.00	7	0.12	10	0.00
75lin105	13,134	10	0.00	10	0.00	10	0.00
81vm1084-9x9	69,659	10	0.00	10	0.00	10	0.00
100i1000-410	5481	**10**	**0.00**	3	0.26	9	0.07
100i1500-506	5088	**10**	**0.00**	1	0.40	9	0.03
100i2000-604	5316	**10**	**0.00**	0	2.83	9	0.05
100i2500-708	5297	9	0.00	0	3.27	**10**	**0.00**
100i3000-803	5458	8	**0.01**	0	3.54	8	0.01
100nrw1379	10,566	10	0.00	2	0.62	10	0.00
100pcb1173	13,901	**10**	**0.00**	2	0.52	3	0.19
100pr1002	74,269	**10**	**0.00**	4	0.05	7	0.00
100prb1173-10x10	12,644	10	0.00	5	0.19	10	0.00
100rat783	2496	10	0.00	10	0.00	10	0.00
100rat783-10x10	2216	10	0.00	10	0.00	10	0.00
100vm1084	78,440	10	0.00	6	0.08	10	0.00
144pcb1173-12x12	16,412	**10**	**0.00**	1	0.22	0	0.18
144rat783-12x12	2813	10	0.00	5	0.21	10	0.00
150i1000-411	6296	10	0.00	8	0.11	10	0.00
150i1500-507	6085	10	0.00	6	0.41	10	0.00
150i2000-605	5940	9	0.00	1	1.09	**10**	**0.00**
150i2500-709	6158	**10**	**0.00**	0	3.77	6	0.08
150i3000-804	6551	0	0.36	0	5.92	**6**	**0.09**
150nrw1379	13,370	**10**	**0.00**	2	0.52	2	0.14
150pcb1173	17,082	**10**	**0.00**	2	0.49	4	0.16
150pr1002	92,969	10	0.00	7	0.05	10	0.00
150rat783	3131	**9**	**0.01**	3	0.26	6	0.14
150vm1084	95,922	9	0.00	4	0.15	**10**	**0.00**
200i2000-606	7272	**7**	**0.03**	0	1.29	1	0.15
200i2500-710	7191	**4**	**0.07**	0	3.36	1	0.14
200i3000-805	6902	4	0.26	0	4.93	**9**	**0.00**
Average		**9.18**	**0.02**	5.40	0.82	8.44	0.03

Table A9
Large-Lib, GLNS (mode=medium) vs GLKH (parameters set using Large-Lib mode from Helsgaun (2015)) and GK. All experiments were run for 1200 s. An E is used to mark instances in which the GK solver faulted. *Averages only reflect instances where all three solvers were able to find solutions.

Name	Best Val.	GLNS		GLKH		GK	
		# B	Avg.	# B	Avg.	# B	Avg.
10C1k	2,522,585	10	0.00	10	0.00	10	0.00
31C3k	3,553,142	10	0.00	10	0.00	10	0.00
49usa1097	10,337	10	0.00	10	0.00	10	0.00
200C1k	6,375,154	10	0.00	10	0.00	10	0.00
200E1k	9,662,857	4	**0.05**	2	0.21	4	0.10
235pcb1173	23,399	**7**	**0.02**	2	0.42	4	0.30

(*continued on next page*)

Table A9 (*continued*)

Name	Best Val.	GLNS		GLKH		GK	
		# B	Avg.	# B	Avg.	# B	Avg.
259d1291	28,400	2	0.26	3	0.23	**4**	**0.11**
261rl1304	150,468	**8**	**0.04**	3	0.22	1	0.22
265rl1323	154,023	1	**0.05**	1	0.12	0	0.08
276nrw1379	20,050	0	0.38	**4**	0.28	2	**0.23**
280fl1400	15,316	10	0.00	10	0.00	10	0.00
287u1432	54,469	0	0.39	0	0.45	**1**	**0.13**
316fl1577	14,182	**10**	**0.00**	6	0.00	5	0.00
331d1655	29,443	**4**	**0.07**	3	0.23	0	0.31
350vm1748	185,459	0	0.41	0	**0.18**	0	0.19
364u1817	25,530	0	0.40	1	**0.22**	1	0.26
378rl1889	184,034	0	0.60	**1**	0.26	0	**0.24**
421d2103	40,049	0	0.63	0	**0.62**	0	0.64
431u2152	27,614	0	0.75	0	0.80	0	**0.56**
464u2319	65,758	0	1.05	0	1.70	0	**0.71**
479pr2392	169,874	0	0.91	0	1.24	0	**0.74**
608pcb3038	52,416	0	**1.51**	0	1.67	0	2.69
633C3k	10,255,031	0	0.26	**3**	**0.13**	0	1.24
633E3k	16,197,552	0	1.91	0	**1.63**	0	3.62
759fl3795	18,662	0	0.29	0	**0.13**	0	0.87
893fnl4461	63,163	0	2.93	0	**2.85**	0	6.45
1183rl5915	309,243	0	7.36	0	**2.67**	0	E
1187rl5934	295,767	0	6.96	0	**2.33**	0	E
1480pla7397	12,732,870	0	7.14	0	**1.02**	0	E
100C10k	6,158,999	0	**0.27**	0	3.31	0	E
2000C10k	18,044,846	0	5.58	0	**1.17**	0	E
2000E10k	28,769,011	0	8.46	0	**3.77**	0	E
2370rl11849	427,996	0	9.47	0	**3.68**	0	E
Average*		**3.31**	**0.50**	3.04	0.52	2.77	0.76

Table A10
BAF-Lib, GLNS (mode=medium) vs GLKH (parameters set using Large-Lib mode from Helsgaun (2015)) and GK. All experiments were run for 300 63s.

Name	Best Val.	GLNS		GLKH		GK	
		# B	Δ(%)	# B	Δ(%)	# B	Δ(%)
baf20kroD100	5266	10	0.00	10	0.00	10	0.00
baf20kroE100	5449	10	0.00	10	0.00	10	0.00
baf20rat99	230	10	0.00	10	0.00	10	0.00
baf20rd100	1747	10	0.00	10	0.00	10	0.00
baf21eil101	105	10	0.00	10	0.00	10	0.00
baf21lin105	2758	10	0.00	10	0.00	10	0.00
baf22pr107	6849	10	0.00	10	0.00	10	0.00
baf24gr120	1377	10	0.00	10	0.00	10	0.00
baf25pr124	10,745	10	0.00	10	0.00	10	0.00
baf26bier127	11,740	10	0.00	10	0.00	10	0.00
baf28pr136	17,824	10	0.00	10	0.00	10	0.00
baf29pr144	14,070	10	0.00	10	0.00	10	0.00
baf30kroA150	7005	10	0.00	10	0.00	10	0.00
baf30kroB150	5855	10	0.00	10	0.00	10	0.00
baf31pr152	13,002	10	0.00	10	0.00	10	0.00
baf32u159	7301	10	0.00	5	2.50	10	0.00
baf39rat195	477	**10**	**0.00**	3	0.75	9	0.04
baf40d198	1466	10	0.00	10	0.00	8	0.12
baf40kroA200	7113	10	0.00	10	0.00	10	0.00
baf40kroB200	7126	10	0.00	2	2.57	10	0.00
baf41gr202	3531	10	0.00	10	0.00	10	0.00
baf45ts225	25,697	10	0.00	10	0.00	10	0.00
baf46pr226	13,555	10	0.00	2	11.17	10	0.00
baf53gil262	571	**10**	**0.00**	2	0.86	1	0.47
baf53pr264	7716	**10**	**0.00**	4	1.90	9	0.02
baf60pr299	10,047	10	0.00	0	6.91	10	0.00
baf64lin318	7489	**10**	**0.00**	4	5.79	9	0.07
baf80rd400	3254	**10**	**0.00**	2	2.97	9	0.04

(*continued on next page*)

Table A10 (*continued*)

Name	Best Val.	GLNS		GLKH		GK	
		# B	Δ(%)	# B	Δ(%)	# B	Δ(%)
baf84fl417	2226	10	0.00	0	8.76	10	0.00
baf87gr431	10,569	10	0.00	1	2.40	10	0.00
baf88pr439	13,882	1	**0.03**	0	4.37	**9**	0.23
baf89pcb442	8749	10	0.00	0	22.11	10	0.00
baf99d493	3081	10	0.00	3	0.88	10	0.00
baf107att532	3880	**10**	**0.00**	0	15.96	1	0.55
baf107si535	8912	10	0.00	9	0.00	10	0.00
baf113pa561	431	0	1.07	0	9.51	**1**	**0.97**
baf115rat575	1330	**10**	**0.00**	0	8.36	0	1.59
baf131p654	5824	4	**0.03**	0	52.17	4	0.03
baf132d657	8132	**10**	**0.00**	0	18.27	8	0.09
baf145u724	7354	**10**	**0.00**	0	14.09	0	3.64
baf157rat783	1700	**8**	**0.06**	0	28.26	1	3.62
baf201pr1002	48,400	0	1.78	0	22.38	0	**1.33**
baf207si1032	18,836	7	**0.01**	0	3.33	**8**	0.01
baf212u1060	38,639	1	0.12	0	43.26	**8**	**0.09**
baf217vm1084	44,681	10	0.00	0	3.55	10	0.00
Average		**8.91**	**0.07**	5.04	6.51	8.11	0.29

Table A11
GTSP + -Lib, GLNS (mode=medium) vs GLKH (parameters set using Large-Lib mode from Helsgaun (2015)) and GK. All experiments were run for 300 s.

Name	Best Val.	GLNS			GLKH			GK		
		Best	B/F	Δ(%)	Best	B/F	Δ(%)	Best	B/F	Δ(%)
35si175_x05	5574	5574	7/10	0.04	5,574	10/10	0.00	5,574	10/10	0.00
35si175_x10	5574	5588	0/10	0.39	5,578	0/10	0.47	**5,574**	**8**/10	**0.24**
35si175_x15	5574	5595	0/10	0.46	5,627	0/10	1.49	**5,574**	**10**/10	**0.00**
35si175_x20	5574	5600	0/10	0.47	5,640	0/10	2.32	**5,574**	**7**/10	**0.28**
35si175_x25	5574	5608	0/10	**0.61**	5,755	0/10	4.08	**5,574**	**3**/10	0.62
35si175_x30	5586	5624	0/10	**0.71**	5,703	0/9	4.01	**5,586**	**1**/10	1.28
35si175_x35	5633	5658	0/10	**0.64**	5,783	0/4	3.98	**5,633**	**1**/10	1.08
35si175_x40	5635	5730	0/10	1.69	-	0/0	-	**5,635**	**1**/10	**1.05**
35si175_x45	5710	5736	0/10	**0.49**	-	0/0	-	**5,710**	**1**/10	1.52
35si175_x50	5626	5736	0/10	**2.02**	-	0/0	-	**5,626**	**1**/10	2.52
35si175_x55	5764	**5764**	**4**/10	**0.07**	-	0/0	-	5,768	0/10	0.88
35si175_x60	5781	5785	0/**10**	**0.18**	-	0/0	-	**5,781**	**1**/5	3.17
35si175_x65	5712	5739	0/**10**	**0.78**	-	0/0	-	**5,712**	**1**/2	1.98
35si175_x70	5787	**5801**	0/**10**	**0.52**	-	0/0	-	-	0/0	-
35si175_x75	5784	**5805**	0/**10**	**1.05**	-	0/0	-	5,972	0/2	4.49
53gil262_x05	1013	1015	0/10	0.20	1,014	0/10	0.21	**1,013**	**8**/10	**0.17**
53gil262_x10	1013	1015	0/10	0.21	1,031	0/10	3.30	**1,013**	**6**/10	**0.19**
53gil262_x15	1015	1026	0/10	1.52	1,028	0/10	3.40	**1,015**	**1**/10	**0.99**
53gil262_x20	1017	1019	0/10	**0.40**	1,064	0/9	6.60	**1,018**	0/10	4.07
53gil262_x25	1018	**1021**	0/10	**1.01**	1,119	0/8	11.95	1,024	0/10	3.43
53gil262_x30	1022	**1023**	0/10	**0.38**	1,092	0/4	9.34	1,024	0/10	5.23
53gil262_x35	1027	**1029**	0/10	**0.62**	1,138	0/9	14.65	1,058	0/10	9.28
53gil262_x40	1021	**1021**	**1**/10	**0.23**	-	0/0	-	1,084	0/10	11.16
53gil262_x45	1026	**1026**	**2**/**10**	**0.63**	1,306	0/3	30.47	1,105	0/8	15.22
53gil262_x50	1028	**1028**	**1**/**10**	**3.63**	-	0/0	-	1,170	0/9	21.26
53gil262_x55	1026	**1048**	0/**10**	**6.76**	-	0/0	-	1,220	0/4	36.70
53gil262_x60	1027	**1066**	0/**10**	**8.78**	-	0/0	-	-	0/0	-

(*continued on next page*)

Table A11 (*continued*)

Name	Best Val.	GLNS			GLKH			GK		
		Best	B/F	Δ(%)	Best	B/F	Δ(%)	Best	B/F	Δ(%)
53gil262_x65	1032	**1043**	0/**10**	**7.43**	-	0/0	-	-	0/0	-
53gil262_x70	1041	**1108**	0/**10**	**14.67**	-	0/0	-	-	0/0	-
64lin318_x05	21,061	21,318	0/10	1.66	21,156	0/10	0.98	**21,061**	**8**/10	**0.44**
64lin318_x10	21,228	21,559	0/10	2.30	21,930	0/8	4.99	**21,228**	**1**/10	**1.96**
64lin318_x15	21,447	21,876	0/10	2.25	22,696	0/1	5.82	**21,447**	**1**/10	**1.52**
64lin318_x20	21,826	22,062	0/10	**1.37**	23,265	0/1	6.59	**21,826**	**1**/10	2.83
64lin318_x25	22,284	22,547	0/10	**1.43**	24,068	0/1	8.01	**22,284**	**1**/10	3.68
64lin318_x30	22,423	**22,536**	0/10	**1.42**	-	0/0	-	22,769	0/10	3.89
64lin318_x35	22,718	**22,779**	0/10	**1.04**	-	0/0	-	23,326	0/10	7.57
64lin318_x40	22,874	**22,876**	0/10	**0.95**	-	0/0	-	23,979	0/10	7.67
64lin318_x45	22,954	**23,029**	0/**10**	**1.66**	-	0/0	-	25,137	0/5	17.84
64lin318_x50	23,217	**23,240**	0/**10**	**1.40**	-	0/0	-	-	0/0	-
64lin318_x55	23,176	**23,204**	0/**10**	**3.67**	-	0/0	-	-	0/0	-
99d493_x05	20,042	20,198	0/10	**0.81**	20,263	0/10	1.69	**20,042**	**1**/10	0.84
99d493_x10	20,318	20,484	0/10	**1.08**	20,530	0/9	2.74	**20,318**	**1**/10	1.78
99d493_x15	20,413	**20,483**	0/**10**	**0.86**	-	0/0	-	20,921	0/8	3.88
99d493_x20	20,677	**20,763**	0/**10**	**0.77**	-	0/0	-	22,638	0/2	11.08
99d493_x25	20,800	**20,988**	0/**10**	**2.55**	-	0/0	-	-	0/0	-
99d493_x30	20,873	**21,556**	0/**10**	**6.03**	-	0/0	-	-	0/0	-
157rat783_x05	3297	**3301**	0/10	**0.63**	3,514	0/7	7.75	3,309	0/10	2.42
157rat783_x10	3357	**3363**	0/10	**0.80**	3,638	0/1	8.37	3,904	0/10	33.39
157rat783_x15	3413	**3450**	0/**10**	**2.52**	-	0/0	-	-	0/0	-
157rat783_x20	3482	**3557**	0/**10**	**3.76**	4,045	0/1	16.17	-	0/0	-

Table A12
SAT-Lib, GLNS (mode=medium) vs GLKH (parameters set using Large-Lib mode from Helsgaun (2015)) and GK. All experiments were run for 1200 s. An E is used to mark instances in which the GK solver faulted. *Averages only reflect instances where all three solvers were able to find solutions.

Name	Best Val.	GLNS		GLKH		GK	
		# B	Avg.	# B	Avg.	# B	Avg.
uf20_s114_v316	0	**10**	**0**	0	1	0	1
aim_s153_v403	0	**8**	**0**	0	2	0	1
aim_s263_v682	0	0	**1**	0	4	0	2
uf50_s271_v757	0	**10**	**0**	0	4	0	4
aim_s303_v802	0	0	**1**	0	6	0	4
flat30_s393_v813	0	**10**	**0**	0	10	0	6
uf75_s403_v1128	0	**10**	**0**	0	11	0	9
aim_s523_v1362	0	0	**1**	0	11	0	9
uf100_s533_v1493	0	**1**	**1**	0	15	0	14
aim_s603_v1602	0	0	**1**	0	18	0	19
ais6_s645_v1476	0	**2**	**0**	0	8	0	21
uf125_s666_v1867	0	0	**1**	0	18	0	25
flat50_s698_v1443	0	**1**	**1**	0	23	0	25
uf150_s798_v2238	0	**2**	**1**	0	22	0	36
jnh204_s903_v4117	0	0	**1**	0	28	0	99
jnh205_s903_v4114	0	0	**1**	0	31	0	89
jnh207_s903_v4139	0	0	**2**	0	29	0	92
jnh209_s903_v4105	0	0	**1**	0	30	0	98
jnh210_s903_v4118	0	**6**	**0**	0	29	0	85
jnh212_s903_v4135	0	0	**2**	0	30	0	94
jnh213_s903_v4103	0	0	**1**	0	33	0	90
jnh217_s903_v4142	0	0	**1**	0	29	0	95
jnh218_s903_v4108	0	**1**	**1**	0	28	0	94
jnh220_s903_v4126	0	0	**1**	0	31	0	92
uf175_s931_v2612	0	0	**4**	0	28	0	65
jnh1_s953_v4595	0	0	**3**	0	33	0	120
jnh7_s953_v4363	0	0	**2**	0	38	0	105
jnh301_s1003_v4857	0	0	**3**	0	40	0	141
uf200_s1063_v2983	0	0	**7**	0	35	0	E
flat75_s1068_v2208	0	0	**7**	0	36	0	E
uf225_s1188_v3333	0	0	**16**	0	37	0	E
Average*		**2.18**	**1.37**	0.00	21.62	0.00	55.29

Appendix B. Details of solver settings for tuning

In the following we describe each of the settings for the experiment shown in Table 3. Each setting is a modification of the configuration described in Table 2, and here we detail the modification for each.

Table B1
Details of each solver setting used in the tuning results of Table 3.

Tuning Setting	Description
Default GLNS configuration	Medium configuration as described in Table 2.
Random insertion T_{init}	Initial tour is constructed using rand insert instead of random.
Nearest, farthest, random	Insertion mechanisms are limited to nearest, farthest, and random insertion. This is done using insertion values λ of 0, 1, and ∞.
No adaptive weights	Adaptive weights are turned off by setting $\epsilon = 0$.
No cheapest insertion	Cheapest insertion is disabled.
No noise	Additive noise is disabled in the insertion mechanisms.
No local optimizations	Local optimizations (Section 5.5) after each iteration are disabled.
Subset and additive noise	In addition to additive noise, subset noise (Section 3.5) is enabled, and is configured with f values of 0.25 and 0.5.
Only subset noise	Subset noise is enabled and Additive noise is disabled. Subset noise used f values of 0.25 and 0.5.
No restarts	Warm restarts are disabled by setting `num_warm` $= 0$ and `num_trials` is increased to 10, so that the total runtime is approximately equal to that of the default setting.
Worst and random removal	Removal mechanisms are limited to worst and random removal. This is done using removal values λ of 1 and ∞.
One trial	One long trial is used, with no warm restarts. This is done by setting `num_trials` to 1, `num_warm` to 0, and increasing `num_iterations` to $800m$ such that the total runtime is approximately equal to that of the default setting.
Settings 3 and 11	Insertions are limited to nearest, farthest and random (i.e., $\lambda \in \{0, 1, \infty\}$). Removals are limited to worst and random (i.e., $\lambda \in \{1, \infty\}$).

References

Ben-Arieh, D., Gutin, G., Penn, M., Yeo, A., Zverovitch, A., 2003. Transformations of generalized ATSP into ATSP. Oper. Res. Lett. 31 (5), 357–365.

Bontoux, B., 2008. Techniques Hybrides de Recherche Exacte et Approchée: Application à des Problèmes de Transport. Université d'Avignon Ph.d. Thesis.

Bontoux, B., Artigues, C., Feillet, D., 2010. A memetic algorithm with a large neighborhood crossover operator for the generalized traveling salesman problem. Comput. Oper. Res. 37, 1844–1852.

Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., Schulenburg, S., 2003. Hyper-heuristics: an emerging direction in modern search technology. In: Handbook of Metaheuristics. Springer, pp. 457–474.

Burke, E.K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.R., 2010. A classification of hyper-heuristic approaches. In: Handbook of Metaheuristics. Springer, pp. 449–468.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2009. Introduction to Algorithms, third ed. MIT Press.

Drexl, M., 2013. On the generalized directed rural postman problem. J. Oper. Res. Soc. 65 (8), 1143–1154.

Fischetti, M., González, J., Toth, P., 1995. The symmetric generalized traveling salesman polytope. Networks 26 (2), 113–123.

Fischetti, M., González, J., Toth, P., 1997. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. Oper. Res. 45 (3), 378–394.

Garg, N., Konjevod, G., Ravi, R., 2000. A polylogarithmic approximation algorithm for the group steiner tree problem. J. Algo. 37 (1), 66–84.

Gutin, G., Karapetyan, D., 2010. A memetic algorithm for the generalized traveling salesman problem. Nat. Comput. 9 (1), 47–60.

Helsgaun, K., 2000. An effective implementation of the Lin–Kernighan traveling salesman heuristic. Eur. J. Oper. Res. 126 (1), 106–130.

Helsgaun, K., 2015. Solving the equality generalized traveling salesman problem using the Lin–Kernighan–Helsgaun algorithm. Math.s Program. Comput. 7 (3), 269–287.

Hemmelmayr, V.C., Cordeau, J.-F., Crainic, T.G., 2012. An adaptive large neighborhood search heuristic for two-echelon vehicle routing problems arising in city logistics. Comput. Oper. Res. 39 (12), 3215–3228.

Hoos, H., Stiitzle, T., 2000. SATLIB: an online resource for research on SAT. In: SAT2000: Highlights of Satisfiability Research in the Year 2000, pp. 283–292.

Imeson, F., Smith, S.L., 2015. Multi-robot task planning and sequencing using the SAT-TSP language. In: IEEE International Conference on Robotics and Automation, pp. 5397–5402.

Karapetyan, D., Gutin, G., 2011. Lin–Kernighan heuristic adaptations for the generalized traveling salesman problem. Eur. J. Oper. Res. 208 (3), 221–232.

Laporte, G., Asef-Vaziri, A., Sriskandarajah, C., 1996. Some applications of the generalized travelling salesman problem. Jo. Oper. Res. Soc. 1461–1467.

Lin, S., Kernighan, B.W., 1973. An effective heuristic algorithm for the traveling-salesman problem. Oper. Res. 21 (2), 498–516.

Mathew, N., Smith, S.L., Waslander, S.L., 2015. Multirobot rendezvous planning for recharging in persistent tasks. IEEE Trans. Robot. 31 (1), 128–142.

Mauri, G.R., Ribeiro, G.M., Lorena, L.A.N., Laporte, G., 2016. An adaptive large neighborhood search for the discrete and continuous berth allocation problem. Comput. Oper. Res. 70, 140–154.

Mestria, M., Ochi, L.S., de Lima Martins, S., 2013. GRASP with path relinking for the symmetric euclidean clustered traveling salesman problem. Comput. Oper. Res. 40 (12), 3218–3229.

Noon, C., Bean, J., 1991. A Lagrangian based approach for the asymmetric generalized traveling salesman problem. Oper. Res. 39 (4), 623–632.

Noon, C.E., Bean, J.C., 1993. An efficient transformation of the generalized traveling salesman problem. INFOR 31 (1), 39.

Ny, J.L., Feron, E., Frazzoli, E., 2012. On the Dubins traveling salesman problem. IEEE Trans. Autom. Control 57 (1), 265–270.

Obermeyer, K.J., Oberlin, P., Darbha, S., 2012. Sampling-based path planning for a visual reconnaissance UAV. AIAA J. Guid. Control Dyn. 35 (2), 619–631.

Pisinger, D., Ropke, S., 2007. A general heuristic for vehicle routing problems. Comput. Oper. Res. 34 (8), 2403–2435.

Renaud, J., Boctor, F.F., 1998. An efficient composite heuristic for the symmetric generalized traveling salesman problem. Eur. J. Oper. Res. 108 (3), 571–584.

Ribeiro, G.M., Laporte, G., 2012. An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem. Comput. Oper. Res. 39 (3), 728–735.

Ropke, S., Pisinger, D., 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transport. Sci. 40 (4), 455–472.

Rosenkrantz, D.J., Stearns, R.E., Lewis, P.M., 1977. II, an analysis of several heuristics for the traveling salesman problem. SIAM J. Comput. 6 (3), 563–581.

Saha, M., Roughgarden, T., Latombe, J.-C., Sánchez-Ante, G., 2006. Planning tours of robotic arms among partitioned goals. Int. J. Robot. Res. 25 (3), 207–223.

Shaw, P., 1997. A new local search algorithm providing high quality solutions to vehicle routing problems. Tech. Rep. Dept of Computer Science, University of Strathclyde, Glasgow, Scotland, UK.

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. Springer, pp. 165–181.

Snyder, L.V., Daskin, M.S., 2006. A random-key genetic algorithm for the generalized traveling salesman problem. Eur. J. Oper. Res. 174 (1), 38–53.

Sutton, R.S., Barto, A.G., 1998. Reinforcement Learning: An Introduction. MIT Press.

Tasgetiren, M.F., Suganthan, P.N., Pan, Q.-Q., 2007. A discrete particle swarm optimization algorithm for the generalized traveling salesman problem. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 158–167.

Wolff, E.M., Topcu, U., Murray, R.M., 2013. Optimal control of non-deterministic systems for a computationally efficient fragment of temporal logic. In: IEEE Conference on Decision and Control, pp. 3197–3204.