A GRASP with efficient neighborhood search for the integrated maintenance and bus scheduling problem

Vitor Mitsuo FUKUSHIGUE HAMA*, Wei WU** and Mutsunori YAGIURA*

* Department of Mathematical Informatics, Nagoya University
Furocho, Chikusaku, Nagoya, 464-8601, Japan
E-mail: vitorhama@nagoya-u.jp

** Faculty of Science and Technology, Seikei University
3-3-1, Kita, Kichijoji, Musashino, Tokyo, 180-8633, Japan

Abstract
The overall planning process undertaken by a bus company is traditionally composed of five sub-processes: timetabling, vehicle scheduling, maintenance scheduling, crew scheduling, and crew rostering. Solving the full optimization problem is believed to be computationally intractable, and therefore in practice the five sub-processes are usually optimized in sequence. In this paper, we present a model that integrates the problems of vehicle scheduling and maintenance scheduling. The objective is to minimize the differences in mileage between buses, the total distance traveled, and the daily differences in the number of maintenance tasks. We propose a heuristic algorithm using the framework of greedy randomized adaptive search procedure (GRASP), and we improve the neighborhood search procedure by using an ordered list of possible trips. We compare the neighborhood search procedure with and without this mechanism and show that the ordered list reduces the number of neighbors to be checked by more than 95%, and it reduces the time to obtain solutions of the same or better quality by an average of 70%. Through computational experiments on instances generated from real-world data, we show that the proposed algorithm finds solutions that are as good as or better than those obtained by a commercial solver in less than 5% of the time required by the latter.

Keywords : Bus scheduling, Maintenance scheduling, Integrated model, Greedy randomized adaptive search procedure, Quadratic programming

1. Introduction
The operational planning process of a bus company is composed of several recurrent tasks that are traditionally performed in sequence. Figure 1 shows the relationship between these operational planning tasks in a bus transport company.
Integrating all these planning processes would be highly complex, so in practice they are planned sequentially by different sectors of the company (Naumann et al., 2011). In this paper, we consider the integration of two of these steps: vehicle scheduling and maintenance scheduling. Although the optimization of these two steps in sequence is computationally easier, the sequential approach provides an overall suboptimal solution. In the worst case, the bus schedule may fail to satisfy the maintenance constraints of an otherwise feasible problem (Papadakos, 2009).

Buses, as with any machine, have an expected usable time range. However, if a bus is overloaded with work, the usable lifetime is reduced and the bus will have to be replaced earlier than expected. In fact, the travel distances of actual buses vary considerably, as can be seen in the example in Fig. 2. The figure shows the annual travel distances of buses in 2014 for a company Viação Garcia, which is one of the five biggest bus transportation companies in Brazil. It was founded in 1934 and currently has a fleet of more than 500 buses, whose total travel distance is about 5.5 million kilometers per month. From the figure, we can observe that some buses have traveled less than 20,000 km, whereas others have traveled more than 200,000 km. In the model presented in this paper, we consider a quadratic penalty function for traveled distances greater than the ideal traveling distance so that buses need not be replaced earlier than expected.

Bus maintenance can be divided into four categories: daily inspection, preventive maintenance, corrective maintenance, and predictive maintenance. Daily inspection consists of a list of items that need to be checked before starting a trip. This type of inspection does not require any specialized facility and can easily be executed by the driver. Corrective maintenance is executed if a bus has an unexpected breakdown. Because of the random nature of this maintenance, it cannot be scheduled in advance. A breakdown during a trip is extremely disruptive and costly, both financially and (arguably more importantly) reputationally. Predictive maintenance consists of operations that help to check the condition of in-service equipment, such as the fuel indicator and the check engine light. Although both preventive and predictive maintenance can be scheduled, we focus here on preventive maintenance because of the online characteristics of predictive maintenance.

The amount of daily preventive maintenance that can be executed at a maintenance facility is limited by the number of employees and the capacity of the facility. In real-world situations, if there are insufficient workers at a facility to accomplish all the daily maintenance tasks assigned to that facility, it is common either to call upon expensive part-time workers or, in the worst case, to postpone some maintenance tasks. Figure 3 shows the number of maintenance tasks scheduled at a facility for each day of a year. On some days, the facility is used beyond its standard capacity, whereas on other days the labor is wasted. In our model, we quadratically penalize the number of maintenance tasks in excess of a planned number.

First, we formulate the problem as a quadratic programming (QP) problem and investigate the size of problem instances that a commercial solver can handle. We then propose a heuristic algorithm based on the framework of greedy randomized adaptive search procedure (GRASP), which is a well-known metaheuristic framework that repeats applying local search to solutions generated by randomized greedy methods (Feo and Resende, 1989). Our GRASP uses two different greedy algorithms to generate initial solutions. Next, we show how to improve the neighborhood search by using an ordered list. Finally, we present computational results for several instances that were randomly generated based on real-world data so as to simulate actual bus transportation systems.
2. Literature Review

Any transportation company that has to assign buses to cover a given set of timetabled trips is faced with the problem of bus scheduling. Such a schedule must take several practical requirements into consideration, such as vehicle type, multiple depots, and other extensions. In the literature, an optimal schedule is one that minimizes the fleet size, the deadhead time/distance, and/or the operational costs.

Vehicle scheduling has been studied extensively over the past 40–50 years. Various papers give overviews of the models and solution approaches that are adopted (see Bodin and Golden (1981), Daduna and Paixão (1995), Bunte and Kliewer (2009) and Wren et al. (2003)). Further considerations such as robustness and the integration of vehicle and crew scheduling are addressed by papers such as Békési et al. (2009) and Mesquita et al. (2009). Various practical extensions are also considered, such as multiple vehicles types (see Hassold and Ceder (2014)), which is NP-hard even for the single-depot case (see Lenstra and Rinnooy Kan (1981)), variable arrival and departure times (which is NP-hard even for one vehicle and one depot; see Savelsbergh (1985)), and route constraints (see Bodin and Golden (1981)).

The single depot vehicle scheduling problem (SDVSP) is known to be solvable in polynomial time (Bunte and Kliewer, 2009). The minimum fleet size for the single depot case was first solved by Saha (1970); however, the model does not consider the operational costs. The problem with operational costs was later solved by Orloff (1976), where the SDVSP was formulated as an assignment problem.

The multi-depot vehicle scheduling problem (MDVSP) was proved to be NP-hard even for two depots (see Bertossi et al. (1987)). Some exact algorithms, presented by Fischetti et al. (1989), Forbes et al. (1994) and Kliewer et al. (2006), were able to solve instances with up to thousands of trips.

Pepin et al. (2009) presented and compared five representative heuristics: a tabu search, a large neighborhood search, a truncated column generation, a Lagrangian heuristic and a truncated branch-and-cut. The instances were composed of 1500 trips and up to 8 depots. Guedes and Borenstein (2015) applied an improved column generation method to solve the multi-depot multi-vehicle type scheduling problem. Otsuki and Aihara (2016) proposed an algorithm based on a variable depth search framework, which could find better results than previous local search based heuristics.

Maintenance scheduling is a combinatorial optimization problem for which several methods have been proposed, such as genetic algorithms, simulated annealing, integer programming, and branch-and-bound techniques (Adonyi et al., 2013). The majority of the relevant scientific literature deals with maintenance in manufacturing, but a portion concentrates specifically on bus maintenance. A mathematical programming approach was introduced by Haghani and Shafahi (2002). Their model aims at designing daily maintenance schedules for buses so as to minimize interruptions to the daily operating schedule and maximize the utilization of the maintenance facility. Zhou et al. (2004) proposed a multi-agent model that not only generates good schedules but can also react dynamically to unforeseen events.

Most integrated models in the transportation literature focus on integrating the problems of vehicle scheduling and crew scheduling. A few models in the airline transportation literature consider maintenance integration (e.g., Safaei and Jardine (2017), Papaikakis (2009) and Barnhart et al. (1998)). However, none of those models can be applied directly to a mileage-based maintenance system used in bus transportation.
3. Problem Description

In this section we describe the integrated maintenance and bus scheduling problem, and then we propose a mathematical formulation.

3.1. Nomenclature

Before we propose a mathematical model for the integrated maintenance and bus scheduling problem, we define the following terms:

- Trip: a planned journey from one depot to another;
- Arc: a feasible connection between two trips that can be executed by the same bus;
- Maintenance: a period of time, between trips, when a bus needs to be out of service for repair;
- Duty: a feasible sequence of trips executed by a bus (an empty sequence is considered feasible);
- Schedule: a feasible assignment of duties and maintenance tasks to buses.

A deadhead usually signifies a travel of a bus between two depots without a passenger (i.e., an out-of-service travel); however, for convenience, we also use this term throughout this paper to represent an empty travel between the same depot so that every arc from a trip to another corresponds to the deadhead travel of a bus necessary to execute both of the two trips.

3.2. The integrate bus and maintenance scheduling problem

We are given a set of \( n \) trips \( J = \{1, 2, \ldots, n\} \), a set of \( m \) buses \( B = \{1, 2, \ldots, m\} \), a set of \( l \) depots \( L = \{1, 2, \ldots, l\} \), the maximum mileage \( \mu_{\text{max}} \) that a bus can travel without maintenance, the minimum mileage \( \mu_{\text{min}} \) that a bus must travel prior to maintenance (i.e., the mileage between two successive maintenance events for a bus must be in the interval \([\mu_{\text{min}}, \mu_{\text{max}}]\)), the amount of time \( \theta \) spent on a maintenance task, and the number of maintenance tasks \( \eta \) that can be performed simultaneously at depot \( l \in L \). For each trip \( j \in J \), we are also given the starting time \( \tau_s(j) \), the ending time \( \tau_e(j) \), the starting location \( \lambda_s(j) \), the ending location \( \lambda_e(j) \), and the total distance \( \Delta(j) \) of the trip. Even if two trips have the same starting and ending locations, they need not have the same duration or distance. The minimum (deadhead) distance \( D_{\text{min}}(l_1, l_2) \) and the minimum (deadhead) duration \( T_{\text{min}}(l_1, l_2) \) between two depots are given for all \( l_1, l_2 \in L \). For each bus \( b \in B \), we are given an ideal traveling distance \( d_b^* \), a penalty coefficient \( \alpha \), and the initial mileage \( \hat{d}^b \) that signifies the distance traveled since the most recent maintenance before the scheduling period. For each depot \( l \in L \), we are given the number \( r_l^* \) of planned daily maintenance tasks, and a penalty coefficient \( \beta_l \).

The problem requires duties to be assigned to buses in such a way that a weighted sum of the following objectives over all buses \( b \) and facilities \( l \) is minimized:

1. the distance traveled beyond \( d^*_b \) and
2. the number of executed maintenance tasks in excess of \( r_l^* \).

In such a schedule, the following constraints must be satisfied:

(a) Each trip is assigned to exactly one bus.
(b) Each bus does not perform more than one duty.
(c) The interval between two consecutive maintenance tasks for a bus is not more than \( \mu_{\text{max}} \) and not less than \( \mu_{\text{min}} \).
(d) For each depot \( l \in L \), the number of simultaneous maintenance tasks does not exceed the capacity \( \eta_l \).
(e) Each maintenance task is scheduled either immediately before or after a trip and at either the starting or ending location.

3.3. Quadratic integer programming

Let trips \( 0 \) and \( n + 1 \) be dummy trips, and define the following notation:

- \( d_{ij} \): the distance traveled when executing trip \( j \) after trip \( i \), that is, \( d_{ij} = \Delta(j) + D_{\text{min}}(\lambda_e(i), \lambda_s(j)) \);
The set of days $[0, 1, 2, \ldots, \max_{i \in J} \lfloor \tau_{l}(i) \rfloor]$ (we assume for convenience that each integer time increment represents a day, e.g., $t = 3.5$ represents midday on day 3);

- $A$: the set of arcs representing all possible deadhead travels, $\{(i, j) \mid \tau_{l}(j) \geq \tau_{l}(i) + T_{\min}(\lambda_{l}(i), \lambda_{l}(j)) \}$, $i, j \in J \cup \{0, n+1\}$;

- $A^{l-2}$: the set of arcs $(i, j) \in A$ such that trip $j$ starts in time interval $[t_1, t_2]$, $\{(i, j) \in A \mid t_1 \leq \tau_{l}(j) \leq t_2\}$;

- $M$: the set of arcs representing deadhead travels during which maintenance is possible, $\{(i, j) \in A \mid \tau_{l}(j) \geq \tau_{l}(i) + T_{\min}(\lambda_{l}(i), \lambda_{l}(j)) + \theta) \}$;

- $P^{l-2}$: the set of arcs $(i, j) \in M$ such that a maintenance task can be executed immediately after trip $i$ that ends in time interval $[t_1, t_2]$ at depot $l$, $\{(i, j) \in M \mid t_1 \leq \tau_{l}(i) < t_2, \lambda_{l}(i) = l\}$;

- $Q^{l-2}$: the set of arcs $(i, j) \in M$ such that a maintenance task can be executed immediately before trip $j$ and can be started in time interval $[t_1, t_2]$ at depot $l$, $\{(i, j) \in M \mid t_1 \leq \tau_{l}(j) - \theta < t_2, \lambda_{l}(j) = l\}$;

- $R^{l-2}$: the set of arcs $(i, j) \in M$ such that a maintenance task can be executed immediately after trip $i$ that ends in time interval $[t_1, t_2]$ at depot $l$, $\{(i, j) \in M \mid t_1 \leq \tau_{l}(i) \leq t_2, \lambda_{l}(i) = l\}$;

- $S^{l-2}$: the set of arcs $(i, j) \in M$ such that a maintenance task can be executed immediately before trip $j$ and can be started in time interval $[t_1, t_2]$ at depot $l$, $\{(i, j) \in M \mid t_1 \leq \tau_{l}(j) - \theta \leq t_2, \lambda_{l}(j) = l\}$.

We define the decision variables that will be used in the mathematical programming formulation:

\[
\begin{align*}
    x_{0j}^b &= \begin{cases} 1 & \text{if $j$ is the first trip executed by bus $b$} \\ 0 & \text{otherwise}; \end{cases} \\
    x_{n+1}^b &= \begin{cases} 1 & \text{if $i$ is the last trip executed by bus $b$} \\ 0 & \text{otherwise}; \end{cases} \\
    x_{ij}^b &= \begin{cases} 1 & \text{if trip $j$ is executed after trip $i$ by bus $b$} \\ 0 & \text{otherwise}; \end{cases} \\
    p_{ij}^b &= \begin{cases} 1 & \text{if maintenance for bus $b$ is done immediately after trip $i$ before starting the deadhead travel to the} \\ \text{starting location of trip $j$} \\ 0 & \text{otherwise}; \end{cases} \\
    q_{ij}^b &= \begin{cases} 1 & \text{if maintenance for bus $b$ is done immediately before trip $j$ after the deadhead travel from the} \\ \text{location of trip $i$} \\ 0 & \text{otherwise}. \end{cases}
\end{align*}
\]

We also define the auxiliary variables:

\[
\begin{align*}
    w_b^i &: \text{the distance traveled by bus $b$ in excess of $d_{hi}^r$}; \\
    z_{il} &: \text{the number of executed maintenance tasks in excess of $r_{ij}^l$ in depot $l$ on day $t$}.
\end{align*}
\]

Figure 4 illustrates the sets defined above for an example in which there are three depots 1, 2 and 3 and three trips $a$, $b$ and $c$ from depot 1 to 2, 2 to 3, and 2 to 1, respectively. In Fig. 4, the trips that can be done after trip $a$ are $b$ and $c$, and the trip that can be done after $b$ is only $c$. Thus the set of all possible arcs $A$ (i.e., feasible pair of trips that can be executed by the same bus) is composed of arcs $(a, b), (b, c)$ and $(a, c)$. The set $A^{20,22}$ is composed of the arcs such that the trip after the deadhead starts in the time interval $[20, 22]$, and hence $A^{20,22} = \{(a, b)\}$. Suppose that the time $\theta$ required to execute a maintenance task is three hours, and the time to travel from depot 3 to 2 is one hour. Then the arcs that have enough time to execute maintenance tasks are $(b, c)$ and $(a, c)$, and thus $M = \{(b, c), (a, c)\}$. For arc $(b, c)$, the maintenance task can be executed in depot 3 after trip $b$ in the time interval $[23, 26]$ or in depot 2 before trip $c$ in the time interval $[24, 27]$. For arc $(a, c)$, the maintenance task can be executed after trip $a$ in depot 2 in the time interval $[21, 24]$ or before trip $c$ in depot 2 in the time interval $[24, 27]$. The set $P_{24}^{24}$ (resp., $R_{24}^{24}$) is the subset of $M$ such that a maintenance task can be executed in depot 2 immediately after a trip that ends in the time interval $[0, 24]$ (resp., $[0, 24]$); hence $P_{24}^{24} = \{(a, c)\}$ and $R_{24}^{24} = \{(a, c)\}$. The set $S_{24}^{24}$ (resp., $S_{24}^{24}$) is a subset of $M$ such that a maintenance task can...
be executed immediately before a trip to depot 2 and can be started in the time interval [0, 24) (resp., [0, 24]); hence $Q_2^{0.24} = \emptyset$ and $x_2^{0.24} = \{(b, c), (a, c)\}$. Suppose that there are two buses and that trip $a$ and $c$ are assigned to one bus and trip $b$ is assigned to the other bus; in this case, the duties are the sequences of trips $(a, c)$ and $(b)$.

\[
A = \{(a, b), (b, c), (a, c)\} \\
M = \{(b, c), (a, c)\} \\
A^{0.22} = \{(a, b)\} \\
P_2^{0.24} = \{(a, c)\} \\
Q_2^{0.24} = \emptyset \\
R_2^{0.24} = \{(a, c)\} \\
S_2^{0.24} = \{(b, c), (a, c)\}
\]

For convenience, we say that bus $b$ travels arc $(i, j)$ if trips $i$ and $j$ are both executed by bus $b$ and $j$ is executed by $b$ immediately after $i$. We also say that maintenance is scheduled for arc $(i, j)$ if maintenance is scheduled before or after the deadhead corresponding to the arc (i.e., after trip $i$ and before the deadhead, or after the deadhead and before trip $j$).

We consider the following QP problem, in which constraint (c) is relaxed to a necessary condition for the simplicity of the formulation:

\[
\begin{align*}
\text{min} & \quad \alpha \sum_{b \in B} w_b^2 + \sum_{t \in T} \sum_{b \in B} \beta_t x_{b,t}^2 \\
\text{s.t.} & \quad w_b \geq 0 \quad \forall b \in B, (2) \\
& \quad w_b \geq -d_b^i + \sum_{(i, j) \in A} d_{ij} x_{ij}^b \quad \forall b \in B, (3) \\
& \quad z_b \geq 0 \quad \forall l \in L, \forall t \in T, (4) \\
& \quad z_b \geq -r_i^j + \sum_{b \in B} \left( \sum_{(i, j) \in A} p_{ij}^b + \sum_{(i, j) \in Q_{ij}^{t+1}} q_{ij}^b \right) \\
& \quad \sum_{(i, j) \in A} x_{ij}^b = 1 \quad \forall b \in B, (5) \\
& \quad \sum_{b \in B} \sum_{j \in j \in A} x_{0j}^b = 1 \quad \forall b \in B, (6) \\
& \quad \sum_{b \in B} \sum_{i \in i \in A} x_{ij}^b = 1 \quad \forall b \in B, (7) \\
& \quad \sum_{b \in B} \sum_{k \in k \in A} x_{ik}^b = 0 \quad \forall b \in B, (8) \\
& \quad p_{ij}^b + d_{ij}^b - x_{ij}^b \leq 0 \quad \forall b \in B, \forall j \in J, (9) \\
& \quad \mu^{\min} \sum_{t \in T} \left( \sum_{(i, j) \in A} p_{ij}^b + \sum_{(i, j) \in A} q_{ij}^b \right) - \delta^b - \sum_{(i, j) \in A} d_{ij} x_{ij}^b \leq 0 \quad \forall b \in B, (10) \\
& \quad \delta^b + \sum_{(i, j) \in A} d_{ij} x_{ij}^b - \mu^{\max} \sum_{t \in T} \left( \sum_{(i, j) \in A} p_{ij}^b + \sum_{(i, j) \in A} q_{ij}^b \right) \leq \mu^{\max} \quad \forall b \in B, (11)
\end{align*}
\]
4. Proposed Method

This section describes the proposed method. We first introduce two greedy heuristic algorithms that are used in our GRASP heuristic to generate initial solutions. Next, we present the penalty function and define the neighborhoods used in the local search procedure. Then, we present the proposed heuristic based on the GRASP framework, and a method to reduce the number of neighbors to be examined in the local search procedure.

4.1. Randomized Concurrent Scheduler Algorithm

We extend the well-known greedy heuristic process known as the concurrent scheduler algorithm proposed by Bodin (1981). We call the extended algorithm the randomized concurrent scheduler (RCS). The pseudocode of the RCS is given in Algorithm 1. The RCS begins by making copies \( J' \) and \( B' \) of the given set of jobs \( J \) and that of buses \( B \), respectively (line 2 of Algorithm 1). Then the list of trips are sorted chronologically according to their starting times (line 3), and from this list, trip \( j \) is chosen randomly (line 6). For the first trip among the trips that start after trip \( j \) has been completed, the

\[
\sum_{b \in B} \left( \sum_{(i,j) \in P_{b,i}^*} p_{i,j}^b + \sum_{(i,j) \in Q_{i,j}^*} q_{i,j}^b \right) \leq \eta I \quad \forall I \in L, \forall t \in \{ \tau_c(k) + \theta | k \in J \} \cup \{ \tau_c(k) | k \in J \}, \quad (14)
\]

\( x_{i,j}^b \in \{0, 1\} \)

\( p_{i,j}^b, q_{i,j}^b \in \{0, 1\} \)

The objective function (1) minimizes the weighted sum of the squared distances traveled beyond \( d_b^* \) for all buses and the squares of the number of executed maintenance tasks in excess of \( r_l^* \) for all depots, where \( \alpha \) and \( \beta_l \) are penalty weights that specify the balance between the terms. The reason that we adopt quadratic penalty in the objective function (1) is explained as follows. Suppose that Fig. 5 represents the total traveled distance of two buses in two different schedules. Because it is more preferable to reduce the exceeding mileage uniformly among buses, Fig. 5 (a) represents a better schedule than Fig. 5 (b). If we linearly penalize the distances traveled by buses 1 and 2 in excess of \( d_1^* \) and \( d_2^* \), both schedules have the same penalty cost \( w_1 + w_2 = w + w = 2w \) and \( w_1 + w_2 = 0 + (2w) = 2w \). On the other hand, the quadratic objective function penalizes the schedule in Fig. 5 (b) higher: the cost for the schedule of Fig. 5 (a) is \( w_1^2 + w_2^2 = w^2 + w^2 = 2w^2 \), while that of Fig. 5 (b) is \( w_1^2 + w_2^2 = 0^2 + (2w)^2 = 4w^2 \). For the same reason, we also quadratically penalize \( z_b \).

Fig. 5 Total traveled distance for two different schedules

Constraints (2) and (3) ensure that the value of \( w_b \) is equal to the distance traveled in excess of \( d_b^* \) when \( w_b \) is minimized. Constraints (4) and (5) ensure that the value of \( z_b \) is equal to the number of executed maintenance tasks in excess of \( r_l^* \) when \( z_b \) is minimized. Constraints (6) and (7) define the initial and final locations of each bus, and constraints (8) and (9) mean that each trip (except the dummy trips) is covered exactly once by one bus. Equation (10) is the statement of flow conservation for each bus.

Constraints (11)–(14) are related to maintenance. Constraint (11) ensures that no maintenance for bus \( b \) is scheduled for arc \((i,j)\) unless the arc is traveled by bus \( b \). Constraints (12) and (13) are necessary conditions for condition (c), which requires the interval between two maintenance tasks to be not less than \( \mu_{min} \) and not more than \( \mu_{max} \). The necessary and sufficient conditions are too complex to be written concisely in an integer program. Finally, constraint (14) ensures that not more than \( \eta_I \) maintenance tasks are executed simultaneously at depot \( l \).
algorithm chooses the trip $k$ with the earliest start time among those that can be executed after trip $j$ (line 10). The next trip after trip $k$ is selected in the same way, and this process is repeated until no more trips can be executed. Restarting from the above-mentioned trip $j$, the algorithm selects the latest-starting trip that can be executed before trip $j$ (line 15), and this process is repeated until no more trips can be executed. This process of constructing a duty can be executed $O(n)$ time, because the trips are sorted in line 3 and hence lines 10 and 15 can be done in $O(n)$ time in total per duty, by tracing the sorted list twice, once for line 10 during the entire repetitions of lines 8–13, and once in the opposite direction for line 15 during the entire repetitions of lines 15–21. Of the available buses, the devised duty is assigned to the one whose ideal traveling distance is closest to the distance of the devised duty (line 23). (The algorithm proposed by Bodin (1981) is a special case of Algorithm 1 such that in line 6, the the earliest possible trip in $J$ is chosen as trip $j$, and the operation of growing the duty in the opposite direction (i.e., lines 15–21) is not contained). At the end, if there are trips that remain unassigned, they are added to a trip set $N$ (line 26). We explain how we handle the unassigned trips in $N$ in Section 4.3.

**Algorithm 1** Randomized concurrent scheduler (RCS)

1. **def** Random_Concurrent_Scheduler($J$, $B$):
2. $J' := J$, $B' := B$, $N' := \emptyset$;
3. sort the trips chronologically in order of starting time;
4. **while** $J' \neq \emptyset$ and $B' \neq \emptyset$:
5. $Duty :=$ empty list;
6. $j :=$ a trip chosen randomly from $J'$;
7. $i := j$;
8. add $j$ to the rightmost place of $Duty$;
9. remove $j$ from $J'$;
10. search in $J'$ for the trip $k$ with minimum $\tau_s(k)$ among those that can be executed after $j$;
11. **if** such a trip $k$ is found:
12. $j := k$;
13. go to step 8;
14. **end if**;
15. search in $J'$ for the trip $k$ with maximum $\tau_s(k)$ among those that can be executed before $i$;
16. **if** such a trip $k$ is found:
17. $i := k$;
18. add $i$ to the leftmost place of $Duty$;
19. remove $i$ from $J'$;
20. go to step 15;
21. **end if**;
22. $Total :=$ total distance of $Duty$;
23. assign trips in $Duty$ to a bus $b \in B'$ such that $(d_b^+ - Total)$ is minimized;
24. remove $b$ from $B'$;
25. **end while**;
26. $N' := J'$;

### 4.2. Random Greedy Assignment Algorithm

We propose another greedy algorithm, which we call the random greedy assignment (RGA) algorithm. Whereas the RCS greedily arranges trips so as to reduce the intervening time interval, the RGA greedily assigns trips to buses so that the overall difference in mileage among the buses is minimized. The pseudocode of the RGA is given in Algorithm 2. The RGA begins by making copies $J'$ and $B'$ of the given set of jobs $J$ and that of buses $B$, respectively (line 2 of Algorithm 2). Then the list of buses $B'$ is sorted in non-increasing order of ideal traveling distance (line 4). The RGA randomly chooses a trip $i$ from an unassigned trip set $J'$ (line 5), and the algorithm tries to assign trip $i$ to the bus at the beginning of the bus list. If the bus cannot execute trip $i$, the RGA tries to assign trip $i$ to the next bus in the bus list until it finds a bus to execute trip $i$ (while loop in line 7). For a trip that is not able to be assigned to any bus in the bus list, we remove it from $J'$ and add it to a trip set $N$ (lines 10 and 11). If trip $i$ is assigned to bus $b$, the ideal traveling distance of bus $b$ is reduced.
by the distance traveled and this process is repeated until the list $J'$ becomes empty (while loop in line 3). We explain how we handle the unassigned trips in $N$ in Section 4.3.

Algorithm 2 Random greedy assignment (RGA)

```
def Random_Greedy_Assignment_Algorithm($J, B$):
    $J' := J, B' := B, N := \emptyset$;
    while $J' \neq \emptyset$:
        sort $B'$ in non-increasing order of ideal distance;
        $i := \text{a trip chosen randomly from } J'$;
        $k := 1$;
        while trip $i$ cannot be assigned to the $k^{\text{th}}$ bus in $B'$:
            $k := k+1$;
        if $k > \text{the length of } B'$ :
            add $i$ to $N$;
            remove trip $i$ from $J'$;
            go to step 3;
        end if;
    end while;
    assign trip $i$ to the $k^{\text{th}}$ bus in $B'$;
    remove trip $i$ from $J'$;
end while;
```

4.3. Penalty Function

Allowing the search to visit the infeasible region alleviates the difficulty of searching solely within the feasible region, which we encounter in those instances whose feasible region is very small or consists of many separate small regions. It is known that such strategy produces good results (see Yagiura et al. (2004)).

Consider the following reasons for a solution being infeasible: (a) a bus is required to execute more than one trip at the same time; (b) more than $\eta_l$ maintenance tasks must be executed simultaneously in depot $l \in L$; (c) maintenance is required to occur either before $\mu_{\text{min}}$ or after $\mu_{\text{max}}$; (d) not all trips are assigned.

In the local search procedure, we allow the search to visit solutions that violate (c) and (d) above, and we penalize the degree of violation. Given penalty coefficients $\gamma$ and $\kappa$, the evaluation function $f(S)$ for solution $S$ is defined to be the sum of the following four parts:

(i) $\alpha \cdot \text{(max([total distance traveled by bus $b$] − $d_o^b, 0)))^2,}$
(ii) $\beta_l \cdot \text{(max([total number of maintenance tasks executed in depot $l$ on day $t$] − $r_t^l, 0)))^2,}$
(iii) $\gamma \cdot \text{(max($\mu_{\text{min}}$ − [mileage after most recent maintenance], 0) +max([mileage after final maintenance] − $\mu_{\text{max}}$, 0))},$
(iv) $\kappa \sum_{j \in N} \Delta(j)(\tau_e(j) − \tau_s(j)).$

4.4. Local Search

A local search is a heuristic method for solving computationally hard optimization problems. Local search algorithms begin with an initial solution $S$ and then move repeatedly to a better solution $S'$ with respect to $f$ in a neighborhood $N(S)$, which is the set of solutions obtainable from $S$ by applying local changes (i.e., neighborhood operations) to $S$, until no better solution exists in the neighborhood. The neighborhoods used in our local search are as follows:

- insert neighborhood $N_{\text{ins}}(S) = \{S' \mid S' \text{ is obtainable from } S \text{ by adding one trip after another}\},$
- swap neighborhood $N_{\text{swp}}(S) = \{S' \mid S' \text{ is obtainable from } S \text{ by exchanging a trip of one bus with a trip of another bus}\},$
- add neighborhood $N_{\text{add}}(S) = \{S' \mid S' \text{ is obtainable from } S \text{ by adding a maintenance task before a trip}\cup\{S'' \mid S'' \text{ is obtainable from } S' \text{ by adding a maintenance task after a trip}\},$
- remove neighborhood $N_{\text{rem}}(S) = \{S' \mid S' \text{ is obtainable from } S \text{ by removing a maintenance task}\},$
• slide neighborhood \( N_{sl}(S) = \{ S' \mid S' \text{ is obtainable from } S \text{ by postponing a maintenance task by one trip} \} \cup \{ S'' \mid S'' \text{ is obtainable from } S \text{ by bringing forward a maintenance task by one trip} \}. \)

The local search procedure used in this paper is given in Algorithm 3.

**Algorithm 3** Best-improvement local search

```
def Local_Search(S, N):
    k := 1, S(1) := S;
    if \{ S ∈ N(S(k)) \mid f(S) < f(S(k)) \} = ∅:
        output S(k) and stop;
    else:
        k := k+1;
        S(k) := a best-improved solution from \{ S ∈ N(S(k-1)) \mid f(S) < f(S(k-1)) \};
        go to step 3;
go to step 3;
```

4.5. Entire Framework of the Proposed Heuristic

In our GRASP heuristic, to increase the variety of initial solutions, the initial solution of each call to the local search is generated randomly by Algorithm 1 or 2 (with a 50% chance for either). The solution is then improved through a sequence of local search procedures based on the five neighborhoods. The process of generating an initial solution and improving it with a local search is repeated until a time limit \((Time\_limit)\) is reached. The pseudocode of the proposed GRASP is given in Algorithm 4.

**Algorithm 4** Our GRASP

```
def GRASP():
    while the computation time ≤ Time\_limit:
        with probability 0.5, generate an initial schedule \( S \) using the RCS,
        else generate an initial schedule \( S \) using the RGA;
        while the solution \( S \) is improved by:
            S := Local_Search(S, Nswp);
            S := Local_Search(S, Nins);
            do in a random order (execute each line only once):
                S := Local_Search(S, Nan);
                S := Local_Search(S, Nrem);
            end do;
        end while;
        if a better feasible solution was found:
            update the incumbent solution;
        end if;
    end while;
```

4.6. Reduction of Neighborhood Size

The sizes of the add, remove, and slide neighborhoods scale linearly with the number of trips, whereas those of the insert and swap neighborhoods scale quadratically with the number of trips. For the insert and swap neighborhoods, we propose the following method to reduce the number of neighbors to be examined in the local search procedure:

1. For each trip \( j ∈ J \), create a list of all possible trips \( H_j \) that can be executed after trip \( j \), that is, \( H_j = \{ i ∈ J \mid τ_i(i) ≥ τ_j(j) + T_{min}(λ_e(j), λ_s(i)) \} \).
2. For each \( j \), sort all elements in \( H_j \) chronologically according to ending time.

3. When searching for a trip to be inserted after trip \( j \) and before trip \( k \), starting from the beginning of list \( H_j \), evaluate the insertions of \( i \in H \) until \( \tau_e(i) > \tau_s(k) \) holds. Because the list is sorted, all the remaining trips in \( H_j \) produce infeasible schedules.

4. When searching for a trip to be swapped with trip \( i \) that is preceded by trip \( j \) and followed by trip \( k \), starting from the beginning of list \( H_j \), evaluate the swaps of \( i \) with \( h \in H \) until \( \tau_e(h) > \tau_s(k) \) holds. Because the list is sorted, all the remaining trips in \( H_j \) produce infeasible schedules.

It costs \( O(n^2) \) of memory to store all \( n \) lists, and it takes \( O(n^2 \log n) \) time to create and sort them. However, this procedure is executed only once at the beginning of the search. We refer to the GRASP algorithm with this improved neighborhood-search procedure as GRASP$^\ast$.

5. Computational Results

5.1. Computational Environment

All the algorithms presented in this paper were implemented in Python (version 3.5.2) and tested on a computer with a dual-core 64-bit 2.20 GHz Intel i7 processor and 8 GB of RAM. The commercial optimization solver used to solve the QP problem was Gurobi (version 7.0.1) for Windows (64 bit). We set the parameter \( \text{Time\_limit} \) to 3600 s and the penalty coefficients \( \alpha, \beta (\forall l \in L) \), \( \gamma \), and \( \kappa \) to 1, 3000, \( 10^6 \), and \( 10^4 \), respectively.

5.2. Instances

The test instances were generated using the method proposed by Fischetti et al. (2001) for the multi-depot vehicle scheduling problem. First, \( l \) depots were located randomly in a circle of radius 250 km. The deadhead distance from one depot to another was then set as the Euclidean distance between them. To generate a trip, an average speed for the trip was chosen in the interval [30, 80] km/h with a probability based on the average velocity distribution of Viação Garcia buses. Next, the start and end points of the trip were chosen randomly among the depots. The starting time of the trip was chosen randomly from the schedule horizon (because the starting time and average speed are defined, it is easy to calculate the ending time of the trip). The distance of the trip to be traveled for operation was set to the deadhead distance between its start and end points times a random factor in the interval [1.1, 1.3]. Finally, the deadhead duration from one depot to another was set to be the deadhead distance divided by 80 (i.e., the maximum speed).

As for the maintenance parameters, the maintenance duration was set to 3 h and the minimum and maximum mileage intervals between maintenance tasks were set to 3,000 km and 5,000 km, respectively. The number of maintenance tasks that can be executed simultaneously in each depot was set randomly as zero (i.e., no maintenance tasks can be executed), one, or two. The initial mileage was set using the mileage distribution of Viação Garcia buses. The main characteristics of each generated instance, whose IDs range from 1 to 21, are summarized in Table 1.

Table 1 Instances characteristics

<table>
<thead>
<tr>
<th>ID</th>
<th>#Trips</th>
<th>#Buses</th>
<th>#Depots</th>
<th>#Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>10</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>10</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>20</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>10</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>10</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>20</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>10</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>10</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>10</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

5.3. Parameter Tuning

We examine the influence of the penalty coefficients. First, we keep \( \gamma = 10^6 \) and test \( \kappa \in \{10^3, 10^4, 10^5, 10^6\} \). Next, we keep \( \kappa = 10^4 \) and test \( \gamma \in \{10^3, 10^4, 10^5, 10^6\} \). The results in Table 2 shows that the best tested option is to set \( \gamma = 10^6 \) and \( \kappa = 10^4 \).

5.4. Local Search Strategy

In this section, we compare several strategies of local search. First, we examine the order in which the neighborhoods are searched by the local search algorithm. Next, we examine if it is better to search each neighborhood until it reaches a locally optimal solution or to search neighborhoods alternately in such a way that whenever an improved solution is found in a neighborhood, another neighborhood is searched next, and then the remaining neighborhoods are searched in a random order. In these three cases, the neighborhoods are searched by the local search algorithm. Next, we examine if it is better to search each neighborhood until it reaches a locally optimal solution or to search neighborhoods alternately in such a way that whenever an improved solution is found in a neighborhood, another neighborhood is searched next, and then the remaining neighborhoods are searched in a random order. In these three cases, the neighborhoods are searched alternately in a random order.

The “Ins, Swp, Random” column shows the case in which the insert neighborhood is searched first, the swap is searched next, and then the remaining neighborhoods are searched in a random order. The “Swp, Ins, Random” column shows the case in which the swap neighborhood is searched first, the insert is searched next, and then the remaining neighborhoods are searched in a random order. In these three cases, the neighborhoods are searched alternately in such a way that whenever an improved solution is found in a neighborhood, another neighborhood is searched, and the process is repeated until no improvement is found in any of the neighborhoods. In the following three columns, the orders of neighborhoods are the same, but each neighborhood is searched until a locally optimal solution with respect to that neighborhood is found. We can confirm that the results in column “Swp, Ins, Random” are better than the others, and hence we adopt this strategy in our GRASP.

Table 3 Different strategies for the local search

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>168990</td>
<td>168990</td>
<td>168990</td>
<td>168990</td>
<td>168990</td>
<td>168990</td>
</tr>
<tr>
<td>2</td>
<td>49965</td>
<td>49965</td>
<td>49965</td>
<td>49965</td>
<td>49965</td>
<td>49965</td>
</tr>
<tr>
<td>3</td>
<td>218696</td>
<td>218696</td>
<td>218696</td>
<td>218696</td>
<td>218696</td>
<td>218696</td>
</tr>
<tr>
<td>4</td>
<td>287456</td>
<td>275804</td>
<td>287456</td>
<td>287456</td>
<td>287456</td>
<td>287456</td>
</tr>
<tr>
<td>5</td>
<td>150795</td>
<td>150795</td>
<td>150795</td>
<td>150795</td>
<td>150795</td>
<td>150795</td>
</tr>
<tr>
<td>6</td>
<td>279103</td>
<td>279103</td>
<td>279103</td>
<td>279103</td>
<td>279103</td>
<td>279103</td>
</tr>
<tr>
<td>7</td>
<td>395300</td>
<td>375350</td>
<td>395300</td>
<td>395300</td>
<td>395300</td>
<td>395300</td>
</tr>
<tr>
<td>8</td>
<td>152867</td>
<td>152867</td>
<td>152867</td>
<td>152867</td>
<td>152867</td>
<td>152867</td>
</tr>
<tr>
<td>9</td>
<td>195184</td>
<td>195184</td>
<td>195184</td>
<td>195184</td>
<td>195184</td>
<td>195184</td>
</tr>
<tr>
<td>10</td>
<td>190743</td>
<td>190743</td>
<td>190743</td>
<td>190743</td>
<td>190743</td>
<td>190743</td>
</tr>
<tr>
<td>11</td>
<td>145821</td>
<td>145821</td>
<td>145821</td>
<td>145821</td>
<td>145821</td>
<td>145821</td>
</tr>
<tr>
<td>12</td>
<td>262736</td>
<td>254166</td>
<td>262736</td>
<td>262736</td>
<td>254166</td>
<td>262736</td>
</tr>
<tr>
<td>13</td>
<td>608974</td>
<td>555127</td>
<td>608974</td>
<td>608974</td>
<td>608974</td>
<td>608974</td>
</tr>
<tr>
<td>14</td>
<td>451396</td>
<td>4311463</td>
<td>451396</td>
<td>451396</td>
<td>4483835</td>
<td>4070487</td>
</tr>
<tr>
<td>15</td>
<td>3167504</td>
<td>2968450</td>
<td>3167504</td>
<td>3167504</td>
<td>343880</td>
<td>346078</td>
</tr>
<tr>
<td>16</td>
<td>5790965</td>
<td>5224489</td>
<td>5300213</td>
<td>577613</td>
<td>634407</td>
<td>5224489</td>
</tr>
<tr>
<td>17</td>
<td>9469023</td>
<td>8670859</td>
<td>9690411</td>
<td>10072085</td>
<td>8976425</td>
<td>8670859</td>
</tr>
<tr>
<td>18</td>
<td>5980904</td>
<td>6481472</td>
<td>5014752</td>
<td>6387698</td>
<td>6481472</td>
<td>6129712</td>
</tr>
<tr>
<td>19</td>
<td>4973054</td>
<td>447502</td>
<td>5663596</td>
<td>5663596</td>
<td>556392</td>
<td>5473408</td>
</tr>
<tr>
<td>20</td>
<td>743057</td>
<td>571454</td>
<td>816737</td>
<td>886831</td>
<td>5473408</td>
<td>504734</td>
</tr>
<tr>
<td>21</td>
<td>736728</td>
<td>504734</td>
<td>667865</td>
<td>724008</td>
<td>504734</td>
<td>703489</td>
</tr>
</tbody>
</table>
5.5. Results

We compared the results obtained using Gurobi with those obtained using GRASP and GRASP∗ (see Table 4), where the latter two were run until Time_limit elapsed. For each method, the time given in Table 4 is the time at which the best solution was found. It can be seen that our algorithms performed well for small instances. For every instance for which Gurobi could not find an optimal solution, our algorithms found better upper bounds. Although we relaxed constraint (c) in the QP model, we noticed that the solutions found by Gurobi for the tested instances with ID 1–10 fulfill constraint (c).

We compared the extent to which our improved search method reduces the neighborhood size. For different instance sizes, we ran GRASP and GRASP∗ with 10 initial solutions that were chosen randomly but were the same for both procedures. We counted the number of times that the insert and swap local searches were called, and we counted the number of neighbors visited in each call. The average neighborhood size is the total number of visited neighbors divided by the number of calls. The results given in Table 5 indicate that our method reduced the size of the neighborhood by a factor greater than 90% for most large instances.

To examine how much the computation time was reduced with our improved search method, we ran GRASP until Time_limit elapsed. Next, we ran GRASP∗ until an equal or better solution was found. The results given in Table 6 indicate that our method reduced the computation time by an average of 70%.

Table 4 Comparison between Gurobi and GRASP

<table>
<thead>
<tr>
<th>ID</th>
<th>Gurobi</th>
<th>GRASP</th>
<th>GRASP∗</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower bound</td>
<td>Upper bound</td>
<td>Time (s)</td>
</tr>
<tr>
<td>1</td>
<td>168,990</td>
<td>168,990</td>
<td>3.1</td>
</tr>
<tr>
<td>2</td>
<td>49,965</td>
<td>49,965</td>
<td>280.3</td>
</tr>
<tr>
<td>3</td>
<td>218,696</td>
<td>218,696</td>
<td>3,162.5</td>
</tr>
<tr>
<td>4</td>
<td>119,280</td>
<td>278,373</td>
<td>3,600.0</td>
</tr>
<tr>
<td>5</td>
<td>150,795</td>
<td>150,795</td>
<td>195.1</td>
</tr>
<tr>
<td>6</td>
<td>279,103</td>
<td>279,103</td>
<td>2,455.3</td>
</tr>
<tr>
<td>7</td>
<td>148,618</td>
<td>403,420</td>
<td>3,600.0</td>
</tr>
<tr>
<td>8</td>
<td>152,867</td>
<td>152,867</td>
<td>870.1</td>
</tr>
<tr>
<td>9</td>
<td>175,968</td>
<td>192,988</td>
<td>3,600.0</td>
</tr>
</tbody>
</table>

Table 5 Neighborhood size reduction for insert and swap

<table>
<thead>
<tr>
<th>ID</th>
<th>Avg. size for insert</th>
<th>Avg. size for swap</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>625</td>
<td>300</td>
<td>66.4</td>
</tr>
<tr>
<td>11</td>
<td>625</td>
<td>300</td>
<td>66.1</td>
</tr>
<tr>
<td>12</td>
<td>2,500</td>
<td>1,225</td>
<td>66.1</td>
</tr>
<tr>
<td>13</td>
<td>10,000</td>
<td>4,950</td>
<td>66.1</td>
</tr>
<tr>
<td>14</td>
<td>40,000</td>
<td>19,900</td>
<td>66.1</td>
</tr>
<tr>
<td>15</td>
<td>90,000</td>
<td>44,850</td>
<td>66.1</td>
</tr>
<tr>
<td>16</td>
<td>160,000</td>
<td>79,800</td>
<td>66.1</td>
</tr>
<tr>
<td>17</td>
<td>250,000</td>
<td>124,750</td>
<td>66.1</td>
</tr>
<tr>
<td>18</td>
<td>250,000</td>
<td>124,750</td>
<td>66.1</td>
</tr>
<tr>
<td>19</td>
<td>250,000</td>
<td>124,750</td>
<td>66.1</td>
</tr>
<tr>
<td>20</td>
<td>250,000</td>
<td>124,750</td>
<td>66.1</td>
</tr>
<tr>
<td>21</td>
<td>250,000</td>
<td>124,750</td>
<td>66.1</td>
</tr>
</tbody>
</table>

6. Conclusion

We presented a new model that integrated the problems of bus scheduling and maintenance scheduling. The objective was to minimize the mileage differences between buses, the total distance traveled, and the differences in the number of daily maintenance tasks. We formulated the problem as a QP problem and solved it using a commercial optimization solver. We observed that the solver could not find an optimal solution (in our computational environment) for instances
Table 6  The computation time of GRASP∗ to obtain as good solutions as GRASP for large instances

<table>
<thead>
<tr>
<th>ID</th>
<th>Objective (s)</th>
<th>Time (s)</th>
<th>Objective (s)</th>
<th>Time (s)</th>
<th>Reduction(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>190,743</td>
<td>0.1</td>
<td>190,743</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>11</td>
<td>145,821</td>
<td>0.2</td>
<td>145,821</td>
<td>0.1</td>
<td>50.0</td>
</tr>
<tr>
<td>12</td>
<td>254,166</td>
<td>857.4</td>
<td>254,166</td>
<td>269.1</td>
<td>68.6</td>
</tr>
<tr>
<td>13</td>
<td>585,661</td>
<td>2,898.5</td>
<td>555,127</td>
<td>580.2</td>
<td>80.0</td>
</tr>
<tr>
<td>14</td>
<td>4,317,586</td>
<td>1,272.0</td>
<td>4,311,463</td>
<td>191.0</td>
<td>85.0</td>
</tr>
<tr>
<td>15</td>
<td>2,968,450</td>
<td>1,542.1</td>
<td>2,968,450</td>
<td>429.1</td>
<td>72.2</td>
</tr>
<tr>
<td>16</td>
<td>5,224,489</td>
<td>710.0</td>
<td>5,224,489</td>
<td>131.2</td>
<td>81.5</td>
</tr>
<tr>
<td>17</td>
<td>10,300,928</td>
<td>2,360.1</td>
<td>8,670,859</td>
<td>619.4</td>
<td>73.8</td>
</tr>
<tr>
<td>18</td>
<td>6,665,350</td>
<td>1,200.5</td>
<td>6,481,472</td>
<td>305.0</td>
<td>74.6</td>
</tr>
<tr>
<td>19</td>
<td>4,447,502</td>
<td>2,456.0</td>
<td>4,447,502</td>
<td>100.0</td>
<td>95.9</td>
</tr>
<tr>
<td>20</td>
<td>571,454</td>
<td>3,500.4</td>
<td>571,454</td>
<td>820.6</td>
<td>76.6</td>
</tr>
<tr>
<td>21</td>
<td>504,734</td>
<td>3,000.4</td>
<td>504,734</td>
<td>820.6</td>
<td>76.6</td>
</tr>
</tbody>
</table>

with more than 30 trips. We then proposed a GRASP heuristic with two different greedy algorithms for generating an initial solution. In the GRASP, we incorporated a method to improve the neighborhood search procedure by using an ordered list. This method reduced the size of the neighborhood appreciably without losing any improved solutions in the neighborhood. The results for small instances showed that our GRASP could find optimal solutions in much shorter computation times compared with the commercial solver. For instances whose optimal solutions were not known, our algorithm found better upper bounds than those found by the commercial solver.

References


