European Journal of Operational Research 283 (2020) 426–440

Contents lists available at ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

A Greedy Randomized Adaptive Search Procedure for the Orienteering Problem with Hotel Selection

Somayeh Sohrabi, Koorush Ziarati*, Morteza Keshtkaran

Department of Computer Engineering and Information Technology, Shiraz University, Shiraz, Iran

ARTICLE INFO

Article history:
Received 29 September 2017
Accepted 9 November 2019
Available online 16 November 2019

Keywords:
Metaheuristics
Orienteering Problem with Hotel Selection
Orienteering Problem
Greedy Randomized Adaptive Search Procedure
Dynamic programming

ABSTRACT

The Orienteering Problem with Hotel Selection (OPHS) is one of the most recent variants of the Orienteering Problem (OP). The sequence of hotels has a significant effect on the quality of the OPHS solutions. According to prior studies, it is not efficient to consider all feasible sequences of hotels for constructing a tour. For this reason, solution methods for this problem should be independent of the Total Number of Feasible Sequences of hotels (TNFS).

This paper proposes a novel approach based on a well-known metaheuristic called Greedy Randomized Adaptive Search Procedure (GRASP) to tackle the OPHS. In the previously introduced algorithms for this problem, the OP solutions among all feasible pairs of hotels are considered to construct a proper sequence of hotels. Our suggested GRASP algorithm does not follow this policy; instead, it uses a novel, potent, and fast dynamic programming method for hotel selection. This dynamic idea makes the introduced GRASP approach independent of the TNFS and solving the OPs.

Considering 400 benchmark instances with and five instances without known optimal values, the proposed method in this article obtains the optimal solutions for 231 instances (57.75%), as opposed to 174 instances (43.5%) for the best formerly suggested algorithm. GRASP constructs better tours than the state-of-the-art algorithm for 142 instances (35.5%) and finds new best results for three out of five instances with unknown optimal values. Moreover, GRASP can produce high quality solutions for 76 new large instances constructed using the instances of a similar problem to the OPHS.

1. Introduction

The Orienteering Problem (OP) is a combinatorial optimization problem introduced by Tsiligirides (1984). Assume a complete graph that each of its vertices has a specific score and can be visited at most once. The OP aims to discover a time-limited path with a maximum score in this graph, which must be started from and ended in specific vertices. Different variants of this problem have been defined by adding new constraints to the basic conditions (Gunawan, Lau & Vansteenwegen, 2016). The Orienteering Problem with Hotel Selection (OPHS) is one of the most recent forms of the OP (Divsalar, Vansteenwegen & Cattrysse, 2013). The goal of this article is to suggest a novel method to tackle the OPHS.

Consider a complete graph $G = (V, E)$, in which V is the set of vertices, and $E = \{(i, j)|i, j \in V\}$ is the set of edges. The vertex set includes h hotels $(H_0, H_1, \ldots, H_{h-1})$ and n nodes $(0, 1, \ldots, n-1)$. A specific score is assigned to each node, which is represented by s_i for $i \in V$; the visitation of a hotel, however, has no score. Two kinds of paths exist in the OPHS: trip and tour. A tour is a path that consists of a fixed number of contiguous trips. The first trip of a tour must start from H_0, and the origin of any other trip must be the end of the previous one. A trip visits several nodes within a specific time limit beginning from its start-hotel. The last trip must terminate in H_1, and the intermediate hotels of a tour can be any of the available hotels. Although a tour must visit each node at most once, there is not any restriction on the number of times that the hotels can be visited. The aim is to find a tour with a maximum score containing a fixed number of trips (D) so that each trip time limitation, $T_d (d = 0, 1, \ldots, D-1)$, is respected. In this article, the Euclidean distance between each pair of vertices in G is considered as the time it takes to travel among them (Divsalar et al., 2013).

The OPHS has several realistic applications, one of which is scheduling a multi-day tour for a tourist. Assume that a tourist wants to visit spectacular points in a region during a specific number of days. Since all of the spectacular locations cannot be visited in the available traveling time, the tourist must select some of them based on his preferences to have the most satisfaction when his travel terminates. He must also choose a hotel to rest at

* Corresponding author.
E-mail addresses: s.sohrabi@cse.shirazu.ac.ir (S. Sohrabi), ziarati@shirazu.ac.ir (K. Ziarati), mkeshtkaran@cse.shirazu.ac.ir (M. Keshtkaran).

https://doi.org/10.1016/j.ejor.2019.11.010
0377-2217/

the end of each day, which is the origin of his next day travel path. This problem can be modeled as an OPHS, in which the spectacular places are nodes whose scores depend on the interests of the tourist (Divsalar, Vansteenwegen, Sörensen & Cattrysse, 2014).

Proposed algorithms for the OPHS can also be used to tackle the technician routing problem. A technician has a specific working time limit each day. Thus, he cannot service all of his customers in one day and must select some of them based on their profits. The profit of each customer can be investigated from different aspects including loyalty, financial issues, and advertisements. The technician also carries limited equipment that is wholly used after serving a number of customers. Therefore, he must select a depot along his path after a while to re-equip (Divsalar et al., 2013). A multi-day path planning for a traveling salesman can also be tackled using the algorithms proposed for the OPHS. Scheduling the path of a truck driver who must park his car in a suitable place after driving for a specific maximum time is another problem that can be considered as an OPHS instance (Divsalar et al., 2014).

Two algorithms based on the Skewed Variable Neighborhood Search (SVNS) and the Memetic Algorithm (MA) have been proposed for the OPHS until now (Divsalar et al., 2013, 2014). The common idea of the MA and SVNS is to start the main procedure of the algorithm from feasible sequences of hotels that are likely to produce the maximum score. According to each feasible pair of hotels and each trip time limitation, an OP instance is considered (a "sub-OP") and solved heuristically. The constructed OP solutions are applied to determine an estimated score that can be obtained from each feasible sequence of hotels. The SVNS is completely dependent on the Total Number of Feasible Sequences of hotels (TNFS). It finds all feasible sequences of hotels, and then, according to the estimated scores, employs the best sequences to construct tours. The performance of the SVNS decreases dramatically when the TNFS gets larger. Therefore, the MA is the best algorithm that has been suggested to solve the OPHS (Divsalar et al., 2014).

In this paper, a novel approach based on the Greedy Randomized Adaptive Search Procedure (GRASP) is proposed to tackle the OPHS. This algorithm is independent of the TNFS and does not spend any time in solving sub-OPs to determine likely good feasible sequences of hotels. Instead, it uses an innovative dynamic programming approach to improve the sequence of hotels regarding the visitation order of the nodes along a considered tour.

The remainder of the article is organized as follows. Problems related to the OPHS are described in Section 2, and the standard GRASP procedure is overviewed in Section 3. Our algorithm is introduced in Section 4 and its complexity is analyzed in Section 5. Experimental results are investigated in Section 6, and Section 7 is devoted to the conclusion and future work.

2. Literature review

The OPHS relates to the variants of the Vehicle Routing Problem (VRP) incorporating concepts such as "intermediate facility" or "inter depot route". In these problems, besides the main stop (depot), there are also other stop choices for the vehicles where they can do their necessary intermediate tasks. The Periodic Vehicle Routing Problem with Intermediate Facilities (PVRP-IF) is the first variant of these problems (Angelelli & Speranza, 2002a). The PVRP-IF is an extension of the Periodic Vehicle Routing Problem (PVRP) (Gaudioso & Paletta, 1992). Tabu search is the first metaheuristic that was employed to solve this problem (Angelelli & Speranza, 2002a). Angelelli and Speranza (2002b) also modified their published model for the PVRP-IF to define a unique model for waste collection problems.

The Vehicle Routing Problem with Time Windows and Intermediate Facilities (VRPTW-IF) was introduced by Kim, Kim and Sahoo (2006). Two algorithms have been proposed to tackle the VRPTW-IF. The first is an extension of Solomon's insertion algorithm (Solomon, 1987) and the second approach is based on clustering. Crevier, Cordeau and Laporte (2007) defined a new extension of the Multiple Depot Vehicle Routing Problem (MDVRP) in which the cars may renew their capacity at some inter-depots along their paths. The heuristic that was developed to solve this problem is a combination of tabu search and integer programming. An approach based on tabu search, variable neighborhood search, and guided local search was proposed by Tarantilis, Zachariadis and Kiranoudis (2008) for the Vehicle Routing Problem with Intermediate Replenishment Facilities (VRPIRF).

Two forms of the Distance-constrained Capacitated Vehicle Routing Problem (DCVRP) were introduced by Kek, Cheu and Meng (2008). "Intermediate facility" is a critical concept in the descriptions of these two problems. A combination of tabu search and variable neighborhood search was suggested to solve the VRPTW-IF by Benjamin and Beasley (2010). Setak, Jalili Bolhassani, and Karimi (2014) defined an extension of the Location Routing Problem (LRP), which is named LRP with Intermediate Replenishment Facilities under capacity constraints (LRPIRF). They proposed two approaches based on the genetic algorithm and the tabu search method to solve the LRPIRF.

A hybrid heuristic was suggested for the Electric Vehicle-Routing Problem with Time Windows and recharging stations (E-VRPTW) by Schneider, Stenger and Goeke (2014). This problem is structurally similar to the VRPs with intermediate stop requests. An Adaptive Variable Neighborhood Search (AVNS) was applied to the Vehicle Routing Problem with Intermediate Stops (VRPIS) by Schneider, Stenger and Hof (2015). The Mixed Capacity Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF) was introduced by Willemse and Joubert (2016a), where four constructive heuristics were used to solve the problem. Willemse and Joubert (2016b) also employed an optimal approach and a quick-near optimal splitting procedure to tackle this problem.

The integration of a heterogeneous fixed fleet and a flexible assignment of destination depots in the waste collection VRP with intermediate facilities was investigated by Markov, Varone and Bierlaire (2016). The proposed AVNS for the VRPIS (Schneider et al., 2015) was also applied to tackle the battery swap station location-routing problem with capacitated electric vehicles by Hof, Schneider and Goeke (2017).

The "Hotel selection" term initially appeared in the Traveling Salesman Problem with Hotel Selection (TSPHS) (Vansteenwegen, Souffriau & Sörensen, 2012). In the TSPHS, the vertex set is composed of two subsets including customers and hotels. The aim of this problem is to find the best multi-day tour for the salesman in which all the customers are served. The number of working days (i.e., the number of trips), which must be minimized, is the primary measure that is used to evaluate the quality of a TSPHS tour. The secondary objective of this problem is to minimize the total traveling time. Contrary to the OPHS, the number of trips is not a parameter in the TSPHS and a feasible TSPHS tour must visit all the customers.

Four algorithms have been suggested to tackle the TSPHS since it was introduced. The first introduced algorithm is a multi-start heuristic that applies several local search operators to improve an initial TSPHS solution (Vansteenwegen et al., 2012). The second method is a combination of the memetic and tabu search metaheuristics. A memetic algorithm is used for the hotel selection while routing the customers is done by a tabu search approach embedded in the memetic algorithm as an improvement process (Castro, Sörensen, Vansteenwegen & Goos, 2013). The third algorithm for the TSPHS is based on variable neighborhood search. Although the two prior methods produce high-quality solutions considering the available benchmark instances for the TSPHS, they are

expensive in terms of computational effort. The primary focus of the third proposed algorithm is on forming a good tour in shorter execution time than the other approaches. For this purpose, a TSP path is created by the Lin-Kernighan heuristic (Lin & Kernighan, 1973). This path is then partitioned into feasible TSPHS trips using an auxiliary graph. Next, several local search operators are applied to improve the constructed tour (Castro, Sörensen, Vansteenwegen & Goos, 2015). The most recent method to tackle the TSPHS is a hybrid dynamic programming and memetic algorithm (Lu, Benlic & Wu, 2018). It generates an initial population, including feasible and infeasible TSPHS solutions, using two construction methods. The first method transforms a TSP path, constructed by the Lin-Kernighan heuristic, into a TSPHS tour by means of a dynamic programming approach. In the second method, random sequences of hotels are used to build TSPHS tours. After creating the initial population, the algorithm tries to find the best TSPHS tour employing crossover, mutation, and local search operators. It should be mentioned that the dynamic programming approach is also incorporated into the local search procedure of this algorithm.

The OPHS was introduced by Divsalar et al. (2013) getting inspiration from the TSPHS. Two methods have been proposed to solve the OPHS. The first one is based on the Skewed Variable Neighborhood Search (SVNS) metaheuristic and the second one is a Memetic Algorithm (MA). The initial step of the methods is similar to each other. They estimate the score that can be produced from each feasible sequence of hotels. For this purpose, a three-dimensional array called Matrix of Pairs of Hotels (MPH) is constructed by solving the OP between each feasible pair of hotels considering each trip time limitation (i.e., by solving sub-OPs). To tackle each OP, all nodes are involved and a simple heuristic is applied. The element $MPH\ [i]\ [j]\ [k]$ contains the constructed OP solution between the hotels H_i and H_j with respect to the time limit of trip k (T_k) (Divsalar et al., 2013).

The SVNS consists of two phases: initialization and improvement. In the initialization phase, all feasible sequences of hotels are determined, and the estimated score that can be obtained from each combination is calculated based on the OP solutions in the MPH. Then, sequences with estimated scores above a threshold, different local search operators, and the OP solutions in the MPH are employed to construct several tours. Finally, the best tour created in this way is considered as the initial solution, and the improvement phase is started (Divsalar et al., 2013).

The SVNS is completely dependent on the TNFS. This drawback was eliminated by the next proposed method, which is a memetic algorithm that uses the MPH to create the initial population of OPHS solutions. According to the MPH, the MA creates a list of possible end-hotels corresponding to each trip and each hotel as the start-hotel of the trip. Next, each population member is constructed as follows. Initially, a feasible sequence of hotels is formed by setting hotel H_0 as the start-hotel of the sequence. The next hotels are selected randomly among the possible end-hotels. Each possible end-hotel has a value based on the estimated scores in the MPH, and feasible end-hotels with higher values have a large probability of selection during this process. After a feasible combination of hotels is built, nodes are selected and inserted in the constructed sequence using several local search operators. The obtained population is improved by crossover and mutation procedures, and the algorithm is terminated after a specific number of iterations (Divsalar et al., 2014).

3. General description of GRASP

Greedy Randomized Adaptive Search Procedure (GRASP) is a multi-start constructive metaheuristic that includes a construction phase and a local search phase (Fleurent & Glover, 1999). The procedure is repeated for a specific number of times, and finally, the best-obtained solution is the output of the algorithm.

Construction phase: This phase starts with an empty solution and is iterated until no candidate element exists that can be added to the solution. At each iteration of this phase, an empty Candidate List (CL) is created initially. Then, taking into account constraints in the problem, elements that can be used to make the current partially constructed solution more complete are determined and included in the CL. Afterward, the Restricted Candidate List (RCL) is formed from some of the candidates in the CL. There are different approaches for creating the RCL. It is constructed using the following equation here:

$$RCL = \{e \in CL | benefit(e) \rangle \alpha \times benefit(best)\}$$

Each $e \in CL$ has a value that is represented by $benefit(e)$. $best$ is the member with the highest value in the CL, and $\alpha \in [0, 1]$ is a parameter that must be tuned according to the problem (Resende & Ribeiro, 2010). Thus, the elements of the CL whose values are higher than a threshold form the RCL. Next, a member is randomly selected from this list and its corresponding element is included in the partial solution.

Local search phase: While the exploration (i.e., probing the solution space) is covered in the construction phase, the focus of this step is on the exploitation (i.e., searching the space around a specific solution). In other words, the neighborhood of the solution constructed in the first step is explored in this phase to find a better solution (Feo & Resende, 1995).

Although the standard GRASP is simple and powerful, it has some drawbacks, which can be eliminated using some techniques. The Proximate Optimality Principle (POP) is one of these techniques that is employed to improve the construction phase. According to this principle, local search operators can be used not only at the end of the construction phase but also during it. If the partially built solution at each iteration of the construction phase is improved using some local search operators, a better result is usually obtained at the end of the first phase (Fleurent & Glover, 1999). Starting the local search part of GRASP with a high-quality solution increases the probability of producing a solution nearer to the optimum and usually enhances the convergence speed. Therefore, the POP typically raises the convergence speed of a GRASP method as well as the quality of the final solution. Fig. 1 shows the pseudocode of a GRASP that uses the POP in the construction phase.

4. Main algorithm description

Our proposed algorithm is based on a GRASP approach called GRASP with Segment Remove (GRASP-SR) that was suggested for the OP (Keshtkaran & Ziarati, 2016). We extend GRASP-SR, which is a solution method for a single trip problem, and embed it with a novel algorithm to tackle a routing problem including multiple trips. Our method, whose pseudocode is shown in Fig. 2, employs the POP (see Section 3) in the construction phase and uses an innovative dynamic programming approach to improve the sequence of hotels in a tour.

At each iteration, the construction phase is started from a feasible tour that contains empty trips. Initially, at most one node is included in each trip using two procedures called Limited_Bidirectional_Insertion and Limited_POP_Improvement. Next, the algorithm employs Common_Insertion and Bidirectional_Insertion to insert an unvisited node in the tour as much as possible. Since our algorithm follows the POP, the POP_Improvement procedure is applied to improve the modified tour after each insertion. When there is no node left that can be visited during the tour, the local search phase, which contains two operators (i.e., Common_RemoveReplace and Bidirectional_RemoveReplace), is started

```
function GRASP-POP
    for k = 0 to MAX_iterations
        Solution = {}
        //Construction phase
        while there is an element that can be included in Solution
            CL = {}
            Add candidate elements to the CL
            Create the RCL and select a candidate element randomly
            Add the selected element to Solution
            // POP in the construction phase
            Improve Solution by some operators
        //Local search phase
        Improve Solution by local search operators
        Update the global best solution if Solution is better than the current global best solution
```

Fig. 1. Pseudocode of a GRASP that uses the POP.

```
function main
    Construct an initial feasible solution, which includes empty trips, using a dynamic
    programming approach                                                    //section 4.1
    for iter = 1 to MAX_iterations
        GRASP-OPHS( )

function GRASP-OPHS( )
    //Construction phase (sections 4.2 and 4.4)
    while Limited_Bidirectional_Insertion( ) do Limited_POP_Improvement( );   //section 4.2.2
    while Common_Insertion( ) do POP_Improvement( );                          //section 4.2.1
    while Bidirectional_Insertion( ) do POP_Improvement( );                   //section 4.2.2
    //Local search phase (sections 4.3)
    while Common_RemoveReplace( ) do;
    while Bidirectional_RemoveReplace( ) do;
```

Fig. 2. Pseudocode of the main algorithm.

to explore the neighborhood of the incumbent solution. Our proposed algorithm uses a novel dynamic programming method to identify the best sequence of hotels, and even to create the initial feasible tour. By means of this dynamic programming approach, the combination of hotels in a tour can be improved during the construction and local search phases. In the remainder of this section, this novel method is introduced, and the tour construction techniques and local search operators are described.

4.1. Using dynamic programming for improving the sequence of hotels

As described before, the previously introduced algorithms have a preprocessing step in which they solve sub-OPs. They solve the OP between each pair of hotels considering each trip time limitation to obtain information that can be used to create proper combinations of hotels. Our proposed algorithm does not spend any time on determining likely good sequences of hotels by solving sub-OPs. Instead, it uses a novel dynamic programming method to create the initial solution with empty trips and to improve the combination of hotels in the solution at each step of the algorithm. Our dynamic programming approach is independent of the TNFS (i.e., it does not determine all feasible sequences of hotels). It improves the sequence of hotels in a solution given the visited nodes and their visiting order. Therefore, the dynamic programming approach only alters the origins and destinations of the trips in a tour.

We define the optimal sequence of hotels for a specific tour as the one that produces the minimum length given the order of the visited nodes. This sequence can be identified using a dynamic programming technique. Suppose $LS\ [i]\ [j]$ is the length of the shortest feasible sub-tour from trip 0 to trip i when H_j is the end of trip i. The optimal-value function is as follows:

$$LS[i][j] = \min\left[LS[i-1][k] + Length\left(H_k, H_j, i\right)\right]$$

$$subject\ to:$$

$$0 \le k \le h-1$$

$$Length\left(H_k, H_j, i\right) \le T_i$$

$$LS[i-1][k] < \infty$$

where T_i represents the time limit of trip i, and $Length(H_s, H_e, i)$ represents the travel time of trip i in the solution if its origin and destination are replaced with H_s and H_e, respectively. The given constraints in this model signify that H_k is involved in the above-described minimization if it has two main characteristics: first, it produces a feasible trip shorter than T_i given the visited nodes in trip i and H_j as the end-point of the trip, and second, it must be possible to select an origin for trip $i-1$ such that H_k is its destination. This recursive function is initiated from the following stage-zero problem:

$$LS[0][j] = Length\left(H_0, H_j, 0\right)$$

$$subject\ to:$$

$$Length\left(H_0, H_j, 0\right) \le T_0$$

Therefore, $LS[D-1][1]$ is the minimum length that can be obtained while respecting the order of the visited nodes along the tour. The sequence of hotels that produces this length can be created using a two dimensional array called *BSH*, which keeps track of the optimal sequence.

BSH $[i]$ $[j]$ denotes the best start-hotel for trip i such that hotel H_j is its destination (i.e., the one that produces $LS[i][j]$). The initial value of *BSH* $[i]$ $[j]$ is -1, and *LS* $[i]$ $[j]$ is initialized to a large value. If $BSH[i][j]$ does not equal -1 after the *BSH* has been built, there is a feasible subsequence of hotels from trip 0 to trip i when H_j is the end-hotel of trip i. In other words, a feasible sub-tour with the minimum length can be constructed by changing the subsequence of hotels from trip 0 to trip i in the solution according to the *BSH*.

The optimal sequence of hotels for the tour is created by starting a chain process from the final trip. *BSH* $[D-1]$ $[1]$ is the best start-hotel for the last trip, and the origin of trip $D-1$ is the destination of trip $D-2$. Thus, the best start-hotel for trip $D-2$ is *BSH* $[D-2]$ [*BSH* $[D-1]$ $[1]$], and so on. We describe this procedure using an example.

Suppose that the number of hotels is four, and the goal is to find the best tour including five trips. The current solution is as follows:

Trip 0 : $H_0, 1, 81, 96\ H_0$

Trip 1 : $H_0, 55, 87, 49, 98, 17, 65, 89,\ H_1$

Trip 2 : $H_1, 13, 15, 20, 8, 99, 4, 44,\ H_3$

Trip 3 : $H_3, 18, 33, 26, 34, 41, 47, 52, 46, 40, 32, 24,\ H_1$

Trip 4 : $H_1, 88, 43, 39,\ H_1$

The best hotel sequence for the tour can be obtained using the *BSH*. At first, the *BSH* is constructed according to the dynamic programming model. This matrix is as follows:

End-hotel number		0	1	2	3
Trip number	**0**	0	***0***	0	0
	1	***1***	2	3	2
	2	2	2	***0***	3
	3	1	0	2	***2***
	4	−1	***3***	−1	−1

BSH $[i][j] = -1$ signifies that trip i cannot be started from any of the available hotels when its destination is hotel H_j. Starting from the last trip, *BSH* determines the best sequence of hotels. The best origin for the last trip is hotel H_3 (i.e., $BSH[4][1]$), and hotel H_2 is the best start-hotel for trip 3 when its destination is H_3 (i.e., $BSH[3][3]$). In the above table, the bold italic cells are the hotels that belong to the best sequence. Therefore, the combination of the hotels in the tour under study is modified as follows:

Trip 0 : $\mathbf{H_0}, 1, 81, 96\ \mathbf{H_1}$

Trip 1 : $\mathbf{H_1}, 55, 87, 49, 98, 17, 65, 89,\ \mathbf{H_0}$

Trip 2 : $\mathbf{H_0}, 13, 15, 20, 8, 99, 4, 44,\ \mathbf{H_2}$

Trip 3 : $\mathbf{H_2}, 18, 33, 26, 34, 41, 47, 52, 46, 40, 32, 24,\ \mathbf{H_3}$

Trip 4 : $\mathbf{H_3}, 88, 43, 39,\ H_1$

Similarly, let *BEH* $[i]$ $[j]$ denotes the best end-hotel for trip i when it starts from hotel H_j. This hotel produces the minimum travel time from trip i to trip $D-1$, which we denote by *LE* $[i]$ $[j]$. A recursive formula is applied to build these two matrices:

$$LE[i][j] = \begin{cases} Length\left(H_j, H_l, i\right) & if\ i = D-1\ and\ Length\left(H_j, H_l, i\right) \le T_{D-1} \\ \min_{0 \le k \le h-1\ and\ Length\left(H_j, H_k, i\right) \le T_i\ and\ BEH[i+1][k]!=-1} \left(Length\left(H_j, H_k, i\right) + LE[i+1][k]\right) & if\ 0 \le i < D-1 \end{cases}$$

If *BEH* $[i]$ $[j]$ does not equal -1 in the constructed matrix, there exists a feasible subsequence of hotels from trip i to trip $D-1$ when trip i is started from H_j. It should be noted that the best sequence of hotels for the solution can also be built by starting a chain process from *BEH* [0] [0]. The length of the modified tour obtained in this way is *LE* [0] [0].

To improve the sequence of hotels, the introduced matrices are employed in the construction phase during the insertion of nodes, and in the local search phase. Although the best sequence of hotels for an OPHS solution can be obtained either from the *BSH* or *BEH* in the local search phase, we use both of these two-dimensional arrays in the construction phase to reduce the required time for insertions. This issue is elucidated after the main algorithm is explained in more detail.

4.2. Construction phase: Common_Insertion and Bidirectional_Insertion procedure

Our algorithm constructs a feasible tour using two procedures called Common_Insertion and Bidirectional_Insertion. We borrow the basic idea of these procedures from the construction phase in GRASP-SR (Keshtkaran & Ziarati, 2016) and use the *BSH* and *BEH* in the Bidirectional_Insertion procedure to have more beneficial insertions in a tour.

Our investigations show that the bidirectional method is more suitable to build an OPHS solution in the construction phase; however, it is computationally more expensive than Common_Insertion. The execution time of our algorithm is reduced following this process: First, at most one node is included in each empty trip of the initial tour using Bidirectional_Insertion. Then, unvisited nodes are inserted in the tour as far as possible by the Common_Insertion procedure. When there are no more nodes that can be inserted in the tour using this procedure, Bidirectional_Insertion is employed to insert nodes that can be included in the solution if the sequence of hotels is modified during the insertion process. Common_Insertion and Bidirectional_Insertion are elaborated thoroughly in the next two subsections.

4.2.1. Common_Insertion procedure

One unvisited node is added to a trip at each call of this procedure. For this purpose, an empty Candidate List (CL) is created initially. Afterward, the best insertion place for each unvisited node k, in each trip m, is determined. This place is the one that produces the minimum increase in the trip travel time. If the insertion of the node in this place violates the time limit of trip m, suitable segments, by removing which the trip becomes feasible, must be found. A segment is a contiguous subset of visited nodes in a trip, and the score of the segment is the sum of the scores of the nodes visited in it. A segment in trip m is suitable if it has less score than node k, or has the same score as node k but its exclusion produces a shorter tour than the current one. Among the suitable segments that start from the same position of a trip, only the one with the minimum number of nodes is considered (Keshtkaran and Ziarati, 2016).

After suitable segments have been found, each of these is considered for updating the CL. Each member of the CL is a 5-tuple which has the following attributes: node number, trip number, index of the best insertion position, start- and end-index of the segment that has to be removed. If trip m remains feasible after inserting node k in its best position, there is no need to determine suitable segments and a member corresponding to the node and the trip is included in the CL by setting the start and

```
bool Common_Insertion
    CL = {}
    for m = 0 to D − 1
        for k = 0 to n − 1
            if k has not been visited in the tour
                pos = arg min_i len(m, k, i)
                if len(m, k, pos) ≤ T_m
                    Add (k, m, pos) to the CL
                else
                    for each suitable segment (i, j)
                        Add (k, m, pos, i, j) to the CL
    if CL is empty return false
    else
        Create the RCL and select a candidate from it
        Modify the tour with respect to the selected candidate
        return true
```

Fig. 3. Pseudocode of the Common_Insertion procedure. $len(m, k, i)$ is the length of trip m if node k is inserted in index i.

end-index of the segment equal to -1. This process is illustrated in the following example.

Suppose that the best insertion place for node 19 in trip 3 has index 2 and that the trip is as follows after the insertion:

Trip 3 : H_2, 18, **19**, 33, 26, 34, 41, 47, 52, 46, 40, 32, 24, H_3,

Assume that this insertion makes the trip infeasible because its length exceeds the time limit, T_3. Hence, suitable segments whose eliminations make the trip feasible must be determined:

Segment [**8**, **8**] : H_2, 18, **19**, 33, 26, 34, 41, 47, 46, 40, 32, 24, H_3
Segment [**9**, **9**] : H_2, 18, **19**, 33, 26, 34, 41, 47, 52, 40, 32, 24, H_3
Segment [**11**, **12**] : H_2, 18, **19**, 33, 26, 34, 41, 47, 52, 46, 40, H_3

Therefore, three following members are included in the CL:

$(19, 3, 2, 8, 8)$, $(19, 3, 2, 9, 9)$, *and* $(19, 3, 2, 11, 12)$

Each candidate member in the CL has a value that is equal to the difference between the score of its corresponding node and the score that is lost due to the removal of the determined segment. For example, the value of CL member $(19, 3, 2, 8, 8)$ is the difference between the score of node 19 and the score of segment [8,8].

After the CL has been constructed, the Restricted Candidate List (RCL) is created by selecting the best members of the CL. Next, one member is randomly selected from the RCL. The chosen node is added to the trip, and any required segment removal is carried out. After applying the POP_Improvement procedure to improve the currently constructed solution, the next iteration of Common_Insertion is started to include another unvisited node in the tour. This procedure is iterated until it is possible to increase the obtained score. Fig. 3 shows the pseudocode of the Common_Insertion procedure, in which D and n represent the number of trips and the number of nodes, respectively.

4.2.2. Bidirectional_Insertion procedure

Two challenges posed by the Common_Insertion procedure motivate us to define another construction method called Bidirectional_Insertion:

1. Common_Insertion does not modify the combination of hotels in a solution and constructs the CL while respecting the sequence of hotels in the currently constructed tour. However, without changing the start- and end-hotels of the trips, there is no chance for some unvisited nodes to become a node of the tour.
2. Starting the construction phase with Common_Insertion may cause the algorithm to be trapped in a local maximum. As described earlier, the construction phase starts from a tour that consists of empty trips. If the first phase is started by the Common_Insertion procedure, nodes that can be added to the tour are restricted by the current hotels in the solution. In other words, not all nodes have a similar chance to be selected as the first member of a trip. However, our experiments show that the first node inserted in each trip has a significant effect on the quality of a solution since it determines the direction of the movement on that trip.

Bidirectional_Insertion is similar to Common_Insertion, except that in the former, the sequence of hotels may be changed during the insertion of a node. The Bidirectional_Insertion procedure is applied, at the beginning of the algorithm, to insert at most one node in each trip of the initial tour in order to resolve the second challenge described (Limited_Bidirectional_Insertion in Fig. 2). This process is also used when Common_Insertion is not capable of adding another node to the tour.

One unvisited node is inserted in a trip at each iteration of Bidirectional_Insertion. Initially, *BSH*, *LS*, *BEH*, and *LE* are created according to the visitation order of nodes in the current tour and an empty Candidate List (CL) is built. Afterwards, the best feasible pair of hotels is determined for each trip, m, and for each unvisited node, k. For this purpose, feasible pairs of hotels for trip m are found using the dynamically constructed matrices. We define (x, y) as a feasible pair of hotels for a trip, i, if the pre- and post-trips remain feasible according to the *BSH* and *BEH* matrices after the current initial and final hotels of trip i are replaced by H_x and H_y, respectively (i.e., it must be checked that $BSH[i-1][x]$ and $BEH[i+1][y]$ differ from -1). After all feasible pairs of hotels for trip m have been found, the best insertion place for node k in trip m (i.e., the one that produces the minimum increase in the trip travel time) is determined considering each feasible pair of hotels as the start- and end-hotel of trip m. The best pair of hotels is the one that produces the shortest trip after the insertion of node k. The process of finding this pair is facilitated by constructing the *BSH*, *LS*, *BEH*, and *LE* at the beginning of each iteration since all information required to check the feasibility of each pair is available in these matrices.

After determining the best feasible pair of hotels for node k and trip m, the feasibility of the trip is checked. For this purpose, it is assumed that the determined hotels in the best pair are the origin and destination of trip m, and node k is in its best insertion place in the trip. If the length of the constructed trip in this way exceeds T_m, suitable segments (discussed in Section 4.2.1), whose eliminations make the trip feasible, must be found and a member corresponding to each of them must be included in the CL. Otherwise, it is not necessary to find suitable segments, and only one candidate must be added to the CL corresponding to node k and trip m. Each member in the CL is a 7-tuple that consists of the following attributes: the node number, trip number, index of the best position, best start-hotel, best end-hotel, start-index of the segment, and end-index of the segment.

To illustrate this, suppose that we want to find the best pair of hotels for node 28 and trip 2 in a feasible OPHS solution, which is as follows:

Trip 2 : H_0, 13, 15, 20, 8, 99, 4, 44, H_2 (Time limit of the trip = 20)

Assume that (H_0, H_2) and (H_1, H_3) are feasible pairs of hotels for this trip and that the best insertion place for node 28 for each of the two pairs is as follows:

$\mathbf{H_0}$, 13, 15, 20, 8, 99, **28**, 4, 44, $\mathbf{H_2}$ (Length = 28.9)

$\mathbf{H_1}$, 13, **28**, 15, 20, 8, 99, 4, 44, $\mathbf{H_3}$ (Length = 25)

The best feasible pair of hotels for trip 2 and node 28 is (H_1, H_3) since this pair produces the shortest path after inserting the node. However, the resulting trip violates the time limit of trip

```
bool Bidirectional_Insertion
    CL = {}
    for m = 0 to D − 1
        for k = 0 to n − 1
            if k has not been visited during the tour
                Find the best feasible pair of hotels:
                    BH_s, BH_e, Bpos = arg min_{H_s, H_e, i} len(H_s, H_e, m, k, i)

                if len(BH_s, BH_e, m, k, Bpos) ≤ T_m
                    Add (k, m, Bpos, BH_s, BH_e) to the CL
                else
                    for each suitable segment (i,j)
                        Add (k, m, Bpos, BH_s, BH_e, i, j) to the CL
    if CL is empty return false
    else
        Create the RCL and select a candidate from it
        Modify the tour with respect to the selected candidate
```

Fig. 4. Pseudocode of the Bidirectional_Insertion procedure. $len(H_s, H_e\, m, k, i)$ is the length of trip m if node k is inserted in index i and the origin and destination of the trip are replaced with H_s and H_e, respectively.

2. Therefore, it is necessary to find suitable segments. Given that [3,6] is the only suitable segment, we eventually obtain:

Segment $[3, 6]$: $\mathbf{H_1}, 13, \mathbf{28}, 4, 44, \mathbf{H_3}$.

Finally, the following 7-tuple is included in the CL: (28, 2, 2, H_1, H_3, 3, 6).

After the CL has been created, the Restricted Candidate List (RCL) is constructed and one member is randomly selected from it. The chosen node is included in the trip, and the appropriate alterations (i.e., the modification of hotels and segment removal) are made to the tour. For example, if the above 7-tuple is a member of the RCL and is selected, the following changes occur:

1. Node 28 is inserted in trip 2.
2. Segment [3,6] is eliminated from trip 2.
3. The current start- and end-hotel of trip 2 are replaced with H_1 and H_3, respectively.
4. Initial and destination hotels of the post-trips are modified concerning the best subsequence of hotels from trip 3 to the final trip. This combination is obtained from *BEH* considering *BEH* [3] [3] as the end-point of trip 3.
5. The subsequence of hotels before trip 2 is changed according to the *BSH*. The best start-hotel for trip 1 when its destination is hotel H_1 is available in *BSH* [1] [1].

Therefore, Bidirectional_Insertion alters start- and end-hotels of the pre-trips using the *BSH* while the *BEH* is employed to improve the subsequence of hotels in the post-trips.

The solution created in this way is improved as much as possible using the POP_Improvement procedure. Then, the next iteration of Bidirectional_Insertion is started by recalculating the *BSH*, *BEH*, *LS*, and *LE* given the new order of visited nodes along the tour. Fig. 4 shows the pseudocode of Bidirectional_Insertion.

4.3. Local search phase: Common_Removereplace and Bidirectional_Removereplace procedure

Based on the suggested local search phase by Keshtkaran and Ziarati (2016), the second phase of our proposed method applies two operators to improve the solution created in the construction stage. These two operators are based on Common_Insertion and Bidirectional_Insertion. We examine how the score and length of the constructed tour are changed if a visited node is omitted and unvisited nodes are inserted as much as possible.

For each node visited along the tour, the Common_RemoveReplace procedure checks which unvisited nodes can be included in the tour using Common_Insertion if the node is discarded. As in the construction phase, the POP_Improvement procedure is applied after each insertion to improve the created tour. The re-insertion of the eliminated node in the trip, from which it was excluded, is prevented in this process. Considering the constructed solutions with the highest score in this way, the best tour is the one that has the shortest length. This tour is the new incumbent solution if it either has a higher score than the current incumbent solution or the same score and a shorter length. This procedure is repeated until no more improvement is possible. Next, Bidirectional_RemoveReplace is started. Bidirectional_RemoveReplace is similar to Common_RemoveReplace, except that it employs Bidirectional_Insertion instead of the Common_Insertion procedure.

4.4. POP_Improvement procedure

As described before, we use the POP to improve the search performance of the construction phase in our proposed algorithm. Every time an insertion procedure includes an unvisited node in the tour, POP_Improvement is applied to improve the currently constructed solution. As described in Section 4.3, POP_Improvement is also used in the Bidirectional_RemoveReplace and Common_RemoveReplace procedures. The POP_Improvement procedure is repeated until no more improvement is possible by any of its containing operators.

The overall description of POP_Improvement is shown in Fig. 5. Each function, except ImproveHotels-DP, involves an iterative procedure, which is repeated until the corresponding operator can no longer improve the tour. In the following, these operators are described in details.

1. *Two-Opt:* This operator uses the best improvement strategy and is applied to reduce the length of each trip, as much as possible, by eliminating cross edges (Croes, 1958).
2. *Exchange:* The strategy of this operation is the best improvement. At each iteration, the two nodes located in two different

```
function POP_Improvement
    while an improvement is possible
        Two-Opt( )
        Exchange( )
        if the obtained solution is better than the incumbent one continue
        ImproveHotels-TSPHS( )
        if the obtained solution is better than the incumbent one continue
        ImproveHotels-DP( )
        if the obtained solution is better than the incumbent one continue
```

Fig. 5. Pseudocode of the POP_Improvement procedure. The *continue* statement corresponds to the *while* statement.

trips whose exchange maximizes the reduction in the tour length are swapped.

3. *ImproveHotels-TSPHS:* This operation, which is suggested by Vansteenwegen et al. (2012), improves the sequence of hotels and the visitation order of the nodes in the constructed tour. It starts from the first two consecutive trips. These two trips are reconstructed in order to reduce the length of the tour. It should be mentioned that, in this way, only the hotel between the two trips and the order of the nodes visited by them may be changed. Initially, each feasible hotel that is at least the nearest hotel to one of the visited nodes in the two trips is added to a list called *PIH* (Possible Intermediate Hotels). Each hotel in the *PIH* is assumed as the middle hotel, and the two trips are reconstructed by placing the nodes in their best positions. If the built trips include all of the nodes, the total length of the trips is reduced using Two-Opt; otherwise, they must be ignored. At the end of this process, the two considered trips are replaced with the best pair of trips obtained in this way (i.e., the one that maximizes the decrease in the tour length). Afterwards, given the next two consecutive trips, ImproveHotels-TSPHS is continued.
4. *ImproveHotels-DP:* This operation is employed to improve the sequence of hotels. Using the *BSH* matrix, the sequence of hotels is changed to the best one while respecting the current order of the visited nodes in the constructed tour. In this way, a chain process is started considering $BSH\ [D-1]\ [1]$ as the initial hotel of the last trip. Since this hotel is the destination of trip $D-2$, the best initial hotel for this trip can also be obtained from the matrix, and so on.

It should be mentioned that Limited_POP_Improvement in Fig. 2 is a limited version of the POP_Improvement procedure. It only includes Exchange and ImproveHotels-DP since the goal of the first step in our proposed method is to insert at most one unvisited node in each trip of the initial empty tour.

5. Algorithm complexity analysis

Assume that n is the total number of nodes, l is the number of visited nodes in a tour, H is the number of hotels, and D is the number of trips. The complexity of each procedure discussed in this article is as follows:

- Dynamic programming for improving the sequence of hotels: $O(DH^2)$
- Common_Insertion: $O(l(n-l))$
- Bidirectional_Insertion: $O(DH^2 + H^2 l(n-l))$
- Common_RemoveReplace: $O(l^2(n-l))$
- Bidirectional_RemoveReplace: $O(DH^2 l + H^2 l^2(n-l))$
- POP_Improvement: $O(l^2)$

The value l has the largest effect on the complexity of each iteration in our GRASP algorithm, and D and H in the above time complexity notations can be considered as the fix coefficients due to their small values. According to the above explanations, the total complexity of each iteration of GRASP in the worst case is proportional to $O(n^3)$. The complexity of the novel idea for improving a sequence of hotels is negligible in comparison with that of other local search operators (i.e., DH^2 vs. n^2).

Our experiments show that the execution time of GRASP increases with the growth of the number of visited nodes. Although this observation is true for almost all OPHS instances, it is violated by some others. For these instances, there is no direct control on the number of times the Insertion and RemoveReplace procedures are iterated, which is influenced by the quality of decisions made during the execution of the algorithm.

6. Experimental results

Our proposed GRASP algorithm has been implemented in C++, and all the computations have been done using an Intel Core i7 with 3.5 gigahertz CPU. Divsalar et al. (2013, 2014) introduced 405 standard benchmark instances for the OPHS. Based on the number of available hotels (excluding hotels H_0 and H_1) and the number of trips in each instance, 395 instances are divided into 16 sets. The name of each set is of the form "SET x–y", where x is the number of hotels (excluding hotels H_0 and H_1) and y is the number of trips. The other ten instances with five available hotels and two or three trips are located in SET 4. It should be mentioned that the optimal scores are known for all OPHS instances except for five instances in SET 4.

Our algorithm has two parameters, the values of which influence the quality of the solutions significantly: α which controls the size of the restricted candidate list in the construction phase, and the maximum number of iterations. As the basic idea of our algorithm in this article has been borrowed from Campos, Martí, Sánchez-Oro and Duarte (2014) and Keshtkaran and Ziarati (2016), following the advice from these authors, we set α and the maximum number of iterations to 0.2 and 500, respectively. However, regarding the difference between the computation time of GRASP with 500 iterations and that of the state-of-the-art algorithm, the results produced by GRASP with less number of iterations are also investigated.

There are only two algorithms in the literature to tackle the OPHS: the Skewed Variable Neighborhood Search (SVNS) (Divsalar et al., 2013) and the Memetic Algorithm (MA) (Divsalar et al., 2014). As explained in Sections 1 and 2, the SVNS is dependent on the Total Number of Feasible Sequences of hotels (TNFS), the maximum of which is equal to $h^{(D-1)}$ for an instance with D trips and h hotels. It is clear that the SVNS cannot provide a feasible solution within a reasonable time when the TNFS becomes large. Although the SVNS outperforms the MA in some of the small instances, the MA is considered as the state-of-the-art algorithm for the OPHS since it obtains better solutions in shorter execution times when the instances become large (Divsalar et al., 2014).

In the next sections, our GRASP algorithm is compared with these two approaches in terms of the quality of the produced solutions and execution time. The details of the solutions constructed by GRASP with 500 iterations are available in Tables A.1 to A.17 in the appendix. The results presented for the SVNS in the following subsections and in the appendix are the ones reported by Divsalar et al. (2014). These results have been obtained on an Intel Core 2 duo at 3 gigahertz CPU. Thanks to Prof. Divsalar, the values reported for the MA are the results obtained running the original MA code on our system.

6.1. The comparison of the SVNS and GRASP

The solutions produced by the SVNS within 360 seconds are compared with the best solutions over three runs of GRASP in Table 1. Although within this time limit, GRASP can find solutions for all 400 instances with known optimal results, our algorithm is

Table 1
The comparison between the best scores over three runs of GRASP and the solutions provided by the SVNS.

Instances				Optimal solutions			SVNS vs. GRASP-10 iter		SVNS vs. GRASP-500 iter		Avg. total CPU times (seconds)		
	#Ins	#VIns	MAX TNFS	SVNS	GRASP		SVNS	GRASP	SVNS	GRASP	SVNS	GRASP	
					10 iter	500 iter						10 iter	500 iter
SET 1–2	35	35	3	18	**25**	**28**	2	**12**	0	**13**	0.12	0.19	7.24
SET 2–3	35	35	16	19	**23**	**26**	2	**9**	0	**10**	0.14	0.23	9.60
SET 4 (OPT)	5	5	25	**4**	**4**	**4**	**0**	**0**	**0**	**0**	0.24	0.07	3.77
SET 5–3	35	35	49	19	**26**	**26**	0	**11**	0	**11**	0.21	0.29	12.51
SET 3–4	35	35	125	18	**22**	**26**	**7**	**7**	0	**10**	0.41	0.28	11.83
SET 6–4	35	35	512	17	**21**	**26**	5	**9**	0	**13**	0.88	0.32	15.18
SET 10–4	22	22	1728	3	3	**11**	7	**10**	0	**15**	5.12	1.97	94.81
SET 12–4	22	22	2744	2	3	**10**	6	**14**	0	**18**	5.01	2.12	104.84
SET 15–4	22	22	4913	2	**6**	**12**	3	**13**	0	**18**	4.95	2.37	118.81
SET 10–5	22	22	20,736	4	3	**10**	4	**14**	0	**17**	4.26	2.10	99.36
SET 12–5	22	22	38,416	4	1	**9**	7	**12**	0	**17**	4.21	2.23	107.55
SET 15–5	22	22	83,521	4	4	**10**	5	**13**	0	**17**	4.2	2.58	121.69
SET 10–6	22	22	248,832	4	4	**10**	3	**16**	0	**18**	3.83	2.13	102.84
SET 12–6	22	22	537,824	3	5	**11**	3	**16**	0	**19**	3.84	2.38	112.53
SET 15–6	22	22	1,419,857	4	4	**10**	3	**17**	0	**18**	4.29	2.78	130.87
SET 15–8	13	7	410,338,673	0	0	**2**	1	**6**	0	**7**	53.63	3.09	154.50
SET 15–10	9	0	1.18 × 10 ^11	–	–	–	–	–	–	–	–	–	–
All instances	400	385	1.18 × 10 ^11	125	**154**	**231**	58	**179**	0	**221**	3.41	1.36	64.74

compared with the SVNS regarding only the instances for which the SVNS could find a solution.

In Table 1, the first column shows the information about each set of the instances in four sub-columns. The first sub-column shows the name of each set. In this column, the "(OPT)" label in front of "SET 4" signifies that only the five instances with known optimal values in SET 4 are considered for this comparison. Sub-column "#Ins" specifies the number of instances available in each set. The number of instances in each set for which the SVNS can find a solution within 360 seconds is shown in sub-column "#VIns". Sub-column "MAX TNFS" specifies the maximum TNFS value for each set. Column "Optimal solutions" shows for how many of the instances in each set the optimal scores are obtained by the algorithms. The next two columns relate to the comparison of the provided solutions by the SVNS and GRASP. Each of these columns has two sub-columns. The number of instances for which the SVNS finds better solutions is available in sub-column "SVNS", and sub-column "GRASP" presents the number of instances for which the tours constructed by GRASP have higher scores. The average total execution time of each algorithm in each set is available in the last column. Since all instances in SET 15–10 and the 6 instances of SET 15–8 are not solvable by the SVNS within 360 seconds, these instances are not considered for the comparisons of this section.

Concerning the execution times of the MA code on our system, it can be concluded that the execution times of the SVNS will be reduced to the half of the values reported in Table 1 if this algorithm is run on our system. Therefore, as it can be seen in Table 1, the execution time of GRASP with 10 iterations is almost equal to that of the SVNS. This version of GRASP outperforms the SVNS in 179 instances while there are only 58 instances for which the SVNS produces better results than GRASP with 10 iterations.

GRASP with 500 iterations outperforms the SVNS in 221 instances while there is not any instance for which the SVNS can find a better solution than GRASP. The computational efforts of the GRASP algorithm with 500 iterations are larger than that of the SVNS for small instances; however, the superiority of GRASP over the SVNS in terms of execution time becomes clear when the TNFS gets larger. Since the SVNS spends a long time evaluating all feasible sequences of hotels, it cannot produce a feasible solution, even within 6 minutes, for instances with a large TNFS. On the other hand, GRASP is independent of the TNFS and can find acceptable solutions for all instances in SET 15–8 and SET 15–10 within this time limit. Thus, the computation time of the SVNS is influenced by the TNFS; but, as discussed in Section 5, the computation time of GRASP is affected by the number of visited nodes along a tour, and the number of times the RemoveReplace and Insertion procedures are applied.

We have also compared our algorithm with the SVNS in terms of the mean of the gaps in each set. The gap for each instance is defined as follows:

$$gap = \left(\frac{optimal\ value - best_{\ SVNS\ or\ GRASP}}{optimal\ value} \right) * 100$$

The results of this comparison are shown in Fig. 6. In this figure, the superiority of GRASP over the SVNS in terms of the quality of the solutions, especially in the instances with large TNFS values, is clear. Moreover, the means of the gaps over the 385 instances are 2.99%, 2.30%, and 0.42% for the SVNS, GRASP with 10 iterations, and GRASP with 500 iterations, respectively.

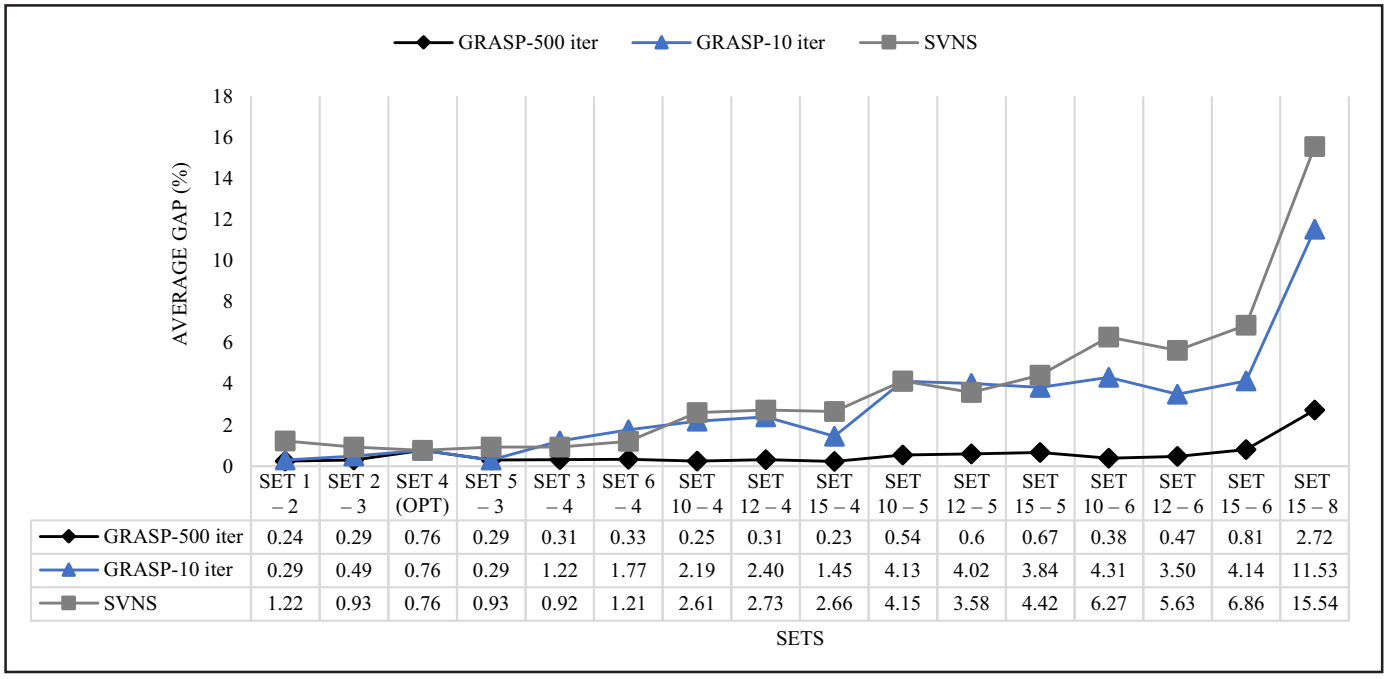
6.2. The comparison of the MA and GRASP

As described before, the MA is the state-of-the-art algorithm for the OPHS. In this section, our proposed method is compared with the MA in terms of the following criteria:

6.2.1. The best-obtained results

In this part, the best-results obtained over three runs of the MA and GRASP are compared. When the number of iterations of GRASP is considered as 35, the average total computation time of GRASP is almost as the same as that of the MA. The MA provides better solutions than GRASP with 35 iterations for 90 instances while GRASP finds better results for 103 instances. The optimal solutions are produced by GRASP for 187 instances while the number of instances for which the MA constructs the optimal tours is 174. It should be mentioned that, in this experiment, the exploration power of GRASP has been reduced.

As it can be seen in Table 2, by increasing the number of iterations, GRASP can find better results. This table has the same description as Table 1. GRASP with 500 iterations constructs the optimal tours for 231 instances. When looking at the quality of the obtained results, our method outperforms the MA in 142 instances while the MA outperforms GRASP only in 17 instances. There



	SET 1 – 2	SET 2 – 3	SET 4 (OPT)	SET 5 – 3	SET 3 – 4	SET 6 – 4	SET 10 – 4	SET 12 – 4	SET 15 – 4	SET 10 – 5	SET 12 – 5	SET 15 – 5	SET 10 – 6	SET 12 – 6	SET 15 – 6	SET 15 – 8
GRASP-500 iter	0.24	0.29	0.76	0.29	0.31	0.33	0.25	0.31	0.23	0.54	0.6	0.67	0.38	0.47	0.81	2.72
GRASP-10 iter	0.29	0.49	0.76	0.29	1.22	1.77	2.19	2.40	1.45	4.13	4.02	3.84	4.31	3.50	4.14	11.53
SVNS	1.22	0.93	0.76	0.93	0.92	1.21	2.61	2.73	2.66	4.15	3.58	4.42	6.27	5.63	6.86	15.54

Fig. 6. The mean of the gaps for GRASP and the SVNS.

are ten sets for which the MA cannot outperform our proposed method in any of the instances in them.

The comparison between GRASP with 500 iterations and the MA indicates that the former method is more expensive in terms of computational time. GRASP with 500 iterations constructs a feasible high-quality solution for an instance within 0.5 minutes on average. Taking into account the quality of the solutions and the existing applications of the OPHS, we can conclude that the execution time of GRASP is acceptable.

6.2.2. Increasing the execution time of the MA

As described before, GRASP is diluted by decreasing the number of iterations. Hence, it is more fair to do another experiment by increasing the computation time of the MA. For this purpose, the solutions produced over three runs of GRASP with 500 iterations are compared with those of three modified versions of the MA. Modifying three parameters of the original MA, these three versions are created as follows:

1. *MA-MaxIteration:* Setting the maximum number of iterations to 1500.
2. *MA-PopSize:* Setting the population size to 500.
3. *MA-NTRI (the Number of Times Running on each Instance):* Setting the number of times that the algorithm is run on each instance to 50.

It should be mentioned that, in the original MA, the maximum number of iterations and the population size are 100 and 30, respectively. In addition, this algorithm has been applied on each instance three times by Divsalar et al. (2014). The results obtained in this experiment are presented in Table 2. As shown in this table, GRASP is competitive with these three augmented versions of the MA. These three versions of the MA cannot outperform GRASP in even one instance of the six sets whose names have been bolded.

6.2.3. The mean of the gaps in each set

We also compared our algorithm with the MA in terms of the mean of the gaps in each set. Fig. 7 demonstrates that GRASP with 500 iterations produces a lower mean gap in all sets, except in SET 15–10, which contains nine of the most difficult instances. Although GRASP with 500 iterations constructs better tours for two instances in this set, the reduction of the gap in these instances is smaller than its increase in the other seven. It should also be noted that GRASP-SR, which is our inspiration source, performs worse when the number of visited nodes along an OP path becomes greater than 50 (Keshtkaran & Ziarati, 2016). Therefore, it is logical that our algorithm has a poor performance in SET 15–10 since the tours provided by GRASP for the instances in this set have the largest number of visited nodes, which is about 74 on average. As it can be seen in Table 3, increasing the number of iterations, better results will be found by GRASP. It should be mentioned that the mean of the gaps over the 400 instances is equal to 1.24% for the MA.

In Table 4, with respect to the average gap in each set, GRASP with 500 iterations is compared with the MA and its three versions described before. In this table, the minimum average gap produced in each set has been bolded. In terms of the average gap, GRASP outperforms MA-PopSize and MA-MaxIteration in almost all sets. Concerning the average gap over 400 instances, MA-NTRI is stronger than MA-PopSize and MA-MaxIteration. The solutions of GRASP for the instances in nine sets are closer to the optimal results than those of the MA-NTRI; however, the average gap of GRASP is larger than that of the MA-NTRI in five sets. The results show that the number of trips affects the performance of GRASP. When the number of trips is not more than 6, GRASP almost outperforms the MA-NTRI. On the other hand, the difference between the average gap of GRASP with that of the MA-NTRI is increased dramatically when the number of trips is more than six.

6.2.4. Statistical tests: Z-test and Mann–Whitney test

According to the obtained gaps, some statistical tests have been performed. In Table 5, the detailed results of these tests, which are described as follows, are presented:

Test 1. The original MA vs. GRASP with 500 iterations: With respect to the solutions produced by the original MA, it has been determined if GRASP with 500 iterations can provide a significant

Table 2
The comparison of GRASP with 500 iterations and the MA.

	# Ins	Optimal solutions					MA vs. GRASP		MA-MaxIteration vs. GRASP		MA-PopSize vs. GRASP		MA-NTRI vs. GRASP		Avg. total CPU times (seconds)				
		MA	MA-MaxIteration	MA-PopSize	MA-NTRI	GRASP	MA	GRASP	MA	GRASP	MA	GRASP	MA	GRASP	MA	MA-MaxIteration	MA-PopSize	MA-NTRI	GRASP
SET 1–2	35	14	14	14	16	**28**	0	**21**	0	**21**	0	**19**	0	**17**	2.45	36.74	40.43	40.06	7.24
SET 2–3	35	23	23	23	**26**	**26**	0	**6**	0	**6**	0	**4**	**0**	**0**	2.02	29.34	32.06	32.73	9.60
SET 4 (OPT)	5	**4**	**4**	**4**	**4**	**4**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1.21	15.64	17.44	17.66	3.77
SET 5–3	35	24	25	25	**26**	**26**	0	**4**	0	**2**	0	**2**	**0**	**0**	2.04	29.46	32.75	33.12	12.51
SET 3–4	35	23	23	24	24	**26**	0	**6**	0	**5**	0	**3**	0	**2**	1.57	22.56	24.42	25.66	11.83
SET 6–4	35	23	**26**	25	25	**26**	0	**4**	**1**	**1**	0	**2**	**1**	**1**	1.68	24.11	24.71	27.14	15.18
SET 10–4	22	7	9	6	**11**	**11**	0	**10**	1	**7**	0	**8**	1	**4**	9.24	129.85	135.57	149.27	94.81
SET 12–4	22	6	9	9	**10**	**10**	1	**11**	1	**7**	1	**9**	3	**6**	8.91	128.40	136.20	146.71	104.84
SET 15–4	22	5	7	8	10	**12**	0	**14**	0	**9**	0	**8**	0	**6**	9.15	126.99	134.75	146.93	118.81
SET 10–5	22	6	9	7	**10**	**10**	1	**10**	1	**5**	1	**7**	1	**3**	7.82	106.22	108.29	122.87	99.36
SET 12–5	22	6	8	7	8	**9**	0	**9**	0	**4**	1	**6**	1	**5**	7.31	106.04	108.57	121.24	107.55
SET 15–5	22	7	9	9	8	**10**	1	**10**	2	**4**	2	**6**	2	**6**	7.66	105.43	107.72	122.83	121.69
SET 10–6	22	9	9	9	**10**	**10**	1	**10**	2	**5**	3	**5**	**2**	1	6.51	91.99	88.70	105.38	102.84
SET 12–6	22	6	9	9	**11**	**11**	0	**10**	0	**5**	0	**5**	**3**	2	6.45	91.79	91.19	105.14	112.53
SET 15–6	22	8	9	10	**11**	10	3	**9**	4	**7**	**5**	**5**	**5**	4	7.04	91.58	89.60	107.41	130.87
SET 15–8	13	1	3	3	**4**	2	3	**6**	3	**5**	3	**5**	**6**	1	7.57	101.79	94.65	114.80	208.88
SET 15–10	9	2	**4**	3	3	0	**7**	2	**8**	1	**8**	1	**9**	0	6.88	94.41	93.65	110.67	281.20
All instances	400	174	200	195	217	**231**	17	**142**	23	**94**	24	**95**	34	**58**	5.12	71.88	73.94	82.36	72.73

Table 3
The results obtained for the given number of iterations using GRASP over 400 instances.

	Optimal solutions	Avg. total CPU times (seconds)	Average gap (%)
25 iterations	179	3.89	1.70
35 iterations	187	5.27	1.55
50 iterations	192	7.50	1.37
500 iterations	231	72.73	0.60
1000 iterations	236	152.13	0.48

Table 4
The mean of gaps for GRASP with 500 iterations and for the four versions of the MA.

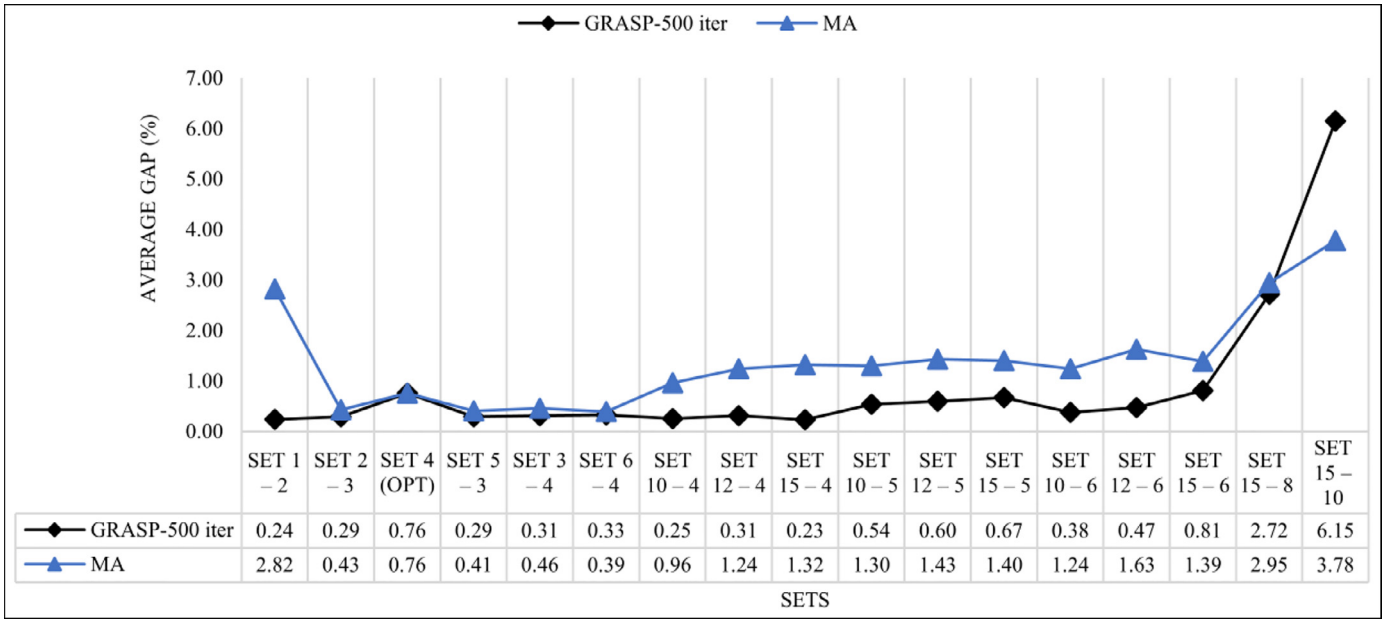
	GRASP	MA	MA-MaxIteration	MA-PopSize	MA-NTRI
SET 1–2	**0.24**	2.82	2.82	2.52	2.47
SET 2–3	**0.29**	0.43	0.43	0.51	**0.29**
SET 4 (OPT)	**0.76**	**0.76**	**0.76**	**0.76**	**0.76**
SET 5–3	**0.29**	0.41	0.34	0.41	**0.29**
SET 3–4	**0.31**	0.46	0.42	0.35	0.34
SET 6–4	0.33	0.39	**0.32**	0.36	**0.32**
SET 10–4	**0.25**	0.96	0.49	0.92	0.29
SET 12–4	**0.31**	1.24	0.62	1.18	0.42
SET 15–4	**0.23**	1.32	0.58	0.62	0.47
SET 10–5	**0.54**	1.3	0.89	0.92	0.58
SET 12–5	**0.60**	1.43	0.95	0.88	0.72
SET 15–5	**0.67**	1.4	0.93	0.94	0.86
SET 10–6	**0.38**	1.24	0.76	0.93	0.40
SET 12–6	0.47	1.63	0.96	0.98	**0.43**
SET 15–6	0.81	1.39	0.73	0.60	**0.46**
SET 15–8	2.72	2.95	2.17	1.97	**0.84**
SET 15 – 10	6.15	3.78	2.32	1.67	**0.90**
Over 400 ins.	**0.60**	1.24	0.89	0.91	0.64

improvement. We claim that GRASP with 500 iterations produces the optimal solutions for more instances than the MA.

Thus, the null hypothesis is that the proportion of the instances for which GRASP produces the optimal solution is less than that of the MA. The *Z*-test for proportions indicates that the number of instances for which GRASP finds the optimal tour is significantly larger than that of the MA (P-value < 0.05). The proportion of the instances for which GRASP can find the optimal result is at least 8.5% more than that of the MA.

GRASP with 500 iterations and the MA cannot produce the optimal solutions for 163 instances. Considering these instances, the mean of the gaps is reduced from 2.12% to 1.32% using GRASP. Therefore, the next hypothesis is that, in comparison with GRASP, the MA produces closer solutions to the optimal results for these 163 instances. Two groups of data are considered: the solutions produced by the MA for the 163 instances and the ones obtained using GRASP. According to the Anderson-Darling test, the distributions of these two groups of data are not normal. Thus, using the Mann–Whitney as a non-parametric test, the null hypothesis is disproved. The Mann–Whitney test shows that our suggested method produces solutions closer to the optimum (P-value < 0.05).

Test 2. The MA-NTRI vs. GRASP with 500 iterations: Regarding the solutions found by these two algorithms, two statistical tests have been performed. The descriptions of these two tests are similar to the ones explained in the previous comparison. When the solutions for all the instances are involved in these tests, the null hypothesizes are rejected at the confidence level of 80%. However, as shown in Table 5, when the results for the instances in SET 15–8 and SET 15–10 are not considered, the confidence level is increased to 90% and 95% for the *Z*-test and Mann–Whitney test, respectively. Thus, it can be concluded that GRASP produces better solutions than the MA for the instances with at most six trips. It should be noticed that the same conclusion can also be derived from Table 2, Fig. 7 and Table 4.



	SET 1 – 2	SET 2 – 3	SET 4 (OPT)	SET 5 – 3	SET 3 – 4	SET 6 – 4	SET 10 – 4	SET 12 – 4	SET 15 – 4	SET 10 – 5	SET 12 – 5	SET 15 – 5	SET 10 – 6	SET 12 – 6	SET 15 – 6	SET 15 – 8	SET 15 – 10
GRASP-500 iter	0.24	0.29	0.76	0.29	0.31	0.33	0.25	0.31	0.23	0.54	0.60	0.67	0.38	0.47	0.81	2.72	6.15
MA	2.82	0.43	0.76	0.41	0.46	0.39	0.96	1.24	1.32	1.30	1.43	1.40	1.24	1.63	1.39	2.95	3.78

Fig. 7. The mean of gaps for GRASP and for the MA.

6.2.5. The average-obtained results over three runs

The comparison between the average scores of the solutions obtained from three runs of GRASP with 500 iterations and the MA for each instance of the problem is summarized in Table 6. Each pair of numbers in each row of this table indicates how many instances have the property specified in that row considering the average solution scores. As an example, 26 in row 2 shows that there are 26 instances for which the average solution score obtained over three runs of the MA is better than that of our proposed GRASP method.

6.2.6. The solutions of the five instances with unknown optimal solutions in SET 4

The results provided by the MA and GRASP for the five instances with unknown optimal solutions are reported in Table 7. In this table, the first column contains the names of these five instances. For each instance, the next column presents the Best Feasible (BF) solution and Upper Bound (UB) value reported by Divsalar et al. (2014). The best scores and the average CPU times for each instance over three runs of the MA and GRASP are shown in the last two columns. GRASP with 10 and 500 iterations have been considered in this experiment. The results produced by the MA and its other three versions are also presented. As shown in Table 7, GRASP provides new best results for three out of these five instances even within 0.24 seconds on average (i.e., using GRASP with 10 iterations). The MA-PopSize and MA-NTRI also can find two of these new best results but on average within 30.77 and 32.63 seconds, respectively.

6.2.7. New instances for the OPHS

As explained by Divsalar et al. (2014), 405 available benchmark instances for the OPHS have been constructed using the standard OP instances. Considering these instances, the effect of increasing the number of hotels and trips on the performance of the algorithms can be studied. Using the TSPHS instances introduced by Vansteenwegen et al. (2012), we have produced new 76 instances. These new instances can be employed to investigate the performance of the algorithms when the number of nodes is increased.

There are four sets of the TSPHS instances called SET 1, SET 2, SET 3, and SET 4. The instances in SET 1 and SET 4 have 6 and 10 hotels, respectively, while the instances in SET 3 have 4, 6, 10, or 11 hotels. The instances in SET 2 have at most 40 customers. These instances have not been used in the construction of the OPHS instances since we have aimed to construct large instances. The TSPHS instances in SET 1 have between 48 and 288 customers while SET 3 and SET 4 contain more complex instances with 51 to 1002 nodes.

To derive an OPHS instance from an instance of the TSPHS, each customer of the TSPHS instance has been assigned a random score from the set $\{10x | x = 1, 2, \ldots, 10\}$. The time budget of the trips in the TSPHS instance is used as the time limit for all the trips in the OPHS instance. Since the service time of the customers in the TSPHS instances of SET 1 is greater than zero, for these instances, the half of the time budget of the TSPHS trips is considered as the time limit for the trips of the OPHS instance.

The TSPHS instances with at most 500 customers have been considered to construct five new sets of OPHS instances. In each set, the instances have the same number of trips. The sets are referred as "SET x–y–z" where x is the number of hotels excluding H_0 and H_1, y is the number of trips, and z is the name of the problem (VRP or TSP) whose instances have been used by Vansteenwegen et al. (2012) to construct the TSPHS instances.

Table 8 summarizes the results obtained using the new instances. All algorithms, except MA-NTRI, have been run three times on each instance. To have a fair comparison in terms of the execution time, in MA-NTRI, the number of times that the MA is run on each instance is set to 75. The first column of the table shows the name of the sets along with the number of instances available in them. The next three columns are related to the comparison of GRASP with the MA in terms of the solutions quality. In these columns, the number of instances for which one algorithm outperforms the other one is presented for each set. The average total CPU time of each algorithm is available in the last five columns. The detailed results of this experiment are available in Tables B.1 to B.5 in the appendix.

There are 12 instances in SET 4–4-VRP for which all the nodes are visited in the tours constructed by all versions of the MA and GRASP. Therefore, these solutions are the optimal results of these instances. In addition, GRASP with 500 iterations outperforms the MA-NTRI in all sets except in SET-9–7-TSP whose instances have more than six trips. Regarding the instances constructed by

Table 5
Statistical tests.

Z-test					
Test	α		Z	P-value	Lower bound for difference
1	0.05	$P_{GRASP} - P_{MA} = 0.143$	4.07	0.000	0.085 (95% lower bound)
2	0.1	$P_{GRASP} - P_{MA\text{-}NTRI} = 0.050$	1.40	0.080	0.004 (90% lower bound)

Normality test			
Test		Anderson–Darling	P-Value
1	GRASP-500 iter; MA	16.757; 9.453	< 0.005; < 0.005
2	GRASP-500 iter; MA-NTRI	9.383; 18.298	< 0.005; < 0.005

Mann–Whitney test			
Test	α	Wilcoxon	P-Value
1	0.05	21,377	0.0000
2	0.05	18874.5	0.0247

Table 6
The comparison between the average scores of GRASP and the MA solutions.

Both obtain solutions with equal score	150 (37.5%)
The solution obtained by the MA is better than that of GRASP	26 (6.5%)
The solution obtained by GRASP is better than that of the MA	224 (56%)
GRASP obtains the optimal solution	199 (49.75%)
The MA obtains the optimal solution	125 (31.25%)

Divsalar et al. (2014), this conclusion has also been drawn in the previous sub-sections.

6.3. Evaluating the effect of each part of GRASP on the solution quality

We also investigated the impact of each part of the algorithm on the final solutions of all 405 standard instances. For this purpose, three variants of the proposed GRASP algorithm with 500 iterations are studied concerning the obtained results over three runs:

1. *Simple GRASP:* This algorithm only uses the construction phase described in Section 4.2.
2. *Common GRASP:* This method employs Common_Insertion to build a tour in the construction phase and uses Common_RemoveReplace in the local search phase.
3. *Bidirectional GRASP:* This algorithm applies Bidirectional_Insertion and Bidirectional_RemoveReplace in the construction phase and local search phase, respectively.

It should be mentioned that all of these algorithms use the POP_Improvement procedure.

Considering the 5 instances with unknown optimal solutions, the results obtained by all the variants are the same as the reported results in the previous section. Regarding the other 400 instances with known optimal solutions, the best solutions provided by these methods and the main algorithm (i.e., the full presented algorithm in this article) are compared with those of the MA in Table 9. Sub-column "MA" in column "Competitive solutions" presents the number of instances for which the MA produces better solutions than the considered GRASP. The second sub-column in this section of the table shows in how many instances each GRASP version outperforms the MA. The number of instances for which the GRASP under study finds the optimal solutions is reported in the next column. The average execution time of each GRASP algorithm on all 405 instances is available in the last column.

The average computational time of Common GRASP is smaller than that of Bidirectional GRASP. However, Bidirectional GRASP produces solutions with higher quality in comparison with Common GRASP. Our main algorithm establishes an equilibrium between the execution time and the quality of the tours. The quality of the solutions provided by this method is almost similar to Bidirectional GRASP; however, the main algorithm is less expensive in terms of computational effort. The results obtained by Simple GRASP indicate that the construction phase used in this article, along with POP_Improvement, has an acceptable performance. There is a negligible difference between the average computational time of this GRASP method and that of the MA. Therefore, it can be concluded that the RemoveReplace procedures are the most time-consuming parts of the main algorithm.

7. Conclusion and future work

In this article, a novel algorithm based on GRASP was proposed to tackle the OPHS. This algorithm applies a new, fast, and powerful dynamic programming approach to improve the sequence of hotels in a tour. The experiments show that the suggested GRASP method produces competitive solutions in a reasonable time for all available benchmark instances. It provides the optimal solutions for 57.75% of the instances while the state-of-the-art algorithm does so only for 43.5% of the instances. GRASP out-

Table 7
The solutions of the five instances with unknown optimal solutions in SET 4.

Instances	Best known results		Best results						Avg. of the total CPU times (seconds)					
	BF	UB	MA	MA-MaxIteration	MA-PopSize	MA-NTRI	GRASP-10 iter	GRASP-500 iter	MA	MA-MaxIteration	MA-PopSize	MA-NTRI	GRASP-10 iter	GRASP-500 iter
100_20_3	357	376	368	368	368	368	368	368	1.98	27.47	29.14	31.08	**0.19**	9.05
100_25_3	495	568	524	524	**528**	**528**	**528**	**528**	3.77	50.20	50.28	57.08	**0.39**	19.36
102_35_3	230	380	324	324	324	324	324	324	1.55	20.66	22.98	23.28	**0.13**	5.70
102_40_3	299	493	386	386	386	386	**389**	**389**	1.74	23.67	23.98	23.69	**0.22**	9.02
102_45_3	356	579	444	444	**447**	**447**	**447**	**447**	1.75	25.22	27.47	28.05	**0.26**	11.78
Average	–	–	–	–	–	–	–	–	2.16	29.44	30.77	32.63	**0.24**	10.98

Table 8
The comparison of GRASP with the MA regarding the new OPHS instances.

Sets		Competitive solutions						Avg. total CPU times (seconds)				
		MA vs. GRASP-10 iter		MA vs. GRASP-20 iter		MA-NTRI vs. GRASP-500 iter						
	#Ins	MA	GRASP	MA	GRASP	MA	GRASP	MA	MA-NTRI	GRASP-10 iter	GRASP-20 iter	GRASP-500 iter
SET 4–4-VRP	16	0	**4**	0	**4**	1	**3**	113.29	2824.18	87.04	175.26	4396.05
SET 2–3-TSP	15	0	**15**	0	**15**	0	**10**	187.49	4868.83	69.94	137.24	3423.18
SET 4–4-TSP	15	4	**10**	3	**11**	1	**9**	150.75	3696.58	54.87	110.95	2810.90
SET 9–7-TSP	15	**11**	1	**11**	4	**6**	3	90.37	2226.00	45.89	90.72	2225.93
SET 8–4-TSP	15	**6**	**6**	5	**6**	1	**7**	162.63	4161.60	84.23	167.13	4261.18
All instances	76	21	**36**	19	**40**	9	**32**	140.54	3545.82	68.64	136.77	3436.24

Table 9
The impact of each part of the proposed GRASP method on the solution quality.

	Competitive solutions		Optimal solutions	Avg. total CPU times on 405 instances (seconds)
	MA	GRASP	GRASP	GRASP
Simple GRASP:				
Construction phase (i.e., Common_Insertion and Bidirectional_Insertion)	134	74	160	8.65
Common GRASP:				
Common_Insertion and Common_RemoveReplace	49	137	210	50.65
Bidirectional GRASP:				
Bidirectional_ Insertion and Bidirectional_RemoveReplace	20	143	227	106.14
Main algorithm	17	142	231	71.96

performs the state-of-the-art algorithm in 35.5% of the instances while the state-of-the-art solving method constructs tours with higher scores only for 4.25% of the instances. Our suggested algorithm also improves the best-reported results for three out of five instances without known optimal solutions.

Although our proposed method has an acceptable execution time when taking into account the quality of the constructed solutions, it is more computationally expensive than the state-of-the-art. This is due to the local search phase, which is the most time-consuming part. Therefore, future research should perhaps focus on other fast local search operators that are compatible with the OPHS.

Acknowledgments

The authors are deeply grateful to Professor Ali Divsalar for answering the questions about the MA and SVNS and providing us the code of the MA. We would also like to show our gratitude to the reviewers for their expert advice that significantly improved this manuscript.

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ejor.2019.11.010.

References

Angelelli, E., & Speranza, M. G. (2002a). The application of a vehicle routing model to a waste-collection problem: Two case studies. *Journal of the Operational Research Society, 53*(9), 944–952. doi:10.1057/palgrave.jors.2601402.

Angelelli, E., & Speranza, M. G. (2002b). The periodic vehicle routing problem with intermediate facilities. *European Journal of Operational Research, 137*(2), 233–247. doi:10.1016/S0377-2217(01)00206-5.

Benjamin, A. M., & Beasley, J. E. (2010). Metaheuristics for the waste collection vehicle routing problem with time windows, driver rest period and multiple disposal facilities. *Computers and Operations Research, 37*(12), 2270–2280. doi:10.1016/j.cor.2010.03.019.

Campos, V., Martí, R., Sánchez-Oro, J., & Duarte, A. (2014). GRASP with path relinking for the orienteering problem. *Journal of the Operational Research Society, 65*(12), 1800–1813. doi:10.1057/jors.2013.156.

Castro, M., Sörensen, K., Vansteenwegen, P., & Goos, P. (2013). A memetic algorithm for the travelling salesperson problem with hotel selection. *Computers and Operations Research, 40*(7), 1716–1728. doi:10.1016/j.cor.2013.01.006.

Castro, M., Sörensen, K., Vansteenwegen, P., & Goos, P. (2015). A fast metaheuristic for the travelling salesperson problem with hotel selection. *4OR, 13*(1), 15–34. doi:10.1007/s10288-014-0264-5.

Crevier, B., Cordeau, J. F., & Laporte, G. (2007). The multi-depot vehicle routing problem with inter-depot routes. *European Journal of Operational Research, 176*(2), 756–773. doi:10.1016/j.ejor.2005.08.015.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research, 6*(6), 791–812.

Divsalar, A., Vansteenwegen, P., & Cattrysse, D. (2013). A variable neighborhood search method for the orienteering problem with hotel selection. *International Journal of Production Economics, 145*(1), 150–160. doi:10.1016/j.ijpe.2013.01.010.

Divsalar, A., Vansteenwegen, P., Sörensen, K., & Cattrysse, D. (2014). A memetic algorithm for the orienteering problem with hotel selection. *European Journal of Operational Research, 237*(1), 29–49. doi:10.1016/j.ejor.2014.01.001.

Feo, T. A., & Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization, 6*(2), 109–133. doi:10.1007/BF01096763.

Fleurent, C., & Glover, F. (1999). Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory. *INFORMS Journal on Computing, 11*(2), 198–204. doi:10.1287/ijoc.11.2.198.

Gaudioso, M., & Paletta, G. (1992). A heuristic for the periodic vehicle routing problem. *Transportation Science, 26*(2), 86–92. doi:10.1287/trsc.26.2.86.

Gunawan, A., Lau, H. C., & Vansteenwegen, P. (2016). Orienteering Problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*. doi:10.1016/j.ejor.2016.04.059.

Hof, J., Schneider, M., & Goeke, D. (2017). Solving the battery swap station location-routing problem with capacitated electric vehicles using an AVNS algorithm for vehicle-routing problems with intermediate stops. *Transportation Research Part B: Methodological, 97*, 102–112. doi:10.1016/j.trb.2016.11.009.

Kek, A. G. H., Cheu, R. L., & Meng, Q. (2008). Distance-constrained capacitated vehicle routing problems with flexible assignment of start and end depots. *Mathematical and Computer Modelling, 47*(1–2), 140–152. doi:10.1016/j.mcm.2007.02.007.

Keshtkaran, M., & Ziarati, K. (2016). A novel GRASP solution approach for the Orienteering Problem. *Journal of Heuristics, 22*, 699–726. doi:10.1007/s10732-016-9316-7.

Kim, B. I., Kim, S., & Sahoo, S. (2006). Waste collection vehicle routing problem with time windows. *Computers and Operations Research, 33*(12), 3624–3642. doi:10.1016/j.cor.2005.02.045.

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research, 21*(2), 498–516. doi:10.1287/opre.21.2.498.

Lu, Y., Benlic, U., & Wu, Q. (2018). A hybrid dynamic programming and memetic algorithm to the traveling salesman problem with hotel selection. *Computers & Operations Research*. doi:10.1016/j.cor.2017.09.008.

Markov, I., Varone, S., & Bierlaire, M. (2016). Integrating a heterogeneous fixed fleet and a flexible assignment of destination depots in the waste collection vrp with intermediate facilities. *Transportation Research Part B: Methodological, 84*, 256–273. doi:10.1016/j.trb.2015.12.004.

Resende, M. G. C., & Ribeiro, C. C. (2010). Greedy randomized adaptive search procedures: Advances, hybridizations, and applications, *Handbook of metaheuristics* (57, pp. 283–319). Boston: Springer. doi:10.1007/b101874.

Schneider, M., Stenger, A., & Goeke, D. (2014). The electric vehicle-routing problem with time windows and recharging stations. *Transportation Science, 48*(4), 500–520. doi:10.1287/trsc.2013.0490.

Schneider, M., Stenger, A., & Hof, J. (2015). An adaptive VNS algorithm for vehicle routing problems with intermediate stops. *OR Spectrum, 37*(2), 353–387. doi:10.1007/s00291-014-0376-5.

Setak, M., Jalili Bolhassani, S., & Karimi, H. (2014). A node-based mathematical model towards the location routing problem with intermediate replenishment facilities under capacity constraint. *International Journal of Engineering, 27*(6), 911–920. doi:10.5829/idosi.ije.2014.27.06c.09.

Solomon, M. M. (1987). Algorithms for and scheduling problems the vehicle routing with time window constraints. *Operations Research, 35*(2), 254–265. doi:10.1287/opre.35.2.254.

Tarantilis, C. D., Zachariadis, E. E., & Kiranoudis, C. T. (2008). A hybrid guided local search for the vehicle-routing problem with intermediate replenishment facilities. *INFORMS Journal on Computing, 20*(1), 154–168. doi:10.1287/ijoc.l070.0230.

Tsiligirides, T. (1984). Heuristic methods applied to orienteering. *Journal of the Operational Research Society, 35*(9), 797–809. doi:10.1057/jors.1984.162.

Vansteenwegen, P., Souffriau, W., & Sörensen, K. (2012). The travelling salesperson problem with hotel selection. *Journal of the Operational Research Society, 63*(2), 207–217. doi:10.1057/jors.2011.18.

Willemse, E. J., & Joubert, J. W. (2016a). Constructive heuristics for the mixed capacity arc routing problem under time restrictions with intermediate facilities. *Computers and Operations Research, 68*, 30–62. doi:10.1016/j.cor.2015.10.010.

Willemse, E. J., & Joubert, J. W. (2016b). Splitting procedures for the mixed capacitated arc routing problem under time restrictions with intermediate facilities. *Data in Brief, 8*(5), 972–977. doi:10.1016/j.dib.2016.06.067.