Computers & Operations Research 62 (2015) 36–50

Contents lists available at ScienceDirect

Computers & Operations Research

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

Heuristics for the time dependent team orienteering problem: Application to tourist route planning ☆, ☆☆

Damianos Gavalas [a,f,*], Charalampos Konstantopoulos [b,f], Konstantinos Mastakas [c,f], Grammati Pantziou [d,f], Nikolaos Vathis [e,f]

[a] Department of Cultural Technology and Communication, University of the Aegean, Mytilene, Greece
[b] Department of Informatics, University of Piraeus, Piraeus, Greece
[c] School of Applied Mathematical and Physical Sciences, National Technical University of Athens, Athens, Greece
[d] Department of Informatics, Technological Educational Institution of Athens, Athens, Greece
[e] School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece
[f] Computer Technology Institute and Press 'Diophantus' (CTI), Patras, Greece

ARTICLE INFO

Available online 11 April 2015

Keywords:
Time Dependent Team Orienteering Problem with Time Windows
Tourist Trip Design Problem
Iterated Local Search
Clustering

ABSTRACT

The Time Dependent Team Orienteering Problem with Time Windows (TDTOPTW) can be used to model several real life problems. Among them, the route planning problem for tourists interested in visiting multiple points of interest (POIs) using public transportation. The main objective of this problem is to select POIs that match tourist preferences, taking into account a multitude of parameters and constraints while respecting the time available for sightseeing in a daily basis and integrating public transportation to travel between POIs (Tourist Trip Design Problem, TTDP). TDTOPTW is NP-hard while almost the whole body of the related literature addresses the non-time dependent version of the problem. The only TDTOPTW heuristic proposed so far is based on the assumption of periodic transit service schedules. Herein, we propose efficient cluster-based heuristics for the TDTOPTW which yield high quality solutions, take into account time dependency in calculating travel times between POIs and make no assumption on periodic service schedules. The validation scenario for our prototyped algorithms involved the transit network and real POI datasets compiled from the metropolitan area of Athens (Greece). Our TTDP algorithms handle arbitrary (i.e. determined at query time) rather than fixed start/end locations for derived tourist itineraries.

1. Introduction

In the Team Orienteering Problem with Time Windows (TOPTW) [25] we are given a set of nodes each associated with a profit, a visiting time and a time window, as well as a travel time between each pair of nodes; the objective is to find a fixed number of disjoint routes from a starting node to a destination node, each not exceeding a given time limit, that maximize the overall profit collected by visiting the nodes in all routes without violating their time windows. The TOPTW applies to several real-life problems. In this paper, we focus on the Tourist Trip Design Problem (TTDP) [28] which refers to a route-planning problem for tourists interested in visiting multiple points of interest (POIs). Solving the TTDP we derive daily tourist tours comprising ordered sets of POIs that match tourist preferences, thereby maximizing tourist satisfaction, while taking into account a multitude of parameters and constraints (e.g., distances among POIs, time estimated for visiting each POI, POIs' opening hours) and respecting the time available for sightseeing in daily basis. The problem is further complicated when considering the complexity of metropolitan transit networks commonly used by tourists to move from a POI to another. In this case, the required travel time depends on the departure time from the origin POI; hence, the TTDP can be modeled as a Time Dependent TOPTW (TDTOPTW) i.e. as a TOPTW with time dependent travel time between each pair of nodes.

The TOPTW and the TDTOPTW are NP-hard. While a significant number of heuristic approaches have been proposed in the literature for tackling the TOPTW (for a survey see [10,26]), to the best of our

☆This work has been supported by the EU FP7/2007-2013 under grant agreement No. 288094 (project eCOMPASS) and the EU CIP 2007-2013 under grant agreement No. 621133 (project HoPE).

☆☆A preliminary short version entitled "Efficient Heuristics for the Time Dependent Team Orienteering Problem with Time Windows", appears in the Proceedings of the International Conference on Applied Algorithms (ICAA 2014), Springer-Verlag, LNCS, vol. 8321; 2014. p. 151–62.

* Corresponding author.
E-mail addresses: dgavalas@aegean.gr (D. Gavalas), konstant@unipi.gr (C. Konstantopoulos), kmast@math.ntua.gr (K. Mastakas), pantziou@teiath.gr (G. Pantziou), nvathis@softlab.ntua.gr (N. Vathis).

http://dx.doi.org/10.1016/j.cor.2015.03.016
0305-0548/

knowledge, the only TDTOPTW heuristic has been recently proposed by Garcia et al. [9]. The algorithm of Garcia et al. is based on the assumption of periodic service schedules which is clearly not valid in realistic transportation networks, wherein arrival/departure frequencies typically vary within the service's operational periods.

Herein, we propose two novel randomized metaheuristic approaches based on the technique of iterated local search [18], the Time Dependent CSCRoutes (TD_CSCR) and the Time Dependent Slack CSCRoutes (TD_SℓCSCR) algorithms which address the above described shortcoming of the existing TDTOPTW approach. The main incentive behind our approaches is to motivate visits to topology areas featuring high density of 'promising' (i.e. highly profitable) candidate vertices, while taking into account time dependency (i.e. multimodality) in calculating travel times from one vertex to another; the aim is to derive high quality routes (i.e. maximizing the total collected profit) and minimize the time delays incurred in transit stops, while not sacrificing the time efficiency required for online applications. The two algorithms favor solutions with increased number of walking over public transit transfers (the latter are considered costly and typically less attractive to tourists than short walking transfers). Both algorithms are extended to tackle the case which involves arbitrary (i.e. determined at query time) rather than fixed starting/destination locations for derived tourist itineraries.

In addition to the TD_CSCR and TD_SℓCSCR algorithms, we have also implemented the Average travel times CSCRoutes (AvgCSCR) algorithm which uses average (rather than time dependent) travel times between POIs. In effect, the AvgCSCR algorithm reduces the TDTOPTW to TOPTW. Having obtained a TOPTW solution, AvgCSCR employs two additional steps to ensure route feasibility and further improve the solution's quality.

Our prototyped algorithms have been tested in terms of various performance parameters (solutions' quality, execution time, percentage of transit transfers over total transfers, etc.) upon real test instances (i.e. set of POIs and accommodation facilities) compiled from the wider area of Athens, Greece; the calculation of time dependent travel times has been carried out over the Athens metropolitan transit network. The performance of the TD_CSCR, TD_SℓCSCR and AvgCSCR algorithms has been compared against a time dependent extension of the most efficient known TOPTW heuristic (Vansteenwegen et al. [27]) as well as the approach proposed by Garcia et al. [9] using precalculated average travel times between POIs.

The remainder of this paper is organized as follows: Section 2 overviews the related work while in Section 3 our novel cluster-based heuristics for the TDTOPTW are presented. An algorithmic solution for the TTDP is presented in Section 4. The experimental results are discussed in Section 5 while Section 6 concludes our work.

2. Related work

The TOPTW is an extension of the Orienteering Problem (OP) [24,26] also known as the Maximum Collection Problem. In the OP, several locations with an associated profit have to be visited within a given time limit. The goal of the problem is to maximize the overall score collected on a single tour starting from and ending at a depot node. The OP is NP-hard [12,15]. The team orienteering problem (TOP) [3] extends the OP considering multiple routes while the TOP with time windows (TOPTW) [25] considers visits to locations within a predefined time window. The TOPTW is NP-hard, since it extends OP, hence exact solutions for the TOPTW can be applied only to instances with a limited number of nodes. As a result, the main body of the TOPTW literature exclusively involves heuristic algorithms [8,14,13,17,19,23,27]. ACS [19], Enhanced ACS [8] and the approach of Tricoire et al. [23] are known to yield the highest quality solutions. The most efficient known heuristic is based on Iterated Local Search (ILS) [27], offering a fair compromise with respect to execution time versus deriving routes of reasonable quality [26]. However, the ILS approach treats each POI separately, thereby commonly overlooking highly profitable areas of POIs situated far from current location considering them too time-expensive to visit. In [11] CSCRatio and CSCRoutes, two cluster-based extensions to ILS, have been proposed to address the aforementioned weakness. The main incentive behind these approaches is to favor visits to topology areas featuring high density of good candidate nodes. This is achieved through a clustering phase which groups nodes based on geographical criteria, and encouraging visiting topology areas, even distant ones.

Erkut and Zhang in [6] considered the Maximum Collection Problem with Time Dependent Rewards (MCPTDR) where each node's profit decreases linearly over time, and the objective is to maximize the sum of the rewards collected in a single tour. The problem may find applications in cases where there is a time-dependent penalty for delays in service. The authors proposed a mixed integer programming formulation, and a penalty-based greedy heuristic algorithm and an exact branch-and-bound algorithm for solving the MCPTDR. In [22] the problem of scheduling technicians for planned maintenance of geographically distributed equipment is formulated as a Multiple Tour Maximum Collection Problem with Time-Dependent rewards (MTMCPTD). In the MTMCPTD, the rewards are assigned to the tasks based on the "urgency" for completing a task on a given day and the objective is to determine a set of tours, each corresponding to a technician's schedule on a particular day, such that the total reward collected during the scheduling horizon is maximized. The authors introduced a tabu-based search heuristic for solving the MTMCPTD. The Orienteering Problem with Variable Profits was introduced by Erdogan and Laporte [5] as a variant of the OP in which the percentage of the collected profit at each node depends either on the number of discrete passes or in an alternative model, on the continuous amount of time spent at the node. Yu et al. recently [30] presented a mixed integer programming approach for solving a more general problem that allows multiple starting nodes (depots) and the profit collected at each node is characterized by some non-decreasing function over the time spent at this node. Their method is extended to the case of multiple tours.

The previous paragraph referred to variants of the OP and TOP with time dependent rewards. The Time Dependent OP (TDOP) considering time dependent travel costs, was first introduced by Formin and Lingas in [7]. TDOP is MAX-SNP-hard since a special case of the TDOP, the time-dependent maximum scheduling problem is MAX-SNP-hard [21]. Fomin and Lingas [7] give a $(2+\epsilon)$ approximation algorithm for rooted and unrooted TDOP. Verbeeck et al. [29] suggested a mathematical formulation of the TDOP and proposed a fast local search based metaheuristic to tackle the problem. This algorithm is inspired by an ant colony system (ACS) and utilizes a speed-up time-dependent local search procedure equipped with a local evaluation metric. Abbaspour et al. [1] investigated a variant of the Time Dependent OP with Time Windows (TDOPTW) in urban areas, and proposed a genetic algorithm for solving the problem. The work of Garcia et al. [9] is the first to address algorithmically the TDTOPTW. The authors presented two different approaches to solve TDTOPTW, both applied on real urban test instances (POIs and bus network of San Sebastian, Spain). The first approach involves a pre-calculation step, computing the average travel times between all pairs of POIs, allowing reducing the TDTOPTW to a regular TOPTW, solved using the insertion phase part of ILS. In case that the derived TOPTW solution is infeasible (due to violating the time windows of nodes included in the solution), a number of visits are removed. The second approach uses time-dependent travel times but it is based

on the simplified assumption of periodic service schedules; this assumption, clearly, does not hold in realistic urban transportation networks, especially on non-fixed-rail services (e.g. buses). Herein, we propose an algorithmic approach that relaxes this assumption and is applicable to realistic transit networks.

3. The proposed TDTOPTW heuristics

The TDTOPTW extends TOPTW considering time dependent travel costs among nodes i.e., travel costs using public transportation. In the TOPTW we are given a complete directed graph $G=(V,E)$ where V denotes the set of locations with $N=|V|$; a set $P=\{p_1,p_2,\ldots,p_{N_p}\}\subseteq V$ denoting the set of POIs; an integer m denoting the number of days the trip shall last, and a time budget B. The main attributes of each node $p_i\in P$ are the service or visiting time (visit_i), the profit gained by visiting p_i (profit_i), and each day's time window $[\text{open}_{ir},\text{close}_{ir}], r=1,2,\ldots,m$, (a POI may have different time windows per day). Every link $(u,v)\in E$ denotes the transportation link from u to v and is assigned a travel time. The objective is to find m disjoint routes each starting from a starting location $s\in V$ and ending at a location $t\in V$, each with overall duration limited by the time budget B, that maximize the overall profit collected by the visited POIs in all routes. The TDTOPTW is the extension of TOPTW where the travel time from a location $u\in V$ to a location $v\in V$ (as well as the arrival time at v) depends on the time leaving from u and the chosen transportation mode (e.g. on foot or public transportation).

In the TDTOPTW we assume that the starting and ending locations may be different for different routes. Therefore, s_r, $t_r\in V$ denote the starting, terminal location respectively, of the rth route, and st_r, et_r denote the starting, ending time respectively, of the rth route, $r=1,2,\ldots,m$. Note that s_r, t_r denote locations which may coincide with POIs. Suppose that the terminal location t_r coincides with POI p and that p is also considered as a candidate for inclusion; in that case, the travel cost from p to t_r equals zero. Therefore, the starting and terminal locations are not associated with visiting times and/or time windows and the time budget B for each route r is $B=\text{et}_r-\text{st}_r$, $r=1,2,\ldots,m$.

Fig. 1 depicts a typical tourist route from a start location s_r to an end location t_r via POIs p_i, p_j and p_k, each associated with a time window and a required time to visit. Each transit transfer among two locations is subject to a delay (e.g. t_3-t_2 when leaving from p_i to p_j). Such delays do not occur when taking the walking transfer option (e.g. transfer from p_k to t_r). Each visit is also likely to be delayed if the tourist arrives at a POI before its opening hour (e.g. waiting of t_5-t_4 prior to start visiting p_j). A TDTOPTW solver

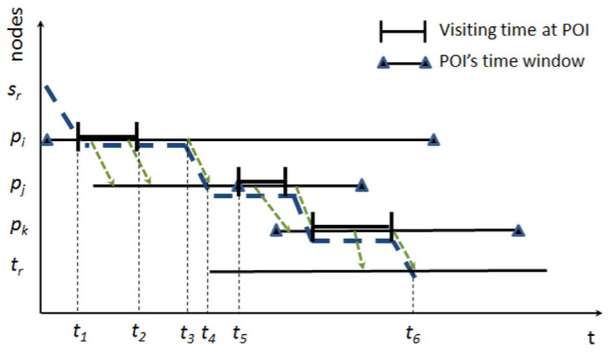

Fig. 1. Illustration of a tourist route (blue dashed line) from s_r to t_r via POIs p_i, p_j and p_k. Green dashed arrows indicate available multimodal transfer options among POIs. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

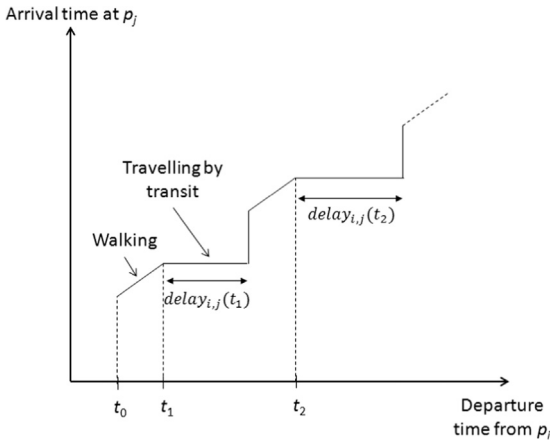

Fig. 2. Arrival time walking and using public transportation.

should minimize the overall delays incident along the routes and exploit the time saved in order to accommodate visits to additional POIs.

Fig. 2 illustrates the arrival time at a POI p_j for a tourist that has previously visited p_i. Delays incurred to embark on the next service may vary, i.e. we make no assumption of periodic service schedules. Also, when considering certain departure times (e.g. between t_0 and t_1) walking might be a faster option than waiting for the next transit service.

The solution strategy employed by the proposed TDTOPTW algorithms is based on the CSCRoutes algorithm for the TOPTW presented in [11]. CSCRoutes is an iterated local search-based [18] heuristic which employs the global k-means clustering algorithm [2,16] to organize the POIs into an appropriate number of clusters. Once the clusters of POIs have been formed, a list (listOfClusterSets) containing a specific number of different sets of m clusters, is constructed. CSCRoutes executes a loop for a number of times equal to the size of the listOfClusterSets. Within this loop, firstly, all POIs in the current solution's routes are removed and the route initialization phase **RouteInitPhase** is executed. RouteInitPhase takes as an argument a set of m clusters from the listOfClusterSet and proceeds as follows: for each cluster in the set, it finds the POI with the highest ratio of profit squared over insertion cost and inserts it into one of the empty routes. Note that each of the m inserted POIs comes from a different cluster. In this way the algorithm encourages searching different topology areas and avoids getting trapped at specific high-scored nodes. After the RouteInitPhase ends, the algorithm executes an inner loop combining a local search procedure and a shake step to escape from a local optimum. The local search procedure iteratively applies an insertion step. At each insertion step (**CSCRoutes_Insert**) a node is inserted in a route, ensuring that all following nodes along the route remain feasible to visit, i.e. the visit at each node starts within its time window and the length of each route remains at most B. At the shake step (**Shake**) a specified number of successive nodes in each route are removed. The intuition of this step is to escape from the current local optimum decreasing each route's length in order to reach a better local optimum in the next local search procedure. Note that before the inner loop starts, the parameters removeNumber (indicating the number of the consecutive nodes to be removed from each route) and startNumber (indicating where to start removing nodes on each route of the current solution) of the Shake step are initialized to 1. Then the loop is executed for a specific number of times (maxIterations) while the profit of the best solution is not improved. For the sake of comple-

teness, the pseudo-code of CSCRoutes Algorithm is given below (Algorithm 1) (details may be found in [11]).

Algorithm 1. CSCRoutes(numberOfClusters, maxIterations).

```
run the global k-means algorithm with k=numberOfClusters
construct the list listOfClusterSets
while listOfClusterSets is not empty do
   remove all POIs visited in the currentSolution
   theClusterSetIdToInsert ← listOfClusterSets.pop
   RouteInitPhase(theClusterSetIdToInsert)
   startNumber ← 1; removeNumber ← 1; notImproved ← 0
   while notImproved < maxIterations do
     while not local optimum do
       CSCRoutes_Insert
     end while
     if currentSolution.profit > bestSolution.profit then
       bestSolution ← currentSolution; removeNumber ← 1;
   notImproved ← 0
     else increase notImproved by 1
     end if
     if removeNumber > currentSolution.sizeOfLargestTour / 2 then
       removeNumber ← 1
     end if
     Shake(removeNumber,startNumber)
     increase startNumber by removeNumber
     increase removeNumber by 1
     if
   startNumber ≥ currentSolution.sizeOfSmallestTour then
       decrease startNumber by currentSolution.
   sizeOfSmallestTour
     end if
   end while
end while
return bestSolution
```

Unlike CSCRoutes, the algorithms introduced in this paper, handle time dependent travel times among different locations/POIs. Therefore, they employ different insertion steps from CSCRoutes. Namely, the insertion steps take into account the time dependency of the travel time between each pair of locations, i.e. the travel time from location u to location v may vary between different departure times, and the waiting time for public transport depends on the arrival time at u. Apart from the insertion step, another local search step is also applied, the **Replace** step. In the Replace step, a node included in the solution is replaced by a non-included node with a higher profit as long as this replace retains the route feasible, i.e. time windows and route travel budget are not violated, and furthermore the replace respects the cluster routes. In Section 3.1 we present the feasibility criterion for inserting a new POI in a route and replacing a visited node by a non-included node in the case of time dependent travel costs while in the following three subsections the algorithmic approaches for solving the TDTOPTW are described. Namely, the TD_CSCR algorithm, the TD_SℓCSCR algorithm and the AvgCSCR algorithm are described in detail.

3.1. Time dependent insertion feasibility

In order to have the pairwise time dependent travel cost among all locations, for each (u,v), u, $v \in V$, we precalculate the walking time from u to v (might be ∞, when too far to walk) and a set S_{uv} of transit transfer travel times based on the timetable information of the transit network. Specifically, S_{uv} contains all the non-dominated departure-travel time pairs $(\mathrm{dep}_i^{uv}, \mathrm{trav}_i^{uv})$, $i = 1, 2, \ldots, |S_{uv}|$, in ascending order of dep_i^{uv}, where dep_i^{uv} is a departure time and trav_i^{uv} is the corresponding travel time of a transit service connecting u and v. We consider that a departure-travel time pair from node u to node v $(\mathrm{dep}_1, \mathrm{trav}_1)$ dominates a pair $(\mathrm{dep}_2, \mathrm{trav}_2)$ if $\mathrm{dep}_1 + \mathrm{trav}_1 \leq \mathrm{dep}_2 + \mathrm{trav}_2$ and $\mathrm{dep}_1 > \mathrm{dep}_2$. Note that departing from u at time t with $\mathrm{dep}_i^{uv} < t \leq \mathrm{dep}_{i+1}^{uv}$, actually means either departing from u at dep_{i+1}^{uv} using public transportation, or start at time t walking from u to v. More specifically, the arrival time at v will be equal to the earliest of the times $\mathrm{dep}_{i+1}^{uv} + \mathrm{trav}_{i+1}^{uv}$ and $t + \mathrm{walking}_{u,v}$, where $\mathrm{walking}_{u,v}$ is the walking time from u to v. To determine all the non-dominated pairs in S_{uv} we employ the algorithm of Dibbelt et al. [4].

For a specified time t, the departure time from u to v at t using public transport, $\mathrm{deptime}_{u,v}(t)$, is defined as the earliest possible departure time from u to v, i.e.,

$$\mathrm{deptime}_{u,v}(t) = \min_i \{\mathrm{dep}_i^{uv} \mid (\mathrm{dep}_i^{uv}, \mathrm{trav}_i^{uv}) \in S_{uv} \text{ and } t \leq \mathrm{dep}_i^{uv}\} \quad (1)$$

Then, the travel time from u to v at t using public transport, $\mathrm{travtime}_{u,v}(t)$, is such that $(\mathrm{deptime}_{u,v}(t), \mathrm{travtime}_{u,v}(t)) \in S^{uv}$, and the departure delay at time t due to the use of public transport, is $\mathrm{delay}_{u,v}(t) = \mathrm{deptime}_{u,v}(t) - t$. For instance, in Fig. 3, $\mathrm{deptime}_{u,v}(t_1) = \mathrm{dep}_1^{uv}$. Therefore, the total traveling cost from u to v at a specified time t, $\mathrm{traveling}_{u,v}(t)$, is

$$\mathrm{traveling}_{u,v}(t) = \min\{\mathrm{walking}_{u,v}, \mathrm{delay}_{u,v}(t) + \mathrm{travtime}_{u,v}(t)\} \quad (2)$$

For a POI p_i in a route r the following variables are defined:

- wait_i, denoting the waiting time at p_i before its time window starts; $\mathrm{wait}_i = \max(0, \mathrm{open}_{ir} - \mathrm{arrive}_i)$.
- start_i, denoting the starting time of the visit at p_i; $\mathrm{start}_i = \mathrm{arrive}_i + \mathrm{wait}_i$.
- leave_i, denoting the time the visit at p_i completes, i.e., the departure time from p_i; $\mathrm{leave}_i = \mathrm{start}_i + \mathrm{visit}_i$.
- arrive_i, denoting the arrival time at p_i; $\mathrm{arrive}_i = \mathrm{leave}_{\mathrm{prev}(i)} + \mathrm{traveling}_{\mathrm{prev}(i),i}(\mathrm{leave}_{\mathrm{prev}(i)})$, where $\mathrm{leave}_{\mathrm{prev}(i)}$ is the departure time from the previous node of p_i in route r ($\mathrm{prev}(i)$). We assume that $\mathrm{arrive}_{s_r} = \mathrm{st}_r$.
- $\mathrm{maxStart}_i$, denoting the latest time the visit at p_i can start without violating the time windows of the nodes following p_i; $\mathrm{maxStart}_i = \min(\mathrm{close}_{ir}, \max\{t : t + \mathrm{traveling}_{i,\mathrm{next}(i)}(t) \leq \mathrm{maxStart}_{\mathrm{next}(i)}\} - \mathrm{visit}_i)$, where $\mathrm{next}(i)$ is the node following p_i in r. We assume that $\mathrm{maxStart}_{t_r} = \mathrm{et}_r$.

A POI p_k can be inserted in route r between POIs p_i and p_j if the arrival time at p_k does not violate p_k's time window and the arrival at p_j does not violate the time window of p_j as well as the time windows of the nodes following p_j in r. The total time cost for p_k's insertion is defined as shift_k^{ij} (insertion cost) and is equal to the time the arrival at p_j will be delayed. In particular shift_k^{ij} equals to the time required to travel from p_i to p_j having visited p_k in between minus the time taken for traveling directly from p_i to p_j.

$$\mathrm{shift}_k^{ij} = (\mathrm{traveling}_{i,k}(\mathrm{leave}_i) + \mathrm{wait}_k^i + \mathrm{visit}_k + \mathrm{traveling}_{k,j}(\mathrm{leave}_k)) - \mathrm{traveling}_{i,j}(\mathrm{leave}_i) \quad (3)$$

where wait_k^i denotes the waiting time at POI p_k following a visit at p_i. Fig. 4 illustrates an example of inserting p_k, between p_i and p_j shifting the visit at p_j later on time.

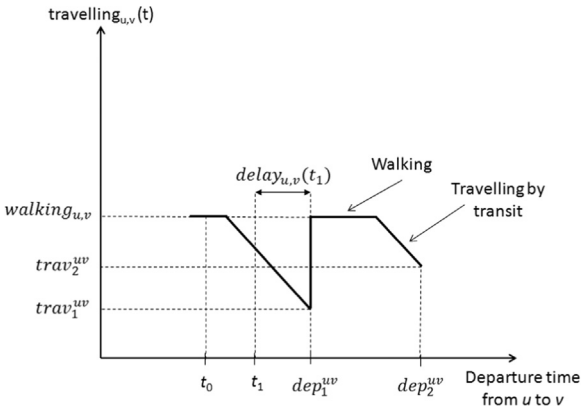

Fig. 3. Traveling time from u to v as a function of the departure time from u.

Note that the insertion of p_k between p_i and p_j is feasible when

$$\text{arrive}_k \le \text{close}_{kr} \quad \text{and} \quad \text{shift}_k^{ij} \le \text{maxStart}_j - \text{arrive}_j \tag{4}$$

A pseudo-code implementation of the function shift(k, i, j, r) follows Algorithm 2, which calculates the insertion cost shift_k^{ij} in route r. The function returns ∞ if the insertion of p_k is infeasible.

Algorithm 2. shift(k, i, j, r).

```
result ← ∞
arrive_k ← leave_i + traveling_{i,k}(leave_i)
if arrive_k ≤ close_kr then
    wait_k^i ← max(0, open_kr − arrive_k)
    leave_k ← arrive_k + wait_k + visit_k
    costAfterInsert ← traveling_{i,k}(leave_i) + wait_k + visit_k
     + traveling_{k,j}(leave_k)
    shift_k^ij ← costAfterInsert − traveling_{i,j}(leave_i)
    if shift_k^ij ≤ maxStart_j − arrive_j then
       result ← shift_k^ij
    end if
end if
return result
```

Apart from inserting a new POI in a route, our algorithms encourage replacing an already included POI by a higher profit POI. Considering that in route r, the POIs p_i, p_l, p_j are consecutive and p_k is not included in the solution, p_k can replace p_l if the arrival at p_k takes place within its time window and the time windows of p_j and the POIs following p_j are not violated. Extending the terminology introduced previously, let $\text{shift}_{k,l}^{ij}$ be the difference in the insertion cost from replacing POI p_l by p_k. Then,

$$\begin{aligned}\text{shift}_{k,l}^{ij} = {} & (\text{traveling}_{i,k}(\text{leave}_i) + \text{wait}_k^i + \text{visit}_k + \text{traveling}_{k,j}(\text{leave}_k)) \\ & - (\text{traveling}_{i,l}(\text{leave}_i) + \text{wait}_l + \text{visit}_l + \text{traveling}_{l,j}(\text{leave}_l))\end{aligned} \tag{5}$$

Note that replacing p_l by p_k is feasible if

$$\text{arrive}_k \le \text{close}_{kr} \quad \text{and} \quad \text{shift}_{k,l}^{ij} \le \text{maxStart}_j - \text{arrive}_j \tag{6}$$

3.2. The Time Dependent CSCRoutes (TD_CSCR) algorithm

The TD_CSCR algorithm extends CSCRoutes in order to handle time dependent travel times among different locations/POIs. The insertion step (**TD_CSCR_Insert**) modifies the insertion step of CSCRoutes (**CSCRoutes_Insert**) taking into account the time dependency in travel costs and additionally incorporating a factor

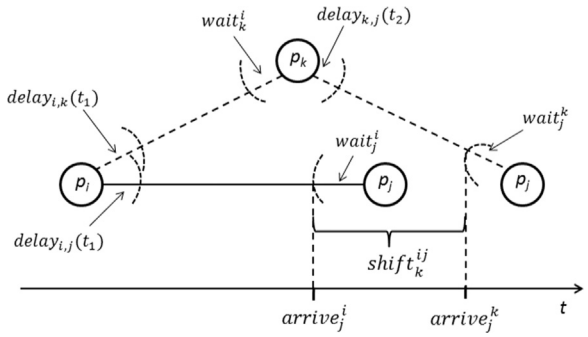

Fig. 4. Illustration of p_k insertion between p_i and p_j.

of randomness to add diversification in the solution search. Apart from the insertion step another local search step is also applied, namely a **Replace** step. In the **Replace** step a visited POI is replaced by a higher profit POI in order to improve solution's quality.

CSCRoutes uses the notion of *Cluster Route* (*CR*) defined as follows: given a route r of a TOPTW solution, any maximal sub-route in r comprising a sequence of nodes within the same cluster C is called a *Cluster Route* (*CR*) of r *associated with cluster* C and denoted as CR_C^r. The CSCRoutes algorithm is designed to construct routes that visit each cluster at most once, i.e. if a cluster C has been visited in a route r it cannot be revisited in the same route and therefore, for each cluster C there is only one cluster route in any route r associated with C. The only exception allowed is when the start and the terminal nodes of a route r belong to the same cluster C'. In this case, a route r may start and end with nodes of cluster C', i.e. C' may be visited twice in the route r and therefore, for a route r there might be two cluster routes $CR_{C'}^r$. The insertion step of CSCRoutes does not allow the insertion of a POI p_k in a route r, if this insertion creates more than one cluster routes CR_C^r for some cluster C. Therefore, a POI cannot be inserted at any position in the route r [11].

The **TD_CSCR_Insert** comprises a modification of **CSCRoutes_Insert** taking into consideration the time dependency of travel times among locations/POIs, while enhancing randomization. Given a route r let CR_f^r be the first cluster route (starting at s_r) in r, and CR_l^r be the last cluster route (ends at t_r) in r. Let also clustersIn(r) be a set containing any cluster C visited in r, and cluster(p) be the cluster containing p. Given a candidate for insertion POI p_k and a route r **TD_CSCR_Insert** distinguishes among the following cases:

- cluster(s_r) = cluster(t_r)
  - if clustersIn(r) = {cluster(s_r)} then p_k can be inserted anywhere in the route.
  - if clustersIn(r) ≠ {cluster(s_r)} and cluster(p_k) = cluster(s_r) then p_k can be inserted in CR_f^r and CR_l^r
  - if clustersIn(r) ≠ {cluster(s_r)} and cluster(p_k) ≠ cluster(s_r) and cluster(p_k) ∉ clustersIn(r) then p_k can be inserted after every end of a CR except for CR_l^r
  - if clustersIn(r) ≠ {cluster(s_r)} and cluster(p_k) ≠ cluster(s_r) and cluster(p_k) ∈ clustersIn(r) then p_k can be inserted anywhere in $CR_{\text{cluster}(p_k)}^r$
- cluster(s_r) ≠ cluster(t_r)
  - if cluster(p_k) = cluster(s_r) then p_k can be inserted everywhere in CR_f^r
  - if cluster(p_k) = cluster(t_r) then p_k can be inserted everywhere in CR_l^r
  - if cluster(p_k) ∈ clustersIn(r)/{cluster(s_r), cluster(t_r)}, then p_k can be inserted everywhere in $CR_{\text{cluster}(p)}^r$

- if cluster(p_k)$\notin$clustersIn(r) then p_k can be inserted after the end of any CR in r except for CR_l^r

For each POI p_k not included in a route r, the position with the minimum time consumption (shift) is considered, i.e. the insertion between POIs p_i, p_j. Then, the POI selected for insertion is the one with the highest ratio:

$$\text{ratio}_k^{ij} = \frac{\text{profit}_k^2}{\text{shift}_k^{ij}}(1+\alpha \cdot f(\text{shift}_k^{ij}, \text{wait}_j^i + \text{delay}_{j,next(j)})) \cdot \text{rand} \quad (7)$$

where $f(x,y)=1$ if $x \le y$ and 0 otherwise, and α takes the values of $0, \frac{1}{2}$ and 1, depending on the number of iterations executed by the algorithm. In particular, for the first $\frac{1}{3}$ iterations α is equal to 0, it increases to $\frac{1}{2}$ in the second $\frac{1}{3}$ iterations and becomes 1 in the final iterations. The last factor of the ratio (rand), is a number randomly generated from a specified interval [1, maxFactor], where maxFactor is a predefined constant. To see the incentive behind (7) notice that $\text{profit}_k^2/\text{shift}_k^{ij}$ denotes preference for important (i.e. highly profitable) POIs associated with relatively small insertion cost, while function f takes the value of 1 when $\text{shift}_k^{ij} \le \text{wait}_j + \text{delay}_j$, i.e. the cost of inserting k between i and j is less than the waiting time plus the delay time at j and, therefore, k's insertion does not affect the arrival time at the node following j. In the first iterations ($\alpha=0$) the algorithm gives preference to the insertion of POIs with high profit and relatively small insertion cost while in the last iterations ($\alpha=1$) favors insertion of POIs that not only have high profit and relatively small insertion cost but also best take advantage of any left unexploited time (i.e. waiting and delays) remaining throughout the routes. Furthermore, the randomization factor of the ratio, makes the search of the solution space more effective, not allowing the solution to get trapped in the same local optimum over and over again and thus more diversity is obtained. Among all candidate POIs, TD_CSCR algorithm selects for insertion the one associated with the highest ratio.

Another feature of the TD_CSCR algorithm's insertion step (**TD_CSCR_Insert**) is that it gives preference to the insertion of POIs that belong to high "quality" clusters i.e. to clusters having several "high" profit POIs not yet included into a route. This is done by using for each – candidate for insertion – POI p_k a modified value of its profit in the calculation of ratio_{ij}^k. We first define the notion of the "quality" of a cluster. Let PROFIT be the highest profit of a POI in P. We say that a cluster has i-quality if it has at least i POIs (not yet inserted in a route) each with profit at least $(i-1)*\text{PROFIT}/10+1$. Then, the quality of a cluster C is defined as the maximum i such that C has i-quality. The i-quality metric has been inspired by the h-index, which measures the productivity and impact of the published work of a scientist or scholar. To illustrate, if PROFIT=100 then a cluster C has 2-quality if it has at least 2 POIs with profit in the range [11,100] while it has 5-quality if it has at least 5 POIs with profit in the range [41,100]. If C does not include at least 6 POIs with profit in the range [51,100] then the quality of C is 5. The profit_k of a *POI* k is modified according to the quality of the cluster it belongs to by as follows: if the quality of the cluster is i then the value of the profit of k is considered to be equal to $(1+0,02 \cdot i) \cdot \text{profit}_k$. This new value is used for the calculation of the ratio_{ij}^k.

Once a POI p_k is inserted between p_i and p_j in a route r, the variable values of all POIs in r need to be updated. The variables of p_k are updated as follows:

$$\text{arrive}_k = \text{leave}_i + \text{traveling}_{i,k}(\text{leave}_i)$$
$$\text{wait}_k^i = \max(0, \text{open}_{kr} - \text{arrive}_k)$$
$$\text{start}_k = \text{arrive}_k + \text{wait}_k^i$$
$$\text{leave}_k = \text{arrive}_k + \text{wait}_k^i + \text{visit}_k$$
$$\text{maxStart}_k = \min(\text{close}_{kr}, \max\{t : t + \text{traveling}_{k,j}(t) \le \text{maxStart}_j\} - \text{visit}_k)$$

Note that for each POI after p_k, the variables arrive, wait, start and leave should be updated in a similar way with p_k while variable maxStart remains unchanged. For each POI p_l before p_k the value of maxStart_l is the only one that should be updated, recursively computed as follows:

$$\text{maxStart}_l = \min(\text{close}_{lr}, \max\{t : t + \text{traveling}_{l,next(l)}(t) \le \text{maxStart}_{next(l)}\} - \text{visit}_l) \quad (8)$$

The pseudocode of **TD_CSCR_Insert** is given below (Algorithm 3).

Algorithm 3. TD_CSCR_Insert.

```
for each candidate POI p_k do
  for each route r do
    if cluster(s_r)= cluster(t_r) then
      if clustersIn(r) = {cluster(s_r)} then
        Search all positions in r for the highest ratio
      else // clustersIn(r) ≠ {cluster(s_r)}
        clusterID ← cluster(p_k)
        if clusterID = cluster(s_r) then
          Search all positions in CR_f^t and CR_l^t for the highest ratio
        else // clusterID ≠ cluster(s_r)
          if clusterID ∉ clustersIn(r) then
          Search all positions in r that are the end of a CR, for the highest ratio
          else // clusterID ∈ clustersIn(r)
            Search all positions in CR^r_clusterID for the highest ratio
          end if
        end if
      end if
    else // cluster(s_r) ≠ cluster(t_r)
      clusterID ← cluster(p_k)
      if clusterID = cluster(s_r) then
        Search all positions in CR_f^r for the highest ratio
      else // clusterID ≠ cluster(s_r)
        if clusterID = cluster(t_r) then
        Search all positions in CR_l^r for the highest ratio
        else // clusterID ≠ cluster(t_r)
          if clusterID ∉ clustersIn(r) then
          Search all positions in r that are the end of a CR, for the highest ratio
          else // clusterID ∈ clustersIn(r)
            Search all positions in CR^r_clusterID for the highest ratio
          end if
        end if
      end if
    end if
  end for
end for
Insert the POI p_l with the highest ratio.
Update the variables of each POI in r and the set of cluster
    members for each cluster.
```

The **TD_CSCR** algorithm iteratively applies the insertion step followed by another local search step namely, the **Replace** step. The **Replace** is also applied with a given probability (probReplace) within the first local search procedure (with the insert neighborhood) to add diversification and further increase the solution's profit. The replace step tries to improve a solution's profit by

replacing an included POI v by another non-included POI u. The pair of POIs (u,v) selected will have the highest possible difference in the profit ($\text{profit}_u - \text{profit}_v$), while the replacement must not violate the time window as well as the cluster route constraints. Taking into account the cluster route constraints, a non-included POI cannot replace any POI included in the solution. For each non-included POI p_k and a route r of the solution, let replacableSet(p_k, r) be the set of nodes visited in r that can be replaced by p_k without violating the cluster route constraints. Then the set replacableSet (p_k, r) is constructed by distinguishing among cases depending on whether or not the cluster of p_k belongs to the set clustersIn(r) as follows:

- if $\text{cluster}(p_k) \notin \text{clustersIn}(r)$, then replacableSet($p_k$, r) consists of the first node of each cluster route in r apart from CR_f^r, and the last node of each cluster route in r apart from CR_l^r;
- if $\text{cluster}(p_k) \in \text{clustersIn}(r) \neq \{\text{cluster}(s_r), \text{cluster}(t_r)\}$, then replacableSet($p_k$, r) consists of each node in the cluster route associated with cluster(p_k) as well as the node preceding the start of the cluster route (if it is not s_r) and the node following the end of the cluster route (if it is not t_r);
- if $\text{cluster}(p_k) = \text{cluster}(s_r) \neq \text{cluster}(t_r)$, then replacableSet($p_k$, r) consists of each node in CR_f^r except for s_r, as well as the node following the end of CR_f^r (if it is not t_r);
- if $\text{cluster}(p_k) = \text{cluster}(t_r) \neq \text{cluster}(s_r)$, then replacableSet($p_k$, r) consists of each node in CR_l^r apart from t_r and additionally the node preceeding the start of its cluster route (as long as it is not s_r);
- if $\text{cluster}(p_k) = \text{cluster}(s_r) = \text{cluster}(t_r)$ and $\text{clustersIn}(r) = \{\text{cluster}(s_r)\}$, replacableSet($p_k$, r) consists of each node in r except for the start and the end of the route;
- if $\text{cluster}(p_k) = \text{cluster}(s_r) = \text{cluster}(t_r)$ and clustersIn(r) contains at least two clusters, then replacableSet(p_k, r) consists of each node in CR_f^r except for s_r, the node following the end of CR_f^r, each node in CR_l^r except for t_r as well as the node preceeding the start of CR_l^r.

The replace step proceeds as follows: first, for each non-inserted POI p_k and for each route r in the current solution the set replacableSet(p_k, r) is constructed as described above. Then for each node v in replacableSet(p_k, r), it is checked (using (6)) whether replacing v by p_k is feasible. If p_k can replace v, it is checked whether the difference between the profits of p_k and v is the maximum found so far. If this is the case, p_k and v are stored in variables bestIn and bestOut respectively. At the end of the replace step the pair of nodes with the highest difference of profit (bestIn, bestOut) is found and bestIn replaces bestOut. Finally, the variable values of all POIs in the route r where the replacement is taking place are updated in a similar way with the update followed in the insertion step. In the sequel the pseudocode of the **Replace** (Algorithm 4)is given.

Algorithm 4. Replace.

```
bestDif ← 0
for each non-included POI p_k do
  for each route r do
    construct replacableSet(p_k, r)
    for each node v ∈ replacableSet(p_k, r) do
      if it is feasible to replace v by p_k then
        if profit_{p_k} − profit_v > bestDif then
          bestIn ← p_k, bestOut ← v, bestDif ← profit_{p_k} − profit_v
        end if
      end if
    end for
  end for
end for
Replace bestOut by bestIn.
Update the variables of each POI in r containing bestIn and the
    set of cluster members for each cluster.
```

At each iteration of the **TD_CSCR** algorithm the insertion step is applied followed by the replace step until a local optimum is reached. Then the shake step follows to escape from the local optimum. During the shake step removeNumber consecutive POIs are removed from each route staring from the position startNumber; if, during the removal process, the route's terminal location is reached then the removal continues after the starting location. Note that due to different route sizes, the values of removeNumber and startNumber are different for each route. In each route r, after the removal, all POIs following the removed ones are shifted towards the beginning of the route. Therefore, the variables of each POI in r as well as the cluster members of each cluster are updated accordingly. Specifically, suppose that the nodes $p_{i+1}, p_{i+2}, \ldots p_{i+k}$ are removed from the route $r = (s_r, \ldots, p_i, p_{i+1}, p_{i+2}, \ldots p_{i+k}, p_j, \ldots, t_r)$. Then the variables of p_j are updated as follows:

$$\text{arrive}_j = \text{leave}_i + \text{traveling}_{i,j}(\text{leave}_i)$$

$$\text{wait}_j^i = \max(0, \text{open}_{jr} - \text{arrive}_j)$$

$$\text{start}_j = \text{arrive}_j + \text{wait}_j^i$$

$$\text{leave}_j = \text{arrive}_j + \text{wait}_j^i + \text{visit}_j$$

Note that the value of p_j's maxStart variable remains unchanged. For each POI after p_j, the variables arrive, wait, start and leave are also updated in a similar way with p_j, while maxStart variable remains unchanged. The value of p_i's maxStart variable is updated as follows:

$$\text{maxStart}_i = \min(\text{close}_{ir}, \max\{t : t + \text{traveling}_{i,j}(t) \leq \text{maxStart}_j\} - \text{visit}_i)$$

while all other variables of p_i remain unchanged. For each POI p_l before p_i, only the value of maxStart_l variable is updated using Eq. (8).

Now, consider the case that the removal continues after the terminal location of r i.e., the nodes $p_{i+1}, p_{i+2}, \ldots p_{i+k}$ to removed, are located in r in the following way: $r = (s_r, p_{i+l+1}, \ldots, p_{i+k}, p_j, \ldots, p_i, p_{i+1}, p_{i+2}, \ldots, p_{i+l}, t_r)$. Then, the value of p_i's maxStart variable is set to

$$\text{maxStart}_i = \min(\text{close}_{ir}, \max\{t : t + \text{traveling}_{i,t_r}(t) \leq \text{et}_r\} - \text{visit}_i)$$

while the values of the maxStart variable of the other remaining POIs in r are updated using equation (8). The other variables of p_j are updated as follows:

$$\text{arrive}_j = \text{leave}_{s_r} + \text{traveling}_{s_r,j}(\text{st}_r)$$

$$\text{wait}_j^{s_r} = \max(0, \text{open}_{jr} - \text{arrive}_j)$$

$$\text{start}_j = \text{arrive}_j + \text{wait}_j^{s_r}$$

$$\text{leave}_j = \text{arrive}_j + \text{wait}_j^{s_r} + \text{visit}_j$$

while for each POI after p_j, the variables arrive, wait, start and leave are also updated similar to p_j. In the sequel the pseudocodes of the **Shake** (Algorithm 5), and the **TD_CSCR** (Algorithm 6) are given.

Algorithm 5. TD_CSCR_Shake(removeNumber, startNumber).

```
for each route r in the solution do ▷ route's starting and ending
    locations are excluded
  if startNumber = 0 or startNumber = r.size then
    startNumber ← 1
  end if
  index ← startNumber mod r.size
```

removeNumber ← removeNumber mod *r*.size
for removeNumber times **do**
if index = *r*.size **then**
index ← 1
end if
remove POI at position index from route ▷ first POI is at position 1
index ← index + 1
end for
update the variables of each POI remaining in *r* and the set of cluster members for each cluster
end for

Algorithm 6. **TD_CSCR**(numberOfClusters, maxIterations).

run the global *k*−means algorithm with *k* = numberOfClusters
construct the list listOfClusterSets
while listOfClusterSets is not empty **do**
remove all POIs visited in the currentSolution
theClusterSetIdToInsert ← listOfClusterSets.pop
RouteInitPhase(theClusterSetIdToInsert)
startNumber ← 1; removeNumber ← 1; notImproved ← 0
while notImproved < maxIterations **do**
repeat
TD_CSCR_Insert
With probability probReplace apply **Replace** step
until no insertion could be performed in the Insert step
repeat
Replace
until no replace could be performed in the Replace step
if currentSolution.profit > bestSolution.profit **then**
bestSolution ← currentSolution; removeNumber ← 1;
notImproved ← 0
else increase notImproved by 1
end if
if
removeNumber > $\min\left\{\frac{\text{currentSolution.sizeOfLargestTour}}{2}, \frac{N}{3k}\right\}$ **then**
removeNumber ← 1
end if
Shake(removeNumber,startNumber)
increase startNumber by removeNumber
increase removeNumber by 1
if
startNumber ≥ currentSolution.sizeOfSmallestTour **then**
decrease startNumber by currentSolution.
sizeOfSmallestTour
end if
end while
end while
return bestSolution

3.3. The Time Dependent Slack CSCRoutes (TD_SℓCSCR) algorithm

The **TD_SℓCSCR** algorithm modifies the insertion step of **TD_CSCR** i.e., it takes an alternative approach for determining the POI p_k that will be selected for insertion in a route *r*. Specifically, while TD_CSCR algorithm's criterion for selecting the inserted POI in a route is based on the insertion cost, TD_SℓCSCR involves a more global criterion taking into consideration the effect of this insertion in the whole route.

The TD_SℓCSCR uses an additional variable slack_i (see Fig. 5) defined for each node p_i in a tourist route *r* as follows:

$$\text{slack}_i = \text{maxStart}_i - \text{arrive}_i \tag{9}$$

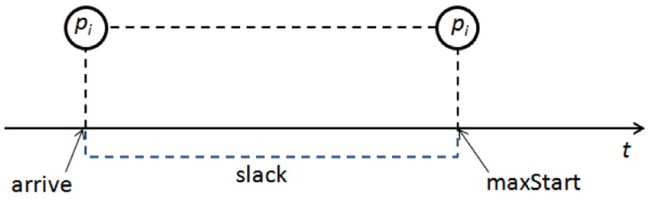

Fig. 5. Illustration of slack's duration for POI p_i.

Note that if the value of slack_i is close to 0, it is highly unlikely the insertion of a new POI between $p_{\text{prev}(i)}$ and p_i to be feasible.

Let $p_1, p_2, \ldots, p_{n'}$ be the successive POIs of a route *r* with $p_1 = s_r$ and $p_{n'} = t_r$. Let p_k be a candidate POI for insertion between POIs p_i and p_{i+1} of *r*. The insertion of p_k will likely shift further the arrival time at p_j (arrive_j), for $j = i+1, \ldots, n'$. That depends on the waiting time before the visit of each POI and the time dependent travel time between successive nodes along the route. Let arrive_j^k be the new arrival time at POI p_j after the insertion of p_k, for $j = i+1, \ldots, n'$. The above insertion may shift the latest time the visit at p_j can start (maxStart_j) earlier for $j = 1, \ldots, i$. Let maxStart_j^k be the new latest time the visit at p_j can start after the insertion of p_k, for $j = 1, \ldots, i$.

Let also $\text{slack}_j^k = \text{maxStart}_j - \text{arrive}_j^k$, for $j = i+1, \ldots, n'$, and $\text{slack}_j^k = \text{maxStart}_j^k - \text{arrive}_j$, for $j = 1, \ldots, i$, be the corresponding values of the "slack" variables. We define the quantity A_k^i as follows:

$$A_k^i = \frac{\sum_{j=1}^{i} \text{slack}_j^k + \text{slack}_k + \sum_{j=i+1}^{n'} \text{slack}_j^k}{n'+1}$$

Note that a large value of A_k^i implies that even after the insertion of p_k, there are many possibilities left for inserting new POIs along each leg of trip (that is, prior and after visiting p_k).

Then for each POI p_k, the maximum possible A_k^i is determined, i.e. the best possible insert position. Let the maximum value A_k^i over all possible insert positions be A_k. Then, in order to determine the POI that will be selected for insertion, the slackWeight for each POI p_k is calculated as

$$\text{slackWeight}_k = \text{profit}_k^2 \cdot A_k \cdot \text{rand}$$

where rand is a number randomly generated in a specified interval, in a similar way with TD_CSCR. The POI with the highest slackWeight is inserted.

The main idea of the above derivations is that for each POI p_k and for each possible insert position (node p_i) within a route *r* we need to calculate A_k^i which involves the updated values of the maxStart and arrive variables for all POIs along *r*. This involves a global rather than a local decision perspective regarding possible insertion positions along the whole route. In order to develop a fast heuristic, a quick calculation of A_k^i is necessary. We may have a quick calculation of a good approximation of A_k^i, by making two assumptions, fairly reasonable in practice. The first one is that the time windows at the POIs are fairly long spanning the most part of the day and therefore the waiting time (wait_j) before visiting each POI p_j ($j = 1 \ldots n'$) is typically zero. This clearly holds for most tourist sites. We also assume that $\text{traveling}_{jj+1}(\text{leave}_j) \approx \text{traveling}_{jj+1}(\text{leave}_j^k)$, $j = i+1, \ldots n'$, where leave_j^k is the new leave time of all nodes following the newly inserted node p_k along the route. The rational behind this approximation is that the additional delay caused by the new detour for visiting node p_k is expected to be relatively short and so a tourist arrives and then departs from each POI subsequent to the insertion point during the same part of the day as that before the insertion. As a result, the transit service frequencies remain unchanged and so the traveling time between two successive nodes on the route can be considered the same as it was before the insertion. Note that for

a particular leg of the route, walking may be faster than taking a transport means. Due to the unchanged frequencies of transit services, we also assume that walking is still beneficial if it was before the insertion and so the traveling time remains the unaffected in that case. Also, although $\text{traveling}_{j,j+1}$ depends on the departure delay at each POI for getting the next service as well, we have ignored this factor. Considering that the POIs along a route are selected by the proposed heuristics according to the preferences of a random user who does not faithfully adhere to the estimated POIs visiting time, in practice, at each leg of the route after the insertion point,the departure delay may as well be increased or decreased. For getting a handy approximation of A_k^i, we now assume that these delay fluctuations are canceling out when summed along the route and thus since the ratio A_k^i is going to be calculated incrementally from expressions concerning the route before that insertion, we can disregard these delays.

Thus, we can consider that

$$\text{arrive}_j^k - \text{arrive}_j \approx \text{shift}_k^{i(i+1)}, \quad j = i+1 \ldots n'.$$

and for reasons similar to those mentioned previously, we also consider that

$$\text{maxStart}_j - \text{maxStart}_j^k \approx \text{maxStart}_i - \text{maxStart}_i^k, \quad j = 1 \ldots i$$

In that case, if $L\_\text{slack}_i = \sum_{j=1}^{i} \text{slack}_j$ and $R\_\text{slack}_i = \sum_{j=i+1}^{n'} \text{slack}_j$ (that is the sum of slack parameters for the part of the trip from POI p_1 up to POI p_i and p_{i+1} up to POI p_n, respectively), as has been estimated in previous global iteration, the new A_k^i for inserting POI p_k will be

$$A_k^i = \frac{L\_\text{slack}_i + i \cdot (\text{maxStart}_i^k - \text{maxStart}_i) + \text{slack}_k + R\_\text{slack}_i - (n'-i) \cdot \text{shift}_k^{i(i+1)}}{n'+1}$$

and since $L\_\text{slack}_i + R\_\text{slack}_i = \sum_{j=1}^{i} \text{slack}_j + \sum_{j=i+1}^{n'} \text{slack}_j = \sum_{j=1}^{n'} \text{slack}_j$ is equal to the sum of slacks of route r, then by storing this sum in a route's variable (sumSlack(r)) and updating this variable after every insertion, replace and shake step, the quantity A_k^i is calculated as

$$A_k^i = \frac{\text{sumSlack}(r) + i \cdot (\text{maxStart}_i^k - \text{maxStart}_i) + \text{slack}_k - (n'-i) \cdot \text{shift}_k^{i(i+1)}}{n'+1}$$

3.4. The Average Travel Times CSCRoutes (AvgCSCR) algorithm

In this subsection we discuss the AvgCSCR algorithm. Similar to the approach proposed by Garcia et al. [9], AvgCSCR is based on average travel times to handle time dependent travel costs among locations and integrate public transportation. For each pair of locations $u, v \in V$ the average travel cost ($\text{avTravel}_{u,v}$) is precalculated using the time dependent traveling costs with time steps of 1 min ($24 \times 60 = 1440$ time steps per day).

$$\text{avTravel}_{u,v} = \frac{\sum_{r=1}^{7} \sum_{t=0}^{1439} \text{traveling}_{u,v}^r(t)}{7 \times 1440}$$

where r represents the day of the week, and $\text{traveling}_{u,v}^r(t)$ is the traveling cost from u to v at time t on the rth day of the week. Then, for each POI p_i in a route r, the values of variables wait_i, start_i, leave_i and arrive_i are determined using the average travel costs among locations, while the value of maxStart_i is calculated as follows:

$$\text{maxStart}_i = \min(\text{close}_{ir}, \text{maxStart}_{\text{next}(i)} - \text{avTravel}_{i,\text{next}(i)} - \text{visit}_i)$$

Note that once the average travel times are available, the problem can be solved using a TOPTW algorithm, thereby removing time dependency. AvgCSCR algorithm invokes the CSCRoutes TOPTW algorithm modified as follows: (a) a randomization factor is added in the definition of the variable ratio of the insertion step of the CSCRoutes algorithm to add diversification and make the search of the solution space more effective in a way similar to the rand factor of the **TD_CSCR_Insert** algorithm; (b) a replace step is added similar to the replace step of the **TD_CSCR** algorithm. Certainly, the routes created by this modified CSCRoutes algorithm will not take into account the real time dependent travel times between successive POIs. For this reason, AvgCSCR applies the following update procedure upon the TOPTW solution, to update the travel costs appropriately:

1. For each route $r = p_1, p_2, \ldots, p_l$, starting from the pair (p_1, p_2) and for each following pair (p_i, p_{i+1}) in r, $i = 2, \ldots, l-1$, the time dependent travel time $\text{traveling}_{i,i+1}^r(\text{leave}_i)$ is calculated using the set of non-dominated pairs $S_{p_i p_{i+1}}$.
2. If the time dependent travel time from p_i to p_j is shorter than the average one, then the visit at p_j starts earlier. In the opposite case, the visit at p_j (and, most probably, at some nodes following p_j) starts later. In both cases the variables of each POI in r are updated appropriately. Note that the above steps may violate the feasibility of one or more visits along a route r; in such case, the whole route r becomes infeasible.
3. In the case that one or more routes are infeasible, the following repair step is applied: While a route r is infeasible, the first, according to the visiting order, POI p_k in r with starting time (calculated based on the time dependent travel costs) greater than the starting time (calculated based on the average travel cost) is removed from r; the POIs following p_k in r are time shifted backwards and their arrival times are then updated. If p_k coincides with the end of the route, then the previous POI is removed.
4. At this point of the procedure, all routes in the solution are feasible, but there might exist "gaps" between POIs, allowing possible insertions of new POIs along the routes. Since the routes are almost "full", it seems that a good criterion for an insertion is to insert the highest profit POI in a position with the least shift (calculated based on the time dependent travel times).Thus, the last step of the procedure is the following: Sort the POIs that do not belong to the routes of the solution in descending order of profit. Let L be the sorted list of POIs. Starting from the highest profit POI p_i and until the list L is empty do the following: if there exists one or more feasible insert position for p_i find one with the lowest shift over all routes and insert p_i; delete p_i from L and repeat.

4. Handling arbitrary start/terminal locations in the Tourist Trip Design Problem

By solving the TTDP we expect to derive k routes with a specified allowed length, that maximize the overall collected profit. Each route may start and end at the tourist's accommodation location, or alternatively, at different user-defined starting and ending locations. TTDP may be formulated and solved as a TDTOPTW, where the POIs as well as the route starting and ending locations are formulated as nodes of the graph G (see Section 3). In the sequel, we consider also the case where the starting and the ending locations of a route may be any location in the destination city, i.e., they are both determined at runtime. This is in accordance with the typical envisaged usage scenario, whereby the TTDP solver will be queried by a mobile client; the tourist's starting location will be typically fixed to his current position and the ending location will be also defined arbitrarily by the user at query time. Clearly, the formulation of the TDTOPTW problem using precalculated travel costs among a fixed set of predefined locations/nodes (e.g. POIs and hotels) cannot support the above described dynamic usage scenario. Therefore, the dynamic setting of the TTDP cannot be solved by the TDTOPTW algorithms

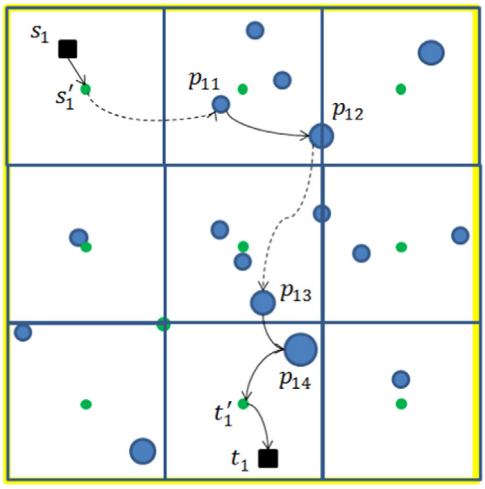

Fig. 6. Illustration of a solution to the TTDP. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

presented in Section 3, and we need to further elaborate on an approach for its solution.

In the sequel we present an algorithm for handling the previously described TTDP scenario. The algorithm comprises a preprocessing phase and an on-line phase. The preprocessing phase comprises the following steps: first the global k-means clustering algorithm is applied on the set of POIs of the destination city and a set of clusters of POIs is constructed. Then the city is partitioned into small square regions (e.g. 500 m × 500 m), covering the whole geographical area where POIs are located in. Within each region R_i a central location is chosen as the location that represents R_i, called region representative rep_i. Consider the complete directed graph $G=(V,E)$, where V consists of all the POIs and all region representatives. Then for each pair of locations (i,j) in V, the set S_{ij} of non-dominated pairs of departure and travel times is calculated (see Section 3.1). Finally, for each representative rep_i of a region i, consider that rep_i belongs to the nearest cluster based on the geometric distance of the mean of the POIs (i.e. centroid) of the cluster.

In the on-line phase of the algorithm, we assume that we are given a set of pairs (s_i, t_i), $i=1,\ldots,k$, denoting the starting and terminal locations of the ith route, and a set of pairs $(\text{st}_i, \text{et}_i)$, $i=1,\ldots,k$, denoting the starting and ending times of the ith route. Then, the on-line phase of the algorithm proceeds as follows:

1. For each route r_i, $i=1,\ldots,m$, (i) find the representatives s_i' and t_i' of the regions where s_i and t_i belong to. (ii) Compute the walking distances $d_s=\text{dist}(s_i,s_i')$ and $d_t=\text{dist}(t_i,t_i')$ and the walking travel times t_{d_s} and t_{d_t} respectively. Note that due to the small size of the regions, it is fairly fast to compute the walking distances and also walking is highly likely to be faster than public transportation. (iii) Set $\text{st}_i'=\text{st}_i+t_{d_s}$ and $\text{et}_i'=\text{et}_i-t_{d_t}$
2. Execute the TD_CSCR algorithm with input the new attributes s_i', t_i', st_i', et_i', for $i=1,\ldots,k$.
3. For each route $r_i=(s_i', b_i=p_{i1}, p_{i2}, \ldots, p_{ik}=l_i, t_i')$ obtained by TD_CSCR, replace s_i' by s_i, st_i' by st_i, t_i' by t_i and et_i' by et_i. In the case that the actual travel time from s_i to b_i is shorter than the walking time from s_i to s_i' plus the traveling time from s_i' to b_i, the visit to b_i may start earlier. Therefore, the calculation of the actual travel time from s_i to b_i has to be performed followed by an update of the arrival, wait, start and leave variables of all nodes apart from the arrival at t_i. Also, the actual travel time from l_i to t_i may be shorter than the traveling time from l_i to t_i' plus the walking time from t_i' to t_i. Then an appropriate subset of $S_{l_i t_i}$ has to be calculated containing the non-dominated pairs $(\text{dep}_j^{l_i t_i}, \text{trav}_j^{l_i t_i})$ with $\text{dep}_j^{l_i t_i} \geq \text{leave}_{l_i}$ and $\text{dep}_j^{l_i t_i}+\text{trav}_j^{l_i t_i} \leq \text{et}_i$. Then, the arrival time at t_i is calculated using $S_{l_i t_i}$ and the maxStart variables of each POI in r_i are appropriately updated.
4. Note that after step 3, time "gaps" may appear between POIs along r_i. To fill these gaps and improve solution's quality, another step is applied as follows: Sort the POIs that do not belong to any route r_i, $i=1,\ldots,k$, in descending order of profit. Let L be the sorted list of POIs. Starting from the highest profit POI u_j and until the list L is empty do the following: if there exists one or more feasible insert position for u_j in any route r_i, $i=1,\ldots,k$ between POIs b_i and l_i find the one with the lowest shift over all routes and insert u_j; delete u_j from L and repeat.

The pseudo-code of the algorithm for solving the TTDP (handling arbitrary start/terminal locations) follows (Algorithm 7).

Algorithm 7. TTDP algorithm.

Preprocessing Phase
Cluster the POIs using global k-means
Partition the city into square regions. For each region R_i, choose a representative rep_i. Consider as locations the representatives of the regions and the POIs
For each region representative rep_i consider that rep_i belongs to the nearest cluster
Calculate the time dependent travel times between all locations
On−line Phase
for each pair s_i, t_i **do**
  Find the representatives of s_i and t_i, s_i' and t_i', respectively.
  Compute the walking distances $d_s=d(s_i,s_i')$ and $d_t=d(t_i,t_i')$
  set $\text{st}_i'=\text{st}_i+t_{d_s}$
  set $\text{et}_i'=\text{et}_i-t_{d_t}$
end for
Execute the TD_CSCR with the new attributes $(s_i', t_i', \text{st}_i', \text{et}_i')$, $i=1,\ldots,k$
for each route $r_i, r_i=(s_i', b_i=p_{i1}, p_{i2}, \ldots, p_{ik}=l_i, t_i')$ obtained by TD_CSCR **do**
  Replace s_i' by s_i, st_i' by st_i, t_i' by t_i and et_i' by et_i
  Compute the actual travel time from s_i to b_i
  Update the arrival, wait, start and leave times of each node in r_i except for t_i
  Calculate the appropriate subset of $S_{l_i t_i}$
  Update the arrival time of t_i and the maxStart value of all nodes in r_i
end for
Sort the POIs that do not belong to any route r_i, $i=1,\ldots,k$, in descending order of profit; let $u_1,\ldots,u_m$ be the sorted list
for $j=1$ to m **do**
  if there exists one (or more) feasible insert position for u_j in any route r_i, $i=1,\ldots,k$ between b_i and l_i **then**
    Find the position with the lowest shift over all routes and insert u_j
  end if
end for

A typical solution to the TTDP is illustrated in Fig. 6. The tourist destination area is partitioned in nine square regions and a central location (representative) is calculated for each region (indicated by green circles). The start/end locations of the route (s_1 and t_1, respectively) are determined at the user query time and are indicated by the black squares. The representatives s_1' (of the area where s_1 belongs to) and t_1' (of the area where t_1 belongs to) are visited in the beginning and in the end of the trip, respectively.

POIs p_{11}, p_{12}, p_{13} and p_{14} are visited in between. Solid and dashed lines denote walking and transit transfers, respectively.

5. Experimental results

5.1. Test instances

While several datasets exist for testing (T)OP(TW) problems, this is not the case for their time-dependent counterparts. To some extent, this is because only a limited body of literature focuses on the time dependent variants of OP; most importantly though, it is due to the difficulty in producing realistic synthetic multimodal timetabled data (respecting the FIFO property and the triangular inequality, among others). Hence, relevant algorithmic solutions should unavoidably be tested upon real transit network data (for instance, Garcia et al. [9] used timetabled data of the San Sebastian bus network, provided by the local transportation authority), to validate their solutions. Fortunately, the wider adoption of the GTFS[1] (General Transit Feed Specification) standard, used by major transportation authorities worldwide to describe and publish their timetabled data, has made access to such data easier than in the past.

In our experiments, we have used the GTFS data of the transit network deployed on the metropolitan area of Athens, Greece, provided by the OASA (Athens Urban Transport Organization). The network comprises 3 subway lines, 3 tram lines and 287 bus lines, 7825 transit stops, 26 192 trips and 1 003 188 daily departure events; in the time-dependent route model graph [20] this results in 29 055 vertices and 63 424 arcs. The walking network consists of 287 003 vertices and 685 850 arcs. For our purposes, we require to know pairwise fastest routes between POIs, for all departure times of the day. Using the method of Dibbelt et al. [4], we compute offline pairwise full (24 h range) multimodal time-dependent travel time profiles. Namely, for each pair of POIs we compute S_{uv}, which contains all the non-dominated pairs $(\text{dep}_i^{uv}, \text{trav}_i^{uv})$, $i = 1, 2, \ldots, |S_{uv}|$, in ascending order of dep_i^{uv}, where dep_i^{uv} is a departure time and trav_i^{uv} is the corresponding travel time of a service of public transport connecting u and v. We also maintain the walking time among u and v, provided that this is shorter than using the transit network at any departure time within the day (otherwise, walking time is set to infinite). The overall shortest time dependent travel time information is pre-calculated and stored in a three-dimensional array of size $N \times N \times 1440$, where N is the number of specified locations/POIs and $1440 (= 24 \times 60)$ the time steps/minutes per day. This memory structure (of size 3.5 GB in our implementation) ensures instant access to time dependent travel times, given a specified pair of POIs (u, v), upon receiving a user query.

The POIs dataset used in our experiments features 113 sites (museums and art galleries, archaeological sites, monuments and landmarks, streets and squares, neighborhoods, churches and religious heritage, parks) mostly situated around Athens downtown and Piraeus areas (see Fig. 7a). The POIs have been compiled from various tourist portals[2] and web services offering open APIs.[3] Profits have been set in a 1–100 scale and visiting times vary from 1 min (e.g. for some outdoor statues) to 2 h (e.g. for some do-not-miss museums and wide-area archaeological sites). The POIs have been grouped in $\lfloor N/10 \rfloor = 11$ disjoint clusters.

The above described POIs dataset has been used to generate 20 different 'topologies'. The real POIs coordinates have been maintained in all cases (in order to maintain the utility of the realistic time dependent travel times calculated offline); however, their respective profits, visiting times and opening hours (i.e. time windows) have been randomized. The incentive of the randomization has been to ensure a fair validation of the evaluated algorithms mitigating the potential bias of individual topologies. In particular, on each topology we consider three equal subsets of POIs: (a) admission-free POIs (e.g. squares, parks, statuses, etc., with 24×7 opening hours) with relatively low profits and short visiting times, (b) do-no-miss POIs (major museums, archaeological sites and neighborhoods), with large profit values and long visiting times, daily open 9am–5pm, (c) relatively less-important visitable sites, open 9am–5pm on working days and closed on weekends with moderate profit values and visiting times.

Our algorithms have been tested using 100 different 'user preference' inputs, each applied to all the 20 abovementioned topologies. Each 'preference' input is associated with a different start/end location, corresponding to a potential accommodation (hotel) option. In particular, we have used a set of 100 hotels scattered around the city, but mostly situated nearby POIs (see Fig. 7b). Furthermore, for each 'preference' input (a) a POI is disregarded on all routes with a 10% probability (this 'simulates' preferences provided by real visitors, such as no interest on religious sites, which enables the algorithms to disregard a subset of available POIs), and (b) each of the remaining POIs is removed from a specific route with a 10% probability (this caters for the possibility of unsuitable weather conditions; for instance, a TTDP solver should disqualify a visit to an open-air POI in a rainy day.[4]) The total time budget available for sightseeing in daily basis (B_r) has been set to 5 h (10:00–15:00) in all experiments.

All test instances-related files are accessible from: http://dgavalas.ct.aegean.gr/public/tdtoptw_instances/index.html. Further, a prototyped web-based TTDP solver is accessible from: http://ecompass.aegean.gr/.

5.2. Results

We have implemented the following five algorithms: (a) TD_CSCR (see Section 3.2), (b) TD_SℓCSCR (see Section 3.3), (c) Time Dependent ILS (TD_ILS – in effect, this is an extension of the standard ILS TOPTW algorithm [27], wherein we take into account time dependent, rather than constant, travel times in the insertion of nodes into routes), (d) AvgCSCR (see Section 3.4), and (e) Average ILS (AvgILS).

The AvgILS refers to the average travel time approach proposed by Garcia et al. [9], wherein the standard ILS algorithm [27] is used to construct routes based on pre-computed average travel times. AvgILS exercises a repair procedure, introducing the real travel times between the POIs of the final TOPTW solution. If this causes a visit to become infeasible, the latter is removed from the route and the remainder of the route is shifted forward. AvgCSCR employs a similar repair step and then a 'gap filling' step (see Step nos. 3 and 4 in Section 3.4); the latter inserts new POIs into the routes, if feasible, thereby further improving the solution's quality.

Note that in both AvgCSCR and AvgILS, the travel times to move from one location to another are constant (average values of time-dependent transfer times, where both algorithms average 24×60 transfer times). The implementation of AvgCSCR and AvgILS is based on CSCRoutes and ILS, respectively (the two best known TOPTW algorithms for TTDP applications [11]), further implementing repair steps. Due to considering constant travel times, these

[1] https://developers.google.com/transit/gtfs/reference
[2] http://www.tripadvisor.com/, http://index.pois.gr/
[3] https://developers.google.com/places/documentation/

[4] Weather forecast information may be easily retrieved from freely available web services like Yahoo Weather (http://weather.yahoo.com/) and fed into the TTDP solver.

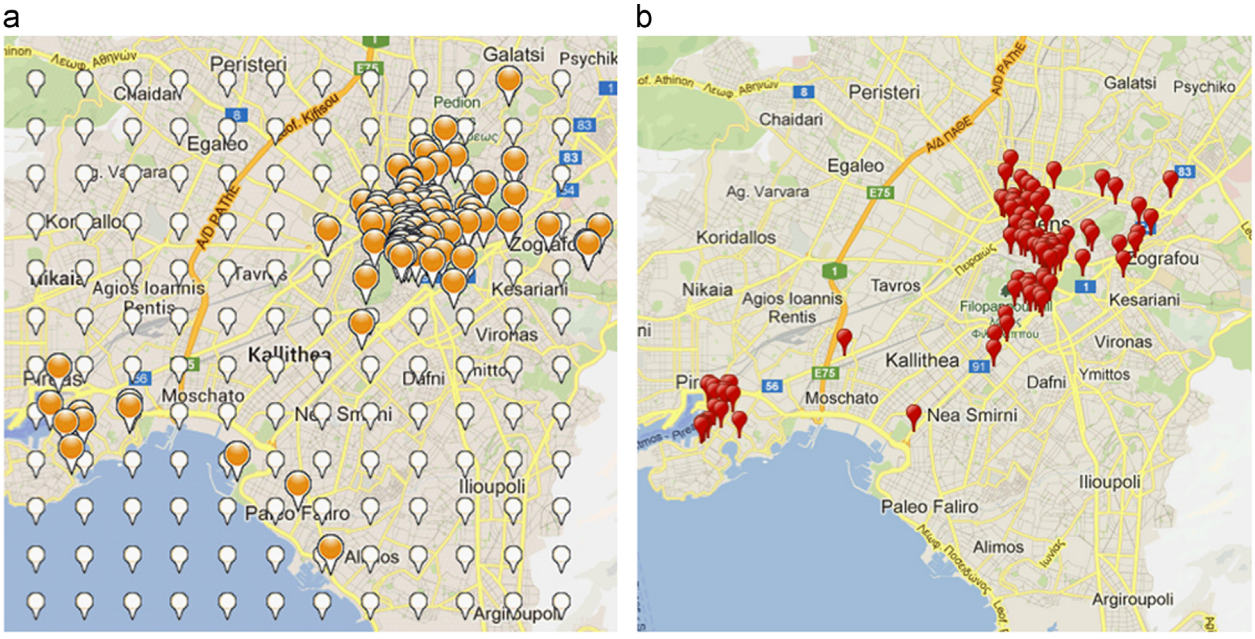

Fig. 7. (a) POIs locations and (b) hotels locations, in the Athens metropolitan area.

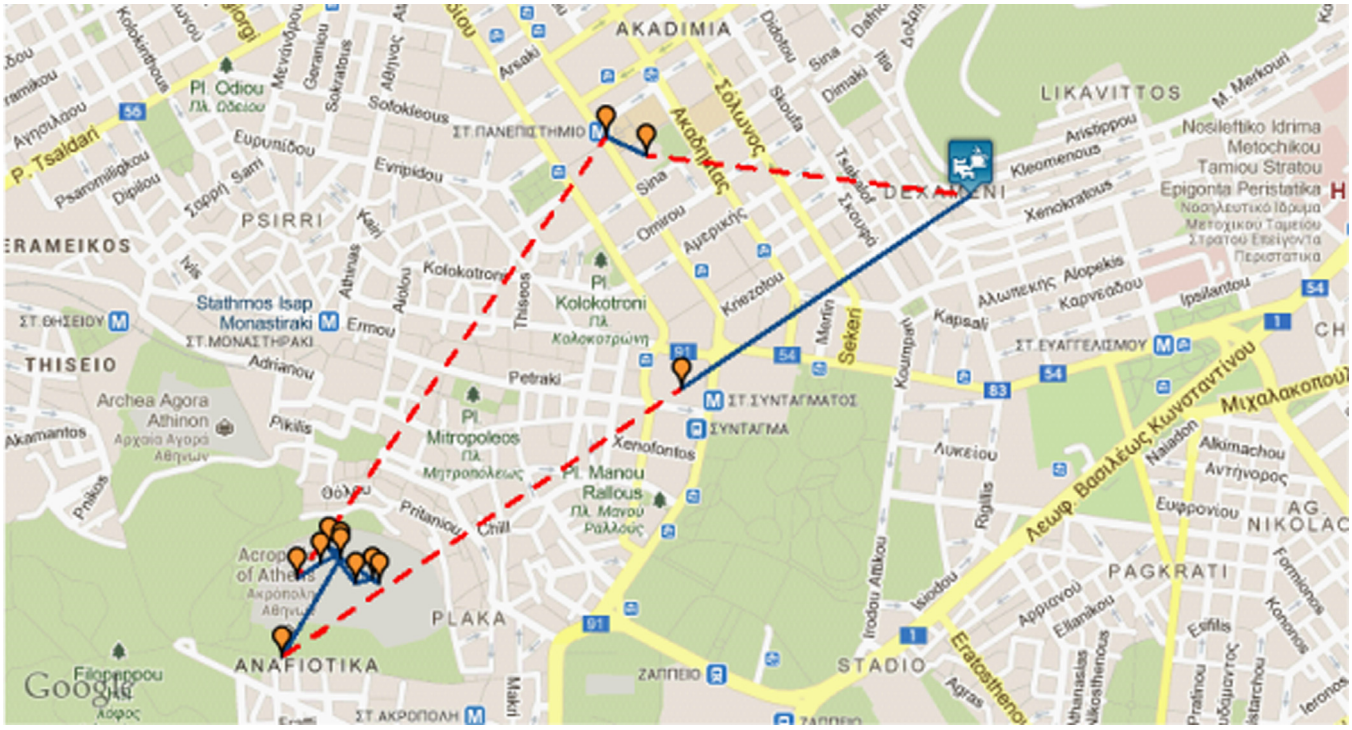

Fig. 8. Illustration of a tourist route starting/ending at a hotel; solid lines indicate walking transfers while dashed lines indicate public transit transfers.

algorithms practically reduce TDTOPTW to TOPTW, wherein AvgCSCR degenerates to CSCRoutes and AvgILS degenerates to ILS.

All algorithms have been employed upon all different topologies and user preferences as described in the previous subsection, deriving k daily personalized routes, $k=1\ldots4$, each for every day of stay at the destination. Since all our proposed algorithms are randomized, they have been executed 5 times on each topology. For all our algorithms, the maximum value of the number randomly generated in the insert step equals to 1.1, while the value of probReplace equals to 0.05. Further, we have considered all 100 available hotels as possible start/end locations for each route. That accounts for $20 \times 5 \times 100 = 10\,000$ execution runs for our algorithms. On the other hand, TD_ILS and AvgILS, being deterministic, have had $20 \times 100 = 2000$ runs. Fig. 8 illustrates a typical tourist route visualized on a map.

Note that all algorithms have been programmed in C++ and executed on a PC Intel Core i5, clocked at 2.80 GHz, with 4GB RAM.

Table 1 demonstrates the experimental results compiled for the five implemented algorithms when employing the standard profit criterion, i.e. we consider as best-found solution the one with the highest aggregate profit. The table includes results averaged over the number of execution runs indicated in the previous subsection (hence, the decimal numbers), for 1–4 daily tourist routes.

Table 1
Experimental results when employing the standard profit criterion.

Algorithm	1 Routes			2 Routes		
	Profit (gap %)	POIs	Time (gap %)	Profit (gap %)	POIs	Time (gap %)
AvgILS	298.53 (100.00)	3.80	**16.32 (100.00)**	561.92 (100.00)	7.87	**60.86 (100.00)**
TD_ILS	326.28 (109.30)	4.14	21.90 (134.22)	641.39 (114.14)	8.98	85.45 (140.40)
AvgCSCR	332.01 (111.21)	4.86	32.05 (196.45)	643.25 (114.47)	9.78	66.10 (108.61)
TDSℓCSCR	**342.06 (114.58)**	5.85	46.97 (287.89)	**657.88 (117.08)**	11.57	109.08 (179.24)
TD_CSCR	337.78 (113.15)	4.94	38.69 (237.15)	654.30 (116.44)	9.87	85.04 (139.74)
	3 Routes			4 Routes		
AvgILS	819.73 (100.00)	12.27	149.91 (100.00)	1078.68 (100.00)	16.53	264.18 (100.00)
TD_ILS	939.84 (114.65)	13.93	217.16 (144.86)	1219.23 (113.03)	18.77	383.27 (145.08)
AvgCSCR	933.73 (113.91)	14.54	**119.87 (79.96)**	1209.30 (112.11)	19.14	**195.64 (74.06)**
TDSℓCSCR	946.50 (115.46)	16.96	198.82 (132.63)	1219.05 (113.01)	22.15	323.92 (122.61)
TD_CSCR	**948.09 (115.66)**	14.60	158.06 (105.44)	**1225.51 (113.61)**	19.26	260.06 (98.44)

Table 2
Comparison of cluster-based vs. non-cluster-based approaches.

Algorithm	1 Routes		2 Routes	
	Profit (gap %)	Time (gap %)	Profit (gap %)	Time (gap %)
AvgCSCR_nonclust	332.43 (100.00)	35.73 (100.00)	644.11 (100.00)	75.28 (100.00)
AvgCSCR	332.01 (99.87)	32.05 (89.70)	643.25 (99.87)	66.10 (87.81)
TDSℓCSCR_nonclust	342.50 (100.00)	66.69 (100.00)	658.54 (100.00)	155.30 (100.00)
TDSℓCSCR	342.06 (99.87)	46.97 (70.43)	657.88 (99.90)	109.08 (70.24)
TD_CSCR_nonclust	338.30 (100.00)	46.25 (100.00)	655.09 (100.00)	102.39 (100.00)
TD_CSCR	337.78 (99.85)	38.69 (83.65)	654.30 (99.88)	85.04 (83.05)
	3 Routes		4 Routes	
AvgCSCR_nonclust	934.48 (100.00)	137.41 (100.00)	1210.56 (100.00)	223.64 (100.00)
AvgCSCR	933.73 (99.92)	119.87 (87.24)	1209.30 (99.90)	195.64 (87.48)
TDSℓCSCR_nonclust	947.49 (100.00)	286.13 (100.00)	1220.19 (100.00)	463.10 (100.00)
TDSℓCSCR	946.50 (99.90)	198.82 (69.49)	1219.05 (99.91)	323.92 (69.95)
TD_CSCR_nonclust	949.09 (100.00)	193.09 (100.00)	1226.55 (100.00)	319.88 (100.00)
TD_CSCR	948.09 (99.89)	158.06 (81.86)	1225.51 (99.92)	260.06 (81.30)

The results shown are the overall collected profit (over all routes), the total number of POIs (over all routes), and the execution time (in msec). High quality solutions are those featuring high aggregate profit, derived in short execution time. Note that the maximization of POI visits has not been an optimization criterion and is not considered to be a quality indicator. In addition to the absolute profit/time values, Table 1 also illustrates normalized performance values assigning a value 100 to AvgILS (which serves as a reference point) and adapting the rest accordingly (this allows illustrating relative performance gaps among tested algorithms). Performance values shown in bold designate the best performing algorithm with respect to each performance parameter.

As a general remark applied to all algorithms, the increase of the overall collected profit with the increase of the number of routes is sublinear, since the average POI profits is higher when considering low numbers of routes (i.e. for short stays, tourists tend to visit the do-not-miss POIs). Furthermore, all implemented algorithms perform remarkably well with respect to execution time since they derive solutions in less than 0.5 s in all cases (considering topologies of 113 POIs and up to four routes).

As expected, the algorithms working with average travel times (i.e. AvgCSCR and AvgILS) execute considerably faster than their time dependent counterparts (up to 45% gap among AvgILS and TD_ILS, up to 91% gap among AvgCSCR and TD_CSCR / TD_SℓCSCR) at the expense of lower quality solutions (up to 14% gap among TD_ILS and AvgILS and up to 3,4% gap among TD_CSCR / TD_SℓCSCR and AvgCSCR). This is because AvgCSCR and AvgILS disregard time dependency on the insertion decision. Hence they derive TOPTW solutions which are only improved to a limited extend by the repair procedure. On the other hand, they use smaller memory structures to hold travel time information (i.e., the required travel times are retrieved more efficiently), hence they execute faster. Nevertheless, we argue that the results obtained by AvgCSCR and AvgILS could be worse when considering either less frequent transit services or timetables where transit frequencies changes considerably along the day (e.g. frequent services in peak hours and infrequent services in off-peak hours) or even when considering tourist visits in off-peak hours (e.g. afternoon to night time budgets). In such scenarios, using the average travel time would not serve as a good approximation.

The AvgCSCR derives solutions of considerably higher quality than the AvgILS ($\geq 11.21\%$). This is mainly due to the advantages of the modified CSCRoutes algorithm over ILS (because of the clustering phase, the randomization introduced, and the replace step) as well as the extra 'gap filling' step applied by AvgCSCR, which considerably improves the quality of its solutions and corrects potential suboptimal node insertion decisions made during the main execution (insertion) phase. As regards the TDTOPTW algorithms, TD_CSCR and TD_SℓCSCR maintain a clear lead over TD_ILS. The performance gap is higher for smaller number of routes ($=1$ or 2) and becomes as high as 5.28% among TD_SℓCSCR and TD_ILS for a single tour. This is because our proposed algorithms prove more effective in escaping local optima, as a result of the randomized insertion step. Apart from the randomization, the replace step, iteratively applied when insert step can no more be applied, improves the solution's quality

Table 3
Improvement from including the *i*-quality metric.

Algorithm	1 Route Profit (gap %)	2 Routes Profit (gap %)	3 Routes Profit (gap %)	4 Routes Profit (gap %)
TDS∉CSCR_no_quality	341.07 (100.00)	657.37 (100.00)	946.08 (100.00)	1218.16 (100.00)
TDS∉CSCR	342.06 (100.29)	657.88 (100.08)	946.50 (100.04)	1219.05 (100.07)
TD_CSCR_no_quality	336.67 (100.00)	654.17 (100.00)	947.00 (100.00)	1223.11 (100.00)
TD_CSCR	337.78 (100.32)	654.30 (100.02)	948.09 (100.12)	1225.51 (100.20)

Table 4
Comparison of results for profit vs. walk motivation insertion criterion.

Algorithm	1 Routes		2 Routes	
	Profit (gap %)	Trans (gap %)	Profit (gap %)	Trans (gap %)
TD_CSCR	337.78 (100.00)	1.92 (100.00)	654.30 (100.00)	3.77 (100.00)
TD_CSCR_walk_motiv	330.47 (97.84)	0.76 (39.67)	642.79 (98.24)	1.83 (48.67)
TDS∉CSCR	342.06 (100.00)	1.80 (100.00)	657.88 (100.00)	3.63 (100.00)
TDS∉CSCR_walk_motiv	334.39 (97.76)	0.58 (32.34)	647.74 (98.46)	1.40 (38.57)
	3 Routes		4 Routes	
TD_CSCR	948.09 (100.00)	5.55 (100.00)	1225.51 (100.00)	7.35 (100.00)
TD_CSCR_walk_motiv	933.44 (98.46)	2.98 (53.61)	1208.80 (98.64)	4.17 (56.72)
TDS∉CSCR	946.50 (100.00)	5.44 (100.00)	1219.05 (100.00)	7.27 (100.00)
TDS∉CSCR_walk_motiv	932.63 (98.53)	2.24 (41.07)	1201.71 (98.58)	3.07 (42.28)

significantly. Furthermore, due to the RouteInit phase our algorithms consider visits to relatively distant clusters. On the other hand, ILS is commonly trapped in isolated areas with few high profit nodes, failing to explore remote areas with considerable numbers of fairly profited candidate nodes. When considering higher number of daily routes ($=3$ or 4), all algorithms tend to include all important (i.e. high profit) POIs, hence, differences on the overall profit are smoothed out. Interestingly, TD_S∉CSCR marginally prevails over TD_CSCR for small number of routes. This is due to the fact that POI insertions increasingly reduce the waiting time of each POI in comparison to the case of higher number of routes. Thus, an accidental local insert step of TD_CSCR will affect other parts along the route since the relatively short waiting time cannot absorb the extra cost resulting by the inserted POI. In contrast, TD_S∉CSCR fares better in this case since its insert step is based on a criterion that takes into account the impact of a new insertion on all parts of the route. In contrast, TD_CSCR marginally prevails for in the case of larger number of routes.

The AvgILS executes a lot faster than the AvgCSCR for small number of routes while the gap reverses for larger number of routes (when the execution time becomes more noticeable to the user). This is explained by the fact that for each solution improvement ILS employs a maximum of 150 extra iterations while CSCRoutes employs additional $400/|\text{listOfClusterSets}| \cdot (k+1)/2 \cdot k$ steps, where k denotes the number of routes. Clearly, this favors ILS for small number of routes wherein the number of solutions improvements is low as opposed to scenarios considering higher number of routes.

Table 2 compares our three proposed algorithms against their non-cluster-based counterparts. The table clearly demonstrates a trade-off among solution quality end execution efficiency. This is mainly attributed to the focal design objective of the cluster-based approaches to prioritize the insertion of successive nodes which belong to the same cluster. This evidently reduces the search space, hence the execution time. This gain comes at the expense of solutions quality as those approaches are not left free to consider candidate nodes irrespective of their cluster association. However, our cluster-based algorithms are most preferable since they achieve time savings up to 30%, while their solutions quality is only compromised up to 0.16%.

Table 3 compares TD_CSCR and TD_S∉CSCR algorithms against their counterparts which do not incorporate the *i*-quality metric of each cluster at their insertion step (recall that the *i*-quality metric favors the insertion of POIs which belong to clusters with several high-profit POIs, not yet included into a route). Although the table reports only marginal improvement in the case of using the *i*-quality metric, this is mostly due to the particular characteristics of the utilized datasets. In particular, the POI clusters of the metropolitan area of Athens are crowded in a relatively restricted geographical area, which allows to easily accessing (by foot or public transit) the most highly profitable POIs one after the other, irrespective of their cluster assignment. This results in deriving almost identical solutions when not incorporating the *i*-quality metric, especially for relatively short time budgets. The effect of the *i*-quality metric is expected to become more evident in scenarios featuring relatively distant clusters and prolonged time budgets.

In the sequel we present experimental results compiled for our TDTOPTW algorithms, using an alternative criterion for selecting the best found solution. Specifically, instead of selecting the solution with the highest aggregate profit over all routes, we select as best found solution the one combining high profit along with a small number of transit transfers occurring along all routes, favoring the insertion of POIs within walking distance from their preceding and following POIs. Namely, we select the solution that maximizes $\text{profit}(2+\frac{1}{\text{transit}+1})$, where profit equals to the aggregate profit over all routes and transit refers to the overall transit transfers occurring along all routes.

The results (see Table 4) indicate a clear tradeoff between profit and number of transit transfers. In particular, since profit is not the sole criterion used for picking the best solution, the overall profit is reduced (up to 2.24%) compared to the results corresponding to the profit criterion. On the other hand, the incorporation of the occurring transit transfers into the criterion for finding best solutions has considerably reduced the overall number of transit transfers along the derived routes (typically, each route features one transit transfer less than in the previous result sets).

Interestingly, higher gap on transit transfers is achieved when applying the walk motivation to TD_S∉CSCR. This is due to the

objective of TD_SℓCSCR to maximize the available time between successive visits for selecting slower transportation modes, such as walking. Thus, when motivating walks, TD_SℓCSCR considerably reduces the transit transfers in favor of walking.

6. Conclusions

We introduced TD_CSCR and TD_SℓCSCR, two new cluster-based heuristics for solving the TDTOPTW. To the best of our knowledge, these are the first TDTOPTW approaches making no assumption on the periodicity of transit services. The main design objectives of the two algorithms are to derive high quality TDTOPTW solutions (maximizing tourist satisfaction), while minimizing the number of transit transfers and executing fast enough to support online web and mobile applications. We have also proposed extensions on our TTDP algorithms to handle arbitrary (i.e. determined at query time) rather than fixed start/end locations for tourist itineraries. Our algorithmic solutions have been tested on realstic datasets compiled from the metropolitan area of Athens, Greece.

With respect to the overall collected profit, TD_CSCR has been shown to perform marginally better for large number of routes, while TD_SℓCSCR prevails for small number of routes. Both our proposed TDTOPTW algorithms clearly outperform the time dependent extension of best known TOPTW algorithm, suitable for online applications. In scenarios comprising very large datasets, AvgCSCR could be the most suitable choice as it efficiently derives solutions of reasonably good quality (this conclusion agrees with that reported in [9], wherein a TDTOPTW algorithm has been evaluated against AvgILS). Nevertheless, its suitability largely depends on the high frequency of public transit services, so that average travel times represent a good guess.

References

[1] Abbaspour RA, Samadzadegan F. Time-dependent personal tour planning and scheduling in metropolises. Expert Syst Appl 2011;38:12439–52.
[2] Bagirov M. Modified global k-means algorithm for minimum sum-of-squares clustering problems. Pattern Recognit 2008;41(10):3192–9.
[3] Chao I-M, Golden BL, Wasil EA. The team orienteering problem. Eur J Oper Res 1996;88(3):464–74.
[4] Dibbelt J, Pajor T, Wagner D. User-constrained multi-modal route planning. In: Proceedings of the 14th meeting on algorithm engineering and experiments (ALENEX'12); 2012. p. 118–29.
[5] Erdogan G, Laporte G. The orienteering problem with variable profits. Networks 2013;61(2):104–16.
[6] Erkut E, Zhang J. The maximum collection problem with time-dependent rewards. Naval Res Logist 1996;43(5):749–63.
[7] Fomin FV, Lingas A. Approximation algorithms for time-dependent orienteering. Inf Process Lett 2002;83(2):57–62.
[8] Gambardella LM, Montemanni R, Weyland D. Coupling ant colony systems with strong local searches. Eur J Oper Res 2012;220(3):831–43.
[9] Garcia A, Vansteenwegen P, Arbelaitz O, Souffriau W, Linaza MT. Integrating public transportation in personalised electronic tourist guides. Comput Oper Re 2013;40(3):758–74.
[10] Gavalas D, Konstantopoulos C, Mastakas K, Pantziou G. A survey on algorithmic approaches for solving tourist trip design problems. J Heuristics 2014;20(3):291–328. http://dx.doi.org/10.1007/s10732-014-9242-5.
[11] Gavalas D, Konstantopoulos C, Mastakas K, Pantziou G, Tasoulas Y. Cluster-based heuristics for the team orienteering problem with time windows. In: Proceedings of 12th international symposium on experimental algorithms (SEA'13); 2013. p. 390–401.
[12] Golden BL, Levy L, Vohra R. The orienteering problem. Naval Res Logist (NRL) 1987;34(3):307–18.
[13] Labadi N, Mansini R, Melechovský J, Wolfler Calvo R. The team orienteering problem with time windows: an lp-based granular variable neighborhood search. Eur J Oper Res 2012;220(1):15–27.
[14] Labadi N, Melechovský J, Wolfler Calvo R. Hybridized evolutionary local search algorithm for the team orienteering problem with time windows. J Heuristics 2011;17:729–53.
[15] Laporte G, Martello S. The selective travelling salesman problem. Discrete Appl Math 1990;26(2–3):193–207.
[16] Likas A, Vlassis N, Verbeek J. The global k-means clustering algorithm. Pattern Recognit 2003;36(2):451–61.
[17] Lin S-W, Yu VF. A simulated annealing heuristic for the team orienteering problem with time windows. Eur J Oper Res 2012;217(1):94–107.
[18] Helena Lourenco, Olivier Martin, Thomas Stützle. Iterated local search. In: Glover F, Kochenberger G. editors, Handbook of metaheuristics, International Series in Operations Research & Management Science, vol. 57, New York: Springer; 2003. pp. 320–53.
[19] Montemanni R, Gambardella LM. An ant colony system for team orienteering problems with time windows. Found Comput Decis Sci 2009;34(4):287–306.
[20] Pyrga E, Schulz F, Wagner D, Zaroliagis C. Efficient models for timetable information in public transportation systems. J Exp Algorithmics 2008 12:2.4:1–2.4:39.
[21] Spieksma FCR. On the approximability of an interval scheduling problem. J Sched 1999;2:215–27.
[22] Tang H, Miller-Hooks E, Tomastik R. Scheduling technicians for planned maintenance of geographically distributed equipment. Transp Res Part E: Logist Transp Rev 2007;43(5):591–609.
[23] Tricoire F, Romauch M, Doerner KF, Hartl RF. Heuristics for the multi-period orienteering problem with multiple time windows. Comput Oper Re 2010;37(2):351–67.
[24] Tsiligirides T. Heuristic methods applied to orienteering. J Oper Res Soc 1984;35(9):797–809.
[25] Vansteenwegen P. Planning in tourism and public transportation—attraction selection by means of a personalised electronic tourist guide and train transfer scheduling [Ph.D. thesis]. Katholieke Universiteit Leuven; 2008.
[26] Vansteenwegen P, Souffriau W, Van Oudheusden D. The orienteering problem: a survey. Eur J Oper Res 2011;209(1):1–10.
[27] Vansteenwegen P, Souffriau W, Vanden Berghe G, Van Oudheusden D. Iterated local search for the team orienteering problem with time windows. Comput Oper Res 2009;36:3281–90.
[28] Vansteenwegen P, Van Oudheusden D. The mobile tourist guide: an or opportunity. Oper Res Insight 2007;20(3):21–7.
[29] Verbeeck C, Sorensen K, Aghezzaf E-H, Vansteenwegen P. A fast solution method for the time-dependent orienteering problem. Eur J Oper Res 2014;236(2):419–32. http://dx.doi.org/10.1016/j.ejor.2013.11.038.
[30] Yu J, Aslam J, Karaman S, Rus D. Optimal tourist problem and anytime planning of trip itineraries, arxiv:1409.8536. 2014.