Theoretical Computer Science 1023 (2025) 114900

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Algorithms for the thief orienteering problem on directed acyclic graphs ☆

Andrew Bloch-Hansen [a], Roberto Solis-Oba [a,*], Daniel R. Page [b,c]

[a] Western University, 1151 Richmond Street, London, N6A 3K7, Ontario, Canada
[b] PageWizard Games, Learning & Entertainment, Manitoba, Canada
[c] New College of Florida, 5800 Bay Shore Road, Sarasota, 34243, FL, United States

ARTICLE INFO

Keywords:
Thief orienteering problem
Knapsack problem
Dynamic programming
Approximation algorithm
Approximation scheme

ABSTRACT

We consider the scenario of routing an agent called a *thief* through a weighted graph $G = (V, E)$ from a start vertex s to an end vertex t. A set I of items each with weight w_i and profit p_i is distributed among $V \setminus \{s, t\}$. The thief, who has a knapsack of capacity W, must follow a simple path from s to t within a given time T while packing in the knapsack a set of items taken from the vertices along the path of total weight at most W and maximum profit. The travel time across an edge depends on the edge length and current knapsack load.

The thief orienteering problem (ThOP) is a generalization of the orienteering problem, the longest path problem, and the 0-1 knapsack problem. We prove that there exists no approximation algorithm for ThOP with constant approximation ratio unless P = NP, and we present a polynomial-time approximation scheme (PTAS) for a relaxed version of ThOP when G is directed and acyclic that produces solutions that use time at most $T(1+\epsilon)$ for any constant $\epsilon > 0$. We also present a fully polynomial-time approximation scheme (FPTAS) for ThOP on arbitrary undirected graphs where the travel time depends only on the lengths of the edges and T is the length of a shortest path from s to t plus a constant K. Finally, we present a FPTAS for a restricted version of the problem where the input graph is a clique.

1. Introduction

The *thief orienteering problem* (ThOP) is defined as follows. Let $G = (V, E)$ be a weighted graph with n vertices, where two vertices $s, t \in V$ are designated the *start* and *end* vertices, and every edge $e = (u, v) \in E$ has a length $d_{u,v} \in \mathbb{Q}^+$. In addition, let there be a set I of items, where each item $i_j \in I$ has a non-negative integer weight w_j and profit p_j. Each vertex $u \in V \setminus \{s, t\}$ stores a subset $S_u \subseteq I$ of items such that $S_u \cap S_v = \emptyset$ for all $u \neq v$ and $\bigcup_{u \in V \setminus \{s,t\}} S_u = I$. There is an agent called a thief that has a knapsack with capacity $W \in \mathbb{Z}^+$ and the goal of the problem is for the thief to travel a simple path from s to t within a given time $T \in \mathbb{Q}^+$ while collecting items in the knapsack taken from the vertices along the path of total weight at most W and maximum total profit.

The amount of time needed to travel between two adjacent vertices u, v depends on the length of the edge connecting them and on the weight of the items in the knapsack when the edge is traveled; specifically, the travel time between adjacent vertices u and

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
E-mail addresses: ablochha@uwo.ca (A. Bloch-Hansen), solis@csd.uwo.ca (R. Solis-Oba), drpage@pagewizardgames.com, dpage@ncf.edu (D.R. Page).

https://doi.org/10.1016/j.tcs.2024.114900
Received 30 November 2023; Received in revised form 9 May 2024; Accepted 3 October 2024

Available online 9 October 2024

v is $d_{u,v}/\mathcal{V}$ where $\mathcal{V} = \mathcal{V}_{\max} - w(\mathcal{V}_{\max} - \mathcal{V}_{\min})/W$, w is the current weight of the items in the knapsack, and $\mathcal{V}_{\min}$ and $\mathcal{V}_{\max}$ are the minimum and maximum velocities of the thief.

ThOP is a generalization of the orienteering problem [9], the longest path problem, and the 0-1 knapsack problem, so it is NP-hard. The problem was first formulated by Santos and Chagas [16] in 2018, and they provided the first two heuristics for ThOP: An iterated local search algorithm and a biased random-key genetic algorithm. In 2020, Faêda and Santos [6] presented a genetic algorithm for ThOP. Chagas and Wagner [4] designed a heuristic using an ant colony algorithm which they later further improved [5].

While ThOP is a relatively new problem, the closely related family of travelling problems, such as the travelling thief problem [3] and some variants of orienteering [17], are well-studied and have applications in areas as diverse as route planning [7,8,12], circuit design [3], and logistics [11], among others.

In 2017, Polyakovskiy and Neumann [14] introduced the packing while travelling problem (PWTP). This is a problem similar to ThOP, but in PWTP the thief must follow a fixed path and the goal is to maximize the difference between the profit of the items collected in the knapsack and the transportation cost. Polyakovskiy and Neumann provided two exact algorithms for PWTP, one based on mixed-integer programming and another on branch-infer-and-bound. In 2019, Roostapour et al. [15] presented three evolutionary algorithms on variations of PWTP. Most recently, Neumann et al. [13] presented an exact dynamic programming algorithm and a fully polynomial-time approximation scheme (FPTAS) for PWTP. Their dynamic programming algorithm, unfortunately, cannot be applied to ThOP because in ThOP we must bound both the total weight of the items and the travelling time of the thief.

To the best of our knowledge, study on ThOP to this date has focused on the design of heuristics and no previous work has presented an approximation algorithm for it.

In this paper we show that ThOP cannot be approximated within a constant factor unless P = NP and we present a dynamic programming-based polynomial-time approximation scheme (PTAS) for a relaxed version of ThOP when the input graphs are directed acyclic graphs (DAGs) and the time used by the thief is at most $T(1+\epsilon)$ for any constant $\epsilon > 0$.

There are several challenges involved in the design of a PTAS for ThOP on DAGs. To achieve polynomial running time the parameters of the problem (weight, profit, and travelling time) need to be rounded to keep the size of the dynamic programming table polynomial. Rounding weights needs to be done carefully because (1) rounding the weights of small weight items with high profit can yield solutions with profit much smaller than the optimum one and (2) rounding the weights of items located far away from the destination vertex can cause large errors in the travelling times.

We solve the first problem through enumeration by ensuring that a constant number of the items with largest profit in an optimum solution belong also to the solution computed by our algorithm. We solve the second problem by using actual item weights and not rounded item weights when computing travelling times, and by allowing the thief slightly more time to travel from s to t.

Variations of our algorithm yield FPTAS on special versions of ThOP on arbitrary undirected graphs, like (i) the case when $\mathcal{V}_{\min} = \mathcal{V}_{\max}$, edge lengths are integer, and T is equal to the length L of a shortest path from s to t plus a constant K, and (ii) the case when $\mathcal{V}_{\min} = \mathcal{V}_{\max}$, edges have unit length, and the input graph is a clique. This latter case is a generalization of the knapsack problem when the items are partitioned into groups and we must select items only from a certain number of these groups.

The rest of the paper is organized in the following way. We begin in Section 2 by proving the inaproximability of ThOP. In Section 3 we present an exact algorithm for ThOP on DAGs using dynamic programming and in Section 4, we transform this algorithm into a PTAS that produces solutions that use time at most $T(1+\epsilon)$. In Section 5 we show our algorithm can be converted to a FPTAS for a restricted version of ThOP on undirected graphs. In Section 6 we present our FPTAS for ThOP on cliques. A preliminary version of this paper appeared in [1], and this extended version includes the complete proofs for the lemmas and theorems that were omitted in [1], including a correction to Lemma 2 and Theorem 3 as our algorithm needs time $T(1+\epsilon)$ and not time T. We also include the algorithm for the case when the input graph is a clique with unit length edges.

2. Inapproximability

We first show that even when the input graph is a clique, each vertex has at most one item, and edge lengths are only 1 or 2, ThOP has no PTAS unless P = NP. To prove this we show that ThOP is a generalization of the rooted orienteering problem.

In the *rooted orienteering problem* (OP) we are given a weighted complete graph $G = (V, E)$ with metric distances $w : E \to \mathbb{Z}^+$, vertex profits $\pi : V \to \mathbb{Z}_{\geq 0}$, root vertex $s' \in V$, and a path length limit $L \in \mathbb{Z}_{\geq 0}$. The goal in this problem is to compute a path starting at s' of total length at most L for which the sum of the profits of the vertices in the path is maximum.

Theorem 1. *There is no PTAS for ThOP unless* P = NP. *This is true even when the input graph is a clique, every vertex has at most one item, and edge lengths are only* 1 *or* 2.

Proof. Blum et al. [2] proved that there is no PTAS for OP even if the input graph is a clique, edge lengths are 1 or 2, and the root vertex has zero profit. Consider an instance of this restricted version of OP. We create an instance of ThOP on $\{1,2\}$-metrics, by performing the following steps:

1. Set s to be the root vertex and add an ending vertex t to the input graph $G = (V, E)$ of OP.
2. Add an edge from every vertex of G to t of length 1.
3. Assign velocities $v_{max} = v_{min} = 1$, knapsack capacity $W = |V|$, and time limit $T = L + 1$.
4. Finally, for each vertex $v \in V$ where profit $\pi(v) > 0$, add an item i to I with weight $w_i = 1$ and profit $p_i = \pi(v)$, and place i as the only item in the subset I_v of items associated with vertex v.

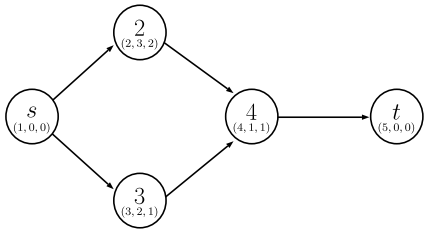

Fig. 1. An example of a DAG that has two possible paths from s to t. In each vertex u there is a triplet (index, profit, weight) for each item in I_u.

A solution for this instance of ThOP yields a solution of the same value for the corresponding instance of OP, as a solution for ThOP includes all items stored at the vertices of the selected path. Hence, a PTAS for ThOP would be a PTAS for OP. □

To build to our main result in this section, we consider the optimization version of the classic *longest path problem*. In this problem, we are given an arbitrary simple connected unweighted graph $G = (V, E)$, and the goal is to compute a path P in G with a maximum number of edges. The longest path problem has no approximation algorithm with constant approximation ratio, unless $\mathsf{P} = \mathsf{NP}$ [10]. Furthermore, for any $\epsilon > 0$, there is no algorithm for the longest path problem with approximation ratio $O(\log^{1-\epsilon} n)$, unless $\mathsf{NP} \in \mathsf{DTIME}(2^{O(\log^{1-\epsilon} n)})$.

Theorem 2. *There is no approximation algorithm for ThOP with constant approximation ratio, unless* $\mathsf{P} = \mathsf{NP}$. *Furthermore, for any* $\epsilon > 0$, *there is no approximation algorithm for ThOP with approximation ratio* $2^{O(\log^{1-\epsilon} n)}$ *unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(2^{O(\log^{1-\epsilon} n)})$. *These hardness results hold even if the input graph has bounded degree, all edges have unit length, and each vertex stores only one item of unit weight and profit.*

Proof. The longest path problem [10] is a special case of ThOP, where s and t are the endpoints of a longest path in the input graph $G = (V, E)$, every edge has length 1, every vertex $u \in V \setminus \{s, t\}$ stores one item of weight 1 and profit 1, the capacity of the knapsack is $W = |V| - 2$, the bound on the time is $T = |V| - 1$, and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$. Since there are $O(|V|^2)$ possible choices for the endpoints of a longest path in G then the inapproximability properties of the longest path problem [10] apply also to ThOP. □

The fractional version of ThOP allows the thief to select a fraction of each item.

Corollary 1. *The fractional version of ThOP cannot be approximated in polynomial time within any constant factor unless* $\mathsf{P} = \mathsf{NP}$.

Proof. Consider the same reduction as in the proof of Theorem 2. Note than an optimal fractional solution must collect the whole items stored in the vertices of an optimal path. □

3. Algorithm for thief orienteering on DAGs

To simplify the description of our algorithms, for each vertex u that does not store any items we add to I a new dummy item of weight 0 and profit 0 and store it in u; hence, every vertex stores at least one item (see Fig. 1).

We can assume that the minimum and maximum velocities $\mathcal{V}_{\min}$ and $\mathcal{V}_{\max}$ are δ and 1, respectively, where $\delta \in \mathbb{Q}^+$ and $\delta \leq 1$. Then, the *travel time* from vertex u to vertex v when the knapsack's total weight is w just prior to leaving vertex u is equal to $d_{u,v}/\eta$, where $\eta = 1 - \frac{(1-\delta)w}{W}$.

Since DAGs have no cycles, every path from s to t is a simple path. We index the vertices of the graph using a topological ordering. We delete from G all vertices that are unreachable from s and all vertices that cannot reach t. For a vertex u, let $u.index$ be the index of the vertex as determined by the topological ordering.

Note that s and t have the lowest and highest indices, respectively. Additionally, observe that the indices of the vertices encountered along a simple path from s to any vertex u appear in increasing order. We index the items stored in the vertices so that item indices are unique and items in vertex u have smaller indices than items in vertex v if $u.index < v.index$. Let the items stored in the vertices of the input graph G be $i_1, i_2, ..., i_{|I|}$.

We define the *parents* of a vertex u to be the vertices v such that (v, u) is a directed edge of G. Our algorithm for ThOP on DAGs is shown in Algorithm 1.

3.1. Profit table

Let S be a subset of items. In the sequel, we define the *travel time* of S to a vertex u as the minimum time needed by the thief to collect all items in S while travelling along a simple path p_{su} from s to u that includes all the vertices storing the items in S. Path p_{su} is called a *fastest path* of S to u. Additionally, we define the *total travel time* of S as the travel time of S to t and the *total travel time of S through u* as the travel time of S to t using a simple path that includes u.

Algorithm 1 ThOPDAG($G = (V, E), W, T, s, t, I, \delta$).

Input: DAG G, knapsack capacity W, time limit T, start vertex s, end vertex t, vertex item assignments I, and minimum velocity δ.
Output: An optimum solution for ThOP.
Delete from G vertices unreachable from s and vertices that cannot reach t.
Compute a topological ordering for G.
Index V by increasing topological ordering and index items as described above.
Let A be an empty profit table. Set $A[1] = (0, 0, 0, -1)$.
for $i = s.index$ *to* $t.index$ **do**
    Let u be the vertex with index i.
    Call *UpdateProfitTable*(W, T, δ, A, u).
end for
Return *BuildKnapsack*(A, I).

Definition 1. Let $S_z = \{i_1, ..., i_z\}$ for all $z = 1, ..., |I|$, and let vertex u hold item i_z. A subset S of S_z is a feasible subset with respect to u if it has weight $w_S = \sum_{i_j \in S} w_j \leq W$ and total travel time through u at most T.

Our algorithm builds a *profit table* A where each entry $A[j]$, for $j = 1, ..., |I|$, corresponds to the item i_j with index j, and $A[j]$ is a list of tuples $(w, p, time, prev)$. Let u be the vertex that contains item i_j; a tuple $(w, p, time, prev)$ in the list of $A[j]$ indicates that there is a subset S of S_j such that:

- the weight of the items in S is $w \leq W$,
- the profit of the items in S is p,
- the travel time of S to u is $time \leq T$,
- a fastest path of S to u includes a vertex storing i_{prev}. Note that item i_{prev} does not need to be in S.

A tuple $(w, p, time, prev)$ *dominates* a tuple $(w', p', time', prev')$ if $p \geq p'$, $w \leq w'$, and $time \leq time'$. We remove dominated tuples from each list of A such that no tuple in the list of each entry $A[j]$ dominates another tuple in the same list. Therefore, we can assume each list $A[j]$ has the following properties: (i) the tuples are sorted in non-decreasing order of their profits, (ii) there might be multiple tuples with the same profit and these tuples are sorted in non-decreasing order of their weights, and (iii) if several tuples in $A[j]$ have the same profit p and weight w, only the tuple with the smallest value of $time$ is kept in $A[j]$.

3.2. UpdateProfitTable

Algorithm 2 shows how we update the profit table with the items stored in vertex u. The start vertex $s \in V$ has no parents and holds a single item i_1 of weight and profit 0; therefore, we initialize $A[1]$ to store the tuple $(0, 0, 0, -1)$.

Algorithm 2 UpdateProfitTable(W, T, δ, A, u).

Input: Knapsack capacity W, time limit T, minimum velocity δ, profit table A, and vertex u.
Output: The entries of the profit table A corresponding to u's items are updated to represent the subsets of items found along all paths from s to u.
for each item i_j of u **do**
    Let w be the weight and p the profit of i_j.
    if i_j is u's first item **then**
        $A[j] = \emptyset$.
        for each parent v of u **do**
            Let id_v be the index of v's last item.
            for each $(w', p', time', prev') \in A[id_v]$ **do**
                Let $d_{u,v}$ be the distance from v to u.
                Let $\eta = 1 - (1 - \delta)w'/W$.
                Let $travel = \frac{d_{u,v}}{\eta}$.
                if $time' + travel \leq T$ **then**
                    Append $(w', p', time' + travel, id_v)$ to $A[j]$.
                end if
            end for
        end for
    else
        Copy all tuples of $A[j-1]$ into $A[j]$.
        For each tuple $(w', p', time', prev')$ in $A[j]$, $prev' = j - 1$.
    end if
    for each $(w', p', time', prev') \in A[j]$ **do**
        if $w + w' \leq W$ **then**
            Append $(w + w', p + p', time', prev')$ to $A[j]$.
        end if
    end for
    Remove dominated tuples from $A[j]$.
end for

When two (or more) different paths from s to t are routed through some intermediate vertex u, it is vital that subsets of items corresponding to each of the paths are recorded correctly: The entries in the profit table A must represent the item subsets of each path from s to u, but none of the tuples in A should contain information from items stored in vertices from disjoint sections of two different paths.

Every item is stored in a unique vertex; therefore, by including in each tuple $(w, p, time, prev)$ the value $prev$, which indicates that the tuple was built using a tuple from the entry $A[prev]$ corresponding to item i_{prev}, the profit table stores the necessary information to determine which of the parents of a vertex u were used to create the tuples at entries $A[j]$ of the items i_j stored in u.

Observe that travel times need to be computed when creating the tuples for the first item of a vertex u; however, this travel time is the same for the tuples corresponding to the other items stored in u.

3.2.1. Selecting a solution from the profit table

Recall that the vertex with the largest index is t; therefore, t is visited last by Algorithm 1 and its last item $i_{|I|}$ corresponds to the final entry in the profit table A. Therefore, after executing Algorithm 1 the last entry in A contains the list of all dominating tuples whose weights are at most W and total travel times are at most T. Within this list there exists a tuple $(w^*, p^*, time^*, prev^*)$ with maximum profit, and this tuple represents a feasible subset S of $S_{|I|}$ with respect to t with maximum profit. Algorithm 3 shows how to recover S from the information stored in the profit table.

Algorithm 3 BuildKnapsack(A, I).

Input: Profit table A and vertex item assignments I.
Output: The subset of items along a simple path from s to t with the most profit.
$path = \emptyset$, $knapsack = \emptyset$.
Let $j = |I|$, the index of t's last item, and let $u = t$.
Let $(w, p, time, prev)$ be the tuple with maximum profit from $A[j]$.
while $j \geq 0$ **do**
    Let $\eta = 1 - (1 - \delta)w/W$.
    If $u \notin path$, prepend u to $path$.
    Let w_j be the weight of item i_j with index j and p_j be its profit.
    if u is the vertex where the item with index $j - 1$ is located **then**
        if a tuple $(w, p, time, prev')$ is in $A[j - 1]$ **then**
            $(w, p, time, prev) = (w, p, time, prev')$.
        else
            Append the item i_j with index j to $knapsack$.
            $(w, p, time, prev) = (w - w_j, p - p_j, time, prev')$.
        end if
        $j = j - 1$.
    else
        Let u_α be the vertex where item $prev$ is located.
        Let $d_{u_\alpha, u}$ be the distance from u_α to u.
        if a tuple $(w, p, time' = time - \frac{d_{u_\alpha, u}}{\eta}, prev')$ is in $A[prev]$ **then**
            $j = prev$.
            $(w, p, time, prev) = (w, p, time', prev')$.
        else
            Append the item i_j with index j to $knapsack$.
            $j = prev$.
            $(w, p, time, prev) = (w - w_j, p - p_j, time - \frac{d_{u_\alpha, u}}{\eta}, prev')$.
        end if
    end if
end while
return $path$.

Algorithm 3 considers one item at a time, starting with t's last item $i_{|I|}$ and determines whether $i_{|I|}$ belongs in the solution. The next item considered by the algorithm is i_{prev^*}, and so on. Consider an item i_j located at some vertex u. Let entry $A[j]$ contain a tuple $\tau = (w, p, time, prev)$. If item i_{j-1} is also located at u, then a tuple $(w, p, time, prev')$ in $A[j - 1]$ indicates that i_j is not in the (partial) solution corresponding to tuple τ. If $A[j - 1]$ corresponds to an item not located at u, then let the vertex u_α store item i_{prev}; then a tuple $(w, p, time', prev')$ in $A[prev]$ with $time' = time - \frac{d_{u_\alpha, u}}{\eta}$ indicates that i_j was not used to create tuple τ.

If there is no tuple $(w, p, time', prev')$ in $A[j - 1]$ (or $A[prev]$) as described above then i_j belongs in the (partial) solution corresponding to τ.

3.3. Algorithm analysis

Recall that the items are indexed such that for two items with indices h and j where $h < j$, item i_h must belong to a vertex whose index is less than or equal to the index of the vertex containing item i_j.

To prove that our algorithm is correct we must prove that each entry $A[z]$ of the profit table is such that for every feasible subset S of S_z with respect to the vertex holding item i_z the entry $A[z]$ contains either (i) a tuple $(w_S, p_S, time_S, prev)$, where $w_S = \sum_{i_j \in S} w_j$, $p_S = \sum_{i_j \in S} p_j$, and $time_S$ is the travel time of S to the vertex u storing i_z, or (ii) a tuple $(w', p', time', prev')$ that

dominates $(w_S, p_S, time_S, prev)$. This implies that A contains a tuple representing a simple path from s to t whose vertices store a maximum profit set S^* of items of weight at most W and total travel time at most T.

Lemma 1. *Let $1 \leq z \leq |I|$ and let vertex u hold item i_z. For each feasible subset S of S_z with respect to u there is a tuple $(w, p, time, prev)$ in the profit table A at entry $A[z]$ such that $w \leq w_S = \sum_{i_j \in S} w_j$ and $p \geq p_S = \sum_{i_j \in S} p_j$.*

Proof. We use a proof by induction on the number of entries of the profit table A. The base case is trivial as there are only two feasible subsets of S_1 with respect to s, and so $A[1]$ stores the tuple $(0, 0, 0, -1)$.

Assume that the lemma holds true for every feasible subset S of S_q with respect to the vertex holding item i_q for all $q = 1, ..., z-1$. Let S be a feasible subset of S_z with respect to the vertex holding item i_z; we show that there is a tuple $(w, p, time, prev) \in A[z]$ such that $w \leq w_S$ and $p \geq p_S$. Let u be the vertex storing i_z. We consider two cases:

- Case 1: Item $i_z \in S$. Let $S' = S - \{i_z\}$.
  - If i_z is u's first item, let u_α be a parent of u, let i_α be the last item at u_α, and let $time_\alpha$ be the travel time of S' from u_α to u. By the induction hypothesis, there is a tuple $(w', p', time', prev')$ in $A[\alpha]$ such that $w' \leq w_{S'}$ and $p' \geq p_{S'}$. Lines 7 to 17 and 22 to 26 in Algorithm 2 consider all parents u_α of u and add tuples $(w' + w_z, p' + p_z, time' + time_\alpha, prev')$ to $A[z]$ where $w' + w_z \leq w_{S'} + w_z = w_S$, $p' + p_z \geq p_{S'} + p_z = p_S$, and $time' + time_\alpha \leq T$.
  - If i_z is not u's first item, then item i_{z-1} is also located at u. By the induction hypothesis, there is a tuple $(w', p', time', prev')$ in $A[z-1]$ such that $w' \leq w_{S'}$ and $p \geq p_{S'}$. Lines 19 to 26 in Algorithm 2 add the tuple $(w' + w_z, p' + p_z, time', prev')$ to $A[z]$.
- Case 2: Item $i_z \notin S$. Since S is a feasible subset with respect to u, by the induction hypothesis, either (i) there is a tuple $(w, p, time, prev)$ in $A[z-1]$ such that $w \leq w_S$ and $p \geq p_S$ that Algorithm 2 would have copied to $A[z]$ in lines 19-20 if i_z is not u's first item, or (ii) there is a tuple $(w, p, time, prev)$ in $A[\alpha]$ (where i_α is the last item in some vertex u_α as defined above) such that $w \leq w_S$ and $p \geq p_S$ that Algorithm 2 would have copied to $A[z]$ in lines 7-17 if i_z is u's first item. □

4. PTAS for thief orienteering on DAGs

Algorithm 1 might not run in polynomial time because the size of the profit table might become too large. We can convert Algorithm 1 into a PTAS by carefully rounding down the profit and rounding up the weight and travel times associated with each item.

Note that if we simply use rounded weights in the profit table described in Section 3.1, then we might introduce a very large error to the travel times computed by Algorithm 2. To prevent this, we modify the profit table so that every entry $A[j]$ holds a list of tuples $(w_r, w_t, p, time_r, prev)$, where each tuple indicates that there is a subset S of S_j in which the sum of their rounded weights is w_r, the sum of their true weights is w_t, the sum of their rounded profits is p, the rounded travel time of S to the vertex u holding item i_j is $time_r$, and the path of S to u corresponding to this tuple includes the vertex which contains i_{prev}. Let P_{max} be the maximum profit of an item that can be transported within the allotted time T from the vertex that initially stored it to the destination vertex. Our PTAS is described in Algorithm 4.

Algorithm 4 ThOPDAGPTAS$(G = (V, E), W, T, s, t, I, \delta, \epsilon)$.

1: **Input:** DAG G, knapsack capacity W, time limit T, start vertex s, end vertex t, item assignments I, minimum velocity δ, and constant $\epsilon > 0$.
2: **Output:** A solution for ThOP of profit at least $(1 - 3\epsilon)OPT$ that uses time at most $T(1 + \epsilon)$, where OPT is the value of an optimum solution.
3: Delete from G vertices unreachable from s and vertices that cannot reach t.
4: Compute a topological ordering for G.
5: Index V by increasing topological ordering and index items as described in Section 3.
6: Let $K = \frac{1}{\epsilon}$.
7: Let $\mathbb{S}$ be the set of all feasible subsets S of $S_{|I|}$ with respect to t such that $|S| \leq K$.
8: **for** each $S \in \mathbb{S}$ **do**
9: Let A be an empty profit table. Set $A[1] = (0, 0, 0, 0, -1)$.
10: Let $W' = W - \sum_{i_j \in S} w_j$.
11: Round down the profit of each item in $I - S$ to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$.
12: Round up the weight of each item in $I - S$ to the nearest multiple of $\frac{\epsilon W'}{|I|^2}$.
13: **for** $i = s.index$ *to* $t.index$ **do**
14: Let u be the vertex with index i.
15: Call *UpdateProfitTable**(W, T, δ, A, u, S).
16: **end for**
17: **end for**
18: Select the profit table A^* storing the tuple with maximum profit.
19: Return *BuildKnapsack*(A^*, I).

Algorithm *UpdateProfitTable** is a slight modification of Algorithm 2 that is described in Algorithm 5. Travel times are rounded up to the nearest multiple of $\frac{\epsilon T}{n}$.

A tuple $(w_r, w_t, p, time_r, prev)$ dominates a tuple $(w'_r, w'_t, p', time'_r, prev')$ if $p \geq p'$, $w_r \leq w'_r$, $time_r \leq time'_r$, and $w_t \leq w'_t$. Therefore, we can assume each list $A[j]$ has the following properties: (i) the tuples are sorted in non-decreasing order of their rounded profits,

(ii) there might be multiple tuples with the same rounded profit and these tuples are sorted in non-decreasing order of their rounded weights, (iii) there might be multiple tuples with the same rounded weight and these tuples are sorted in non-decreasing order of their rounded travel times, and (iv) if several tuples in $A[j]$ have the same rounded values of profit, weight, and travel time, only the tuple with the smallest true weight is kept in $A[j]$.

Algorithm 5 UpdateProfitTable*(W, T, δ, A, u, S).

Input: Knapsack capacity W, time limit T, minimum velocity δ, profit table A, vertex u, and item subset S.
Output: The entries of the profit table A corresponding to u's items are updated to represent the subsets of items found along all paths from s to u.
for each item i_j of u **do**
  Let w_t, w_r, and p_r be the weight, rounded weight, and rounded profit of i_j, respectively. If $i_j \in S$, set $w_r = w_t$ and $p_r =$ profit of i_j.
  if i_j is u's first item **then**
    $A[j] = \emptyset$.
    for each parent v of u **do**
      Let id_v be the index of v's last item.
      for each $(w'_r, w'_t, p'_r, time'_r, prev') \in A[id_v]$ **do**
        Let $d_{u,v}$ be the distance from v to u.
        Let $\eta = 1 - (1-\delta)w'_t/W$.
        Let $travel = \frac{d_{u,v}}{\eta}$.
        if $time'_r + travel \leq T(1+\epsilon)$ **then**
          Append $(w'_r, w'_t, p'_r, t'_r, id_v)$ to $A[j]$, where t'_r is the nearest multiple of $\frac{\epsilon T}{n}$ that is larger than or equal to $time'_r + travel$.
        end if
      end for
    end for
  else
    Copy all tuples of $A[j-1]$ into $A[j]$.
    For each $(w'_r, w'_t, p'_r, time'_r, prev')$ in $A[j]$, set $prev' = j - 1$.
  end if
  for each $(w'_r, w'_t, p'_r, time'_r, prev') \in A[j]$ **do**
    if $w_r + w'_r \leq W$ **then**
      Append $(w_r + w'_r, w_t + w'_t, p_r + p'_r, time'_r, prev')$ to $A[j]$.
    end if
  end for
  if $i_j \in S$ **then** remove tuples that do not include i_j from $A[j]$.
  Remove dominated tuples from $A[j]$.
end for

4.1. PTAS analysis

Since the weights of items are rounded up, the solution produced by Algorithm 4 might have unused space where the algorithm could not fit any rounded items. However, an optimal solution would not leave empty space if there were items that could be placed in the knapsack while still travelling from s to t in at most T time, so we need to bound the maximum profit lost due to the rounding of the weights and profits, and we also need to bound the increase in travel time caused by the rounding.

Lemma 2. *For any constant $\epsilon > 0$, Algorithm 4 computes a feasible solution with profit at least $(1-3\epsilon)OPT$ that uses travel time at most $(1+2\epsilon)T$, where OPT is the profit of an optimum solution.*

Proof. Let S_{OPT} be the set of items in an optimum solution and let $OPT = \sum_{i_j \in S_{OPT}} p_j$. If $|S_{OPT}| \leq K$ then Algorithm 4 computes an optimum solution; hence, for the rest of the proof we assume that $|S_{OPT}| > K$. Let S_K be the set of the K items with largest profit from S_{OPT}, where $K = \frac{1}{\epsilon}$, and let $W' = W - \sum_{i_j \in S_K} w_j$. Let S_A be the set of items in the solution selected by our algorithm, and let $SOL = \sum_{i_j \in S_A} p_j$.

In the sequel, we will use w'_j and p'_j to refer to the rounded weight and profit of item i_j, and w_j and p_j to refer to the true weight and profit of item i_j. Given a set X of items let $weight(X) = \sum_{i_j \in X} w_j$, $weight'(X) = \sum_{i_j \in X} w'_j$, $profit(X) = \sum_{i_j \in X} p_j$, and $profit'(X) = \sum_{i_j \in X} p'_j$. We let $profit'(S_K) = profit(S_K)$ and $weight'(S_K) = weight(S_K)$.

For this proof a subset S of $S_{|I|}$ is feasible if $weight'(S) \leq W$ and the total travel time of S using the real weights of the items in S (not their rounded weights) is at most $(1+\epsilon)T$. Recall that our algorithm considers solutions that include every possible feasible subset of at most K items in the knapsack. Therefore, our algorithm must have included S_K in the knapsack in one of the iterations and filled the remainder of the knapsack using the profit table. Let S^*_A be the solution computed by our algorithm in the iteration where it chose to include the items of S_K in the knapsack, and let $SOL^* = \sum_{i_j \in S^*_A} p_j$. Since our algorithm returns the best solution that it found over all iterations, then $SOL \geq SOL^*$.

To compare SOL^* to OPT, we round up the weight of each item in $S_{OPT} - S_K$ to the nearest multiple of $\frac{\epsilon W'}{|I|^2}$; note that the weights and profits of the items in S_K are not rounded.

Rounding up the weight of a single item increases the weight of that item by at most $\frac{\epsilon W'}{|I|^2}$, so $weight'(S_{OPT}) \leq weight(S_{OPT}) + \frac{\epsilon W'}{|I|} \leq W + \frac{\epsilon W'}{|I|}$ as $|S_{OPT}| \leq |I|$. Let A_{OPT} be a subset of $S_{OPT} - S_K$ with $weight'(A_{OPT}) \leq W'$ and maximum rounded profit.

Note that $A_{OPT} \cup S_K$ is a feasible set of items. Let $\hat{i}_1, \hat{i}_2, ..., \hat{i}_D$ be the set of items in $A_{OPT} \cup S_K$ listed in increasing order of index, and let $\hat{C}_j$ be the set formed by the first j items of $A_{OPT} \cup S_K$ for all $j = 0, 1, ..., D$. We show that in the entry of the profit table corresponding to item $\hat{i}_j$ our algorithm must have included a tuple $\tau^* = (w_r^*, w_t^*, p^*, time_r^*, prev^*)$ that corresponds to a feasible subset such that $w_t^* \leq weight(\hat{C}_j)$, $w_r^* \leq weight'(\hat{C}_j)$, $p^* \geq profit'(\hat{C}_j)$, and $time_r^*$ is at most $(1+\epsilon)time_j + \epsilon T\frac{m_j}{n}$, where m_j is the number of vertices in the path selected by the optimum solution from s to the vertex u storing the tuple τ and $time_j$ is the time needed by the optimum solution to transport the items in $\hat{C}_j$ to u. To show that this tuple exists, we use a proof by induction on the number of items in $\hat{C}_j$.

The base case, when $j = 0$, so $|\hat{C}_0| = 0$, is trivial as our algorithm adds to the profit table the entry $(0, 0, 0, -1)$. Assume then that the claim is true for $\hat{C}_j$ with $j < D$. In $\hat{C}_{j+1}$, $\hat{i}_{j+1}$ is the item with largest index. By the induction hypothesis, there is an entry of the profit table corresponding to $\hat{i}_j$ storing a tuple $\tau^* = (w_r^*, w_t^*, p^*, time_r^*, prev^*)$ as above.

Let u be the vertex storing $\hat{i}_j$, v be the vertex storing $\hat{i}_{j+1}$, and let $d_{u,v}$ be the distance between u and v in the path P selected by OPT from s to v. Our algorithm considers transporting the items corresponding to the tuple τ^* from u to v and that would add a tuple $(w_r'', w_t'', p'', time_r'', prev'')$ to the entry $A[\pi(\hat{i}_{j+1})]$ corresponding to item $\hat{i}_{j+1}$, where by the induction hypothesis $w_t'' = w_t^* \leq weight(\hat{C}_j)$. Since our algorithm removes dominated tuples from the profit table, then entry $A[\pi(\hat{i}_{j+1})]$ must store a tuple $\tau^\sim = (w_r^\sim, w_t^\sim, p^\sim, time_r^\sim, prev^\sim)$ such that $w_r^\sim \leq w_r''$, $w_t^\sim \leq w_t''$, $p^\sim \geq p''$, and $time_r^\sim \leq time_r''$. Since $w_t^\sim \leq w_t'' \leq weight(\hat{C}_j)$, then the time $time_{uv}^\sim$ that the thief needs to carry weight w_t'' from u to v following the path selected by OPT is at most the time $time_{uv}$ needed in the optimum solution to carry the items in $\hat{C}_j$ from vertex u to vertex v.

In line 14 of Algorithm 5 we round the traveling times, and since a path between u and v has $m_{uv} \leq n$ vertices, then the number of times that we round traveling times when computing the tuples needed to produce $\tau^\sim$ from τ^* is $m_j + m_{uv} \leq n$ and so $time_r^\sim \leq time_r^* + time_{uv} + \epsilon T\frac{m_{uv}}{n}$. Hence, by the induction hypothesis, $time_r^\sim \leq (1+\epsilon)time_j + time_{uv} + \epsilon T\frac{m_j+m_{uv}}{n} \leq (1+\epsilon)time_{j+1} + \epsilon T\frac{m_j+m_{uv}}{n}$, where $time_{j+1} = time_j + time_{uv}$ is the time needed by the optimum solution to transport $\hat{C}_{j+1}$ from s to v.

After adding item $\hat{i}_{j+1}$ to the tuple $\tau^\sim$ and removing duplicated tuples, the profit table must include a tuple $(w_r^\#, w_t^\#, p^\#, time_r^\#, prev^\#)$ such that $w_r^\# \leq w_r^\sim + weight'(\hat{i}_{i+j}) \leq weight'(\hat{C}_{j+1})$, $p^\# \geq profit'(\hat{C}_{j+1})$, $time_r^\# \leq (1+\epsilon)time_{j+1} + \epsilon T\frac{m_j+m_{uv}}{n}$, and $w_t^\# \leq weight(\hat{C}_{j+1})$, as for tuples with the same rounded weight, profit, and travel times the algorithm selects that with smallest true weight.

Hence, our algorithm will compute a tuple $(w_r, w_t, p, time_r, prev)$, where $w_r \leq weight'(A_{OPT} \cup S_K)$, $w_t \leq weight(A_{OPT} \cup S_K)$, $p \geq profit'(A_{OPT} \cup S_K)$, and $time_r \leq (1+\epsilon)time_D + \epsilon T\frac{n}{n} \leq (1+2\epsilon)T$, where $time_D \leq T$ is the time needed by the optimum solution to transport the items in $A_{OPT} \cup S_K$ from s to t.

Since

$$SOL^* \geq p \geq profit'(A_{OPT} \cup S_K) \tag{1}$$

we need to bound $profit'(A_{OPT} \cup S_K)$.

Let $S_{OPT}^- = S_{OPT} - S_K - A_{OPT}$, this set includes the items in the optimum solution whose profit is not included in the right hand side of (1). Note that if S_{OPT}^- is empty, then $S_K \cup A_{OPT} = S_{OPT}$ and so $SOL^* \geq profit'(S_{OPT})$. If S_{OPT}^- is not empty, we show that $weight'(S_{OPT}^-) \leq \frac{\epsilon W'}{|I|} + w_L$, where w_L is the largest weight of the items in S_{OPT}^-. To see this, recall that $weight'(S_{OPT}) \leq W + \frac{\epsilon W'}{|I|}$ and note that $weight(S_K) + weight'(A_{OPT}) \leq W$, but by the way in which A_{OPT} was defined the empty space that $S_K \cup A_{OPT}$ leave in the knapsack is not large enough to fit the rounded weight of another item from S_{OPT}^-. Therefore,

$$\begin{aligned} weight'(S_{OPT}^-) &= weight'(S_{OPT}) - (weight(S_K) + weight'(A_{OPT})) \\ &\leq W + \frac{\epsilon W'}{|I|} - (W - w_L) \\ &= \frac{\epsilon W'}{|I|} + w_L \end{aligned} \tag{2}$$

Now we bound $profit'(S_{OPT}^-)$. If S_{OPT}^- consists of only one item i_j, then since the least profitable item in S_K has profit at most $\frac{1}{K}OPT$ and i_j is not in S_K, then $p_j \leq \frac{1}{K}OPT$, which means that $profit'(S_{OPT}^-) \leq \frac{1}{K}OPT = \epsilon OPT$. If S_{OPT}^- consists of two or more items, we can partition S_{OPT}^- into the singleton $\{i_L\}$ consisting of the item with largest weight w_L and the set i_* of remaining items, which by (2) has $weight'(i_*) \leq \frac{\epsilon W'}{|I|}$.

Item i_L is not in S_K, so it has profit $p_L \leq \frac{1}{K}OPT$. As for i_* we show that $profit'(i_*) \leq \frac{1}{K}OPT$:

- $W' - weight'(A_{OPT}) < \frac{\epsilon W'}{|I|} = \frac{W'}{K|I|}$, as otherwise the items i_* would have been included in A_{OPT}, and so

$$weight'(A_{OPT}) = \sum_{i_j \in A_{OPT}} w_j' > W' - \frac{W'}{K|I|} > \frac{K-1}{K}W', \text{ as } |I| \geq 1 \tag{3}$$

- There is at least one item i_ψ in A_{OPT} with $w'_\psi \geq \frac{W'}{K|I|}$. To see this, note that if all of the items in A_{OPT} had weight strictly less than $\frac{W'}{K|I|}$, then $\sum_{i_j \in A_{OPT}} w'_j < \frac{W'}{K|I|}|A_{OPT}| < \frac{1}{K}W'$, as $|A_{OPT}| \leq |I|$, which contradicts (3).
- By definition, A_{OPT} includes items from $S_{OPT} - S_K$ with $weight'(A_{OPT}) \leq W'$ and maximum profit, and since the items i_* are not in A_{OPT} then $profit'(i_*) \leq p'_\psi$, as otherwise the items i_* would be in A_{OPT} instead of i_ψ. Since item i_ψ is not in S_K then $p'_\psi \leq \frac{1}{K}OPT$ and so $profit'(i_*) \leq \frac{1}{K}OPT$.

Therefore, $profit'(S^-_{OPT}) = profit'(i_L) + profit'(i_*) \leq \frac{2}{K}OPT$. Since $S_{OPT} = S_K \cup A_{OPT} \cup S^-_{OPT}$ then $profit'(A_{OPT}) + profit'(S_K) = profit'(S_{OPT}) - profit'(S^-_{OPT}) \geq profit'(S_{OPT}) - \frac{2}{K}OPT$. By (1),

$$\begin{aligned} SOL^* &\geq profit'(A_{OPT}) + profit'(S_K) \\ &\geq profit'(S_{OPT}) - \frac{2}{K}OPT \\ &\geq profit(S_{OPT}) - \frac{\epsilon P_{max}}{|I|}|S_{OPT}| - \frac{2}{K}OPT \\ &\geq OPT - \epsilon P_{max} - 2\epsilon OPT \geq (1-3\epsilon)OPT \end{aligned}$$

Since $SOL \geq SOL^*$, then $SOL \geq (1-3\epsilon)OPT$. □

Theorem 3. *There is a PTAS for ThOP on DAGs that produces solutions with travel time at most $T(1+\epsilon)$ for any constant $\epsilon > 0$.*

Proof. As shown by Lemma 2, Algorithm 4 computes solutions with profit at least $(1-3\epsilon)OPT$. The profit table A has $|I|$ entries, and each entry $A[j]$ can have at most $O(P_r W_r T_r)$ tuples, where P_r is the number of different values for the rounded profit of any subset of items, W_r is the number of different values for the rounded weight of any subset of items of total weight at most W, and T_r is the maximum number of rounded travel times. Since profits are rounded down to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$ then P_r is $O(\frac{|I|^2}{\epsilon})$. Since weights of items not in the selected feasible subsets S are rounded up to the nearest multiple of $\frac{\epsilon W'}{|I|^2}$, then W_r is $O(\frac{|I|^2}{\epsilon})$; T_r is $O(\frac{n}{\epsilon})$.

Algorithm 4 iterates through the list at each entry $A[j]$ exactly once if i_j is not the last item in a vertex u; if i_j is the last item in a vertex u, then our algorithm iterates through the list at entry $A[j]$ once for each outgoing edge of u. Thus, for each feasible subset S with at most K items the algorithm loops through $|I|$ rows in the profit table, iterates over a particular row at most $n = |V|$ times, and each row can have at most $O(\frac{|I|^4 n}{\epsilon^3})$ tuples in it; since the number of feasible subsets S is $O(|I|^{\frac{1}{\epsilon}})$ the running time is $O(\frac{n^2}{\epsilon^3}|I|^{5+\frac{1}{\epsilon}})$. □

Corollary 2. *There is a PTAS for ThOP on DAGs when $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ that produces solutions with travel time at most T.*

Proof. Without loss of generality we can assume that $\mathcal{V}_{\min} = \mathcal{V}_{\max} = 1$, so the travel time between vertices u and v is $d_{u,v}$. Therefore, in Algorithms 4 and 5 we do not need to keep rounded travel times or real item weights in the tuples. Hence, a tuple $(w_r, p, time, prev)$ dominates a tuple $(w'_r, p', time', prev')$ if $p \geq p'$, $w_r \leq w'_r$, and $time \leq time'$. Since the algorithm now uses real travel times, the solution produced by the algorithm uses total travel time at most T. □

5. Thief orienteering on undirected graphs

In this section we consider the case when the input graph $G = (V, E)$ is undirected and has integer edge lengths of at least 1 and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$. Notice that if $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ the travel time for any edge is equal to the length of the edge and it is independent of the weight of the items in the knapsack. First, we consider when T is equal to the length L of a shortest path from s to t.

Theorem 4. *There is a FPTAS for ThOP where $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ and the time T is equal to the length L of a shortest path from s to t.*

Proof. Delete from G every edge (u, v) that does not belong to a shortest path from s to t. If the input graph G is undirected, direct the remaining edges towards the endpoint closer to t. Note that G is now a DAG. We modify Algorithm 1 so that it rounds down the profit of each item to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$, and a tuple with profit p dominates a tuple with profit p' if $p \geq p'$ and $w \leq w'$. Observe that since the paths from s to t are shortest paths, the travel times for both of these tuples must be the same, and hence the weights and travel times of the items do not need to be rounded up.

Furthermore, this modified Algorithm 1 does not need to consider all possible feasible subsets S of at most K items because in Section 4 we needed to consider all these subsets to bound the maximum profit loss due to the rounding of the weights. Hence, this modified Algorithm 1 is a FPTAS. □

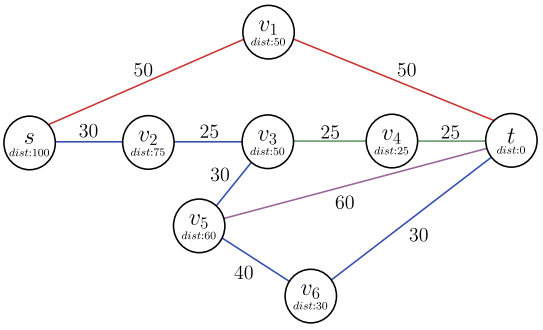

Fig. 2. Let σ be the path from s to t that includes the edges (s,v_2), (v_2,v_3), (v_3,v_5), (v_5,v_6), (v_6,t). The edges $((s,v_1),(v_1,t))$ belong to a shortest path (of length $s.dist = 100$) between s and t, the edges (v_3,v_4), (v_4,t) belong to a shortest path (of length $v_3.dist = 50$) between v_3 and t, and the edge (v_5,t) belongs to a shortest path (of length $v_5.dist = 60$) between v_5 and t. The path σ decomposes into the unique list $D = D_1, D_2$ of detours where $D_1 = \{s, v_2\}$ and $D_2 = \{v_3, v_5, v_6\}$.

In the rest of this section we show that Theorem 4 can be extended to the case when $T = L + K$, where $K > 0$ is a integer constant. In the sequel we assume that the input graph is undirected. It is not hard to see how our ideas can be extended to directed graphs. Note that when $T = L + K$ we cannot simply transform G into a DAG using the same process as described in the proof of Theorem 4, as in this version of the problem a feasible solution might include edges that do not belong to a shortest path between s and t.

Let $u.dist$ be the shortest distance from vertex u to t, let σ be a path from s to t with length at most $L + K$, and let the vertices in σ be $s = u_1, u_2, u_3, ..., u_{|\sigma|} = t$. If σ includes an edge (u_i, u_{i+1}) that does not belong to any shortest path from u_i to t or, more formally, if $u_i.dist < u_{i+1}.dist + d_{u_i,u_{i+1}}$, then we say that (u_i, u_{i+1}) is part of a *detour*.

Definition 2. A simple path σ from s to t decomposes into a unique list $D = D_1, D_2, ..., D_{n_D}$ of vertex disjoint *detours*, where D might be empty. Each detour D_i forms a longest subpath of σ where no edge (u,v) of D_i belongs to a shortest path from u to t, for all $i = 1, 2, ..., n_D$. All edges (u,v) in σ that do not belong to a shortest path from u to t belong to exactly one detour; hence, the list D is unique.

We show a few examples of detours in Fig. 2; note that we have omitted the items information. Next, we show that the number of edges that belong to detours is at most the constant K. This fact will help us to reduce the running time of our algorithm, as it enumerates all possible sets of detours.

Lemma 3. *A simple path σ from s to t of length at most $L + K$ has at most K edges (u,v) such that (u,v) is in a detour.*

Proof. We use a proof by contradiction. Assume that path σ from s to t of length at most $L + K$ includes edges $(u_1, v_1), (u_2, v_2), ..., (u_a, v_a)$ with $a > K$ that do not belong to any shortest path from u_i to t. Let $\ell(u,v)$ be the length of the subpath of σ from u to v and let $u_{a+1} = t$. Then $d_{u_i,v_i} + \ell(v_i, u_{i+1}) + u_{i+1}.dist > u_i.dist$, for all $i = 1, 2, ..., a$. Since $\ell(u_i, u_{i+1}) = d_{u_i,v_i} + \ell(v_i, u_{i+1})$ then $\ell(u_i, u_{i+1}) \geq u_i.dist - u_{i+1}.dist + 1$ as all edges have length at least 1.

Therefore, $\ell(s,t) = \ell(s,u_1) + \sum_{i=1}^{a} \ell(u_i, u_{i+1}) \geq \ell(s,u_1) + \sum_{i=1}^{a} (u_i.dist - u_{i+1}.dist + 1) = \ell(s,u_1) + u_1.dist + a \geq L + a$. Since the length $\ell(s,t)$ of σ is at most $L + K$ but $a > K$ then we have reached a contradiction. □

Corollary 3. *A simple path σ from s to t of length at most $L + K$ decomposes into a list D containing at most K detours.*

Note that by the definition of a detour, for a simple path σ from s to t the edges (u,v) that do not belong to any detours must belong to a shortest path from u to t. We show next how to decompose σ into alternating subpaths of detours and subpaths that are shortest paths between their first and last vertices.

Corollary 4. *A simple path σ from s to t of length at most $L + K$ decomposes into subpaths $\sigma = P_1, D_1, P_2, D_2, ..., P_r, D_r, P_{r+1}$ where $r \leq K$, each D_i is a detour, and each P_i is a shortest path between its first and last vertices.*

Proof. Let $P_i = u_{i_1}, u_{i_2}, ..., u_{i_e}$. Each edge $(u_{i_j}, u_{i_{j+1}})$ is in a shortest path from u_i to t as otherwise the edge would belong to a detour. Hence, $u_{i_j}.dist = d_{u_{i_j},u_{i_{j+1}}} + u_{i_{j+1}}.dist$ for each $i = 1, ..., e-1$. Therefore, the length of P_i is $\sum_{j=1}^{e-1} d_{u_{i_j},u_{i_{j+1}}} = \sum_{j=1}^{e-1} (u_{i_j}.dist - u_{i_{j+1}}.dist) = u_{i_1}.dist - u_{i_e}.dist$ and thus P_i is a shortest path between u_{i_1} and u_{i_e}.

The remaining subpaths in σ are the detours $D_1, D_2, ..., D_{n_D}$ and by Corollary 3, $n_D \leq K$. □

In the sequel we say that a path σ from s to t *decomposes* into a list D of detours.

5.1. Building the lists of detours

We define the *profit of a path* as the maximum profit achievable by collecting items of total weight at most W along that path.

Let σ^* be a simple path from s to t with maximum profit of length at most $L+K$. If σ^* is a shortest path from s to t, then Theorem 4 provides an FPTAS for it. Hence, in the sequel we assume that σ^* is not a shortest path from s to t, and so it must have at least one detour. The path σ^* decomposes into a unique list $D^* = D_1^*, D_2^*, ..., D_{n_{D^*}}^*$ of n_{D^*} detours, where $n_D^* \leq K$ (by Corollary 3), D has at most K edges (by Lemma 3), and the subpaths of σ^* that do not belong to any detours are shortest paths between their first and last vertices (by Corollary 4).

Since we do not know how many detours are in D^*, or which vertices are in each of them, our algorithm to compute a path σ from s to t of length at most $L+K$ and near maximum profit will generate the set $\mathcal{D}$ of all possible *valid* lists of detours as shown in Algorithm 6.

Definition 3. A list D of detours $D_1, D_2, ..., D_{n_D}$ is *valid* if there exists a simple path σ from s to t of length at most $L+K$ that decomposes into D.

Algorithm 6 GenerateDetours($G=(V,E)$).

```
1: Input: Graph G.
2: Output: The set 𝒟 of all valid detours for paths from s to t in G.
3: Let 𝒟 = ∅.
4: for each subset S_V of V such that 2 ≤ |S_V| ≤ 2K do
5:     for n_D = 1 to K do
6:         for each possible partitioning of S_V into a list D of n_D detours do
7:             Compute a shortest path σ from s to t in G that decomposes into D.
8:             if σ has length at most L + K then
9:                 𝒟 = 𝒟 ∪ D.
10:            end if
11:        end for
12:    end for
13: end for
14: Output 𝒟.
```

Note that D^* must be included within the set $\mathcal{D}$ computed by Algorithm 6. For each list $D \in \mathcal{D}$ we will create a DAG G_D containing all the paths from s to t of length at most $L+K$ that go through the detours in D. Algorithm 4 will then be used on these DAGs, and the highest profit path found will be output.

5.2. Transforming the problem into a shortest paths problem

We transform the problem of finding in a given graph G a simple path with maximum profit from s to t of length at most $L+K$ into the problem of finding in a collection of directed graphs shortest paths with maximum profit from s to t. We create a graph G_D from G with the following properties: (i) for each simple path σ in G from s to t of length at most $L+K$ that decomposes into a valid list D of detours there is an equivalent shortest path in G_D from s to t, and (ii) for every shortest path from s to t in G_D there is an equivalent simple path σ in G of length at most $L+K$ that decomposes into a valid list D of detours.

Definition 4. An undirected path σ from s to t in G and a directed path σ_D from s to t in G_D are *equivalent* if they include the same vertices and edges.

To ensure that all paths in G from s to t that travel through D are shortest paths in G_D we will carefully reduce the lengths of the edges in D (allowing some of them to have negative lengths) so that a shortest path in G_D must travel through D. Algorithm 7 describes how to construct G_D. For any detour D_i in D, an *internal* vertex of D_i is any vertex that is not the first or last vertex of D_i.

Corollary 5. *Let D be a valid list of detours. For each simple path σ in G from s to t that decomposes into D, the graph G_D output by Algorithm 7 contains all the edges of σ and hence σ is a path in G_D from s to t.*

Proof. Recall that a path σ in G from s to t that decomposes into D has the form $P_1, D_1, P_2, D_2, ..., P_r, D_r, P_{r+1}$, where each D_i is a detour in D and each P_j is a shortest path between its first and last vertices for all $i = 1, 2, ..., r$ and $j = 1, 2, ..., r+1$. Since initially G_D was a copy of G and the only edges that are deleted from G_D are edges not in detours incident on internal vertices of detours or edges (u, v) not in D for which $u.dist = v.dist$, then all the edges of σ are also directed in G_D. □

Let σ be a simple path from s to t in G of length at most $L+K$ that decomposes into the non-empty valid list $D = D_1, D_2, ..., D_{n_D}$ of detours and let G_D be the graph output by Algorithm 7. By Corollary 5, σ is a path in G_D from s to t. To avoid confusion, let σ_D denote the directed version of path σ when referring to the graph G_D. For any two vertices u and v of σ, let $\ell(u, v)$ be the length of

Algorithm 7 DetourToDAG($G=(V,E)$, D, s, t, I).

Input: Graph G, valid list of detours D, start vertex s, end vertex t, and item assignments I.
Output: A DAG G_D containing paths from s to t equivalent to the paths from s to t in G that travel through all of the detours in D.
Let G_D be a copy of G, with the same item assignments I.
For each vertex $u \in G_D$ compute $u.dist$.
for each internal vertex $u_i \in D_i$, for all $D_i \in D$ **do**
    Delete from G_D all edges incident on u_i except for (u_{i-1}, u_i) and (u_i, u_{i+1}), where u_{i-1} is the vertex preceding u_i in D_i and u_i is the vertex preceding u_{i+1} in D_i.
end for
Let σ be a simple path from s to t that decomposes into D.
for each $D_i \in D$ **do**
    Direct the edges in D_i from its first to its last vertex according to the order in which the vertices appear in σ.
end for
Index the directed edges $E_D = e_1, e_2, ..., e_{m_D}$ of the detours in D in the order they appear in σ.
for directed edge e_i with endpoints (u_i, v_i), for all $i = 1, 2, ..., m_D$ **do**
    $d_{u_i,v_i} = u_i.dist - v_i.dist - 1$.
end for
Delete any edges (u, v) in G_D that do not belong to any detours of D such that $u.dist = v.dist$.
Direct any remaining undirected edges in G_D towards the endpoint closer to t.
Output the DAG G_D.

the subpath in σ between u and v and let $\ell_D(u,v)$ be the length of the subpath in σ_D between u and v. Let $d'_{u,v}$ be the length of edge (u,v) in G_D. For a vertex u, let $u.dist$ be the length of a shortest path in G from u to t; note that Algorithm 7 does not change the value $u.dist$, the algorithm only modifies the lengths of edges in G_D and not those in G.

Let $E_D = e_1, e_2, ..., e_{m_D}$ be the list of all edges in D in the order in which they appear when σ_D is traversed from s to t. We will prove in the next lemma that for edge e_z with endpoints (u_z, v_z), $\ell_D(s, v_z) = s.dist - v_z.dist - z$, for all $z = 1, 2, ..., m_D$.

Lemma 4. *Let σ be a simple path from s to t in G of length at most $L+K$ that decomposes into the non-empty valid list D of detours, let G_D be the graph output by Algorithm 7, and let σ_D denote the directed version of path σ when referring to the graph G_D. Let $E_D = e_1, e_2, ..., e_{m_D}$ be the list of all edges in D in the order in which they appear when σ_D is traversed from s to t. For edge e_z with endpoints (u_z, v_z), $\ell_D(s, v_z) = s.dist - v_z.dist - z$, for all $z = 1, 2, ..., m_D$.*

Proof. We use a proof by induction on the edges in E_D. For the base case we consider the first edge e_1 with endpoints (u_1, v_1). In line 14 of Algorithm 7 edge (u_1, v_1) is assigned length $d'_{u_1,v_1} = u_1.dist - v_1.dist - 1$ (note that in G_D we allow negative lengths but only for the edges in the detours). By Corollary 4, $\ell(s, u_1) = s.dist - u_1.dist$ (as the subpath of σ from s to u_1 is a shortest path between s and u_1) and since all edges in the subpath of σ_D from s to u_1 have the same lengths as the corresponding edges in σ, then $\ell_D(s, u_1) = s.dist - u_1.dist$, and since $\ell_D(s, v_1) = \ell_D(s, u_1) + d'_{u_1,v_1}$, then $\ell_D(s, v_1) = s.dist - u_1.dist + u_1.dist - v_1.dist - 1 = s.dist - v_1.dist - 1$.

Assume now that the claim holds true for every edge e_q in E_D with endpoints (u_q, v_q), for all $q = 1, 2, ..., j-1$, so that $\ell_D(s, v_q) = s.dist - v_q.dist - q$. We show for the edge e_j in E_D with endpoints (u_j, v_j) that $\ell_D(s, v_j) = s.dist - v_j.dist - j$.

Edge e_j with endpoints (u_j, v_j) is assigned length $d'_{u_j,v_j} = u_j.dist - v_j.dist - 1$ in line 14 of Algorithm 7. Let e_j be part of detour D_i. We need to consider two cases:

- Case 1: Edge e_j is the first edge of D_i. Since $\ell_D(s, v_j) = \ell_D(s, v_{j-1}) + \ell_D(v_{j-1}, u_j) + d'_{u_j,v_j}$, by the induction hypothesis $\ell_D(s, v_{j-1}) = s.dist - v_{j-1}.dist - (j-1)$. By Corollary 4 $\ell(v_{j-1}, u_j) = v_{j-1}.dist - u_j.dist$ (as the subpath of σ from v_{j-1} to u_j is a shortest path between these vertices) and all edges in the subpath of σ_D from v_{j-1} to u_j have the same lengths as the corresponding edges in σ, then $\ell_D(v_{j-1}, u_j) = v_{j-1}.dist - u_j.dist$. Since $d'_{u_j,v_j} = u_j.dist - v_j.dist - 1$, then $\ell_D(s, v_j) = s.dist - v_{j-1}.dist - (j-1) + v_{j-1}.dist - u_j.dist + u_j.dist - v_j.dist - 1 = s.dist - v_j.dist - j$.
- Case 2: Edge e_j is not the first edge of D_i. Since $\ell_D(s, v_j) = \ell_D(s, u_j) + d'_{u_j,v_j}$, and by the induction hypothesis $\ell_D(s, u_j) = s.dist - u_j.dist - (j-1)$, then $\ell_D(s, v_j) = s.dist - u_j.dist - (j-1) + u_j.dist - v_j.dist - 1 = s.dist - v_j.dist - j$. □

Corollary 6. *Let σ be a simple path from s to t in G of length at most $L+K$ that decomposes into the non-empty valid list $D = D_1, D_2, ..., D_{n_D}$ of detours and let G_D be the graph output by Algorithm 7. The path σ_D in G_D has length $L - m_D$, where m_D is the number of edges in D.*

Proof. Let $E_D = e_1, e_2, ..., e_{m_D}$ be the list of all edges in D in the order in which they appear when σ_D is traversed from s to t. By Lemma 4 for edge e_z with endpoints (u_z, v_z), $\ell_D(s, v_z) = s.dist - v_z.dist - z$, for all $z = 1, 2, ..., m_D$. Then, since $\ell_D(s, t) = \ell_D(s, v_{m_D}) + \ell_D(v_{m_D}, t)$, where $e_{m_D} = (u_{m_D}, v_{m_D})$ and by Corollary 4 $\ell(v_{m_D}, t) = v_{m_D}.dist$ (as the subpath of σ from v_{m_D} to t contains no edges that belong to a detour) and since the edges in the subpath of σ_D from v_{m_D} to t have the same lengths as the corresponding edges in σ, then $\ell_D(v_{m_D}, t) = v_{m_D}.dist$; therefore, σ_D has length $\ell_D(s, t) = \ell_D(s, v_{m_D}) + \ell_D(v_{m_D}, t) = s.dist - v_{m_D}.dist - m_D + v_{m_D}.dist = s.dist - m_D = L - m_D$. □

By Lemma 4 and Corollary 6, we can see that the length of a shortest path in G_D from s to t is less than the length of a shortest path in G from s to t. This is important, because we do not want the shortest paths in G from s to t which do not decompose into D

to also be shortest paths in G_D from s to t. We show that all the shortest paths in G_D from s to t must decompose into D as this is critical to our algorithm.

Lemma 5. *Let σ be a simple path from s to t in G of length at most $L+K$ that decomposes into the non-empty valid list $D = D_1, D_2, ..., D_{n_D}$ of detours and let G_D be the graph output by Algorithm 7. The path σ_D is a shortest path in G_D from s to t and every shortest path in G_D from s to t decomposes into D.*

Proof. By Corollary 6, the path σ_D in G_D from s to t that corresponds to the path σ has length $L - m_D$, where L is the length of a shortest path in G and m_D is the number of edges in D.

Assume, for the sake of contradiction, that there exists a path σ'_D in G_D from s to t with length smaller than $L - m_D$. Since there are only m_D edges that belong to detours, and the edges in σ'_D that belong to detours are the only edges whose lengths were changed with respect to the lengths of the edges in the corresponding path σ' in G, then it must be the case that for some detour edge (u, v) the algorithm set its length to a value smaller than $u.dist - v.dist - 1$. However, this is not possible because line 14 of Algorithm 7 sets the length of each detour edge (u, v) to $u.dist - v.dist - 1$, and so σ_D is a shortest path from s to t in G_D.

Additionally, assume for the sake of contradiction, that there exists a shortest path σ'_D in G_D from s to t that does not include all the edges in D. Since only m_D edges belong to detours, and the edges in G_D that belong to detours are the only edges whose lengths were changed with respect to the lengths of the corresponding edges in G, then it must be the case that for some detour edge (u, v) the algorithm set its length to a value smaller than $u.dist - v.dist - 1$. However, this is not possible because line 14 of Algorithm 7 sets the length of each detour edge (u, v) to $u.dist - v.dist - 1$, and so every shortest path in G_D from s to t decomposes into D. □

Corollary 7. *Let D be a valid list of detours in G. For the graph G_D output by Algorithm 7 corresponding to D (i) for every path σ' from s to t in G of length $\Delta \leq L + K$ that decomposes into D, G_D has an equivalent shortest path σ'_D from s to t that decomposes into D and (ii) for every shortest path σ'_D from s to t in G_D there is an equivalent path σ' from s to t in G of length $\Delta \leq L + K$ that decomposes into D.*

Proof. (i) Let σ' be a path from s to t in G of length $\Delta \leq L + K$ that decomposes into D. By Corollary 5 σ' is also a path of G_D from s to t that decomposes into D, and by Lemma 5 σ' is a shortest path in G_D from s to t. Since G_D has the same vertices, edges, and item assignments as G then the two paths σ' in G and G_D are equivalent.

(ii) By Lemma 5, σ'_D decomposes into D. Since Algorithm 7 initially creates G_D by copying G and it does not add any extra edges to G_D, then σ'_D is also a path in G from s to t that decomposes into D. Since D is a valid list of detours, then the length of the path σ'_D in G is at most $L + K$. □

Algorithm 8 ThOPLengthLPlusK($G = (V, E), W, T, K, s, t, I, \epsilon$).

1: **Input:** Graph G, knapsack capacity W, time limit $T \leq L + K$, constant K, start vertex s, end vertex t, item assignments I, and constant $\epsilon > 0$, where L is the length of a shortest path from s to t.
2: **Output:** A path with length at most T from s to t with profit at least $(1 - \epsilon)$ times the maximum possible profit.
3: Let $bestPath$ be an empty path.
4: Let D be the set of all possible valid lists of detours as described in Section 5.1.
5: **for** each $D \in \mathcal{D}$ **do**
6: $G_D = DetourToDAG(G, D, s, t, I)$.
7: Let D have m_D edges and let $T' = L - m_D$.
8: $path = ThOPDAGPTAS^*(G_D, W, T', s, t, I, 1, \epsilon)$.
9: **if** profit of $path$ > profit of $bestPath$ **then**
10: $bestPath = path$.
11: **end if**
12: **end for**
13: Return $bestPath$.

Our algorithm for ThOP where $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ and the time T is equal to the length L of a shortest path from s to t plus a constant K is described in Algorithm 8. Since some edges in G_D might have negative lengths algorithm *ThOPDAGPTAS** uses a slightly modified *UpdateProfitTable* algorithm that discards the condition $time' + travel \leq T$ on line 13 of Algorithm 2. Therefore, the profit tables produced by Algorithm *ThOPDAGPTAS** might contain infeasible solutions whose total travel time is greater than T, but this is not a problem since we select the best solution that does not exceed the time limit T.

Theorem 5. *There is a FPTAS for ThOP where $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ and the time T is equal to the length L of a shortest path from s to t plus a constant K.*

Proof. Let σ^* be a path from s to t with length $\Delta \leq L + K$ with maximum profit, and let D^* be the list of detours that σ^* decomposes into. The list D^* is valid; therefore, since Algorithm 8 considers all possible valid lists of detours, then Algorithm 8 considers the list of detours D^* in one of its iterations.

By Corollary 4 and Corollary 7 the problem of finding a path in G from s to t that decomposes into the list D of detours, has length at most $L + K$, and has maximum profit is equivalent to finding a shortest path in G_D from s to t with maximum profit. For

every valid list of detours, and in particular, for the list D^*, Algorithm 8 finds a path in G from s to t that decomposes into D^*, has length at most $L + K$, and has profit at least $(1 - \epsilon)OPT$, where OPT is the value of an optimum solution.

Note that when the profit tables produced by Algorithm *ThOPDAGPTAS** contain multiple tuples with the same profit, only the tuple with the smallest weight is kept in the profit table, as $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ and paths from s to t are shortest paths and so the travel time times for these tuples must be the same, and hence the weights and travel times of the items do not need to be rounded up; therefore, each profit table produced by Algorithm *ThOPDAGPTAS** has at most $O(P_r)$ tuples, where P_r is the number of different values for the rounded profit of any subset of items. Moreover, since the item weights are not rounded then Algorithm *ThOPDAGPTAS** does not need to iterate over all possible subsets using at most K items. Hence, Algorithm *ThOPDAGPTAS** has time complexity $O(n|I|^2)$.

Furthermore, the number of possible lists of detours is polynomial with respect to the constant K and verifying whether a list of detours is valid requires computing a constant number of shortest paths, so Algorithm 8 is a FPTAS. □

6. Thief orienteering on cliques

In this section we study a version of ThOP where the graph is a clique, every edge has length 1, and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$ so the travel time of an edge does not depend on the current load in the backpack. A clique is a graph where every pair of vertices is connected by an edge. Observe that solving ThOP on a clique is equivalent to finding the $T - 1$ vertices that a path σ from s to t in G should include for the thief to collect items of maximum value and total weight at most W.

Note that since all edges have length 1 and $\mathcal{V}_{\min} = \mathcal{V}_{\max}$, the order in which a path σ traverses the vertices does not matter when computing the best path. Therefore, we index the vertices in V from 1 to $|V|$ assigning to s index 1 and t index $|V|$. We index the items stored in the vertices so that item indices are unique and items in vertex u have smaller indices than items in vertex v if $u.index < v.index$. Let the items stored in the vertices of the input graph G be $i_1, i_2, ..., i_{|I|}$.

6.1. Clique profit table

Definition 5. Let $S_z = \{i_1, ..., i_z\}$ for all $z = 1, ..., |I|$, and let vertex u hold item i_z. A subset S of S_z is a *feasible subset* (for ThOP on cliques) if it has weight $w_S = \sum w_j \leq W$ and the items in S belong to a most $T - 1$ different vertices.

Our algorithm builds a *clique profit table* where each entry $A[j]$, for $j = 1, ..., |I|$, corresponds to the item i_j with index j, and $A[j]$ is a list of tuples (w, p, v). A tuple (w, p, v) in the list of $A[j]$ indicates that there is a subset S of S_j such that:

- the items in S are stored in the vertices in v, where $|v| \leq T - 1$,
- the weight of the items in S is $w \leq W$, and
- the profit of the items in S is p.

A tuple (w, p, v) *dominates* a tuple (w', p', v') if $p \geq p'$, $w \leq w'$, and $|v| \leq |v'|$. We remove dominated tuples from each list of A such that no tuple in the list of each entry $A[j]$ dominates another tuple in the same list. Therefore, we can assume each list $A[j]$ has the following properties: (i) the tuples are sorted in non-decreasing order of their profits, (ii) there might be multiple tuples with the same profit and these tuples are sorted in non-decreasing order of their weights, and (iii) if several tuples in $A[j]$ have the same profit p and weight w, only the tuple with the smallest value of $|v|$ is kept in $A[j]$.

6.2. Computing the clique profit table

Algorithm 9 shows how we update the clique profit table with the items stored in vertex u. The start vertex s holds a single item i_1 of weight and profit 0; therefore, we initialize $A[1]$ to store the tuple $(0, 0, \emptyset)$.

Algorithm 9 UpdateCliqueProfitTables(W, T, A, u).

```
1: Input: Knapsack capacity W, time T, clique profit table A, and vertex u.
2: Output: The entries of the clique profit table are updated to represent the subsets of items taken from at most T − 1 vertices along a path from s to t that routes through u.
3: for each item i_j of u do
4:     Let w be the weight and p the profit of i_j.
5:     Copy all tuples of A[j − 1] into A[j].
6:     for each tuple (w', p', v') ∈ A[j] do
7:         if (u ∈ v') OR (u ∉ v' AND |v'| < T − 1) then
8:             if w + w' ≤ W then
9:                 Append (w + w', p + p', v' ∪ u) to A[j].
10:            end if
11:        end if
12:    end for
13:    Remove dominated tuples from A[j].
14: end for
```

6.3. Algorithm analysis

Our algorithm for ThOP on cliques is shown below. The algorithm computes a path σ from s to t and a set S of items such that the sum of profits of the items is maximum, the sum of weights of the items is at most W, σ contains at most $T+1$ vertices (including s and t), and the items from S are located at the vertices in σ. Algorithm *BuildKnapsack* recovers the set of items from the selected path with maximum profit by using a similar approach as in Algorithm 3.

Algorithm 10 ThOPClique($G=(V,E), W, T, s, t, I$).

1: **Input:** Clique graph G, knapsack capacity W, time limit T, start vertex s, end vertex t, and item assignments I.
2: **Output:** An optimum solution for ThOP.
3: Index vertices and items as described in Section 3.
4: Let A be an empty clique profit table. Set $A[1]=(0,0,\emptyset)$.
5: **for** $i=2$ to $|V|-1$ **do**
6: Let u be the vertex with index i.
7: Call $UpdateCliqueProfitTables(W,T,A,u)$.
8: **end for**
9: Return $BuildKnapsack(A,I)$.

A similar inductive proof as to the proof of Lemma 1 shows that for each feasible subset S of S_z the clique profit table contains a tuple (w_S, p_S, v_S) (or a dominating tuple (w'_S, p'_S, v'_S)) at entry $A[z]$, where $w_S=\sum_{i_j\in S} w_j$, $p_S=\sum_{i_j\in S} p_j$, $|v_S|\leq T-1$, and v_S is the set of vertices that stores the items in S.

Theorem 6. *There is a FPTAS for ThOP when the graph G is a clique, every edge has length* 1*, and* $\mathcal{V}_{\min}=\mathcal{V}_{\max}$.

Proof. We modify Algorithm 10 so that it rounds down the profit of each item to the nearest multiple of $\frac{\epsilon P_{max}}{|I|}$. Note that each entry of the clique profit table contains at most T tuples with the same profit (one for each value of $|v|$), and the weights of the items do not need to be rounded up; therefore, each entry in the clique profit table has at most $O(P_r T)$ tuples, where P_r is the number of different values for the rounded profit of any subset of items and $T<n$ is the time limit. Hence, this modified Algorithm 10 has time complexity $O(\frac{|I|}{\epsilon}n^2)$ and so it is a FPTAS. □

CRediT authorship contribution statement

Andrew Bloch-Hansen: Writing – review & editing, Writing – original draft, Visualization, Methodology, Formal analysis. **Roberto Solis-Oba:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization. **Daniel R. Page:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization.

Funding

Andrew Bloch-Hansen and Roberto Solis-Oba were partially supported by the Natural Sciences and Engineering Research Council of Canada, grants 6636-548083-2020 and RGPIN-2020-06423, respectively.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Roberto Solis-Oba reports financial support was provided by Natural Sciences and Engineering Research Council of Canada. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Bloch-Hansen, D. Page, R. Solis-Oba, A polynomial-time approximation scheme for thief orienteering on directed acyclic graphs, in: International Workshop on Combinatorial Algorithms (IWOCA), Springer, 2023, pp. 87–98.
[2] A. Blum, S. Chawla, D. Karger, T. Lane, A. Meyerson, M. Minkoff, Approximation algorithms for orienteering and discounted-reward TSP, SIAM J. Comput. 37 (2) (2007) 653–670.
[3] M. Bonyadi, Z. Michalewicz, L. Barone, The travelling thief problem: the first step in the transition from theoretical problems to realistic problems, in: IEEE Congress on Evolutionary Computation (CEC), IEEE, 2013, pp. 1037–1044.
[4] J. Chagas, M. Wagner, Ants can orienteer a thief in their robbery, Oper. Res. Lett. 48 (6) (2020) 708–714.
[5] J. Chagas, M. Wagner, Efficiently solving the thief orienteering problem with a max-min ant colony optimization approach, Optim. Lett. 16 (8) (2022) 2313–2331.
[6] L. Faêda, A. Santos, A genetic algorithm for the thief orienteering problem, in: 2020 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2020, pp. 1–8.
[7] S. Fang, E. Lu, V. Tseng, Trip Recommendation with Multiple User Constraints by Integrating Point-of-Interests and Travel Packages, IEEE 15th International Conference on Mobile Data Management, vol. 1, IEEE, 2014, pp. 33–42.
[8] N. Freeman, B. Keskin, İ. Çapar, Attractive orienteering problem with proximity and timing interactions, Eur. J. Oper. Res. 266 (1) (2018) 354–370.

[9] B. Golden, L. Levy, R. Vohra, The orienteering problem, Nav. Res. Logist. 34 (3) (1987) 307–318.
[10] R. Karger, R. Motwani, G. Ramkumar, On Approximating the Longest Path in a Graph, Algorithmica 18 (1) (1997) 82–98, Springer.
[11] G. Konstantakopoulos, S. Gayialis, E. Kechagias, Vehicle routing problem and related algorithms for logistics distribution: a literature review and classification, Oper. Res. (2020) 1–30.
[12] C. Lin, K. Choy, G. Ho, S. Chung, H. Lam, Survey of green vehicle routing problem: past and future trends, Expert Syst. Appl. 41 (4) (2014) 1118–1138.
[13] F. Neumann, S. Polyakovskiy, M. Skutella, L. Stougie, J. Wu, A fully polynomial time approximation scheme for packing while traveling, in: International Symposium on Algorithmic Aspects of Cloud Computing, Springer, Cham, 2019, pp. 59–72.
[14] S. Polyakovskiy, F. Neumann, The packing while traveling problem, Eur. J. Oper. Res. 258 (2) (2017) 424–439.
[15] V. Roostapour, M. Pourhassan, F. Neumann, Analysis of baseline evolutionary algorithms for the packing while travelling problem, in: Proceedings of the 1th ACM/SIGEVO Conference on Foundations of Genetic Algorithms, 2019, pp. 124–132.
[16] A. Santos, J. Chagas, The thief orienteering problem: formulation and heuristic approaches, in: IEEE Congress on Evolutionary Computation (CEC), IEEE, 2018, pp. 1–9.
[17] P. Vansteenwegen, W. Souffriau, D. Van Oudheusden, The orienteering problem: a survey, Eur. J. Oper. Res. 209 (1) (2011) 1–10.