Scenic Routes Now: Efficiently Solving the Time-Dependent Arc Orienteering Problem

Ying Lu[†⋆], Gregor Jossé[‡⋆], Tobias Emrich[‡], Ugur Demiryurek[†]

Matthias Renz[#], Cyrus Shahabi[†], Matthias Schubert[‡]

[†]Integrated Media Systems Center, University of Southern California, Los Angeles, CA 90089, USA
[‡] LMU Munich, Oettingenstraße 67, 80583 Munich, Germany
[#] George Mason University, 4400 University Dr, Fairfax, VA 22030, USA
[†]{ylu720, demiryur, shahabi}@usc.edu, [‡]{josse, emrich, schubert}@dbs.ifi.lmu.de, [#]mrenz@gmu.edu

ABSTRACT

Due to the availability of large amounts of transportation (*e.g.*, road network sensor data) and transportation-related (*e.g.*, pollution, crime) data as well as the ubiquity of navigation systems, recent route planning techniques need to optimize for multiple criteria (*e.g.*, travel time, *utility/value* such as safety or attractiveness). In this paper, we introduce a novel problem called Twofold Time-Dependent Arc Orienteering Problem (2TD-AOP), which seeks to find a path from a source to a destination maximizing an accumulated value (*e.g.*, attractiveness of the path) while not exceeding a cost budget (*e.g.*, total travel time). 2TD-AOP has many applications in spatial crowdsourcing, real-time delivery, and online navigation systems (*e.g.*, safest path, most scenic path). Although 2TD-AOP can be framed as a variant of AOP, existing AOP approaches cannot solve 2TD-AOP accurately as they assume that travel-times and values of network edges are constant. However, in real-world travel-times and values are time-dependent, where the actual travel time and utility of an edge depend on the arrival time to that edge.

We first discuss the practicality of this novel problem by demonstrating the benefits of considering time-dependency, empirically. Subsequently, we show that optimal solutions are infeasible (NP-hard) and solutions to the static problem are often invalid (*i.e.*, exceed the cost budget). Therefore, we propose an efficient approximate solution with spatial pruning techniques, optimized for fast response on dynamic road networks where both travel times and values are time-dependent. Experiments on a large-scale, fine-grained, real-world road network demonstrate that our approach always produces valid paths, is orders of magnitude faster than any optimal solution with acceptable accumulated value.

CCS CONCEPTS

•Information systems →Query optimization; Location based services; Mobile information processing systems;

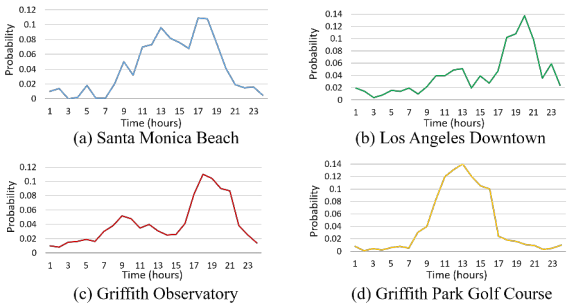
⋆These authors contributed equally to this work.

CIKM'17, November 6–10, 2017, Singapore.

 ISBN 978-1-4503-4918-5/17/11…$15.00
DOI: http://dx.doi.org/10.1145/3132847.3132874

1 INTRODUCTION

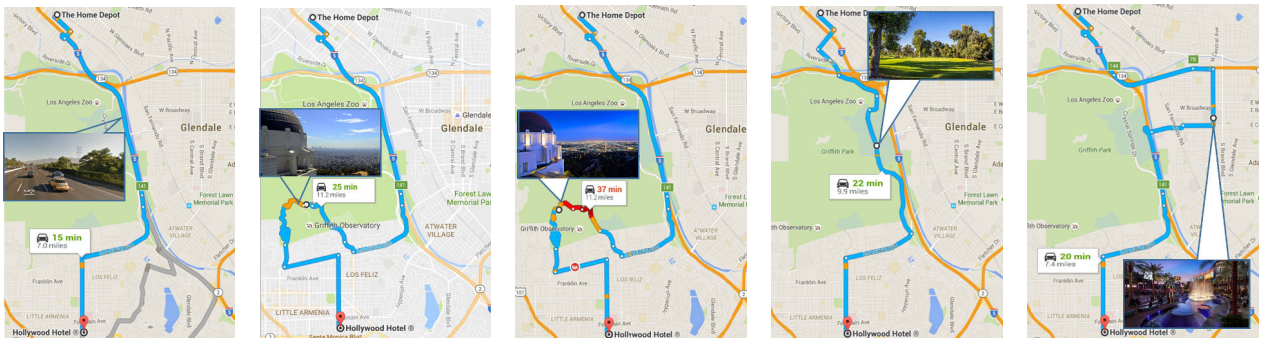
Not long ago, finding the shortest or fastest path in road networks was the main focus of route planning algorithms [11, 12]. Nowadays due to the availability of sensor data collected from traffic sensors, cell phones and in-car navigation systems novel path planning applications emerge. For example, navigation systems are being extended to take multiple criteria (*e.g.*, travel time or distance, *accuracy/value*[1] such as safety or attractiveness) cost functions into account to increase both driver's convenience and traffic efficiency. In practice, however, while one may prefer a utility rich (*e.g.*, most scenic) path, there is usually a limit on the budget (time or distance) that needs to be traded-off for this purpose. This problem can be formulated as a variant of the Arc Orienteering Problem (AOP) [28, 38], which finds a path from a source to a destination maximizing an accumulated value (*e.g.*, attractiveness of the path) while not exceeding a cost budget (*e.g.*, total travel time). While AOP seems similar to the shortest path problem, due to its budget restriction and its (utility) maximization objective (instead of minimization), it is proved to be an NP-hard combinatorial optimization problem [38]. Applications of AOP include finding the safest path [1] and finding the most scenic path [28, 30]. AOP also applies to spatial crowdsourcing [7, 25] where individuals complete tasks in order to collect rewards. A recent popular example of this application is the augmented reality game Pokémon Go. A worker who is willing to complete crowdsourced tasks wants to maximize the collected rewards.



(a) Santa Monica Beach (b) Los Angeles Downtown (c) Griffith Observatory (d) Griffith Park Golf Course

Figure 1: The distribution of the visiting probability over time at different locations. These distributions are derived from the geo-metadata of Flickr photos [29].

[1]The two terms are interchangeably used in this paper.



(a) Fastest path (b) AOP path (c) Invalid AOP path (06:00 pm) (d) 2TD-AOP path (01:00 pm) (e) 2TD-AOP path (06:00 pm)

Figure 2: Exemplary paths.

Until recently, AOP algorithms have only focused on solving the problem on networks with static travel costs and static utility functions. The static AOP approaches make the simplifying assumption that the travel time and utility function for each edge of the network are constant over time. However, in real-world the actual travel time and utility on a road segment heavily depend on the time of the day and, therefore, are time-dependent. The significance of time dependence was substantiated by its incorporation into many recent algorithms [4, 10, 11] and navigation services. The utility functions may also vary throughout the day, *e.g.*, a view of the sea might only be worthwhile during the day, whereas a view of the skyline might be particularly worthwhile at night. Fig. 1 shows the time-dependent attractiveness of some popular locations in Los Angeles. The same argument can be made for other utilities such as safety and Pokémon Go power levels that also vary over time.

Problem statement. In this paper, we generalize AOP to time-dependent travel times and time-dependent utilities, termed Twofold Time-Dependent AOP (2TD-AOP). Without loss of generality, we restrict to the application of scenic route planning. [2] With this use case, given a road network with time-dependent travel times and time-dependent attractiveness values, our goal is to find the most "scenic" path (*i.e.*, the path with the highest value) from a source to a destination while the travel time is within a given budget.

Existing variations [2, 3, 20] of AOP do not consider the time dependence of costs or values. Further, they are all evaluated on small graphs with hundreds of network nodes (see Sec 2.3). A recent work [28] introduces efficient speed-up techniques that scale and expedite the response time of AOP in large spatial networks but also focusing on static networks. Our proposed 2TD-AOP is the first time-dependent variant of AOP where both cost and value functions are time-dependent. In addition, we aim to solve 2TD-AOP on large-scale real-world road networks with fast responses in order to meet the requirements of mobile applications.

Challenge (1). By simply applying a solution to the static variant of the problem, such as the AOP approach proposed in [28], to a time-dependent dataset, we will end up generating very different paths, sometimes invalid (beyond the time budget) and sometimes suboptimal (low utility). To illustrate this, consider the example in Fig. 2 which shows different paths for a scenic path query with the same source and destination. Fig. 2(a) shows the time-dependent fastest path as computed by [10]. In this example, the fastest path does not pass through scenic areas (*e.g.*, Griffith Observatory). Now consider hourly value distributions of these two scenic locations as given in Fig. 1. At first glance it is evident that the value functions show great variance over the course of a day. If not allowing for time-dependent values, these functions are commonly averaged. Suppose a driver is willing to spend 25 minutes for his trip. Employing an algorithm for AOP in static networks [28], we obtain the static scenic path displayed in Fig. 2(b). The static travel time along an arc is set as the average of all travel times along that particular arc. The static scenic values are derived analogously. The result path has a static travel time of 25 minutes which abides by the budget. However, the actual travel time might of course deviate in either direction. For a desired departure at 1 pm, the time-dependent travel time coincides with the static time of 25 minutes, but if departing at 6 pm, the time-dependent path takes 37 minutes (see Fig. 2(c)). Thus, at 6 pm the result path clearly exceeds the budget and is therefore invalid. Similarly, the averaged scenic values can deviate from the time-dependent ones. While the static scenic path at 1 pm does not exceed the budget, it is not optimal either. Griffith Observatory, which it passes along the way, is a popular sight but particularly scenic during sunset or in the evening. Instead, the best solution at 1 pm is displayed in Fig. 2(d). This time-dependent scenic path takes 22 minutes and passes a golf course, offering particularly nice views at daytime. The best solution at 6 pm is shown in Fig. 2(e). This path takes 20 minutes and passes through central Glendale, a bustling area in the evening and at night.

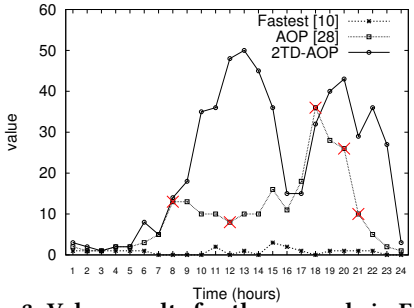


Figure 3: Value results for the example in Fig. 2.

Fig. 3 displays the (scenic path) values gained for the three different variations of the example in Fig. 2: Fastest [10], AOP [28] and our 2TD-AOP paths for different departure times. The Fastest approach considers the time dependence on travel time but not

[2] Other use cases may include finding the safest path [1] or the path with most Pokémon Go items.

for the values. The AOP approach is the static AOP that considers constant travel times and values on road segments. The 2TD-AOP approach (introduced in this paper) takes into account both the time-dependent travel times and time-dependent (attractiveness) values (see Sec. 7 for how time-dependent values are calculated). Two main observations can be made from this analysis. First, the value changes with (departure) time, and the change in value is significant. For instance, 2TD-AOP can achieve up to four times higher values than AOP during 11 am - 2 pm and 10 - 11 pm. Second, some of the paths computed in the static network are invalid (crossed-out in red in Fig. 3) in the time-dependent setting as the actual (time-dependent travel) time exceeds the budget. While there are cases that static AOP may seem to yield better value than that of 2TD-AOP (*e.g.*, at 6 pm), the total travel time exceeds the budget.

Challenge (2). The interplay of time dependence between utility and cost in 2TD-AOP renders it much harder than static AOP. In our first attempt to solve this problem, we extend the static AOP approach [28] by substituting the static shortest path algorithm with a time-dependent shortest path technique (*e.g.*, [10]) to find feasible arcs (*i.e.*, arcs to be inserted into the solution without exceeding the budget). And then utilize the correct time-dependent values to select promising feasible arcs as well. This baseline approach is rather inefficient (two orders of magnitude slower than our following proposed approach) as the time dependence in costs and values causes more expensive feasible arc identifying (see Sec 7.1).

Technical overview. To process 2TD-AOP efficiently, we propose an efficient metaheuristic approach which takes advantage of pruning techniques to limit the candidate set and reduce intermediate path computations (Sections 4–6). Specifically, to efficiently identify the feasible arcs, instead of expensively finding the time-dependent shortest paths from scratch for each arc/node at each iteration, we incrementally update the earliest arrival time and the latest departure time for each node, so that we can safely insert a node into the solution as long as its latest departure time is no less than its earliest arrival time. Additionally, to preserve the increasing trend for the solution value, we collect the time-dependent values along the paths with the earliest arrival time and the latest departure time, and use them to select potential arcs with high values and low detour costs.

We evaluate our algorithm on a large-scale real-world road network with more than 500K vertices and 1M arcs (note that this is the largest dataset found in the literature solving AOP and existing studies on AOP are mainly evaluated on small graphs with hundreds of vertices as explained more in Sec. 2). We show the performance and the quality (accuracy or value) of the results (*i.e.*, generated paths). Specifically, (1) compared to the optimal solution (NP-hard) that returns exact results, our approximate algorithm is up to four orders of magnitude faster while attaining the path quality to be 50% - 60%. For example, on the real-world dataset, the optimal solution needs 3 - 6 hours to answer a query but our algorithm takes only 1 - 2 seconds. (2) Compared to the baseline approach, our algorithm is also two orders of magnitude faster with almost the same path quality. (3) Compared to the static AOP approach, our algorithm returns paths with 3 to 4 times higher quality at the expense of at most 2x performance degradation. Note that, our algorithm can still return the results within 2 seconds on the large dataset.

2 RELATED WORK

We divide the related work into three groups. First, we review research on how to attain value functions, which is complementary to our work. Next, we briefly introduce related research from the database community. Finally, we present research on the family of OP/AOP which mostly stems from the field of Operations Research.

2.1 Attaining Value Functions

An important aspect of OP/AOP is how to attain the value functions, in particular when aiming to solve a time-dependent instance. Most studies from the field of Operations Research rely on specific benchmark datasets, some synthetic, others real. The authors of [34], for instance, evaluate their approach to cycle trip planning on real-world tourist data from Flanders, Belgium. As these approaches usually lack scalability, their required dataset is small. Crowdsourced data (*e.g.*, Twitter, Flickr) is a rich source of information. It is for instance used for general POI recommendation [24], or for determining opening hours [23]. Instead of focusing on how to attain value functions, in this paper, we assume the values are given. For our experimental evaluation, we derive these values from the Flickr data (see Sec. 7).

2.2 Trip Planning Queries

There are many studies on trip planning queries [8, 13, 22, 27, 32, 33] in the database community. But most of them are not congruent to the problem setting of 2TD-AOP. For example, the Trip Planning Query (TPQ) [8] (alternatively, Route Planning Query [6] or Route Search Query [27]). Given a set of POI categories, *e.g.*, "ATM", "restaurant", the result of a TPQ is the fastest path from a given source to a given destination visiting exactly one instance of each category. The most related query is the multi-preference (*i.e.*, travel time and utility such as safety or attractiveness) path search [13, 22, 33, 39]. To solve this type of queries, some studies [13, 22] find the scenic path by optimizing a single objective function which combines the attractiveness value and travel cost (*e.g.*, maximizing their difference [13]). Some other studies [33, 39] apply Pareto optimizations or skyline query techniques trying to maximize the value and also minimize the cost. However, in practice, while we may prefer a more attractive (or safer) path, there is usually a limit on how much deviation from the shortest/fastest path we can tolerate. Therefore, unlike these studies, 2TD-AOP's goal is to find the most scenic as long as the total travel time stays within a specified *budget*.

2.3 OP and AOP

This work is also related the classical orienteering problem (OP) where the difference from AOP is that values are not assigned to arc but to vertices. Although NP-hard [19], exact solutions to OP exist [14, 26, 31]. While in older research specific heuristic approaches were proposed [5, 36], the trend shifted towards algorithms following particular meta-heuristics such as Tabu Search [18], Genetic Algorithms [35] or Ant Colony Algorithms [37]. This holds similarly for AOP where exact algorithms [9, 38] are rare but available. Generating approximative solutions is the standard approach [17], most effectively following the meta-heuristics Iterative Local Search (ILS) [38] and Greedy Randomized Adaptive Search Procedure (GRASP) [34]. Recently, ILS and GRASP have been improved to solve AOP on large road networks in near-interactive time [28].

OP with time-dependent (TD-OP) costs has gained attention recently [15, 16, 21, 37]. However, all of them allow the travel times of arcs are time dependent but assume values are static. Numerous variations [2, 3, 20] of AOP do not consider the time dependence for costs or values. For instance, Team AOP [20] extends AOP to multiple vehicles. Capacitated AOP [2] constraints the capacity of each vehicle. Nevertheless, to the best of our knowledge, there is no time-dependent variants of AOP in the literature.

The above studies are all evaluated on small graphs (except [28]) with hundreds of vertices. On such small graphs, the shortest path pre-computation between each pair of vertices is usually used to solve OP/AOP. However, this is not feasible for real-world road networks, which are large-scale (*e.g.*, millions of vertices/arcs) and time-dependent, as there are numerous fastest paths from a given vertex to another, depending on the departure time.

3 PROBLEM DEFINITION

We model the road network as a graph $G = (V, A, val, tt)$, where V denotes the set of vertices (or nodes), $A \subseteq V \times V$ denotes the set ofdirected arcs (or edges), val and tt denote thenon-negative functions mapping each arc onto its respective value and travel time: $val_{i,j}(t) := val(v_i, v_j, t) \geq 0$, $tt_{i,j}(t) := tt(v_i, v_j, t) \geq 0$.

We represent a path as a sequence of arcs. A path from a source v_0 to a destination v_N departing at t_0 is denoted by$p = ((v_0, v_1), (v_1, v_2), \ldots, (v_{N-1}, v_N), t_0)$. The travel time (or cost) of the path p is defined as $tt(p) = \sum_{i=0,\ldots,N-1} tt_{i,i+1}(t_i)$, where t_i denotes the time of arrival at (and departure from) vertex v_i. t_i is dependent on the travel timesalong the preceding arcs and is defined iteratively: $t_i := \sum_{j=0}^{i-1} tt_{j,j+1}(t_j)$. The value of the path p is the total value collected from the arcs in the path: $val(p) = \sum_{i=0,\ldots,N-1} val_{i,i+1}(t_i)$.

Definition 3.1 (2TD-AOP). Given a graph G, a source vertex v_0, a destination vertex v_N, a departure time t_0, and a time budget b, the **Twofold Time-Dependent AOP (2TD-AOP)** finds a path p from t_0 to v_N such that its travel time $tt(p)$ is no more than the budget b and its value $val(p)$ is maximum.

We say a path p is a *feasible* path if its travel time $tt(p)$ is no more than the budget b. Among all feasible paths, the *optimal* path is the path with the maximum value. In this paper, we allow multiple visits of a vertex/arc in the path. However, the value of an arc is not collected twice in the solution.

4 SOLVING THE 2TD-AOP

In this section, we present terminology and insights essential to our algorithm for solving 2TD-AOP.

Definition 4.1 (earliest arrival). Given a graph G, a source v_0, a vertex v_i and a departure time t_0, we define the *earliest arrival (time)* of reaching v_i from v_0 when departing at t_0, denoted by $ea_{0,i}$ (ea_i for short), as follows:

$$ea_i := t_0 + \min_{p \in \mathcal{P}(v_0 \to v_i)} tt(p)$$

Definition 4.2 (latest departure). Given a graph G, a destination v_N, a vertex v_i and a latest arrival time $t_0 + b$, we define the *latest departure (time)* of reaching v_N from v_i no later than $t_0 + b$, denoted by $ld_{i,N}(t_0 + b)$ (ld_i for short), as follows:

$$ld_i := t_0 + b - \min_{p \in \mathcal{P}(v_i \to v_N)} tt(p)$$

Definition 4.3 (forward-reachable). Given a graph G, a source v_0, a departure time t_0, and a time budget b, we refer to the vertices reachable within the budget b from v_0 (when departing at t_0) as *forward-reachable* vertices, denoted by $\mathrm{FWR}_0(b)$.

$$\mathrm{FWR}_0(b) := \{v_i \in V \mid ea_i \leq t_0 + b\}$$

Definition 4.4 (backward-reachable). Given a graph G, a destination v_N, a departure time t_0, and a budget b, we refer to the vertices from which v_N can be reached within b when departing no earlier than t_0 as *backward-reachable* vertices, denoted by $\mathrm{BWR}_N(b)$.

$$\mathrm{BWR}_N(b) := \{v_i \in V \mid ld_i \geq t_0\}$$

Intuitively, all forward-reachable vertices lie in a quasi-circular region around the source. This region is similar to the circular expansion of a Dijkstra-search. The difference is that in this case the cost criterion is time-dependent travel time. Analogously, all backward-reachable vertices lie in a quasi-circular region around the destination. This graphical intuition is visualized in Fig. **??**. If the destination is not forward-reachable from the source, then the query has no answer.

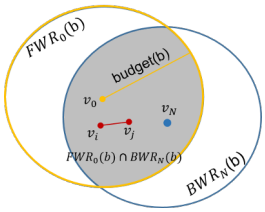


Figure 4: $\mathrm{FWR}_0(b)$ and $\mathrm{BWR}_N(b)$ and their intersection.

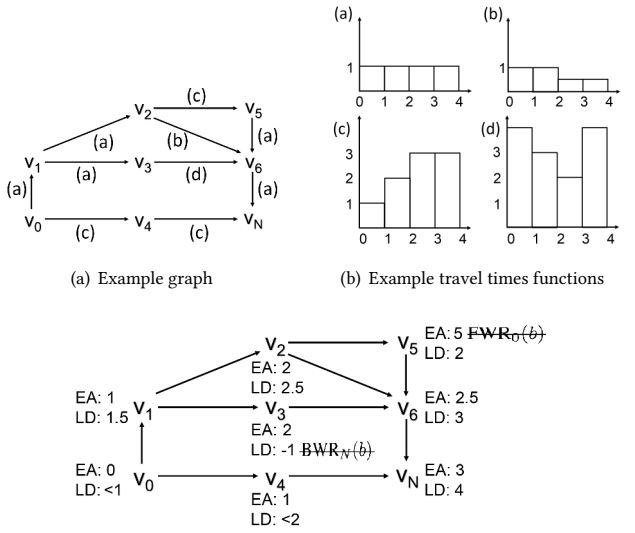
Our solution to 2TD-AOP is based on the observation: For an arc (v_m, v_n) to be part of a feasible path, it is a necessary condition that v_m is forward-reachable from the source v_0 and v_n is backward-reachable from the destination v_N. Hence, if an arc is not contained in the intersection of the forward-reachability and the backward-reachability regions, it cannot be part of a feasible path (see Lemma 4.5). In our solution to 2TD-AOP, arcs which increase the value are recursively inserted until the budget is exhausted. Hence, the reachability regions iteratively contract until no more arcs can be inserted.

Lemma 4.5. *All vertices along all feasible paths are contained in the intersection of the sets of forward-reachable and backward-reachable vertices.*

Proof. Suppose there exists a feasible path p visiting a vertex $v_i \notin \mathrm{FWR}_0(b) \cap \mathrm{BWR}_N(b)$. Case 1: $v_i \notin \mathrm{FWR}_0(b)$, *i.e.*, $ea_i > t_0 + b$, meaning that the subpath from v_0 to v_i already exceeds the budget, which contradicts the assumption that p is feasible. Case 2: if $v_i \notin \mathrm{BWR}_N(b)$, then $ld_i < t_0$, hence p is not feasible. □

Fig. 5(a) shows an example of a graph with source v_0 and destination v_N. The arcs in Fig. 5(a) are labeled with letters which correspond to the travel time functions in Fig. 5(b). Let the departure time $t_0 = 0$, and the time budget $b = 4$. Fig. 5(c) shows the earliest arrivals and latest departures according to this query. For example, v_5 is not forward-reachable as $ea_5 = 0 + tt((v_0, v_1), (v_1, v_2), (v_2, v_5), 0) = 1 + 1 + 3 = 5 > b$. There are two feasible paths given below from v_0 to v_N, and the former is the optimal path.

$$((v_0, v_4), (v_4, v_N), 0) = 1 + 2 = 3$$
$$((v_0, v_1), (v_1, v_2), (v_2, v_6), (v_6, v_N), 0) = 1 + 1 + 0.5 + 1 = 3.5$$



(a) Example graph

(b) Example travel times functions

(c) Earliest arrivals and latest departures

Figure 5: Example graph for a query with source v_0, destination v_N, departure time 0 and time budget $b = 4$.

Let us turn to the computation of latest departures. For example, $ld_6 = 0 + 4 - 1 = 3$, implying that in order to arrive at v_N no later than $t_0 + b$, one must depart at time 3. Now, consider ld_4: the only path to the destination follows the arc (v_4, v_N). Departing at time 3 results in exceeding the budget, since $tt_{4,N}(3) = 3$. The same holds for time 2. However, when departing at time $t < 2$, then $tt_{4,N}(t) \leq 2$, and thus it arrivals within the budget b. Hence, in order to compute the latest departure time, a linear scan among the time windows might be required.

Algorithm 1: FWR $(G, v_0, t_0, t_0 + b)$

```
1  Q ← empty queue of vertices v_i sorted asc. by ea_i;
2  result = ∅;
3  ea_0 = t_0;
4  eaval_00 = 0;
5  Q.add(v_0);
6  while Q is not empty do
7      v_i ← Q.removeFirst;
8      for (v_i, v_j) ∈ G.A do
9          êa_j ← ea_i + tt_{i,j}(ea_i);
10         êaval_j ← eaval_j + val_{i,j}(ea_i);
11         if êa_j < t_0 + b then
12             if ¬ result.contains(v_j) then
13                 ea_j ← êa_j;
14                 eaval_0j ← êaval_0j;
15                 Q.add(v_j);
16                 result.add(v_j);
17             else
18                 if êa_j < ea_j then
19                     ea_j ← êa_j;
20                     eaval_0j ← êaval_0j;
21 return result
```

Algorithm 2: BWR $(G, v_N, t_0, t_0 + b)$

```
1  Q ← empty queue of vertices v_i sorted desc. by ld_i;
2  result = ∅;
3  ld_N = t_0 + b;
4  ldval_N = 0;
5  Q.add(v_N);
6  while Q is not empty do
7      v_j ← Q.removeFirst;
8      for (v_i, v_j) ∈ G.A do
9          τ_k ← arg max_{τ_l} {τ_l | τ_l < ld_j};
10         while τ_k + tt_{i,j}(τ_k) > ld_j do
11             k ← k − 1;
12         l̂d_i ← ld_j − tt_{i,j}(τ_k);
13         ldv̂al_i ← ldval_j − val_{i,j}(τ_k);
14         if l̂d_i ≥ t_0 then
15             if ¬ result.contains(v_i) then
16                 ld_i ← l̂d_i;
17                 ldval_i ← ldv̂al_i;
18                 Q.add(v_i);
19                 result.add(v_i);
20             else
21                 if l̂d_i > ld_i then
22                     ld_i ← l̂d_i;
23                     ldval_i ← ldv̂al_i;
24 return result;
```

5 COMPUTING REACHABILITY

In this section, we go into the details of the reachable vertex computation. Both sets $\text{FWR}_0(b)$ and $\text{BWR}_N(b)$ are generated by expansions around source and destination with "time-radius b". According to Lemma 4.5, all the vertices along all the feasible paths are contained in the intersection of the two reachability regions.

In order to compute the set of forward-reachable vertices, we perform a modified time-dependent Dijkstra algorithm, called FWR and illustrated in Algorithm 1. The earliest arrival times of all the vertices are initialized as ∞ except the source v_0 for which it is the departure time t_0. We maintain a priority queue containing the visited vertices sorted in ascending order by their earliest arrival times. At each iteration, the top vertex v_i with the smallest arrival time in the queue is dequeued, and its outgoing arcs are explored (Lines 8–20). Specifically, for each neighbor v_j of v_i, the travel time from v_i to v_j is added to the earliest arrival time $\hat{ea}_j$ of v_j (Line 9). If v_j is forward-reachable and was previously not visited, ea_j is set accordingly and v_j is added to the queue as well as the result set. If v_j was previously visited and the updated earliest arrival time is better than the previous, it is updated accordingly. v_j is ignored if it is not forward-reachable. This is equivalent to the Dijkstra property which ensures that any processed vertex is reached by the shortest path, in this case the fastest time-dependent path. Additionally, while computing ea_j, we additionally collect the value $eaval_j$ along the fastest path reaching v_j from v_0 at the earliest time (Lines 4, 10, 14 and 20). These collected values can be used later for selecting the promising arcs to update solutions.

The set of backward-reachable vertices is computed in a similar fashion but slightly more complicated. The procedure is presented

in Algorithm 2 and called BWR. In addition to the result set containing all backward-reachable vertices, we maintain a queue of vertices sorted in descending order by ld_i. Initially, the latest departure at the destination v_N is set to the latest possible arrival time, $t_0 + b$. The algorithm operates backwards from the destination, following incoming arcs which are explored in the `for`-loop spanning lines 8-23. As was illustrated in the example in Fig. 5, in order to find the latest departure, it is required to linearly scan the time windows. When exploring an incoming arc from v_i to v_j, τ_k is initially set to the latest time window not exceeding the latest departure from v_j. If at τ_k the travel time from v_i to v_j is too high (*i.e.*, v_j cannot be reached in time for its latest departure), k is decremented implying an earlier departure at v_i but possibly also a different travel time along the arc. Once a valid latest departure time for v_i is found, ld_i is set accordingly. If v_i was previously visited, ld_i is updated if it is later than the previous latest departure. Similar to FWR, while computing ld_i in BWR, the value $ldval_i$ along the fastest path departing from each vertex v_i at ld_i to v_N is also collected additionally (Lines 4, 13, 17 and 23). Upon termination, all vertices which are backward-reachable are in the result set and labeled with their respective latest departure.

6 HEURISTIC SOLUTION

Due to the NP-hardness of the problem, we forgo giving an optimal solution as it would be time consuming for large graphs (optimal approaches take hours per query, see Sec. 7). Instead, we propose a heuristic algorithm to 2TD-AOP. So far no other solution to 2TD-AOP exists. At the high level, our algorithm recursively fills the gaps between query source v_0 and destination v_N with arcs in order to improve the value while not exceeding the budget, until the budget is exhausted. The pseudo-code of the algorithm is presented in Algorithm 3, referred as RecInsert.

RecInsert maintains a list of gaps gaps that holds vertex pairs between which arcs are to be inserted. Each gap, denoted by $(v_i \rightsquigarrow v_j)$, is the time-dependent shortest path from v_i to v_j. The solution path is concatenated by the gaps in gaps. First of all, we will examine whether $(v_0 \rightsquigarrow v_N)$ is within the budget b. If no, it returns no solution; otherwise, gaps will be initialized with $(v_0 \rightsquigarrow v_N)$, and then it steps into the procedure of determining which gap in the solution should be perturbed and which arc should be selected to insert into the gap such that the solution path has the highest value improvement while keeping the travel time within the budget (Lines 2–16). Before inserting an arc into a gap, we compute the set G'_{ij} of arcs that can be *feasibly inserted*, *i.e.*, without exceeding the budget (Lemma 4.5, Lines 6–9). This is crucial to reduce the search space. Gap and arc selections (among the feasible arcs) are based on a heuristic. Intuitively, if the new path after being inserted with an arc (v_m, v_n) between a gap $(v_i \rightsquigarrow v_j)$ gains a higher value and detours with a lower cost, then we say the arc w.r.t the gap has a higher potential to improve the solution. The "potential" is evaluated by the quotient of the value gain val_{gain} and the detour cost $cost_{detour}$, and both of them can be estimated by the earliest arrivals and latest departures. As shown in Fig. 6, $cost_{detour}$ is estimated by $\left[(ea_m - t_i) + tt_{m,n}(ea_m) + (b + t_i - ld_n) - (v_i \rightsquigarrow v_j).cost\right]$ (Line 10), where t_i is the departure time from v_i, and val_{gain} is estimated by $\left[eaval_{i,m} + val_{m,n}(ea_m) + ldval_{n,j} - (v_i \rightsquigarrow v_j).val\right]$ (Line 11).

Algorithm 3: RecInsert(G, gaps, b)

```
1  G' ← ∅;
2  bestArc ← (null, null);
3  bestGap ← (null, null);
4  highestPotential ← 0;
5  for (v_i ⇝ v_j) ∈ gaps do
6      FWR_i ← FWR(G, v_i, b);
7      BWR_j ← BWR(G, v_j, b);
8      G'_ij ← FWR_i ∩ BWR_j;
9      for (v_m, v_n) in G'_ij: ea_m + tt_{m,n}(ea_m) < ld_n do
10         cost_detour ← b + ea_m + tt_{m,n}(ea_m) − ld_n − (v_i ⇝ v_j).cost;
11         val_gain ← eaval_{i,m} + val_{m,n}(ea_m) + ldval_{n,j} − (v_i ⇝ v_j).val;
12         potential ← val_gain / cost_detour;
13         if potential > highestPotential then
14             highestPotential ← potential;
15             bestArc ← (v_m, v_n);
16             bestGap ← (v_i ⇝ v_j);
17     G' ← G' ∪ G'_ij;
18 if bestArc = null then
19     return;        // no value gain or no remaining budget
20 else
21     Update gaps by inserting bestArc into bestGap;
22     Let b_rmng be the remaining cost of the updated solution;
23     RecInsert(G', gaps, b_rmng);
```

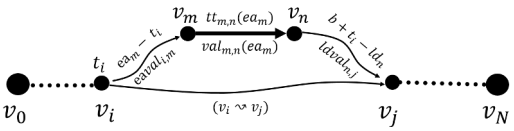


Figure 6: Potential Criteria Illustration

Note that we provide tight estimations for the actual travel time and value of the new path: from v_i to v_m, the travel time and collected value are the actual ones along the time-dependent fastest path departing at t_i; while from v_n to v_j, they are approximated with the ones along a path departing at time ld_n. Further, these estimations are calculated efficiently, which are based on the earliest arrivals, latest departures and values already collected during FWR and BWR without computing the actual fastest paths from scratch. The arc with the best potential is recursively inserted until no qualified arc can be inserted into any of the gaps (Lines 18–23).

The dominate component in Algorithm 3 is the feasible arc computing in FWR and BWR at each iteration. Its time complexity is $O(S * (E + D \log D))$, where S is the number of gaps in the solution, and E (resp. D) is the number of arcs (resp. vertices) in the search space. The initial search space is the entire graph. To reduce the search space, we proceed to present three crucial pruning techniques.

6.1 Pruning Strategies

Let us discuss three variations which are omitted in the algorithm description for reasons of brevity. (*i*), it is recommended to employ a simple forward estimation during FWR. (*ii*), it is recommended to compute the intersection of FWR_i and BWR_j during computation of BWR_j. (*iii*), it is possible to make the recursive call with a

restricted portion of the graph. Variations (*i*) and (*ii*) are based on the following observation. The sets FWR_i and BWR_j are not needed separately, only their intersection is needed. Therefore, we may exclude vertices which are not contained in the intersection during computation of the respective sets.

Regarding (*i*): We propose to employ a simple forward estimation for pruning. If information about the speed limits is available, the following lower bound can be used with negligible overhead. During FWR, in line 11, the `if`-clause may be tightened:

$$ea_j + ed_{jN}/ms < t_0 + b$$

where ed_{jN} is the Euclidean distance from v_j to the destination v_N and ms is maximum speed in the given network. If traveling from v_j to v_N at speed ms exceeds the budget, so will any actual path. Any such vertex cannot be forward-reachable. We refer to this pruning strategy as *FWEST* pruning. At marginal computational cost, FWEST yields a perceptible reduction in candidates during FWR (see Sec. 7.1). Of course, the method holds analogously for BWR and can equally be applied.

Regarding (*ii*): One may use the restriction $ea_i \leq ld_i$ for each vertex at each iteration in both FWR and BWR to reduce the search space. This follows from the definition:

$$\begin{aligned} ea_i > ld_i \quad &\Leftrightarrow \quad t_0 + tt(p^\star_{0i}) > t_0 + b - tt(p^\star_{iN}) \\ &\Leftrightarrow \quad tt(p^\star_{0i}) + tt(p^\star_{iN}) > b \end{aligned}$$

where $p^\star_{ij}$ denotes the fastest path from v_i to v_j at ea_i. Hence, the `if`-clause in line 14 of BWR can be tightened to $ld_i \geq ea_i$. By construction, ea_i is based on an actual path expansion and therefore serves as a tighter bound for BWR than any estimation. Meanwhile, the `if`-clause in line 11 of FWR can also be tightened to $ea_j \leq ld^*_j$, where ld^*_j is the latest departure time of v_j calculated in the previous iteration. We refer to this pruning strategy as EALD pruning. In this case, the simple forward estimation cannot be applied because $ea_i \geq t_0 \frac{ed_{0i}}{ms} \geq t_0$. Thus, instead of computing both reachability regions separately and subsequently their intersection, the information of both FWR and BWR may be used reciprocally. This limits the search space and implicitly computes the intersection. Unvisited vertices are either not forward-reachable or backward-reachable. The effect of this pruning strategy is remarkable. Particularly when taking into account that it causes no computational overhead.

Regarding (*iii*): It is possible to issue the recursive call of RecInsert with a restricted portion of the graph. In Algorithm 3, we obtain the union G_{next} of each feasible arc set G'_{ij}, and use/inherit it as the candidate feasible arc set for the next arc insertion:

$$G_{\text{next}} = \bigcup \{G'_{ij} \mid (v_i, v_j) \in \text{gaps}\}$$

We refer this mechanism as Inherit. Clearly, G_{next} will not miss the feasible arcs in the next recursion (or next iteration). Moreover, according to Lemma 6.1, Inherit can reduce the search space for computing the feasible arcs. The intuition is that, as established in Lemma 4.5, all vertices along feasible paths are in the intersection of the reachability regions, and this property holds recursively.

Lemma 6.1. *In Algorithm 3,* $G_{next} \subseteq G$.

Proof. For any an arc $a(v_m, v_n) \in G_{next}$, the solution path $p = ((v_0, \ldots, v_i \rightsquigarrow (v_m, v_n) \rightsquigarrow v_j \ldots, v_N), t_0)$ after inserting a is feasible, *i.e.*, its travel time $tt(p) \leq b_{rmng}$. As $b_{rmng} \leq b$, $tt(p) \leq b$, and thus $a(v_m, v_n) \in G$. Hence Lemma 6.1 holds. □

7 EXPERIMENTAL EVALUATION

We evaluate our solution to 2TD-AOP on a time-dependent road network of Los Angeles, CA, USA [11] which contains about 500K vertices and 1M arcs. The travel time functions in this network are derived from large-scale and high-resolution (both spatial and temporal) sensor data (both live and historic) from different transportation authorities in California. The step length of the piecewise constant travel time functions is 5 minutes. Based on the data, streets are uncongested between 9 pm and 6 am. Therefore, travel times are assumed to be static during this interval.

The time-dependent value functions are derived from a geo-tagged set of Flickr photos [29] consisting of 217,391 photos in Los Angeles. For each arc (v_i, v_j) the value $val_{i,j}(t)$ is the number of photos which have (v_i, v_j) as their nearest arc (but not further than one kilometer) and which were taken in the hour-interval that contains t. Hence, the step length of the piecewise constant value functions is 60 minutes. Fig. 7 displays the network and the distribution of non-zero value arcs. Each arc which has positive value in some time window is visualized as a colored shape. Shape and color depend on the time interval in which the value of the arc is the highest. For example, an arc with highest value between 6:00 and 7:00 pm will be displayed as red circle. Table 1 presents the absolute and relative numbers of arcs with non-zero value. The number of photos taken during the day is about three times higher than during night.

In our experiments, we arrange paths into six time buckets. We randomly draw source and destination vertices from the network and compute the respective time-dependent fastest paths using [10]. If the travel time of a path is k minutes $\pm\varepsilon$, its source and destination pair is assigned to the k minutes time bucket for $k \in \{10, 20, 30, 40, 50, 60\}$. In general, the travel time of the fastest path between two locations in a city is within one hour. Each time bucket consists of twenty source and destination pairs. Since travel times are empirically static at night and value peaks less frequent at night, we randomly sample departure times from the interval 8 am to 8 pm. We make a case for twofold time dependence, therefore, a network with significant time-dependent impact on travel time and value is essential. Our standard experimental setting has the following inputs. The standard query set is the 20 minutes time bucket, the standard query budget is 200% (*i.e.*, 40 minutes), and departure times are randomly drawn from 8 am to 8 pm. Deviating values are explicitly mentioned. For each query, the departure time is the same across time-dependent algorithms.

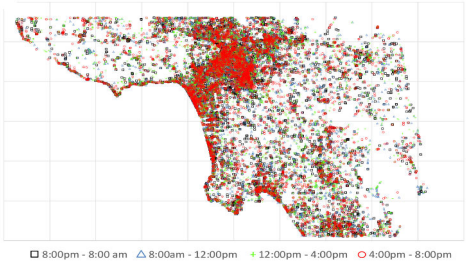


Figure 7: Value arcs in experimental road network of Los Angeles. Each value arc corresponds to a colored shape which indicates the arc's value peak interval.

peak interval	# of arcs	% of all arcs
8:00 am - 12:00 pm	6,238	0.57%
12:00 pm - 4:00 pm	7,366	0.67%
4:00 pm - 8:00 pm	5,851	0.53%
8:00 pm - 8:00 am	6,417	0.59%
all intervals	25,872	2.38%

Table 1: Statistics about arcs with non-zero value.

We omit providing results of the MIP formulation given in Sec. 3 generated by a complex solver. This is due to the exorbitant computational requirements. Even for small instances with at most 100 vertices, solving a less complex MIP often requires days [21, 37]. However, for comparison, we compute optimal solutions with a custom dynamic programming method.

7.1 Experimental Results

In the first set of experiments, we compare the value of result paths generated by different algorithms taking varying degrees of information into account. As in our introductory example, we compare 2TD-AOP paths computed by RecInsert with time-dependent fastest paths and static scenic paths. To generate the time-dependent fastest paths, we use the solution proposed in [10] which relies on hierarchical routing and forward estimations to conduct bidirectional path computation. We refer to the results to this algorithm as Fastest. For the static paths, we use the AOP solution presented in [28] which employs an Iterative Local Search approach combined with spatial pruning techniques. In accordance with the metaheuristic, it generates an initial solution which is subsequently improved by inserted arcs and perturbed by deleting arcs. The insertion follows a heuristic, the deletion is pseudo-random. We refer to solutions created by this approach as AOP paths and call this algorithm AOP-ILS. AOP operates on a network with static travel time and static value which are both derived by averaging the respective time-dependent functions. Note that AOP queries are given the same time budget but require no departure time. The results generated by RecInsert are referred to as 2TD-AOP paths. We also implemented the baseline approach (Baseline) that extends the AOP solution [28] by substituting the static shortest path method with the time-dependent shortest path algorithm [10] to find feasible arcs and then, in-tandem, utilized the correct time-dependent values to find 2TD-AOP paths. Baseline algorithm generates slightly (no more than 10%) higher values than 2TD-AOP as it can have marginally more accurate travel costs and values for arc selection. However, it needs to expensively compute the time-dependent shortest paths from scratch for each arc at each iteration. As we expected, Baseline is significantly slower than both 2TD-AOP and AOP by two orders of magnitude (see Fig. 10), and hence we omit it in the aftermentioned experiments.

Fig. 8 shows the value for the different approaches. Fastest paths serve as a baseline which shows how much value can be attained "by chance" as they do not optimize for value. Indeed, Fastest paths collect negligible value. Since the non-zero value arcs are rather scarce, this is not surprising. 2TD-AOP and AOP, in contrast, optimize for value. It can be observed that on average 2TD-AOP generates three times the value of AOP. This gap widens for increasing path length.

Fig. 9 illuminates the effect of the query budget. We observe that with increasing budget, 2TD-AOP paths generate significantly more value than that of AOP solutions (3–4 times higher). This is particularly noteworthy as the problem becomes more complex with increasing budget. A higher budget allows for greater detours which in turn increases the search space. RecInsert leverages this effect to create better solutions. In contrast, AOP results barely improve. Therefore, albeit increasing complexity, considering time dependency is able to enhance results considerably.

As we mentioned, static paths may prove invalid when considering time-dependent travel time. The percentage of AOP which are invalid increases from 25% to almost 50% across the increasing time buckets. Figures 8 and 9 show the values of all AOP paths, both valid and invalid, as there is no significant difference. However, taking into account the percentage of invalid results and the low overall value, static results still cannot compete with twofold time-dependent results (they obtain 20%–25% less values).

In terms of general complexity, 2TD-AOP is considerably more intricate than AOP. While both problems are NP-hard, the interplay of time dependence between value and cost in 2TD-AOP renders it much harder than AOP. In light of these challenges, the efficiency of RecInsert is more surprising. Fig. 10 shows the processing times of RecInsert solving the 2TD-AOP and AOP-ILS solving AOP. 2TD-AOP solutions can be computed in less than twice the processing time of AOP solution, usually under two seconds. The increase in processing time for both algorithms is linear in the travel time of the fastest path. Taking the result quality into consideration, the additional processing time of RecInsert compensates for the often invalid AOP paths with less value. It should also be noted that in an application where fixed response times are required, it would be possible to terminate the arc insertion in RecInsert prematurely and construct an early result path.

Fig. 11 evaluates our proposed pruning strategies. It shows the percentage of visited vertices with pruning relative to the number

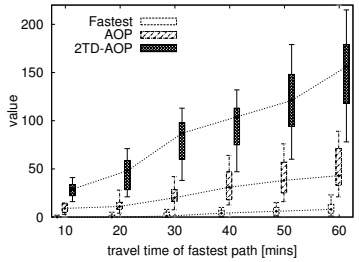


Figure 8: Varying fastest path lengths.

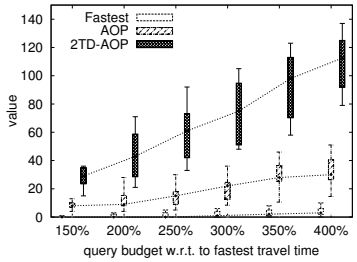


Figure 9: Varying time budget.

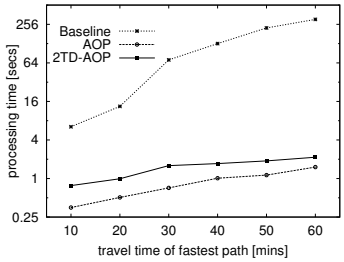
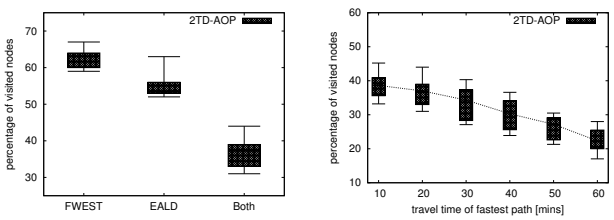
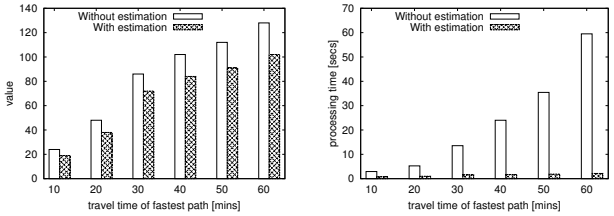


Figure 10: Processing time for varying path lengths (log2-scale)



(a) FWEST and EALD (b) Inherit

Figure 11: Evaluation of pruning techniques



(a) value (b) Processing time

Figure 12: Evaluation on potential computation with estimations

of visited vertices when employing no pruning. In this case, all vertices in FWR and BWR have to be visited. In Sec. 6 we proposed three pruning techniques for RecInsert, FWEST, EALD and Inherit.

(1) Fig. 11(a) shows the effects of FWEST and EALD pruning techniques separately as well as their combination with Inherit technique is applied. When only employing FWEST (during both FWR and BWR), almost 40% of the vertices can be pruned. EALD prunes about 45% of the vertices. As FWEST causes negligible and EALD causes no computational overhead, we recommended to employ both. Our experiments validates this fact that combining both approaches, 65% of all vertices in FWR and BWR can be pruned. FWEST is a basic forward estimation which can be applied during FWR and BWR. Since EALD is superior to FWEST, when both are available, FWEST is obsolete during BWR. However, EALD cannot be applied during BWR at the first iteration, thus it provides the best performance when both are available. In Fig. 11, we show the effect of each pruning technique separately as well as the impact of combining both pruning techniques.

(2) Fig. 11(b) shows the benefit of Inherit by varying the time bucket. Here FWEST and EALD techniques are applied. We observe that with Inherit, it visits up to 40% less vertices since it could reduce the search space of finding the feasible arcs without computing feasible arcs from the entire graph at each iteration (*i.e.*, for each arc insertion). Further, we note that as the time bucket increases, the less percentage of nodes it accesses since the more arcs to be inserted and thus the more iterations will be involved.

Fig. 12 evaluates the performance of the estimation method for the potential computation. As illustrated in Fig. 6, when we calculate the potential of an arc (v_m, v_n) to be inserted into a gap $(v_i \rightsquigarrow v_j)$, the travel time and collected value from v_n to v_j are approximated with the ones along a path departing at time ld_n (not the actual departure time). The comparison method (referred as "without estimation") calculates the actual time-dependent shortest path [10] from v_n to v_j when computing arcs' potentials. We observe that with estimations, the value declines (less than 20%, see Fig. 12(a)) but the processing time (up to 20×, see Fig. 12(b)) speeds up drastically. This is because latest departures are available (already computed during BWR) and they provide tight estimates.

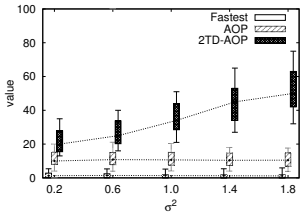


Figure 13: Vary arc values

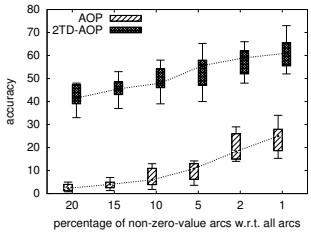


Figure 14: Vary value arc density

Fig. 13 evaluates the effect of the variances of arc values over time. In this set of experiments, we generate values for each arc following a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, where μ is the average of the time-dependent values calculated based on the Flickr photos [29], and σ^2 is the variance. As shown in Fig. 13, as σ^2 increases, *i.e.*, the more variances in values for each arc, 2TD-AOP paths can obtain more values comparing to the competitors because RecInsert can select the promising arcs depending on the arrival time.

Finally, we compare 2TD-AOP and AOP paths to optimal solutions in terms of their value. The optimal solutions are computed with a custom brute force approach using a depth-first search and several pruning techniques (*e.g.*, FWEST). Any naïve algorithm, much like a complex solver, would not be able to compute solutions in feasible time. Our custom approach takes between 3 to 6 hours per optimal path computation for the standard query setting. Even for minor instances where the fastest path takes 5 minutes and the budget is 10 minutes, computing an optimal solution takes between 1 and 2 hours. Therefore, generating optimal solutions to 2TD-AOP is infeasible. This is irrespective of the application. Users will not tolerate response times in the hours, and precomputation is not possible in time-dependent networks. Fig. 14 compares the accuracy of 2TD-AOP paths generated by RecInsert and of AOP paths generated by AOP-ILS. Values are shown relative to values of optimal paths. Fig. 14 also illustrates the effect that the density of non-zero value arcs has on the result. The value is given relative to the optimal value for different networks when varying the percentage of non-zero value arcs. In our original network about 2% of the arcs have non-zero value during some time window. For this experiment, we randomly copied value functions to arcs with zero value to increase density to 1%, 5%, 10%, 15% and 20% of all arcs. In addition, we set some of the value functions to zero, generating a network with 1% value arcs. As both algorithms follow a heuristic during arc insertion, they are more easily sidetracked in a network with many value arcs. When the density of value arcs decreases, the employed heuristics prove effective. In this case, RecInsert result paths attain between 50% and 60% accuracy. The static solutions, in contrast, hardly exceed 25%. This establishes the superiority of 2TD-AOP over AOP solutions. While computing an optimal solution takes up to 6 hours, RecInsert takes about 2 seconds. Thus, RecInsert produces paths with about 50% accuracy in 10^{-5} the processing time. As 2TD-AOP cannot be expected to be solved optimally in efficient time, RecInsert yields a promising trade-off.

In conclusion, our experimental evaluation clearly demonstrates the importance of twofold time-dependence. Solutions to 2TD-AOP

are superior to solutions of static AOP. For increasingly complex query settings, the superiority is more obvious. Furthermore, RecInsert computes solutions that achieve 50% - 60% accuracy compared to optimal results within seconds. The efficiency can essentially be attributed to the pruning techniques employed in RecInsert. As of now, no time-dependent AOP has been evaluated on large-scale real-world road networks. We prove that even in such networks, the highly complex 2TD-AOP can be feasibly solved.

8 CONCLUSIONS

We introduced the Twofold Time-Dependent Arc Orienteering Problem (2TD-AOP), a novel extension of the family of Orienteering Problems (OP). The goal of 2TD-AOP is to find a path from a given source to a given destination within a given time budget which, additionally, maximizes the value collected along the way. In comparison to static versions of this problem, 2TD-AOP allows for both travel times and value functions to be time-dependent. The incorporation of time-dependent values is a novel extension which has not been studied before. The importance of time-dependent values is showcased and experimentally substantiated. Due to the NP-hardness and the twofold time dependence, we provide a heuristic approximation for solving 2TD-AOP. Our approach is evaluated on a large-scale real-world road network. Time-dependent results gain significantly more value than static results, showing the importance of twofold time dependence empirically. Our algorithm produces result paths with 50% - 60% accuracy in value accumulation as compared to optimal, where optimal solutions are impractical.

9 ACKNOWLEDGMENT

This research has been funded by NSF grants IIS-1320149, CNS-1461963 and the USC Integrated Media Systems Center. This work has also been partially supported by DAAD Germany (PPP, project ID: 57052426). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any of the sponsors.

REFERENCES

[1] S. Aljubayrin, J. Qi, C.S. Jensen, R. Zhang, Z. He, and Z. Wen. 2015. The safest path via safe zones. In *Proc. of ICDE*. 531 – 542.
[2] Claudia Archetti, Dominique Feillet, Alain Hertz, and M. Grazia Speranza. 2010. The undirected capacitated arc routing problem with profits. *Computers & Operations Research* 37, 11 (2010), 1860–1869.
[3] C. Archetti and M G. Speranza. 2013. Arc routing problems with profits. *Arc Routing: Problems, Methods, and Applications, MOS-SIAM Series on Optimization* (2013), 257–284.
[4] G. Veit Batz, Daniel D., Peter S., and C. Vetter. 2009. Time-dependent Contraction Hierarchies. In *Proc. of Meet. on Algo. Eng. & Exp.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 97–105.
[5] I-M. Chao, B. L Golden, and E. A Wasil. 1996. A fast and effective heuristic for the orienteering problem. *EJOR* 88, 3 (1996), 475–489.
[6] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. 2008. The Multi-Rule Partial Sequenced Route Query. In *Proc. of ACM GIS*. 10:1–10:10.
[7] Z. Chen, R. Fu, Z. Zhao, Z. Liu, L. Xia, L. Chen, P. Cheng, C. C. Cao, Y. Tong, and C. J. Zhang. 2014. gmission: A general spatial crowdsourcing platform. *Proc. of VLDB* 7, 13 (2014), 1629–1632.
[8] D. Cheng, M. Hadjieleftheriou, G. Kollios, F. Li, and S.-H. Teng. 2005. On Trip Planning Queries in Spatial Databases. In *Proc. of SSTD*. 273–290.
[9] R. Deitch and S. P Ladany. 2000. The one-period bus touring problem: Solved by an effective heuristic for the orienteering tour problem and improvement algorithm. *EJOR* 127, 1 (2000), 69–77.
[10] U. Demiryurek, F. Banaei-Kashani, C. Shahabi, and A. Ranganathan. 2011. Online computation of fastest path in time-dependent spatial networks. In *Intl. Symp. on Spatial and Temporal Databases*. 92–111.
[11] U. Demiryurek, F. Banaei Kashani, and C. Shahabi. 2010. A case for time-dependent shortest path computation in spatial networks. In *Proc. of 18th Intl. Conf. on ACM SIGSPATIAL / GIS*. 474–477.
[12] B. Ding, J. X. Yu, and L. Qin. 2008. Finding Time-dependent Shortest Paths over Large Graphs. In *Proc. of the 11th Intl. Conf. on EDBT*. 205–216.
[13] Dominique Feillet, Pierre Dejax, and Michel Gendreau. 2005. The Profitable Arc Tour Problem: Solution with a Branch-and-Price Algorithm. *Transportation Science* 39, 4 (Nov. 2005), 539–552.
[14] D. Feillet, P. Dejax, and M. Gendreau. 2005. Traveling salesman problems with profits. *Transportation science* 39, 2 (2005), 188–205.
[15] A. Garcia, M. T. Linaza, O. Arbelaitz, and P. Vansteenwegen. 2009. Intelligent routing system for a personalised electronic tourist guide. *Info. and Comm. Tech. in Tourism* (2009), 185–197.
[16] D. Gavalas, C. Konstantopoulos, K. Mastakas, G. Pantziou, and N. Vathis. 2014. Efficient Heuristics for the Time Dependent Team Orienteering Problem with Time Windows. In *Intl. Conf. on Applied Algorithms, ICAA*. 152–163.
[17] D. Gavalas, C. Konstantopoulos, K. Mastakas, G. Pantziou, and N. Vathis. 2015. Approximation algorithms for the arc orienteering problem. *Inform. Process. Lett.* 115, 2 (2015), 313–315.
[18] M. Gendreau, G. Laporte, and F. Semet. 1998. A tabu search heuristic for the undirected selective travelling salesman problem. *EJOR* 106, 2 (1998), 539–545.
[19] B. L Golden, L. Levy, and R. Vohra. 1987. The orienteering problem. *Naval research logistics* 34, 3 (1987), 307–318.
[20] A. Gunawan, H.-C. Lau, and P. Vansteenwegen. 2016. Orienteering Problem: A survey of recent variants, solution approaches and applications. *EJOR* (2016).
[21] A. Gunawan, Z. Yuan, and H. C. Lau. 2014. A Mathematical Model and Metaheuristics for Time Dependent Orienteering Problem. In *Proc. of Practice and Theory of Automated Timetabling*. 202–217.
[22] Abdeltawab M. Hendawi, Aqeel Rustum, Amr A. Ahmadain, David Hazel, Ankur Teredesai, Dev Oliver, Mohamed Ali, and John A. Stankovic. 2017. Smart Personalized Routing For Smart Cities. In *IEEE ICDE*.
[23] H.-P. Hsieh and C.T. Li. 2014. Mining and Planning Time-aware Routes from Check-in Data. In *Proc. of the 23rd ACM CIKM*. 481–490.
[24] H.-P. Hsieh, C.-T. Li, and S.-D. Lin. 2012. Exploiting Large-scale Check-in Data to Recommend Time-sensitive Routes. In *SIGKDD Workshop UrbComp*. 55–62.
[25] L. Kazemi and C. Shahabi. 2012. Geocrowd: enabling query answering with spatial crowdsourcing. In *Proc. of ACM SIGSPATIAL / GIS*. 189–198.
[26] G. Laporte and S. Martello. 1990. The selective travelling salesman problem. *Discrete applied mathematics* 26, 2-3 (1990), 193–207.
[27] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv. 2009. An interactive approach to route search. In *Proc. of ACM SIGSPATIAL / GIS*. 408–411.
[28] Y. Lu and C. Shahabi. 2015. An arc orienteering algorithm to find the most scenic path on a large-scale road network. In *ACM SIGSPATIAL/GIS*. 46:1–46:10.
[29] Hatem Mo.-S., D. Watzinger, B. Huber, M. Döller, E. Egyed-Zsigmond, and H. Kosch. 2014. World-wide Scale Geotagged Image Dataset for Automatic Image Annotation and Reverse Geotagging. In *ACM Multimedia Systems*. 47–52.
[30] Daniele Quercia, Rossano Schifanella, and Luca Maria Aiello. 2014. The Shortest Path to Happiness: Recommending Beautiful, Quiet, and Happy Routes in the City. In *ACM Hypertext and Social Media*. 116–125.
[31] R. Ramesh and K. M Brown. 1991. An efficient four-phase heuristic for the generalized orienteering problem. *Comput. Oper. Res.* 18, 2 (1991), 151–165.
[32] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. 2008. The Optimal Sequenced Route Query. *VLDB* 17, 4 (2008), 765–787.
[33] G. Skoumas, K. Arthur S., G. Jossé, M. Schubert, M. A. Nascimento, A. Züfle, M.s Renz, and D. Pfoser. 2015. Knowledge-Enriched Route Computation. In *Advances in Spatial and Temporal Databases, SSTD*. 157–176.
[34] W. Souffriau, P. Vansteenwegen, G. V. Berghe, and D.-V. Oudheusden. 2011. The planning of cycle trips in the province of East Flanders. *Omega* 39, 2 (2011), 209–213.
[35] M F. Tasgetiren. 2001. A genetic algorithm with an adaptive penalty function for the orienteering problem. *JESR* 4, 2 (2001), 1–26.
[36] T. Tsiligirides. 1984. Heuristic methods applied to orienteering. *Journal of the Operational Research Society* 35, 9 (1984), 797–809.
[37] C. Verbeeck, K. Sörensen, E.-H. Aghezzaf, and P. Vansteenwegen. 2014. A fast solution method for the time-dependent orienteering problem. *EJOR* 236, 2 (2014), 419–432.
[38] C. Verbeeck, P. Vansteenwegen, and E.-H. Aghezzaf. 2014. An extension of the arc orienteering problem and its application to cycle trip planning. *Transp. Res. Part E: Logis. Transp. Review* 68 (2014), 64 – 78.
[39] Bin Yang, Chenjuan Guo, Yu Ma, and Christian S. Jensen. 2015. Toward Personalized, Context-aware Routing. *The VLDB Journal* 24, 2 (April 2015), 297–318.