Efficient Stochastic Routing in PAth-CEntric Uncertain Road Networks.

Master's thesis - Georgi Andonov - Group dpw1010f18 - Computer Science (IT)

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science(IT)
Aalborg University
http://www.aau.dk

Title:
Efficient Stochastic Routing in PAth-CEntric Uncertain Road Networks.

Theme:
Scientific Theme

Project Period:
Spring Semester 2018

Project Group:
dpw1010f18

Participant(s):
Georgi Andonov

Supervisor(s):
Bin Yang

Copies: 1

Page Numbers: 17

Date of Completion:
June 8, 2018

Project Summary

This project investigates the Shortest path with on time arrival probability (SPOTAR) problem which tries to find an a priori path that maximizes the on time arrival probability in destination within a predefined time budget. We apply the Path Centric (PACE) uncertain road network model to the SPOTAR solution. PACE maintain joint distributions not only for edges but also for paths. This aims to improve in comparison with the classical model which maintain travel time only for edges and uses convolution of probability distributions to find the distribution of a paths. Using convolution, could discard the joint dependency between the edges of a path and drops the quality of the SPOTAR solution. We propose an A* based algorithm for computing a SPOTAR path within a PACE uncertain road network model where travel times are maintained not only for edges but also for paths. The algorithm uses heuristic to prune the search space which is discovered, hence it minimizes the search space resulting in a speedup. Different heuristic functions have been examined. Euclidean distance to destination is referred as a baseline approach (BA), minimum travel time to destination using shortest path (SP) and Arc-Potentials. All the heuristics are able to be used with the proposed A* search. In order to compute SP heuristic, first we reverse the graph by reversing all edges, second we run a shortest path tree computation from destination vertex in the reversed graph. Finally, all vertices are labeled during the shortest path tree computation with their minimum travel time to destination.

We study the Stochastic on time arrival problem (SOTA). SOTA tries to find a routing policy that maximizes the probability of arriving on time at destination within a predefined time budget. The solution of the SOTA problem tells us at each vertex which is the next optimal adjacent vertex to follow and what is the associated probability of following this vertex to destination within a time budget. This information can be used to compute more advanced heuristic functions in comparisons with BA and SP heuristics. In order to compute SOTA problem for some destination and a time budget one has to first find the order of vertices for which the SOTA policy must be computed. We proposed an recursive procedure that explores the graph similarly to a depth-first search while it generates an update graph which holds the sequence of vertices for which the optimal policy must be computed for some time. Once the update graph is generated, we extract a topologically sorted array of pairs of the form (vertex,time). Next, we reverse the topologically sorted array the topologically sorted array and we obtain the update sequence. We propose an algorithm that computes the SOTA policy for a given source-destination pair and some time. The algorithm uses dynamic programming. We consider each vertex for which the SOTA policy to be computed for some time to be a sub problem. We use a bottom-up procedure since we sort our subproblems at the beginning. The SOTA policy is computed only for vertices and times which have not been computed before. If a specific policy to some destination for some time has been computed before, we reuse the computation

instead of recalculating it again. We furthermore improve the accuracy of the SOTA by applying the PACE model to the computation. Because the SOTA problem provides an natural upper bound for the search, we can stop the search once we find destination vertex. This is in contrast with the naive heuristic approaches, i.e. BA and SP which continue to explore the search space because nothing can guarantee that there isn't a better solution .

To this end we propose an efficient method that can guide the search towards a destination be using the SOTA policy. Unfortunately, computing and maintaining the SOTA policy can require prohibitively large amount of space. To contest with this problem Arc-Potential is proposed. By using Arc -Potentials lower memory consumption can be achieved without sacrificing the running time of queries. Arc-Potentials partition the graph into destination regions and label each edge of the graph with an array of size equal to the size of destination regions. Each entry contains the minimum travel time needed by the edge to become a part of an optimal policy to a corresponding destination region. This information is then used to prune the search space and speed up the search. We conduct experiments to show useful inside about the proposed method. For the experiments we varied the time budget and the distance between source-destination pairs. We perform experiments with different number of destination regions in order to show that Arc Potentials can be used to lower the space consumption without affecting the run time.

Contents

Efficient Stochastic Routing in PAth-CEntric Uncertain Road Networks.

Abstract—**We investigate the area of stochastic path finding which we apply to the PAth-CEntric (PACE) uncertain road network model. PACE maintain uncertain travel time for all edges and the most popular paths in a graph. By using trajectories, we instantiate and maintain uncertain travel times for edges and paths which aims to exploit the dependences in a road network. This increases the accuracy of the uncertain travel cost distribution estimation under the PACE model in comparison with the traditional edge based model where uncertain travel times are maintained only for edges but not for paths.**

We aim to find the shortest path with on-time arrival reliability (SPOTAR) under the PACE road network model. SPOTAR aims at finding an a priori path that has the highest probability of reaching destination from source within a predefined time budget. We propose algorithm for solving SPOTAR problem that uses different heuristic functions to speed up the search by pruning the explored search space. In this article, we shows how Arc-Potentials could be used as heuristic for our A* based search.

Arc-Potentials is an efficient precomputation technique, a stochastic version of Arc-Flags. To generate Arc-Potentials we first need to solve the Stochastic on time arrive problem (SOTA). SOTA problem tries to find the most reliable path between source and destination within a predefined time budget. It uses an optimal policy that can tell which is the next optimal node to follow on each junction based on already realized travel time. We adopt the PACE road network model to the computation of the SOTA policy, then we use this policy to generate Arc-Potentials.

We conduct series of using PACE model in the experiments showing that Arc-Potential heuristic performs well in different settings. We instantiate PACE road network model based on real wold trajectory data.

I. Introduction

The growing volume of spatially and temporally related data can be harnessed to extend and improve existing routing services making possible reduction in greenhouse gas emission, traffic congestions as well as reduction in travel time. As a consequence of this information growth, new road network models have been proposed and established, allowing for better and reliable routing. Stochastic path finding uses the classical uncertain road network model where each link is labeled by a uncertain travel time represented as a probability function. Although, the classical uncertain road network model provides framework for solving problems which involve probability distributions as edge weights, it also lacks accuracy because it discards the dependencies between the edges in the network, disallowing for accurate joint distribution estimations. To solve this, the PAth-CEntric road network model has been proposed [2], [3]. The model can be seen as an extension of the classical 'edge' based uncertain road network model.

In this paper, we study the SPOTAR problem which tries to find an a priori path that maximizes the on-time arrival probability within a predefined time budget. SPOATR has been studied previously in literature [4], [5]. The authors considered two constraints regarding SPOAR. First, independence between the edge's uncertain travel times, and second, uncertain travel times are maintained only for edges but not for a sequence of edges i.e. paths.

Recently, PACE uncertain road network model has been successfully integrated within SPOTAR [1], the solution is an A* search which finds a priori the most reliable path form source to destination within a predefined time budget. It exploits trajectory data to derive cost distributions of paths which are then used by the algorithm. Previously, the algorithm has been tested against a few heuristics [1]. The best performed one is the minimal travel time to destination as an upper bound of the search. We refer this heuristic as (SP) heuristic. Euclidean distance to destination was also used as heuristic, we refer to it as a baseline approach(BA). BA was completely outperformed by the SP heuristic as showed [1], .

[1] show that, minimum travel time heuristic performs well but it is consider 'too loose', meaning that it is not likely that only minimum travel time will be realized traveling from source to destination. In this article, we propose a better heuristic function which can outperform the SP heuristic, i.e. Arc-Potentials. Arc-Potential is a powerful preprocessing technique which can speed up the queries in graph by pruning edges considered not relevant. In order to derive an Arc-Potentials, the Stochastic on-time arrival problem (SOTA) must be solved first. SOTA tries to find an optimal policy for each vertex in a graph toward a destination. If a travel is located in a vertex, the policy tells us which is the next optimal adjacent vertex to follow considering the already realized travel time, this is the vertex that maximizes the on-time arrival probability. We adopt the PACA uncertain road network model to the SOTA policy computation increasing the accuracy of the method. Naively implemented Arc-Potentials will be calculated for all destinations which is going to consume a lot of space. We show how only subset of all destination vertices (called boundary vertices) could be used to speed up the preprocessing time. Additionally, Arc-Potentials can optimized the space consumption on the prize of slowing down the running time. By partition the graph with more finer partition, more space is consumed and queries are fast. By using more coarser partition, less space is required but queries are slow. This suggest that by using Arc-Potentials space can be trade for time and the opposite.

Contribution To the best of our knowledge, this is the first paper that utilizes the SOTA policy to be used within the PACE uncertain road network model which exploits the unchristian travel time dependency in a graph. We show how to use the SOTA policy to generate Arc-Potentials and used them as a heuristic in the algorithm for solving SPOTAR. We conduct series of experiments in order to provide useful inside of the quality and the performance of the proposed Arc-Potentials heuristic function.

II. Preliminaries

We use G to denote a directed graph that represents a road network, formally $G = (V, E)$, where V is a set of vertices i.e. road network intersections and $E \subseteq V \times V$ is a set of edges i.e. road network segments. The number of vertices in the graph G is $m = |V|$, and the number of edges in G is $n = |E|$. Each edge in the graph G has a tail and a head. For example in Figure 1, the edge e_1's head is a vertex v_s and e_1's tail is vertex v_e ,therefore we can also denote edge e_1 with (v_s, v_e). A model of a road network represented as a directed graph can be seen in Figure 1. It consists of 6 vertices and 9 edges.

A path has been defined as a sequence of adjacent vertices $P = \langle v_1, v_2, \ldots, v_g\rangle$. A sub-path P' of a path P is a subsequence of vertices from path P i.e. $P' = \langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j\rangle$, where $1 \leq i < j \leq g \leq n$. We consider the vertices that form a path P to be unique, which implies the following two constraints. First, $v_i \neq v_j$, if $1 \leq i, j \leq n$ and $i \neq j$, and second, the number of vertices in a path must be greater than one i.e. $n \geq 1$.

We consider two road network models, the classic edge-centric uncertain road network model, i.e. $EDGE$, and the path-centric uncertain road network model, i.e. $PACE$. They both are based on the preliminaries defined in Section II. The difference between the two models is whether or not independence between the edges in the road network has been considered. $PACE$ and $EDGE$ road network models have been previously studied in [1], [2], [3].

A. *Edge-centric uncertain road network model (EDGE)*

The edge centric uncertain road network model maintain travel cost for each edge. This can be done by using a function $W : E \longmapsto TC$ that accept as input an edge $e_i \in E$ and returns the associated uncertain travel cost TC. The edge-centric model is considered to be the classical road network model in stochastic routing [4], [5], [6], [7], [8], [9], [16].

In order to instantiate travel cost, trajectories have been used. We collect travel cost information from all trajectories that are traversing a particular edge, hence we are able to derive travel cost value for all edges that have been traversed by any trajectories. If an edge has not been traversed by any trajectory, speed limits have been used to derived a travel cost. When the travel cost is defined as travel time, then we can divide the length of an edge by the speed limit of the same edge which returns a travel time associated with the edge.

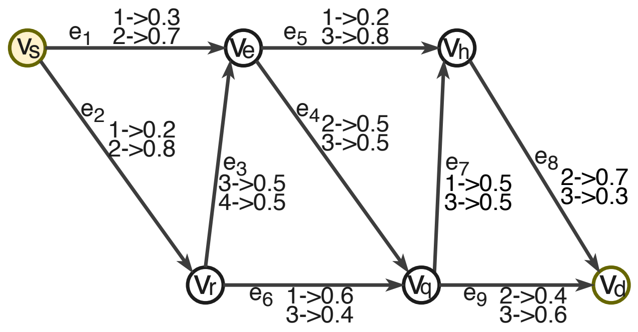


Fig. 1: Road Network and Uncertain Edge Weights

Table I summarizes information about 140 trajectories. 100 trajectories are traversing path $\langle e_1, e_4\rangle$ and 140 trajectories path $\langle e_4\rangle$. Based on this information we can instantiate travel cost distribution for edge e_1 and e_4. Since there are 100 trajectories traversing edge e_1, 30 trajectories took 1 mins, and 70 trajectories took 2 mins. Therefore we can instantiate $W(e_1) = \{(1, 0.3), (2, 0.7)\}$ as shown in Figure 1. We perform the same procedure to instantiate travel cost distribution for edge e_4. There are 140 trajectories which traverse edge e_4. 70 trajectories took 2 mins and 70 trajectories took 3 mins. Based on this we instantiate $W(e_4) = \{(2, 0.5), (3, 0.5)\}$ as shown in Figure 1.

Traversed Path	Costs on edges	# of trajectories
$\langle e_1, e_4\rangle$	1, 2	30
$\langle e_1, e_4\rangle$	2, 3	70
$\langle e_4\rangle$	2	40

TABLE I: Trajectory Examples

In the classical edge-based road network model convolution has been used to derive a travel cost distribution of paths [4], [5], [6], [7], [8], [9], [16]. Convolution of probability distribution assumes that the random variable which have to be convolve are independent, which is the case in the classical edge-based road network model. Unfortunately, this approach lacks accuracy because it ignores the dependencies among the edges in a path. If we consider independence between edges in our representation, we calculate the cost distribution of a path by first computing the joint distribution of the path and second deriving travel cost form the joint distribution. This process has been shown in Table II, considering path $P_{e_1,e_4} = \langle e_1, e_4\rangle$.

B. *Path-centric uncertain road network model (PACE)*

In addition to the travel cost distribution maintain by function W in the $EDGE$ based model, the $PACE$ road network model also maintain joint distribution of paths. Therefore, in the $EDGE$ based model, function W has to accept a path as an input and has to output the associated travel cost. Formally,

e_1	e_4	Probability
1	2	0.15
1	3	0.15
2	2	0.35
2	3	0.35

(a) Joint Travel Time Distribution

$P_{e1,e4}$	Probability
3	0.15
4	0.55
5	0.35

(b) Total Travel Time Distribution

TABLE II: Joint vs. Total Travel Time Distribution

we define function W for the $PATH$ based road network model as follows: $W : P \longmapsto TC$, where P is some path in G and TC is the travel cost of path P.

Considering the example in Figure 1 additionally to the travel costs $W(e_1)$ and $W(e_4)$, we also maintain $W(P_{e_1,e_4})$. Trajectories are used to directly derive joint distribution of paths. For example if we consider Table I, we can instantiate joint distribution of path P_{e_1,e_4}, the results are shown in Table III.

e_1	e_4	Probability
1	2	0.3
2	3	0.7

(a) Joint Travel Time Distribution

P_{e_1,e_4}	Probability
3	0.3
5	0.7

(b) Total Travel Time Distribution

TABLE III: Joint Distributions in PACE

The number of possible paths in a road network can be significantly large, it grows exponentially in the number of edges in the network. Therefore, we can not afford to maintain all paths in a road network. Usually, a parameter b is used to control the number of paths which are maintained as explained in [2]. The parameter specifies the minimum number of trajectories that have to traverse a path in order to maintain this path. Hight value of b means less paths, while low values of b mans that more paths will be maintained.

Next, we show by example, how to compute joint distribution of a path in PACE road network model. The graph in Figure 1 is used as a uncertain road network in the example. In this example travel time distributions are maintained by function W for paths $P_{e_2,e_6} = \langle e_2, e_6 \rangle$ and $P_{e_6,e_9} = \langle e_6, e_9 \rangle$, i.e. $W(P_{e_2,e_6})$ and $W(P_{e_6,e_9})$ respectively. We are interested in finding the joint distribution of path $P_{e_2,e_6,e_9} = \langle e_2, e_6, e_9 \rangle$. There are multiple path compositions which cover path $P_{e2,e6,e9}$. The first one is $\{W(e_2), W(P_{e_4,e_9})\}$, the second is $\{W(P_{e_2,e_6}), W(e_9)\}$ and the third is $\{W(P_{e_2,e_6}), W(P_{e_6,e_9})\}$. It has been proven that the composition with the longest overlap (sub-path) shared by the paths in the composition, provides the most accurate uncertain travel time estimation [2], [3]. This composition has been referred as the *coarsest* composition. In our example, we identify the coarsest composition to be $\{W(P_{e_2,e_6}), W(P_{e_6,e_9})\}$, since it has the longest sub-path $\langle e_6 \rangle$.

Let the coarsest composition $P_1, P_2, ...P_c$ is identified for a query path P, then the joint distribution of P is computed according to Equation 1

$$prob(P) = \frac{\Pi_{i=1}^{c} W(P_i)}{\Pi_{i=1}^{c-1} W(P_i \cap P_{i+1})} \tag{1}$$

In Equation 1, $P_i \cap P_{i+1}$ denotes the overlapped path of path P_i and path P_{i+1}. We use Equation 1 in our example to calculate the distribution of path $P = \langle e_2, e_6, e_9 \rangle$ as follows: $prob(P_{e_2,e_6,e_9}) = \frac{W(P_{e_2,e_6}) \cdot W(P_{e_6,e_9})}{W(\langle e_6 \rangle)}$, where the overlapping sub-path is $P_{e_2,e_6} \cap P_{e_6,e_9} = \langle e_6 \rangle$.

Problem Definition: Find the path P^* that starts in vertex $v_s \in V$, ends in vertex $v_d \in V$ and results in the maximum probability of reaching the destination v_d within a time budget T. P^* is formally defined in Equation 2, where $Path$ is the set of all paths that starts in vertex v_s and ends in vertex v_d and $P.TravelTime$ is the travel time of path P.

$$P^* = \arg\max_{P \in Path} Prob(P.TravelTime \leq T) \tag{2}$$

We also have to ensure that we find a solution of the problem efficiently.

III. Related work

We start by distinguishing two types of road network models, the classical which assumes that uncertain travel times are assigned and maintained only for edges, discarding possible uncertain travel time dependencies among consecutive edges which form a path. The classical model has been widely used in the literature [4], [5], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. It was suggested, that convolution of probability distribution can be used to derive a travel time distribution of paths. Convolution of probability distribution discard any dependency between the random variables which are convolved , hence resulting in inaccurate uncertain travel time estimation. To challenge this, the PACE uncertain road network model has been proposed [2], [3]. The model exploits trajectory data in order to derive and maintain travel cost distributions not only for edges but also for sequence of edges i.e. paths.

Next, we distinguish two categories in the domain of stochastic path finding, namely, the Shortest path with on-time arrival reliability problem (SPOTAR) and the Stochastic on time arrival problem (SOTA) [4], [5], [6]. SPOTAR aims at finding an a priori path that maximizes the on-time arrival probability. SOTA problem considers a vehicle that keeps on moving and has to be rerouted based on its current location considering the already realized travel time. This means that if a vehicle is located at a junction, the SOTA policy will tell which is the next optimal edge to follow, i.e. the edge that will maximizes the on-time arrival probability at destination. SOTA problem is usually formalized as a dynamic programing problem and then can be solved with a dynamic programing algorithm [4], [5], [6] In order to achieve speed up of the queries in deterministic path finding, preprocessing techniques such as Arc-Flags [17], Contraction hierarchy [20], Transit node routing [21] and Reach based routing [18], [18] have been used. Stochastic variants of Arc-Flags and Reach [5] have been previously used with the the SOTA problem. The main disadvantage of stochastic version of Arc-Flags and Reach is the significant amount of space which they consumed. This can be explain with the fact that they require storing arc flags for all time budgets up to some time. To content with the large amount of space which has been used, a new preprocessing techniques has been proposed, i.e. Arc-Potential. It can be applied to stochastic path finding and it can speed up the query time over no preprocessed network with an order of magnitude.

IV. Proposal for solution

We present an efficient algorithm for solving the Shortest path with on-time arrival reliability (SPOTAR) problem under the PACE road network model. The proposed algorithm is an A* search that uses heuristic to speed up the search by decreasing the search space discovered by the algorithm.

A. SOTA

Previous work on the SPOTAR problem identifies *minimum travel time with shortest path tree heuristic* i.e. SP as shown in [1]. The idea is to label all the vertices of a graph with the minimum travel time to destination. This has been achieved by running a backward Dijkstra search from the destination vertex in the reversed graph G_{rev} of G and label all nodes with their minimum travel time to destination. Only the nodes which are visited by the shortest path tree computation are labeled with a positive travel time, all unvisited nodes are labeled with negative travel time. Based on that, all nodes with negative labels can be discarded by the algorithm for solving SPOTAR. The minimum travel time heuristic performs well but it is highly unlikely that only the minimum travel time has been realized during a journey. *Euclidean distance to destination*,i.e. (BA) have been also used as a upper bound heuristic but it was outperformed by (SP).

In this article we investigate the Stochastic on-time arrival problem (SOTA) which we adopt as heuristic for our proposal for solution. Pre-processing of the graph is in the core of SOTA. The SOTA policy is obtained by a pre-computation of the graph and has been used to speed up the query time. The solution of the SOTA problem provides an optimal policy for each vertex in the graph towards a destination. The optimal policy of a vertex tells us which is the next optimal vertex to follow on the path to destination, as well as the probability of following this vertex. By following the optimal policy we maximizes the probability of arriving at destination within a predefined time budget.

Let v_i be a vertex such that $v_i \in V$. We define $u_{v_i v_d}(t)$ to be the maximum probability of reaching the destination vertex

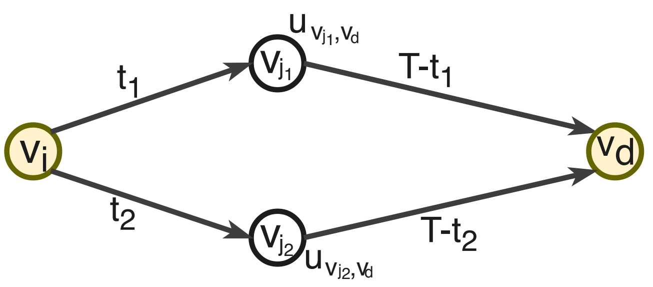


Fig. 2: SOTA policy and the time spent while traveling.

v_d from vertex v_i for a time budget $t \leq T$, by following the optimal policy to destination v_d. We define $v_j = f_{v_i v_d}(t)$ to return the vertex that has to be visited next when traveler is located at vertex v_i and by following the optimal policy towards the destination vertex v_d for a time budget $t \leq T$. This means that at each vertex v_i the traveler must pick edge (v_i, v_j) that maximizes the on time arrival probability to destination v_d within time t. As shown in Figure 2, since the traveler can spent at most time t traveling along edge (v_i, v_j) with probability $p_{v_i,v_j}(t)$, then the remaining time $T - t$ is spent traveling from vertex v_j to destination vertex v_d with probability $u_{v_j,v_d}(T-t)$ where the initial time budget is equal to T.

The SOTA problem has been defined in discrete time with the system of Equations 3, 4, which are using convolution of probability distributions. Equation 3 calculates the probability of following the optimal policy from a given vertex v_i towards the destination v_d.

$$\begin{aligned} u_{v_i v_d}(t) &= \max_{v_j:(v_i,v_j)\in E} \sum_{0}^{T} p_{v_i v_j}(t).u_{v_j v_d}(T-t) \\ &\qquad \forall v_i \in V, v_i \neq v_d, 0 \leq t \leq T \\ u_{v_d v_d}(t) &= 1, \forall t \leq T \end{aligned} \tag{3}$$

Equation 4 returns the next optimal vertex from a given vertex v_i by following the outgoing edge that maximizes the on time arrival probability at destination v_d.

$$\begin{gathered} f_{v_i v_d}(t) = \arg \max_{v_j:(v_i,v_j)\in E} \sum_{0}^{T} p_{v_i v_j}(t).u_{v_j v_d}(T-t) \\ \forall v_i \in V, v_i \neq v_d, 0 \leq t \leq T \end{gathered} \tag{4}$$

B. SOTA problem and the PACE road network model

The definition of the SOTA problem as it is given until now does not exploit the dependencies between the edges in the graph during the computation of the policy. For example in Figure 2 we consider edge (v_i, v_{j_1}) independent from (v_{j_1}, v_d)

and also edge (v_i, v_{j_2}) independent from (v_{j_2}, v_d). This can be seen also in Equations 3, 4 where convolution is used to derive the sum of the distributions $p_{v_i v_j}(t)$ and $u_{v_j v_d}(T-t)$. This consideration drops the quality of the solution which can be a subject to optimization. We try to improve the accuracy of the SOTA policy be applying the PACE road network model in the computation of the policy. Applying the PACE model to the SOTA policy computation increases the accuracy and can be further used in our SOTA heuristic computation.

In order to apply the PACE road network model in the SOTA policy computation, we consider two cases which examine whether there are dependencies in the SOTA path. Next we define the two cases and provide an examples to clarify them. We use Figure 3 in the example, the figure represent a path centric uncertain road network. The network consists of 10 vertices, 12 edges. We define $P_1 = \langle a, b, c\rangle$, $P_2 = \langle a, g, h\rangle$ and $P_3 = \langle g, h, i\rangle$. P_1, P_2, and P_3 have been assigned with path weights.

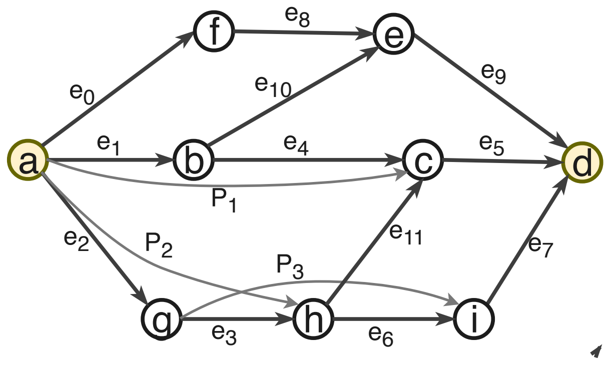


Fig. 3: SOTA policy using the PATH road network model showing the cases we considered in the policy computation.

If we use Figure 3 as an example where we want to compute the optimal policy for vertex v_a, then we have to find the next optimal edge to follow i.e. e_0, e_1 or e_2 according to Equations 3, 4. We propose two cases to be examined as follows.

Case 1

The first case considers independence between the edges in the graph, i.e. the classical model, this case has been defined with the system of Equations 3, 4. In this case, we examine edge e_0 and the probability of reaching destination v_d from v_a following edge e_0. Since this case considers independence, it happens when function W do not maintain any paths that covers edge e_0, hence Equations 3, 4 are used.

Case 2

In order to explain the second case we use the same example we already used in Case 1, where we are interested in computing the optimal policy for vertex v_a. In this case, we compute the probability of reaching destination v_d from source v_a, following edge e_1. If the optimal edge to follow after vertex v_b is e_4, we check for dependency between edges e_1, e_4. In this case, we maintain a path P_{e_1,e_4} that covers e_1,e_4 which means that there is a dependency between this two edges. We treat this as a path in the computation since it has associated uncertain travel time which is maintained by the function W. We denote the start vertex of the path with v_i. We denote the end vertex of the path with v_k. Following the example we alter the formulation as it was in Equations 3, 4 and instead of the probability $p_{v_i v_k}(t)$ we use the probability $p_{\widehat{v_i v_k}}(t)$ which denote the travel cost of a path that starts in vertex v_i and ends in vertex v_k, we again use $u_{v_k v_d}(T-t)$ as in case 1. This definition can be seen in Equation 5 case 2. If multiple paths satisfy the conditions defined in case 2, then the one that maximizes the on-time arrival probability, according to Equation 5 has been used.

Given the two cases which have been identified we define the discrete time SOTA policy under the PACE road network model with equations 5, 6 Equation 5 gives the probability of following the next optimal edge or path to destination v_d within a predefined time budget.

$$\begin{aligned} u_{v_i v_d}(t) &= \max_{v_k:(\widehat{(v_i, v_k)}} \sum_{t=0}^{T} \widehat{p_{v_i v_k}}(t).u_{v_k v_d}(T-t) \\ &\quad \forall v_i \in V, v_i \neq v_d, 0 \leq t \leq T \\ u_{v_d v_d}(t) &= 1, \forall t \leq T \end{aligned} \tag{5}$$

Equation 5 specifies which is the next vertex in which a path or an edge are ending.

$$\begin{aligned} f_{v_i v_d}(t) = \arg &\max_{v_k:(\widehat{(v_i, v_k)}} \sum_{t=0}^{T} \widehat{p_{v_i v_k}}(t).u_{v_k v_d}(T-t) \\ &\forall v_i \in V, v_i \neq v_d, 0 \leq t \leq T \end{aligned} \tag{6}$$

The SOTA definition given in Equations 5, 6 is in contrast with the original definition given in Equations 3, 4 because it considers joint distributions of paths in the policy computation.

The first case, given by Equations 3, 4 considers independence between the uncertain travel time of edges, while the second case given by Equations 5, 6 considers dependency between the uncertain travel time of a sequence of edges i.e. a path.

C. The update graph and the vertex ordering

The optimal policy can be calculated by solving Equations 5, 6. In order to compute $u_{v_i v_d}(t)$, first $u_{v_j v_d}(t')$ has to be computed, where $t \ \leq T$, $0 < t' \ \leq t - min_{v_i,v_j}$, min_{v_i,v_j} is the minimum travel time along edge (v_i, v_j), and (v_i, v_j) is an edge or a path. We compute the optimal policy of vertex

Array entry:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Time budget:	9	8	7	5	6	4	5	3	1	2	0	4	2	8	6	4	5	3	7	5
Vertex:	v_s	v_r	v_q	v_d	v_h	v_d	v_e	v_q	v_d	v_h	v_d	v_h	v_d	v_e	v_q	v_d	v_h	v_d	v_h	v_d

TABLE IV: Update array A

Array entry:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Time budget:	5	7	3	5	4	6	8	2	4	0	2	1	3	5	4	6	5	7	8	9
Vertex:	v_d	v_h	v_d	v_h	v_d	v_q	v_e	v_d	v_h	v_d	v_h	v_d	v_q	v_e	v_d	v_h	v_d	v_q	v_r	v_s

TABLE V: Reversed update array A_{rev}

v_j i.e. $u_{v_j v_d}(t')$ for time budget up to t' before computing the optimal policy of vertex v_i i.e. $u_{v_i v_d}(t)$. By propagating this constraint starting from the source v_s towards the destination v_d and decreasing the time budget up to which the convolution value has been computed, was used to derive an order of the updates required to compute the SOTA policy. We explore the graph systematically in a depth-first manner while we preserve each last-active path in order to prevent loops. We do not consider loops since we search for the path that maximizes the on-time arrival probability a priori and cycles can not increase the on-time arrival probability in such a scenario.

We start by traversing graph G, but instead of marking all visited veracities in order to prevent visiting a vertex twice, we preserve all last-active paths to prevent loops. During the traversal of G, a new graph G_{upd} has been generated. We add a new vertex to G_{upd} every time a vertex from G is visited during the traversal, we also add the edge which connects the new vertex with the last one. The vertices which we add to the update graph G_{upd} store the original vertex identifier (from graph G) and the time budget up to which the optimal policy for the vertex has to be computed. Lets say we visit vertex $v_i \in G.V$, and we have to compute the optimal policy for this vertex up to T_i. Therefore, we add a unique new vertex in the graph G_{upd} associated with the pair (v_i, T_i). An example of G_{upd} can be found in Figure 4. The update graph has been derived from the graph shown in Figure 1 with an initial time budget T equal to 9.

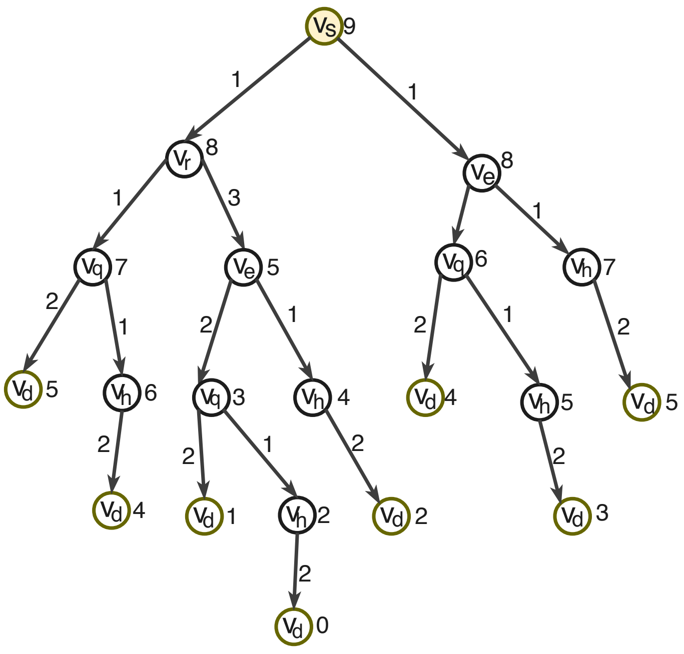


Fig. 4: G_{upd} the graph which contains all vertices and the time budgets up to which an optimal policy must be computed. The graph shows the dependencies between the vertices which have to be visited in order to compute the SOTA policy for vertex v_s and a time budget of 9. The graph in Figure 1 is used to derive G_{upd}

Since graph G_{upd} does not contain cycles, we are able to extract a topologically sorted array A of the vertices of G which are going to be touched by the SOTA computation to some time budget. Table IV shows a topologically sorted array derived from the update graph in Figure 4 for a time budget equal to 7. The reverse array A_{rev} of array A holds the sequence of updates needed to compute the SOTA policy starting from destination vertex and going to source. If traveler is located at destination and has to reach destination, then the associated probability is consider to be 1.0 and it appears to be a base case. Once we have the sequence of updates stored in A_{rev}, we are able to compute the optimal policy using equations 3, 4. The result of this pre-computation of the SOTA policy is saved in two matrices. The first one denoted with M_{ud} holds all $u_{v_i v_d}$ values, where $v_i \in V$ and v_d is a destination vertex. The second matrix M_{fd} holds all $f_{v_i v_d}$ values for $v_i \in V$ and v_d is a destination. Example of matrix M_{ud} and matrix M_{fd} are given in Table VI and Table VII respectively. The results have been obtained by solving the SOTA problem for the road network in Figure 1 with a time budget T equal to 9. Table VI holds the on time arrival probability in destination v_d within a time budget T equal to 9. Table VI shows which is the vertex that has to be followed, the vertex that provides the best on time arrival probability of reaching destination vertex v_d within a time budget of 9. The results for time $t = 7, 8, 9$ are summarized into one column, since maximum probabilities are already achieved from all sources for time equal to 7, i.e. probability of 1.0.

We propose Algorithm 1 which can be used to generate

	t=1	t=2	t=3	t=4	t=5	t=6	t=7,8,9
$\boldsymbol{v_s}$	0.0	0.0	0.0	0.12	0.42	0.68	1.0
$\boldsymbol{v_r}$	0.0	0.0	0.24	0.6	0.76	1.0	1.0
$\boldsymbol{v_e}$	0.0	0.0	0.12	0.2	0.6	1.0	1.0
$\boldsymbol{v_q}$	0.0	0.4	1.0	1.0	1.0	1.0	1.0
$\boldsymbol{v_h}$	0.0	0.6	1.0	1.0	1.0	1.0	1.0
$\boldsymbol{v_d}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE VI: Matrix M_{ud}. The matrix holds all $u_{v_i v_d}$ values, where $v_i \in V$, v_d is a destination vertex, and T is the time budget for which the optimal police has been computed.

	t=1	t=2	t=3	t=4	t=5	t=6	t=7,8,9
$\boldsymbol{v_s}$	-	-	-	v_r	v_r	v_r	v_r
$\boldsymbol{v_r}$	-	-	v_q	v_q	v_q	v_q	v_q
$\boldsymbol{v_e}$	-	-	v_h	v_q	v_q	v_q	v_q
$\boldsymbol{v_q}$	-	v_d	v_d	v_d	v_d	v_d	v_d
$\boldsymbol{v_h}$	-	v_d	v_d	v_d	v_d	v_d	v_d
$\boldsymbol{v_d}$	-	-	-	-	-	-	-

TABLE VII: Matrix M_{fd}. The matrix holds all $f_{v_i v_d}$ values, where $v_i \in V$, v_d is a destination vertex, and T is the time budget for which the optimal police has been computed.

G_{upd}. It is implemented as a recursive procedure which is exploring the graph G according to constraints involving the time budget up to which we have to compute the optimal policy. While exploring the graph G, the procedure creates the update graph G_{upd} which we then topologically sort to extract the sequence of updates needed to compute the SOTA policy.

Algorithm 1 Compute $G_{upd}(G, v_i, v_d, path, T, G_{upd})$

```
1:  procedure VISIT($G, v_j, v_d, path, T, G_{upd}$)
2:      if $v_i ! = v_d$ then
3:          $adjNodeArr = list(G.neighbors(v_i))$
4:          if $len(adjNodeArr) > 0$ then
5:              for $v_j \in adjNodeArr$ do
6:                  if $v_j \notin path$ then
7:                      $e_{joint} = W.getEdgeWeight(v_i, v_j)$
8:                      $T' = T - e_{joint}.getMinTravTime()$
9:                      if $T' \geq 0$ then
10:                         $path = path + [v_j]$
11:                         $v_1 = NewNode(v_i, T)$
12:                         $v_2 = NewNode(v_j, T_{min})$
13:                         $G_{upd}.addNode(v_2)$
14:                         $G_{upd}.addEdge(v_1, v_2)$
15:                         VISIT($G, v_j, v_d, path, T', G_{upd}$)
16:                     end if
17:                 end if
18:             end for
19:         end if
20:     end if
21: end procedure
```

The VISIT procedure takes as an input the original graph G, vertex v_i which is the vertex that has to be visited, v_d is the destination vertex, the $path$ variable is a list of vertices visited starting from source toward vertex v_i, at the begging $path = \emptyset$. T is the time budget up to which the optimal policy has to be computed for vertex v_d. G_{upd} is the update graph which does contain only the source vertex v_s at the beginning of the traversal. In the beginning, we call $VISIT(G, v_s, v_d, \emptyset, T, \emptyset)$, since $path = \emptyset$ and $G_{upd} = \emptyset$

In line 2 the algorithm checks if the visited vertex v_i is a destination, the algorithm proceed only if vertex v_i is not equal to destination vertex v_d . In line 4 the algorithm checks if the adjacent vertices are more then zero and it proceed only if this is true. Next, the algorithm iterate over the adjacent vertices (line 5). For each such a vertex v_j, it checks if the vertex is already in the path (line 6), and it continues only if the path does not form a loop. If the path does not form a loop the edge joint distribution e_{joint} is obtained in line 7. Next, the algorithm calculates the time budget T', up to which we have to compute the optimal policy for node v_j minimum travel time (line 8). After that, the algorithm checks if T' is greater then zero (line 9), and it continues only if the condition is equal to true. If $T' \geq 0$, vertex v_j is added to $path$ (line 10), vertices v_1 and v_2 have been initialized in lines 11, 12. Next we add vertex v_2 to G_{upd} in line 13 and edge (v_1, v_2) to G_{upd} in line 14. Finally, we call the visit procedure recursively (line 15). The result of the algorithm is the update graph G_{upd} required in order to obtain the order of the vertices for which the optimal policy has to be computed. Figure 4 shows G_{upd} generated by the algorithm 1 using he graph from Figure 1 to be traversed.

D. Computing the optimal policy

The solution of the SOTA problem provides an optimal policy toward a destination. We can calculate the policy based on Equations 5, 6. We propose Algorithm 2 to compute the optimal policy between a vertex v_i and a destination v_d. Algorithm 2 is executed in a loop for all $v_i \in A_{rev}$, where array A_{rev} refer to the reversed update array extracted by topologically sorted graph G_{upd}. It holds the sequence of vertices as they have to be computed without violating the constraints, regarding the time budget up to which the optimal policy has to be computed for each vertex.

The optimal policy is calculated by using dynamic programming algorithm. We consider each vertex that has to be visit with the time budget up to which the policy has to be computed as a subproblem. The subproblems have been sorted, by first, generating graph G_{upd}, then deriving an array A of topologically ordered vertices, and finally reverse array A into an array A_{rev}. Since we first order and then solve the subproblems, we define this algorithm as a bottom-up dynamic programing algorithm [22] which iterates over the ordered array of vertices A_{rev} and computes the optimal policy for each vertex according to Algorithm 2. We save the solution of subproblems which have been already computed, because we do not want to recompute them more then one time. Since bottom-up dynamic programing algorithm has been used, recursion is not required for the implementation, considering that it can be slower in comparison with a simple loop, it also can require more space.

Algorithm 2 computes the optimal policy for an intermediate vertex v_i toward the destination vertex v_d within a predefined time budget t. The algorithm takes the following parameters as input: vertex $v_i \in A_{rev}$, destination vertex $v_d \in V$, matrix M_{ud}, matrix M_{fd}, time t up to which the policy has been computed for vertex v_i, original graph G and function W. Matrix M_{ud} holds the optimal policy probabilities toward a destination v_d for all $v_i \in A_{rev} \in V$ and time budget less then t, the initial time budget. Matrix Mfd holds the optimal next vertex to follow, i.e. the vertex that maximizes the on-time arrival probability toward a destination vertex v_d.

The algorithm starts by obtaining all adjacent vertices of vertex v_i and assign them to a list $adjNodeArr$ (line 1). Next, the algorithm checks if vertex v_i is not a destination vertex v_d and the length of $adjNodeArr > 0$ (line 2). It also checks whether the optimal policy for vertex v_i has been computed up to the time t or the optimal policy for vertex v_i was not computed (line 3). This check is performed to avoid recomputing the police again if it was already computed. The algorithm continues only if both the conditions are satisfied. In line 4 variable $size$ is initialized to be equal to zero. Next, the algorithm checks if the optimal policy for vertex v_i has been computed for times smaller then the time budget t (line 5) and if so it set the $size$ variable to be equal to the time up to which the optimal policy for vertex v_i was computed (line 6). This way the algorithm does not recompute the policy for all times up to t, instead it computes the optimal policy of vertex v_i for times greater then $size$ and smaller then t. Next, a list $maxu_{v_iv_d}$ is defined and initialized with length equal to $(T - size)$, all its entries are set to be equal to 0.0 (line 8). $maxu_{v_iv_d}$ list is used to store the probability values of the optimal policy for vertex v_i. i.e. the values which was not yet computed. Since we can have more then one adjacent vertex, the variable $maxu_{v_iv_d}$ stores the value which maximizes the probability of reaching the destination vertex v_d among the adjacent nodes. Similarly, the algorithm initialize a variable $maxf_{v_iv_d}$ which holds the next optimal vertex to follow(line 9). Next, Algorithm 2 iterates over the adjacent vertices of vertex v_i (line 10). For each such a vertex v_j, the algorithm extract list of paths that traverse edge (v_i, v_j) (line 11), this is where the PACE road network model is applied to the SOTA computation. The algorithm does not only check the adjacent edges of vertex v_i, i.e (v_i, v_j), it also considers all the paths traversing the edge (v_i, v_j) as explained in Section IV-B. We use a function $getAllPaths$ which takes as input graph G, weighted function W and an edge (v_i, v_j) and returns all paths which traverse edge (v_i, v_j) and which are starting in edge (v_i, v_j). The algorithm iterate over the paths which traverse edge (v_i, v_j) (line 13). Next, a variable v_l is initialized to hold the vertex in which the path under consideration ends (line 14), then joint distribution of the $path$ is obtained using the function W (line 15) and then the minimum travel time for the path (line 16) is derived. The algorithm proceed by checking if the remaining time $t - path_{min} > 0$ when located in vertex v_l, and whether the optimal policy for vertex v_l has been computed (line 17), if so the algorithm has to convolute the path distribution with the $u_{v_l,v_d}(t)$ distribution. Since we might have computed the optimal policy up to some time, we wold like to reuse it, instead of recompute the optimal policy for all time budgets.

Algorithm 2 Compute SOTA policy for a single source $(v_i, v_d, M_{ud}, M_{fd}, t, G, W)$

```
1:  $adjNodeArr = G.neighbors(v_i)$
2:  if $v_i! = v_d \wedge len(adjNodeArr) > 0$ then
3:      if $(v_i \in M_u \wedge len(M_{ud}[v_i]) < t) \vee v_i \notin M_{ud}$ then
4:          $size = 0$
5:          if $v_i \in M_u \wedge len(M_{ud}[v_i]) < t$ then
6:              $size = len(M_{ud}[v_i])$
7:          end if
8:          $maxu_{v_iv_d} = [0.0] * (t - size)$
9:          $maxu_{v_iv_d} = [0.0] * (t - size)$
10:         for $v_j \in adjNodeArr$ do
11:             $paths_{ij} = getAllPaths(G, W, (v_i, v_j))$
12:             $sizePaths = len(paths_{ij})$
13:             for $path \in paths_{ij}$ do
14:                 $v_l = path[len(path)]$
15:                 $path_{joint} = W.getPathWeight(path)$
16:                 $path_{min} = path_{joint}.minTravelT()$
17:                 if $t - path_{min} >= 0 \wedge v_l \in M_{ud}$ then
18:                     if $v_i \in M_{ud} \wedge len(M_{ud}[v_i]) < t$ then
19:                         $c = cnv_{ext}(path_{joint}, M_{ud}[v_l], t)$
20:                     else
21:                         $c = cnv(path_{joint}, M_{ud}[v_l], t)$
22:                     end if
23:                     for $l \in range(0, t - size)$ do
24:                         if $maxu_{v_iv_d}[l] < c[l]$ then
25:                             $maxu_{v_iv_d}[l] = c[l]$
26:                             $maxf_{v_iv_d}[l] = v_l$
27:                         end if
28:                     end for
29:                 end if
30:             end for
31:         end for
32:         if $v_i \notin M_{ud}$ then
33:             $M_{ud}[v_i] = maxu_{v_iv_d}$
34:             $M_{fd}[v_i] = maxf_{v_iv_d}$
35:         else if $len(M_u[v_i]) < t$ then
36:             $M_{ud}[v_i].extend(maxu_{v_iv_d})$
37:             $M_{fd}[v_i].extend(maxf_{v_iv_d})$
38:         end if
39:     end if
40: end if
```

This is why the algorithm checks if the optimal policy for vertex v_i was computed to a time t, line 18. If the optimal policy was not computed, the algorithm commutes it (line 21), If the optimal policy was computed up to some time less then time t (line 20), we compute the optimal policy only for times greater then the times we already have been computed (line 19). Next, the algorithm maximizes the values of the optimal policy over all the adjacent vertices of vertex v_i for all considered times(line 23,line 24,line 25,line 26). Finally, the algorithm assigns the maximized values in matrix M_{ud} and matrix M_{fd} which store the results. (line 32to line 37)

Algorithm 2 computes the SOTA policy only for a single source towards a destination, therefore we have to execute it in a loop to compute the optimal policy for all sources $v_s \in A_{rev}$. We propose Algorithm 4 which perform the same, it iterates over the sources $v_s \in A_{rev}$ and compute the optimal policy for each source towards the destination.

Algorithm 3 Compute SOTA for all sources in A_{rev}

```
1: procedure CompSOTA($G, A_{rev}, v_d, M_{ud}, M_{fd}, t, G, W$)
2:     for $v_s \in A_{rev}$ do
3:         Run Algorithm 2($v_s, v_d, M_{ud}, M_{fd}, t, G, W$)
4:     end for
5: end procedure
```

Since the solution of the SOTA problem has to be computed for all sources, all destinations and all time budgets, it is considered to be a time consuming operation. Equations 5, 6 define SOTA problem in discrete time. We use Δt to denote the discretization interval of interest, $0 < \Delta t < t_{min}$, where t_{min} is the minimum travel time among all edge in the set E. In reality selecting proper discretization can be important consideration. We can decrease Δt which can increase the quality of the solution, but will decrease the running time of Algorithm 1,Algorithm 2. This suggests that time can be trade for quality and the opposite while computing the SOTA policy.

E. Example of computing SOTA policy

We provide a simple example of SOTA policy computation for a four vertices, i.e. v_d, v_q, v_h and v_e. We try to show how the computation starts and we leave to the reader the rest of the SOTA computation. The example considers the graph in Figure 1 and initial time budget $T = 9$. We first generate an update graph G_{upd} as in Figure 4. Next, an update array A is extracted from G_{upd} by using topological sorting, Figure IV, Next the reverse of A, A_{rev} is instantiated V. We clarify that the probability of reaching a destination when situated at destination is equal to 1.0 as shown in Equation 5. This holds for all times $t \leq T$. This can be seen in Table VI in the row with key v_d all values for all time budgets are equal to 1.0. We set all values of this row to 1.0 at the very beginning.

After, A_{rev} is obtained we can run Algorithm 3. This means that we have to loop over all vertices $v_s \in A_{rev}$ and compute the SOTA policy for each vertex v_s toward the destination v_d.

We start with vertex v_d and time $T' = 5$. It is already computed no need to recomputed. Next entry in A_{rev} is vertex v_h with time $T' = 7$. The algorithm has to compute $u_{v_h v_d}(T')$ for $T' \in [1, 7]$. Since there is only one adjacent vertex, the algorithm computes the optimal policy for this vertex which will be the maximum over all adjacent vertices. The computation is performed as follows:

$u_{v_h v_d}(1) =$
$= \sum_{k=1}^{T'} p_{v_h v_d}(k).u_{v_d v_d}(T'-k) = \sum_{k=1}^{1} p_{v_h v_d}(k).u_{v_d v_d}(1-k) =$
$= p_{v_h v_d}(1).u_{v_d v_d}(1-1) = p_{v_h v_d}(1).u_{v_d v_d}(2) = 0.0 * 1.0 = 0.0$

$u_{v_h v_d}(2) =$
$= \sum_{k=1}^{T'} p_{v_h v_d}(k).u_{v_d v_d}(T'-k) = \sum_{k=1}^{2} p_{v_h v_d}(k).u_{v_d v_d}(2-k) =$
$= p_{v_h v_d}(1).u_{v_d v_d}(2-1) + p_{v_h v_d}(2).u_{v_d v_d}(2-2) =$
$= p_{v_h v_d}(1).u_{v_d v_d}(2) + p_{v_h v_d}(2).u_{v_d v_d}(1) =$
$= 0.0 * 1.0 + 0.6 * 1.0 = 0.0 + 0.6 = 0.6$

$u_{v_h v_d}(3) =$
$= \sum_{k=1}^{T'} p_{v_h v_d}(k).u_{v_d v_d}(T'-k) = \sum_{k=1}^{3} p_{v_h v_d}(k).u_{v_d v_d}(3-k) =$
$= p_{v_h v_d}(1).u_{v_d v_d}(3-1) + p_{v_h v_d}(2).u_{v_d v_d}(3-2) + p_{v_h v_d}(3).u_{v_d v_d}(3--3) =$
$= p_{v_h v_d}(1).u_{v_d v_d}(2) + p_{v_h v_d}(2).u_{v_d v_d}(1) + p_{v_h v_d}(3).u_{v_d v_d}(0) =$
$= 0.0 * 1.0 + 0.6 * 1.0 + 0.4 * 1.0 = 0.0 + 0.6 + 0.4 = 1.0$

$u_{v_h v_d}(1) = 0.0$, $u_{v_h v_d}(2) = 0.6$ and $u_{v_h v_d}(3) = 1.0$ are inserted into matrix M_{ud} for row with key v_h and $t \in [1, 3]$ as it is in Table VI $u_{v_h v_d}(T') = 1.0$, for each $T' > 3$ and $T' \leq 7$. This is the case because the travel cost distribution is a cumulative function which is increasing for input t. We insert $u_{v_h v_d}(T') = 1.0$, for each $T' > 3$ and $T' \leq 7$ values in matrix M_{ud} in row with key v_h.

Next vertex in A_{rev} is v_d with $t = 3$, it is already computed, no need to recomputed again. Next vertex in A_{rev} is v_h with $t = 5$. We already have computed v_h for $t = 7$, since $5 < 7$ no need to recomputed again. Next vertex is v_d, it is already computed.

Next vertex in A_{rev} is v_q with $t = 5$.
$u_{v_q v_d}(T') =$
$\max(\sum_{k=1}^{T'} p_{v_q v_h}(k).u_{v_h v_d}(T'-k), \sum_{k=1}^{T'} p_{v_q v_d}(k).u_{v_d v_d}(T'-k))$
We do not maintain any paths that cover edges (v_q, v_h),(v_q, v_d), therefore SOTA Case 1 is used for both edges (v_q, v_h),(v_q, v_d). The algorithm has to compute and maximize over $u_{v_q v_d}(T')$ and $u_{v_q v_h}(T')$ for $T' \leq t$. First, we compute $u_{v_q v_d}(T')$ as follows:

$u_{v_q v_d}(1) =$
$= \sum_{k=1}^{T'} p_{v_q v_d}(k).u_{v_d v_d}(T'-k) = \sum_{k=1}^{1} p_{v_q v_d}(k).u_{v_d v_d}(1-k) =$
$= p_{v_q v_d}(1).u_{v_d v_d}(1-1) = p_{v_q v_d}(1).u_{v_d v_d}(2) = 0.0*1.0 = 0.0$

$u_{v_q v_d}(2) =$
$= \sum_{k=1}^{T'} p_{v_q v_d}(k).u_{v_d v_d}(T'-k) = \sum_{k=1}^{2} p_{v_q v_d}(k).u_{v_d v_d}(2-k) =$
$= p_{v_q v_d}(1).u_{v_d v_d}(2-1) + p_{v_q v_d}(2).u_{v_d v_d}(2-2) =$
$= p_{v_q v_d}(1).u_{v_d v_d}(2) + p_{v_q v_d v_d}(2).u_{v_d v_d}(1) =$
$= 0.0 * 1.0 + 0.4 * 1.0 = 0.0 + 0.4 = 0.4$

$u_{v_q v_d}(3) =$
$= \sum_{k=1}^{T'} p_{v_q v_d}(k).u_{v_d v_d}(T'-k) = \sum_{k=1}^{3} p_{v_q v_d}(k).u_{v_d v_d}(3-k) =$
$= p_{v_q v_d}(1).u_{v_d v_d}(3-1) + p_{v_q v_d}(2).u_{v_d v_d}(3-2) + p_{v_q v_d}(3).u_{v_d v_d}(3--3) =$
$= p_{v_q v_d}(1).u_{v_d v_d}(2) + p_{v_q v_d}(2).u_{v_d v_d}(1) + p_{v_q v_d}(3).u_{v_d v_d}(0) =$
$= 0.0 * 1.0 + 0.4 * 1.0 + 0.6 * 1.0 = 0.0 + 0.4 + 0.6 = 1.0$

$u_{v_h v_d}(T') = 1.0$, for $T' > 3$ and $T' \leq 5$.

Second, we compute $u_{v_q v_h}(T')$ as follows:

$u_{v_q v_h}(1) =$
$= \sum_{k=1}^{T'} p_{v_q v_h}(k).u_{v_h v_d}(T'-k) = \sum_{k=1}^{1} p_{v_q v_h}(k).u_{v_h v_d}(1-k) =$
$= p_{v_q v_h}(1).u_{v_h v_d}(1-1) = p_{v_q v_h}(1).u_{v_h v_d}(2) = 0.5 * 0.0 = 0.0$

$u_{v_q v_h}(2) =$
$= \sum_{k=1}^{T'} p_{v_q v_h}(k).u_{v_h v_d}(T'-k) = \sum_{k=1}^{2} p_{v_q v_h}(k).u_{v_h v_d}(2-k) =$
$= p_{v_q v_h}(1).u_{v_h v_d}(2-1) + p_{v_q v_h}(2).u_{v_h v_d}(2-2) =$
$= p_{v_q v_h}(1).u_{v_h v_d}(2) + p_{v_q v_h}(2).u_{v_h v_d}(1) =$
$= 0.5 * 0.0 + 0.5 * 0.0 = 0.0 + 0.0 = 0.0$

$u_{v_q v_h}(3) =$
$= \sum_{k=1}^{T'} p_{v_q v_h}(k).u_{v_h v_d}(T'-k) = \sum_{k=1}^{3} p_{v_q v_h}(k).u_{v_h v_d}(3-k) =$
$= p_{v_q v_h}(1).u_{v_h v_d}(3-1) + p_{v_q v_h}(2).u_{v_h v_d}(3-2) + p_{v_q v_h}(3).u_{v_h v_d}(3--3) =$
$= p_{v_q v_h}(1).u_{v_h v_d}(2) + p_{v_q vhd}(2).u_{v_h v_d}(1) + p_{v_q v_h}(3).u_{v_h v_d}(0) =$
$= 0.5 * 0.0 + 0.5 * 0.0 + 0.5 * 0.6 = 0.0 + 0.0 + 0.3 = 0.3$

We stop because $u_{v_q v_d}(T') = 1.0$, for $T' \in [4, 5]$ and we can not obtain greater value than $u_{v_q v_h}(T')$, for $T' \in [4, 5]$ After we have computed $u_{v_q v_d}(T')$ and $u_{v_q v_h}(T')$ for $T' \in [1, 5]$ we maximize. It is obvious that $u_{v_q v_d}(T')$ shows greater values for $T' \in [1, 5]$.

We inserted $u_{v_q v_d}(1) = 0.0$, $u_{v_h v_d}(2) = 0.4$ and $u_{v_h v_d}(3) = 1.0$ into matrix M_{ud} into row with key v_q and $t \in [1, 3]$ as it is in Table VI For each $u_{v_q v_d}(T')$, where $T' > 3$ and $T' \leq 5$ are going to evaluate also to 1.0. We insert these values in matrix M_{ud} in row with key v_q and the corresponded times.

Next vertex from A_{rev} is v_e with time $t = 8$.

$u_{v_e v_d}(T') =$
$\max(\sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h v_d}(T'-k), \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_d v_d}(T'-k))$

We do not maintain any paths that cover edges (v_e, v_h),(v_e, v_q), therefore SOTA Case 1 is used for both edges (v_e, v_h) and (v_e, v_q). The algorithm has to compute and maximize over $u_{v_e v_q}(T')$ and $u_{v_e v_h}(T')$ for $T' \leq t$. First, we compute $u_{v_e v_q}(T')$ for $T' \leq t$ as follows:

$u_{v_e v_q}(1) =$
$= \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_q}(T'-k) = \sum_{k=1}^{1} p_{v_e v_q}(k).u_{v_q}(1-k) =$
$= p_{v_e v_q}(1).u_{v_q}(1-1) = p_{v_e v_q}(1).u_{v_q}(0) = 0.0 * 0.0 = 0.0$

$u_{v_e v_q}(2) =$
$= \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_q}(T'-k) = \sum_{k=1}^{1} p_{v_e v_q}(k).u_{v_q}(1-k) =$
$= p_{v_e v_q}(1).u_{v_q}(2-1) + p_{v_e v_q}(2).u_{v_q}(2-2) = p_{v_e v_q}(1).u_{v_q}(1) + p_{v_e v_q}(2).u_{v_q}(0) = 0.0*0.0+0.5*0.0 = 0.0$

$u_{v_e v_q}(3) =$
$= \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_q}(T'-k) = \sum_{k=1}^{3} p_{v_e v_q}(k).u_{v_q}(3-k) =$
$= p_{v_e v_q}(1).u_{v_q}(3-1) + p_{v_e v_q}(2).u_{v_q}(3-2) + p_{v_e v_q}(3).u_{v_q}(3-3) = p_{v_e v_q}(1).u_{v_q}(2) + p_{v_e v_q}(2).u_{v_q}(1) + p_{v_e v_q}(3).u_{v_q}(0) = 0.0 * 0.0 + 0.5 * 0.0 + 0.5 * 0.0 = 0.0$

$u_{v_e v_q}(4) =$
$= \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_q}(T'-k) = \sum_{k=1}^{4} p_{v_e v_q}(k).u_{v_q}(4-k) =$
$= p_{v_e v_q}(1).u_{v_q}(4-1) + p_{v_e v_q}(2).u_{v_q}(4-2) + p_{v_e v_q}(3).u_{v_q}(4-3) + p_{v_e v_q}(4).u_{v_q}(4-4) =$
$= p_{v_e v_q}(1).u_{v_q}(3) + p_{v_e v_q}(2).u_{v_q}(2) + p_{v_e v_q}(3).u_{v_q}(1) + p_{v_e v_q}(4).u_{v_q}(0) =$
$= 0.0 * 1.0 + 0.5 * 0.4 + 0.5 * 0.0 = 0.2$

$u_{v_e v_q}(5) =$
$= \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_q}(T'-k) = \sum_{k=1}^{5} p_{v_e v_q}(k).u_{v_q}(5-k) =$
$= p_{v_e v_q}(1).u_{v_q}(5-1) + p_{v_e v_q}(2).u_{v_q}(5-2) + p_{v_e v_q}(3).u_{v_q}(5-3) + p_{v_e v_q}(4).u_{v_q}(5-4) + p_{v_e v_q}(5).u_{v_q}(5-5) =$
$= p_{v_e v_q}(1).u_{v_q}(4) + p_{v_e v_q}(2).u_{v_q}(3) + p_{v_e v_q}(3).u_{v_q}(2) + p_{v_e v_q}(4).u_{v_q}(1) + p_{v_e v_q}(5).u_{v_q}(0) =$
$= 0.0 * 1.0 + 0.5 * 1.0 + 0.5 * 0.2 + 0.0 * 0.0 + 0.0 * 0.0 = 0.6$

$u_{v_e v_q}(6) =$
$= \sum_{k=1}^{T'} p_{v_e v_q}(k).u_{v_q}(T'-k) = \sum_{k=1}^{6} p_{v_e v_q}(k).u_{v_q}(6-k) =$
$= p_{v_e v_q}(1).u_{v_q}(6-1) + p_{v_e v_q}(2).u_{v_q}(6-2) + p_{v_e v_q}(3).u_{v_q}(6-3) + p_{v_e v_q}(4).u_{v_q}(6-4) + p_{v_e v_q}(5).u_{v_q}(6-5) + p_{v_e v_q}(6).u_{v_q}(6-6) =$
$= p_{v_e v_q}(1).u_{v_q}(5) + p_{v_e v_q}(2).u_{v_q}(4) + p_{v_e v_q}(3).u_{v_q}(3) + p_{v_e v_q}(4).u_{v_q}(2) + p_{v_e v_q}(5).u_{v_q}(1) + p_{v_e v_q}(6).u_{v_q}(0) =$
$= 0.0 * 1.0 + 0.5 * 1.0 + 0.5 * 1.0 + 0.0 * 0.2 + 0.0 * 0.0 = 1.0$

For each $u_{v_h v_d}(T') = 1.0$, for $T' > 6$ and $T' \leq 8$.

Second, we calculate $u_{v_e v_h}(T')$ for $t \in [1, 8]$

$u_{v_e v_h}(1) =$
$= \sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h}(T' - k) = \sum_{k=1}^{1} p_{v_e v_h}(k).u_{v_h}(1 - k) =$
$= p_{v_e v_h}(1).u_{v_h}(1 - 1) =$
$= p_{v_e v_h}(1).u_{v_h}(0) =$
$= 0.2 * 0.0 = 0.0$

$u_{v_e v_h}(2) =$
$= \sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h}(T' - k) = \sum_{k=1}^{2} p_{v_e v_h}(k).u_{v_h}(2 - k) =$
$= p_{v_e v_h}(1).u_{v_h}(2 - 1) + p_{v_e v_h}(2).u_{v_h}(2 - 2) =$
$= p_{v_e v_h}(1).u_{v_h}(1) + p_{v_e v_h}(2).u_{v_h}(0) =$
$= 0.2 * 0.0 + 0.0 * 0.0 = 0.0$

$u_{v_e v_h}(3) =$
$= \sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h}(T' - k) = \sum_{k=1}^{3} p_{v_e v_h}(k).u_{v_h}(3 - k) =$
$= p_{v_e v_h}(1).u_{v_h}(3-1) + p_{v_e v_h}(2).u_{v_h}(3-2) + p_{v_e v_h}(3).u_{v_h}(3-3) =$
$= p_{v_e v_h}(1).u_{v_h}(2) + p_{v_e v_h}(2).u_{v_h}(1) + p_{v_e v_h}(3).u_{v_h}(0) =$
$= 0.2 * 0.6 + 0.0 * 0.0 + 0.8 * 0.0 = 0.12$

$u_{v_e v_h}(4) =$
$= \sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h}(T' - k) = \sum_{k=1}^{4} p_{v_e v_h}(k).u_{v_h}(4 - k) =$
$= p_{v_e v_h}(1).u_{v_h}(4-1) + p_{v_e v_h}(2).u_{v_h}(4-2) + p_{v_e v_h}(3).u_{v_h}(4-3) + p_{v_e v_h}(4).u_{v_h}(4 - 4) =$
$= p_{v_e v_h}(1).u_{v_h}(3) + p_{v_e v_h}(2).u_{v_h}(2) + p_{v_e v_h}(3).u_{v_h}(1) + p_{v_e v_h}(4).u_{v_h}(0) =$
$= 0.2 * 1.0 + 0.0 * 0.6 + 0.8 * 0.0 + 0.0 * 0.0 = 0.2$

$u_{v_e v_h}(5) =$
$= \sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h}(T' - k) = \sum_{k=1}^{5} p_{v_e v_h}(k).u_{v_h}(5 - k) =$
$= p_{v_e v_h}(1).u_{v_h}(5-1) + p_{v_e v_h}(2).u_{v_h}(5-2) + p_{v_e v_h}(3).u_{v_h}(5-3) + p_{v_e v_h}(4).u_{v_h}(5 - 4) + p_{v_e v_h}(5).u_{v_h}(5 - 5) =$
$= p_{v_e v_h}(1).u_{v_h}(4) + p_{v_e v_h}(2).u_{v_h}(3) + p_{v_e v_h}(3).u_{v_h}(2) + p_{v_e v_h}(4).u_{v_h}(1) + p_{v_e v_h}(5).u_{v_h}(0) =$
$= 0.2*1.0 + 0.0*1.0 + 0.8*0.6 + 0.0*0.0 + 0.0*0.0 = 0.68$

$u_{v_e v_h}(6) =$
$= \sum_{k=1}^{T'} p_{v_e v_h}(k).u_{v_h}(T' - k) = \sum_{k=1}^{6} p_{v_e v_h}(k).u_{v_h}(6 - k) =$
$= p_{v_e v_h}(1).u_{v_h}(6-1) + p_{v_e v_h}(2).u_{v_h}(6-2) + p_{v_e v_h}(3).u_{v_h}(6-3) + p_{v_e v_h}(4).u_{v_h}(6 - 4) + p_{v_e v_h}(5).u_{v_h}(6 - 5) + p_{v_e v_h}(6).u_{v_h}(6 - 6) =$
$= p_{v_e v_h}(1).u_{v_h}(5) + p_{v_e v_h}(2).u_{v_h}(4) + p_{v_e v_h}(3).u_{v_h}(3) + p_{v_e v_h}(4).u_{v_h}(2) + p_{v_e v_h}(5).u_{v_h}(1) + p_{v_e v_h}(6).u_{v_h}(0) =$
$= 0.2*1.0 + 0.0*1.0 + 0.8*1.0 + 0.0*0.6 + 0.0*0.0 + 0.0*0.0 = 1.0$

$u_{v_e v_h}(T') = 1.0$, for $T' > 6$ and $T' \leq 8$.

After we have computed $u_{v_e v_q}(T')$ and $u_{v_e v_h}(T')$ for $T' \in [1, 8]$ we maximize. It is obvious that $u_{v_e v_q}(T')$ show greater values for $T' \in [4, 8]$ while $u_{v_e v_q}(T')$ for $T' = 3$.

The algorithm continues in the same way. It calculates the SOTA policy for each element in A_{rev} as a source toward the destination until no more vertices are left inside A_{rev}.

V. Pruning with Arc-Potential

In the case of deterministic path finding, all edges of a graph have associated deterministic values e.g. minimum travel time. There are well known algorithms such as Dijkstra's algorithm and A* which can find the shortest path between source and destination vertex in a deterministic graph. Even though, Dijkstra's algorithm runs in super-linear time, optimization techniques allow for sub-linear running time. By preprocecing a deterministic graph, we can speed up the queries in the graph with an order of magnitude in comparison with no preprocesed graph.

A well known technique for speed up the queries in a graph is Arc-Flags. In the deterministic version of Arc-Flags speed up of the queries is achieved by pruning the discovered search space. The graph is first partitioned into p number destination regions, $p > 0$ and $p \leq m$. For each destination region $D_i \in D$, where $i \leq p$ and D is the set of all destination regions. We also use a function $Q : v_d \rightarrow D_i$ that maps a destination vertex v_d with its destination region D_i, each edge of the graph is associated with a bit vector with a size equal to p, i.e. the number of destination regions. Each entry of the bit vector has value equal to one iff the edge is at the beginning to at least one shortest path to the corresponding destination region or it is set to zero otherwise. This is done by running a shortest path three computation in the reversed graph for each destination vertex v_d in a destination region $Q : v_d \rightarrow D_i$. All vertices which have been touched by the shortest path tree computation are labeled with their travel times to destination.

Unfortunately, the number of destination vertices in a destination region can be large which means that we have to perform large number of shortest path tree computations which can be simple inefficient. To contest with this two important observations are made, first, a shortest path towards a destination can be decomposed into two paths which are also going to be shortest paths, and second, a shortest path which starts in a vertex v_i located outside of a destination region $Q : v_d \rightarrow D_j$, must enter the destination region D_j at some edge. We denote this edge with (v_h, v_t), v_h is the head vertex and v_t the tail vertex of edge (v_h, v_t). Such edges are called *boundary edges*. The tail vertex v_t of a boundary edge is called $boundary\ vertex$. We denote the set of boundary vertices B_i, where i refer to the index of the destination region under consideration. We also define a function $R : D_i \rightarrow B_i$ which returns the set of boundary vertices for a given destination region. Based on the observations, a conclusion was made that only, boundary vertices in a destination region has to be considered as sources of the shortest path tree computations when preprocess the graph.

This reduces the number of destination vertices by using only a subset of the destinations located in a destination region, hence improves the speed of preprocessing.

Example of boundary vertices and boundary edges is provided in Figure 5 where a simple graph has been partition in four destination regions. The destination region under which the boundary vertices and edges are considered is $Region2$. The boundary edges are $(v_b, v_c), (v_j, v_i), (v_h, v_i), (v_k, v_m)$ and the boundary vertices are v_c, v_i, v_m.

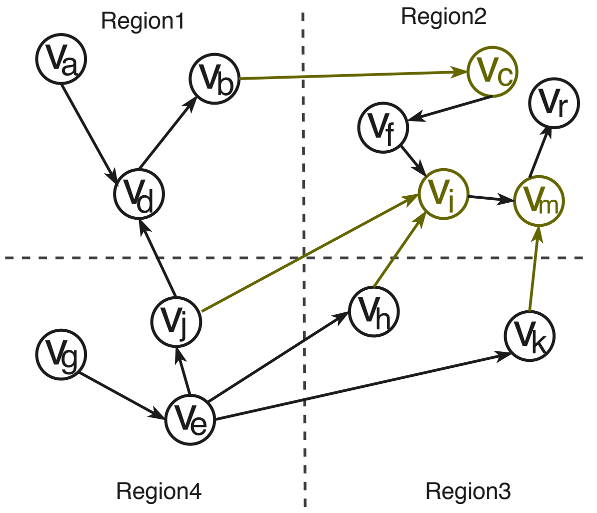


Fig. 5: Arc-Flags, boundary vertices and boundary edges

Arc-Flags already have been upgraded to stochastic Arc-Flgs [5] by changing the concept of *belongs to a shortest path* with *belongs to an optimal policy* instead. Stochastic Arc-Flags associate each arc of the graph with a number of bit vectors, each with a size equal to the number of destination region in the graph. The number of bit vectors associated with each arc is equal to the time budget because we have to compute SOTA for all times less then or equal to the initial time budget T. Each entry of such a bit vector which corresponds to a time t' is equal to one if the arc is on the optimal policy to at least one destination in a destination region within some time less then or equal to the corresponding vector time t'. This means that in the stochastic settings, Arc-Flags has to be computed for all time budget up to T which can be significant overhead for large values of T. Stochastic arc-flag has linear space complexity $\Theta(T)$.

For example Table VIII shows the computed Arc-Flags of some edge from some graph partitioned into four regions, where the initial time budget $T = 5$. Table VIII must be interpreted as follows: for a time equal to 3 the edge belongs to an optimal policy towards destination Region 1, 2 and 4. The arc does not belong to an optimal policy to destination Regions 3 for a time equal to 3.

It can be seen that the number of vectors increases when the time budget increases and also multiple such tables must be preserved, one for each edge in the graph. This could require significant computation power and might be prohibited for large networks.

	Reg. 1	Reg. 2	Reg. 3	Reg. 4
Vector 1, t=1	0	0	0	1
Vector 2, t=2	0	1	0	1
Vector 3, t=3	1	1	0	1
Vector 4, t=4	1	1	0	1
Vector 5, t=5	1	1	1	1

TABLE VIII: Stochastic Arc-Flags

Arc-Potential is a graph preprocessing techniques which is an improvement of stochastic Arc-Flags, it reduces the information associated with each vertex (the number of bit vector equal to the initial time budget T). Instead of storing whether an edge belongs to an optimal policy for all time budgets up to T, Arc-Potential partition the times into time buckets in the form (t-,t+), for each such time bucket, Arc-Potential record whether or not an edge becomes a part of an optimal policy for some destination region. Table IX shows how time buckets are used to store Arc-Potentials. The space consumption drops when the size of the time buckets increases and the opposite. The information form table IX must be interpreted as follows: the edge belongs to some optimal policy to destination Regions 1 ,2 and 4 for a time $t \in [3, 4]$. The same edge does not belong to any optimal policy toward a destination Region 3 for a time $t \in [3, 4]$.

	Reg 1	Reg. 2	Reg. 3	Reg. 4
$t \in [1, 2]$	0	1	0	1
$t \in [3, 4]$	1	1	0	1
$t \in [5, 6]$	1	1	1	1

TABLE IX: Arc-Potential using time buckets

[4] argue that recording only the minimum time budget up to which an arc become a part of an optimal policy for some destination region within a predefined time budget T can be stored without affecting significantly the pruning ability of Arc-Potential. In this paper we consider recording the minimum time budget for which an arc become part of an optimal police to destination within a predefined time budget. Table X shows an example of Arc-Potential using only the minimum time for which the arc becomes a part of an optimal policy towards a destination region. This settings of Arc-Potential allows for liner space complexity in the size of the time budget T

	Reg 1	Reg. 2	Reg. 3	Reg. 4
Min time	3	2	5	1

TABLE X: Arc-Potential using minimum time budget for which the arc becomes part of an optimal policy toward a destination region

A. Algorithm for generating Arc-Potentials

Matrix M_{fd} holds all $f_{v_i v_d}(t)$ values computed by the optimal policy, where $v_i \in V, t \leq T$ and v_d is the destination. We define the set L_d to contain all keys of matrix M_{fd}. For example in Table VII the set of keys L_d consist of the following vertices $v_s, v_h, v_e, v_d, v_q, v_r$ (the column to the left).

We define $\phi_{v_i v_j}(D_k)$ to be the Arc-Potential minimum time for which the edge (v_i, v_j) becomes a part of an optimal policy towards a destination region D_k. We define $\psi_{v_i v_j}(D_k, t)$ to be the Arc-Potential probability of edge (v_i, v_j) to reach destination a destination region D_k for a time t. Algorithm 4 show how Arc-Potentials have been computed by using SOTA policy.

Algorithm 4 Compute $Arc-Potentials(G, T, D, M_u)$

```
1: for $edge(v_i, v_j) \in E$ do
2:     for $D_k \in D$ do
3:         $\phi_{v_i v_j}(D_k) = \infty$
4:         $\Psi_{v_i v_j}(D_k, t) = [0.0] * T$
5:     end for
6: end for
7: for $D_k \in D$ do
8:     $B_k \leftarrow R : D_k$
9:     for $v_d \in B_k$ do
10:        for $v_i \in L_{id}$ do
11:            $T'$ = minimum time budget for which $v_i$
12:            become a part of an opt. pol.
13:            for $T' < t \leq T$ do
14:                $v_j = f_{v_i v_d}(t)$
15:                $pr_{v_j} = u_{v_i v_d}(t)$
16:                $\phi_{v_i v_j}(D_k) = min(\phi_{v_i v_j}(D_k), t)$
17:                $\psi_{v_i v_j}(D_k, t) = max(\psi_{v_i v_j}(D_k, t), pr_{v_j})$
18:            end for
19:        end for
20:    end for
21: end for
```

The algorithm starts by iterating over all edges in G, line 1, for each edge (v_i, v_j) we iterate over all destination regions $D_k \in D$, line 2. The algorithm set the correspond Arc-Potential of edge (v_i, v_j) towards a destination region D_k to infinity in line 3, Arc-Flag probabilities of edge (v_i, v_j) towards a destination region D_k are also initialized in line 4.

Next, the algorithm iterates over all destination regions. line 7. For each such region D_k, the set of boundary vertices B_k is obtained by using function R, line 8. Next, the algorithm iterates over the destinations v_d form the set B_k which contains all boundary vertices for destination region D_k, line 9. Next, the algorithm iterates over all keys $v_i \in L_d$, this are the keys from the M_{ud} matrix, line 10. Next the algorithm finds the minimum time budget for which the vertex can be part of an optimal policy, line 11. How to find the minimum time can be explained using Table VI. Say, we are interested in finding the minimum time for vertex v_e, we get the row with key v_e from M_{ud} and we check for each time $0 < t \leq T$ whether the probability of reaching destination is greater then 0.0. we stop the loop once we find probability grater then zero. Finally, we lock up the associated travel time as a minimum travel time for which the node become a part of optimal policy.

Next the algorithm iterates over all times between T' and T, line 13. In line 14 we initialize vertex v_j to hold the next optimal vertex of vertex v_i for each time t, $0 < t \leq T$. In line 15 the algorithm obtain the probability of edge (v_i, v_j) to become a part of an optimal policy toward a destination vertex and some time. In line 16, the algorithm check if the current Arc-Potential of edge $(v_i v_j)$ towards a destination region D_k is larger in comparison with time t if so we reset the Arc-Potential of edge $(v_i v_j)$ towards a destination region D_k to be equal to the time t In line 17, the algorithm check if the current Arc-Potential probability of edge $(v_i v_j)$ towards a destination region D_k for a time t is larger in comparison with time pr_{v_j} and if so, we set the Arc-Potential probability of edge $(v_i v_j)$ towards a destination region D_k for a time t to be equal to the time pr_{v_j}.

VI. Experiments

In this section we provide experimental results obtained by using a real world GPS trajectories data.

A. Setup

To conduct experiments, a road network of Aalborg, Denmark has been used. It consists of 4.142 nodes and 9,258 edges. 37 million GPS records that occurred in Aalborg from Jan 2007 to Dec 2008 with a sampling rate of one GPS record per second, i.e. 1 Hz, have been used to instantiate the PACE model. Because not all edges are covered by the GPS data, we divide the length of an edge by the speed limit of the same edge to derive a travel time values. Additionally, only paths traversed by more than 10 trajectories are considered to be instantiated using trajectory data. If less then 10 trajectories traverse a path, speed limits have been used to derive joint distributions. Figure 6 provided by [1], shows a visual representation of the edges which are covered and also not covered by the GPS trajectories. The red color is used for edges which have been covered by the trajectory data, the blue color is used for edges that are not covered by any GPS trajectory, which means that speed limits have been used to derived a travel cost.

Preprocessing: In the conducted experiments we consider various settings for Arc-Potentials preprocessing. The preprocessing involves, first, computing the SOTA optimal policy up to a desired time budget, and second, derive Arc-Potentials from the optimal policy. Next, Arc-Potentials are used as heuristic to find the SPOTAR solution efficiently. Computing the SOTA optimal policy takes most of the time for preprocessing. The computation must be preformed for all sources, all destination and all time budgets. For the network used in the experiments, it took around 24 hours while executing the SOTA optimal policy computation in parallel on the server machine on 32 cores.

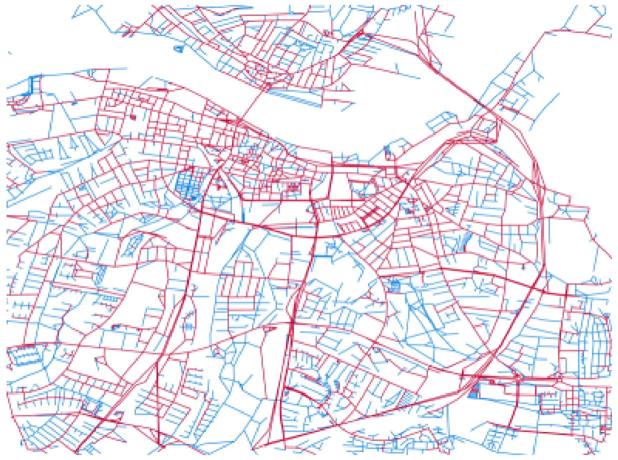


Fig. 6: Aalborg network. The red segments have been rendered thanks to the trajectory GPS data. For these segments exists enough information to derived travel cost distribution. For the blue segments the speed limits have been used to derive travel cost. [1]

In the experiments, we use Arc-Potentials with rectangular partition scheme. We use four different rectangular partitions 7x7, 15x15, 25x25 and maximum regions(one vertex per one destination). We report on running time for preprocessing and the space consumed by the Arc-Potentials for each partition.

Queries: In the conducted experiments we consider various settings to generate SPOTAR queries.

First, we vary the time budget (seconds) from 300, 500, 700, to 1,000. Second, we vary the Euclidean distance (km) between source-destination pairs: $[0, 1)$, $[1, 2)$, $[2, 3)$, and $[3, 4)$. For each of the settings, we randomly generate 20 source-destination pairs. Third, we compare our proposal for solving the SPOTAR problem using different heuristic functions: (1) the proposed solution which uses Arc-Potentials obtained by the SOTA policy where the PACE road network model has been applied. Additionally, we use different partitions to for Arc-Potentials, i.e. 7x7, 15x15, 25x25 and maximum regions(one vertex per one destination) to examine how they affect the SPOTAR solution; (2) the minimum travel time to the destination using shortest path trees (SP) [1]; (3) the baseline heuristic using Euclidean distance divided by the maximum speed limit (BA) [1];

Evaluation Metrics: We report on average run times and sizes of search space for running SPOTAR in different settings. We also report on average run times and sizes consumed for preprocessing with Arc-Potentials.

The experiments have been conducted on two computers. First, a powerful server needed to preprocess the network, i.e. it has to compute the SOTA policy up to a desired time budget and then it derive Arc-Potentials from the policy. The computation can be done in parallel, therefore a multi-core processor were used to speed up the computation. The server HP 585 with 4x16 cores working on 2.294 GHz, 512 GB RAM, 1 TB 7,2k RPM disk. The second less powerfull work station used to execute the queries with the following characteristics: Intel® Core™ i5-4210U CPU @ 1.70 GHz × 4 processors with 12 GB RAM with 64 bit Linux Fedora 25 operation system. The code was implemented in Python 3.

B. *Experimental Results*

Preprocessing:

We report on the preprocessing time as well as the space consumed by the AP using different partitions. We report on 7x7, 15x15, 25x25 and maximum regions where each region has one vertex from the graph and the number of regions is the same as the number of vertices in the graph. We measure the space and time used for generating Arc-Potentials.

Preprocessing runtime: We measure the running time needed to generate AP. The results are shown in Figure 7 b). It is clear that the running time needed for generating Arc-Potentials increase when the number of regions increases.

Preprocessing space consumption: Figure 7 a) shows that the space consumed by the Arc-Potential increases when the number of region increase

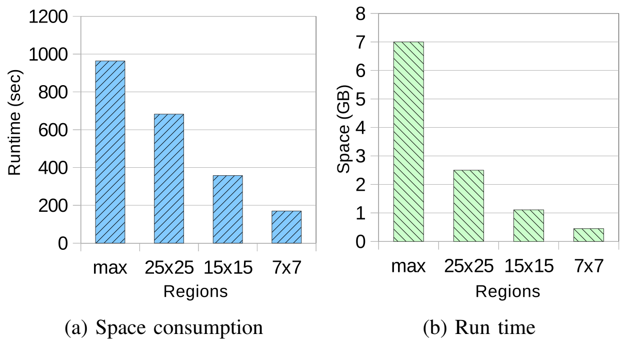
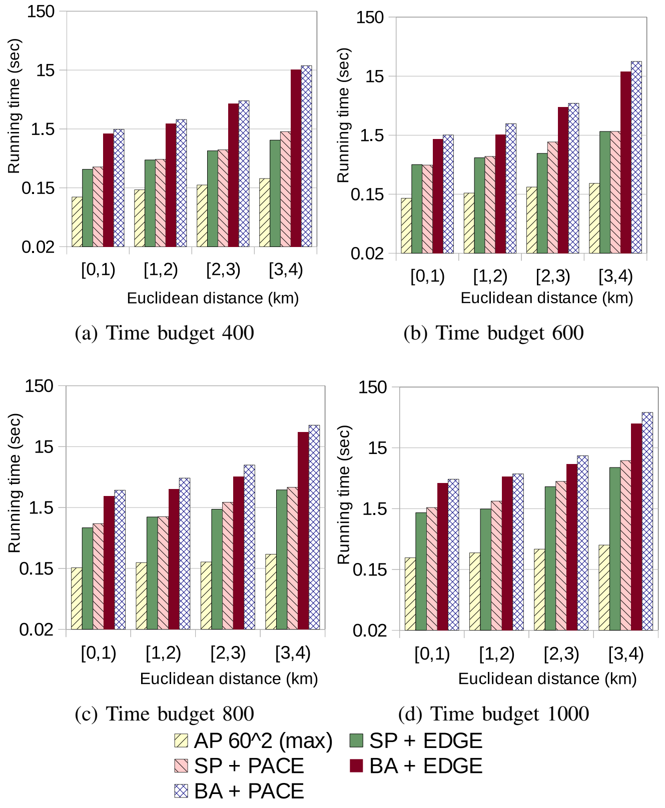


Fig. 7: Preprocessing with Arc-Potentials

Queries:

Queries runtime:

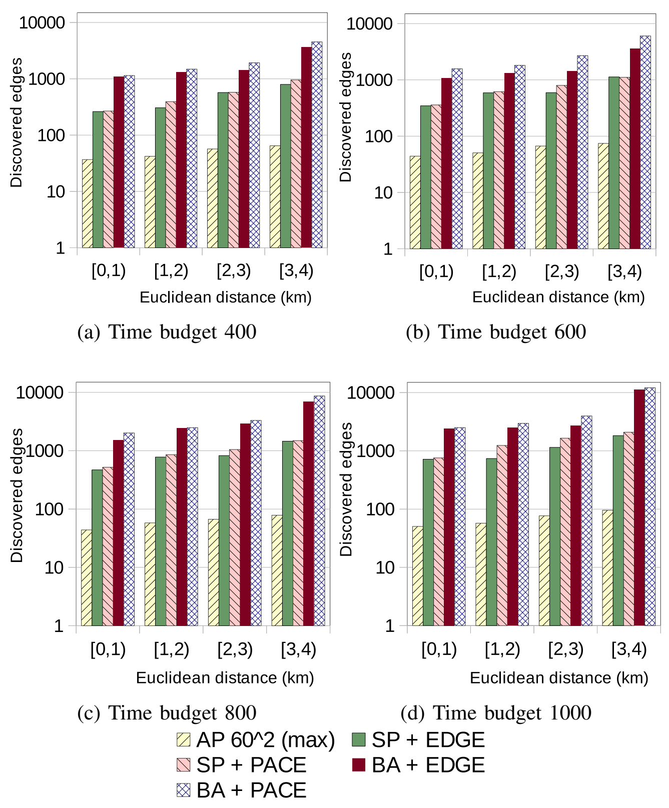
Figure 8 shows the running times when using three methods (AP max, SP and BA) under different settings. We observe that when the distance between a source-destination pair increases, the running times of all methods also increases. Figure 12 also shows that when the time budget increases the running time of all methods also increase. The AP max heuristic shows significantly better running times in comparison with the SP and BA heuristic under all settings. The SP shows better running time in comparison with the BA heuristic but it is outperformed by the AP max. In addition, it is clear that the slowest runtime growth when distance between source-destination increases has been achieved by AP max. The fastest runtime growth when distance between source-destination increases has been achieved by the BA heuristic. SP heuristic growth is faster than the AP max and slower than the BA heuristic. The growth of BA is significantly faster in comparison with the AP heuristic as the distance between a source-destination pair increases.



(a) Time budget 400 (b) Time budget 600

(c) Time budget 800 (d) Time budget 1000

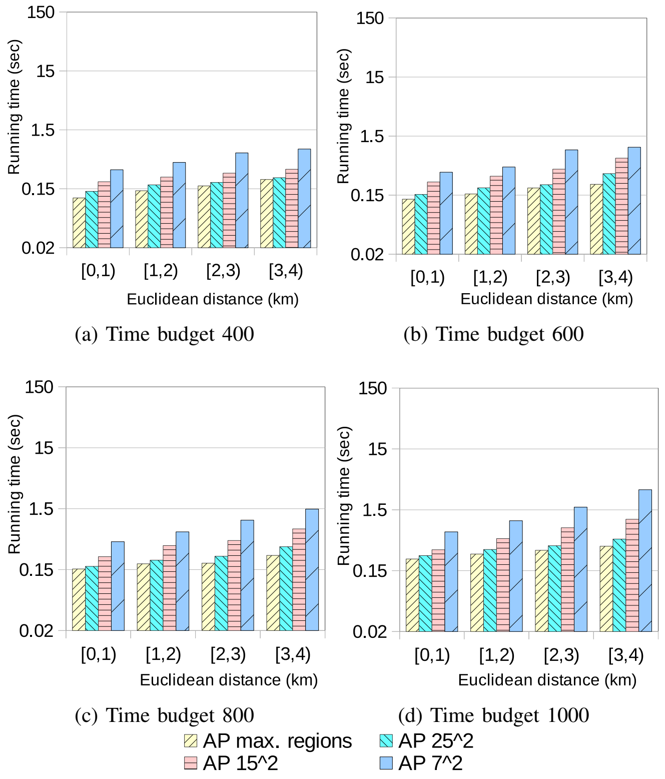
Fig. 8: Runtime



(a) Time budget 400 (b) Time budget 600

(c) Time budget 800 (d) Time budget 1000

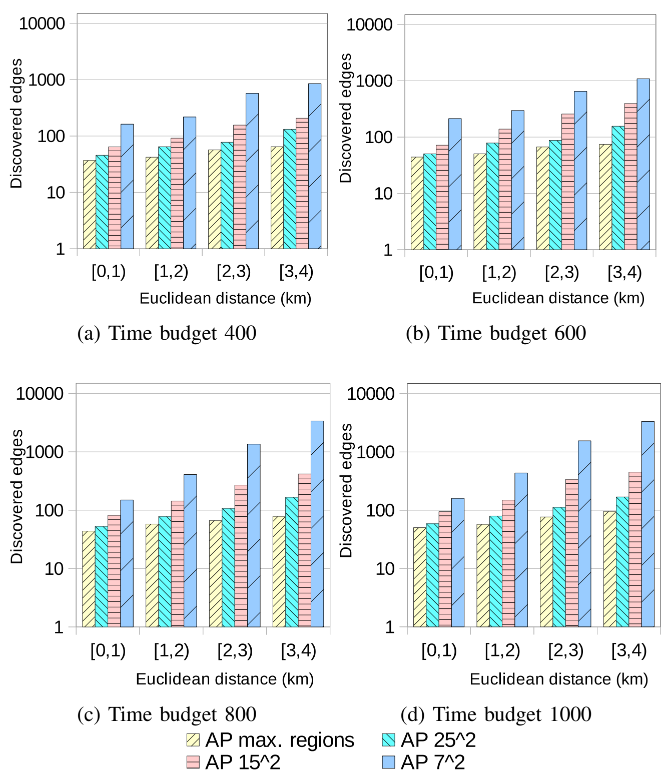
Fig. 10: Search Space



(a) Time budget 400 (b) Time budget 600

(c) Time budget 800 (d) Time budget 1000

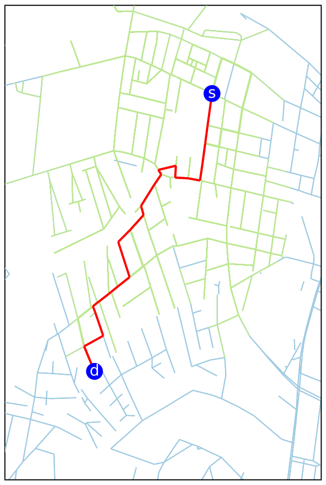
Fig. 9: Runtime AP



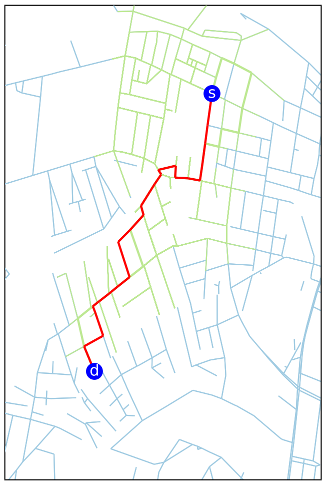
(a) Time budget 400 (b) Time budget 600

(c) Time budget 800 (d) Time budget 1000

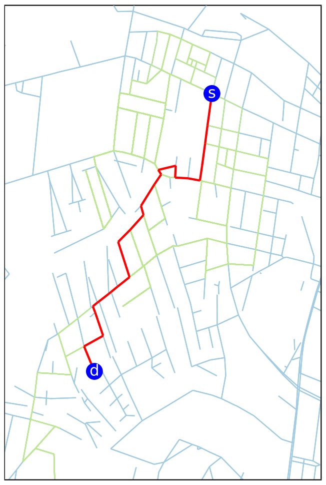
Fig. 11: Search Space AP



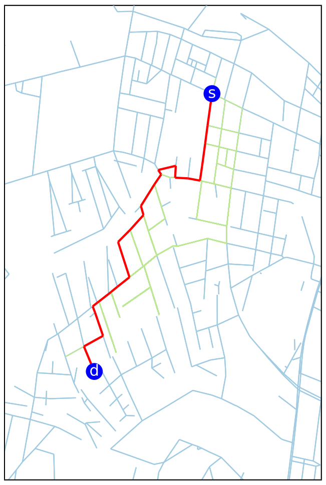
(a) BA



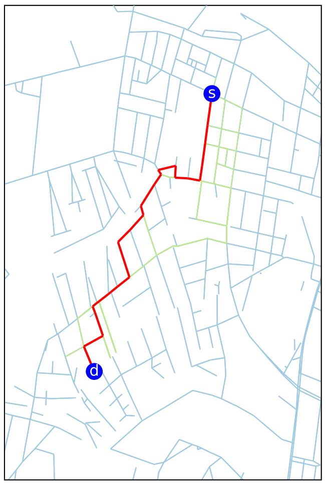
(b) SP



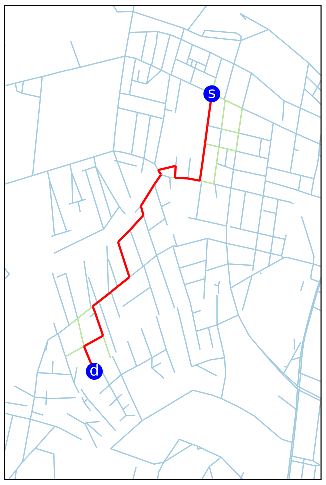
(c) AP 7^2 regions



(d) AP 15^2 regions



(e) AP 25^2 regions



(f) AP maximum regions

Fig. 12: Pruning of Aalborg network

Figure 9 shows the runtime of the SPOTAR solution using Arc-Potential heuristic with different number of destination regions. It can be seen that when the distance between source-destination pair increases the running time also increases, it is also true that when the time budget increases the running time also increases. For all settings the best running time has been achieved by using Arc-Potential with maximum number of partitions (one per destination region). The slowest running time has been achieved by the Arc-Potential with 7x7 regions. It shows running time closer to SP heuristic. The running time of the SPOTAR solution using Arc-Potential with 25x25 regions shows running time very close to the running time of Arc-Potential with maximum number of partitions. According to Figure 9 Arc-Potential with 25x25 regions will consume around 70% of the space required by Arc-Potential with maximum number of partitions. Arc-Potential with 15x15 partitions shows faster running time in comparison with AP 7x7, SP and BA. It consumes only 30% of the space used by AP max. regions.

Queries search space: Now, we show the size of the search space which has been discovered by different methods in order to find a solution to the SPOTAR problem. In this paper, we define the search space to be the edges that have been explored by a method under given settings.

We are further showing comparison between the search spaces explored by different methods in combination with different settings.

The total amount of edges discovered using AP max,SP and BA heuristics, classified by four time budgets can be seen in Figure 10. The results from the figure suggest that the search space explored by AP is the smallest, while the search space discovered by BA is the largest. SP search space is less than the BA and larger compare to AP max. This holds for all time budgets and all distances between source-destination pairs. The search space increase when the distance between a source-destination pair increases for all methods. The search space also increases for all methods when the time budget increases. We report the smallest growth of the explored searched space by using AP max heuristic. SP and BA growths are consider to be faster in comparison with AP max.

Figure 11 shows the searched space explored by the SPOTAR solution using Arc-Potential with different number of destination regions, i.e. 7x7, 15x15, 25x25 and maximum regions. When the distance between source and destination increases, the running time also increases for all methods. The smallest searched space was explored by AP max regions. The largest search space was discovered by AP 7x7. The number of edges explored by AP 25x25 is close to AP max regions. AP 15x15 explored less searched space in comparisons with AP 7x7, SP and BA.

Finally, we show visually the searched space discovered by the algorithm for solving SPOTAR using different heuristic, i.e. Arc-Potentials, SP, BA. The results are presented in Figure 12

VII. Conclusions and Outlook

We investigate an efficient solution of the arriving on time problem which is important for many modern transportation systems and services. We present an effective heuristic function which can be used with our proposal for solving the SPOTAR problem. We apply the PACE model in the SOTA policy, and then we derive Arc-Potentials to solve SPOTAR problem efficiently considering the travel time dependencies of sequence of edges, i.e. paths. We provide experimental results using real-world trajectories which suggest that the proposed algorithm is effective.

In the future, we plan to improve the scalability of the proposal by increasing the discretization factor. This decreases the times for which the optimal policy must be computed by a constant factor, which must reduce the query time.

References

[1] G. Andonov, B. Yang, “Arriving On Time: Stochastic Routing in Path-Centric Uncertain Road Networks”, MDM 2018.

[2] J. Dai, B. Yang, C Guo, C. S. Jensen, and J. Hu, “Path Cost Distribution Estimation Using Trajectory Data”, PVLDB 10(3): 85-96 (2016).

[3] B. Yang, J. Dai, C Guo, C. S. Jensen, and J. Hu, “PACE: a PAth-CEntric paradigm for stochastic path finding”, The VLDB Journal, online first.

[4] M. Niknami, S. Samaranayake, A. Bayen, “Tractable Pathfinding for the Stochastic On-Time Arrival Problem”, August, 2014

[5] G. Sabran, S. Samaranayake, A. Bayen, “Precomputation techniques for the stochastic on-time arrival problem”, 2014.

[6] Yu (Marco) Nie and Xing Wu, “Shortest path problem considering on-time arrival probability.“, Transportation Research Part B: Methodological, 2009.

[7] S. Lim, C. Sommer, E. Nikolova, and D. Rus. Practical route planning under delay uncertainty: Stochastic shortest path queries. Robotics: Science and Systems, 8(32):249–256, 2013.

[8] B. Yang, C. Guo, C. S. Jensen, M. Kaul, and S. Shang. Stochastic skyline route planning under time-varying uncertainty. In ICDE, pages 136–147, 2014.

[9] B. Yang, M. Kaul, and C. S. Jensen. Using incomplete information for complete weight annotation of road networks. TKDE, 26(5):1267–1279, 2014.

[10] C. Guo, B. Yang, J. Hu, C. S. Jensen. Learning to Route with Sparse Trajectory Sets. ICDE, 12 pages, 2018.

[11] H. Liu, C. Jin, B. Yang, A. Zhou. Finding Top-k Optimal Sequenced Routes. ICDE, 12 pages, 2018.

[12] H. Liu, C. Jin, B. Yang, A. Zhou. Finding Top-k Shortest Paths with Diversity. TKDE, 30(3):488-502, 2018.

[13] J. Hu, B. Yang, C. Guo, C. S. Jensen. Risk-aware path selection with time-varying, uncertain travel costs: a time series approach. The VLDB Journal 27(2):179-200, 2018.

[14] J. Hu, B. Yang, C. S. Jensen, Y. Ma. Enabling time-dependent uncertain eco-weights for road networks. GeoInformatica 21(1):57-88, 2017.

[15] Z. Ding, B. Yang, R. H. Güting, Y. Li. Network-Matched Trajectory-Based Moving-Object Database: Models and Applications. IEEE Trans. Intelligent Transportation Systems 16(4):1918-1928, 2015.

[16] B. Yang, C. Guo, Y. Ma, and C. S. Jensen. Toward personalized, context-aware routing. VLDB Journal, 24(2):297–318, 2015.

[17] M. Hilger, E. Köhler, R Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. The Shortest Path Problem: Ninth DIMACS Implementation Challenge, 74:41–72, 2009.

[18] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In ALENEX/ANALC, pages 100–111, 2004.

[19] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A* : Efficient point-to-point shortest path algorithms. In ALENEX, volume 6, pages 129–143. SIAM, 2006.

[20] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies:Faster and simpler hierarchical routing in road networks. In Experimental Algorithms, pages 319–333. Springer, 2008.

[21] H. Bast, S. Funke, and D. Matijevic. Transit: ultrafast shortest-path queries with linear-time preprocessing. 9th DIMACS Implementation Challenge [1], 2006.

[22] Ch. Leiserson, Cl. Stein, R. Rivest and T. Cormen. Introduction to algorithms and DS, 4th edition, pages 319–369., Mit Press Ltd, ISBN13:9780262533058, 2009.
.