Computation of Dynamic Prediction Equilibria

A software project as part of the M.Sc. Mathematics at the University of Augsburg.

Michael Markl

November 17, 2022

Supervised by Prof. Dr. Tobias Harks

Contents

1 Introduction

When analyzing real-world traffic or communication networks, often simulations are used to retrieve structural information on the performance of the network given certain demands. Starting from a more theoretical perspective, equilibria of underlying game-theoretic models propose to give more fundamental insights. In this software project, we aim to compute such an equilibrium called Dynamic Prediction Equilibrium (DPE) which was recently introduced in [4]. The fundamental dynamics adhere to Vickrey's fluid queuing model in which infinitesimal particles enter edges. Once the amount of flow exceeds an edge's capacity, a point queue will form in front of it.

In a DPE, each commodity has a source-sink pair as well as some mean of predicting future travel times of any edge. An agent of a commodity, an infinitesimal particle traveling from the commodity's source s to its sink t, computes a shortest path to the sink t every time it arrives at a node v by instantiating a new prediction on the future edge costs. Then, it enters an outgoing edge of v lying on that path. The predictions of a commodity i are modeled as a function $\hat{q}_{i,e}(\theta, \bar{\theta}, q)$ that given the past queue lengths $q : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$ will return the predicted queue length of edge e at time θ as predicted at time $\bar{\theta}$. If all commodities use a perfect predictor, which predicts all queue lengths correctly, the resulting equilibrium flow is called a Dynamic Equilibrium which has been studied thoroughly in literature, for example in [1].

In this document, we first take a closer look at general time-dependent cost functions following the First-In-First-Out (FIFO) rule in Section 2: We discuss a duality of arrival and departure times that can be used for a simple Dijkstra-based algorithm to compute all so-called *active edges*. In Section 3 we describe the mathematical flow model and define DPEs within this model. The computation of an approximation of such an equilibrium is described in detail in Section 4. Finally, we run our developed algorithms in Section 5 on a small synthetic network and multiple real world traffic networks.

All code snippets presented in this work use the Python 3.8 syntax where type hints (e.g. `i: int`) are often used to make the underlying data encoding clearer. The presented source code is often a simplified version of the original code such that the reader is not distracted by technical details which might improve the performance. Moreover, in this work we abstract from any kind of rounding errors due to the fixed size floating point numbers. It is worth mentioning though, that fixing these kinds of errors was quite laborious. The full source code of this software project is publicly available at `https://github.com/schedulaar/predicted-dynamic-flows`.

2 Time-Dependent FIFO Cost Functions

Time-dependent cost functions that follow the First In First Out (FIFO) rule are very important for the algorithms introduced later. We begin by defining the FIFO rule in a directed graph (V, E) representing our network. Here, we work on a finite set of nodes V and a finite set of edges E. Although, we allow parallel edges, we often write $e = vw \in E$ for a directed edge e from node v to node w. For a node v, we denote the set of *outgoing edges* of v as $\delta_v^- := \{vw \in E\}$ and the set of *incoming edges* as $\delta_v^+ := \{uv \in E\}$.

Definition 2.1. A time-dependent cost function $c : \mathbb{R} \to \mathbb{R}_{\geq 0}^E$ *follows the FIFO rule* if for all edges $e \in E$ the function $T_e : \theta \mapsto \theta + c_e(\theta)$ is monotone increasing. It *follows the strong FIFO rule* if T_e is strictly increasing for all $e \in E$. The function T_e is called *traversal time* of e.

For the following we concentrate on a graph (V, E) where all nodes can reach a specific sink node $t \in V$. Moreover, let $c : \mathbb{R} \to \mathbb{R}_{\geq 0}^E$ be a time-dependent cost function.

The traversal time T_P of a path $P = e_1 \cdots e_k$ is given by the concatenation of the edges' traversal times as $T_P := T_{e_k} \circ \cdots \circ T_{e_1}$. The set of all simple v-t-Paths is denoted as $\mathcal{P}_{v,t}$. The *earliest arrival time* at t when starting at time θ in v is then given by $l_{v,t}(\theta) := \min_{P \in \mathcal{P}_{v,t}} T_P(\theta)$. Paths that attain this minimum are called *shortest v-t-Path at time θ*.

We call an edge $e = vw \in E$ *active at time* θ, if the condition $l_{v,t}(\theta) = l_{w,t}(T_e(\theta))$ holds true. The set of active edges at time θ is collected in $E(\theta)$. It should be noted however, that an active edge does not necessarily lie on a *simple* shortest path.

2.1 Properties of FIFO Cost Functions

To make sure that we can ignore non-simple paths in the definition of shortest paths, we exploit the FIFO property of the cost functions:

Proposition 2.2. *Given cost functions $c : \mathbb{R} \to \mathbb{R}_{\geq 0}^E$ following the FIFO rule, removing a cycle in any v-t-path P does not increase the path's traversal time.*

More specifically, if $P = P_1 C P_2$ is the concatenation of paths P_1, a cycle C and another path P_2, then it holds that $T_P(\theta) \geq (T_{P_2} \circ T_{P_1})(\theta)$ for all $\theta \in \mathbb{R}$.

Proof. The statement is a direct consequence of the monotonicity of T_{P_2} and the fact that $T_C(\theta) \geq \theta$ holds for all $\theta \in \mathbb{R}$. □

Proposition 2.3. *Let $c : \mathbb{R} \to \mathbb{R}_{\geq 0}^E$ be a cost function following the FIFO rule. The vector $(l_{v,t})_{v \in V}$ of functions is the (pointwise) maximal solution of the following system of equations in the function-valued variables $(\tilde{l}_v : \mathbb{R} \to \mathbb{R})_{v \in V}$:*

$$\tilde{l}_v(\theta) = \begin{cases} \theta, & \text{if } v = t, \\ \min_{e=vw \in \delta_v^-} \tilde{l}_w\left(T_e(\theta)\right), & \text{otherwise.} \end{cases}$$

Proof. To see that $(l_{v,t})_{v\in V}$ is a solution of the system, we note that for $v = t$ we have $l_{v,t}(\theta) = \theta$ for all $\theta \in \mathbb{R}$. For $v \neq t$ let $e = vw \in \delta_v^-$. Let P be a shortest w-t-path at time $T_e(\theta)$. Let $P' \coloneqq e \circ P$ be the concatenation of e and P.

If P' contains a cycle, then it must have been introduced by e and v must be visited a second time in P'. Removing this cycle gives a simple v-t-path Q with

$$l_{v,t}(\theta) \leq T_Q(\theta) \leq T_{P'}(\theta) = l_{w,t}(T_e(\theta))$$

by Proposition 2.2. If P' does not contain a cycle, we follow analogously

$$l_{v,t}(\theta) \leq T_{P'}(\theta) = l_{w,t}(T_e(\theta)).$$

This shows that $l_{v,t}(\theta)$ is a lower bound on $\{l_{w,t}(T_e(\theta)) \mid e = vw \in \delta_v^-\}$. Let $P = e_1 \cdots e_k$ be a shortest v-t-path at time θ and let $e_1 = vw$ and $P' = e_2 \cdots e_k$. Furthermore, let Q be a shortest w-t-path at time $T_{e_1}(\theta)$. Assume $T_P(\theta) > l_{w,t}(T_{e_1}(\theta))$. This implies

$$T_P(\theta) > T_Q(T_{e_1}(\theta)),$$

such that removing any cycles in the path $e_1 \circ Q$ would yield a strictly shorter path than P; a contradiction.

We now have to show that $(l_{v,t})_{v\in V}$ is the maximal solution. That means that for any other solution $(\tilde{l}_v)_{v\in V}$ of the system of equations $\tilde{l}_v(\theta) \leq l_v(\theta)$ should hold for all $v \in V$ and $\theta \in \mathbb{R}$. Let $P = e_1 \cdots e_k$ be a shortest v-t-path at time θ and let $v_0 = v, v_k = t$ and $e_i = v_{i-1}v_i$ for $i = 1, \ldots, k$. Then by the system of equations we infer

$$\tilde{l}_v(\theta) \leq \tilde{l}_{v_1}(T_{e_1}(\theta)) \leq \tilde{l}_{v_2,t}(T_{e_1\,e_2}(\theta)) \leq \cdots \leq \tilde{l}_t(T_P(\theta)) = T_P(\theta) = l_{v,t}(\theta).$$

□

The following example illustrates why the system of equations above does not always have a unique solution. The method used works in any cyclic graph: We build a self-confirming cycle proposing an earlier arrival time than actually possible.

Example 2.4. A minimal example without loops consists of three nodes $V = \{s, v, t\}$ with three edges $E = \{st, sv, vs\}$ whose cost functions $c_e : \mathbb{R} \to \mathbb{R}_{\geq 0}$ follow the FIFO rule. We virtually limit the arrival time of s at 0 using $\tilde{l}_s(\theta) \coloneqq \min\{T_{st}(\theta), 0\}$. We define the arrival times of v and t as expected with $\tilde{l}_t(\theta) \coloneqq \theta$ and $\tilde{l}_v(\theta) \coloneqq \tilde{l}_s(T_{vs}(\theta))$. Obviously, the system of equations is satisfied for nodes v and t. For node s, the equation reads

$$\tilde{l}_s(\theta) = \min\left\{\tilde{l}_t(T_{st}(\theta)), \tilde{l}_v(T_{sv}(\theta))\right\}.$$

By inserting the definitions of $\tilde{l}_v$ and $\tilde{l}_t$, the right-hand side equates to

$$\min\left\{T_{st}(\theta), \tilde{l}_s(T_{vs}(T_{sv}(\theta)))\right\} = \min\left\{T_{st}(\theta), T_{st}\Big(T_{vs}(T_{sv}(\theta))\Big), 0\right\}.$$

Proposition 2.2 implies the inequality $T_{st}(\theta) \leq T_{st}(T_{vs}(T_{sv}(\theta)))$, so that the right-hand side reduces to $\min\{T_{st}(\theta), 0\} = \tilde{l}_s(\theta)$. Obviously, this solution $(\tilde{l}_w)_{w\in V}$ is different to the actual arrival times $(l_w)_{w\in V}$: The arrival time when starting in s at any positive time θ fulfills $l_s(\theta) = T_{st}(\theta) \geq \theta > 0 = \tilde{l}_s(\theta)$.

2.2 Duality of Arrival and Departure Times

Sometimes it is useful not to work on the earliest arrival time, but the latest possible departure time. To enable this switch, we define a kind of inverse of a monotonically increasing function:

Definition 2.5. We define the function space

$$\mathcal{F} \coloneqq \left\{ f : \mathbb{R} \to \mathbb{R} \mid \ f \text{ is increasing and } \lim_{|x|\to\infty} |f(x)| = \infty \right\}.$$

The *reversal of* $f \in \mathcal{F}$ is defined as

$$f^{\leftarrow} : \mathbb{R} \to \mathbb{R},\ \theta \mapsto \sup\{\xi \in \mathbb{R} \mid f(\xi) \leq \theta\}.$$

We can interpret the reversal of f in the following way: If $f(\theta)$ is the earliest arrival time when departing at time θ, then $f^{\leftarrow}(\theta)$ is the latest departure time for arriving before or at time θ.

Proposition 2.6. *For $f, g \in \mathcal{F}$ the following statements are true:*

(i) It holds that $f^{\leftarrow} \in \mathcal{F}$, i.e. $f^{\leftarrow}$ is increasing and $\lim_{|x|\to\infty} |f(x)| = \infty$.

(ii) If f is continuous, then $f^{\leftarrow}(\theta) = \max\{\xi \in \mathbb{R} \mid f(\xi) = \theta\}$ holds for all $\theta \in \mathbb{R}$ and $f^{\leftarrow}$ is strictly increasing. Moreover, we have $f \circ f^{\leftarrow} = \mathbf{id}_{\mathbb{R}}$.

(iii) If f is continuous and strictly increasing, then $f^{\leftarrow}$ is the inverse of f.

(iv) It holds that $(g \circ f)^{\leftarrow} = f^{\leftarrow} \circ g^{\leftarrow}$ and $(\min\{f, g\})^{\leftarrow} = \max\{f^{\leftarrow}, g^{\leftarrow}\}$.

(v) If $f \geq \mathbf{id}_{\mathbb{R}}$ holds pointwise, then so does $f^{\leftarrow} \leq \mathbf{id}_{\mathbb{R}}$.

Proof. (i). Let $\theta_1, \theta_2 \in \mathbb{R}$ with $\theta_1 < \theta_2$. Then

$$f^{\leftarrow}(\theta_1) = \sup\{\xi \in \mathbb{R} \mid f(\xi) \leq \theta_1\} \leq \sup\{\xi \in \mathbb{R} \mid f(\xi) \leq \theta_2\} = f^{\leftarrow}(\theta_2) \tag{1}$$

implies the monotonicity. From $\lim_{|\theta|\to\infty} |f(\theta)|$ and the monotonicity of f we conclude

$$\lim_{|\theta|\to\infty} |f^{\leftarrow}(\theta)| = \lim_{|\theta|\to\infty} |\sup\{\xi \in \mathbb{R} \mid f(\xi) \leq \theta\}| = \infty.$$

(ii). We note, that $\{\xi \in \mathbb{R} \mid f(\xi) = \theta\}$ is non-empty, closed and bounded from above because of the condition $\lim_{|\theta|\to\infty} |f(\theta)| = \infty$ and the continuity and monotonicity of f. The reversal $f^{\leftarrow}$ is strictly increasing as the inequality (1) is strict for continuous f.

(iii). Let f be continuous and strictly increasing. Then it holds that

$$f^{\leftarrow}(\theta) = \max\{\xi \mid f(\xi) = \theta\} = f^{-1}(\theta).$$

(iv). After inserting the definition the first statement evaluated in θ becomes

$$l \coloneqq \sup\{\xi \mid g(f(\xi)) \leq \theta\} = \sup\{\xi_f \mid f(\xi_f) \leq \sup\{\xi_g \mid g(\xi_g) \leq \theta\}\} =: r.$$

Let $\xi_f \in \mathbb{R}$ fulfill $f(\xi_f) \leq g^{\leftarrow}(\theta)$. Then for all ξ_g with $g(\xi_g) \leq \theta$ we have $f(\xi_f) \leq \xi_g$. Then because of the monotonicity of g and f we follow $g(f(\xi_f)) \leq g(\xi_g) \leq \theta$ which implies $l \geq r$. To see that $r \geq l$ holds, any $\xi \in \mathbb{R}$ with $g(f(\xi)) \leq \theta$ fulfills $f(\xi) \leq g^{\leftarrow}(\theta)$.

The statement on the minimum of f and g evaluated in θ is of the form

$$l \coloneqq \sup\{\xi \mid \min\{f(\xi), g(\xi)\} \leq \theta\} = \max\{\sup\{\xi \mid f(\xi) \leq \theta\}, \sup\{\xi \mid g(\xi) \leq \theta\}\} =: r.$$

Let $\xi \in \mathbb{R}$ fulfill $\min\{f(\xi), g(\xi)\} \leq \theta$ and assume $f(\xi) \leq g(\xi)$ without loss of generality. Then it holds that $f(\xi) \leq \theta$ and therefore $\xi \leq f^{\leftarrow}(\theta) \leq r$ implying $l \leq r$. On the contrary, assume $f^{\leftarrow}(\theta) \leq g^{\leftarrow}(\theta)$ without loss of generality, so that $r = g^{\leftarrow}(\theta)$. Any $\xi \in \mathbb{R}$ with $g(\xi) \leq \theta$ fulfills $\min\{f(\xi), g(\xi)\} \leq g(\xi) \leq \theta$ and hence $l \geq r$ holds true.

(v). For $f \geq \mathbf{id}_R$ we deduce for all $\theta \in \mathbb{R}$

$$f^{\leftarrow}(\theta) = \sup\{\xi \in \mathbb{R} \mid f(\xi) \leq \theta\} \leq \sup\{\xi \in \mathbb{R} \mid \xi \leq \theta\} = \theta. \qquad \square$$

Corollary 2.7. *For a cost-function $c : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$, that follows the FIFO rule and that fulfills $\lim_{\theta \to -\infty} T_e(\theta) = -\infty$, the following statements hold true:*

(i) For all edges $e \in E$ we have $T_e \in \mathcal{F}$.

(ii) For any path $P = e_1 \cdots e_k$ it holds that $T_P^{\leftarrow} = T_{e_1}^{\leftarrow} \circ \cdots \circ T_{e_k}^{\leftarrow}$.

(iii) For any node $v \in V$ it holds that $l_{v,t}^{\leftarrow} = \max_{P \in \mathcal{P}_{v,t}} T_P^{\leftarrow}$.

The reversal of a function can be utilized for a characterization of active edges in the case of continuous cost function:

Lemma 2.8. *Let $c : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$ be a continuous cost function following the FIFO rule with $\lim_{\theta \to -\infty} T_e(\theta) = -\infty$ for all $e \in E$ and let $t \in V$ be a sink node. Then an edge $e = vw$ is active at time θ if and only if $T_e(\theta) \leq l_{w,t}^{\leftarrow}(l_{v,t}(\theta))$.*

Proof. Let edge e be active at time θ, i.e. $l_{w,t}(T_e(\theta)) \leq l_{v,t}(\theta)$. By definition of the reversal this already implies $l_{w,t}^{\leftarrow}(l_{v,t}(\theta)) \geq T_e(\theta)$.

If on the contrary $T_e(\theta) \leq l_{w,t}^{\leftarrow}(l_{v,t}(\theta))$ holds, then the monotonicity of $l_{w,t}$ implies

$$l_{w,t}(T_e(\theta)) \leq l_{w,t}\left(l_{w,t}^{\leftarrow}(l_{v,t}(\theta))\right),$$

and by the continuity of $l_{w,t}$ the claim follows from Proposition 2.6 (ii) which states that $l_{w,t} \circ l_{w,t}^{\leftarrow} = \mathbf{id}_{\mathbb{R}}$. $\square$

2.3 The Dynamic Dijkstra Algorithm

The first algorithm we discuss is a simple modification of the Dijkstra Algorithm to determine the earliest arrival times $(l_{s,w}(\theta))_{w \in V'}$ at nodes of some subset $V' \subseteq V$ when departing from a source node s at time θ. Here, only those nodes are relevant for us that are reachable from s and that can reach t. Moreover, we only need to determine the arrival times of nodes w that can be reached before t, i.e. that fulfill $l_{s,w}(\theta) \leq l_{s,t}(\theta)$.

Algorithm 1 The Dynamic Dijkstra Algorithm

```
def dynamic_dijkstra(
    theta: float, source: Node, sink: Node, relevant_nodes: Set[Node],
    costs: List[Callable[[float], float]]
) -> Dict[Node, float]:
  arrival_times: Dict[Node, float] = {}
  queue: PriorityQueue[Node] = PriorityQueue([(source, theta)])
  while len(queue) > 0:
    arrival_time, v = queue.min_key(), queue.pop()
    arrival_times[v] = arrival_time
    if v == sink:
      break
    for e in v.outgoing_edges:
      w = e.node_to
      if w in arrival_times.keys() or w not in relevant_nodes:
        continue
      relaxation = arrival_time + costs[e.id](arrival_time)
      if not queue.contains(w):
        queue.push(w, relaxation)
      elif relaxation < queue.key_of(w):
        queue.decrease_key(w, relaxation)
  return arrival_times
```

Hence, the set V' consists of all nodes w that lie on a path from v to t and fulfill $l_{s,w}(\theta) \leq l_{s,t}(\theta)$.

Adjusting the classical Dijkstra Algorithm for static edge costs to our setting yields the Dynamic Dijkstra Algorithm as depicted in Algorithm 1.

A priority queue, consisting of items together with a priority key associated with each item, operates at the heart of the algorithm. This queue has to support the operations `push(item, key)`, `min_key()`, `pop()`, `decrease_key(item, new_key)` as well as `contains(item)`. The operation `push(item, key)` adds the item `item` with priority `key` to the queue, `min_key()` returns the minimum key of an item in the queue, `pop()` returns the item with minimum key and removes it from the queue, `contains(item)` returns whether `item` is contained in the queue and the operation `decrease_key(item, new_key)` replaces the priority key associated to the item `item` with `new_key`.

The idea of Algorithm 1 is to iteratively retrieve an unvisited node v with the current earliest arrival time t of all unvisited nodes. Then, for each outgoing edge $e = vw$ we realize its cost at time t and update the arrival time of the target node w. The priority queue holds all discovered but unvisited nodes where the priority key of a node w is its currently suspected earliest arrival time, an upper bound on $l_{s,w}(\theta)$. Nodes that are not added to the queue have either been already visited and thus have an entry in

`arrival_times` or are not discovered and hence have an imaginary arrival time of ∞.

In the following proposition we show the correctness of the algorithm:

Proposition 2.9. *Given cost functions $c : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$ following the FIFO rule, the Dynamic Dijkstra Algorithm initiated on the source $s \in V$ and a reachable sink $t \in V$ computes the vector $(l_{s,w}(\theta))_{w \in V'}$ with*

$$V' = \{w \in V \mid w \text{ lies on an } s\text{-}t\text{-path and } l_{s,w}(\theta) \leq l_{s,t}(\theta)\}.$$

Proof. As an invariant for the algorithm we proof, that once a value of a node w is written to `arrival_times[w]`, this value coincides with $l_{s,w}(\theta)$. In the beginning this is clearly true as `arrival_times` is initially empty. In the first iteration of the loop, the variable `v` holds the source s and its key $\theta = l_{s,s}(\theta)$ is written to `arrival_times[v]`. Assume the loop invariant holds before entering the body of the loop again at a later time and let v be the popped node. As nodes are added at most once to the queue, we have $v \neq s$. Let u be the node in whose iteration v was added to the queue or the key of v in the queue was decreased the most recently. The invariant implies `arrival_times[u]` $= l_{v,u}(\theta)$ and thus `arrival_time` $= l_{s,u}(\theta) + c_{uv}(l_{s,u}(\theta)) = T_{uv}(l_{s,u}(\theta)) \geq l_{s,v}(\theta)$.

Assume `arrival_time` $> l_{s,v}(\theta)$ and let P be a shortest s-v-path at time θ. If all nodes of P were available in `arrival_times`, then the last node before v in P would have set the key of v in its iteration to $l_{s,v}(\theta)$. Let u be the first node in P that is not available in `arrival_times`. Because u cannot be the source s, the predecessor u' of u in P must have set the key of u to at most $T_{u'u}(l_{s,u'}(\theta)) \leq T_P(\theta)$. As `arrival_time` $> l_{s,v}(\theta) = l_P(\theta)$ the key of u in the queue was smaller than the key of v, so the priority queue would have popped u before v. □

A simple binary min-heap together with a lookup table was implemented to support the operations of the queue efficiently. With this data structure, the worst case running time is logarithmic in the number of queue items for the operations `push(item, key)`, `pop()` and `decrease_key(item, new_key)` and constant for the operations `min_key()` and `contains(item)`. Thus, the Dynamic Dijkstra Algorithm terminates with a running time of $\mathcal{O}((|V| + |E|) \cdot \log |V|)$.

2.4 Computing Active Outgoing Edges

Given a FIFO cost function $c : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$ with nodes $s, t \in V$, a time $\theta \in \mathbb{R}$, we want to compute the active outgoing edges $E(\theta) \cap \delta_s^-$ of s, i.e. the edges $e = sw$ with

$$l_{s,t}(\theta) = l_{w,t}(T_e(\theta)).$$

Unfortunately, from the arrival times $(l_{s,w}(\theta))_{w \in V'}$ obtained by a simple run of the Dynamic Dijkstra Algorithm, we cannot determine all active edges: Usually, the idea is to backtrack all shortest paths by searching edges $e = vw$ backwards starting from t for which equality holds in $l_{s,w}(\theta) \leq T_e(l_{s,v}(\theta))$. This approach is described in Algorithm 2 which aims to return the set $E(\theta) \cap \delta_s^-$. The vector $(l_{s,w}(\theta))_{w \in V'}$ as returned by the Dynamic Dijkstra Algorithm is passed to the function via the parameter `arrivals`.

During the procedure, we only enqueue a node u to the queue `queue`, if there exists a path from u to t in which each edge $e = vw$ fulfills $l_{s,w}(\theta) = T_e(l_{s,v}(\theta))$. Once we discover an edge from s to such an enqueued node that also fulfills the equality, we add it to the set of active outgoing edges of s which is returned by the algorithm.

Algorithm 2 Retrieve Active Edges by Backtracking Shortest Paths

```
def backtrack_shortest_paths(
  costs: List[Callable[[float], float]], arrivals: Dict[Node, float],
  source: Node, sink: Node
) -> Set[Edge]:
  active_edges = set()
  queue: List[Node] = [sink]
  discovered: Set[Node] = {sink}
  while len(queue) > 0:
    w = queue.pop()
    for e in w.incoming_edges:
      v = e.node_from
      if v not in arrivals.keys():
        continue
      if arrivals[v] + costs[e.id](arrivals[v]) <= arrivals[w]:
        if v == source:
          active_edges.add(e)
        if v not in discovered:
          queue.append(v)
          discovered.add(v)
  return active_edges
```

The following proposition proves that paths found using the described approach are in fact shortest paths. Moreover, we show that for strong FIFO costs we find all shortest paths. That means, for strong FIFO costs, Algorithm 2 returns the whole set $E(\theta) \cap \delta_s^-$.

Proposition 2.10. *Let $P = e_1 \cdots e_k$ be a path with $e_i = v_{i-1}v_i$ and $v_0 = s, v_k = t$ and let $c : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$ be a cost function following the FIFO rule. Then*

$$(\forall i : T_{e_i}(l_{s,v_{i-1}}(\theta)) = l_{s,v_i}(\theta)) \implies T_P(\theta) = l_{s,t}(\theta).$$

If c follows the strong FIFO rule, the statements are equivalent.

Proof. Assume $T_{e_i}(l_{s,v_{i-1}}(\theta)) = l_{s,v_i}(\theta)$ holds for all i. Then we have

$$T_P(\theta) = T_P(l_{s,s}(\theta)) = T_{e_2 \cdots e_{k-1}}(l_{s,v_1}(\theta)) = \cdots = l_{s,v_k}(\theta) = l_{s,t}(\theta).$$

Let c now follow the strong FIFO rule and assume there is some $i \in \{1, \ldots, k\}$ with $T_{e_i}(l_{s,v_{i-1}}(\theta)) > l_{s,v_i}(\theta)$. Let P' be a shortest s-v_i-path at time θ. If we extend P' with $e_{i+1} \cdots e_k$ we get an s-t-path P'' which by the strong monotonicity of all T_e fulfills

$$l_{s,t}(\theta) \leq T_{P''}(\theta) = T_{e_{i+1} \cdots e_k}(l_{s,v_i}(\theta)) < T_{e_{i+1} \cdots e_k}(T_{e_1 \cdots e_i}(\theta)) = T_P(\theta). \qquad \square$$

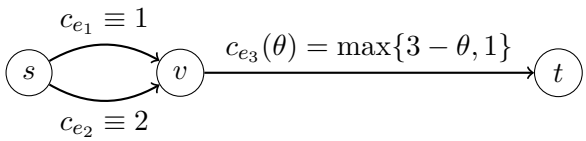

Figure 1: Both e_1 and e_2 are active at time 0 with $T_{e_2}(l_{s,s}(0)) > l_{s,v}(0)$.

For general FIFO costs however, not all subpaths of shortest paths are again shortest paths. That means, there might be edges $e = vw$ that do not fulfill the equality $l_{s,w}(\theta) = T_e(l_{s,v}(\theta))$ but still lie on a shortest s-t-path at time θ. This might be the case if a bottleneck edge e' closer to t has an interval on which $T_{e'}$ is constant. An example of this can be seen in Figure 1.

This leaves us with the question of how to find the rest of the active edges for general FIFO cost functions. For the rest of this section, let $c : \mathbb{R} \to \mathbb{R}^E_{\geq 0}$ be a continuous cost function following the FIFO rule with $\lim_{\theta \to -\infty} T_e(\theta) = -\infty$ for all $e \in E$. Now we want to exploit that once we determined $l_{s,t}(\theta)$, we can do another run of the Dynamic Dijkstra Algorithm on the reverse graph $G^{\leftarrow} = (V, E^{\leftarrow})$ with $E^{\leftarrow} \coloneqq \{e^{\leftarrow} = wv \mid e = vw \in E\}$ to compute the latest departure time starting from any node v to arrive before or at time $l_{s,t}(\theta)$ at t.

We define the corresponding cost function $\tilde{c}$ as

$$\tilde{c} : \mathbb{R} \to \mathbb{R}^{E^{\leftarrow}}_{\geq 0}, \quad \tilde{c}_{e^{\leftarrow}}(\theta) \coloneqq -T_e^{\leftarrow}(-\theta) - \theta.$$

Note, that $T_e \geq \mathbf{id}_{\mathbb{R}}$ and by Proposition 2.6 (v) we infer $T_e^{\leftarrow} \leq \mathbf{id}_{\mathbb{R}}$. Therefore, we have $T_e^{\leftarrow}(-\theta) \leq -\theta$ and thus $\tilde{c}_{e^{\leftarrow}}(\theta) \geq 0$.

We denote the traversal times induced by $\tilde{c}$ by $\tilde{T}_{e^{\leftarrow}}$ and $\tilde{T}_{P^{\leftarrow}}$,where $P^{\leftarrow} = e_k^{\leftarrow} \cdots e_1^{\leftarrow}$ for $P = e_1 \cdots e_k$, and the earliest arrival time due to $\tilde{c}$ as $\tilde{l}_{v,w}$. The traversal times fulfill $\tilde{T}_e = -T_e^{\leftarrow}(-\theta)$ which is by Proposition 2.6 (ii) strictly increasing. Hence, $\tilde{c}$ follows the strong FIFO rule. For a path $P^{\leftarrow} = e_k^{\leftarrow} \cdots e_1^{\leftarrow}$ the traversal time yields

$$\tilde{T}_{P^{\leftarrow}}(\theta) = \tilde{T}_{e^{\leftarrow}_{k-1} \cdots e^{\leftarrow}_1}(-T^{\leftarrow}_{e_k}(-\theta)) = \tilde{T}_{e^{\leftarrow}_{k-2} \cdots e^{\leftarrow}_1}(-T^{\leftarrow}_{e_{k-1}}(T^{\leftarrow}_{e_k}(-\theta)) = \cdots = -T^{\leftarrow}_P(-\theta)$$

and the earliest arrival times are of the form

$$\tilde{l}_{v,w}(\theta) = \min_{P \in \mathcal{P}_{w,v}} \tilde{T}_{P^{\leftarrow}}(\theta) = \min_{P \in \mathcal{P}_{w,v}} -T^{\leftarrow}_P(-\theta) = -\max_{P \in \mathcal{P}_{w,v}} T^{\leftarrow}_P(-\theta) = -l^{\leftarrow}_{w,v}(-\theta)$$

With this setup, we can do a second run of the Dynamic Dijkstra Algorithm to obtain the vector

$$(\tilde{l}_{t,w}(-l_{s,t}(\theta)))_w = (-l^{\leftarrow}_{t,w}(l_{s,t}(\theta)))_w.$$

Using Lemma 2.8 we can now check whether an outgoing edge $sw \in \delta_s^-$ of s is active at time θ by evaluating $T_e(\theta) \leq l^{\leftarrow}_{t,w}(l_{s,t}(\theta))$.

To implement this algorithm, we have to restrict ourselves to cost functions where the reversal of T_e can be evaluated easily. This is the case, for example, if c_e are piecewise linear functions. The resulting algorithm for this class of functions can be

Algorithm 3 Calculating Active Edges

```
def get_active_edges(
  costs: List[PiecewiseLinear], theta: float, source: Node, sink: Node,
  relevant_nodes: Set[Node], graph: DirectedGraph, strong_fifo: bool
) -> Set[Edge]:
  if len(source.outgoing_edges) <= 1:
    return source.outgoing_edges
  arrivals = dynamic_dijkstra(
    theta, source, sink, relevant_nodes, costs
  )
  if strong_fifo:
    return backtrack_shortest_paths(costs, arrivals, source, sink)
  else: # Second run of Dijkstra on the reverse graph.
    graph.reverse()
    traversals = [cost + identity for cost in costs]
    new_costs: List[Callable[[float], float]] = [
      lambda t: -trav.reversal(-t) - t for trav in traversals
    ]
    neg_departures = dynamic_dijkstra(
      -arrivals[sink], sink, source, relevant_nodes, new_costs
    )
    graph.reverse()
    return [
      e for e in source.outgoing_edges
      if traversals[e.id](theta) <= -neg_departures[e.node_to]
    ]
```

seen in Algorithm 3. Here, the flag `strong_fifo` is expected to be set only for cost functions following the strong FIFO rule. For these type of functions, we use the simpler backtracking search; for general FIFO functions, we use the approach described above. Other than that, if s has only up to one outgoing edge, we simply return δ_s^-.

2.5 Computing the Arrival Functions

Often, it is useful not only to compute active edges at some fixed point in time, but to have the earliest arrival functions $(l_{v,t})_{v \in V'}$ at some sink available as functions over time.

A very simple but also quite expensive method of computing these functions is a modification of the Bellman-Ford algorithm as described for example in [2]. It uses the representation found in Proposition 2.3. This means, we want to find the maximal solution of the system of equations

$$\tilde{l}_v(\theta) = \begin{cases} \theta, & \text{if } v = t, \\ \min_{e=vw \in \delta_v^-} \tilde{l}_w\left(T_e(\theta)\right), & \text{otherwise.} \end{cases}$$

Hence, the idea is to initialize all functions with $\tilde{l}_v(\theta) \coloneqq \infty$ for $v \neq t$ and $\tilde{l}_t(\theta) \coloneqq \theta$ and decrease the functions (pointwise) using the operation $\tilde{l}_v \coloneqq \min_{vw \in \delta_v^-} \tilde{l}_w \circ T_{vw}$ until no further changes are made, and the equations are fulfilled for all $v \in V \setminus \{t\}$.

More specifically, if some function $\tilde{l}_w$ changes, then all nodes v with an edge vw leading to w might need to be adjusted as well using the operation $\tilde{l}_v \coloneqq \min\{\tilde{l}_v, \tilde{l}_w \circ T_{vw}\}$. Therefore, we need some operations on the class of functions operated on to formulate the algorithm: To calculate the traversal times $T_e = c_e + \mathbf{id}_{\mathbb{R}}$ we need pointwise addition and a representation of the identity function; for updates of the functions the pointwise minimum and the composition of functions have to be implemented. To detect changes we also need to be able to identify whether one function is everywhere smaller or equal to some other function. Instead of representing the constant function with value ∞, we can simply only add a function once it gets its first update.

Algorithm 4 Dynamic Bellman-Ford Algorithm

```
def dynamic_bellman_ford(
  sink: Node, costs: List[PiecewiseLinear], relevant_nodes: Set[Node],
  theta: float
) -> Dict[Node, PiecewiseLinear]:
  arrivals: Dict[Node, PiecewiseLinear] = { sink: identity }
  traversals = [cost + identity for cost in costs]
  changed_nodes = {sink}
  while len(changed_nodes) > 0:
    change_detected = {}
    for w in changed_nodes:
      for e in w.incoming_edges:
        v = e.node_from
        if v not in relevant_nodes:
          continue
        relaxation = arrivals[w].compose(traversals[e.id])
        if v not in arrivals.keys():
          change_detected.add(v)
          arrivals[v] = relaxation
        elif not arrivals[v] <= relaxation:
          change_detected.add(v)
          arrivals[v] = arrivals[v].minimum(relaxation)
    changed_nodes = change_detected
  return arrivals
```

All the necessary operations explained above have been created for piecewise linear functions. The resulting procedure is shown in Algorithm 4. Here, `cost + identity` is the piecewise linear function representing the sum of `cost` and `identity`, the expression `arrivals[w].compose(traversals[e.id])` computes the piecewise linear function representing `arrivals[w]` $\circ$ `traversals[e.id]`. The decision whether the in-

equality `arrivals[v]`$(\theta) \leq$ `relaxation`(θ) holds for all $\theta \in \mathbb{R}$ is expressed by the term `arrivals[v] <= relaxation`. Finally, the pointwise minimum of the functions `arrivals[v]` and `relaxation` is computed by `arrivals[v].minimum(relaxation)`.

The benefit of calculating the arrival times as functions is that the active outgoing edges of all nodes v can be determined in one run of the Dynamic Bellman-Ford Algorithm by simply evaluating $l_{w,t}(T_e(\theta)) \leq l_{v,t}(\theta)$ for $vw \in E$. Additionally, the result can be used to calculate average travel times as explained in Section 5.

3 Equilibrium Flows

The goal of this project is to compute a specific kind of equilibrium flows. This section gives the appropriate introduction in the mathematical model of the underlying flows and introduces the so-called Dynamic Prediction Equilibria that we aim to compute.

3.1 Dynamic Flows

We begin by defining the model of dynamic flows, also called flows over time. Here, Vickrey's fluid queuing model is used. Each edge $e \in E$ has a positive *rate capacity* $\nu_e > 0$ and a positive *transit time* $\tau_e > 0$. The rate capacity bounds the amount of flow an edge can transfer at a time, which can be imagined as the width of a conveyor belt. The transit time on the other is the time the conveyor belt needs to transfer particles from its beginning to its end.

In this project, we consider multicommodity flows. That means, we have a finite set I of commodities and each commodity $i \in I$ has a *source node* $s_i \in V$ and a *sink node* $t_i \in V \setminus \{s_i\}$. We require, that s_i can reach t_i in (V, E). Furthermore, we define

$$V_i \coloneqq \{v \in V \mid v \text{ lies on a directed path from } s \text{ to } t \text{ in (V, E)}\}$$

as the subset of nodes that are *relevant* to commodity i.

Moreover, the *network inflow rate* of a commodity is given by a locally integrable function $u_i : \mathbb{R} \to \mathbb{R}_{\geq 0}$ with $u_i(\theta) = 0$ for $\theta \leq 0$.

Definition 3.1. A *(dynamic) flow* is a pair $f = (f^+, f^-)$ of families of locally integrable functions with $f^+_{i,e}, f^-_{i,e} : \mathbb{R} \to \mathbb{R}_{\geq 0}$ with $f^+_{i,e}(\theta) = f^-_{i,e}(\theta) = 0$ for $\theta \leq 0$ and $e \in E$ and $i \in I$. Here, $f^+_{i,e}(\theta)$ is called the *inflow rate* of commodity i into edge e at time θ whereas $f^-_{i,e}(\theta)$ describes the *outflow rate* of commodity i out of e at time θ. We denote the *total inflow and outflow rates* of an edge e as $f^+_e(\theta) \coloneqq \sum_{i \in I} f^+_{i,e}(\theta)$ and $f^-_e(\theta) \coloneqq \sum_{i \in I} f^-_{i,e}(\theta)$.

Given a dynamic flow, we can define the *accumulative inflow and outflow* as

$$F^+_e(\theta) \coloneqq \int_0^\theta f^+_e(t)\,\mathrm{d}t \quad \text{and} \quad F^-_e(\theta) \coloneqq \int_0^\theta f^-_e(t)\,\mathrm{d}t,$$

respectively. Based on that, the *queue length* of an edge e at time θ can be determined as $q_e(\theta) \coloneqq F^+_e(\theta) - F^-_e(\theta + \tau_e)$. The time-dependent cost of traversing an edge is thus defined as $c_e(\theta) \coloneqq \tau_e + q_e(\theta)/\nu_e$.

This definition already leads to a good insight on how particles should behave in a dynamic flow of this type: A particle of commodity i is generated at the source s_i at some time θ and immediately decides which outgoing edge $e = s_i v \in \delta^-_{s_i}$ it should take. At this time, the cost of e is $c_e(\theta) = \tau_e + q_e(\theta)/\nu_e$, which means that the particle has to queue for $q_e(\theta)/\nu_e$ time units before it can traverse the edge in τ_e time. It will thus arrive at v at time $T_e(\theta) \coloneqq \theta + c_e(\theta)$. Once it arrives at an intermediary node $v \neq t_i$, it will again have an opportunity to update its routing decision by choosing a new outgoing edge in δ^-_v until it arrives at its destination t_i.

To make the dynamic flows behave like imagined above, we have to make several restrictions forming the class of feasible dynamic flows:

Definition 3.2. A dynamic flow f is called *feasible up to time* $H \in \mathbb{R} \cup \{\infty\}$, if it fulfills the following conditions for almost all $\theta < H$:

(F1) No total outflow rate exceeds the capacity, i.e. $\forall e \in E : f_e^-(\theta) \leq \nu_e$.

(F2) Flow traverses an edge in a FIFO-manner, i.e.

$$\forall e \in E : \quad f_{i,e}^-(\theta) = \begin{cases} f_e^-(\theta) \cdot \frac{f_{i,e}^+(\xi)}{f_e^+(\xi)}, & \text{if } f_e^+(\xi) > 0, \\ 0, & \text{otherwise,} \end{cases}$$

where $\xi \coloneqq \max\{\xi \mid T_e(\xi) = \theta\}$ is the latest departure time a particle can arrive at w at time θ by traversing edge $e = vw$.

(F3) Flow conservation is preserved on intermediary nodes, i.e.

$$\forall i \in I, v \in V \setminus \{s_i, t_i\} : \quad b_{i,v}(\theta) \coloneqq \sum_{e \in \delta_v^-} f_{i,e}^+(\theta) - \sum_{e \in \delta_v^+} f_{i,e}^-(\theta) = 0$$

and $b_{i,s_i}(\theta) = u_i(\theta)$ as well as $b_{i,t_i}(\theta) \leq 0$ for all $\theta \in \mathbb{R}$.

(F4) Edges operate at capacity, i.e.

$$\forall e \in E : \quad f_e^-(\theta) = \begin{cases} \nu_e, & \text{if } q_e(\theta - \tau_e) > 0, \\ f_e^+(\theta - \tau_e), & \text{otherwise.} \end{cases}$$

A feasible flow up to ∞ is also called a *feasible* flow.

Remark 3.3. We note that property (F2) is defined slightly different in literature such as [5] in which $\xi \coloneqq \min\{\xi \mid T_e(\xi) = \theta\}$ is defined as the earliest possible time to arrive at w when traversing e. However, the set $\{\theta \mid \min\{\xi \mid T_e(\xi) = \theta\} < \max\{\xi \mid T_e(\xi) = \theta\}\}$ has zero measure such that the effect on the definition vanishes.

We highlight that for a given feasible flow f the costs c_e follow the FIFO-rule and hence the results from Section 2 can be applied.

Proposition 3.4. *The cost functions c induced by a feasible flow f follow the FIFO-rule.*

Proof. This can be seen by applying property (F1) in the following inequality for $\theta_1 \leq \theta_2$:

$$\begin{aligned} T_e(\theta_2) - T_e(\theta_1) &= \theta_2 - \theta_1 + \frac{q_e(\theta_2) - q_e(\theta_1)}{\nu_e} = \theta_2 - \theta_1 + \frac{\int_{\theta_1}^{\theta_2} f_e^+(t)\,\mathrm{d}t - \int_{\theta_1+\tau_e}^{\theta_2+\tau_e} f_e^-(t)\,\mathrm{d}t}{\nu_e} \\ &\geq \theta_2 - \theta_1 + \frac{\int_{\theta_1}^{\theta_2} f_e^+(t)\,\mathrm{d}t - (\theta_2 - \theta_1) \cdot \nu_e}{\nu_e} \geq 0. \end{aligned}$$

□

3.2 Dynamic Prediction Equilibrium

To enable a game theoretic perspective on dynamic flows, the infinitesimally small particles are interpreted as agents belonging to a commodity $i \in I$. An agent gets generated at some time at the source node s_i. Each time an agent arrives at a node, it can update its route based on historical traffic data of the network. More specifically, each commodity has its own prediction method to predict future queue lengths on all edges of the network. Based on the predicted queue lengths an agent now determines a time-dependent shortest path starting from his current position to t_i.

More formally, for a commodity $i \in I$ the *predicted queue length* of an edge e at time θ as predicted at time $\bar{\theta}$ based on the historical queue lengths $q = (q_e)_{e \in E}$ is given by $\hat{q}_{i,e}(\theta, \bar{\theta}, q)$. Aside from the predicted queue lengths all other predicted values will be denoted with a hat. This means, each commodity i has a set of predictor functions $(\hat{q}_{i,e})_{e \in E}$ each of which has the following the signature:

$$\hat{q}_{i,e} : \mathbb{R} \times \mathbb{R} \times \mathcal{Q}^E \to \mathbb{R}_{\geq 0},$$

where $\mathcal{Q} \coloneqq C(\mathbb{R}, \mathbb{R}_{\geq 0})$ denotes the set of all continuous functions from $\mathbb{R}$ to $\mathbb{R}_{\geq 0}$.

Based on the predicted queue lengths we can also determine *predicted edge costs* by

$$\hat{c}_{i,e}(\theta, \bar{\theta}, q) \coloneqq \tau_e + \frac{\hat{q}_{i,e}(\theta, \bar{\theta}, q)}{\nu_e}.$$

The *predicted traversal time of an edge* $e = vw$ is the predicted arrival time at w when entering edge e at time θ defined as $\hat{T}_{i,e}(\theta, \bar{\theta}, q) \coloneqq \theta + \hat{c}_{i,e}(\theta, \bar{\theta}, q)$. The *predicted traversal time of a path* $P = e_1\, e_2 \cdots e_k$ is thus defined as

$$\hat{T}_{i,P}(\theta, \bar{\theta}, q) \coloneqq \left(\hat{T}_{i,e_k}(\,\cdot\,, \bar{\theta}, q) \circ \cdots \circ \hat{T}_{i,e_1}(\,\cdot\,, \bar{\theta}, q)\right)(\theta).$$

For all nodes v that can reach t_i, we denote the set of all simple v-t_i-paths as $\mathcal{P}_{v,t_i}$. We can then determine the *predicted earliest arrival time* at t_i when starting in v at time θ by

$$\hat{l}_{i,v}(\theta, \bar{\theta}, q) \coloneqq \min_{P \in \mathcal{P}_{v,t_i}} \hat{T}_{i,P}(\theta, \bar{\theta}, q).$$

We call a path that attains this minimum a *$\bar{\theta}$-predicted shortest v-t_i-path at time θ*. Before we continue, we want to make sure, that this definition of shortest-paths is sensible: We can only restrict ourselves to simple paths, if waiting at a node, or traveling through a short cycle, is never helpful. This can be realized by assuming the FIFO-property of predictors:

Definition 3.5. A set of predictors $(\hat{q}_{i,e})_{e \in E}$ is *FIFO-compatible* if $(\hat{c}_{i,e}(\,\cdot\,, \bar{\theta}, q))_{e \in E}$ follows the FIFO-rule for all $\bar{\theta} \in \mathbb{R}_{\geq 0}, q \in \mathcal{Q}^E$.

We recall, that for FIFO-compatible predictors the results from Section 2 apply, and we have $\hat{l}_{i,v}(\theta, \bar{\theta}, q) \leq \hat{l}_{i,w}(\hat{T}_{i,e}(\theta, \bar{\theta}, q))$ for all $v \in V_i$ and $e = vw \in E$.

Definition 3.6. An edge $e = vw$ is called *active* for commodity $i \in I$ at time $\theta \in \mathbb{R}$ as predicted at time $\bar{\theta}$, if the equation

$$\hat{l}_{i,v}(\theta, \bar{\theta}, q) = \hat{l}_{i,w}\Big(\hat{T}_{i,e}(\theta, \bar{\theta}, q), \bar{\theta}, q\Big)$$

holds true. All edges for commodity i that are active at time θ as predicted at $\bar{\theta}$ are collected in the set $\hat{E}_i(\theta, \bar{\theta}, q)$.

We can now define an equilibrium based on these predictors, in which all agents only take shortest paths at all times based on their current prediction.

Definition 3.7. A pair $(\hat{q}, f)$ of a set of predictors $(\hat{q}_{i,e})_{i\in I, e\in E}$ and a dynamic flow f is a *partial dynamic prediction equilibrium up to time* $H \in \mathbb{R} \cup \{\infty\}$ if f is feasible up to time H and for all $e \in E, i \in I$ and $\theta < H$ it holds that

$$f^+_{i,e}(\theta) > 0 \implies e \in \hat{E}_i(\theta, \theta, q).$$

For $H = \infty$ the pair $(\hat{q}, f)$ is called *dynamic prediction equilibrium (DPE)*. Moreover, we call f a *dynamic prediction flow with respect to the predictor* $\hat{q}$.

In other words, a pair of a set of predictors and a flow is a DPE, if whenever flow enters an edge at some time θ, then the prediction of that commodity evaluated at the same time θ says, that the edge lies on a dynamic shortest path to t_i.

DPEs have been introduced in [4]. For the proof of the existence of DPEs the following two properties are important.

Definition 3.8. A predictor $\hat{q}_e$ is called *oblivious*, if the following condition holds

$$\forall \theta, \bar{\theta}, q, q' \colon \quad q_{\leq \bar{\theta}} = q'_{\leq \bar{\theta}} \implies \hat{q}_{i,e}(\theta; \bar{\theta}; q) = \hat{q}_{i,e}(\theta; \bar{\theta}; q'),$$

where $q_{\leq \bar{\theta}}$ denotes the restriction of the function $q : \mathbb{R}_{\geq 0} \times E \to \mathbb{R}_{\geq 0}$ to $[0, \bar{\theta}] \times E$.

Definition 3.9. A predictor $\hat{q}_e$ is *continuous*, if the mapping

$$\hat{q}_e : \mathbb{R} \times \mathbb{R} \times \mathcal{Q}^E \to \mathbb{R}_{\geq 0}$$

with respect to the product topology on the left side and the topology induced by the uniform norm on $\mathcal{Q}$ is continuous.

Theorem 3.10 ([4, Theorem 10])**.** *For any network with a finite set of commodities, each associated with a locally integrable, bounded network inflow rate and oblivious, continuous, and FIFO-compatible predictors $\hat{q}_{i,e}$ there exists a dynamic prediction flow with respect to $\hat{q}$.*

3.3 Applied Predictors

In this section we introduce several predictors that have been used throughout the project. We begin with very primitive predictors and enhance them step by step.

The *Zero-Predictor* $\hat{q}^{\mathrm{Z}}$ predicts that no queues occur at all, i.e.

$$\hat{q}_e^{\mathrm{Z}}(\theta, \bar{\theta}, q) \coloneqq 0$$

for all $e \in E$, $\theta, \bar{\theta} \in \mathbb{R}$ and $q \in \mathcal{Q}^E$. This induces the constant edge costs $\hat{c}_e^{\mathrm{Z}}(\,\cdot\,, \cdot\,, \cdot\,) \equiv \tau_e$ for all edges $e \in E$.

Another approach is the so-called *constant predictor* $\hat{q}^{\mathrm{C}}$ which predicts, that the queue lengths simply stay constant beginning from the prediction time, i.e.

$$\hat{q}_e^{\mathrm{C}}(\theta, \bar{\theta}, q) \coloneqq q_e(\bar{\theta})$$

for all $e \in E$, $\theta, \bar{\theta} \in \mathbb{R}$ and $q \in \mathcal{Q}^E$. Here, the edge costs are constant in the argument θ with $\hat{c}_e^{\mathrm{C}}(\,\cdot\,, \bar{\theta}, q) \equiv \tau_e + q_e(\bar{\theta})/\nu_e$ for all edges $e \in E$. If all commodities use the constant predictor, the dynamic flow of a DPE is simply an instantaneous dynamic equilibrium (IDE) as introduced in [5, Definition 2.1].

A more sophisticated variant is the *linear predictor* $\hat{q}^{\mathrm{L}}$ which predicts that the queue length extends with the current derivative up to some prediction horizon $H \in \mathbb{R}_{>0} \cup \{\infty\}$. Starting from H we assume a constant course. In formulas, this can be written as

$$\hat{q}_e^{\mathrm{L}}(\theta, \bar{\theta}, q) \coloneqq \Big(q_e(\bar{\theta}) + \partial_- q_e(\bar{\theta}) \cdot \min\{\theta - \bar{\theta}, H\}\Big)^+,$$

where $(x)^+ \coloneqq \max\{x, 0\}$ denotes the positive part of $x \in \mathbb{R}$.

As the left derivative of q_e might not be continuous, the linear predictor does not qualify for the requirements of Theorem 3.10. The following predictor eliminates this problem by applying a regularization. The *regularized linear predictor* $\hat{q}^{\mathrm{RL}}$ uses a rolling average of the past gradient with rolling horizon $\delta > 0$, which yields

$$\hat{q}_{i,e}^{\mathrm{RL}}(\theta, \bar{\theta}, q) \coloneqq \Big(q_e(\bar{\theta}) + \frac{q_e(\bar{\theta}) - q_e(\bar{\theta} - \delta)}{\delta} \cdot \min\{\theta - \bar{\theta}, H\}\Big)^+.$$

Furthermore, we introduce a machine learning predictor $\hat{q}^{\mathrm{ML}}$ using a simple linear regression model. More specifically, for an edge $e = vw$ we use k_p samples of past queue lengths of surrounding edges $N(e) = \delta_v^+ \cup \{e\} \cup \delta_w^-$ to predict k_f samples of the future queue length of e. The samples are equidistant with step length $\delta > 0$. For each edge e, we learn coefficient matrices $W^{e'} \in \mathbb{R}^{k_p \times k_f}$ for $e' \in N(e)$ as well as biases $\beta \in \mathbb{R}^{k_f}$ to compute interpolation points

$$\hat{q}_e^{\mathrm{ML}}\Big(\bar{\theta} + j \cdot \delta, \bar{\theta}, q\Big) \coloneqq \left(\sum_{e' \in N(e)} \sum_{i=1}^{k_p} w_{i,j}^{e'} \cdot q_{e'}\Big(\bar{\theta} - (i-1) \cdot \delta\Big) + \beta_j\right)^+$$

for $j \in \{1, \dots, k_f\}$ and $i \in \{1, \dots, k_p\}$. The function $\hat{q}_e^{\mathrm{ML}}(\,\cdot\,, \bar{\theta}, q)$ linearly interpolates these points and is extended constantly with the value $\hat{q}_e^{\mathrm{ML}}(\bar{\theta} + k_f \cdot \delta, \bar{\theta}, q)$ starting from

$\bar{\theta} + k_f \cdot \delta$. As the above predictor is not necessarily FIFO-compatible per se, we manually limit the gradient of $\hat{q}_e^{\mathrm{ML}}(\,\cdot\,, \bar{\theta}, q)$ such that it never subceeds $-\nu_e$. More details on this method are described in Section 4.6.

Finally, we introduce the *perfect predictor* $\hat{q}^{\mathrm{P}}$ which predicts the queues as they will occur with

$$\hat{q}_e^{\mathrm{P}}(\theta, \bar{\theta}, q) \coloneqq q_e(\theta).$$

As this is a non-oblivious predictor, it is not possible to embed it into a simulation nor does it fulfill the constraints of Theorem 3.10. Nevertheless, the flow of a DPE equilibrium, in which all commodities use the perfect predictor, is the same as a dynamic equilibrium as analyzed in [1, Definition 2]. In this scenario particles can exactly predict the traffic conditions at any future time and choose a shortest path already when starting at the source node.

4 Computation of Dynamic Prediction Equilibria

In this section we want to discuss how we can compute an approximated variant of DPEs. We recall from the definition, that in a DPE all infinitesimally small agents update their routing decision every time they arrive at an intermediate node. As we are dealing with continuous flows, these routing decisions take place in a continuous manner. To approximate this, we choose to allow updates of the routing decisions to happen at predefined intervals. This leads to the following definition:

Definition 4.1. Let $\varepsilon > 0$. A pair $(\hat{q}, f)$ of a set of predictors $(\hat{q}_{i,e})_{i\in I, e\in E}$ and a dynamic flow f is a *partial ε-approximated dynamic prediction equilibrium (ε-DPE) up to time* $H \in \mathbb{R} \cup \{\infty\}$ if f is feasible up to time H and for all $e \in E, i \in I$ and $\theta < H$ it holds that

$$f^+_{i,e}(\theta) > 0 \implies e \in \hat{E}_i(\varepsilon \cdot \lfloor \theta/\varepsilon \rfloor, \varepsilon \cdot \lfloor \theta/\varepsilon \rfloor, q).$$

To compute these kinds of dynamic flows, we begin by setting up the data structures representing dynamic flows before we have a look at the extension of feasible flows and finally how to use the predictors and the computation of dynamic shortest paths to extend the flows following the prescript in the definition of ε-DPEs.

Moreover, we focus on dynamic flows with right-constant flow rates only. This means, we assume that all network inflow rates u_i are right-constant.

4.1 Extension Theorem

Before diving into an actual program, we first consider a few theoretical observations involving the feasibility of dynamic flows. More specifically, we want to extend partial flows with constant edge inflow rates while always preserving the feasibility properties (F1), (F2) and (F4). By choosing appropriate edge inflow rates, we will later make sure that the flow conservation property (F3) is fulfilled as well. The following theorem is the building block for the algorithms introduced in the next sections.

Theorem 4.2. *Let f be a dynamic flow fulfilling properties (F1), (F2) and (F4) on an edge $e \in E$ for almost all $\theta \in \mathbb{R}$. We are given new constant inflow rates $(g_{i,e})_{i\in I} \in \mathbb{R}^I_{\geq 0}$ into e beginning from time ϕ and let $g_e \coloneqq \sum_{i\in I} g_{i,e}$. We partially reassign $f^+_{i,e}$ using $f^+_{i,e}|_{[\phi,\infty)} \coloneqq\equiv g_{i,e}$ for all $i \in I$. Moreover, we partially update $f^-_{i,e}$ in the following manner:*

Case I. *If $g_e = 0$, we set $f^-_{i,e}|_{[T_e(\phi),\infty)} \coloneqq\equiv 0$.*

Case II. *If $g_e > 0 \wedge (q_e(\phi) = 0 \vee g_e \geq \nu_e)$, we assign $f^-_{i,e}|_{[T_e(\phi),\infty)} \coloneqq\equiv \min\{\nu_e, g_e\} \cdot \frac{g_{i,e}}{g_e}$.*

Case III. *If $g_e \in (0, \nu_e) \wedge q_e(\phi) > 0$, we use $T_{\text{depl}} \coloneqq \phi + \frac{q_e(\phi)}{\nu_e - g_e}$ for assigning*

$$f^-_{i,e}|_{[T_e(\phi),T_{\text{depl}}+\tau_e)} \coloneqq\equiv \nu_e \cdot \frac{g_{i,e}}{g_e} \quad \text{and} \quad f^-_{i,e}|_{[T_{\text{depl}}+\tau_e,\infty)} \coloneqq\equiv g_{i,e}.$$

Then the updated dynamic flow f still fulfills the properties (F1), (F2) and (F4) on edge e for almost all $\theta \in \mathbb{R}$.

Proof. Let f be the original and h be the transformed dynamic flow and let q^f and q^h denote their corresponding queue length functions. We note, that q^f and q^h as well as T^f and T^h coincide on $(-\infty, \phi]$. Moreover, the outflow rate h_e^- never exceeds ν_e and hence (F1) is fulfilled right away. Next, we show (F4), i.e.

$$h_e^-(\theta) = \begin{cases} \nu_e, & \text{if } q_e^h(\theta - \tau_e) > 0, \\ h_e^+(\theta - \tau_e), & \text{otherwise.} \end{cases}$$

For $\theta < \phi + \tau_e$, this follows solely from the properties of f. Let us analyze the case $\phi + \tau_e \leq \theta < T_e(\phi)$ or equivalently $\phi \leq \theta - \tau_e < \phi + q_e(\phi)/\nu_e$. With (F1) we conclude

$$\begin{aligned} q_e^h(\theta - \tau_e) &= \int_0^{\theta - \tau_e} h_e^+(\psi)\,\mathrm{d}\psi - \int_0^{\theta} h_e^-(\psi)\,\mathrm{d}\psi \\ &\geq \int_0^{\phi} h_e^+(\psi)\,\mathrm{d}\psi - \int_0^{\phi + \tau_e} h_e^-(\psi)\,\mathrm{d}\psi - \nu_e \cdot (\theta - \tau_e - \phi) \\ &= q_e(\phi) - \nu_e \cdot (\theta - \tau_e - \phi) > 0. \end{aligned}$$

With the same reasoning we infer $q_e^f(\theta - \tau_e) > 0$ and therefore $h_e^-(\theta) = f_e^-(\theta) = \nu_e$. Let us finally discuss the case $\theta > T_e(\phi)$; the point $\theta = T_e(\phi)$ can be ignored as we are only interested in almost all times. We distinguish between the three cases:

Case I. Here, $g_e = 0$ implies

$$q_e^h(\theta - \tau_e) = \int_0^{\phi} h_e^+(\psi)\,\mathrm{d}\psi - \int_0^{T_e(\phi)} h_e^-(\psi)\,\mathrm{d}\psi = q_e(\phi) - \int_{\phi}^{\phi + q_e(\phi)/\nu_e} h_e^-(\psi + \tau_e)\,\mathrm{d}\psi$$

And as property (F4) holds for $t \leq T_e(\theta)$, we infer $q_e^h(\theta - \tau_e) = q_e(\phi) - \nu_e \cdot \frac{q_e(\phi)}{\nu_e} = 0$. Together with $h_e^-(\theta) = 0 = h_e^+(\theta - \tau_e)$ this yields the claim.

Case II. We have $g_e > 0 \wedge (q_e(\phi) = 0 \vee g_e \geq \nu_e)$. Then

$$\begin{aligned} q_e^h(\theta - \tau_e) &= q_e(\phi) + g_e \cdot (\theta - \tau_e - \phi) - \int_{\phi + \tau_e}^{\theta} h_e^-(\psi)\,\mathrm{d}\psi \\ &= q_e(\phi) + g_e \cdot (\theta - \tau_e - \phi) - q_e(\phi) - (\theta - T_e(\phi)) \cdot \min\{\nu_e, g_e\} \\ &= g_e \cdot (\theta - \tau_e - \phi) - (\theta - T_e(\phi)) \cdot \min\{\nu_e, g_e\}. \end{aligned}$$

Let us first assume $q_e(\phi) = 0$. Then the above reduces to $(g_e - \min\{\nu_e, g_e\}) \cdot (\theta - T_e(\phi))$ which is positive if and only if $g_e > \nu_e$ holds. In that case $h_e^-(\theta) = \nu_e$ holds by our assignment. If $q_e^h(\theta - \tau_e)$ is non-positive, we have $g_e \leq \nu_e$ which implies $h_e^-(\theta) = g_e = h_e^+(\theta - \tau_e)$.

For the case $q_e(\phi) > 0$ and $g_e \geq \nu_e$, we have

$$q_e^h(\theta - \tau_e) = g_e \cdot (\theta - \tau_e - \phi) - (\theta - T_e(\phi)) \cdot \nu_e = (g_e - \nu_e) \cdot (\theta - \tau_e - \phi) + q_e(\phi) > 0$$

and then the claim follows from $h_e^-(\theta) = \nu_e$.

Case III. We do a case distinction on $\theta - \tau_e < T_{\text{depl}}$. If this is true, then

$$\begin{aligned} q_e^h(\theta - \tau_e) &= q_e(\phi) + g_e \cdot (\theta - \tau_e - \phi) - \nu_e \cdot (\theta - \phi - \tau_e) \\ &= q_e(\phi) - (\nu_e - g_e) \cdot (\theta - \tau_e - \phi) > q_e(\phi) - (\nu_e - g_e) \cdot (T_{\text{depl}} - \phi) = 0. \end{aligned}$$

Moreover, we have $h^-(\theta) = \nu_e$ in this case. Finally, for $\theta - \tau_e \geq T_{\text{depl}}$ we have

$$q_e^h(\theta - \tau_e) = q_e^h(T_{\text{depl}}) + g_e \cdot (\theta - \tau_e - T_{\text{depl}}) - g_e \cdot (\theta - T_{\text{depl}} - \tau_e) = q_e^h(T_{\text{depl}})$$

and by the arguments above we deduce $q_e^h(T_{\text{depl}}) = 0$. As $h_e^-(\theta) = g_e = h_e^+(\theta - \tau_e)$ holds, this completes the proof of the claim.

It remains to show property (F2), i.e. for almost all θ and all $i \in I$ the equation

$$h_{i,e}^-(\theta) = \begin{cases} h_e^-(\theta) \cdot \frac{h_{i,e}^+(\xi)}{h_e^+(\xi)}, & \text{if } h_e^+(\xi) > 0, \\ 0, & \text{otherwise,} \end{cases}$$

holds with $\xi \coloneqq \max\{\xi \mid \xi + c_e^h(\xi) = \theta\}$. For $\theta < T_e(\phi)$, this property is already induced by the original flow f. For $\theta > T_e(\phi)$ we have $\xi > \phi$ because of the monotonicity of T_e^h and $T_e^h(\xi) = \theta > T_e(\phi)$. Therefore, by definition of our assignments the relations $h_{i,e}^+(\xi)/h_e^+(\xi)$ and $h_{i,e}^-(\theta)/h_e^-(\theta)$ coincide with $g_{i,e}/g_e$ in the case $g_e > 0$. For $g_e = 0$, the inflow rates $h_{i,e}^+(\theta)$ and $h_e^+(\xi)$ are zero as well. □

We note that in the scenario of Theorem 4.2, the queue q_e of the transformed flow is given for $\theta \geq \phi$ as follows:

Cases I and III. If we denote the depletion time by $T_{\text{depl}} \coloneqq \phi + q_e(\phi)/(\nu_e - g_e)$, the queue lengths can be determined using

$$q_e(\theta) = \begin{cases} q_e(\phi) - (\theta - \phi) \cdot (\nu_e - g_e), & \text{for } \theta \leq T_{\text{depl}}, \\ 0, & \text{for } \theta \geq T_{\text{depl}}. \end{cases}$$

Case II. Here, no queue depletion occurs and for all $\theta \geq \phi$ we have

$$q_e(\theta) = q_e(\phi) + (\theta - \phi) \cdot \max\{g_e - \nu_e, 0\}.$$

The difference between Case I and Case III is that in Case I, $q_e(\phi) = 0$ is allowed, and thus it is possible that T_{depl} coincides with ϕ.

4.2 Encoding Right Constant and Piecewise Linear Functions

Let us now discuss how the piecewise constant and piecewise linear functions are encoded. Let $f : \mathbb{R} \to \mathbb{R}$ be a piecewise linear function given by

$$f(x) \coloneqq \begin{cases} f(\xi_1) + (x - \xi_1) \cdot s_f & \text{for } x < \xi_1, \\ f(\xi_i) + (x - \xi_i) \cdot \frac{f(\xi_{i+1}) - f(\xi_i)}{\xi_{i+1} - \xi_i} & \text{for } \xi_i \leq x < \xi_{i+1}, i \in [k-1], \\ f(\xi_k) + (x - \xi_k) \cdot s_l & \text{for } x \geq \xi_k, \end{cases}$$

where $\xi_1 < \xi_2 < \cdots < \xi_k$ are the supporting points with $k \geq 1$, and s_f and s_l are the first and last slopes, respectively. Data Structure 1 encodes this function by saving the list $[\xi_0, \xi_1, \ldots, \xi_k]$ in the attribute `times`, $[f(\xi_0), f(\xi_1), \ldots, f(\xi_k)]$ in the attribute `values` and s_f and s_l in the attributes `first_slope` and `last_slope`. Similarly, Data Structure 2 encodes a right-constant function $f : \mathbb{R} \to \mathbb{R}$ given by

$$f(x) \coloneqq \begin{cases} v_f, & \text{if } x < \xi_1 \text{ (or } k = 0), \\ f(\xi_i), & \text{if } \xi_i \leq x < \xi_{i+1}, i \in [k-1], \\ f(\xi_k), & \text{if } x \geq \xi_k, \end{cases}$$

where v_f is written to the attribute `first_value`. Here, $k = 0$ is allowed. In both cases, an evaluation of the function can be done using a simple binary search in the `times` array resulting in a logarithmic complexity in the time dimension k.

Data Structure 1
Piecewise Linear Functions

```python
class PiecewiseLinear:
  times: List[float]
  values: List[float]
  first_slope: float
  last_slope: float

  #  ... (methods)
```

Data Structure 2
Right Constant Functions

```python
class RightConstant:
  times: List[float]
  values: List[float]
  first_value: float

  #  ... (methods)

```

In the next section, we will need the following two operations on these functions. First, we want to be able to extend a piecewise linear function f beginning at some time $\theta \geq \max_{i \in [k]} \xi_i$ by some slope s. This translates to Algorithm 5. We only add a new entry to the lists `times` and `values` if $\theta > \max_{i \in [k]}$.

Algorithm 5 Extension Procedure in `class PiecewiseLinear`

```python
def extend_with_slope(self, time: float, slope: float):
  if time > self.times[-1]:
    val = self.values[-1] + (time - self.times[-1]) * self.last_slope
    self.values.append(val)
    self.times.append(time)
  self.last_slope = slope
```

Similarly, right constant functions should be extendable beginning from some time $\theta \in \mathbb{R}$ with $\theta \geq \sup_{i \in [k]} \xi_i$ by some value $v \in \mathbb{R}$. The corresponding procedure is shown in Algorithm 6. Both of these extension procedures have a constant running time.

Algorithm 6 Extension Procedure in `class RightConstant`

```
def extend(self, time: float, value: float):
  if len(times) > 0 and time == self.times[-1]:
    self.values[-1] = value
  else:
    self.values.append(value)
    self.times.append(time)
```

4.3 Encoding Feasible Flows

We now discuss the data structure used for representing and extending partial dynamic flows as defined in Data Structure 3.

Data Structure 3 Partial Dynamic Flows

```
class MultiComFlow:
  phi: float
  inflow: Dict[Edge,List[RightConstant]]
  outflow: Dict[Edge, List[RightConstant]]
  queues: Dict[Edge, PiecewiseLinear]
  outflow_changes: PriorityQueue[Tuple[Edge, float]]
  depletions: Dict[Edge, Tuple[float, float, List[float]]]

  #  ... (methods)
```

The horizon of such a flow, i.e. the time up to which the flow has been computed, is written to the attribute `phi`. The functions $f^+_{i,e}$ and $f^-_{i,e}$ correspond to the attributes `inflow[e][i]` and `outflow[e][i]`, respectively, both of which are objects of the class `RightConstant`.

Although, we could compute the queue length of an edge e from these two functions on demand, we allocate another attribute `queues[e]` for the queue lengths to speed up the extension procedure. This can be done using an object representing a piecewise linear function because it is the integral of piecewise constant functions.

Next, the priority queue `outflow_changes` holds all upcoming events later than `phi` where some edge outflow will definitely change. It saves tuples (e, θ) with $\theta >$ `phi` interpreted as "`outflow[e]` changes at time θ". This helps us later as we only want to extend the flow at most as long as no edge outflow rate changes in a single extension phase.

Finally, we have a dictionary `depletions` which contains an edge e if there is an upcoming queue depletion at some time $T_{\text{depl}} > \phi$. In that case, `depletions[e]` is a tuple $(T_{\text{depl}}, T_{\text{depl}} + \tau_e, (g_{i,e})_{i \in I})$ where $(g_{i,e})_{i \in I}$ is the new outflow $(f^-_{i,e})_{i \in I}$ starting from time $T_{\text{depl}} + \tau_e$.

4.4 Extension of Feasible Flows

In this section we discuss how to extend a feasible flow for some time period given new constant inflow rates on a subset of the edges.

In the beginning we usually start with an empty flow representing a feasible flow up to time 0. This is an object of the class `MultiComFlow` initialized using the operations

- `phi = 0`,
- `inflow[e][i] = RightConstant(times=[], values=[], first_value=0)` for all edges `e` and commodities `i`,
- `outflow[e][i] = RightConstant(times=[], values=[], first_value=0)` for all edges `e` and commodities `i`,
- `queues[e] = PiecewiseLinear(times=[0], values=[0], first_slope=0, last_slope=0)` for all edges `e`,

with an empty priority queue `outflow_changes` and an empty dictionary `depletions`.

We now want to extend this initialized flow in so-called *extension phases*. That means, given a feasible flow f up to time ϕ, we want to extend this flow to a feasible flow up to time $\phi + \alpha$ for some small $\alpha > 0$. During this extension interval $[\phi, \phi + \alpha)$ we keep all edge inflow and outflow rates constant.

We imagine a scenario in which some external mechanism determines edge inflow rates $g \in \mathbb{R}_{\geq 0}^{I \times E}$ to extend f by $f^+_{i,e}|_{[\phi,\phi+\alpha)} \equiv g_{i,e}$ for some small $\alpha > 0$. Moreover, we want to choose α at least so small, that no edge outflow rate changes within $[\phi, \phi+\alpha)$. That way $b_{i,v}$ remains constant during this interval and once the node inflow $\sum_{e \in \delta^+_v} f^-_{i,e}$ changes, the external mechanism can react to these changes. This enables the mechanism to preserve the flow conservation property (F3) on intermediary nodes. This external mechanism will be introduced in the next section.

After determining the new inflow rates $(g_{i,e})_{i \in I}$ of an edge e beginning at time ϕ, we assign $f^+_{i,e}|_{[\phi,\infty)} \coloneqq g_{i,e}$. Then we determine the outflow rate of e beginning at time $T_e(\phi) = \phi + \tau_e + q_e(\phi)/\nu_e$ according to properties (F2) and (F4) of Definition 3.2:

In the extension method we are about to introduce we want to keep track of all future changes of outflow rates of any edge. This is done by queuing events of this type in the priority queue `outflow_changes`. In all three cases **I**–**III** as defined in Theorem 4.2, the outflow of e changes at time $T_e(\phi)$. Only in case **III**, the outflow might change once again at time $T_{\text{depl}} + \tau_e$, if during the interval (ϕ, T_{depl}) no other changes to the inflow of e are made. Hence, in case **III** we record a planned change event in the dictionary `depletions` for edge e together with the current depletion time as well as the planned change time $T_{\text{depl}} + \tau_e$.

The main procedure is shown in Algorithm 7. We handle the main three cases **I**, **II** and **III** in different functions, that are called in the lines 4-11. These functions extend the queue function `queues[e]` and the inflow functions `inflow[e][i]` starting from time `phi` as well as the outflow functions `outflow[e]` starting from time $T_{\texttt{e}}(\texttt{phi})$ for all commodities i. Furthermore, they add an event to the queue `outflow_changes` to remember that the outflow of edge `e` changes at $T_{\texttt{e}}(\texttt{phi})$. Additionally, they update the

Algorithm 7 Extension Procedure in `class MultiComFlow`

```
1  def extend(
2    self, new_inflow: Dict[Edge, List[float]], max_ext_time: float
3  ) -> Set[Edge]:
4    for e in new_inflow.keys():
5      acc_in = sum(new_inflow)
6      if acc_in == 0:
7        self._extend_case_i(e)
8      elif self.queues[e](self.phi) == 0 or acc_in > capacity[e]:
9        self._extend_case_ii(e, new_inflow[e])
10     else:
11       self._extend_case_iii(e, new_inflow[e])
12
13   self.phi = min(
14     self.outflow_changes.min_key(),
15     min(
16       change_time for (_,change_time,_) in self.depletions.values(),
17       default=float('inf')
18     ),
19     max_ext_time
20   )
21
22   self._process_depletions()
23
24   changed_edges: Set[Edge] = set()
25   while self.outflow_changes.min_key() <= self.phi:
26       changed_edges.add(self.outflow_changes.pop()[0])
27   return changed_edges
```

entry of `e` in the dictionary `depletions` to reflect whether an upcoming queue depletion will occur with the new edge inflow rate.

Once this is done, we can determine the maximum extension span α and thus the new flow horizon `phi` as described in lines 13-20. As we only want to extend at most as long as no edge outflow changes, we take the minimum of such change events. As queue depletions induced by case **III** may also cause outflow rate changes, we also need to consider entries of the dictionary `depletions` here. Finally, we also give the external mechanism the possibility to limit this extension time using the parameter `max_ext_time`.

After determining the extension length via the new flow horizon `phi`, we need to process all queue depletions that happen before or at time `phi`. This procedure is depicted in Algorithm 8. For such a queue depletion of an edge `e`, we need to update the queue function `queues[e]` which will attain 0 at its depletion time and should be

Algorithm 8 Procedure for Processing Queue Depletions in `class MultiComFlow`

```
1  def _process_depletions():
2    while min(
3        depl_time for (depl_time,_,_) in self.depletions.values(),
4        default=float('inf')
5      ) <= self.phi:
6      e = min(self.depletions, key=lambda e: self.depletions[e][0])
7      (depl_time, change_time, new_outflow) = self.depletions.pop(e)
8      self.queues[e].extend_with_slope(depl_time, 0)
9      if change_time < float('inf'):
10       self.outflow_changes.add((e, change_time), change_time)
11       for i in range(n):
12         self.outflow[e][i].extend(change_time, new_outflow[i])
```

extended by a 0 slope. Furthermore, if the queue depletion induces a change of the edge outflow rate, an appropriate event is added to the priority queue `outflow_changes` and the outflow functions `outflow[e][i]` are updated for all commodities `i`.

After processing all queue depletions, all edge outflow changes that will occur up to the new time horizon `phi` have an appropriate entry in `outflow_changes`. Thus, in the main procedure in Algorithm 7 we can collect all edges whose outflow has changed in the returned set `changed_edges`. This is described in lines 24-27.

For completeness, we also describe the extension procedures for the three cases **I**–**III** in more detail. Their corresponding implementations are displayed in Algorithm 9. In all three cases, we first determine the arrival time $T_{\mathtt{e}}(\mathtt{phi})$ when entering the edge at the current time `phi` and write the result into the variable `arrival`. Moreover, the pair `(e, arrival)` is pushed into the priority queue `outflow_changes`.

For case **I**, we extend the inflow rates starting at `phi` and the outflow rates starting at $T_{\mathtt{e}}(\mathtt{phi})$ with zeros. If the edge has a positive queue at time `phi`, the queue is expected to deplete at time `depl_time`. If this event occurs, no further changes to the outflow rates are necessary, hence we can set the value of `e` in the dictionary `depletions` to `(depl_time, float('inf'), None)`. On the other hand, if the queue is empty at time `phi`, we can simply remove `e` from the dictionary, as no depletion will occur.

The case **II** is straight-forward: The new outflow rate starting at $T_{\mathtt{e}}(\mathtt{phi})$ is calculated according to Theorem 4.2 and the assumptions imply that no queue depletion will occur for the given inflow rates. Hence, we do not need to update the entry of `e` in `depletions`. Instead, if an entry of `e` exists, we simply remove it.

Case **III** can only occur, if a queue depletion is expected. Again, we calculate the outflow starting at $T_{\mathtt{e}}(\mathtt{phi})$ according to Theorem 4.2 and update the entry in `depletions` for `e`: Here, the outflow is expected to change at time $T_{\text{depl}} + \tau_e$ once again where the new outflow rates starting from that time match the new inflow rates.

Algorithm 9 Extension Procedure for Cases **I**–**III** in `class MultiComFlow`

```
def _extend_case_i(self, e: Edge):
  cur_queue = self.queues[e](self.phi)
  arrival = self.phi + cur_queue / capacity[e] + travel_time[e]
  self.outflow_changes.push((e, arrival), arrival)
  for i in range(n):
    self.inflow[e][i].extend(self.phi, 0)
    self.outflow[e][i].extend(arrival, 0)

  queue_slope = 0 if cur_queue == 0 else -capacity[e]
  self.queues[e].extend_with_slope(self.phi, queue_slope)
  if cur_queue > 0:
    depl_time = self.phi + cur_queue / capacity[e]
    self.depletions[e] = depl_time, float('inf'), None
  elif e in self.depletions:
    self.depletions.pop(e)


def _extend_case_ii(self, e: Edge, new_inflow: List[float]):
  # cur_queue = [...]; arrival = [...]; self.outflow_changes.push([...])
  for i in range(n):
    self.inflow[e][i].extend(self.phi, new_inflow[i])
    new_out = min(capacity[e], acc_in) * new_inflow[i] / acc_in
    self.outflow[e][i].extend(arrival, new_out)

  queue_slope = max(acc_in - capacity[e], 0)
  self.queues[e].extend_with_slope(self.phi, queue_slope)
  if e in self.depletions:
    self.depletions.pop(e)


def _extend_case_iii(self, e: Edge, new_inflow: List[float]):
  # cur_queue = [...]; arrival = [...]; self.outflow_changes.push([...])
  for i in range(n):
    self.inflow[e][i].extend(self.phi, new_inflow[i])
    new_out = capacity[e] * new_inflow[i] / acc_in
    self.outflow[e][i].extend(arrival, new_out)

  queue_slope = acc_in - capacity[e]
  self.queues[e].extend_with_slope(self.phi, queue_slope)
  depl_time = self.phi + cur_queue / (capacity[e] - acc_in)
  planned_change = depl_time + travel_time[e]
  self.depletions[e] = depl_time, planned_change, new_inflow
```

4.5 Computation of Approximated DPEs

We now use the subroutine introduced in the previous section to compute ε-DPEs. Here, we define the mechanism which decides how the flow is routed from its source to its sink and which defines the constant inflow rates of the extension phases.

As we want each commodity to take shortest paths according to their own traffic forecast, the concept of a predictor is realized as an abstract class as shown in Data Structure 4. A predictor has an implementation for the method `predict`. Attributes of that method are a prediction time `phi` and a list `old_queues` of past queue length functions of all edges. This method should return the queue length functions as predicted at time `phi` using the past queues `old_queues`. In mathematical terms, it should return $(\hat{q}_{i,e}(\,\cdot\,, \bar{\theta}, q))_{e \in E}$ with `phi` $= \bar{\theta}$ and `old_queues` $= q$.

To speed up the calculation of shortest paths, we add a function `is_constant` which should return `True` if and only if the `predict` function of that predictor always returns constant functions. In that case, we can restrict ourselves to algorithms for static edge costs.

Data Structure 4 The abstract `class Predictor`

```
class Predictor(ABC):
  @abstractmethod
  def predict(self, phi: float, old_queues: List[PiecewiseLinear]) \
    -> List[PiecewiseLinear]:
    pass

  @abstractmethod
  def is_constant(self) -> bool:
    pass
```

In Section 4.6 we will discuss the implementations of the different predictors that have been used throughout the experiments. Here, we will assume that these implementations are already available. Each commodity has a predictor assigned to it which is used for its routing decisions. A commodity's network inflow rate is defined as an arbitrary piecewise constant function saved in the attribute `net_inflow`. The class definition of a commodity can be seen in Data Structure 5.

Data Structure 5 Commodities

```
class Commodity:
  source: Node
  sink: Node
  net_inflow: RightConstant
  predictor: Predictor
```

In the remainder of this section, we introduce another class which encapsulates all the state and procedures necessary for building an ε-DPE: `class MultiComFlowBuilder`.

Such a builder is constructed with the following three arguments:

- `network: Network`. Contains the graph, a list of commodities as well as capacities and travel times of the graph's edges.
- `predictors: List[Predictor]`. Contains a list of all predictors that are used in the network's commodities.
- `reroute_interval: float`. The value ε for building an ε-DPE.

The idea for building such a flow is as follows. We start with an "empty" flow, i.e. a flow with time horizon `phi` $= 0$. We calculate shortest paths for each commodity according to predictions taken at time 0. Then, we iteratively extend the flow using constant flow rates (with the `extend` method of `class MultiComFlow`) until some node inflow rate changes or the next reroute time has been reached, i.e. `phi` $= k \cdot \varepsilon$ for some $k \in \mathbb{Z}$. Node inflow rates can only change if some edge outflow rate or the network inflow rate of some commodity change. The extension procedure of `class MultiComFlow` already takes care of changes in the edge outflow rates, so it suffices to keep track of changes in the network inflow rates.

Data Structure 6 The `class MultiComFlowBuilder`

```
class MultiComFlowBuilder:
  network: Network
  predictors: List[Predictor]
  reroute_interval: float
  _flow: MultiComFlow
  _relevant_nodes: Dict[Commodity, Set[Node]]
  _net_inflow_changes: PriorityQueue[Tuple[Commodity, float]]
  _next_reroute_time: float
  _route_time: float
  _costs: Dict[Predictor, Dict[Edge, PiecewiseLinear]]
  _active_edges: Dict[Commodity, Dict[Node, Set[Edge]]]
  _handle_nodes: Set[Node]
```

In Data Structure 6 all internal state attributes of the builder are described prefixed with a `_`. Most importantly the actual flow is written to `_flow`. Furthermore, for calculating shortest paths, we save the set of nodes lying on any path from a source to a sink for each commodity in the attribute `_relevant_nodes`. The priority queue `_net_inflow_changes` is initialized with all events of the type (c, t) where the network inflow rate of a commodity c changes at time t. In the attribute `_next_reroute_time` we save the next point in time at which routes should be recalculated. Initially, it has the value `_flow.phi`. The attributes `_route_time` and `_costs` get updated every time new routes are computed: In `_route_time` we save the time at which the latest routes have been computed (and at which time the predictions of queues have been taken) and the attribute `_costs[p]` saves the corresponding dynamic edge cost functions

$(c_{p,e}(\cdot, \bar{\theta}, q))_{e \in E}$ with $\bar{\theta} =$ `_route_time` for a predictor `p`. As calculating shortest paths is computationally expensive, we employ a cache in the attribute `_active_edges`. Here, `_active_edges[c]` is a dictionary assigning active outgoing edges $\hat{E}(\bar{\theta}, \bar{\theta}, q) \cap \delta_v^-$ to nodes v for the commodity `c`. These dictionaries are cleared at each reroute time step and filled only on demand, that means only if there is some positive inflow of that commodity into the node which needs to be assigned to outgoing edges. Finally, the attribute `_handle_nodes` is a set containing all nodes whose inflow should be redistributed to outgoing edges in the next extension phase: Only a subset of nodes whose inflow has changed or — at reroute steps — all nodes.

Algorithm 10 The Build Procedure in `class MultiComFlowBuilder`

```
def build_until(self, horizon: float) -> MultiComFlow:
  graph = self.network.graph
  travel_time = self.network.travel_time
  capacity = self.network.capacity

  while self._flow.phi < horizon:
    while self._net_inflow_changes.min_key() <= self._flow.phi:
      commodity, t = self._net_inflow_changes.pop()
      self._handle_nodes.add(commodity.source)
    if self._flow.phi >= self._next_reroute_time:
      predictions = {
        p: p.predict(self._flow.phi, self._flow.queues)
        for p in self.predictors
      }
      self._costs = {
        p: [travel_time[e] + predictions[p][e] / capacity[e]
            for e in graph.edges]
        for p in self.predictors
      }
      self._active_edges = {c: {} for c in self.network.commodities}
      self._route_time = self._next_reroute_time
      self._next_reroute_time += self.reroute_interval
      self._handle_nodes = set(graph.nodes)

    new_inflow = self._determine_new_inflow()
    max_ext_time = min(
      self._next_reroute_time, self._net_inflow_changes.min_key()
    )
    changed_edges = self._flow.extend(new_inflow, max_ext_time)
    self._handle_nodes = set(e.node_to for e in changed_edges)

  return self._flow
```

Once the user has successfully constructed such a builder and the attributes have been initialized, the user can call the function `builder.build_until` on it which is shown in Algorithm 10. This method computes an ε-DPE up to some given time horizon. In the following, we discuss the details of this method.

The main loop in lines 6–30 terminates only once the flow has been extended up to the desired time horizon. In each iteration of this main loop, the flow is extended once with new constant inflow rates using the `_flow.extend` method.

Before that, we take care of network inflow rate changes in lines 7–9 by adding all source nodes of commodities whose inflow rate have changed to the set `_handle_nodes`.

Furthermore, the lines 10–23 are executed if the routes have to be recalculated. In that case, we first instantiate new predictions of the queue lengths for each predictor. From that, we can calculate the cost functions by dividing by the edge capacities and adding the travel times. After saving these to the attribute `_costs`, we clear the cache of the active edges and update `_route_time` and `_next_reroute_time`. The set `_handle_nodes` gets the set of all nodes assigned, as any node with positive inflow needs to update its outgoing flow distribution according to the new routes.

Lines 25–30 are executed in every iteration of the loop: We first determine new constant inflow rates using the current dynamic costs functions in `_costs`. The maximum extension time is the minimum of the next reroute time and the next network inflow change time. Then, we call `_flow.extend` with these two arguments to extend the flow with the new inflow rates until either some edge outflow rate changes or the maximum extension time is reached. This function call returns the set of edges whose outflow rate changed at the new time `_flow.phi`. Hence, we assign `_handle_nodes` the set of target nodes of those edges.

Next, we discuss Algorithm 11 which takes care of calculating the new constant inflow rates of all outgoing edges of nodes v in `_handle_nodes`. For all commodities i, we calculate the total inflow rate $b^+_{i,v}(\phi) = \sum_{e\in\delta^+_v} f^-_{e,i}(\phi) + \mathbf{1}_{v=s_i} u_i(\phi)$ into v at time $\phi =$ `_flow.phi`. If this value is positive, we retrieve all active outgoing edges for this commodity and node under the current predictions of the commodity's predictor using the method `_get_active_edges`. We then distribute all inflow uniformly into all active outgoing edges. All other outgoing edges get a zero inflow rate.

The final ingredient of the flow builder is the method `_get_active_edges` for retrieving the currently active edges for a commodity and a node. It is shown in Algorithm 12. Of course, if the active edges of the corresponding node have already been computed, then we simply return the cached value from `_active_edges`. Otherwise, we still have to compute them. We do a case distinction on whether the predictor of a commodity i returns constant queue length functions only.

In the case of constant queue lengths, all edge cost functions $c_{i,e}(\,\cdot\,, \bar{\theta}, q)$ are constant as well with $c_{i,e}(\,\cdot\,, \bar{\theta}, q) \equiv\colon c_{i,e}(\bar{\theta}, q) \in \mathbb{R}_{>0}$. If we denote

$$\operatorname{dist}_{i,v}(\bar{\theta}, q) \coloneqq \min_{e_1 \cdots e_k \in \mathcal{P}_{v,t_i}} \sum_{l=1}^{k} c_{i,e_k}(\bar{\theta}, q)$$

as the $\bar{\theta}$-predicted distance from node v to the sink t_i, the arrival functions $l_{i,v}(\,\cdot\,, \bar{\theta}, q)$

Algorithm 11 The Inflow Calculation in `class MultiComFlowBuilder`

```
def _determine_new_inflow(self) -> Dict[Edge, List[float]]:
  new_inflow = {}
  m = len(self.network.commodities)
  for v in self._handle_nodes:
    new_inflow.update({e: [0.] * m for e in v.outgoing_edges})
    for i, commodity in enumerate(self.network.commodities):
      if v not in self._relevant_nodes[commodity] or v == commodity.sink:
        continue
      inflow = sum(self._flow.outflow[e][i](self._flow.phi)
                   for e in v.incoming_edges)
      if v == commodity.source:
        inflow += commodity.net_inflow(self._flow.phi)
      if inflow > 0:
        active_edges = self._get_active_edges(commodity, v)
        distribution = inflow / len(active_edges)
        for e in active_edges:
          new_inflow[e][i] = distribution
  return new_inflow
```

Algorithm 12 Retrieving Active Edges in `class MultiComFlowBuilder`

```
def _get_active_edges(self, c: Commodity, s: Node) -> Set[Edge]:
  if s in self._active_edges[c]:
    return self._active_edges[c][s]

  nodes = self._relevant_nodes[c]
  if self.predictors[c.predictor].is_constant():
    costs = [cost.values[0] for cost in self._costs[c.predictor]]
    distances = static_dijkstra(c.sink, costs, nodes, reverse=True)
    for v in nodes:
      self._active_edges[c][v] = set(
        e for e in v.outgoing_edges
        if e.node_to in distances.keys() and \
        costs[e.id] + distances[e.node_to] <= distances[v]
      )
  else:
    self._active_edges[c][s] = get_active_edges(
      self._costs[c.predictor], self._route_time, s, c.sink,
      nodes, self.network.graph, False
    )
  return self._active_edges[c][s]
```

can be written as

$$l_{i,v}(\theta, \bar{\theta}, q) = \theta + \mathrm{dist}_{i,v}(\bar{\theta}, q),$$

leaving an edge $e = vw$ $\bar{\theta}$-predicted active at time θ if and only if

$$\mathrm{dist}_{i,v}(\bar{\theta}, q) = c_{i,e}(\bar{\theta}, q) + \mathrm{dist}_{i,w}(\bar{\theta}, q).$$

Using the static version of the Dijkstra Algorithm on the reverse graph yields the distance vector $(\mathrm{dist}_{i,v}(\bar{\theta}, q))_{v \in V_i}$. Thus, as expected we only need a single run of the static Dijkstra Algorithm to compute the active edges of all nodes for that commodity. This is implemented in lines 6–14 of Algorithm 12. The implementation of the function `static_dijkstra` is not included in this report, however it is straight forward and well covered in standard literature.

If the queue length functions are not constant, we utilize the algorithm developed in Section 2.4. This means, that for any node (for which the function `_get_active_edges` is called) two instances of the Dynamic Dijkstra Algorithm have to be evaluated once in every routing phase.

4.6 Implementation of the Predictors

In this section we show how the different predictors introduced in Section 3.3 are implemented.

The simplest predictor is the Zero-Predictor $\hat{q}^{\mathrm{Z}}$ which simply returns a constant zero function for each edge as shown in Algorithm 13.

Algorithm 13 The Zero-Predictor $\hat{q}^{\mathrm{Z}}$.

```
1 class ZeroPredictor(Predictor):
2
3   def is_constant(self) -> bool:
4     return True
5
6   def predict(self, old_queues: List[PiecewiseLinear], phi: float) -> \
7       List[PiecewiseLinear]:
8     zero_fct = PiecewiseLinear([phi], [0.], 0., 0.)
9     return [zero_fct for _ in old_queues]
```

The constant predictor $\hat{q}^{\mathrm{C}}$ is slightly more advanced as it predicts constant functions with value $q_e(\phi)$ as shown in Algorithm 14. As the function $\hat{q}(\,\cdot\,, \bar{\theta}, q)$ is constant for the constant predictor $\hat{q}^{\mathrm{C}}$ and for the Zero-predictor $\hat{q}^{\mathrm{Z}}$, the method `is_constant` returns `true`. For all remaining predictors this is not the case and `is_constant` returns `false` for them.

Algorithm 14 The constant predictor $\hat{q}^{\mathrm{C}}$.

```
class ConstantPredictor(Predictor):

  def is_constant(self) -> bool:
    return True

  def predict(self, old_queues: List[PiecewiseLinear], phi: float) -> \
      List[PiecewiseLinear]:
    return [
      PiecewiseLinear([phi], [queue(phi)], 0., 0.)
      for queue in old_queues
    ]
```

For the linear predictor and the regularized linear predictor, we assume that the prediction horizon $H > 0$ is finite. The linear predictor can then be initialized using a value `horizon` representing H. To determine the gradient $\partial_- q_e(\phi)$, the method `gradient` has been implemented for the `class PiecewiseLinear`. The resulting linear predictor is shown in Algorithm 15.

Algorithm 15 The Linear Predictor

```
class LinearPredictor(Predictor):
  horizon: float

  def __init__(self, horizon: float):
    super(LinearPredictor, self).__init__()
    self.horizon = horizon

  def is_constant(self) -> bool:
      return False

  def predict(self, old_queues: List[PiecewiseLinear], phi: float) \
      -> List[PiecewiseLinear]:
    times = [phi, phi + self.horizon]
    queues = [None for _ in old_queues]
    for i, queue in enumerate(old_queues):
      val = max(0., queue(phi) + self.horizon * queue.gradient(phi))
      queues[i] = PiecewiseLinear(times, [queue(phi), val], 0., 0.)
    return queues
```

The regularized linear predictor is again a bit easier to implement, as we only evaluate the queue q_e at multiple time steps. The regularization constant δ is given as another argument to the constructor of the class as shown in Algorithm 16.

Algorithm 16 The Regularized Linear Predictor

```
class RegularizedLinearPredictor(Predictor):
  horizon: float
  delta: float

  def __init__(self, horizon: float, delta: float):
    super(RegularizedLinearPredictor, self).__init__()
    self.horizon = horizon
    self.delta = delta

  def is_constant(self) -> bool:
      return False

  def predict(self, old_queues: List[PiecewiseLinear], phi: float) \
      -> List[PiecewiseLinear]:
    times = [phi, phi + self.horizon]
    queues = [None for _ in old_queues]
    for i, queue in enumerate(old_queues):
      slope = (queue(phi) - queue(phi - self.delta)) / self.delta
      val = max(0., queue(phi) + self.horizon * slope)
      queues[i] = PiecewiseLinear(times, [queue(phi), val], 0., 0.)
    return queues
```

The most challenging predictor to implement was of course the machine-learned predictor. After some attempts with the *Tensorflow* framework [7], the *Weka* tool [3] and the *Deep Graph Library* [11] together with the *PyTorch* framework [9], more detailed experiments were deducted using the *scikit-learn* framework [10], which provides very efficient implementations of linear regression methods.

As explained in Section 3.3, for each edge $e = vw$ we want to learn coefficient matrices $W^{e'} \in \mathbb{R}^{k_p \times k_f}$ for neighbouring edges $e' \in N(e)$ and biases $\beta \in \mathbb{R}^{k_f}$ to compute the interpolation points

$$\hat{q}_e^{\mathrm{ML}}\left(\bar{\theta} + j \cdot \delta, \bar{\theta}, q\right) := \left(\sum_{e' \in N(e)} \sum_{i=1}^{k_p} w_{i,j}^{e'} \cdot q_{e'}\left(\bar{\theta} - (i-1) \cdot \delta\right) + \beta_j \right)^+,$$

for $j \in \{1, \ldots, k_f\}$, $i \in \{1, \ldots, k_p\}$ and $N(e) = \delta_v^- \cup \{e\} \cup \delta_w^+$.

For graphs with a large number of edges this quickly gets computationally infeasible. Hence, for such a graph we learn a fixed number of matrices and apply them on all edges. More specifically, let $\deg^+ := \max_{v \in V} |\delta_v^+|$ be the maximum in-degree and $\deg^- := \max_{v \in V} |\delta_v^-|$ the maximum out-degree of any node. We learn matrices $W^{+1}, \ldots, W^{+\deg^+}$, W^0, $W^{-1}, \ldots, W^{-\deg^-}$ and biases $\beta \in \mathbb{R}^{k_f}$. These generalized parameters can now be applied to predict the queue lengths of any edge $e = vw$: We arbitrarily order the

neighboring edges using $\delta_v^+ =: \{e^{+1}, \dots, e^{+|\delta_v^+|}\}$, $e =: e^0$ and $\delta_w^- =: \{e^{-1}, \dots, e^{-|\delta_w^-|}\}$ and compute

$$\hat{q}_e^{\mathrm{ML}}\left(\bar{\theta} + j \cdot \delta, \bar{\theta}, q\right) := \left(\sum_{e^l \in N(e)} \sum_{i=1}^{k_p} w_{i,j}^l \cdot q_{e^l}\left(\bar{\theta} - (i-1) \cdot \delta\right) + \beta_j \right)^+ .$$

For graphs of any size, we train the linear regression model on samples of queue functions that have previously been computed by the extension procedure of Section 4.5. In these training flows all commodities exclusively use the constant predictor.

As previously noted, using this prediction method as described by the formulas above does not necessarily lead to a FIFO-compatible predictor. To force the FIFO-compatibility of such a predictor, we have to make sure that the derivative of $\hat{q}^{\mathrm{ML}}(\,\cdot\,, \bar{\theta}, q)$ never subceeds $-\nu_e$. We do this by sequentially reassigning

$$\hat{q}_e^{\mathrm{ML}}\left(\bar{\theta} + j \cdot \delta, \bar{\theta}, q\right) := \max\left\{\hat{q}_e^{\mathrm{ML}}\left(\bar{\theta} + j \cdot \delta, \bar{\theta}, q\right), \hat{q}_e^{\mathrm{ML}}\left(\bar{\theta} + (j-1) \cdot \delta, \bar{\theta}, q\right) - \delta \cdot \nu_e\right\}$$

for $j = 1, \dots, k_f$ in a post-processing step.

The full implementation of the predictor for small graphs with $\delta = 1$ can be seen in Algorithm 17; the predictor for larger graphs is omitted here. The depicted predictor can be constructed using instances of `class sklearn.linear_model.LinearRegression`, i.e. one prelearned linear regression model per edge. Such a model can be utilized by calling its method `predict`. The variable `prediction` in Algorithm 17 then holds the k_f values of the computed interpolation points.

Algorithm 17 The Machine-Learned Predictor $\hat{q}^{\mathrm{ML}}$ for small graphs

```python
class PerEdgeLinearRegressionPredictor(Predictor):
  network: Network
  past_timesteps: int
  future_timesteps: int

  def __init__(self, models: List[LinearRegression], past_timesteps: int,
      future_timesteps: int, network: Network):
    super().__init__(network)
    self.models = models
    self.past_timesteps = past_timesteps
    self.future_timesteps = future_timesteps
    self.network = network

  def is_constant(self) -> bool:
    return False

  def predict(self, old_queues: List[PiecewiseLinear], phi: float) \
      -> List[PiecewiseLinear]:
    times = [phi + t for t in range(0, self.future_timesteps + 1, 1)]
    past_times = [phi - t for t in range(-self.past_timesteps + 1, 1)]
    queues = [None for _ in old_queues]
    for e in self.network.graph.edges:
      model_inputs =
        [old_queues[ie.id](t) for t in past_times
         for ie in e.node_from.incoming_edges] + \
        [old_queues[oe.id](t) for t in past_times
         for oe in e.node_to.outgoing_edges] + \
        [old_queues[e.id](t) for t in past_times]
      prediction = self.models[e.id].predict([model_inputs])[0]
      cap = self.network.capacity[e.id]
      new_values = [old_queues[e.id](phi)]
      for j in range(1, len(new_values)):
        new_values.append(
          max(prediction[j - 1], new_values[j - 1] - cap, 0.)
        )
      queues[e.id] = PiecewiseLinear(times, new_values, 0., 0.)
    return queues
```

5 Results of the Computational Study

We want to compare the different predictors that we introduced in the last section. As a metric for the performance of the different predictors, we monitor their average travel times in an ε-DPE with multiple predictors used side by side: Let i be a commodity with network inflow rate

$$u_i(\theta) \coloneqq \begin{cases} \bar{u}_i, & \text{for } \theta \leq h, \\ 0, & \text{for } \theta > h, \end{cases}$$

where $\bar{u}_i \in \mathbb{R}_{>0}$ is the constant inflow rate up to some time h. We denote the outflow rate of commodity i out of the network by $o_i(\theta) \coloneqq \sum_{e \in \delta_t^+} f_{i,e}^-(\theta) - \sum_{e \in \delta_t^-} f_{i,e}^+(\theta)$. Taking the integral of $u_i(\psi) - o_i(\psi)$ over $[0, \phi]$ yields the amount of flow of commodity i that is inside the network at time ϕ. If we integrate the flow inside the network over some time period $[0, H]$ with $H \geq h$, we obtain the *total travel time* of particles of commodity i up to time H:

$$T_i^{\text{total}} \coloneqq \int_0^H \int_0^\phi u_i(\psi) - o_i(\psi) \,\mathrm{d}\psi \,\mathrm{d}\phi.$$

The *average travel time* is defined as $T_i^{\text{avg}} \coloneqq T_i^{\text{total}} / (h \cdot \bar{u}_i)$.

To determine a commodity's regret of a chosen predictor, we additionally need to compute the minimum travel time given a computed ε-DPE f. We first determine the labels $(l_{i,v}(\,\cdot\,))_{v \in V_i}$ using the Bellman-Ford-Algorithm 4 with the dynamic edge costs induced by the computed queue length functions $(q_e(\,\cdot\,))_{e \in E}$ of f. The minimum travel time at time θ (up to time horizon H) can then be computed using $\min\{H, l_{i,s}(\theta)\} - \theta$. The *total minimum travel time* of all particles of a commodity can be expressed as

$$T_{i,\text{OPT}}^{\text{total}} \coloneqq \int_0^H u_i(\theta) \cdot (\min\{H, l_{i,s}(\theta)\} - \theta) \,\mathrm{d}\theta = \bar{u}_i \cdot \int_0^h \min\{H, l_{i,s}(\theta)\} - \theta \,\mathrm{d}\theta.$$

From that we derive the *average minimum travel time* as

$$T_{i,\text{OPT}}^{\text{avg}} \coloneqq \frac{T_{i,\text{OPT}}^{\text{total}}}{\int_0^h u_i(\theta) \,\mathrm{d}\theta} = \frac{1}{h} \cdot \int_0^h \min\{H, l_{i,s}(\theta)\} - \theta \,\mathrm{d}\theta.$$

5.1 Data

We conduct our experiments on three graphs. The first is a warm-up synthetic graph with 4 nodes and 5 edges. We present the graph in Figure 2. The second graph is the road map of Sioux Falls as given in [6] which is commonly used in the transport science literature. This public data set comes with edge attributes free-flow travel time τ_e and capacity ν_e. The third graph is the center of Tokyo, as obtained from Open Street Maps [8]. This graph includes information about the free-flow speed, the length and the numbers of lanes of each road segment e. We compute the transit time τ_e as the product of the free-flow speed and the length of edge e. The capacity ν_e is calculated by multiplying the number of lanes with the free-flow speed.

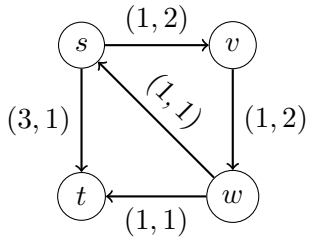

Figure 2: A network with source s and sink t. Edges are labeled with (τ_e, ν_e).

In the first network, we only have one commodity with source s and sink t for each predictor. For the latter two networks, commodities are randomly chosen. More detailed information on all networks are depicted in Table 1. Besides the configuration of all predictors, the average time for computing a single ε-DPE up to time H is shown in row $T^{\text{avg}}_{\text{comp}}$. Here, each predictor is used by exactly one commodity except for the constant predictor which is used by all other commodities. The resulting computation times were taken on a single core of an Intel® Core™ i7-3520M CPU at 2.90GHz.

Network	**Synthetic**	**Sioux Falls**	**Tokyo**
$\lvert E\rvert$	5	75	4,803
$\lvert V\rvert$	4	24	3,538
$\lvert I\rvert$	5	17	40
$[\nu_{\min}, \nu_{\max}]$	$[1, 2]$	$[4823, 25901]$	$[8, 250]$
$[\tau_{\min}, \tau_{\max}]$	$[1, 3]$	$[2, 10]$	$[0.01, 6.6]$
H	100	100	100
h	25	25	25
ε	0.25	1	2.5
H for $\hat{q}^{\text{L}}$	10	20	20
δ, H for $\hat{q}^{\text{RL}}$	$5, 10$	$1, 20$	$1, 20$
δ, k_p, k_f for $\hat{q}^{\text{ML}}$	$1, 10, 10$	$1, 20, 20$	$1, 20, 20$
ML model for $\hat{q}^{\text{ML}}$	single	per-edge	single
$T^{\text{avg}}_{\text{comp}}$	$0.33s$	$10.92s$	$343.93s$

Table 1: Attributes and configuration of the considered networks.

5.2 Comparison of Predictors

In this section we finally carry out the experiments and compare the average travel times of the different predictors.

Results on the Synthetic Network

We first take a closer look at the synthetic network as shown in Figure 2. Here, we want to analyze how the average travel times of competing predictors evolve while increasing

the total network inflow. For each oblivious predictor described in Section 3.3, we add a commodity $i \in \{\hat{q}^{\mathrm{Z}}, \hat{q}^{\mathrm{C}}, \hat{q}^{\mathrm{L}}, \hat{q}^{\mathrm{RL}}, \hat{q}^{\mathrm{ML}}\}$. Each of these commodities has the same source s and sink t and the same constant inflow $\bar{u}_i$ up to time $h = 25$. The outcome of running the simulation with time horizon $H = 100$ for each sampled total inflow in $(0, 30)$ can be seen in Figure 3. The ML based predictor performed best, while notably the Zero-Predictor, which distributes flow along paths (s, t) and (s, v, w, t) uniformly at all times, performs better than the remaining predictors. We note, that here the machine-learned model of the Tokyo network was used.

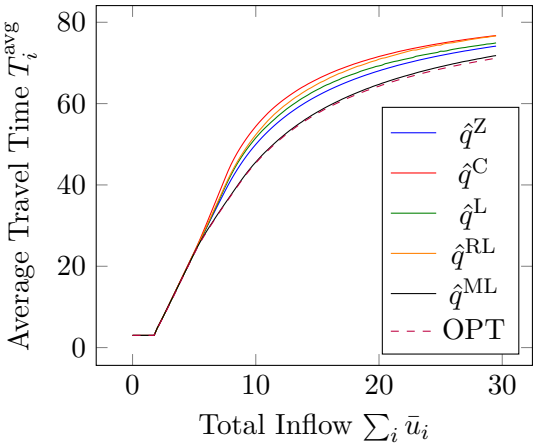

Figure 3: Measured average travel times of competing predictors in the synthetic network in Figure 2.

Results on the Sioux Falls Network

For the road-networks of Sioux Falls and Tokyo, we randomly generate inflow rates according to the edge capacities of the network. For each commodity i, we ran the simulation after adding 5 additional commodities with the same source and sink as i, one for each predictor, with a very small constant inflow rate. We monitored their average travel time as a measure of the performance of the different predictors. All other commodities in the network were assigned the constant predictor, such that the resulting queues should behave similar to the training data.

As the number of edges is small enough in the Sioux Falls network, we trained a separate model for each edge each with a 90%/10% split for the training data and test data. The coefficient of determination was above 0.9 for all edges except for 6 edges but always higher than 0.5.

Evaluating the average travel time of the different predictors for 12 random commodities yields the results depicted in Figure 4. We can see that the linear regression predictor $\hat{q}^{\mathrm{ML}}$ performs similarly well as the regularized linear predictor $\hat{q}^{\mathrm{RL}}$ which is slightly beaten by the constant and the linear predictors. The Zero predictor performs worse than the others many times.

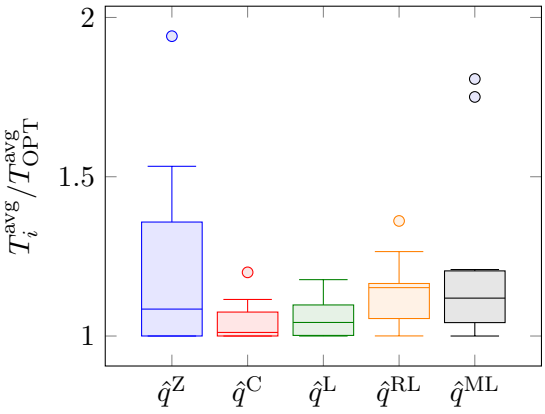

Figure 4: Average travel times compared to the minimum average travel time in the Sioux Falls network.

Results on the Tokyo Network

Because the Tokyo instance has substantially more edges, we decided to train a single model used for all edges. A training and validation split of 90%/10% yields a coefficient of determination of 0.97. Running the evaluation now on 35 randomly chosen commodities gives the results shown in 5. In this scenario, the linear regression predictor performed similarly well as the linear and the constant predictor. Here, the Zero predictor performs slightly better than the regularized linear predictor, but worse than the remaining three.

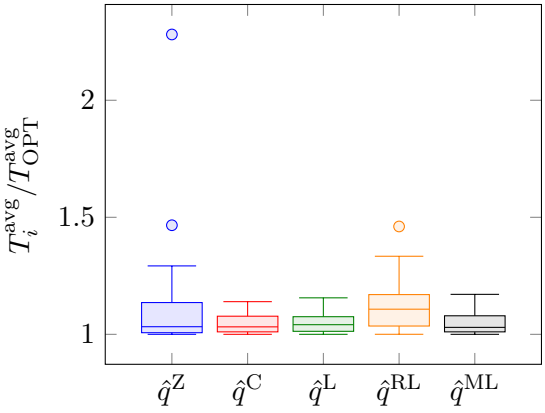

Figure 5: Average travel times compared to the minimum average travel time in the Tokyo network.

6 Conclusion

In this report we have introduced a method to compute ε-DPEs. We introduced several predictors and their implementations and showed how machine learning methods can be embedded in this simulation. This approach can be used to analyze the use of machine learning based navigation systems in everyday traffic. The benefit over other traffic simulations is that it follows a very concise and clear mathematical model that can also be analyzed theoretically.

We want to conclude this report with several aspects for future work.

First, the current version of the software does not come with a general-graph visualization of the calculated flows. It is clear, that a good visualization can improve the understanding and intuition of the underlying model. Moreover, it is interesting to see which edges become bottlenecks for a given scenario and network and inspect how changes to these edges or the insertion of edges influences the resulting flow.

Second, the machine learning model used here can still be improved as can be seen in Section 5. It might be worth changing the feature set given to the learning algorithm: The queue length of surrounding edges might not be the only indicator of an upcoming queue at the future. For example if a surrounding edge has a very high capacity and transports an amount of flow close to its capacity, this flow cannot be detected in the current feature set, because no queue forms in front of the edge. However, changing the feature set to include more than the past queue lenghts also changes the mathematical model of the predictor functions $\hat{q}_{i,e}(\cdot, \cdot, \cdot)$. Changing the arguments of this function might imply that the existence of equilibria as given by Theorem 3.10 does not hold anymore.

Finally, the quality of approximation of ε-DPEs is still to be analyzed. A desirable result would be that for ε tending to 0, the ε-DPE converges to a real DPE. However, in this report we leave this question open for further research.

References

[1] Roberto Cominetti, José Correa, and Omar Larré. "Dynamic Equilibria in Fluid Queueing Networks". In: *Operations Research* 63.1 (2015), pp. 21–34. URL: http://www.jstor.org/stable/24540348.

[2] Bolin Ding, Jeffrey Xu Yu, and Lu Qin. "Finding Time-Dependent Shortest Paths over Large Graphs". In: *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '08. Nantes, France: Association for Computing Machinery, 2008, pp. 205–216. DOI: 10.1145/1353343.1353371.

[3] E. Frank et al. "Weka: A machine learning workbench for data mining." In: *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers*. Ed. by O. Maimon and L. Rokach. Berlin: Springer, 2005, pp. 1305–1314. URL: http://researchcommons.waikato.ac.nz/handle/10289/1497.

[4] Lukas Graf, Tobias Harks, Kostas Kollias, and Michael Markl. *Machine-Learned Prediction Equilibrium for Dynamic Traffic Assignment*. 2021. arXiv: 2109.06713 [cs.GT].

[5] Lukas Graf, Tobias Harks, and Leon Sering. "Dynamic flows with adaptive route choice". In: *Mathematical Programming* 183.1-2 (May 2020), pp. 309–335. DOI: 10.1007/s10107-020-01504-2.

[6] Larry J. LeBlanc, Edward K. Morlok, and William P. Pierskalla. "An efficient approach to solving the road network equilibrium traffic assignment problem". In: *Transportation Research* 9.5 (1975), pp. 309–318. ISSN: 0041-1647. DOI: 10.1016/0041-1647(75)90030-1.

[7] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[8] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2017.

[9] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[10] Fabian Pedregosa et al. "Scikit-Learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435.

[11] Minjie Wang et al. "Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs". In: *CoRR* abs/1909.01315 (2019). arXiv: 1909.01315.