A Scalable Descent Algorithm for Network Design Problems

Jing Yu[1] Qianni Wang[1] Yu (Marco) Nie[1] Jiayang Li[2,*]

Abstract

In a network design problem (NDP), a traffic planner aims to improve the performance of a transportation network by anticipating and manipulating the Wardrop equilibrium (WE) state achieved by the travelers. A critical challenge in solving NDPs lies in computing the derivatives of their objectives, which, according to the chain rule, further requires the gradient of WE. In the literature, this is a long-standing computational bottleneck that has limited the application of general descent algorithms to NDPs. Recently, many have attempted to compute the gradient of WE by unrolling a numerical algorithm that solves WE with automatic differentiation (AD) tools. Although these methods have better time efficiency than conventional benchmarks, they still face severe memory constraints from storing the full computational process for solving WE. In this work, we find that unrolling fixed-point iterations of a WE solver initialized at a WE solution also provides exact WE gradients. We further develop an unrolling scheme whose time efficiency is identical to AD-based approaches. Yet, by recognizing each fixed-point iteration as identical, our method requires only single-iteration memory storage, which is a dramatic improvement over AD-based approaches. Numerical experiments across classical NDP benchmarks confirm our method's superior scalability.

Keywords: network design, bilevel programming, implicit differentiation, automatic differentiation

1 Introduction

In transportation, a network design problem (NDP) aims to optimally design or redesign a transportation network, for whose limited capacity travelers compete to minimize their own travel cost (Migdalas, 1995; Yang and Bell, 1998). It is typically assumed that the competition between the travelers leads the network to a Wardrop (1952) equilibrium (WE), which dictates that no traveler can reduce cost by unilaterally changing their decision (e.g., taking a different route). Such NDPs can be formulated as a bilevel program with a WE being the solution to its lower-level problem. This is a class of optimization problems well known for their strong NP-hardness (Roch et al., 2005; Hoefer et al., 2008; Gairing et al., 2017). To bypass the bilevel structure, one may convert the NDP into a single-level optimization problem with WE constraints, which may be written as the KKT conditions (Bard and Falk, 1982) or an equivalent variational inequality (VI) for

[1]Department of Civil and Environmental Engineering, Northwestern University. [2]Department of Data and Systems Engineering, The University of Hong Kong. *Corresponding author; email: jiayangl@hku.hk.

the lower level problem. However, the KKT conditions introduce mixed-integer nonlinear constraints whose number grows exponentially with network size (Colson et al., 2007), whereas the VI constraint results in a semi-infinite program (Marcotte, 1983) — both are formidable computational challenges in their own right.

Given the inherent complexities, a good stationary point (i.e., a local optimum) is often the best one can hope to achieve for NDPs of practical sizes. Yet, even this goal is not always attainable on large-scale applications. To locate a stationary point, standard descent algorithms improve network designs iteratively based on the gradient of a design objective, which depends on the derivative of WE (Friesz et al., 1990). In literature, this is often done by implicitly differentiating WE conditions. However, implicit differentiation (ID) can be computationally prohibitive since it requires inverting a square matrix whose dimension is no less than the number of routes used at WE (Tobin and Friesz, 1988; Dafermos, 1988; Cho et al., 2000; Yang and Bell, 2007). Several approximation schemes were proposed to obviate direct matrix inversion (see, e.g., Liu et al., 2023, for a discussion). Yet, to the best of our knowledge, none of these schemes can avoid costly matrix operations (e.g., storage and multiplication) altogether.

Recently, automatic differentiation (AD) (Griewank, 1989) has emerged as a potential game-changer to scale descent algorithms for NDPs. AD typically consists of two phases. The forward propagation (FP) phase executes a computing routine while storing it, along with intermediate variables, in a computational graph. The backward propagation (BP) phase then obtains the gradient of the routine by "unrolling" — or applying the chain rule recursively to — the computational graph. A key strength of AD lies in the guarantee that the computational cost of the BP can be bounded by a small multiple of the FP's cost.

It has been suggested that one may compute the gradient of a bilevel program by applying AD to a *smooth* algorithm (i.e., one without non-differentiable operations) that solves its lower-level problem. This idea first gained traction in bilevel programs arising from machine learning (ML) applications, such as hyperparameter tuning (Maclaurin et al., 2015). The adaptation to the NDP soon followed suit. Li et al. (2020) applied AD to the projection method (Bertsekas and Gafni, 1982), which was employed to solve the lower-level WE problem in the NDP. Li et al. (2022) further showed that imitative logit dynamics (ILD), a WE solver derived from evolutionary game theory (Björnerstedt and Weibull, 1994), is more efficient in AD-based unrolling. Notably, they proved that unrolling ILD avoids expensive matrix operations (including inversions and multiplications), which brings about significant computational advantages.

Challenges. Despite the promises, AD-based methods come with their own computational challenges. Chief among these is the substantial memory requirement imposed by the need to store the computational graph generated from solving the lower-level problem. To produce accurate gradients, a highly precise WE solution is needed. Yet, because WE algorithms suitable for AD-based unrolling often converge slowly, it is not uncommon for them to take thousands of iterations to reach a satisfactory solution (Li et al., 2022). Storing such a "deep" computational graph can be a critical constraint, especially on large networks.

Many have sought to address this issue (see, e.g., Liu et al., 2021, for an overview). A popular heuristic, often

called truncated AD, stores only the final portion of the computational graph (Shaban et al., 2019). When applied to NDPs, the heuristic is translated to differentiating through only the last few iterations of the WE solver. An equivalent interpretation is unrolling a warm-started WE solver. Pushing this logic of truncation to the extreme leads to a "quick-and-dirty" heuristic that unrolls only a single iteration from a "fully warmed-up solution," which may be viewed as the WE itself (McKenzie et al., 2024). While such strategies can reduce memory requirements dramatically, they offer relatively crude approximation of the gradient, which can jeopardize the overall convergence of the descent algorithm.

Our contribution. This work improves the existing heuristic by McKenzie et al. (2024), which approximates an NDP's gradient by unrolling a single-step fixed-point iteration at the WE using AD. The proposed algorithm is novel in its capability of computing the gradient of an NDP to any desired precision. This performance is guaranteed by a new theoretical result: increasing the number of unrolled fixed-point iterations at WE leads the approximate gradient to converge to the true gradient of the NDP. Crucially, to make storage manageable even for very large applications, the algorithm does not rely on AD. Instead, by leveraging the fact that each fixed-point iteration is identical, it retains only a single copy of the intermediate variables in such an iteration in memory. Consequently, the proposed algorithm has the best of both worlds: it offers the most accurate solution possible but consumes no more memory than the crudest heuristic (McKenzie et al., 2024). To validate its superiority, we test the proposed algorithm against established methods based on AD and ID across a wide range of benchmark applications.

Organization. The remainder of the paper is organized as follows. Section 2 introduces the basics of NDPs and Section 3 provides a thorough overview of previous descent algorithms for solving NDPs. In Section 4, the proposed approach is established with a detailed explanation of how it is motivated and how it is implemented. In Section 5, we test the proposed approach on various benchmark NDPs and compare it with many benchmark algorithms. Eventually, Section 6 concludes the paper and discusses future work.

Notation. We use $\mathbb{R}$ and $\mathbb{R}_+$ to denote the set of real numbers and non-negative real numbers. For a vector $\boldsymbol{a} \in \mathbb{R}^n$, we denote its support as $\operatorname{supp}(\boldsymbol{a}) = \{i : a_i > 0\}$ and its ℓ_2 norm as $\|a\|_2$. For two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$, their inner product and outer product are denoted as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ and $\boldsymbol{a} \otimes \boldsymbol{b}$, respectively. For a finite set $\mathcal{A}$, we write $|\mathcal{A}|$ as its number of elements and $2^{\mathcal{A}}$ as the set of its subsets.

2 Problem Setting

Consider a transportation network represented as a directed graph $\mathcal{G}(\mathcal{N}, \mathcal{A})$, where $\mathcal{N}$ and $\mathcal{A}$ are the set of nodes and links. In the network, the set of origin-destination (OD) pairs is denoted as $\mathcal{W} \subseteq \mathcal{N} \times \mathcal{N}$, and the set of the routes connecting all OD pairs is denoted as $\mathcal{K} \subseteq 2^{\mathcal{A}}$. Let $\mathcal{K}_w \subseteq \mathcal{K}$ be the set of routes connecting each OD pair $w \in \mathcal{W}$ and $\mathcal{A}_k \subseteq \mathcal{A}$ be the set of links on each route $k \in \mathcal{K}$. Further, we write $\Sigma_{w,k}$ as the OD-route incidence with $\Sigma_{w,k} = 1$ if the route $k \in \mathcal{K}_w$ and 0 otherwise, and $\Lambda_{a,k}$ as the link-route incidence, with $\Lambda_{a,k} = 1$ if $a \in \mathcal{A}_k$ and 0 otherwise. Let $\boldsymbol{\Lambda} = (\Lambda_{a,k})_{a \in \mathcal{A}, k \in \mathcal{K}}$ and $\boldsymbol{\Sigma} = (\Sigma_{w,k})_{w \in \mathcal{W}, k \in \mathcal{K}}$.

2.1 Wardrop equilibrium (WE)

Let travel demand be denoted by $\boldsymbol{d} = (d_w)_{w\in\mathcal{W}}$, where d_w represents the number of travelers between $w \in \mathcal{W}$. The travelers' route choice is represented by a vector $\boldsymbol{p} = (p_k)_{k\in\mathcal{K}}$, where p_k equals the proportion of travelers selecting $k \in \mathcal{K}_w$. The feasible region for $\boldsymbol{p}$ can then be written as $\mathcal{P} = \{\boldsymbol{p} \in \mathbb{R}_+^{|\mathcal{K}|} : \boldsymbol{\Sigma p} = \boldsymbol{1}\}$. Let $\boldsymbol{f} = (f_k)_{k\in\mathcal{K}}$ and $\boldsymbol{x} = (x_a)_{a\in\mathcal{A}}$, with f_k and x_a being the flow (i.e., number of travelers) on route k and link a, respectively. It follows $\boldsymbol{f} = \mathrm{diag}(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{d})\boldsymbol{p}$ and $\boldsymbol{\Lambda f} = \boldsymbol{x}$. Define $\boldsymbol{u} = (u_a)_{a\in\mathcal{A}}$ as a vector of link cost, determined by a continuously differentiable function $u(\boldsymbol{x}) = (u_a(x_a))_{a\in\mathcal{A}}$. Then, the vector of route cost $\boldsymbol{c} = \boldsymbol{\Lambda}^\mathsf{T}\boldsymbol{u}$. To summarize, the route cost function $c : \mathcal{P} \to \mathbb{R}^{|\mathcal{K}|}$ can be defined as $c(\boldsymbol{p}) = \boldsymbol{\Lambda}^\mathsf{T}\boldsymbol{u} = \boldsymbol{\Lambda}^\mathsf{T}u(\boldsymbol{\Lambda f}) = \boldsymbol{\Lambda}^\mathsf{T}u(\bar{\boldsymbol{\Lambda}}\boldsymbol{p})$, where $\bar{\boldsymbol{\Lambda}} = \boldsymbol{\Lambda}\mathrm{diag}(\boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{d})$.

Under the above setting, the definition of WE strategies is given below.

Definition 2.1 (WE strategy)**.** A route choice $\boldsymbol{p}^* \in \mathcal{P}$ is a WE strategy if it dictates all non-optimal routes are not used, i.e. $c_k(\boldsymbol{p}^*) > b_w^* \Rightarrow p_k^* = 0$, where $b_w^* = \min_{k'\in\mathcal{K}_w} c_{k'}(\boldsymbol{p}^*)$, for all $w \in \mathcal{W}$ and $k \in \mathcal{K}_w$.

We write $\mathcal{P}^* \subseteq \mathcal{P}$ as the set of WE strategies and $\mathcal{X}^* = \{\boldsymbol{x} : \boldsymbol{x} = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}^*, \boldsymbol{p}^* \in \mathcal{P}^*\}$ as the set of WE link flow patterns. The following two propositions (Dafermos, 1980) then describe their geometries.

Proposition 2.2. *If $c(\boldsymbol{p})$ is strictly monotone on $\mathcal{P}$, then $\mathcal{P}^*$ is a singleton.*

Proposition 2.3. *If $u(\boldsymbol{x})$ is strictly monotone on $\mathcal{X} = \{\boldsymbol{x} : \boldsymbol{x} = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}, \boldsymbol{p} \in \mathcal{P}\}$, then $\mathcal{X}^*$ is a singleton and $\mathcal{P}^* = \{\boldsymbol{p}^* \in \mathcal{P} : \bar{\boldsymbol{\Lambda}}\boldsymbol{p}^* = \boldsymbol{x}^*\}$, where $\boldsymbol{x}^* \in \mathcal{X}^*$.*

Note that the strict monotonicity of $u(\boldsymbol{x})$ is guaranteed as long as $u_a(x_a)$ is strictly increasing for all $a \in \mathcal{A}$, which can further ensure the monotonicity of $c(\boldsymbol{p})$. The *strict* monotonicity of $c(\boldsymbol{p})$, however, cannot be ensured unless $\boldsymbol{\Lambda}$ has a full column rank, a condition rarely satisfied in realistic networks (Sheffi, 1985).

2.2 Network design problem (NDP)

We are now ready to present the formulation of the NDP, in which a traffic planner chooses a network design to minimize a certain system objective. Let us write the planner's decision as $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^m$. We assume $\boldsymbol{\theta}$ may influence link costs either directly or indirectly, hence leading to a parameterized link cost function $u(\boldsymbol{x}; \boldsymbol{\theta})$. The path cost function $c(\boldsymbol{p}; \boldsymbol{\theta})$ and the set of WE strategies $\mathcal{P}^*(\boldsymbol{\theta})$ can be similarly parameterized. Let the planner's objective be determined by both its decision $\boldsymbol{\theta} \in \Theta$ and the WE link flow $\boldsymbol{x}^*$ through a differentiable function $l : \mathcal{X} \times \Theta \to \mathbb{R}$. The planner's decision problem then reads

$$\begin{aligned} \min_{\boldsymbol{\theta}\in\Theta}\ & l(\boldsymbol{x}^*; \boldsymbol{\theta}), \\ \text{s.t. }\ & \boldsymbol{x}^* = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}^*, \quad \boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta}). \end{aligned} \tag{2.1}$$

Throughout the study, we impose the following assumptions on Problem (2.1).

Assumption 2.4. The feasible region Θ is a convex set.

Assumption 2.5. The function $u(\boldsymbol{x};\boldsymbol{\theta})$ is twice continuously differentiable on $\mathcal{X}\times\Theta$.

Assumption 2.6. For all $\boldsymbol{\theta}\in\Theta$, the function $u(\cdot;\boldsymbol{\theta})$ is strictly monotone on $\mathcal{X}$.

To conclude this section, we introduce three classical applications of NDPs: the continuous network design problem (Abdulaal and LeBlanc, 1979), the second-best congestion pricing problem (Verhoef, 2002), and the profit-maximization pricing problem (Labbé et al., 1998). For more examples, the reader is referred to Migdalas (1995) and Farahani et al. (2013).

Example 2.7 (continuous network design problem, CNDP)**.** *In a CNDP, a traffic planner aims to expand road capacity on a predetermined set of links, denoted as $\widetilde{\mathcal{A}}\subseteq\mathcal{A}$. Let $\boldsymbol{v}_0=(v_{0,a})_{a\in\mathcal{A}}$ be the original capacity of each link and $\boldsymbol{\theta}=(\theta_a)_{a\in\mathcal{A}}$ be the capacity added by the manager, the link cost function is then written as $u(\boldsymbol{x};\boldsymbol{\theta})=u_{time}(\boldsymbol{x};\boldsymbol{v}_0+\boldsymbol{\theta})$. Let $m:\Theta\to\mathbb{R}$ be the cost associated with capacity enhancement. The objective of a CNDP is often written as a weighted sum of total travel time and monetary cost, namely $l(\boldsymbol{x}^*;\boldsymbol{\theta})=\langle\boldsymbol{x}^*,u_{time}(\boldsymbol{x}^*;\boldsymbol{v}_0+\boldsymbol{\theta})\rangle+\beta\cdot m(\boldsymbol{\theta})$, for some $\beta>0$.*

Example 2.8 (second-best congestion pricing problem, SCPP)**.** *In an SCPP, a traffic planner aims to reduce traffic congestion by charging travelers a congestion toll on selected links, denoted as $\widehat{\mathcal{A}}\subseteq\mathcal{A}$. Let $\boldsymbol{\theta}=(\theta_a)_{a\in\mathcal{A}}$ be the toll on each link, the link cost function then can be written as $u(\boldsymbol{x};\boldsymbol{\theta})=u_{time}(\boldsymbol{x})+\gamma\cdot\boldsymbol{\theta}$, where $u_{time}(\boldsymbol{x})$ is the time cost while γ is the time value of money. Under this setting, the feasible region of $\boldsymbol{\theta}$ is $\Theta=\{\boldsymbol{\theta}\in\mathbb{R}_+^{|\mathcal{A}|}:\theta_a=0,\forall a\in\widetilde{\mathcal{A}}\}$ and the objective of the traffic planner can be written as $l(\boldsymbol{x}^*;\boldsymbol{\theta})=\langle\boldsymbol{x}^*,u_{time}(\boldsymbol{x}^*)\rangle$, i.e., the total travel time of all travelers.*

Example 2.9 (profit-maximization pricing problem, PMPP)**.** *PMPP's setting is almost identical to SCPP's, except that the traffic planner's objective function turns from $l(\boldsymbol{x}^*;\boldsymbol{\theta})=\langle\boldsymbol{x}^*,u_{time}(\boldsymbol{x}^*)\rangle$, the total travel time, to $l(\boldsymbol{x}^*;\boldsymbol{\theta})=\langle\boldsymbol{x}^*,\boldsymbol{\theta}\rangle$, the total revenue collected from tolling the travelers. This corresponds to the scenario when the traffic planner aims to pay off the investment cost as soon as possible or the tollable roads are privately owned.*

3 Descent Algorithms for NDP

A popular approach to NDP is to iteratively improve the solution based on an appropriate descent direction until reaching a stationary point (see, e.g., Yang and Yagar, 1994; Chiou, 2005; Yang and Huang, 2005; Zhang and Nie, 2018, 2021; Li et al., 2022). In this section, we provide an outline of this approach.

Most descent algorithms for NDPs rely on an equivalent reformulation of Problem (2.1), which is described as follows. First, per Assumption 2.6, every WE route choice $\boldsymbol{p}^*\in\mathcal{P}^*(\boldsymbol{\theta})$ corresponds to the same $\boldsymbol{x}^*=\bar{\boldsymbol{\Lambda}}\boldsymbol{p}^*$. This means there exists a well-defined implicit function mapping $\boldsymbol{\theta}$ to the corresponding WE link flow $\boldsymbol{x}^*$. Let us write this implicit functions as $x^*(\boldsymbol{\theta})$. Further denoting $l^*(\boldsymbol{\theta})=l(x^*(\boldsymbol{\theta});\boldsymbol{\theta})$, Problem (2.1) can then be equivalently rewritten as

$$\min_{\boldsymbol{\theta}\in\Theta} l^*(\boldsymbol{\theta}). \tag{3.1}$$

The new formulation implies that given $\boldsymbol{\theta} \in \Theta$, the negative gradient of its objective function $-\nabla l^*(\boldsymbol{\theta})$ is the steepest descent direction for improving the solution. The following proposition provides a sufficient condition for the existence of $\nabla x^*(\boldsymbol{\theta})$, and consequently $\nabla l^*(\boldsymbol{\theta})$.

Proposition 3.1 (Patriksson (2004)). *For any $\boldsymbol{\theta} \in \Theta$, as long as there exists a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ that satisfies the strictly complementary (SC) condition, i.e. for all $w \in \mathcal{K}$ and $k \in \mathcal{K}_w$, it holds $p_k^* > 0$ whenever $c_k(\boldsymbol{p}^*; \boldsymbol{\theta}) = b_w^*$, where $b_w^* = \min_{k' \in \mathcal{K}_w} c_{k'}(\boldsymbol{p}^*; \boldsymbol{\theta})$, then $\nabla x^*(\boldsymbol{\theta})$ exists.*

In plain English, a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ satisfies the SC condition as long as there are no minimum-cost routes left unused by travelers. When $\mathcal{P}^*(\boldsymbol{\theta})$ is a polyhedron, its interior points generally satisfy the SC condition. Thus, the set of $\boldsymbol{\theta}$ for which *no* $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ satisfies SC has measure zero (Yang and Huang, 2005) — this allows general descent algorithms to be applicable to Problem (3.1).

In the remainder of this section, we will introduce both implicit and automatic differentiation methods for computing $\nabla l^*(\boldsymbol{\theta})$. Throughout the discussion, we assume the conditions given by Proposition 3.1 are satisfied at $\boldsymbol{\theta}$. To simplify the discussion of their complexities, we denote the upper bound of the number of routes used by travelers at WE, namely $\operatorname{supp}_{\boldsymbol{\theta} \in \Theta} \max_{\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})} |\operatorname{supp}(\boldsymbol{p}^*)|$, as $K_{\max}$.

3.1 Implicit differentiation (ID) methods

We begin by introducing several ID methods for computing $\nabla x^*(\boldsymbol{\theta})$, which, by the chain rule, then enables the computation of $\nabla l^*(\boldsymbol{\theta})$ as follows:

$$\nabla l^*(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} l(x^*(\boldsymbol{\theta}); \boldsymbol{\theta}) + \nabla x^*(\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{x}} l(x^*(\boldsymbol{\theta}); \boldsymbol{\theta}). \tag{3.2}$$

ID methods for computing $\nabla x^*(\boldsymbol{\theta})$ can be generally grouped into two classes: those based on the KKT conditions (Section 3.1.1) and those based on the fixed-point formulation(Section 3.1.2).

3.1.1 ID based on KKT conditions

This class of approaches computes the gradient of WE by applying ID to the KKT conditions of the lower-level problem, as defined in the following proposition.

Proposition 3.2 (The KKT conditions for WE). *Given any $\boldsymbol{\theta} \in \Theta$, a $\boldsymbol{p}^* \in \mathbb{R}^{|\mathcal{K}|}$ is a WE strategy if and only if there exists $\boldsymbol{b}^* \in \mathbb{R}^{|\mathcal{W}|}$ such that*

$$\begin{cases} c(\boldsymbol{p}^*; \boldsymbol{\theta}) - \boldsymbol{\Sigma}^\mathsf{T} \boldsymbol{b}^* \geq \mathbf{0}, \quad \langle c(\boldsymbol{p}^*; \boldsymbol{\theta}) - \boldsymbol{\Sigma}^\mathsf{T} \boldsymbol{b}^*, \boldsymbol{p}^* \rangle \geq 0, \\ \boldsymbol{\Sigma} \boldsymbol{p}^* = \mathbf{1}, \quad \boldsymbol{p}^* \geq \mathbf{0}, \end{cases} \tag{3.3}$$

where b_w^* is the minimum cost of used routes between each OD pair $w \in \mathcal{W}$.

Yet, the KKT conditions given above are not directly amenable to ID because they cannot uniquely define a

$\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$. To address this issue, we may impose additional conditions that force the choice of a unique WE strategy (Tobin and Friesz, 1988; Cho et al., 2000; Yang and Bell, 2007). Below, we briefly introduce Yang and Bell (2007)'s approach for the purpose of illustration, which is based on a special $\bar{\boldsymbol{p}}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ that satisfies the SC condition.

In Yang and Bell (2007), the routes that are not used by $\bar{\boldsymbol{p}}^*$ are removed at first. This simplification does not affect the computation of $\nabla x^*(\boldsymbol{\theta})$ because given $\bar{\boldsymbol{p}}^*$ is SC, there always exists a WE strategy whose support is identical to $\mathrm{supp}(\bar{\boldsymbol{p}}^*)$ with slight perturbation of $\boldsymbol{\theta}$. Consequently, the *local* behavior of $x^*(\boldsymbol{\theta})$ (and thus $\nabla x^*(\boldsymbol{\theta})$) would not be affected. To keep notation simple, let us still denote the set of routes after the removal as $\mathcal{K}$, and its corresponding OD-route and link-route incidence matrices as $\boldsymbol{\Sigma}$ and $\boldsymbol{\Lambda}$. Furthermore, divide $\mathcal{K}$ into two non-overlapping sets $\mathcal{K}_+$ and $\mathcal{K}_0$, where $\mathcal{K}_+$ corresponds to a *maximal set of independent columns* (MSIC) of the matrix $[\boldsymbol{\Lambda}^\mathsf{T}, \boldsymbol{\Sigma}^\mathsf{T}]^\mathsf{T}$. Let $\boldsymbol{\Lambda}_+$ and $\boldsymbol{\Sigma}_+$ be sub-matrices of $\boldsymbol{\Lambda}$ and $\boldsymbol{\Sigma}$, respectively, that contain only columns in $\mathcal{K}^+$. It can be shown that, for any slightly perturbed $\boldsymbol{\theta}$, there exists one and only one $p^*(\boldsymbol{\theta}) \in \mathcal{P}^*(\boldsymbol{\theta})$ such that (1) $p^*(\boldsymbol{\theta}) > \mathbf{0}$ (i.e. all routes are used) and (2) $p_k^*(\boldsymbol{\theta}) = \bar{p}_k^*$ for all $k \in \mathcal{K}^0$. Importantly, the KKT conditions for $p^*(\boldsymbol{\theta})$ are reduced to the following: there exists a $b^*(\boldsymbol{\theta}) \in \mathbb{R}^{|\mathcal{W}|}$ such that

$$\begin{cases} c_+(p^*(\boldsymbol{\theta}); \boldsymbol{\theta}) - \boldsymbol{\Sigma}_+^\mathsf{T} b^*(\boldsymbol{\theta}) = \mathbf{0}, \\ \boldsymbol{\Sigma}_+ p_+^*(\boldsymbol{\theta}) = \mathbf{1} - \boldsymbol{\Sigma}_0 \boldsymbol{p}_0^*, \end{cases} \tag{3.4}$$

where $c_+(\boldsymbol{p}^*(\boldsymbol{\theta}); \boldsymbol{\theta}) = (c_k(\boldsymbol{p}^*(\boldsymbol{\theta}); \boldsymbol{\theta}))_{k \in \mathcal{K}_+}$, $p_+^*(\boldsymbol{\theta}) = (p_k^*(\boldsymbol{\theta}))_{k \in \mathcal{K}^+}$, and $\boldsymbol{\Sigma}_0$ is the sub-matrix of $\boldsymbol{\Sigma}$ that contains the columns in $\mathcal{K}_0$. Implicitly differentiating Equation (3.4) with respect to $\boldsymbol{\theta}$ then yields

$$\begin{bmatrix} \nabla p_+^*(\boldsymbol{\theta})^\mathsf{T} \\ \nabla b^*(\boldsymbol{\theta})^\mathsf{T} \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{\Lambda}}_+^\mathsf{T} \nabla_{\boldsymbol{x}} u(\boldsymbol{x}^*(\boldsymbol{\theta}); \boldsymbol{\theta}) \boldsymbol{\Lambda}_+ & -\boldsymbol{\Sigma}_+^\mathsf{T} \\ \boldsymbol{\Sigma}_+ & \mathbf{0} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \boldsymbol{\Lambda}_+^\mathsf{T} \nabla_{\boldsymbol{\theta}} u(\boldsymbol{x}^*(\boldsymbol{\theta}); \boldsymbol{\theta}) \\ \mathbf{0} \end{bmatrix}, \tag{3.5}$$

and $\nabla x^*(\boldsymbol{\theta}) = \nabla p_+^*(\boldsymbol{\theta}) \cdot \bar{\boldsymbol{\Lambda}}_+^\mathsf{T}$, where $\bar{\boldsymbol{\Lambda}}_+$ is the sub-matrix of $\bar{\boldsymbol{\Lambda}}$ that contains the columns in $\mathcal{K}_+$.

The above approach contains three computationally intensive tasks: (1) finding a WE strategy that satisfies the SC condition; (2) finding an MSIC of the combined incidence matrix, and (3) solving the linear system (3.5). For (1), note that the SC condition implies that $\mathrm{supp}(\bar{\boldsymbol{p}}^*) = \cup_{\boldsymbol{p}^* \in \mathbb{P}^*(\bar{\boldsymbol{\theta}})} \mathrm{supp}(\boldsymbol{p}^*)$, which can be fulfilled by a maximum-entropy WE (MEWE) route choice strategy(Rossi et al., 1989; Bar-Gera and Boyce, 1999). There are specialized algorithms for finding MEWE strategies (e.g., Bar-Gera, 2006; Feng et al., 2024; Li et al., 2024). For (2), we can perform a QR decomposition of the $[\boldsymbol{\Lambda}^\mathsf{T}, \boldsymbol{\Sigma}^\mathsf{T}]^\mathsf{T}$ and take the columns corresponding to the non-zero diagonal elements of the resulting upper triangular matrix. Noting that the size of $[\boldsymbol{\Lambda}^\mathsf{T}, \boldsymbol{\Sigma}^\mathsf{T}]^\mathsf{T}$ is $\mathcal{O}(K_{\max}^2)$, Task (2) has a complexity of $\mathcal{O}(K_{\max}^3)$. Likewise, the complexity of Task (3) is also $\mathcal{O}(K_{\max}^3)$, as the size of the matrix to be inverted in Equation (3.5) is $\mathcal{O}(K_{\max}^2)$. On large regional transportation networks where $|\mathcal{W}|$ is in the order of tens of thousands (if not millions), these tasks, especially (2) and (3), could become highly demanding computationally.

3.1.2 ID based on fixed-point formulation

Alternatively, ID methods could operate on fixed-point formulations of the WE. Dafermos (1988) developed such a method based on the following result (Dafermos, 1980).

Proposition 3.3 (Fixed-point formulation of WE)**.** *For any $r > 0$, letting*

$$h(\boldsymbol{p};\boldsymbol{\theta}) = \arg\min_{\boldsymbol{p}^* \in \mathcal{P}} \|\boldsymbol{p}^* - \boldsymbol{v}\|_2^2, \quad \textit{where } \boldsymbol{v} = \boldsymbol{p} - r \cdot c(\boldsymbol{p};\boldsymbol{\theta}). \tag{3.6}$$

Then, given any $\boldsymbol{\theta} \in \Theta$ and $\boldsymbol{p}^* \in \mathcal{P}$, it holds that $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ if and only if $\boldsymbol{p}^* = h(\boldsymbol{p}^*;\boldsymbol{\theta})$.

This approach, however, is applicable only when $c(\cdot;\boldsymbol{\theta})$ is strictly monotone. When this condition is satisfied, the implicit function $p^*(\boldsymbol{\theta})$ is well-defined and applying ID to the fixed-point equation $p^*(\boldsymbol{\theta}) = h(p^*(\boldsymbol{\theta});\boldsymbol{\theta})$ yields

$$\nabla p^*(\boldsymbol{\theta})^\mathsf{T} = \nabla_{\boldsymbol{\theta}} h(p^*(\boldsymbol{\theta});\boldsymbol{\theta}) \cdot (\boldsymbol{I} - \nabla_{\boldsymbol{p}} h(p^*(\boldsymbol{\theta});\boldsymbol{\theta}))^{-1}. \tag{3.7}$$

To solve the equation, one must first compute $\nabla_{\boldsymbol{p}} h(p^*(\boldsymbol{\theta});\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} h(p^*(\boldsymbol{\theta});\boldsymbol{\theta})$. The problem, however, is that $h(\boldsymbol{p};\boldsymbol{\theta})$ does not have an analytic form. To overcome this challenge, some analytic $h(\boldsymbol{p};\boldsymbol{\theta})$ have been proposed to define WE as a fixed point (see, e.g., Liu et al., 2023). Yet, even though one can compute the two gradients with ease, the approach is still not efficient. This is because the size of the matrix to be inverted in Equation (3.7) is $\mathcal{O}(|K_{\max}|^2)$, which means solving Equation (3.7) has a complexity of $\mathcal{O}(|K_{\max}|^3)$, the same as solving Equation (3.5) in ID approaches based on KKT conditions.

To further avoid the inverting cost, Liu et al. (2023) suggested to approximate the inversion of $\boldsymbol{I} - \nabla_{\boldsymbol{p}} h(p^*(\boldsymbol{\theta});\boldsymbol{\theta})$ with the first T items of its corresponding Neumann series — an approach previously considered by Lorraine et al. (2020) and Liu et al. (2022) under similar scenarios — namely $\sum_{t=0}^{T} (\nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta}))^t$, where $\boldsymbol{p}^* = p^*(\boldsymbol{\theta})$. In doing so, $\nabla l^*(\boldsymbol{\theta})$ is approximated by

$$\nabla_{\boldsymbol{\theta}} l(\boldsymbol{x}^*;\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \sum_{t=0}^{T} \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})^t \cdot \bar{\boldsymbol{\Lambda}}^\mathsf{T} \cdot \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta}), \tag{3.8}$$

where $\boldsymbol{x}^* = x^*(\boldsymbol{\theta})$. In the sequel, we refer to Equation (3.8) as the T-th order approximation of $\nabla l^*(\boldsymbol{\theta})$ based on Neuman series. As this approximation bypasses matrix inversion, the computational cost now becomes less than ID approaches based on KKT conditions. However, to invoke the equation, one still has to first compute $\nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})$, whose size is $\mathcal{O}(|K_{\max}|^2)$ — the storage of this matrix can easily become a bottleneck when $K_{\max}$ is more than tens of thousands.

Remark 3.4 (Extension of fixed-point ID approaches to general cases)**.** In the literature, limited research efforts have been devoted to extending Dafermos (1988)'s approach to the scenario without the strict monotonicity of $c(\cdot;\boldsymbol{\theta})$. In such scenarios, the matrix $\boldsymbol{I} - \nabla_{\boldsymbol{p}} h(p^*;\boldsymbol{\theta})$ will become non-invertible. Recently, however, Liu et al. (2023) have empirically found that in such scenarios, the surrogate given by Equation (3.8)

can provide a nice approximation of $\nabla l^*(\boldsymbol{\theta})$ when T is sufficiently large. This is highly beyond expectation as it holds in such cases that $\lim_{T\to\infty}\sum_{t=0}^{T}\nabla_{\boldsymbol{p}}h(\boldsymbol{p}^*;\boldsymbol{\theta})^t \to \infty$. Yet, the reasons behind this unexpected effectiveness remain unexplored by Liu et al. (2023).

3.2 Automatic differentiation (AD) methods

AD methods attempt to compute $\nabla l^*(\boldsymbol{\theta})$ by unrolling a differentiable algorithm that solves $l^*(\boldsymbol{\theta})$. Practically, such an algorithm may be constructed based on a differentiable function $h : \mathcal{P}\times\Theta\to\mathcal{P}$ with the following property: for all $\boldsymbol{\theta}\in\Theta$, starting from an appropriate $\boldsymbol{p}^0\in\mathcal{P}$, the sequence $\{\boldsymbol{p}^t\}_{t=0}^{\infty}$ iteratively defined by $\boldsymbol{p}^{t+1}=h(\boldsymbol{p}^t;\boldsymbol{\theta})$ converges to $\mathcal{P}^*(\boldsymbol{\theta})$. Typically, $h(\boldsymbol{p};\boldsymbol{\theta})$ can be specified according to a WE algorithm.Li et al. (2022) adopted the imitative logit dynamic (ILD) from evolutionary game theory (Björnerstedt and Weibull, 1994), i.e.,

$$h(\boldsymbol{p};\boldsymbol{\theta})=\boldsymbol{v}/\boldsymbol{\Sigma}^{\mathsf{T}}\boldsymbol{\Sigma}\boldsymbol{v},\quad \text{where } \boldsymbol{v}=\operatorname{diag}(\boldsymbol{p})\cdot\exp(-r\cdot c(\boldsymbol{p};\boldsymbol{\theta})). \tag{3.9}$$

They established the following convergence condition for $\{\boldsymbol{p}^t\}_{t=0}^{\infty}$ defined by the above $h(\cdot;\boldsymbol{\theta})$.

Proposition 3.5 (Convergence conditions for the ILD dynamics)**.** *Suppose that L is the Lipschitz constant of $c(\cdot;\boldsymbol{\theta})$. Given any $\boldsymbol{p}^0\in\mathcal{P}$ such that there exists $\boldsymbol{p}^*\in\mathcal{P}^*(\boldsymbol{\theta})$ with $\operatorname{supp}(\boldsymbol{p}^*)\subseteq\operatorname{supp}(\boldsymbol{p}^0)$, the sequence $\{\boldsymbol{p}^t\}_{t=0}^{\infty}$ then converges to a fixed point in $\mathcal{P}^*(\boldsymbol{\theta})$ as long as $r<1/2L$.*

The overall procedure of AD methods is given in Algorithm 1, which comprises two phases: forward propagation (FP) and backward propagation (BP). The FP phase first iterates $\boldsymbol{p}^{t+1}=h(\boldsymbol{p}^t;\boldsymbol{\theta})$ to solve

Algorithm 1 An AD approach to computing $\nabla l^*(\boldsymbol{\theta})$.

1: **Froward propagation (FP)**: Run Lines 2–6 within an AD tool.
2: **for** $t=0,1,\ldots$ **do**
3: Run $\boldsymbol{p}^{t+1}=h(\boldsymbol{p}^t;\boldsymbol{\theta})$ according to Equation (3.9).
4: If $\delta(\boldsymbol{p}^{t+1};\boldsymbol{\theta})\le\epsilon$, break and set $T=t+1$.
5: **end for**
6: Set $\boldsymbol{x}^T=\bar{\boldsymbol{\Lambda}}\boldsymbol{p}^T$ and $l^T=l(\boldsymbol{x}^T;\boldsymbol{\theta})$.
7: **Backward propagation (BP)**: Invoke the AD tool to compute $\partial l^T/\partial\boldsymbol{\theta}$ and then return $\nabla l^*(\boldsymbol{\theta})=(\partial l^T/\partial\boldsymbol{\theta})^{\mathsf{T}}$.

WE until $\delta(\boldsymbol{p}^T;\boldsymbol{\theta})<\epsilon$, where $\delta:\mathcal{P}\times\Theta\to\mathbb{R}$ is an appropriate WE gap function[1] and ϵ is sufficiently small for $\boldsymbol{p}^T$ to be an exact WE solution in the numerical sense. Then, it computes the planner's objective function value at WE as $l^T=l(x^T;\boldsymbol{\theta})$, where $x^T=\bar{\boldsymbol{\Lambda}}\boldsymbol{p}^T$. By executing the FP phase with an AD tool, e.g., PyTorch (Paszke et al., 2019), a computational graph that stores all intermediate variables and their dependencies can be automatically generated. Subsequently, the AD tool can automatically implement the BP phase, which computes the derivative of the FP's output (i.e. $\partial l^T/\partial\boldsymbol{\theta}$). Eventually, $\partial l^T/\partial\boldsymbol{\theta}$ is accepted as a surrogate for the exact $\nabla l^*(\boldsymbol{\theta})$. As shown by Li et al. (2022), if $\boldsymbol{p}^T$ is a sufficiently accurate WE strategy,

[1] A common choice for the WE gap function is $\delta(\boldsymbol{p};\boldsymbol{z})=-\langle c(\boldsymbol{p};\boldsymbol{z}),\boldsymbol{p}'-\boldsymbol{p}\rangle$, where $\boldsymbol{p}'\in\arg\min_{\boldsymbol{p}''\in\mathcal{P}}\langle c(\boldsymbol{p};\boldsymbol{z}),\boldsymbol{p}''\rangle$.

the difference between $\partial l^T/\partial \boldsymbol{\theta}$ and $\nabla l^*(\boldsymbol{\theta})$ is negligible. On networks of practical sizes, T could number in hundreds or thousands. Even with a large T, however, the AD method can still outperform ID. Li et al. (2022) demonstrated that when $h(\boldsymbol{p};\boldsymbol{\theta})$ is specified by Equation (3.9), both the FP and BP phases of Algorithm 1 require only $\mathcal{O}(T \cdot K_{\max})$ operations. Consequently, for sufficiently large $K_{\max}$, Algorithm 1 is more efficient than ID methods, whose complexity is generally $\mathcal{O}(K_{\max}^3)$.

The limitation of the AD methods is their high memory consumption. For AD tools to implement the BP phase, the entire FP phase, including *all* its intermediate variables, must be stored. For large $K_{\max}$, this requirement can easily overwhelm the memory available to a typical modern personal computer. To reduce memory consumption, a simple heuristic known as truncated AD may be adopted. It stores only the last M ($M \ll T$) iterations of the forward propagation (FP) phase (specifically, from $\boldsymbol{p}^{T-M}$ onward) in the computational graph (Shaban et al., 2019). Consequently, both the time and memory costs are reduced proportionally, though at the expense of incurring large approximation errors. Note that truncated AD essentially unrolls an algorithm for solving WE starting from a warm-started solution already near $\mathcal{P}^*(\boldsymbol{\theta})$. Taking this idea to the extreme yields a "quick-and-dirty" variant that unrolls a one-step iteration from a "fully warmed-up" solution, namely a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ (McKenzie et al., 2024) (see Algorithm 2). As $\boldsymbol{p}^*$ is already a WE, the unrolled one step (Line 2) is indeed a fixed-point iteration. Although this approach reduces both time and memory complexity to $\mathcal{O}(K_{\max})$, it could incur even greater approximation errors.

Algorithm 2 A heuristic algorithm based on AD for approximating $\nabla l^*(\boldsymbol{\theta})$.

1: Solve a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ via any WE algorithm.
2: Run $\boldsymbol{p}^1 = h(\boldsymbol{p}^*;\boldsymbol{\theta})$, $\boldsymbol{x}^1 = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}^1$, and $l^1 = l(\boldsymbol{x}^1;\boldsymbol{\theta})$ within an AD tool.
3: Invoke the AD tool to compute $\partial l^1/\partial \boldsymbol{\theta}$ and then approximate $\nabla l^*(\boldsymbol{\theta})$ by $(\partial l^1/\partial \boldsymbol{\theta})^\mathsf{T}$.

The reader is referred to Liu et al. (2018); Shaban et al. (2019); Li et al. (2023) for other attempts to mitigate the high memory problem of the AD methods. However, to the best of our knowledge, none of these approaches can maintain accurate gradient evaluation while reducing memory usage.

3.3 Summary

Conventional ID methods are inefficient due to costly matrix operations, whereas AD methods are limited by the need to store the entire computational process for solving a sufficiently precise WE. Recent work has attempted to address these challenges. For example, Liu et al. (2023) proposed to accelerate fixed-point ID methods by approximating the matrix inversion with a Neumann series. While this approach significantly reduces computational cost, it still requires storing a large matrix. As another example, McKenzie et al. (2024) reduced the memory consumption of AD methods by unrolling only one fixed-point iteration at WE. This approach may jeopardize the convergence of the descent algorithm because it may not produce sufficiently accurate gradients.

4 A Scalable Algorithm

Having examined the challenge in implementing conventional descent algorithms, we are now ready to develop a highly efficient algorithm for computing the gradient of NDPs, which may be integrated with any descent algorithms. Below, we first provide a general description of the approach.

Idea. Originated from McKenzie et al. (2024)'s heuristic that unrolls only one fixed-point iteration at WE (cf. Algorithm 2), our idea is to unroll up to t fixed-point iterations ($t \geq 1$). To provide a formal description, let us introduce some notation first.

Starting from $h^{(0)}(\boldsymbol{p};\boldsymbol{\theta}) := \boldsymbol{p}$, we define, recursively, $h^{(t+1)}(\boldsymbol{p};\boldsymbol{\theta}) := h(h^{(t)}(\boldsymbol{p};\boldsymbol{\theta});\boldsymbol{\theta})$ ($t = 0, 1, \ldots$). Moreover, let

$$g^{(t)}(\boldsymbol{p};\boldsymbol{\theta}) := l(\boldsymbol{x}^t;\boldsymbol{\theta}), \quad \text{where } \boldsymbol{x}^t = \bar{\Lambda}\boldsymbol{p}^t \quad \text{and} \quad \boldsymbol{p}^t = h^{(t)}(\boldsymbol{p};\boldsymbol{\theta}). \tag{4.1}$$

Now, we can rewrite McKenzie et al. (2024)'s approximation for $\nabla l^*(\boldsymbol{\theta})$ (i.e. the output of Algorithm 2) in our notation as $\nabla_{\boldsymbol{\theta}} g^{(1)}(\boldsymbol{p}^*;\boldsymbol{\theta})$.

We then hypothesize that $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ can provide arbitrarily accurate approximation to $\nabla l^*(\boldsymbol{\theta})$ for sufficiently large t. This hypothesis is counterintuitive at first glance, because $g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ should be constant with respect to t — note that, since $\boldsymbol{p}^*$ is already WE (a fixed point), performing more fixed-point iterations cannot change $\boldsymbol{p}^*$.

Below, we shall show that $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ gradually approaches $\nabla l^*(\boldsymbol{\theta})$ when t increases (Section 4.1), suggesting that $\nabla l^*(\boldsymbol{\theta})$ may be solved by iteratively computing $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ ($t = 0, 1, \ldots$) until convergence. We then develop an algorithm that implements this idea in a scalable manner, both in terms of computing time and memory consumption (Section 4.2). For simplicity, we assume $h(\boldsymbol{p};\boldsymbol{\theta})$ is specified by Equation (3.9). Yet, the algorithm does not depend on any specific form of $h(\boldsymbol{p};\boldsymbol{\theta})$.

4.1 Justification

In this section, we provide a justification that $\nabla_{\theta} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ can provide a high-quality approximation to $\nabla l^*(\boldsymbol{\theta})$ when $t \to \infty$. We assume $\mathcal{P}^*(\boldsymbol{\theta})$ encompasses an SC solution, which guarantees $\nabla l^*(\boldsymbol{\theta})$ to exist per Proposition 3.1.

ID perspective. First, according to the chain rule, it holds

$$\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} l(\boldsymbol{x}^*;\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} h^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \bar{\boldsymbol{\Lambda}}^{\mathsf{T}} \cdot \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta}). \tag{4.2}$$

Then, as $\boldsymbol{p}^* = h(\boldsymbol{p}^*;\boldsymbol{\theta})$, it holds for all $t \geq 0$ that

$$\nabla_{\boldsymbol{\theta}} h^{(t+1)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} h^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta}). \tag{4.3}$$

Recursively applying Equation (4.3) then yields

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} h^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \sum_{t=0}^{1} \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})^t + \nabla_{\boldsymbol{\theta}} h^{(T-2)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})^2 \\ &= \cdots = \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \sum_{t=0}^{T-1} \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})^t.\end{aligned} \tag{4.4}$$

Combining Equations (4.2) and (4.4) then yields

$$\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} l(\boldsymbol{x}^*;\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta}) \cdot \sum_{t=0}^{T-1} \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})^t \cdot \bar{\boldsymbol{\Lambda}}^{\mathsf{T}} \cdot \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta}). \tag{4.5}$$

This closed-form expression of $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ links it to ID. Indeed, comparing Equation (4.5) with the ID approach by Liu et al. (2023) (cf. Equation (3.8)) indicates that $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ coincides with $(T-1)$-th order approximation of $\nabla l^*(\boldsymbol{\theta})$ based on Neumann series. It follows that $\nabla g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \to \nabla l^*(\boldsymbol{\theta})$ provided that $c(\cdot;\boldsymbol{\theta})$ is strongly monotone. However, without strong monotonicity of $c(\cdot;\boldsymbol{\theta})$, the validity of the approximation provided by Neumann series is not theoretically guaranteed (see Remark 3.4), though we have yet to notice performance degradation attributable to lack of strict monotonicity in practice. Thus, we will then proceed to analyze the behavior of $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ through another perspective, which does not rely on its closed-form expression, nor does it require the strong monotonicity of $c(\cdot;\boldsymbol{\theta})$ to be valid.

Algorithmic differentiation perspective: $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ can also be interpreted as differentiating with respect to $\boldsymbol{\theta}$ an algorithm that approximates $l^*(\boldsymbol{\theta})$ when $\boldsymbol{\theta}$ is slightly perturbed.

Note first that whenever $\boldsymbol{\theta}$ is perturbed, $\boldsymbol{p}^*$ is no longer a WE solution. Yet, $h^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ may converge to $\mathcal{P}^*(\boldsymbol{\theta})$ when $T \to \infty$. For this to happen, the following two conditions are required per Proposition 3.5:

(1) $r < 1/2L$, where L is the Lipschitz constant of $c(\cdot;\boldsymbol{\theta})$;

(2) $\mathrm{supp}(\boldsymbol{p}^*)$ covers the routes used by at least one solution from $\mathcal{P}^*(\boldsymbol{\theta})$.

Due to continuous differentiability of $c(\boldsymbol{p};\boldsymbol{\theta})$, it is easy to see Condition (1) holds for all slightly perturbed $\boldsymbol{\theta}$ as long as it holds at the original value. Meanwhile, Condition (2) holds whenever $\mathrm{supp}(\boldsymbol{p}^*)$ covers the routes used by a non-degenerate extreme point of $\mathcal{P}^*(\boldsymbol{\theta})$ (Tobin, 1986; Yang and Huang, 2005). Indeed, for Condition (2) to be violated, $\boldsymbol{p}^*$ has to be a degenerate extreme point that are rare in real transportation networks — a deliberately constructed example of such a pathological case is provided in Remark 4.1.

In summary, under mild conditions $h^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \to \mathcal{P}^*(\boldsymbol{\theta})$, which implies that $g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \to l^*(\boldsymbol{\theta})$, for all slightly perturbed $\boldsymbol{\theta}$. Accordingly, at the original $\boldsymbol{\theta}$, we have

$$\lim_{T\to\infty} \nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \nabla l^*(\boldsymbol{\theta}). \tag{4.6}$$

Strictly speaking, the above argument requires uniform convergence of $g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ to $l^*(\boldsymbol{\theta})$. We have yet to come across or identify any conditions that guarantee the uniform convergence of $g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$. Nor have we encountered any counterexamples. We leave the resolution of this open question to future studies.

The algorithmic differentiation perspective of $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ is closely related to AD methods, which approximates $\nabla l^*(\boldsymbol{\theta})$ by unrolling an algorithm that evaluates $l^*(\boldsymbol{\theta})$. It is straightforward to verify that the approximation given by the AD method of Li et al. (2022) (i.e., the output of Algorithm 1) is $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^0;\boldsymbol{\theta})$, where $\boldsymbol{p}^0$ denotes an arbitrary initialization and T is the number of iterations required to achieve a sufficiently precise WE solution. Thus, $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*\boldsymbol{\theta})$ can be viewed as a variant of that approximation, created by modifying Algorithm 1 to (1) initialize at a WE solution $\boldsymbol{p}^*$ rather than at an arbitrary $\boldsymbol{p}^0$, and (2) run T iterations in the forward pass, even though the initial point is already WE.

Summary. Through two fundamentally different perspectives, our analysis showed $\nabla g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) \to \nabla l^*(\boldsymbol{\theta})$. It suggests that the proposed algorithm blends, in somewhat unexpected ways, the key principles from ID and AD methods. An interesting byproduct of our analysis is the resolution of an open question posed by Liu et al. (2023) regarding the validity of Neumann series-based approximations in the absence of strong monotonicity of $c(\cdot;\boldsymbol{\theta})$. We have shown that the T-th order approximation based on Neumann series can be reinterpreted as unrolling $(T+1)$-step fixed-point iterations at WE.

Remark 4.1 (A case when $\boldsymbol{p}^*$ violates Condition (2))**.** Consider the 3-node-4-link (3N4L) network as shown in Figure 1, in which there is a single OD pair $1 \to 3$ with travel demand $d = 3$. For the four available routes, let us say Route 1 uses Links 1 and 3, Route 2 uses Links 2 and 4, Route 3 uses Links 1 and 4, and Route 4 uses Links 2 and 3. Suppose that $u_1(x_1) = \theta_1 \cdot x_1$, $u_2(x_2) = \theta_2 \cdot x_2$, $u_3(x_3) = \theta_3 \cdot x_3$, and $u_4(x_4) = \theta_4 \cdot x_4$. Let us then consider the case when $\boldsymbol{\theta} = [1,1,1,1]^\mathsf{T}$. It can be checked that there would exist a unique WE

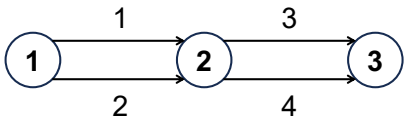


Figure 1: A 3-node-4-link (3N4L) network.

link flow pattern $\boldsymbol{x}^* = [1.5, 1.5, 1.5, 1.5]^\mathsf{T}$, and the set of WE strategies can be written as

$$\mathcal{P}^*(\boldsymbol{\theta}) = \{\boldsymbol{p}^* : \boldsymbol{p}^* = [\lambda, \lambda, 1/2-\lambda, 1/2-\lambda]^\mathsf{T},\ \lambda \in [0, 1/2]\}. \tag{4.7}$$

Setting $\lambda = 0$ then gives rise to a WE strategy $\bar{\boldsymbol{p}}^* = [0, 0, 1/2, 1/2]^\mathsf{T}$, which is a degenerate extreme point of $\mathcal{P}^*(\boldsymbol{\theta})$ (a non-degenerate one should uses 3 routes). In this case, once $\boldsymbol{\theta}$ is perturbed to some $[1+\delta_1, 1+\delta_2, 1, 1]^\mathsf{T}$ where $\delta_1 \neq \delta_2$, there would exist no WE strategy that only uses routes 3 and 4. The example illustrates a pathological case where the convergence of $g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ toward $l^*(\boldsymbol{\theta})$ fails.

4.2 Implementation

To evaluate $\nabla l^*(\boldsymbol{\theta})$, we only need to iteratively compute $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ $(t = 0, 1, \ldots)$ until t reaches a sufficiently large T such that $\|\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*;\boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} g^{(T-1)}(\boldsymbol{p}^*;\boldsymbol{\theta})\|_2$ is smaller than a preset threshold. We now discuss how this can be done efficiently.

Prototype. We start by developing a prototype built on the observation that each $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ can be recursively computed from $\nabla_{\boldsymbol{\theta}} g^{(t-1)}(\boldsymbol{p}^*;\boldsymbol{\theta})$. Specifically, from Equation (4.5) we have

$$\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} g^{(t-1)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \boldsymbol{H}^*_{\boldsymbol{\theta}} \cdot (\boldsymbol{H}^*_{\boldsymbol{p}})^{t-1} \cdot \bar{\boldsymbol{\Lambda}}^{\mathsf{T}} \cdot \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta}), \tag{4.8}$$

where $\boldsymbol{H}^*_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta})$ and $\boldsymbol{H}^*_{\boldsymbol{p}} = \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})$. Thus, one can obtain $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ from $\nabla_{\boldsymbol{\theta}} g^{(t-1)}(\boldsymbol{p}^*;\boldsymbol{\theta}) + \boldsymbol{H}^*_{\boldsymbol{\theta}} \cdot \boldsymbol{a}^t$, where

$$\boldsymbol{a}^t = (\boldsymbol{H}^*_{\boldsymbol{p}})^t \cdot \bar{\boldsymbol{\Lambda}}^{\mathsf{T}} \cdot \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta}).$$

Noting that $\boldsymbol{a}^t = \boldsymbol{H}^*_{\boldsymbol{p}} \cdot \boldsymbol{a}^{t-1}$ leads to Algorithm 3 in which $\dot{\boldsymbol{\theta}}^t \equiv \nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$. It can be easily checked that the time cost of Algorithm 3 is $\mathcal{O}(K^2_{\max})$ per iteration, which is dominated by the cost of computing $\boldsymbol{a}^t = \boldsymbol{H}^*_{\boldsymbol{p}} \cdot \boldsymbol{a}^{t-1}$ (the multiplication between a matrix sized $\mathcal{O}(K^2_{\max})$ and a vector sized $\mathcal{O}(K_{\max})$). Thus, if the algorithm is terminated at $t = T$, the total time cost would be $\mathcal{O}(T \cdot K^2_{\max})$. The memory consumption, however, does not increase with T because each of its iterations only relies on the intermediate variables from the previous iteration. Thus, the total storage required is $\mathcal{O}(K^2_{\max})$, mainly for $\boldsymbol{H}^*_{\boldsymbol{p}}$. Since both time and storage costs still increase with $K_{\max}$ at a quadratic rate, the scalability of Algorithm 3 remains weak.

Algorithm 3 A baseline algorithm for solving $\nabla l^*(\boldsymbol{\theta})$ by iteratively computing $\dot{\boldsymbol{\theta}}^t = \nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$.

1: Solve a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ via any WE algorithm.
2: Set $\boldsymbol{H}^*_{\boldsymbol{p}} = \nabla_{\boldsymbol{p}} h(\boldsymbol{p}^*;\boldsymbol{\theta})$ and $\boldsymbol{H}^*_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} h(\boldsymbol{p}^*;\boldsymbol{\theta})$. Set $\boldsymbol{a}^0 = \bar{\boldsymbol{\Lambda}}^{\mathsf{T}} \cdot \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta})$ and $\dot{\boldsymbol{\theta}}^0 = \nabla_{\boldsymbol{\theta}} l(\boldsymbol{x}^*;\boldsymbol{\theta}) + \boldsymbol{H}^*_{\boldsymbol{\theta}} \cdot \boldsymbol{a}^0$.
3: **for** $t = 1, 2, \ldots$ **do**
4: Set $\boldsymbol{a}^t = \boldsymbol{H}^*_{\boldsymbol{p}} \cdot \boldsymbol{a}^{t-1}$ and then $\dot{\boldsymbol{\theta}}^t = \dot{\boldsymbol{\theta}}^{t-1} + \boldsymbol{H}^*_{\boldsymbol{\theta}} \cdot \boldsymbol{a}^t$. If $\|\boldsymbol{\theta}^t - \boldsymbol{\theta}^{t-1}\|_2 < \delta$, set $T = t$ and return $\nabla l^*(\boldsymbol{\theta}) = \dot{\boldsymbol{\theta}}^T$.
5: **end for**

AD-inspired implementation. To improve scalability, we attempt to redesign Algorithm 3 by incorporating ideas borrowed from AD[2]. The new implementation leverages a key structural property: for any $\boldsymbol{p} \in \mathcal{P}$ and $\boldsymbol{\theta} \in \Theta$, the evaluation of $g^{(t+1)}(\boldsymbol{p};\boldsymbol{\theta})$ can be decomposed into two steps: (1) computing $\boldsymbol{p}' = h(\boldsymbol{p};\boldsymbol{\theta})$, and (2) setting $l = g^{(t)}(\boldsymbol{p}';\boldsymbol{\theta})$. Algorithm 4 describes the two steps, including implementation details for Step (1), which are based on Equation (3.9). Applying the principle of AD to unroll Algorithm 4 yields Algorithm 5, in which a variable with a dot accent represents the gradient of l with respect to the corresponding variable without the accent. Note that the lines in the two algorithms correspond to each other in reverse order. For example, the first line in Algorithm 5 is derived from Algorithm 4's last line. Similarly, Line 2 in Algorithm

[2]The reader is referred to Appendix A for an introduction to AD.

Algorithm 4 Computing $g^{(t+1)}(\boldsymbol{p};\boldsymbol{\theta})$

1: $\boldsymbol{x} = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}$
2: $\boldsymbol{u} = u(\boldsymbol{x};\boldsymbol{\theta})$
3: $\boldsymbol{c} = \boldsymbol{\Lambda}^\mathsf{T}\boldsymbol{u}$
4: $\boldsymbol{e} = \exp(-r\cdot\boldsymbol{c})$
5: $\boldsymbol{v} = \mathrm{diag}(\boldsymbol{e})\boldsymbol{p}$
6: $\boldsymbol{s} = \boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\boldsymbol{v}$
7: $\boldsymbol{p}' = \boldsymbol{v}/\boldsymbol{s}$
8: $l = g^{(t)}(\boldsymbol{p}';\boldsymbol{\theta})$

Algorithm 5 Computing the gradient of $g^{(t+1)}(\boldsymbol{p};\boldsymbol{\theta})$

1: $\dot{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}';\boldsymbol{\theta})$ and $\dot{\boldsymbol{p}}' = \nabla_{\boldsymbol{p}} g^{(t)}(\boldsymbol{p}';\boldsymbol{\theta})$
2: $\dot{\boldsymbol{v}} = \dot{\boldsymbol{p}}'/\boldsymbol{s}$ and $\dot{\boldsymbol{s}} = -\mathrm{diag}(\boldsymbol{v})\dot{\boldsymbol{p}}'/(\boldsymbol{s})^2$
3: $\dot{\boldsymbol{v}} \mathrel{+}= \boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\dot{\boldsymbol{s}}$
4: $\dot{\boldsymbol{p}} = \mathrm{diag}(\boldsymbol{e})\dot{\boldsymbol{v}}$ and $\dot{\boldsymbol{e}} = \mathrm{diag}(\boldsymbol{p})\dot{\boldsymbol{v}}$
5: $\dot{\boldsymbol{c}} = -r\cdot\mathrm{diag}(\boldsymbol{e})\dot{\boldsymbol{e}}$
6: $\dot{\boldsymbol{u}} = \boldsymbol{\Lambda}\dot{\boldsymbol{c}}$
7: $\dot{\boldsymbol{\theta}} \mathrel{+}= \nabla_{\boldsymbol{\theta}} u(\boldsymbol{x};\boldsymbol{\theta})\dot{\boldsymbol{u}}$ and $\dot{\boldsymbol{x}} = \nabla_{\boldsymbol{x}} u(\boldsymbol{x};\boldsymbol{\theta})\dot{\boldsymbol{u}}$
8: $\dot{\boldsymbol{p}} \mathrel{+}= \bar{\boldsymbol{\Lambda}}^\mathsf{T}\dot{\boldsymbol{x}}$

4 is derived from Line 7 in Algorithm 4 based on the chain rule:

$$\dot{\boldsymbol{v}} = (\partial l/\partial \boldsymbol{v})^\mathsf{T} = \left(\partial \boldsymbol{p}'/\partial \boldsymbol{v}\right)^\mathsf{T}\cdot\left(\partial l/\partial \boldsymbol{p}'\right)^\mathsf{T} = \dot{\boldsymbol{p}}'/\boldsymbol{s}; \tag{4.9}$$

$$\dot{\boldsymbol{s}} = (\partial l/\partial \boldsymbol{s})^\mathsf{T} = \left(\partial \boldsymbol{p}'/\partial \boldsymbol{s}\right)^\mathsf{T}\cdot(\partial l/\partial \boldsymbol{s})^\mathsf{T} = -\mathrm{diag}(\boldsymbol{v})\dot{\boldsymbol{p}}'/(\boldsymbol{s})^2. \tag{4.10}$$

Eventually, unrolling the first line of Algorithm 4 corresponds to the last line of Algorithm 5, which returns $\dot{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} g^{(t+1)}(\boldsymbol{p};\boldsymbol{\theta})$ and $\dot{\boldsymbol{p}} = \nabla_{\boldsymbol{p}} g^{(t+1)}(\boldsymbol{p};\boldsymbol{\theta})$ as the final output.

We then proceed to consider the case when $\boldsymbol{p}$ is specified as a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ in Algorithm 4, i.e., $\boldsymbol{p}' = h(\boldsymbol{p};\boldsymbol{\theta}) = \boldsymbol{p}^*$. In this special case, Algorithm 5 computes $\nabla_{\boldsymbol{\theta}} g^{(t+1)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{p}} g^{(t+1)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ with $\nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{p}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ as the inputs. This process gives rise to the following result.

Proposition 4.2. *Given any $\boldsymbol{\theta}\in\Theta$ and $\boldsymbol{p}^*\in\mathcal{P}^*(\boldsymbol{\theta})$, denote $\boldsymbol{x}^* = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}^*$, $\boldsymbol{e}^* = \exp(-r\cdot\boldsymbol{\Lambda}^\mathsf{T}u(\boldsymbol{x}^*;\boldsymbol{\theta}))$, $\boldsymbol{v}^* = \mathrm{diag}(\boldsymbol{e}^*)\boldsymbol{p}^*$, $\boldsymbol{s}^* = \boldsymbol{\Lambda}^\mathsf{T}\boldsymbol{\Lambda}\boldsymbol{v}^*$, $\boldsymbol{U}_{\boldsymbol{x}}^* = \nabla_{\boldsymbol{x}} u(\boldsymbol{x}^*;\boldsymbol{\theta})$, and $\boldsymbol{U}_{\boldsymbol{\theta}}^* = \nabla_{\boldsymbol{\theta}} u(\boldsymbol{x}^*;\boldsymbol{\theta})$. We have*

$$\nabla_{\boldsymbol{p}} g^{(t+1)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \mathrm{diag}(\boldsymbol{e}^*)\dot{\boldsymbol{v}} + \bar{\boldsymbol{\Lambda}}^\mathsf{T}\boldsymbol{U}_{\boldsymbol{x}}^*\cdot\dot{\boldsymbol{u}} \quad \textit{and} \quad \nabla_{\boldsymbol{\theta}} g^{(t+1)}(\boldsymbol{p}^*;\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta}) + \boldsymbol{U}_{\boldsymbol{\theta}}^*\cdot\dot{\boldsymbol{u}}, \tag{4.11}$$

where $\dot{\boldsymbol{v}} = \nabla_{\boldsymbol{p}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})/\boldsymbol{s}^* - \boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\mathrm{diag}(\boldsymbol{v}^*)\cdot\nabla_{\boldsymbol{p}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})/(\boldsymbol{s}^*)^2$ and $\dot{\boldsymbol{u}} = -r\cdot\boldsymbol{\Lambda}\mathrm{diag}(\boldsymbol{e}^*)\mathrm{diag}(\boldsymbol{p}^*)\dot{\boldsymbol{v}}$.

Building on Proposition 4.2, we devise Algorithm 6 to iteratively compute $\dot{\boldsymbol{\theta}}^t = \nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ and $\dot{\boldsymbol{p}}^t = \nabla_{\boldsymbol{p}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$ $(t = 0, 1, \ldots)$. As it turns out, it is much more scalable than Algorithm 3. No operations in

Algorithm 6 An AD-inspired approach to solving $\nabla l^*(\boldsymbol{\theta})$ by iteratively computing $\dot{\boldsymbol{\theta}}^t = \nabla_{\boldsymbol{\theta}} g^{(t)}(\boldsymbol{p}^*;\boldsymbol{\theta})$.

Solve a $\boldsymbol{p}^*\in\mathcal{P}^*(\boldsymbol{\theta})$ via any WE algorithm.
Set $\boldsymbol{x}^* = \bar{\boldsymbol{\Lambda}}\boldsymbol{p}^*$, $\boldsymbol{e}^* = \exp(-r\cdot\boldsymbol{\Lambda}^\mathsf{T}\boldsymbol{u}^*)$, $\boldsymbol{v}^* = \mathrm{diag}(\boldsymbol{e}^*)\boldsymbol{p}^*$, $\boldsymbol{s}^* = \boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\boldsymbol{v}^*$, $\boldsymbol{U}_{\boldsymbol{x}}^* = \nabla_{\boldsymbol{x}} u(\boldsymbol{x}^*;\boldsymbol{\theta})$, $\boldsymbol{U}_{\boldsymbol{\theta}}^* = \nabla_{\boldsymbol{\theta}} u(\boldsymbol{x}^*;\boldsymbol{\theta})$.
Set $\dot{\boldsymbol{x}}^0 = \nabla_{\boldsymbol{x}} l(\boldsymbol{x}^*;\boldsymbol{\theta})$, $\dot{\boldsymbol{p}}^0 = \bar{\boldsymbol{\Lambda}}^\mathsf{T}\dot{\boldsymbol{x}}^0$, and $\dot{\boldsymbol{\theta}}^0 = \nabla_{\boldsymbol{\theta}} l(\boldsymbol{x}^*;\boldsymbol{\theta})$.
for $t = 1, 2, \ldots$ **do**
    Set $\dot{\boldsymbol{v}} = \dot{\boldsymbol{p}}^t/\boldsymbol{s}^* - \boldsymbol{\Sigma}^\mathsf{T}\boldsymbol{\Sigma}\mathrm{diag}(\boldsymbol{v}^*)\cdot\dot{\boldsymbol{p}}^t/(\boldsymbol{s}^*)^2$ and $\dot{\boldsymbol{u}} = -r\cdot\boldsymbol{\Lambda}\mathrm{diag}(\boldsymbol{e}^*)\mathrm{diag}(\boldsymbol{p}^*)\dot{\boldsymbol{v}}$.
    Set $\dot{\boldsymbol{p}}^t = \mathrm{diag}(\boldsymbol{e}^*)\dot{\boldsymbol{v}} + \bar{\boldsymbol{\Lambda}}^\mathsf{T}\boldsymbol{U}_{\boldsymbol{x}}^*\cdot\dot{\boldsymbol{u}}$ and $\dot{\boldsymbol{\theta}}^t = \dot{\boldsymbol{\theta}}^{t-1} + \boldsymbol{U}_{\boldsymbol{\theta}}^*\dot{\boldsymbol{u}}$.
    If $\|\dot{\boldsymbol{\theta}}^t - \dot{\boldsymbol{\theta}}^{t-1}\|_2 \le \delta$, set $T = t$ and return $\nabla l^*(\boldsymbol{\theta}) = \dot{\boldsymbol{\theta}}^T$.
end for

Algorithm 6 has complexity exceeding $\mathcal{O}(K_{\max})$. Specifically, Algorithm 6 only involves (1) element-wise vector operations of size $\mathcal{O}(K_{\max})$, or (2) multiplications between vectors and *sparse* matrices containing $\mathcal{O}(K_{\max})$ non-zero elements. Consequently, if the algorithm terminates at $t = T$, the total time cost would be $\mathcal{O}(T \cdot K_{\max})$. Meanwhile, Algorithm 6 also maintains a constant storage requirement of $\mathcal{O}(K_{\max})$. Thus, Algorithm 6 reduces both time and memory costs by a factor of $K_{\max}$ compared to Algorithm 3.

To close, we wish to note that Algorithm 6 differs from directly invoking an AD tool to compute $\nabla_{\boldsymbol{\theta}} g^{(T)}(\boldsymbol{p}^*; \boldsymbol{\theta})$. With AD, the whole computational process for $g^{(T)}(\boldsymbol{p}^*; \boldsymbol{\theta})$, which involves T fixed-point iterations, must be held in memory. Thus, storage requirement increases with $\mathcal{O}(T \cdot K_{\max})$. In contrast, Algorithm 6 can compute $g^{(T)}(\boldsymbol{p}^*; \boldsymbol{\theta})$ with a constant storage requirement of $\mathcal{O}(K_{\max})$, because it only needs to hold one copy of the fixed-point iterations in memory.

Remark 4.3 (Imprecision of $\boldsymbol{p}^*$)**.** To run Algorithm 6, one must provide a WE solution $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$, which can be obtained by any route-based WE solver. However, as long as this $\boldsymbol{p}^*$ is solved from a numerical procedure, it inevitably deviates from the "true" WE. This issue is not unique to our method; indeed, it arises in almost any algorithm for computing $\nabla l^*(\boldsymbol{\theta})$, including the benchmark by Yang and Bell (2007), since their formula (3.5) also requires a WE solution $\boldsymbol{p}^*$. The right question is not whether this imprecision affects the performance of Algorithm 6, but how much. Of particular interest is whether our algorithm is more susceptible to this problem than the benchmarks. In Section 5.1.1, we shall investigate this numerically.

4.3 Applying the proposed methodology to NDPs

To solve NDP instances, Algorithm 6 must be combined with a suitable descent method to iteratively update $\boldsymbol{\theta}$ using $\nabla l^*(\boldsymbol{\theta})$. Note that the literature has offered a wide range of options tailored to different feasible sets Θ. For instance, when Θ is unconstrained, steepest descent may be used; if Θ is a convex set, projected gradient descent or mirror descent are appropriate alternatives. A convenient option is to use the `optimize.minimize` function from Python's SciPy package. This tool allows users to specify both the objective function and a separate function for evaluating the gradient, and then choose among various first-order methods to solve the instance. In our case, Algorithm 6 can be supplied directly as the gradient function. In numerical experiments conducted in the next section, we adopt this approach, as it leverages existing toolboxes and eliminates the need for extensive hyperparameter tuning.

A final remark is in order here about the special case where $l^*(\boldsymbol{\theta})$ is not differentiable everywhere. As discussed in Section 3, Proposition 3.1 ensures its differentiability of $l^*(\boldsymbol{\theta})$ almost everywhere, which is generally sufficient for the application of first-order methods. However, as noted by Patriksson (2004), if $l^*(\boldsymbol{\theta})$ fails this test at the optimal solution to Problem (3.1), additional care is needed. According toPatriksson (2004), an algorithm can still work if it can return a sub-gradient at non-differentiable points; otherwise, its validity may be in danger. The question that concerns this study is whether Algorithm 6 can provide a sub-gradient of $l^*(\boldsymbol{\theta})$ when the gradient does not exist. Again, we will attempt to explore this issue in our numerical experiments (see Section 5.1.3).

5 Numerical Experiments

To evaluate the effectiveness of the proposed methodology, Section 5.1 examines the performance of Algorithm 6 in computing the gradients of NDPs, and Section 5.2 applies Algorithm 6 to solve a range of NDPs on practical networks. In each experiment, our algorithm is compared against relevant alternative methods from the literature. Throughout all experiments, whenever WE solutions are required, we employ Xie et al. (2018)'s algorithm. All numerical results are produced based on Python on a workstation equipped with an NVIDIA RTX 2080 Ti GPU and an Intel(R) Core(TM) i9-9940X CPU @ 3.30GHz.

5.1 Evaluating NDP's gradients

In this section, we evaluate the performance of our approach in computing gradients for NDPs. The first experiment investigates whether the algorithm can accurately compute the gradient, and how its accuracy is affected by the error in the WE solution provided (Section 5.1.1). The second experiment compares the scalability of the proposed algorithm with several benchmarks as the network size increases (Section 5.1.2). Finally, the third experiment explores whether our scheme can return a sub-gradient at points where the objective function is non-differentiable (Section 5.1.3).

5.1.1 Accuracy test

To test the accuracy of Algorithm 6, we set up a 7-node-12-link network as shown in Figure 2, which has a single OD pair from Nodes 1 to 7, connected by $2^6 = 64$ routes. In the network, the cost function for each link a is set as $u_a(x_a; \theta_a) = \alpha_a + (\beta_a + \theta_a) \cdot x_a^4$, and $l^*(\boldsymbol{\theta})$ is assumed to be total travel time experienced by all travelers at WE. We then examine $\nabla l^*(\boldsymbol{\theta})$ at $\boldsymbol{\theta} = \mathbf{0}$, where the values of all α_a and β_a are set such that $\mathcal{P}^*(\boldsymbol{\theta})$ admits an analytic form. Under this setting, by selecting a $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ that satisfies the SC condition, the exact value of $\nabla l^*(\boldsymbol{\theta})$ can then be computed analytically using Yang and Bell (2007)'s ID method (see Section 3.1). We then conduct two sets of experiments.

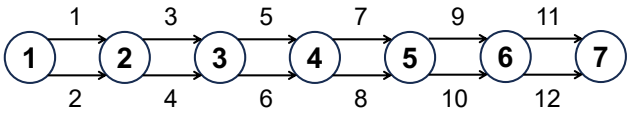


Figure 2: Topology of the 7-node-12-link network.

Firstly, we input Algorithm 6 with different $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$, testing whether all WE solutions can guide the algorithm toward the accurate gradient. In Figure 3, we report $\|\dot{\boldsymbol{\theta}}^t - \nabla l^*(\boldsymbol{\theta})\|_{\max}$ (the approximation error) against t (the number of iterations), where the blue shadow represents the collection of the convergence curves corresponding to all $\boldsymbol{p}^*$, while the red line marks the special case when $\boldsymbol{p}^*$ is the MEWE (see Section 3.1 for its definition). The results demonstrate that all $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$ can steer Algorithm 6 toward the accurate

gradient. However, convergence is fastest when $\boldsymbol{p}^*$ is set at the MEWE. Whether this outcome is coincidental or theoretically justifiable appears worthy of future investigation.

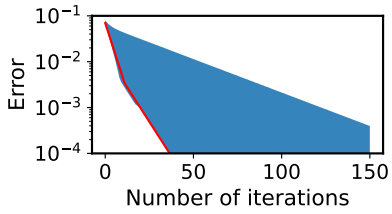


Figure 3: The convergence pattern of Algorithm 6 input with different $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$. The blue area represents the collection of all convergence curves; the red line highlights the curve corresponding to the MEWE.

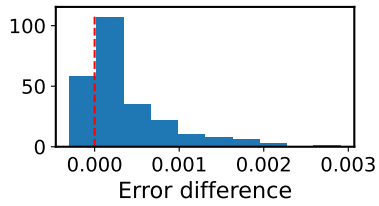


Figure 4: The distribution of the difference between approximation errors of $\nabla l^*(\boldsymbol{\theta})$ returned by Yang and Bell (2007)'s approach and Algorithm 6, input with the $\boldsymbol{p}^*$ added with random noises.

Subsequently, we examine the impact of error in the input $\boldsymbol{p}^* \in \mathcal{P}^*(\boldsymbol{\theta})$, attempting to answer the question raised earlier in Remark 4.3. To this end, we fix $\boldsymbol{p}^*$ as the MEWE and repeatedly run Algorithm 6 with perturbed inputs $\boldsymbol{p}^* + \zeta$, where ζ is random noise satisfying $\|\zeta\|_1 = 0$. In each run, we allow more than 1000 iterations so the algorithm can converge almost exactly, and then record the approximation error of the final output. For comparison, we also feed $\boldsymbol{p}^* + \zeta$ to Yang and Bell (2007)'s, log its approximation error, and calculate the difference between their error and ours (a positive difference means Yang and Bell (2007)'s method incurs a higher error). Figure 4 then shows the distribution of this error difference, where the red dashed line represents zero difference. The results show that the mass is largely distributed to the right of this line, which implies that our method typically achieves lower approximation errors. Thus, the conclusion is that although Algorithm 6 is inevitably influenced by errors in the input WE solution, it appears less vulnerable than the benchmark by Yang and Bell (2007).

5.1.2 Scalability test

We next demonstrate the scalability of Algorithm 6 on networks where the number of OD pairs $|\mathcal{W}|$ is incrementally increased. For easier implementation, we construct these networks by "copying" the 7-node-12-link network N times, thus creating a large network with $|\mathcal{W}| = N$ independent OD pairs. The function $l^*(\boldsymbol{\theta})$ is still set as the total travel at WE. Although this setup is unrealistic, it effectively serves our purpose of evaluating algorithm performance. In the experiment, we tested $|\mathcal{W}|$ from 2^6 to 2^{22}, thus $K_{\max}$ (the number of WE routes in the network) would increase from 2^{16} to 2^{28}.

On each resulting network, we first run Algorithm 6 until its approximation error $\|\dot{\boldsymbol{\theta}}^t - \nabla l^*(\boldsymbol{\theta})\|_\infty$ becomes less than 10^{-6}. Note that because of how these networks are constructed, $\nabla l^*(\boldsymbol{\theta})$ therein is just the $|\mathcal{W}|$-fold concatenation of $\nabla l^*(\boldsymbol{\theta})$ from the previous experiment. We record the time needed to achieve this level of accuracy. We then check whether Algorithm 1 (the AD method proposed by Li et al. (2022)) and Algorithm

3 (essentially the method of Liu et al. (2023); see Section 4.3 for an explanation) can also reach the same accuracy and, if so, how much time they take. Owing to the special structure of our networks, the number of iterations T required for any of the three algorithms to achieve the same accuracy is constant against $|\mathcal{W}|$. Specifically, for Algorithms 6 and 3, $T = 70$; for Algorithm 1, T must be at least 89. The total GPU time required in each case is then reported in Table 1.

Table 1: The total GPU time required by Algorithms 1, 3, and 6 to achieve the same level of accuracy on networks with various sizes.

Network size		GPU time (s)		
$\|\mathcal{W}\|$	$K_{\max}$	Algorithm 6	Algorithm 1	Algorithm 3
2^6	2^{12}	0.11	0.41	12.24
2^8	2^{14}	0.11	0.42	51.79
2^{10}	2^{16}	0.11	0.50	Out of memory
2^{12}	2^{18}	0.11	1.00	
2^{14}	2^{20}	0.11	3.62	
2^{16}	2^{22}	0.35	Out of memory	
2^{18}	2^{24}	1.37		
2^{20}	2^{26}	5.34		
2^{22}	2^{28}	Out of memory		

The result indicates that the time required by Algorithm 6 (our approach) does not even increase when $|\mathcal{W}|$ increases from 2^6 to 2^{14} (thanks to vectorization). Remarkably, even when $|\mathcal{W}| = 2^{20}$ (over one million) and $K_{\max} = 2^{26}$ (around 67 million), the algorithm still completes execution in approximately five seconds. When $|\mathcal{W}|$ tends to 2^{22}, however, Algorithm 6 also becomes infeasible — this is because at this scale, just storing the route-link incidence matrix would suppress our GPU's memory limit. In contrast, Algorithm 1 becomes infeasible once $|\mathcal{W}|$ reaches 2^{16}. At this scale, we observed that PyTorch exhausts memory after roughly 20 iterations. The scalability of Algorithm 3 is even more limited, being feasible only up to $|\mathcal{W}| = 2^8$ and $K_{\max} = 2^{14}$ (around 16000). At this point, the need to store a $2^{14} \times 2^{14}$ matrix nearly exhausts available memory resources. Eventually, we remark that due to the simple structure of the network, T required for a high accuracy is relatively small (less than 100). In more realistic networks, T may be significantly larger, which would further diminish the scalability of standard AD methods. In contrast, our algorithm maintains its scalability as it will not cost more memory by solving the gradient more accurately.

5.1.3 Sub-differentiability test

Eventually, we proceed to investigate the question raised in Section 4.3, namely, whether the proposed approach can return a sub-gradient when $l^*(\boldsymbol{\theta})$ is non-differentiable. Below, we focus on a setting with such non-differentiability, which was constructed by Patriksson (2004) on the 3-node-3-link (3N3L) network as shown in Figure 5. Suppose that there are three OD pairs ($1 \to 2$, $1 \to 3$, and $3 \to 2$) with travel demands $\boldsymbol{d} = [1, 1, 1]^\mathsf{T}$. There are four routes in the network, and let us say Route 1 uses Link 1, Route 2 uses Links

2 and 3, Route 3 uses Link 2, and Route 4 uses Link 3. Let $u_1(x_1) = 2x_1 + \theta_1$, $u_2(x_2) = x_2 + \theta_2$, and $u_3(x_3) = x_3 + \theta_3$. If $\boldsymbol{\theta} = [\theta_1, \theta_2, \theta_2] = [0, 0, 0]$, it can then be checked that there exists a unique WE strategy $\boldsymbol{p}^* = [1, 0, 1, 1]^\mathsf{T}$, and a unique WE link flow pattern $\boldsymbol{x}^* = [1, 1, 1]^\mathsf{T}$. Here, $\boldsymbol{p}^*$ does not satisfy the SC condition: it dictates routes 1 and 2 with identical costs, but route 2 used by nobody. Thus, the condition given by Proposition 3.1 is not met, and hence the differentiability of $x^*(\boldsymbol{\theta})$ and $l^*(\boldsymbol{\theta})$ is not guaranteed.

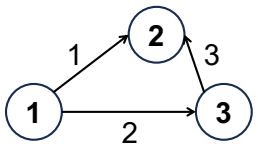
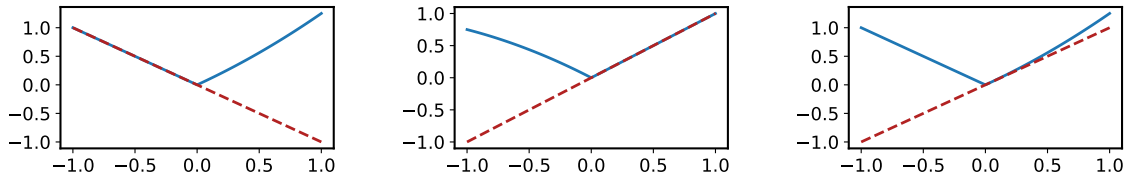


Figure 5: A 3-node-3-link (3N3L) network.

To illustrate the non-differentiability of $l^*(\boldsymbol{\theta})$ at $\boldsymbol{\theta} = \mathbf{0}$, we independently perturb each of θ_1, θ_2, and θ_3 in the range $[-1, 1]$, keeping the other two fixed at zero. Figure 5 plots the resulting changes in $l^*(\boldsymbol{\theta})$ (see the blue solid line), which reveals that $l^*(\boldsymbol{\theta})$ is non-differentiable with respect to all three parameters at $\boldsymbol{\theta} = \mathbf{0}$. We then apply Algorithm 6 in this setting, which returns $[-1, 1, 1]^\mathsf{T}$. These values are highlighted in Figure 6 by the red dashed lines (with slopes -1, -1, and 1, respectively). As shown in the plots, the output of our algorithm corresponds to a sub-gradient of $l^*(\boldsymbol{\theta})$. As noted in Section 4.3, a descent scheme is more robust if it can operate on sub-gradients at points of non-differentiability (Patriksson, 2004). The results here suggest that our approach possesses this capability, at least in this specific example.



(i) Perturb θ_1. **(ii)** Perturb θ_2. **(iii)** Perturb θ_3.

Figure 6: Variation of $l^*(\boldsymbol{\theta})$ as (a) θ_1, (b) θ_2, and (c) θ_3 are individually perturbed (blue solid lines), while the other parameters are fixed at zero; the red dashed lines highlight the "gradients" returned by Algorithm 6 at $\boldsymbol{\theta} = \mathbf{0}$.

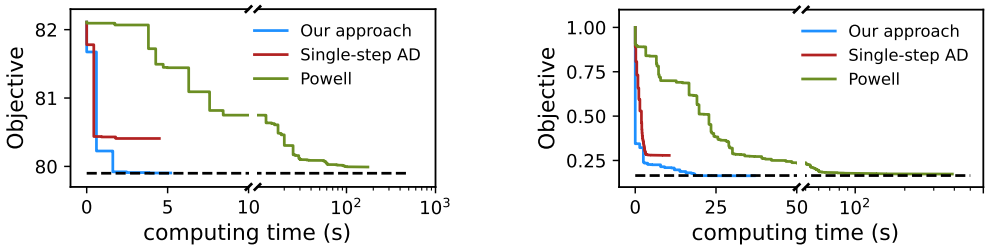
5.2 Solving practical NDPs

Having validated the effectiveness of Algorithm 6 for gradient evaluation, we proceed to apply the approach to several NDPs formulated on practical networks. We first test the proposed method on two benchmark NDPs — one CNDP (Suwansirikul et al., 1987) and the other SCPP (Lawphongpanich and Hearn, 2004) — both set up on the Sioux-Falls network. Next, we construct a PMPP instance on the Chicago-Sketch network (Eash et al., 1979), a much larger network on which most classic NDP algorithms are difficult to apply.

For all tested problems, following the procedure described in Section 4.3, we use the gradient computed by Algorithm 6 as input to the `optimize.minimize` function in SciPy, with the `method` parameter set to `L-BFGS-B` — the limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm with bounds (Byrd et al., 1995). Since the advantages of Algorithm 6 over Algorithms 1 and 3 have already been demonstrated, we do not further test the performance of `L-BFGS-B` using their gradients. Instead, we compare our approach with two alternative methods. The first is the one-step AD approach developed by McKenzie et al. (2024), in which the descent direction is obtained from Algorithm 2, which is computationally scalable but provides only an approximate gradient. The second is Powell's conjugate direction method (or simply Powell's method) — a widely used zeroth-order algorithm for local search, which `optimize.minimize` invokes by default if no gradient information is provided. As such, it also serves as a natural baseline for comparison.

5.2.1 CDNP and SCPP on Sioux-Falls

The Sioux-Falls network consists of 24 nodes, 76 links, and 528 OD pairs. In the CNDP instance from Suwansirikul et al. (1987), 10 links are considered expandable; in the SCPP instance from Lawphongpanich and Hearn (2004), 18 links are identified as tollable. We refer readers to the original papers for more details regarding the experimental settings. In the literature, both instances have been widely used to evaluate the performance of NDP algorithms. To the best of our knowledge, the best-known objective values for the CNDP and SCPP instances are 79.90 and 72.41, respectively (Li et al., 2022; Guo et al., 2025).



(i) The CNDP .

(ii) The SCPP instance.

Figure 7: Objective value versus computation time for our approach, the single-step AD method, and Powell's method on the CNDP and SCPP benchmark problems, both implemented on the Sioux-Falls network; the black dashed line highlights the best-known objective value in the literature.

For each instance, we run our approach, the single-step AD approach, and Powell's method, all starting from the same initial solution ($\boldsymbol{\theta}^0 = 5 \cdot \mathbf{1}$ for CNDP and $\boldsymbol{\theta}^0 = \mathbf{0}$ for SCTP). Figures 7-(a) and -(b) show the convergence trajectories of the three methods on the CNDP and SCPP instances, respectively. As illustrated, the solutions returned by our approach always achieve the best-known objective values on both instances. Compared with alternative approaches, although the single-step AD approach requires less CPU time, it produces noticeably worse objective values, which underscores the importance of accurate gradient computation. Powell's method, on the contrary, achieves better solutions than the single-step heuristic.

However, as it does not utilize gradient information, it converges much more slowly. This result, while expected, further highlights the critical role of first-order information in accelerating descent algorithms.

5.2.2 PMPP on Chicago-Sketch

Finally, we turn to the Chicago-Sketch network (see Figure 8), which consists of 933 nodes, 2,950 links, and 93,512 OD pairs. In Figure 8, all colored links represent highways, with the red segments designated as tollable links for the purpose of setting up the PMPP. It is important to note that this setup does not reflect a real-world scenario, but is constructed purely for experimental purposes. In the PMPP, the parameter γ (the time value of money) is set to 1, and no upper bound is imposed on the tolls for each link.

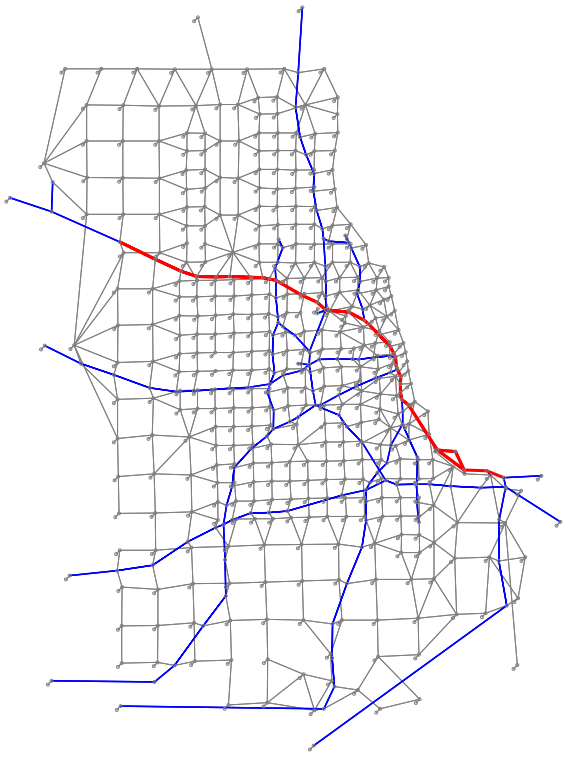
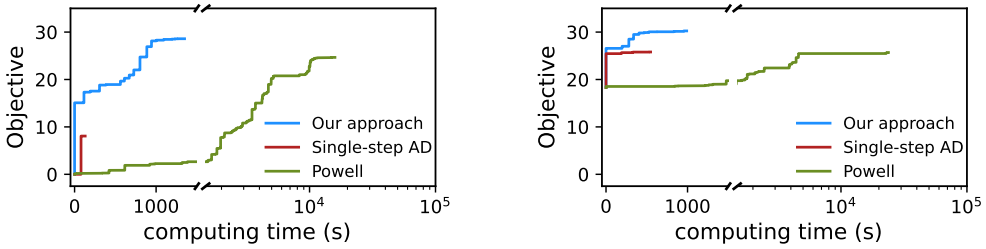


Figure 8: The Chicago-Sketch network (colored links represent highways; red links are designated for tolling).

Like in the last experiment, we test our approach, the single-step AD approach, and Powell's method, each starting from the same initial solution. For this instance, we consider two different initializations: (1) $\boldsymbol{\theta}^0 = \mathbf{0}$, and (2) $\boldsymbol{\theta}^0 = 0.05 \cdot \mathbf{1}$. The convergence trajectories of all three algorithms, starting from these initializations, are shown in Figure 9-(a) and -(b), respectively. In both scenarios, our approach consistently achieves a lower objective value within a shorter or comparable computation time compared to the alternatives. The single-step AD method initially descends rapidly, but quickly stagnates and is unable to reach the lower objective value attained by our approach. Particularly starting from the first initial solution, single-step AD converges to a

solution whose objective value is below 10 while our approach almost attain 30. This again highlights the importance of accurate gradient computation. Powell's method converges extremely slow in this case, and it fails to achieve a solution whose quality is close to ours even after passing around 5 hours. Overall, these findings demonstrate the clear advantage of our method in terms of both efficiency and solution quality.



(i) $\boldsymbol{\theta}^0 = \mathbf{0}$. **(ii)** $\boldsymbol{\theta}^0 = 0.05 \cdot \mathbf{1}$.

Figure 9: Objective value versus computation time for our approach, the single-step AD method, and Powell's method on the PMPP instance implemented on the Chicago-Sketch network, starting from two initial solutions.

6 Conclusion

In this study, we addressed the network design problem (NDP), a fundamental tool for traffic planners seeking to enhance the overall efficiency of transportation networks. Despite extensive research in the literature, many key challenges remain unresolved. In particular, it is widely recognized that even finding a local solution via standard descent methods is difficult, largely due to the computational challenges associated with evaluating NDP gradients. To tackle this longstanding issue, we propose a novel approach that integrates elements of both conventional implicit differentiation (ID) and recent automatic differentiation (AD) methods, which charts a new course that surpasses the limitations of both.

There are several algorithms in literature that make suitable benchmarks against which the performance of our approach (Algorithm 6) can be assessed. The first is the heuristic by McKenzie et al. (2024) (Algorithm 2). If our algorithm terminates after a single iteration, its output coincides with that of the heuristic. Note that the key rationale behind the heuristic was to manage the memory requirements of standard AD methods, at the expense of rough gradient approximations. While, our approach can approximate to any desired accuracy with similar memory usage as the heuristic. Meanwhile, standard AD methods such as that of Li et al. (2022) and our algorithm can, in principle, both reach any desired level of accuracy for computing the gradients of NDPs simply by "iterating more." However, standard AD often struggles to deal with large instances because its memory requirements grow linearly with the number of iterations. By contrast, our algorithm maintains a constant memory — similar to what standard AD needs for a single iteration — thus enabling its applicability to significantly larger instances. Eventually, our algorithm is also related to the approximate ID method proposed by Liu et al. (2023). Specifically, we have found that unrolling T fixed-point iterations at

WE yields $(T-1)$-th order Neumann series approximation of an NDP's gradient. Compared to Liu et al. (2023), our contribution lies mostly in how this surrogate for the gradient can be computed. As Liu et al. (2023) evaluate this surrogate using its closed-form expression, it is at best as efficient as Algorithm 3 (our prototype). For validation, the effectiveness of our approach against these benchmarks has been demonstrated through extensive numerical experiments.

Another notable advantage of the proposed method is its flexibility: it can handle a wide range of NDPs, including continuous network design problems (CNDPs), second-best congestion pricing problems (SCPPs), and profit maximization pricing problems (PMPPs). Indeed, our experimental experience suggests that, when integrated with optimizers in Python's SciPy package, the approach can be directly applied to any problem where the objective depends on the Wardrop equilibrium (WE) state of a transportation network, without requiring any specific adaptation or even hyperparameter tuning. Therefore, we are confident that the method developed in this study is very close to being deployable as an NDP solver distributed as a Python package. Developing and releasing such a package will be the focus of our future work.

A Preliminaries on Automatic Differentiation

Automatic differentiation (AD) is used to numerically evaluate the gradient of functions $f:\mathbb{R}^m\to\mathbb{R}^n$. It can be carried either in a *forward mode* or a *reverse mode*. Which mode is preferable depends on m/n. If the ratio is very small (large), the forward (reverse) model is more efficient and hence preferred. In this paper, we only focus on the reverse mode because f is always a scalar function (i.e., $n=1$) in the applications relevant to our main interest. In reverse-mode AD, derivatives are computed in two phases. In the first — often known as the *forward propagation* (FP) — phase, the function is evaluated, and intermediate variables and their inter-dependencies are stored in a computational graph. In the second — or the *backward propagation* (BP) — phase, the derivatives of all intermediate variables (including the outputs of the function) are evaluated in a reverse order based on the chain rule defined by the inter-dependencies. Below, we explain how it works with an ML example.

Example A.1 (AD for deep learning)**.** *Consider a DNN* $g:\mathbb{R}^n\times\mathbb{R}^{n\times n\times 3}\times\mathbb{R}^{n\times 3}\to\mathbb{R}^n$ *with three fully-connected layers of the following form*

$$g(\boldsymbol{x};\{\boldsymbol{W}^i\}_{i=0}^2,\{\boldsymbol{b}^i\}_{i=0}^2)=f(\boldsymbol{W}^2+f(\boldsymbol{W}^1+f(\boldsymbol{W}^0\boldsymbol{x}+\boldsymbol{b}^0)+\boldsymbol{b}^1)+\boldsymbol{b}^2),\tag{A.1}$$

where the activation function $f:\mathbb{R}^n\to\mathbb{R}^n$ *is an element-wise tanh function. Suppose the DNN is trained with a data pair* $(\boldsymbol{x}^{train},\boldsymbol{y}^{train})$ *to minimize a squared error that reads*

$$l=\frac{1}{2}\cdot\left\|g(\boldsymbol{x}^{train};\{\boldsymbol{W}^i\}_{i=0}^2,\{\boldsymbol{b}^i\}_{i=0}^2)-\boldsymbol{y}^{train}\right\|_2^2.$$

Central to the training process is to compute the gradients of function $l(\{\boldsymbol{W}^i\}_{i=0}^2,\{\boldsymbol{b}\}_{i=0}^2)$ *with respect to* $\boldsymbol{W}^i$ *and* $\boldsymbol{b}^i$*, i.e,* $\overline{\boldsymbol{W}}^i=\partial l/\partial\boldsymbol{W}^i$ *and* $\bar{\boldsymbol{b}}^i=\partial l/\partial\boldsymbol{b}^i$ *(*$i=0,1,2$*). Reverse-mode AD first evaluates function* l

as described in Algorithm 7. Then it calculates the gradients of l in a backward propagation (see Algorithm 8). Note that, in Algorithm 8, any symbol with a bar accent represents a derivative of l with respect to the symbol without the accent. For example, applying the chain rule to line 4 in Algorithm 7 yields, noting $f(\boldsymbol{x}) = \tanh(\boldsymbol{x})$,

$$\bar{\boldsymbol{a}}^i = \operatorname{diag}(\mathbf{1} - (f(\boldsymbol{a}^i))^2) \cdot \frac{\partial l}{\partial \boldsymbol{x}^{i+1}} = \operatorname{diag}(\mathbf{1} - (\boldsymbol{x}^{i+1})^2) \cdot \bar{\boldsymbol{x}}^{i+1},$$

which is precisely line 3 in Algorithm 8. Clearly, AD allows the derivative with respect to any intermediate variable to be evaluated in a single backward pass.

Algorithm 7 FP for deep learning.

1: $\boldsymbol{x}^0 = \boldsymbol{x}^{\text{train}}$
2: **for** $i = 0, 1, 2$ **do**
3: $\quad \boldsymbol{a}^i = \boldsymbol{W}^i \boldsymbol{x}^i + \boldsymbol{b}^i$
4: $\quad \boldsymbol{x}^{i+1} = f(\boldsymbol{a}^i)$
5: **end for**
6: $l = 1/2 \cdot \|\boldsymbol{x}^3 - \boldsymbol{y}^{\text{train}}\|_2^2$

Algorithm 8 BP for deep learning.

1: $\bar{\boldsymbol{x}}^3 = \boldsymbol{x}^3 - \boldsymbol{y}^{\text{train}}$
2: **for** $i = 2, 1, 0$ **do**
3: $\quad \bar{\boldsymbol{a}}^i = \operatorname{diag}(1 - (\boldsymbol{x}^{i+1})^2) \cdot \bar{\boldsymbol{x}}^{i+1}$
4: $\quad \bar{\boldsymbol{W}}^i = \boldsymbol{x}^i \otimes \bar{\boldsymbol{a}}^i$ and $\bar{\boldsymbol{b}}^i = \bar{\boldsymbol{a}}^i$
5: **end for**
6: $\bar{x}^{\text{train}} = \bar{x}^0$

Linnainmaa (1970) is often credited with the first formal description of reverse-mode AD, whereas the first machine-based implementation of reverse-mode AD was proposed by Speelpenning (1980). In a machine-based implementation, one only needs to define the computational graph through FP, and BP is always left to the machine to program automatically. Most modern deep learning frameworks, e.g., TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and JAX (Frostig et al., 2018), are equipped with machine-based AD.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Abdulaal, M. and LeBlanc, L. J. (1979). Continuous equilibrium network design models. *Transportation Research Part B: Methodological*, 13(1):19–32.

Bar-Gera, H. (2006). Primal method for determining the most likely route flows in large road networks. *Transportation Science*, 40(3):269–286.

Bar-Gera, H. and Boyce, D. (1999). Route flow entropy maximization in origin-based traffic assignment. In Ceder, A., editor, *Transportation and traffic theory: Proceedings of the 14th International Symposium on Transportation and Traffic Theory*, pages 397–415. Emerald Group Publishing Limited.

Bard, J. F. and Falk, J. E. (1982). An explicit solution to the multi-level programming problem. *Computers & Operations Research*, 9(1):77–100.

Bertsekas, D. P. and Gafni, E. M. (1982). Projection methods for variational inequalities with application to the traffic assignment problem. In Sorensen, D. C. and Wets, R. J.-B., editors, *Nondifferential and variational techniques in optimization*, pages 139–159. Springer.

Björnerstedt, J. and Weibull, J. W. (1994). Nash equilibrium and evolution by imitation. Technical report, Stockholm University.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208.

Chiou, S.-W. (2005). Bilevel programming for the continuous transport network design problem. *Transportation Research Part B: Methodological*, 39(4):361–383.

Cho, H.-J., Smith, T. E., and Friesz, T. L. (2000). A reduction method for local sensitivity analyses of network equilibrium arc flows. *Transportation Research Part B: Methodological*, 34(1):31–51.

Colson, B., Marcotte, P., and Savard, G. (2007). An overview of bilevel optimization. *Annals of Operations Research*, 153(1):235–256.

Dafermos, S. (1980). Traffic equilibrium and variational inequalities. *Transportation Science*, 14(1):42–54.

Dafermos, S. (1988). Sensitivity analysis in variational inequalities. *Mathematics of Operations Research*, 13(3):421–434.

Eash, R., Chon, K., Lee, Y., and Boyce, D. (1979). Equilibrium traffic assignment on an aggregated highway network for sketch planning. *Transportation Research*, 13:243–257.

Farahani, R. Z., Miandoabchi, E., Szeto, W. Y., and Rashidi, H. (2013). A review of urban transportation network design problems. *European Journal of Operational Research*, 229(2):281–302.

Feng, L., Xie, J., Liu, X., Tang, Y., Wang, D. Z., and Nie, Y. M. (2024). Is order-2 proportionality good enough for approximating the most likely path flow in user equilibrium traffic assignment? *Transportation Research Part B: Methodological*, 186.

Friesz, T. L., Tobin, R. L., Cho, H.-J., and Mehta, N. J. (1990). Sensitivity analysis based heuristic algorithms for mathematical programs with variational inequality constraints. *Mathematical Programming*, 48(1-3):265–284.

Frostig, R., Johnson, M. J., and Leary, C. (2018). Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, pages 23–24.

Gairing, M., Harks, T., and Klimm, M. (2017). Complexity and approximation of the continuous network design problem. *SIAM Journal on Optimization*, 27(3):1554–1582.

Griewank, A. (1989). On automatic differentiation. In Iri, M. and Tanabe, K., editors, *Mathematical programming: Recent developments and applications*, pages 83–108. Springer.

Guo, L., Yin, H., and Zhang, J. (2025). A penalized sequential convex programming approach for continuous network design problems. *INFORMS Journal on Computing*, Ahead of Print.

Hoefer, M., Olbrich, L., and Skopalik, A. (2008). Taxing subnetworks. In Papadimitriou, C. and Zhang, S., editors, *Internet and network economics*, pages 286–294. Springer.

Labbé, M., Marcotte, P., and Savard, G. (1998). A bilevel model of taxation and its application to optimal highway pricing. *Management Science*, 44(12):1608–1622.

Lawphongpanich, S. and Hearn, D. W. (2004). An MPEC approach to second-best toll pricing. *Mathematical Programming*, 101(1):33–55.

Li, J., Wang, Q., Feng, L., Xie, J., and Nie, Y. M. (2024). A day-to-day dynamical approach to the most likely user equilibrium problem. *Transportation Science*, 58(6):1167–1426.

Li, J., Yu, J., Liu, B., Nie, Y., and Wang, Z. (2023). Achieving hierarchy-free approximation for bilevel programs with equilibrium constraints. In *Proceedings of the 40th International Conference on Machine Learning*, pages 20312–20335.

Li, J., Yu, J., Nie, Y. M., and Wang, Z. (2020). End-to-end learning and intervention in games. In *Advances in Neural Information Processing Systems*, volume 33, pages 16653–16665.

Li, J., Yu, J., Wang, Q., Liu, B., Wang, Z., and Nie, Y. M. (2022). Differentiable bilevel programming for Stackelberg congestion games. *arXiv preprint arXiv:2209.07618*.

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7.

Liu, B., Li, J., Yang, Z., Wai, H.-T., Hong, M., Nie, Y., and Wang, Z. (2022). Inducing equilibria via incentives: Simultaneous design-and-play ensures global convergence. *Advances in Neural Information Processing Systems*, 35:29001–29013.

Liu, H., Simonyan, K., and Yang, Y. (2018). DARTS: Differentiable architecture search. In *Proceedings of the 6th International Conference on Learning Representations*.

Liu, R., Gao, J., Zhang, J., Meng, D., and Lin, Z. (2021). Investigating bi-level optimization for learning and vision from a unified perspective: A survey and beyond. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(12):10045–10067.

Liu, Z., Yin, Y., Bai, F., and Grimm, D. K. (2023). End-to-end learning of user equilibrium with implicit neural networks. *Transportation Research Part C: Emerging Technologies*, 150.

Lorraine, J., Vicol, P., and Duvenaud, D. (2020). Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR.

Maclaurin, D., Duvenaud, D., and Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2113–2122. PMLR.

Marcotte, P. (1983). Network optimization with continuous control parameters. *Transportation Science*, 17(2):181–197.

McKenzie, D., Heaton, H., Li, Q., Wu Fung, S., Osher, S., and Yin, W. (2024). Three-operator splitting for learning to predict equilibria in convex games. *SIAM Journal on Mathematics of Data Science*, 6(3):627–648.

Migdalas, A. (1995). Bilevel programming in traffic planning: Models, methods and challenge. *Journal of Global Optimization*, 7(4):381–405.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035.

Patriksson, M. (2004). Sensitivity analysis of traffic equilibria. *Transportation Science*, 38(3):258–281.

Roch, S., Savard, G., and Marcotte, P. (2005). An approximation algorithm for Stackelberg network pricing. *Networks*, 46(1):57–67.

Rossi, T. F., McNeil, S., and Hendrickson, C. (1989). Entropy model for consistent impact-fee assessment. *Journal of Urban Planning and Development*, 115(2):51–63.

Shaban, A., Cheng, C.-A., Hatch, N., and Boots, B. (2019). Truncated back-propagation for bilevel optimization. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, pages 1723–1732. PMLR.

Sheffi, Y. (1985). *Urban transportation networks*. Prentice-Hall.

Speelpenning, B. (1980). *Compiling fast partial derivatives of functions given by algorithms*. PhD thesis, University of Illinois at Urbana-Champaign.

Suwansirikul, C., Friesz, T. L., and Tobin, R. L. (1987). Equilibrium decomposed optimization: A heuristic for the continuous equilibrium network design problem. *Transportation Science*, 21(4):254–263.

Tobin, R. L. (1986). Sensitivity analysis for variational inequalities. *Journal of Optimization Theory and Applications*, 48(1):191–204.

Tobin, R. L. and Friesz, T. L. (1988). Sensitivity analysis for equilibrium network flow. *Transportation Science*, 22(4):242–250.

Verhoef, E. T. (2002). Second-best congestion pricing in general networks. Heuristic algorithms for finding second-best optimal toll levels and toll points. *Transportation Research Part B: Methodological*, 36(8):707–729.

Wardrop, J. G. (1952). Some theoretical aspects of road traffic research. *Proceedings of the Institute of Civil Engineers*, 1(3):325–378.

Xie, J., Nie, Y. M., and Liu, X. (2018). A greedy path-based algorithm for traffic assignment. *Transportation Research Record*, 2672(48):36–44.

Yang, H. and Bell, M. G. H. (1998). Models and algorithms for road network design: A review and some new developments. *Transport Reviews*, 18(3):257–278.

Yang, H. and Bell, M. G. H. (2007). Sensitivity analysis of network traffic equilibrium revisited: The corrected approach. In *Proceedings of the 4th IMA International Conference on Mathematics in Transport*.

Yang, H. and Huang, H.-J. (2005). *Mathematical and economic theory of road pricing*. Emerald Group Publishing Limited.

Yang, H. and Yagar, S. (1994). Traffic assignment and traffic control in general freeway-arterial corridor systems. *Transportation Research Part B: Methodological*, 28(6):463–486.

Zhang, K. and Nie, Y. M. (2018). Mitigating the impact of selfish routing: An optimal-ratio control scheme (ORCS) inspired by autonomous driving. *Transportation Research Part C: Emerging Technologies*, 87:75–90.

Zhang, K. and Nie, Y. M. (2021). To pool or not to pool: Equilibrium, pricing and regulation. *Transportation Research Part B: Methodological*, 151:59–90.