Universität Augsburg

Institut für Mathematik

Ausarbeitung

zum mathematischen Softwareprojekt

Approximation dynamischer Flüsse mit adaptiver Routenwahl in Netzwerken mit mehreren Senken

Abgabe: 11.09.2023

Von: Johannes Hagenmaier

Bei: Prof. Dr. Tobias Harks

Inhaltsverzeichnis

1 Einleitung

Wir betrachten eine spezielle Art von dynamischen Flüssen, bei der Spieler, d.h. infinitesimal kleine Flusspartikel, ihren Weg durch ein vorgegebenes Flussnetzwerk suchen, beginnend bei einem Quellknoten hin zu einem angestrebten Zielknoten, der Senke. Die Besonderheit des hier betrachteten Modells ist dabei, dass Spieler, nach dem Erreichen eines Knotens, diesen unmittelbar über einen momentan kürzesten Weg wieder verlassen. Die Kosten eines solchen Weges sind dabei abhängig von dessen Endpunkt, also von der betrachteten Senke. Enthält ein Netzwerk nur eine einzige Senke und rechtskonstante, endliche Einflussraten, so ist die effiziente Berechnung dieser sog. *IDE-Flüsse* in [1] bereits geklärt.

Sobald in einem Netzwerk mehrere ausgezeichnete Senken existieren, welche wir mit unterschiedlichen Gütern identifizieren wollen, steigt die Komplexität des Problems drastisch: Laut [1] kann zwar die Existenz von IDE-Flüssen weiterhin garantiert werden, die verwendete Methode zur Konstruktion ist jedoch nicht mehr anwendbar. Stattdessen wird in [2], zu finden unter `https://github.com/johanneshage/ide-repository/blob/master/multicom%20IDEs/masterarbeit.pdf`, ein Algorithmus eingeführt, der zumindest Approximationen von IDE-Flüssen berechnen kann. Die Erklärung dieses Algorithmus soll in dieser Arbeit weiter verfeinert werden, wobei insbesondere genauer auf die programmiertechnischen Details eingegangen wird. Der dazu verwendete Quellcode ist geschrieben in der Programmiersprache Python und ist zu finden unter `https://github.com/johanneshage/ide-repository/tree/master/multicom%20IDEs`.

2 Dynamische Flüsse

Gegeben sei ein Digraph $G = (V, E)$, mit Knotenmenge V und Kantenmenge E. Die Menge der Güter bezeichnen wir mit $I := \{1, \dots, k\}$ und für jedes Gut $i \in I$ sei $t_i \in V$ die korrespondierende Senke. Der externe Einfluss in die Quellknoten wird beschrieben über die Einflussratenfunktionen $u_i : S_i \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, wobei $S_i \subseteq V$ die Menge der Quellknoten von Gut i bezeichne. Für einen festen Knoten seien diese Funktionen *rechtskonstant* im zweiten Parameter, d.h. es gelte

$$\forall \theta \in [a, b) : \ \exists\, 0 < \varepsilon < b - \theta : u_i(v, \_)|_{[\theta, \theta+\varepsilon)} \equiv C \in \mathbb{R}. \tag{2.1}$$

Weiter seien für jede Kante $e \in E$ eine *Kantenkapazität* $\nu_e \in \mathbb{R}_+$ und eine *Reiselänge* $\tau_e \in \mathbb{R}_+$ gegeben. Dabei beschreibt ersteres die Menge an Fluss, die Kante e pro Zeiteinheit aufnehmen kann, ohne dass zusätzliche Kosten entstehen und letzteres die Zeitdauer, die ein Flusspartikel benötigt, um e zu überqueren. Die Kosten $c_e(\theta)$ werden initialisiert als τ_e. Übersteigt der Einfluss in die Kante ihre Kapazität, so entsteht eine Warteschlange am Beginn der Kante, die die Kosten weiter erhöht. Dies ist das deterministische Warteschlangenmodell nach Vickrey [3] und wird beispielhaft visualisiert in Abbildung 1: Gegeben eine Einflussrate von 3 in Knoten s für eine

Dauer von 2 Zeiteinheiten, beschrieben durch die Funktion

$$u(s,\_) : \theta \mapsto \begin{cases} 3, & \text{falls } \theta \in [0,2) \\ 0, & \text{sonst} \end{cases},$$

ergibt sich die in der Abbildung dargestellte Situation zum Zeitpunkt $\theta = 2.5$. Aufgrund der ausreichend hohen Kapazität der Kante (s,v) kann sämtlicher Fluss die Kante überqueren, ohne eine Warteschlange zu verursachen. Jedoch ergibt sich ab Zeitpunkt 1 ein Einfluss von 3 in Knoten v, welcher im selben Moment in Kante (v,t) übergeht, deren Kapazität dadurch überschritten wird. Aufgrund der stückweise konstanten Einflussraten ändern sich die Warteschlangen stückweise linear. Es ergibt sich der Aufbau einer Warteschlange mit Rate 2. Zum Zeitpunkt $\theta = 2.5$ hat diese damit eine Länge von 3 erreicht und wird bis zum Zeitpunkt 3 weiter aufgebaut bis eine Länge von 4 erreicht ist, der Einfluss in v stoppt und die Warteschlange von (v,t) mit Rate 1 abgebaut wird. Zum Zeitpunkt $\theta = 7$ ist diese vollständig abgebaut, womit sämtlicher Fluss bis $\theta = 8$ die Senke t erreicht.

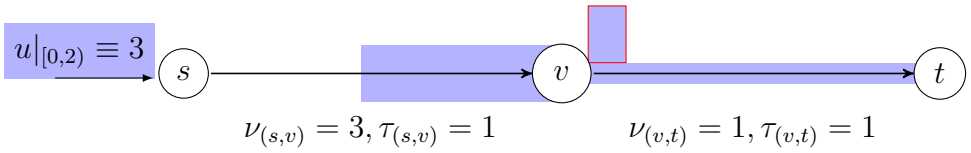


Abbildung 1: Aufbau einer Warteschlange, zum Zeitpunkt $\theta = 2.5$

Insgesamt wird das Tupel $(G, I, \nu, \tau, (u_i)_{i\in I}, (t_i)_{i\in I})$ bezeichnet als *Flussnetzwerk*. In den beiden folgenden Abschnitten werden alle bereits erwähnten Begriffe im Umgang mit dynamischen Flüssen formal definiert.

2.1 Das Modell

Gegeben sei ein Flussnetzwerk $(G, I, \nu, \tau, (u_i)_{i\in I}, (t_i)_{i\in I})$. Wir definieren:

Definition 2.1. Ein *dynamischer Fluss* ist ein Tupel $f = (f^+, f^-)$, wobei f^+, $f^- : I \times E \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ lokal lebesgueintegrierbare Funktionen sind, welche den *Ein-*, bzw. *Abfluss* $f^+_{i,e}(\theta)$, $f^-_{i,e}(\theta)$ eines Guts i zum Zeitpunkt θ in, bzw. aus Kante e beschreiben. Den *Gesamtfluss* in, bzw. aus einer Kante bezeichnen wir mit $f^+_e(\theta) \coloneqq \sum_{i\in I} f^+_{i,e}(\theta)$, bzw. $f^-_e(\theta) \coloneqq \sum_{i\in I} f^-_{i,e}(\theta)$.

Auch alle Ein- und Abflussfunktionen $f^+_{i,e}$ und $f^-_{i,e}$ sind rechtskonstant, wenn alle Einflussraten $u_i(v,\_)$ als rechtskonstant gegeben sind.

Definition 2.2. Der *Zu-*, bzw. *Abfluss bis zum Zeitpunkt* $\theta \in \mathbb{R}_{\geq 0}$ eines Guts $i \in I$ von Kante $e \in E$ wird bezeichnet mit $F^+_{i,e}(\theta)$, bzw. $F^-_{i,e}(\theta)$ und ist definiert als

$$F^+_{i,e}(\theta) := \int_0^\theta f^+_{i,e}(\zeta)\, d\zeta, \quad F^-_{i,e}(\theta) := \int_0^\theta f^-_{i,e}(\zeta)\, d\zeta. \tag{2.2}$$

Der *Gesamtzu-*, bzw. *abfluss* F^+_e und F^-_e von Kante e *bis zum Zeitpunkt* θ ist definiert als

$$F^+_e(\theta) := \sum_{i \in I} F^+_{i,e}(\theta), \quad F^-_e(\theta) := \sum_{i \in I} F^-_{i,e}(\theta). \tag{2.3}$$

Damit kann die Länge der Warteschlange von Kante e in Abhängigkeit von θ gewählt werden als

$$q_e(\theta) := F^+_e(\theta) - F^-_e(\theta + \tau_e). \tag{2.4}$$

Definition 2.3. Ein dynamischer Fluss f heißt *zulässig*, falls folgende Bedingungen erfüllt sind:

1. Für die Senken t_i gilt

$$\sum_{e \in \delta^-_{t_i}} f^-_{i,e}(\theta) - \sum_{e \in \delta^+_{t_i}} f^+_{i,e}(\theta) \geq 0 \tag{2.5}$$

2. Alle Knoten $v \neq t_i$ erfüllen die Flusserhalungsgleichung

$$\sum_{e \in \delta^+_v} f^+_{i,e}(\theta) - \sum_{e \in \delta^-_v} f^-_{i,e}(\theta) = \begin{cases} u_j(\theta), & \text{falls } v = s_j \text{ für einen Quellkonten } s_j \\ 0, & \text{sonst} \end{cases} \tag{2.6}$$

3. Die Warteschlangen werden schnellstmöglich abgebaut, d.h. für alle Kanten $e \in E$ gilt

$$f^-_e(\theta + \tau_e) = \begin{cases} \nu_e, & \text{falls } q_e(\theta) > 0 \\ \min\{f^+_e(\theta), \nu_e\}, & \text{falls } q_e(\theta) = 0 \end{cases} \tag{2.7}$$

4. Beim Durchlaufen von Warteschlangen und der Überquerung von Kanten genügt der Fluss dem FIFO-Prinzip, d.h. Güter Verlassen die Kanten in dem gleichen Verhältnis, in dem sie diese betreten haben:

$$f^-_{i,e}(\theta) = \begin{cases} f^-_e(\theta) \cdot \frac{f^+_{i,e}(\vartheta)}{f^+_e(\vartheta)}, & \text{falls } f^+_e(\vartheta) > 0 \\ 0, & \text{sonst,} \end{cases} \tag{2.8}$$

wobei $\vartheta := \min\{\vartheta \leq \theta | \vartheta + \tau_e + \frac{q_e(\vartheta)}{\nu_e} = 0\}$.

Bemerkung 2.4. Gleichung (2.7) impliziert in der Tat einen schnellstmöglichen (unter Beachtung der Kantenkapazitäten) Abbau der Warteschlangen: Zusammen mit $q'_e(\theta) = \sum_{i\in I} f^+_{i,e}(\theta) - \sum_{i\in I} f^-_{i,e}(\theta + \tau_e)$ ergibt sich

$$q'_e(\theta) = \begin{cases} f^+_e(\theta) - \nu_e, & \text{falls } q_e(\theta) > 0 \\ \max\{0, f^+_e(\theta) - \nu_e\}, & \text{falls } q_e(\theta) = 0 \end{cases}, \text{ für alle } e \in E, \theta \in \mathbb{R}_{\geq 0}. \quad (2.9)$$

2.2 Kürzeste Wege und IDE-Flüsse

Wir wollen nun eine bestimmte Variante von dynamischen Flüssen betrachten, bei der Güter in jedem Knoten immer nur entlang Kanten, die auf einem momentan kürzesten Weg zu ihrer Senke liegen, geschickt werden. Dynamische Flüsse, welche diese Eigenschaft erfüllen, werden als *IDE-Flüsse* (engl.: *Instantaneous Dynamic Equilibrium*) bezeichnet und bilden den Hauptgegenstand dieser Arbeit. Für die genaue Definition müssen wir jedoch zuerst erklären, wie die Kosten eines Weges aufzufassen sind:

Die Kantenkosten einer Kante $e \in E$ werden beschrieben durch die Funktion

$$c_e : \mathbb{R}_{\geq 0} \to \mathbb{R},\ \theta \mapsto \tau_e + \frac{q_e(\theta)}{\nu_e}, \quad (2.10)$$

wobei der letzte Teil auch als *Wartezeit* bezeichnet wird und für jedes $e \in E$ mit der Funktion

$$w_e : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0},\ \theta \mapsto \frac{q_e(\theta)}{\nu_e} \quad (2.11)$$

beschrieben wird. Ist dann $v \in V$ ein Knoten, $i \in I$ ein Gut und P ein v, t_i-Pfad, so berechnen sich die Kosten von P durch

$$c_P(\theta) \coloneqq \sum_{e\in P} c_e(\theta). \quad (2.12)$$

Weiter lassen sich mit Hilfe dieser Kosten Distanzen von Knoten $v \in V$ zu den Senken $t_i \in V$ definieren. Diese beschreiben wir mit den zeitabhängigen Knotenlabelfunktionen ℓ^i_v, gegeben durch:

$$\ell^i_v(\theta) \coloneqq \begin{cases} 0, & \text{falls } v = t_i \\ \min_{e=vw\in E}\{\ell^i_w(\theta) + c_e(\theta)\}, & \text{falls } t_i \text{ von } v \neq t_i \text{ in G erreichbar} \\ \infty, & \text{falls } t_i \text{ von } v \text{ aus nicht erreichbar,} \end{cases} \quad (2.13)$$

für alle $\theta \in \mathbb{R}_{\geq 0}$.

Dann heißt eine Kante $e = vw$ *aktiv* für Gut i zum Zeitpunkt θ, wenn gilt:

$$\ell^i_v(\theta) = \ell^i_w(\theta) + c_e(\theta) < \infty \quad (2.14)$$

und die Menge der für Gut i aktiven Kanten zum Zeitpunkt θ wird bezeichnet mit $E^i_\theta \subseteq E$. Im Fall, dass nur ein Gut gegeben ist, bezeichnen wir die Knotenlabels auch mit ℓ_v und die Menge der aktiven Kanten mit E_θ. Weiter sei $\delta^+_{i,v}(\theta) \coloneqq \delta^+_v \cap E^i_\theta$ die Menge der von $v \in V$ ausgehenden, momentan für Gut $i \in I$ aktiven, Kanten.

Definition 2.5. Ein zulässiger dynamischer Fluss f ist ein *IDE-Fluss*, wenn für alle $i \in I$, $\theta \in \mathbb{R}_{\geq 0}$, $e \in E$ gilt:

$$f^+_{i,e}(\theta) > 0 \Rightarrow e \in E^i_\theta. \tag{2.15}$$

In einem IDE-Fluss betreten Spieler also nur für das jeweilige Gut aktive Kanten, entscheiden sich also in jedem Knoten nur für Kanten, die auf momentan kürzesten Wegen liegen.

Über die Existenz von IDE-Flüssen unter den hier gegebenen Voraussetzungen wurden bereits in [1] entscheidende Resultate bewiesen: Für eine zulässige Eingabeinstanz kann die Existenz von IDE-Flüssen immer garantiert werden. Letztere sind jedoch nicht immer eindeutig und im Fall, dass $|I| > 1$ gilt, ist auch die endliche Termination nicht garantiert, selbst wenn die Einflussraten in Flussmenge und Einflussdauer endlich sind.

Des Weiteren wird eine Idee zur Konstruktion von IDE-Flüssen offenbart: Gegeben ein IDE-Fluss bis zum Zeitpunkt θ_k, werden eine Flussaufteilung und eine Schrittweite $\alpha_k > 0$ gesucht, mit denen der Fluss auf einen größeren Zeitraum erweitert werden kann und gleichzeitig die Eigenschaften eines IDE-Flusses erhalten bleiben.

Um einen IDE-Fluss bis zum Zeitpunkt θ_k erweitern zu können, benötigen wir den Begriff von *feinen* IDE-Flüssen (aus dem Englischen: IDE thin flows), welche im Anschluss definiert werden, zunächst benötigen wir jedoch noch eine Notation für den aktuellen Einfluss in einen Knoten: Zu einem dynamischen Fluss f bezeichnet der Wert

$$b^-_{i,v}(\theta) \coloneqq \sum_{e \in \delta^-_v} f^-_{i,e}(\theta) + u_i(v, \theta) \tag{2.16}$$

den aktuellen *Einfluss* von Gut $i \in I$ in Knoten $v \in V$ zum Zeitpunkt θ. Damit können wir definieren:

Definition 2.6. Für einen gegebenen IDE-Fluss bis zum Zeitpunkt θ_k ist ein Tupel $(x, a) \in \mathbb{R}_{\geq 0}^{I \times E} \times \mathbb{R}^{I \times V}$ ein *feiner IDE-Fluss*, wenn gilt:

$$\sum_{e \in \delta_v^+} x_{i,e} = b_{i,v}^-(\theta_k) \qquad \text{f.a. } i \in I, v \in V \setminus \{t_i\}, \tag{2.17}$$

$$x_{i,e} = 0 \qquad \text{f.a. } i \in I, e \in E \setminus E_{\theta_k}^i, \tag{2.18}$$

$$a_{i,t_i} = 0 \qquad \text{f.a. } i \in I, \tag{2.19}$$

$$a_{i,v} = 0 \qquad \text{f.a. } i \in I, v \in V, \text{ mit } \ell_v^i(\theta_k) = \infty, \tag{2.20}$$

$$a_{i,v} = \min_{e=(v,w) \in E_{\theta_k}^i} \frac{g_e(\sum_{j \in I} x_{j,e})}{\nu_e} + a_{i,w} \quad \text{f.a. } i \in I, v \in V \setminus \{t_i\}, \text{ mit } \ell_v^i(\theta_k) < \infty, \tag{2.21}$$

$$a_{i,v} = \frac{g_e(\sum_{j \in I} x_{j,e})}{\nu_e} + a_{i,w} \qquad \text{f.a. } i \in I, e = (v, w) \in E_{\theta_k}^i \text{ mit } x_{i,e} > 0, \tag{2.22}$$

wobei

$$g_e(x_e) \coloneqq \begin{cases} x_e - \nu_e, & \text{falls } q_e(\theta_k) > 0 \\ \max\{x_e - \nu_e,\ 0\}, & \text{falls } q_e(\theta_k) = 0, \end{cases} \tag{2.23}$$

die aktuelle Änderungsrate der Warteschlange von Kante e bei Gesamtzufluss x_e beschreibt. Wir nennen dann auch x die *Flussaufteilung* und a die *Steigung* zum Zeitpunkt θ_k.

3 Programmaufbau

Wir beginnen mit einer groben Zusammenfassung der Funktionsweise des in [2] vorgestellten Algorithmus, wobei für eine genaue Beschreibung auf ebendiese Quelle verwiesen wird. Die Herangehensweise zur Konstruktion von IDE-Flüssen orientiert sich dabei an der in [1] vorgestellten: Gegeben sei ein IDE-Fluss bis zu einem Zeitpunkt θ_k. Zu diesem sollen ein feiner IDE-Fluss und eine Schrittweite $\alpha > 0$ bestimmt werden, sodass der Fluss bis zu dem neuen Zeitpunkt $\theta_{k+1} := \theta_k + \alpha$ *erweitert* werden kann und dabei ein IDE-Fluss bis zum Zeitpunkt θ_{k+1} entsteht. Die Wahl der Schrittweite ist dabei nicht willkürlich, sondern so, dass folgende Bedingungen gewährleistet sind:

1. Warteschlangen bleiben nicht-negativ, d.h. der Abbau wird beschränkt durch:

$$q_e(\theta_k) + \alpha \cdot \left(\sum_{j \in I} x_{j,e} - \nu_e \right) \geq 0, \quad \text{für alle } e \in E \text{ mit } q_e(\theta_k) > 0 \tag{3.1}$$

2. Das Aktivwerden einer inaktiven Kante beendet die Phase, es gilt also für alle $i \in I$ und $e = (v, w) \in E \setminus E^i_{\theta_k}$:

$$\ell^i_v(\theta_k) + \alpha \cdot a_{i,v} \leq \tau_e + \frac{q_e(\theta_k)}{\nu_e} + \alpha \cdot \frac{g_e \left(\sum_{j \in I} x_{j,e} \right)}{\nu_e} + \ell^i_w(\theta_k) + \alpha \cdot a_{i,w} \tag{3.2}$$

3. Die aktuellen Einflusswerte aller Knoten bleiben die gesamte Phase über konstant, d.h.:

$$b^-_{i,v}(\theta_k + \xi) = b^-_{i,v}(\theta_k), \quad \text{für alle } i \in I, v \in V \setminus \{t_i\}, \xi \in [0, \alpha) \tag{3.3}$$

Für ein $\alpha > 0$, welches diese Bedingungen erfüllt, bezeichnen wir das Intervall $[\theta_k, \theta_k + \alpha)$ als *Phase*.

Mit dieser Methode lässt sich die Existenz von IDE-Flüssen für jede zulässige Eingabeinstanz beweisen; siehe dazu [1] oder [2].

Der nachstehende Algorithmus 1 ist entnommen aus [2] und zeigt den Aufbau des Algorithmus.

Algorithmus 1: `main()`

1 **Eingabe:** Netzwerk $(G, I, \nu, \tau, (u_i)_{i\in I}), (t_i)_{i\in I})$, $T \in \mathbb{R}_+$.
2 **Ausgabe:** Approximation f eines IDE-Flusses bis zum Zeitpunkt T.
3 Methode: Initialisiere $f = (f^+, f^-)$ als den Nullfluss, $q(0) = 0$, $c_e(0) = \tau_e$, für alle $e \in E$, und die Labels $\ell_v^i(0)$, für alle $i \in I$, $v \in V$ mit dem Algorithmus von Dijkstra, sowie die korrespondierenden Mengen E_0^i der aktiven Kanten, für alle $i \in I$;
4 $\theta \leftarrow 0$;
5 **while** $\theta < T$ **do**
6 Prüfe, ob vorige Flussaufteilung x, a noch gültig und verwende diese gegebenenfalls (nicht im Fall $\theta = 0$) ;
7 Sonst x, $a \leftarrow$ `fp_approx(`θ`)`;
8 Aktualisiere f^+, f^- und bestimme Schrittweite α ;
9 Setze $q_e(\theta + \alpha) \leftarrow q_e(\theta) + g_e(f_e^+(\theta)) \cdot \alpha$, für alle Kanten $e \in E$;
10 Setze $c_e(\theta + \alpha) \leftarrow \frac{q_e(\theta+\alpha)}{\nu_e} + \tau_e$, für alle Kanten $e \in E$;
11 Setze $\ell_v^i(\theta + \alpha) \leftarrow \ell_v^i(\theta) + \alpha \cdot a_{i,v}$, für alle $i \in I$, $v \in V$;
12 Bestimme alle $E_{\theta+\alpha}^i$;
13 Überarbeite Distanzwerte mit `refine()`;
14 $\theta \leftarrow \theta + \alpha$;
15 **end while**
16 **return:** $f = (f^+, f^-)$;

Die Hauptschleife (Zeilen **5** - **15**) führt in jeder Iteration folgende Schritte aus:

1. **Bestimmung Flussaufteilung:** Zu diesem Zweck gibt es zwei Möglichkeiten: Entweder die Flussaufteilung der vorherigen Iteration ist immer noch im Sinne eines IDE-Flusses und kann weiterhin verwendet werden, oder es muss eine neue Flussaufteilung mittels `fp_approx(`θ`)` berechnet werden. Letzteres wird wiederum unterteilt in folgende Schritte:

   1.1 Alle Flusswerte $x_{i,e}$ werden mittels unterer und oberer Schranken abgeschätzt, welche im Verlauf des Algorithmus iterativ verfeinert werden. Abweichend von der Notation in [2], werden diese Schranken hier nicht mit $(lb_{i,e})$ und $(ub_{i,e})$ bezeichnet, sondern es wird die Variable

   ```
   bounds: dict[str, dict[str, dict[str, tuple[float, float]]]]
           # Gut, Startknoten, Kante, untere Schranke, obere Schranke
   ```

   Verwaltung der Schranken an die Flusswerte

   verwendet.

   1.2 `calc_flow_by_bounds()`: berechnet eine zulässige Flussaufteilung, die alle Schranken in `bounds` respektiert.

   1.3 `calc_a(`x`)`: berechnet die zur Flussaufteilung x korrespondierende Steigung a.

   1.4 `relax_bounds()`: findet fehlerhafte Schranken und relaxiert diese.

1.5 `fix_nodes()`: findet Knoten, für die alle Flusswerte im Sinne eines IDE-Flusses sind, und setzt diese Werte endgültig fest.

1.6 `fix_fp_comp()`: Für Paare $(i, v) \in I \times V$, bei denen alle Schranken die geforderte Genauigkeit erreicht haben, werden die zugehörigen Flusswerte festgesetzt.

2. **Schrittweite:** Gibt die Länge an, die die aktuelle Phase andauern kann, bis eine neue Phase beginnen muss. Basiert auf der in [1] vorgestellten Methode zur Erweiterung von IDE-Flüssen. Da hier mit approximierten Flusswerten gearbeitet wird, muss dies auch bei der Wahl der Schrittweite beachtet werden.

3. **Aktualisierung der Daten:** Aktualisiert werden q, c und ℓ in Abhängigkeit der bestimmten Flusswerte f^+ und Schrittweite α. Damit werden die Mengen der aktiven Kanten aktualisiert, wobei auch hierfür ein Toleranzbereich gelassen wird, um Unterschiede in der Größenordnung der Approximationsfehler zu vernachlässigen. Im Anschluss werden die Werte q, c und ℓ nochmals mittels `refine()` verfeinert, indem diese an die neuen Mengen der aktiven Kanten angepasst werden, um zukünftige Approximationen zu verbessern.

Insgesamt berechnet Algorithmus 1 einen IDE-Fluss der bereits vor T terminiert, oder andernfalls einen IDE-Fluss bis zum Zeitpunkt T.

Die zur Ausführung des Algorithmus notwendigen Kenntnisse werden in den folgenden Abschnitten vermittelt.

3.1 Eingabe

Für die Ausführung unerlässlich sind die Eingabe des Graphs G, sowie die Einflussratenfunktion u. Diese werden in der Datei data.py nach folgendem Muster angegeben:

```
1 graph: dict[str, dict[str, tuple[float, float]]]
2         # Startknoten, Endknoten, Kapazität, Reiselänge
3
4 u: list[list[list[tuple[float, float]]]]
5         # Gut, Knoten, Einflussstartzeit, Einflussrate
```

Codeabschnitt 1: data.py

Dabei ist zu beachten: die Reihenfolge in der die Knoten in `graph` aufgeführt sind, ist die Reihenfolge in der die Knoten intern gespeichert werden. In genau dieser Reihenfolge müssen auch die Einflussraten aller Güter in `u` spezifiziert werden. Für jedes Gut $i \in I$ enthält `u`$[i]$ eine (potentiell leere) Liste für jeden Knoten, die die Einflussraten für den jeweiligen Knoten enthält.

Damit ist die Eingabe der Instanz in data.py vollständig. Diese wird später in das Hauptprogramm in ContAppMulti.py eingespeist. Letzteres enthält u.a. die folgenden Parameter:

```python
class ContAppMulti:
  """Vom Benutzer vorgegeben"""
  bd_tol: float  # Approximationsgenauigkeit
  T: float  # Zeithorizont

  """interne Parameter, die letztendlich die Ausgabe bilden"""
  fp: list[list[list[tuple[float, float]]]]  # Zufluss
  fm: list[list[list[tuple[float, float]]]]  # Abfluss
  q_global: list[list[float]]  # alle Warteschlangenlängen
  q_ind: list[list[float]]  # zugehörige Zeitpunkte
  global_phase: list[float]  # alle Phasen

  """Einige weitere ausgewählte, interne Parameter"""
  E_active: lil_matrix[|I|, |E|]  # aktuell aktive Kanten
  b: lil_matrix[|I|, |V|]  # Einflüsse
  labels: list[list[float]]  # Knotenlabels
  q: list[float]  # aktuelle Warteschlangenlängen
  c: list[float]  # Kantenkosten
  x_total: lil_matrix[|I|, |E|]  # aktuelle Flusswerte
  x_sum: list[float]  # entsprechende Gesamtflusswerte
  a: lil_matrix[|I|, |V|]  # Steigung
  argmin: list[dict[Node, list[Edge]]]  # für jedes Gut: enthält für
    jeden Knoten Liste aller ausgehenden Kanten, die momentan
   günstigsten a - Wert erzeugen
  init_coms: dict[Node, list[I]]  # für relevante Knoten: Liste der
    Güter, die sich momentan in diesem Knoten befinden
  coms_lens: list[int]  # coms_lens[v_ind] entspricht Länge von
   init_coms[v_ind]
  top_ords: list[list[str]]  # für jedes Gut: aktuelle,
   topologische Sortierung auf Teilgraph der aktiven Kanten
```

Codeabschnitt 2: Essentielle Parameter aus ContAppMulti.py

Der Parameter `bd_tol` beschreibt dabei die Genauigkeit, mit der die Flusswerte approximiert werden sollen und bildet somit das Analogon zur Variable ϵ aus [2]. Mit `fp` und `fm` wird der Fluss beschrieben, wobei die Liste `fp`$[i][e]$ alle Änderungen von $f^+_{i,e}$ als 2-Tupel der Form $(\theta, f^+_{i,e}(\theta))$ beinhaltet. Beispielsweise geschieht folgende Interpretation:

$$\texttt{fp}[i][e] = [(0,2),(0.5,1),(1,0)] \;\Leftrightarrow\; f^+_{i,e}(\theta) = \begin{cases} 2, & \text{falls } \theta \in [0, 0.5) \\ 1, & \text{falls } \theta \in [0.5, 1) \\ 0, & \text{falls } \theta \geq 1 \end{cases} \tag{3.4}$$

Der gleiche Zusammenhang besteht zwischen `fm`$[i][e]$ und $f^-_{i,e}$. Weiter wird für jede Kante $e \in E$ in `q_global` die Liste `q_global`$[e]$ gespeichert, die alle benötigten Warteschlangenlängen von Kante e beinhaltet, damit die stückweise konstante Warteschlange q_e beschrieben werden kann. Die Liste `q_ind`$[e]$ beinhaltet dann die Indizes der zugehörigen Zeitpunkte und `global_phase` die Startzeitpunkte *aller* Phasen. Das Format `lil_matrix` („List of Lists“) stammt hierbei aus der Bibliothek `scipy.sparse`, gedacht zum Speichern von dünnbesetzten Matrizen. Die Speicherung ist zeilenbasiert und eignet sich für die iterative Konstruktion der Parameter `E_active`, `b`, `x_total` und `a`.

Ausgeführt wird das Programm über ContMainMulti.py, welche im wesentlichen ein Objekt der Klasse `ContAppMulti` mit den Daten aus data.py erzeugt:

```
import data
from ContAppMulti import ContAppMulti

class ContMainMulti:
  app: ContAppMulti(graph, u)
  # ...
```

Codeabschnitt 3: Erzeugung eines Objekts der Klasse `ContAppMulti` in ContMainMulti.py, mit den Daten aus data.py

3.2 Ausgabe

Für die Ausgabe wird die Pythonbibliothek `pickle` verwendet. Nach der Berechnung eines Flusses in ContAppMulti.py erfolgen nachstehende Befehle, welche alle notwendigen Ergebnisse zur Beschreibung des Flusses in die Textdatei ausgabe.txt schreiben.

```
import pickle
# ...
with open('ausgabe.txt', 'ab') as f:
            pickle.dump(self.fp, f)
            pickle.dump(self.fm, f)
            pickle.dump(self.q_global, f)
            pickle.dump(self.q_ind, f)
            pickle.dump(self.global_phase, f)
            f.close()
```

Codeabschnitt 4: Erzeugung Ausgabedatei in ContAppMulti.py

Das Schreiben der Daten mit `pickle.dump()` erfolgt mit binärer Serialisierung in Form von Bytestreams. Diese Methode ist gut geeignet für das Schreiben großer Datenmengen. Um die Daten les- und verwendbar zu machen, werden diese im Anschluss umgeschrieben ins JSON-Format:

```
def loadall(filename):
    with open(filename, "rb") as f:
        while True:
            try:
                yield pickle.load(f)
            except EOFError:
                break

items = loadall("ausgabe.txt")
path = "flow.json"
output_json = open(path, "w")
# ... Beispielspezifisch ...
```

Codeabschnitt 5: Umwandlung der Ausgabe von .txt -, zu .json - Format

Mit diesem Vorgehen lassen sich die Daten einlesen und im Anschluss in der Datei flow.json abspeichern.

Eine weitere Möglichkeit zur Ausgabe bietet die Datei ContAppMultiOT.py unter Verwendung der Klasse `OutputTableMulti`. Dabei wird eine Tabelle erzeugt, welche alle Flusswerte enthält und die interaktiv vom Benutzer verändert werden kann. Diese Form der Ausgabe eignet sich nur für einfache Instanzen und sollte sowohl aus laufzeittechnischen, als auch aus praktischen Gründen nicht für größere Beispiele verwendet werden.

```
import tkinter as tk

class OutputTableMulti:
  table = tk.Tk()  # Ausgabetabelle
  menu = tk.Menu(self.table)  # Menüleiste
  # ...
```

Codeabschnitt 6: OutputTableMulti.py

Zur Visualisierung der Ausgabetabelle verwenden wir das GUI-Toolkit Tk, mit Hilfe der Bibliothek `tkinter`. Eine solche Tabelle wird erzeugt in ContAppMultiOT.py. Der Aufbau dieser Datei ist dem von ContAppMulti.py sehr ähnlich und beinhaltet lediglich einige Unterschiede bei der Speicherung der Daten. Wir betrachten beispielsweise:

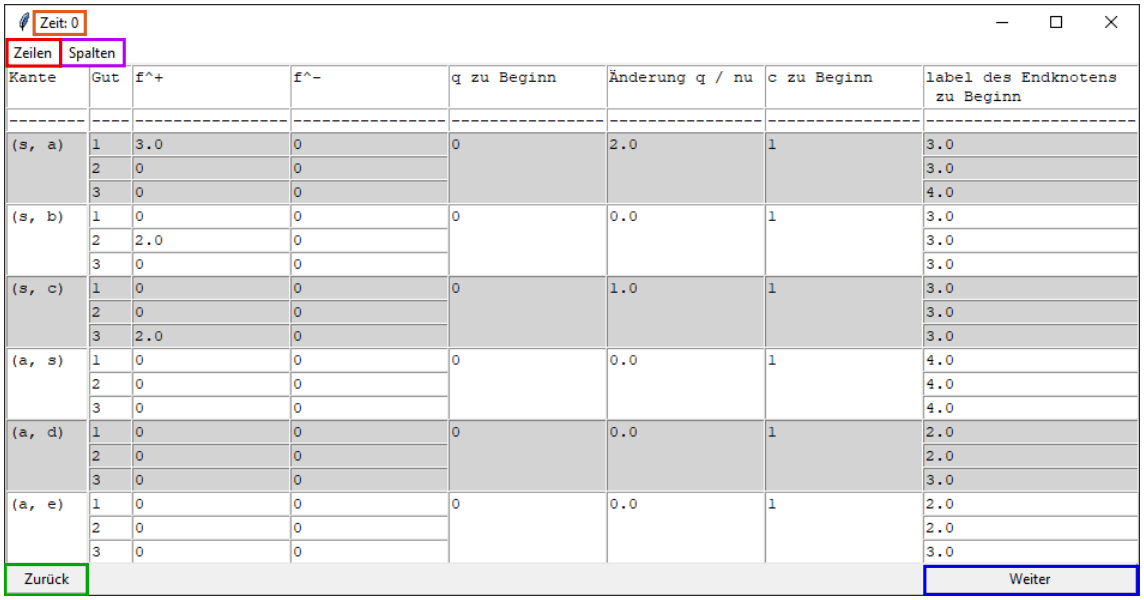
```
from OutputTableMulti import OutputTableMulti

class ContAppMultiOT:
  flow_vol: list[lil_matrix[|V|]]  # für jeden Zeitpunkt: Menge des
    momentan in einem Knoten vorhanden Flusses, für jeden Knoten
  b: list[lil_matrix[|I|, |V|]]  # wie b aus ContAppMulti.py, jedoch hier
    für jeden Zeitpunkt
  # ...
  OutputTableMulti[...]
```

Codeabschnitt 7: Erzeugung einer `OutputTableMulti` in ContAppMultiOT.py

Auch die Variablen `labels` und `c` werden in ContAppMultiOT.py für jeden Zeitpunkt gespeichert. Der benötigte Speicher steigt damit im Vergleich zu ContAppMulti.py enorm, was einer der Gründe ist, weshalb sich die Erzeugung einer Ausgabetabelle nur für kleine Beispiele eignet. Die Speicherung all dieser Daten ist jedoch notwendig, um die Tabelle füllen zu können.

Bei diesem Vorgehen wird ein Ausgabefenster erzeugt, mit dem der Benutzer interagieren kann. Der Aufbau dieses Fensters ist wie in Abbildung 2:



Kante	Gut	f^+	f^-	q zu Beginn	Änderung q / nu	c zu Beginn	label des Endknotens zu Beginn
(s, a)	1	3.0	0	0	2.0	1	3.0
	2	0	0				3.0
	3	0	0				4.0
(s, b)	1	0	0	0	0.0	1	3.0
	2	2.0	0				3.0
	3	0	0				3.0
(s, c)	1	0	0	0	1.0	1	3.0
	2	0	0				3.0
	3	2.0	0				3.0
(a, s)	1	0	0	0	0.0	1	4.0
	2	0	0				4.0
	3	0	0				4.0
(a, d)	1	0	0	0	0.0	1	2.0
	2	0	0				2.0
	3	0	0				3.0
(a, e)	1	0	0	0	0.0	1	2.0
	2	0	0				2.0
	3	0	0				3.0

Abbildung 2: Darstellung eines Objekts der Klasse OutputTableMulti.

- Zeilen - Kontextmenüpunkt: Verwaltung der angezeigten Kanten mittels Checkboxen. Siehe auch Abbildung 3 (links).
- Spalten - Kontextmenüpunkt: Verwaltung der angezeigten Spalten mittels Checkboxen. Siehe auch Abbildung 3 (rechts).
- Zurück - Button: Gehe zur vorherigen Phase.
- Weiter - Button: Gehe zur nächsten Phase.
- Zeitanzeige: Startzeitpunkt der betrachteten Phase.

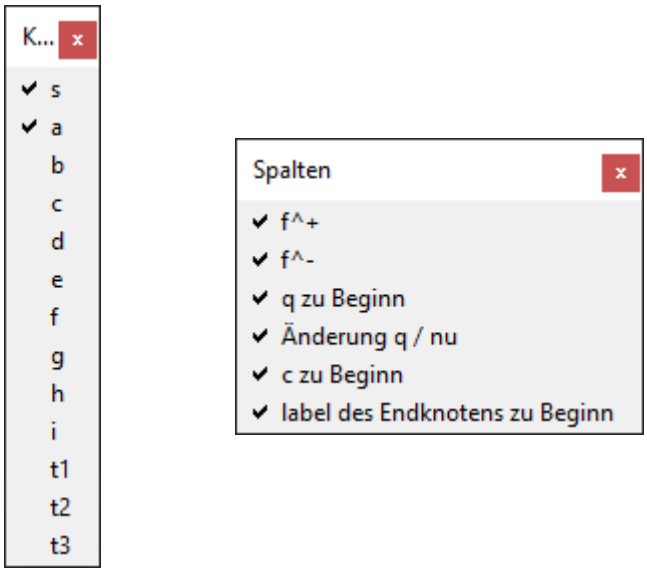


Abbildung 3: Unter Menüpunkt Zeilen: Auswahl der Knoten, deren ausgehende Kanten angezeigt werden (links).
Unter Menüpunkt Spalten: Auswahl der angezeigten Spalten (rechts).

Eine `OutputTableMulti` zeigt immer die Situation zu Beginn der betrachteten Phase. Demnach enthält die Spalte `q zu Beginn` die Warteschlangenlängen zu dem in der • Zeitanzeige angegebenen Zeitpunkt. Selbiges gilt für `c zu Beginn` und die Kantenkosten c, sowie `label des Endknotens zu Beginn` und das Label $\ell_w^i(\theta)$, wobei w den Endknoten der betrachteten Kante bezeichne. Die Spalten `f^+`, `f^-` und `Änderung q / nu` für Gut i und Kante e zum Zeitpunkt θ, korrespondieren zu den Werten $f_{i,e}^+(\theta)$, $f_{i,e}^-(\theta)$ und $g_e(f_e^+(\theta))/\nu_e$ und bleiben während der gesamten Phase mit Startzeitpunkt θ konstant.

Zur Ausführung des Programms mit einer solchen Ausgabetabelle muss die Zeile 5 von Codeabschnitt 3 folgendermaßen angepasst werden:

```
1 app: ContAppMultiOT(graph, u)
```

Codeabschnitt 8: Anpassung von ContMainMulti.py, zur Erzeugung einer `OutputTableMulti`.

3.3 Fehlerberechnung

Bei der Konstruktion des Algorithmus in [2] wurde notwendigerweise das Konzept der aktiven Kanten erweitert zu dem der approximiert aktiven Kanten. Dabei gilt eine Kante $e = (v, w) \in E$ für ein Gut $i \in I$ als *approximiert aktiv* zum Zeitpunkt θ, wenn gilt :

$$\ell_v^i(\theta) - \ell_w^i(\theta) - c_e(\theta) > -\delta, \tag{3.5}$$

wobei $\delta > 0$ in [2] genauer erklärt wird. Anschaulich gesehen wird dadurch für Kante e ein Toleranzbereich zugelassen, um den der durch diese Kante induzierte Weg zu t_i die Kosten eines tatsächlich kürzesten Weges überschreiten darf und dennoch als kürzester Weg behandelt wird.

Durch die Approximation der Flusswerte und dem dadurch implizierten Vorgehen bei der Berechnung von IDE-Flüssen entstehen Ungenauigkeiten, welche u.a. mittels folgender beider Fehlermaße beurteilt wurden:

Definition 3.1 ([2])**.** Es sei $(G, I, \nu, \tau, (u_i)_{i\in I}, (t_i)_{i\in I})$ ein Flussnetzwerk und f ein zugehöriger approximierter IDE-Fluss. Gilt für einen Zeitpunkt $\theta \in \mathbb{R}_{\geq 0}$, einen Knoten $v \in V$ und ein Gut $i \in I$, von welchem sich momentan Fluss in v befindet, dass $|\delta_{i,v}^+(\theta)| > 1$, so bezeichnet

$$Err_{i,v}(\theta) := \max_{e^+=(v,w^+)\in\widetilde{\delta}_{i,v}^+(\theta):f_{i,e^+}^+(\theta)>0} \ell_{w^+}^i(\theta) + c_{e^+}(\theta) \tag{3.6}$$

$$- \min_{e^-=(v,w^-)\in\widetilde{\delta}_{i,v}^+(\theta)} \ell_{w^-}^i(\theta) + c_{e^-}(\theta) \tag{3.7}$$

den *IDE-Fehler* von Gut i in Knoten v zum Zeitpunkt θ.

Dabei beschreibt $\widetilde{\delta}_{i,v}^+(\theta)$ die Menge der von v ausgehenden, zum Zeitpunkt θ für i approximiert aktiven Kanten.

Folgender Codeausschnitt zeigt das Vorgehen beim Berechnen der IDE-Fehler aller (i, v) - Paare:

```
for i in range(self.I):
  err_i = 0  # gesamter IDE-Fehler für Gut 'i'
  for v in range(self.n):
    if self.b[i, v] > 0:  # befindet sich Gut 'i' in 'v'?
      if len(dp_act_old[i][v]) == 1:
        continue
      vmax = -np.Inf
      vmin = np.Inf
      for e_ind in dp_act_old[i][v]:
        w = self.V.index(self.E[e_ind][1])
        val = self.labels[i][w] + self.c[e_ind]
        if self.fp[i][e_ind][-1][1] > 0 and val > vmax:
          vmax = val
        if val < vmin:
          vmin = val
      err_i += vmax - vmin
  if err_i < self.eps:
    err_i = 0
  s_err[i].append((theta, err_i))
```

Codeabschnitt 9: IDE-Fehlerberechnung in ContAppMain.py

Es werden hier für jedes Gut die IDE-Fehler aller Knoten in der Variable `err_i` aufsummiert und am Ende zusammen mit dem aktuellen Zeitpunkt in `s_err[i]` gespeichert. Die Liste `dp_act_old[i][v]` beschreibt dabei die Menge $\widetilde{\delta}^+_{i,v}$ zu Beginn der aktuell betrachteten Phase. Das Schlüsselwort „`old`“ deutet hier auf den Stand der Menge zu Beginn der Phase hin. Intern wird $\widetilde{\delta}^+_{i,v}$ während einer Phase durch zwei unterschiedliche Listen repräsentiert: Einmal wird die Menge zu Beginn der Phase gespeichert, später auch am Ende der Phase, wobei letzteres mit Hilfe der Liste `delta_p_act` geschieht.

Während einer Iteration wird obiger Codeabschnitt *zweimal* ausgeführt. Zum ersten Mal geschieht dies unmittelbar nach der Erweiterung des Flusses bis zur nächsten Phase. Die Berechnung an dieser Stelle liefert damit einen Überblick über den Verlauf des Fehlers in der aktuell betrachteten Phase. Anschließend werden mittels `refine()` die Knotendistanzen erneut angepasst und Codeabschnitt 9 wird zum zweiten Mal ausgeführt, womit ein Überblick der IDE-Fehler zu Beginn der nächsten Phase erzielt wird.

Das zweite hier betrachtete Fehlermaß ist der *Fehler im Knotenlabel*:

Definition 3.2 ([2])**.** In der Situation von Definition 3.1 sei $\ell^*_{i,v}(\theta)$ die exakte Distanz von Knoten v zur Senke t_i zum Zeitpunkt θ. Dann ist

$$\lambda_{i,v}(\theta) \coloneqq \ell_{i,v}(\theta) - \ell^*_{i,v}(\theta) \tag{3.8}$$

der *Fehler im Knotenlabel* von Gut i im Knoten v zum Zeitpunkt θ.

Da Algorithmus 1 nur mit den approximierten Labels ℓ arbeitet, müssen zur Berechnung des Fehlers im Knotenlabel die exakten Distanzen ℓ^* mittels zusätzlicher Berechnungen bestimmt werden. Dazu wird die Speicherung der exakten Kantenkosten `self.c_exact` benötigt, für die dann der Algorithmus von Dijkstra für jedes Gut

ausgeführt werden kann. Diese Methode ist mit einem hohen Rechenaufwand verbunden, ist aber notwendig, da eine Zwischenspeicherung der exakten Knotenlabels mit den gegebenen Werten nicht möglich ist: Die vom Algorithmus verwendete Variable a für die Steigung der Labels ℓ ist ebenfalls eine Approximation an die tatsächliche, unbekannte, Steigung a^*. Ansonsten ist die Bestimmung der Fehler im Knotenlabel unkompliziert und geschieht durch ähnliches Vorgehen wie im Fall der IDE-Fehler, unter Verwendung folgender Variablen:

```
l_star: list[list[float]]  # λ*
l_err_max: list[list[tuple[Time, float]]]  # für jedes Gut: maximale
    Überschreitung von λ* durch λ; wird gespeichert zusammen mit
    Zeitpunkt
l_err_min: list[list[tuple[Time, float]]]  # analog für Minimum
```

Codeabschnitt 10: Variablen in ContAppMain.py

Deren Berechnung geschieht folgendermaßen:

```
1  l_star = []
2  for i in range(self.I):
3    l_star.append(self.dijkstra(self.graphReversed, 't{}'.format(i+1)
       , visited=[], distances={}, exact=True))
4    l_max = 0
5    l_min = 0
6    for v_ind in range(self.n):
7      if self.b[i][v_ind] > 0:
8        l_diff = self.labels[i][v_ind] - l_star[i][v_ind]
9        if l_diff > l_max:
10         l_max = l_diff
11       elif l_diff < l_min:
12         l_min = l_diff
13   l_err_max[i].append((theta, l_max))
14   l_err_min[i].append((theta, l_min))
```

Codeabschnitt 11: Berechnung der Fehler im Knotenlabel in ContAppMain.py

Beim Berechnen der Distanzen in Zeile 3 mittels `self.dijkstra()` wird hier der optionale Parameter `exact=True` übergeben, sodass für die Berechnung die exakten Kantenkosten verwendet werden. In Zeile 8 wird die Differenz des approximierten Knotenlabels und des exakten Knotenlabels berechnet. Ist Ersteres größer, so ist die Differenz positiv und wird bei der Setzung von `l_max` für das Paar (i, v) berücksichtigt. Analog werden die negativen Differenzen für die Setzung von `l_min` verwendet. Diese Werte werden nach Durchlauf aller Knoten in `l_err_max[i]`, bzw. `l_err_min[i]`, zusammen mit dem aktuellen Zeitpunkt gespeichert.

3.4 Zwischenergebnisse

Bei der Arbeit mit dem Algorithmus kann es sinnvoll sein, Ergebnisse zwischenzuspeichern, um Zeit einzusparen. Wird der Algorithmus mehrmals mit demselben Eingabegraph ausgeführt, so reicht es, den Algorithmus von Dijkstra (für jedes Gut) einmal auszuführen und die Knotendistanzen abzuspeichern. Dies geschieht im folgenden Codeausschnitt aus ContAppMulti.py, erneut mit der Funktion `dump()` aus der Bibliothek `pickle`.

```
# alle Kanten werden berücksichtigt
self.E_active = np.ones((self.I, self.m))
for i in range(self.I):
  # Ausführung Dijkstra
  self.labels.append(self.dijkstra(self.graphReversed, 't{}'.format
    (i+1), visited=[], distances={}))

# Speichern der Werte in Datei distanzen.txt
with open('distanzen.txt', 'wb') as f:
  for i in range(self.I):
    pickle.dump(self.labels[i], f)
  f.close()
```

Codeabschnitt 12: Speichern der Knotenlabels

Damit müssen die Daten in späteren Aufrufen nur noch eingelesen werden:

```
items = loadall('distanzen.txt')
for item in items:
  self.labels.append(item.copy())
```

Codeabschnitt 13: Wiederverwenden der gespeicherten Knotenlabels

4 Anwendung

Wir wollen nun beispielhaft einen IDE-Fluss für das Flussnetzwerk aus Abbildung 4 berechnen. Dieses Netzwerk wurde entnommen aus [2]. Da der ursprüngliche Graph keine Kreise enthielt, wurde das Beispiel hier noch erweitert, indem zu allen Kanten die jeweilige Rückwärtskante mit derselben Kapazität und Reiselänge hinzugefügt wurde. Die Kanten sind in der Abbildung als Doppelpfeile dargestellt, womit genau genommen zwei unterschiedliche Kanten, mit entgegengesetzter Orientierung, gemeint sind.

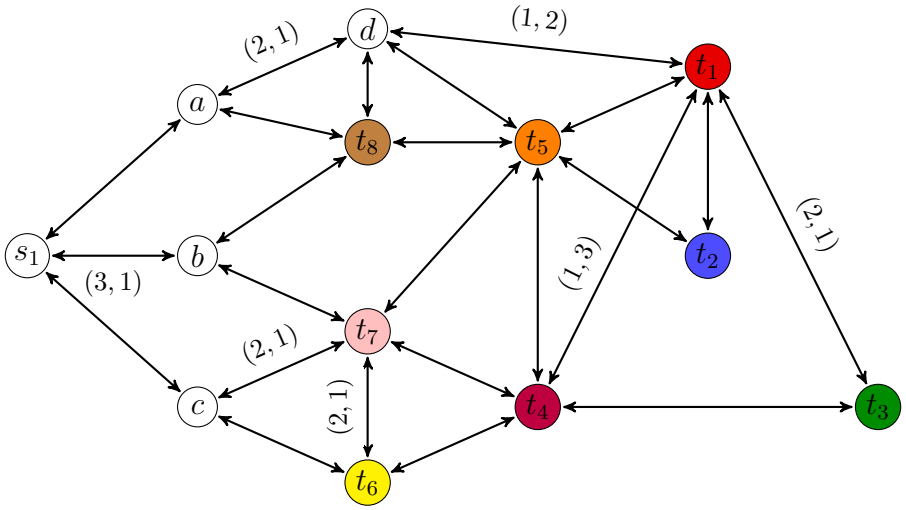


Abbildung 4: Beispielgraph G_1. Von 1 verschiedene Kapazitäten und Reiselängen sind angegeben in der Form (ν_e, τ_e).

Die Einflussraten der ersten drei Güter sind beschrieben in folgender Tabelle, während die Einflussraten der übrigen Güter vorerst auf 0 festgesetzt werden:

Knoten	s			b	d	t_8	t_6	t_5		
Einflussrate	3	2	2	3	4	4	7	5	5	5
Einflusszeitraum	$[0,1)$			$[0,1)$	$[0,1)$	$[1,2)$	$[0,2)$	$[0,1)$		

Tabelle 1: Einflussraten Gut 1 (rot), Gut 2 (blau) und Gut 3 (grün).

Wir betrachten außerdem noch weitere Varianten des Beispiels:

A. Instanz mit Graph G_1 wie in Abbildung 4, mit Einflussraten aus Tabelle 1.

B. Ursprüngliche Instanz aus [2], ohne Rückwärtskanten, mit Einflussraten aus Tabelle 1.

C. Wie A, jedoch mit reduzierter Einflussmenge in Knoten t_5:
$u_1(t_5, \theta) = u_2(t_5, \theta) = u_3(t_5, \theta) = 2$, für $\theta \in [0, 1)$.

D. Wie A, jedoch mit zusätzlicher Kante (b, t_1), mit $\nu_{(b,t_1)} = 1$, $\tau_{(b,t_1)} = 3$.

E. Wie A, jedoch mit positivem Einfluss über den Zeitraum $[0, 1)$ von fünf weiteren Gütern in Knoten s, alle mit Rate 1. Die Farbgebung ist wie in Abbildung 4, also Gut 4 (lila), Gut 5 (orange), Gut 6 (gelb), Gut 7 (pink) und Gut 8 (braun), und impliziert somit die Senke des jeweiligen Guts.

Mit einer Rechengenauigkeit von `bd_tol` $= 10^{-5}$ erhalten wir folgende Ergebnisse:

Variante	A	B	C	D	E
gesamter Einfluss	47	47	38	47	52
# Iterationen	229	104	225	281	469
davon mit Flussberechnung	154	56	165	186	331
davon ohne Flussberechnung	75	48	60	95	138
Terminationszeitpunkt	13.078	13.769	10.777	11.946	13.198

Tabelle 2: Vergleich der Varianten A - E.

Es ist zu beobachten: In Variante A existieren natürlicherweise deutlich mehr Wege zu den Zielknoten als in Variante B. Bei der Berechnung des IDE-Flusses entstehen daher signifikant mehr Iterationen (bzw. Phasen), dafür terminiert der berechnete Fluss letztendlich um fast 0.7 Zeiteinheiten früher. Durch die Reduktion der externen Einflussraten in Knoten t_5 in Variante C, wird auch die Wartezeit der von diesem Knoten ausgehenden Kanten reduziert, sodass der Fluss in diesem Fall weitere ≈ 2.3 Zeiteinheiten früher terminiert. Variante D bietet im Vergleich zu Variante A eine Abkürzung für alle drei Güter und entlastet so andere Kanten innerhalb des Netzwerks. Dadurch werden etwa 1.1 Zeiteinheiten eingespart.

In Variante E wird die Anzahl der Güter auf 8 erhöht, das gesamte dem Netzwerk zugeführte Flussvolumen auf 52. Da die Senken der Güter 4 - 8 einen kleineren Abstand zu s_1 haben, als die ersten drei und jeweils nur eine Flusseinheit transportieren müssen, verzögert sich die Termination des Flusses lediglich um etwa 0.12 Zeiteinheiten.

Die hier berechneten Flüsse können im Unterordner output_examples im JSON-Format eingesehen werden. Zur Visualisierung können diese Dateien an das Tool unter https://github.com/ArbeitsgruppeTobiasHarks/dynamic-flow-visualization/tree/main übergeben werden.

Zum Abschluss wird nun noch ein weiteres Beispiel betrachtet, für das auch die IDE-Fehler und Fehler im Knotenlabel berechnet werden. Der Beispielgraph ist hierbei das aus der Literatur bekannte Sioux-Falls[1] Netzwerk, dargestellt mit den hier verwendeten Quell- und Zielknoten in Abbildung 5. Während die Reiselängen übernommen wurden, wurden die Kapazitäten aller Kanten zur besseren Übersichtlichkeit angepasst, indem die Kapazitäten aus dieser Quelle[2] durch 1000 dividiert und nach der dritten Nachkommastelle abgeschnitten wurden.

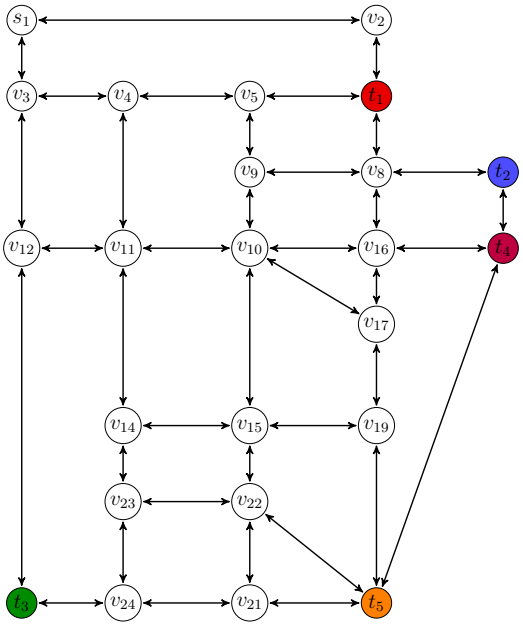


Abbildung 5: Sioux-Falls Netzwerk.

Zur Berechnung verwenden wir folgende externe Einflussraten in Knoten s_1:

Einflussrate	80	70	60	50	40
Einflusszeitraum	$[0, 3)$	$[0, 3)$	$[0, 2)$	$[0, 1.5)$	$[0, 2.2)$

Tabelle 3: Einflussraten in Knoten s_1: Gut 1 (rot), Gut 2 (blau), Gut 3 (grün), Gut 4 (lila) und Gut 5 (orange).

[1] Entnommen von https://github.com/bstabler/TransportationNetworks/tree/master/SiouxFalls.

[2] siehe Fußnote 1.

Mit einer Genauigkeit von `bd_tol` $= 10^{-5}$ ergibt sich ein approximierter IDE-Fluss, mit 410 Phasen, von denen in 287 eine Flussberechnung mittels `fp_approx()` erfolgt. Der berechnete Fluss terminiert etwa zum Zeitpunkt $\theta \approx 59.939$. Davon treten in den ersten 40 Zeiteinheiten IDE-Fehler für die Güter 1, 2 und 4 auf, welche jeweils im oberen Teil von Abbildung 6 aufgetragen sind. Die Berechnung geschieht wie in Codeabschnitt 9, d.h. es werden für jedes Gut die IDE-Fehler in allen Knoten aufsummiert. Die IDE-Fehler der übrigen beiden Güter sind stets 0 und wurden daher aus der Abbildung ausgeschlossen.

Der untere Teil der Abbildung stellt dann den resultierenden absoluten, bzw. relativen, IDE-Gesamtfehler dar.

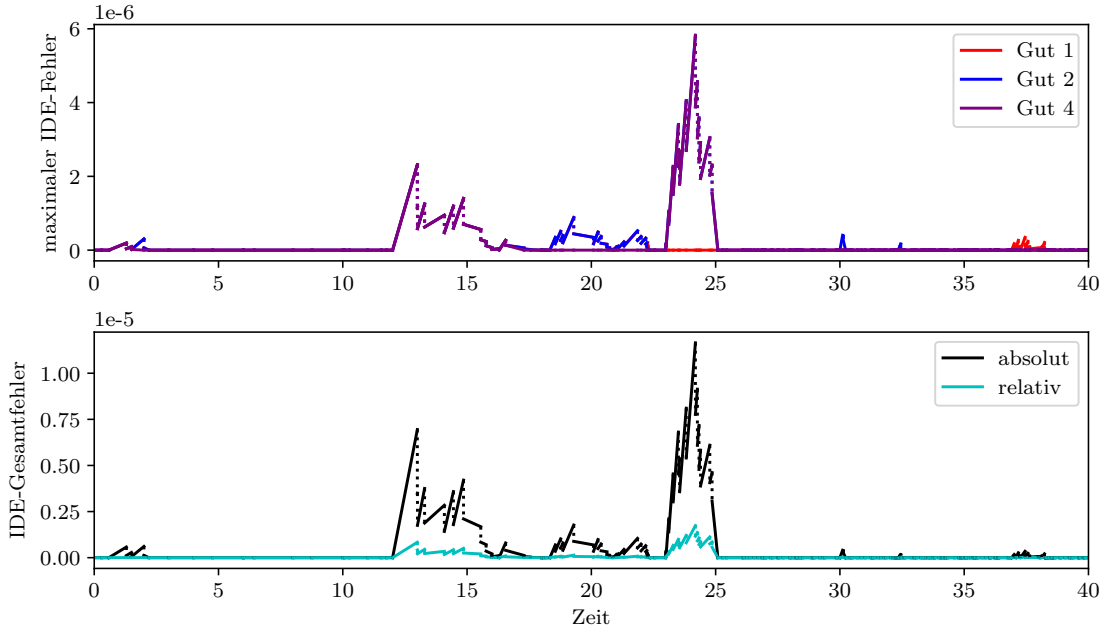


Abbildung 6: maximale IDE-Fehler der Güter 1, 2 und 4 (oben), sowie entsprechender absoluter und relativer Gesamtfehler (unten), für `bd_tol` $= 10^{-5}$.

Beginnend mit Phase 22 bei $\theta_{22} \approx 12.004$ bis zu $\theta_{34} \approx 16.291$ ist der Verlauf der IDE-Fehler aller drei betrachteter Güter gleich. Dies liegt daran, dass die Ursache für die Fehler aller drei Güter die gleiche ist: Die Flussaufteilung im Knoten v_2 auf die Kanten (v_2, s) und (v_2, t_1). Diese Kanten sind im gesamten betrachteten Zeitraum für alle drei Güter aktiv, sodass die resultierenden Labeländerungen und IDE-Fehler der Güter 1, 2 und 4 ebenfalls gleich sind.

Der Verlauf der Fehler im Knotenlabel wird für alle Güter beschrieben in Abbildung 7:

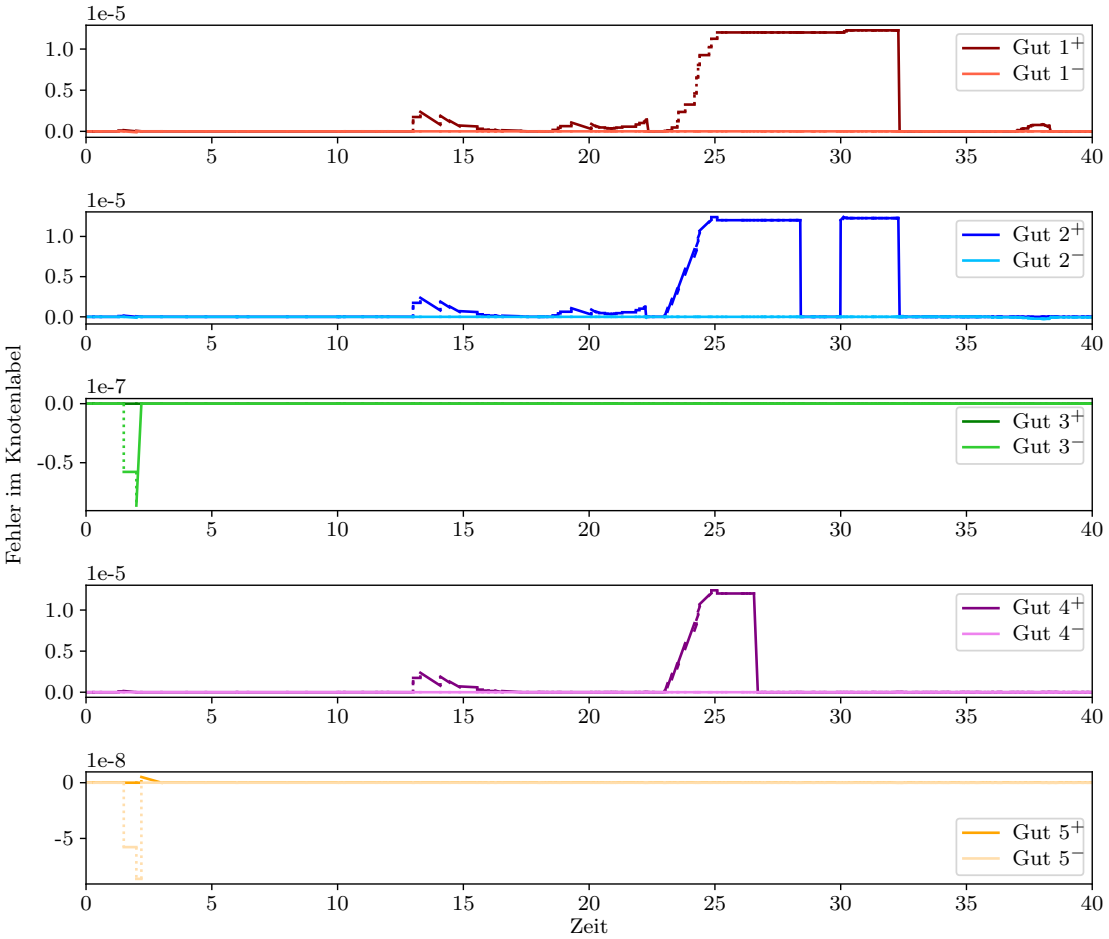


Abbildung 7: Positiv- und Negativteil der Fehler im Knotenlabel für alle Güter, für `bd_tol` $= 10^{-5}$.

Die Aufteilung in Positiv - und Negativteil geschieht hier wir in Codeabschnitt 11. Auch die Fehler im Knotenlabel der Güter 1, 2 und 4 sind in den Phasen 22 - 33 gleich. Die Ursache dieser Fehler liegt auch hier bei den Kanten (v_2, s) und (v_2, t_1), die für alle Güter approximiert aktiv, mit den selben Kostenunterschieden, sind. Des Weiteren ist auffällig, dass in diesem Beispiel die Fehler im Knotenlabel deutlich in die positive Richtung tendieren, d.h. die approximierten Knotenlabels sind in den meisten Fällen größer als die exakten. Dies wird vor allem augenfällig ab Phase 68, mit $\theta_{68} \approx 23.116$, bei der der stärkste Anstieg der Fehler der Güter 1, 2 und 4 geschieht. Die Fehler der Güter 2 und 4 verhalten sich hier wieder sehr ähnlich, deren Unterschiede liegen stets in einer Größenordnung von unter 10^{-8} und damit deutlich unterhalb der verwendeten Approximationsgenauigkeit von `bd_tol` $= 10^{-5}$. Die Tendenz der approximierten Labels, die exakten Labels zu übersteigen, ist hier lediglich ein Spezialfall des Algorithmus. Einerseits sind die approximierten Flusswerte an den entscheidenden Stellen derartig, dass die Fehler im Knotenlabel steigen, andererseits werden auch in `refine()` die Labels größtenteils nach oben korrigiert, s.d. die meisten Sprungstellen der Fehler der Güter 1, 2 und 4 in positive Richtung erfolgen. Dieser Fall kann eintreten, da `refine()` nur eine Angleichung der approximierten Labels anstrebt und sich nicht an den exakten Labels orientiert.

5 Fazit

In dieser Arbeit wurde der in [2] vorgestellte Algorithmus genauer betrachtet, wobei insbesondere auf einige Details in der Implementierung eingegangen wurde, die in der ursprünglichen Quelle nicht im Fokus standen. Wir haben weiterhin in Kapitel 4 einige weitere Anwendungsbeispiele betrachtet, in denen wir sinnvolle Approximationen von IDE-Flüssen berechnen konnten.

6 Literatur

[1] Lukas Graf, Tobias Harks, and Leon Sering. Dynamic flows with adaptive route choice. *Mathematical Programming*, 183(1):309–335, 2020.

[2] Johannes Hagenmaier. Approximation dynamischer Flüsse mit adaptiver Routenwahl in Netzwerken mit mehreren Senken. Master's thesis, Universität Augsburg, 2023.

[3] William S. Vickrey. Congestion theory and transport investment. *The American Economic Review*, 59(2):251–260, 1969.