Karlsruher Institut für Technologie

Algorithmen für Routenplanung

19. Vorlesung, Sommersemester 2016

Ben Strasser | 6. Juli 2016

Institut für Theoretische Informatik · Algorithmik · Prof. Dr. Dorothea Wagner

Beschleunigungstechniken auf zeitexpandiertem Graph

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Beschleunigungstechniken

Idee:

- Bilde zeitexpandierter Graph
- Wende bestehende Techniken für Straßen darauf an

Beschleunigungstechniken

Idee:

- Bilde zeitexpandierter Graph
- Wende bestehende Techniken für Straßen darauf an

- Funktioniert nicht so gut [BDGM09]
- Graphen haben andere Struktur

Beschleunigungstechniken

Idee:

- Bilde zeitexpandierter Graph
- Wende bestehende Techniken für Straßen darauf an

- Funktioniert nicht so gut [BDGM09]
- Graphen haben andere Struktur

- Es gibt eigene Techniken für Fahrpläne (nicht Gegenstand der Vorlesung)
- Siehe [BDG+16] für eine Übersicht

Cuts

Beobachtung:

- Stop s entspricht Pfad in zeitexpandiertem Graph
- Beispiel: $(s, 1) \to (s, 3) \to (s, 4) \to \cdots$
- Trips entsprechen auch Pfaden
- Pfade sind Knoten-disjunkt
- Jeder Knoten ist in einem Trip oder Stop

Cuts

- Jeder Zeitpunkt ist ein Cut:
  - Knoten davor bilden eine Seite
  - Knoten danach bilden eine Seite
- Es kann sinnvoll sein den Zeitpunkt pro Stop/Trip zeitlich leicht zu verschieben um die Cuts kleiner zu machen
- Eine Bipartition der Stops und Trips ist ein Cut:
  - Die Knoten in den Pfaden gehören zur Seite ihres Stops/Trips
- Es gibt noch Cuts die eine Mischung von beiden sind

Cuts

- Jeder Zeitpunkt ist ein Cut:
  - Knoten davor bilden eine Seite
  - Knoten danach bilden eine Seite
- Es kann sinnvoll sein den Zeitpunkt pro Stop/Trip zeitlich leicht zu verschieben um die Cuts kleiner zu machen
- Eine Bipartition der Stops und Trips ist ein Cut:
  - Die Knoten in den Pfaden gehören zur Seite ihres Stops/Trips
- Es gibt noch Cuts die eine Mischung von beiden sind

Frage:

- Was sind die kleinsten balanzierten Cuts?

Cuts

Frage:

- Was sind die kleinsten balanzierten Cuts?

Antwort:

- Betrachte Graph mit sehr langem Zeithorizont, z.B. ein Jahr
- Die Stop-Pfade bilden sehr lange Ketten
- Wir können nicht zwischen ihnen scheiden da dazwischen zu viele Züge fahren
- Wenn nur ein Zug pro Tag fährt, dann sind das schon 365 Kanten
- Wir müssen durch die Stop-Pfade schneiden
- Anzahl an Stops ist untere Schranke für Cut-Größe
- Das ist zu groß für partitionsbasierte Techniken

Exkurs: Reachability Compression

Problemstellung

- **Eingabe**: Gerichteter Graph G, Knoten s und t
- **Ausgabe**: Ob es einen Pfad von s nach t gibt

Exkurs: Reachability Compression

Karlsruher Institut für Technologie

Problemstellung

- **Eingabe**: Gerichteter Graph G, Knoten s und t
- **Ausgabe**: Ob es einen Pfad von s nach t gibt

Aufbau

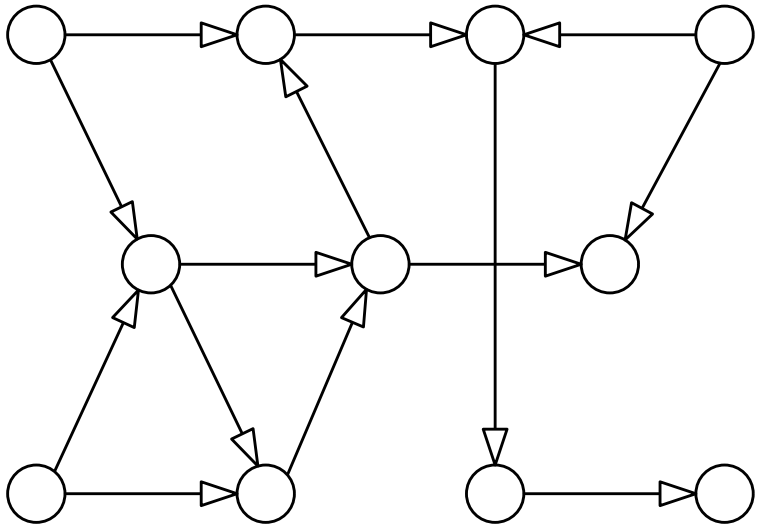
- 2-Phasen Ansatz:
  - Vorberechung kennt nur G
  - Query kennt auch s und t
- Läuft unter dem Namen “Reachability Compression”
- Viel Literatur

Exkurs: Reachability Compression

Häufiger Ansatz: Chain-Decomposition

- Finde möglichst kleine disjunkte Menge von Pfaden, genannt Chains, die alle Knoten überdecken
- Kanten außerhalb der Pfade werden aufgefasst als Quadrupel aus
  - Ausgangspfad-ID
  - Zielpfad-ID
  - Position im Ausgangspfad
  - Position im Zielpfad
- Siehe [Jag90]

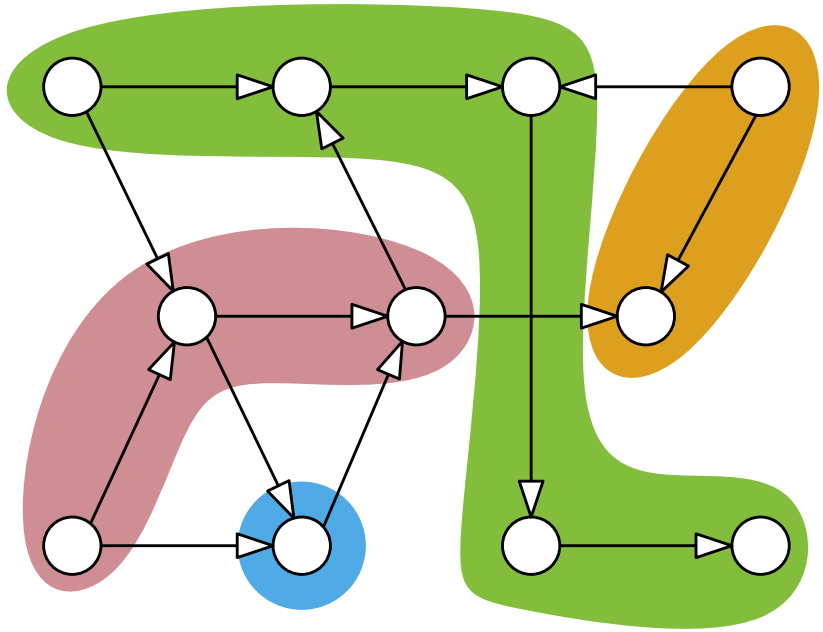
Exkurs: Chain-Decomposition



Der Eingabegraph G

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

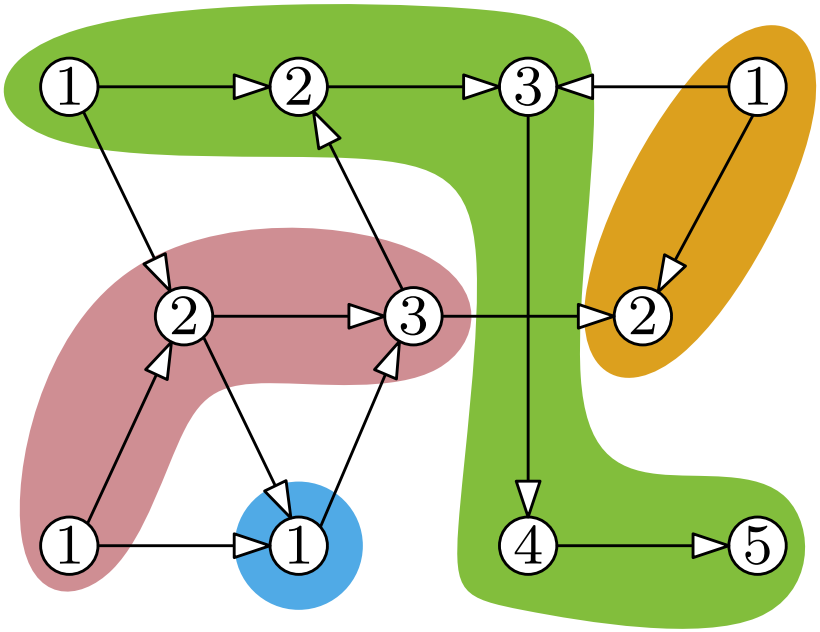
Exkurs: Chain-Decomposition



Eine Chain-Decomposition mit 4 Pfaden

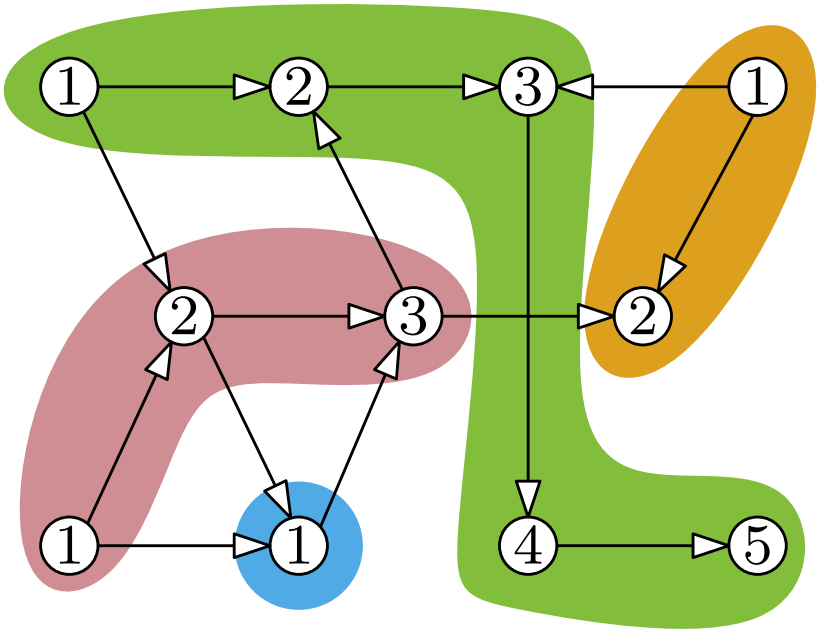
Institut für Theoretische Informatik
Lehrstuhl Algorithmik

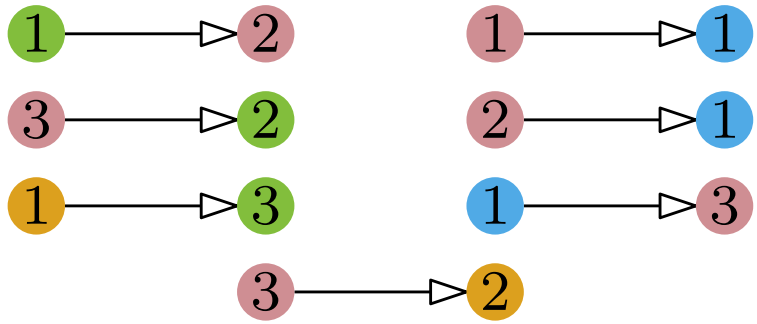
Exkurs: Chain-Decomposition



Positionen entlang der Pfade

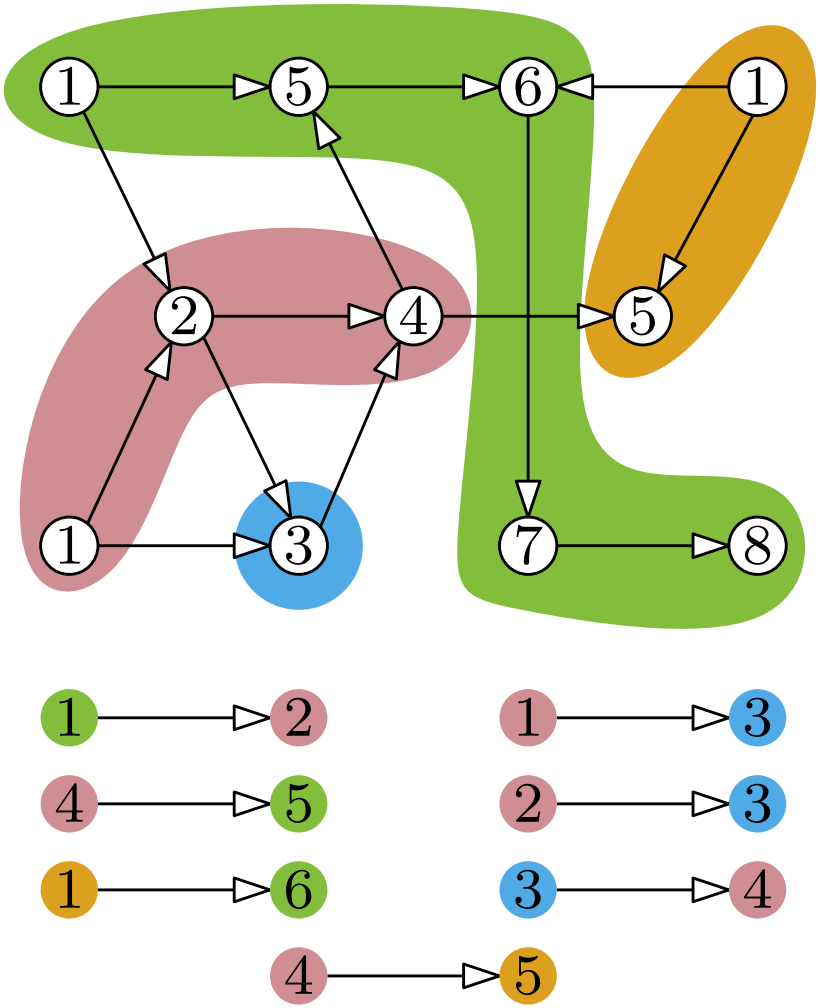
Exkurs: Chain-Decomposition





Die Kanten zwischen den Pfaden

Exkurs: Chain-Decomposition



Wir können die Positionen anpassen, so dass die Zielposition immer höher als die Startposition

Exkurs: Chain-Decomposition

Wie Erreichbarkeit prüfen?

- Nutze Variante von Dijkstras Algorithmus auf Graph G'
- Chains sind Knoten von G'
- Kanten von G außerhalb der Chains sind Kanten von G'
- Speichere für jede Chain die kleinste Position die erreichbar ist

Exkurs: Chain-Decomposition

Wie Erreichbarkeit prüfen?

- Nutze Variante von Dijkstras Algorithmus auf Graph G'
- Chains sind Knoten von G'
- Kanten von G außerhalb der Chains sind Kanten von G'
- Speichere für jede Chain die kleinste Position die erreichbar ist

Modifizierte Kantenrelaxierung in G':

- Es sei (u, v, u_p, v_p) die Quadrupel der Kante
  - Kante von u nach v
  - Von Position u_p nach v_p
- Es sei $P(x)$ die kleinste bekannte Position auf Chain x
- Die Relaxierung lautet:
  - Wenn $P(u) \leq u_p$ dann setze $P(v) \leftarrow \min\{P(v), v_p\}$

Was hat das mit Fahrplänen zu tun?

- Stop $\leftrightarrow$ Chain
- Zeit $\leftrightarrow$ Position
- Connection $\leftrightarrow$ Kante außerhalb der Chain

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Was hat das mit Fahrplänen zu tun?

- Stop $\leftrightarrow$ Chain
- Zeit $\leftrightarrow$ Position
- Connection $\leftrightarrow$ Kante außerhalb der Chain

Schlussfolgerung

- Fahrpläne sind bereits komprimiert
- Fahrpläne expandieren um sie dann wieder zu komprimieren macht nicht unbedingt Sinn

Zeitabhängige Variante von Dijkstras Algorithmus

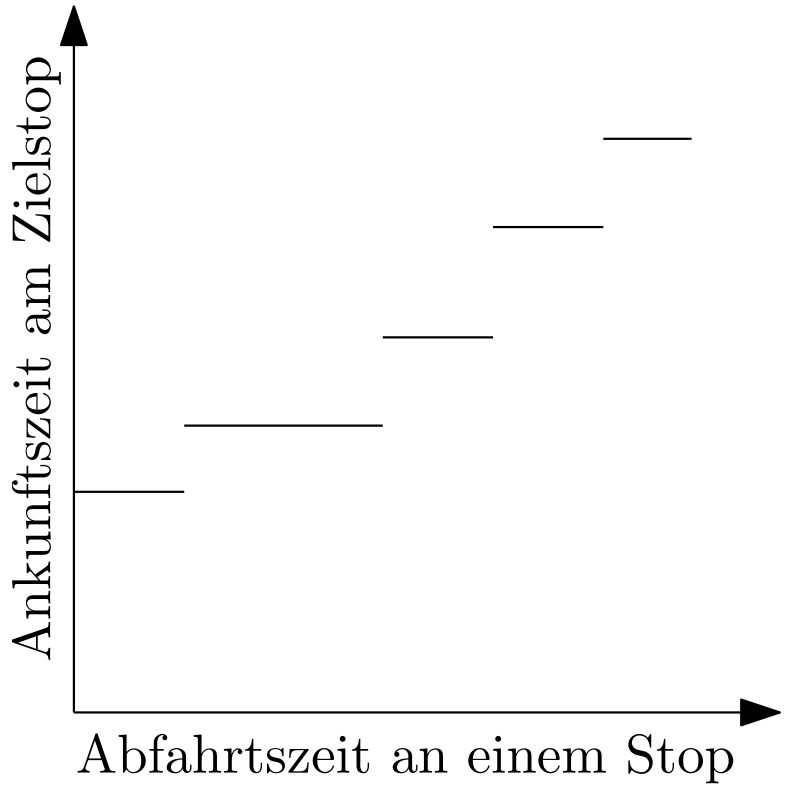
Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Idee

Idee

- Nutze ähnlichen Ansatz wie bei zeitabhängiger Straße
- Knoten sind (erstmal) Stops
- Wenn es eine Connection zwischen zwei Stops gibt, dann gibt es eine Kante zwischen den Knoten
- mehere Connections werden zusammengefasst
- Kanten sind gewichtet durch Reisezeitfunktion oder Ankunftszeitfunktion

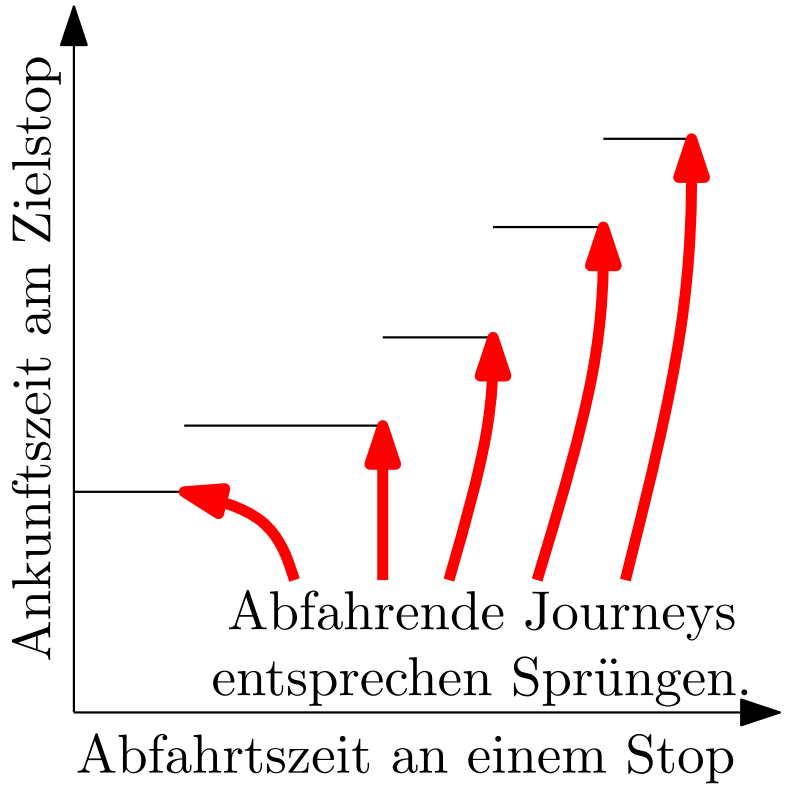
Ankunftszeitfunktion



Idee

- Ankunftszeitfunktion ist Stufenfunktion
- Stufen entsprechenden abfahrenden Zügen
- (Wegen Fußwegen kann es lineare Komponenten geben)

Ankunftszeitfunktion



Idee

- Ankunftszeitfunktion ist Stufenfunktion
- Stufen entsprechenden abfahrenden Zügen
- (Wegen Fußwegen kann es lineare Komponenten geben)

FIFO?

First-In-First-Out

- “Warten soll sich nicht lohnen”
- Sind die Ankunftszeitfunktionen FIFO? d.h. gilt

$$f(x) < f(y) \text{ für } x < y$$

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

FIFO?

First-In-First-Out

- “Warten soll sich nicht lohnen”
- Sind die Ankunftszeitfunktionen FIFO? d.h. gilt

$$f(x) < f(y) \text{ für } x < y$$

- Genau genommen nein, da $f(x) = f(y)$ sein kann
- Warten lohnt sich trotzdem nicht
- Also quasi-FIFO

FIFO?

First-In-First-Out

- “Warten soll sich nicht lohnen”
- Sind die Ankunftszeitfunktionen FIFO? d.h. gilt

$$f(x) < f(y) \text{ für } x < y$$

- Genau genommen nein, da $f(x) = f(y)$ sein kann
- Warten lohnt sich trotzdem nicht
- Also quasi-FIFO

- Die Algorithmen für Linken & Mergen die auf der Straße verwendet werden sind nicht direkt anwendbar, da f^{-1} nicht existiert

Ankunftszeitfunktion

- Funktion kann als Menge von Paaren (d_i, a_i) aufgefasst werden
- Jedes Paar besteht aus:
  - Abfahrtszeit d_i am Abfahrtsstop
  - Ankunftszeit a_i am Zielstop
- Menge darf keine dominierte Einträge enthalten
  - Das ist äquivalent mit quasi-FIFO
- Paare werden sortiert in einem Array gespeichtert
  - Wegen Dominierung simultan sortiert nach d_i und a_i

Dominierung

Ziel:

- **Eingabe**: Menge von Paaren (d_i, a_i)
- **Ausgabe**: Teilmenge nicht dominierter Paare

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Dominierung

Ziel:

- **Eingabe**: Menge von Paaren (d_i, a_i)
- **Ausgabe**: Teilmenge nicht dominierter Paare

Algorithmus:

- Sortiere nach einem der Elemente, z.b., sortiere nach d_i aufsteigenden
- Iteriere über die Element
- Lösche die Paare (d_i, a_i) für die gilt $a_{i+1} \leq a_i$
  - **Interpretation**: Wenn jetzt ein Zug fährt und später einer fährt und der spätere früher oder gleichzeitig ankommt, dann nimm den späteren.

Funktionsevaluation

Ziel:

- **Eingabe**: Funktion f
- **Ausgabe**: Der Wert $f(x)$

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Funktionsevaluation

Ziel:

- **Eingabe**: Funktion f
- **Ausgabe**: Der Wert $f(x)$

Algorithmus:

- Es seien (d_i, a_i) die sortierten Paare von f
- Finde das kleinste j mit $x \leq d_j$
  - mittels binärer Suche
  - oder: gutes Anfangs j raten und dann sequentiell ab da suchen
- Gebe a_j aus
- Falls es kein j gibt, dann fährt kein Zug mehr und gebe ∞ zurück

Link

Ziel:

- **Eingabe**: Zwei Funktionen f und g
- **Ausgabe**: Eine Funktion $g \circ f$
  (d.h. $g(f(x))$ für jedes x)

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Link

Ziel:

- **Eingabe**: Zwei Funktionen f und g
- **Ausgabe**: Eine Funktion $g \circ f$
  (d.h. $g(f(x))$ für jedes x)

Algorithmus:

- Es seien (d_i, a_i) die Paare von f
- Berechne $(d_i, g(a_i))$
- Filtere dominierte Einträge

Merge

Ziel:

- **Eingabe**: Zwei Funktionen f und g
- **Ausgabe**: Eine Funktion $\min\{f, g\}$
  (d.h. $\min\{f(x), g(x)\}$ für jedes x)

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Merge

Ziel:

- **Eingabe**: Zwei Funktionen f und g
- **Ausgabe**: Eine Funktion $\min\{f, g\}$
  (d.h. $\min\{f(x), g(x)\}$ für jedes x)

Algorithmus:

- Vereinige die Paare von f und g
  - Geht in Mergesort-artigem Merge-Schritt
- Filtere dominierte Einträge

Problem der frühesten Ankunft

$d[x] \leftarrow \infty$ für alle x
$d[s] \leftarrow 0$
$Q.clear()$, $Q.add(s, 0)$

while $!Q.empty()$ **do**
  $u \leftarrow Q.deleteMin()$
  for *all edges* $e = (u, v) \in E$ **do**
    if $d[u] + f_e(\tau + d[u]) < d[v]$ **then**
      $d[v] \leftarrow d[u] + f_e(\tau + d[u])$
      if $v \in Q$ **then** $Q.decreaseKey(v, d[v])$
      else $Q.insert(v, d[v])$

- Normaler zeitabhängige Erweiterung von Dijkstras Algorithmus
- Mit Reisezeitfunktionen f_e

Problem der frühesten Ankunft

$d[x] \leftarrow \infty$ für alle x
$d[s] \leftarrow \tau$
$Q.clear()$, $Q.add(s, \tau)$

while $!Q.empty()$ **do**
  $u \leftarrow Q.deleteMin()$
  for *all edges* $e = (u, v) \in E$ **do**
    if $d[u] + f_e(d[u]) < d[v]$ **then**
      $d[v] \leftarrow d[u] + f_e(d[u])$
      if $v \in Q$ **then** $Q.decreaseKey(v, d[v])$
      else $Q.insert(v, d[v])$

- Normaler zeitabhängige Erweiterung von Dijkstras Algorithmus
- Mit Reisezeitfunktionen f_e
- Mit τ initialisieren macht es leicht einfacher

Problem der frühesten Ankunft

$d[x] \leftarrow \infty$ für alle x
$d[s] \leftarrow \tau$
$Q.clear()$, $Q.add(s, \tau)$

while $!Q.empty()$ **do**
  $u \leftarrow Q.deleteMin()$
  for *all edges* $e = (u, v) \in E$ **do**
    if $g_e(d[u]) < d[v]$ **then**
      $d[v] \leftarrow g_e(d[u])$
      if $v \in Q$ **then** $Q.decreaseKey(v, d[v])$
      else $Q.insert(v, d[v])$

- Normaler zeitabhängige Erweiterung von Dijkstras Algorithmus
- Mit Ankunftszeitfunktion g_e
- Mit τ initialisieren macht es leicht einfacher

Profile Berechnen

- Dijkstra-Profil-Erweiterung für Straßengraphen kann verwendet werden
- Siehe Vorlesung bezüglich Wegesuche in zeitabhängigen Straßennetzen

Profile Berechnen

- Dijkstra-Profil-Erweiterung für Straßengraphen kann verwendet werden
- Siehe Vorlesung bezüglich Wegesuche in zeitabhängigen Straßennetzen

Wiederholung:

- $d(x)$ bildet nicht mehr auf einen Zeitpunk ab
- $d(x)$ ist eine Reisezeitfunktion vom Startstop zu x
- Relaxierung benutzt Link und Merge Operation
- Knoten können mehrfach aus der Queue genommen werden
- Keine polynomielle Laufzeitgarantie

Self-Pruning

Idee:

- Alternative Profilberechnung
- Starte eine Suche für jeden abfahrenden Zug am Startstop s
- Jede Suche löst das einfacherere Problem der frühesten Ankunft
- Self-Pruning: Nutze Informationen aus einen Suche um andere zu prunen

Self-Pruning

Idee:

- Alternative Profilberechnung
- Starte eine Suche für jeden abfahrenden Zug am Startstop s
- Jede Suche löst das einfacherere Problem der frühesten Ankunft
- Self-Pruning: Nutze Informationen aus einen Suche um andere zu prunen

Details:

- Es seien $\tau_1 \dots \tau_n$ die Abfahrtszeitpunkte von früh nach spät
- Führe Suchen nacheinander aus von spät bis früh aus
- **Self-Pruning**: Tentative Distanzen zwischen Anfragen **nicht** zurück setzen

Self-Pruning

$Q.clear()$
$d[x] \leftarrow \infty$ für alle x
for *Abfahrtszeitpunkte* τ_i *absteigend* **do**
  $d[s] \leftarrow \tau_i$
  $Q.add(s, \tau_i)$
  while $!Q.empty()$ **do**
    $u \leftarrow Q.deleteMin()$
    for *all edges* $e = (u, v) \in E$ **do**
      if $g_e(d[u]) < d[v]$ **then**
        $d[v] \leftarrow g_e(d[u])$
      if $v \in Q$ **then** $Q.decreaseKey(v, d[v])$
      else $Q.insert(v, d[v])$
  Extrahiere Ausgabe für τ_i

Self-Pruning

Äquivalente Formulierung

- Es seien $d_i(x)$ die tentativen Distanzen nach der Suche mit Startzeit τ_i
- Prune x in der Suche mit Abfahrt τ_i wenn

$$d_i(x) = d_{i+1}(x)$$

Self-Pruning

Äquivalente Formulierung

- Es seien $d_i(x)$ die tentativen Distanzen nach der Suche mit Startzeit τ_i
- Prune x in der Suche mit Abfahrt τ_i wenn

$$d_i(x) = d_{i+1}(x)$$

Interpretation

- Wenn ich später losfahren kann aber nicht früher beim Zwischenstop x ankomme, dann fahre ich später los
- Die jetztige Suche kann gestoppt werden, da die Suche mit einem späteren Startzeitpunkt den Weg finden wird

Fußwege und Umstiegszeiten

Fußwege

- Modelliert als Kante mit konstanter Reisezeitfunktion

Fußwege und Umstiegszeiten

Fußwege

- Modelliert als Kante mit konstanter Reisezeitfunktion

Umstiegszeiten

- Schwieriger
- Braucht Dummy-Knoten

Linien

Linien

- Eine Linie ist eine Menge von Trips
- Trips in einer Linie haben identische Stopsequenz
- Trips in einer Linie überholen sich nicht
- Linien haben eine Richtung

Linien

Linien

- Eine Linie ist eine Menge von Trips
- Trips in einer Linie haben identische Stopsequenz
- Trips in einer Linie überholen sich nicht
- Linien haben eine Richtung

Beispiele

- Tram 3 ist keine Linie
  - Stopsequenz ist je nach Richtung anders
  - Tram 3 Richtung Tivoli ist eine Linie
  - Tram 3 Richtung Heide ist eine andere Linie

Linien

Linien

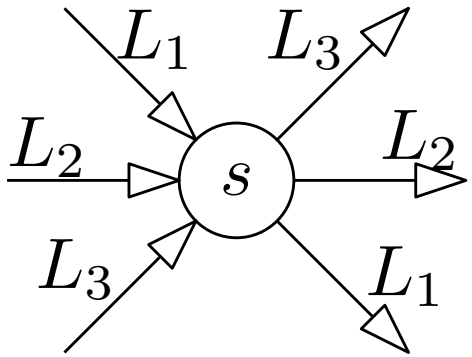
- Eine Linie ist eine Menge von Trips
- Trips in einer Linie haben identische Stopsequenz
- Trips in einer Linie überholen sich nicht
- Linien haben eine Richtung

Beispiele

- Tram 3 ist keine Linie
  - Stopsequenz ist je nach Richtung anders
  - Tram 3 Richtung Tivoli ist eine Linie
  - Tram 3 Richtung Heide ist eine andere Linie
- Tram 6 Richtung Hirtenweg is keine Linie
  - Hälfte der Bahnen fährt ab Rappenwörth
  - Andere Hälft ab Daxlanden
  - Stopsequenz der Tram 6 Richtung Hirtenweg variiert
  - “Jeder zweiter Zug des Tram 6 Richtung Hirtenweg” ist eine Linie

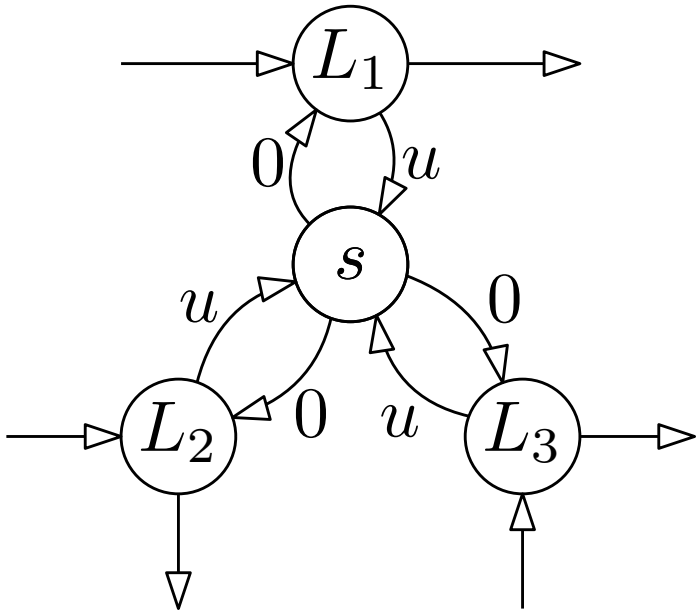
Umstiegszeiten

- Stops werden durch Stern-Graphen ersetzt
- Der Stern hat einen Strahl für jede Linie die am Stop hält
- Kanten von den Linien zum Zentrum sind mit Umstiegszeit gewichtet
- Kanten von der Mitte zu den Linien haben Gewicht 0



Umstiegszeiten

- Stops werden durch Stern-Graphen ersetzt
- Der Stern hat einen Strahl für jede Linie die am Stop hält
- Kanten von den Linien zum Zentrum sind mit Umstiegszeit gewichtet
- Kanten von der Mitte zu den Linien haben Gewicht 0



Mehrere Kriterien

- Mehrere Kriterien werden mit Bags behandelt
- Analog zu Dijkstras Algorithmus auf zeitexpandiertem Graph

RAPTOR

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

RAPTOR [DPW14]

- Algorithmus um Wegeanfragen in Fahrplänen zu beantworten
- Steht für: Round-Based Public Transit Optimized Router
- Benutzt keine Prioritätswarteschlange
- Arbeitet in Runden
- **Idee**: Bestimme in Runde *i* wohin man mit *i* oder weniger Umstiegen kommen kann
- (Wirrwar: im RAPTOR-Paper heißen die Linien Routen)

RAPTOR

Datenstruktur:

- Tentative Ankunftszeit $d(x)$ für jeden Stop x

Basisalgorithmus:

- Solange sich $d(x)$ an irgendeinem x ändert:
  - Fahre alle Linien ab

RAPTOR

Datenstruktur:

- Tentative Ankunftszeit $d(x)$ für jeden Stop x

Basisalgorithmus:

- Laufe alle Fußwege ab
- Solange sich $d(x)$ an irgendeinem x ändert:
  - Fahre alle Linien ab
  - Laufe alle Fußwege ab

Linie abfahren

- Organisiere jede Linien als zweidimensionale Tabelle
- Erste Dimension: Die Trip-Nummer
- Zweite Dimension: Die Stop-Sequenz
- Zur einfacheren Darstellung nehmen wir im folgenden an, dass Züge nur sehr kurz halten, d.h.,

$$a_{\text{arr\_time}} = b_{\text{dep\_time}}$$

für zwei aufeinanderfolgende Connections a und b

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:40	8:45	∞	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:40	8:45	∞	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:40	8:45	∞	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:30	8:45	∞	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:30	8:45	∞	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:30	8:45	9:50	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Stop	A	B	C	D	E	F
Trip 1	7:00	7:10	7:15	7:30	7:33	7:50
Trip 2	8:00	8:10	8:15	8:30	8:33	8:50
Trip 3	9:00	9:10	9:30	9:40	9:50	10:01
Trip 4	10:00	10:20	10:30	10:40	10:50	11:42
d(x)	∞	9:55	10:30	8:45	9:50	7:30

- Iteriere über Stops entlang der Linie
- Speichere frühest erreichbare Trip-ID
- An jedem Stop:
  - Wechsele in früheren Trip falls möglich
  - Verringere $d(x)$ falls nötig

Linie abfahren

Effizient Trip wechseln

- **Beobachtung**: Oft ist ein Tripwechel nicht nötig
- $\rightarrow$ frühesten Trip per binärer Suche bestimmen ist deswegen oft nicht sinnvoll
- **Besser**: Teste ob der aktuelle Trip noch der früheste ist
  - Falls nicht: Wechsel iterative in den nächsten Trip bis der früheste passende gefunden wurde

Column- oder row-major?

- Der Arbeitsspeicher eines Rechners ist ein großes lineares Array und kein 2D-Array
- Es gibt deswegen kein 2D-Cache

Column- oder row-major?

- Der Arbeitsspeicher eines Rechners ist ein großes lineares Array und kein 2D-Array
- Es gibt deswegen kein 2D-Cache
- Ein $n \times m$-Array wird als $n \cdot m$ großes lineares Array dargestellt
- Zelle (x, y) steht an Position $x \cdot m + y$ oder an Position $x + y \cdot n$
- Das ist der Unterschied zwischen column- und row-major

1	4	7	10
2	5	8	11
3	6	9	12

Column-major

- **Stark vereinfacht**: Cache lädt immer ganze Spalte oder Zeile
- Was ist hier besser?

Column- oder row-major?

- Der Arbeitsspeicher eines Rechners ist ein großes lineares Array und kein 2D-Array
- Es gibt deswegen kein 2D-Cache
- Ein $n \times m$-Array wird als $n \cdot m$ großes lineares Array dargestellt
- Zelle (x, y) steht an Position $x \cdot m + y$ oder an Position $x + y \cdot n$
- Das ist der Unterschied zwischen column- und row-major

1	2	3	4
5	6	7	8
9	10	11	12

Row-major

- **Stark vereinfacht**: Cache lädt immer ganze Spalte oder Zeile
- Was ist hier besser?

Column- oder row-major?

- Der Arbeitsspeicher eines Rechners ist ein großes lineares Array und kein 2D-Array
- Es gibt deswegen kein 2D-Cache
- Ein $n \times m$-Array wird als $n \cdot m$ großes lineares Array dargestellt
- Zelle (x, y) steht an Position $x \cdot m + y$ oder an Position $x + y \cdot n$
- Das ist der Unterschied zwischen column- und row-major

1	2	3	4
5	6	7	8
9	10	11	12

Row-major

- **Stark vereinfacht**: Cache lädt immer ganze Spalte oder Zeile
- Was ist hier besser?
- **Antwort**: Row-major
  - Wir schauen uns einen Eintrag in jeder Spalte an, aber nicht unbedingt in jeder Zeile

Optimierungen

Aktive Linien

- Eine Linie ist aktiv wenn sich $d(x)$ in der vorherigen Route für ein Stop x auf der Linie verändert hat

Optimierungen

Aktive Linien

- Eine Linie ist aktiv wenn sich $d(x)$ in der vorherigen Route für ein Stop x auf der Linie verändert hat

Erster Stop

- Speichere für jede Linie die früheste Einstiegsmöglichkeit
- Fahre Linie erst ab da ab

Optimierungen

Aktive Linien

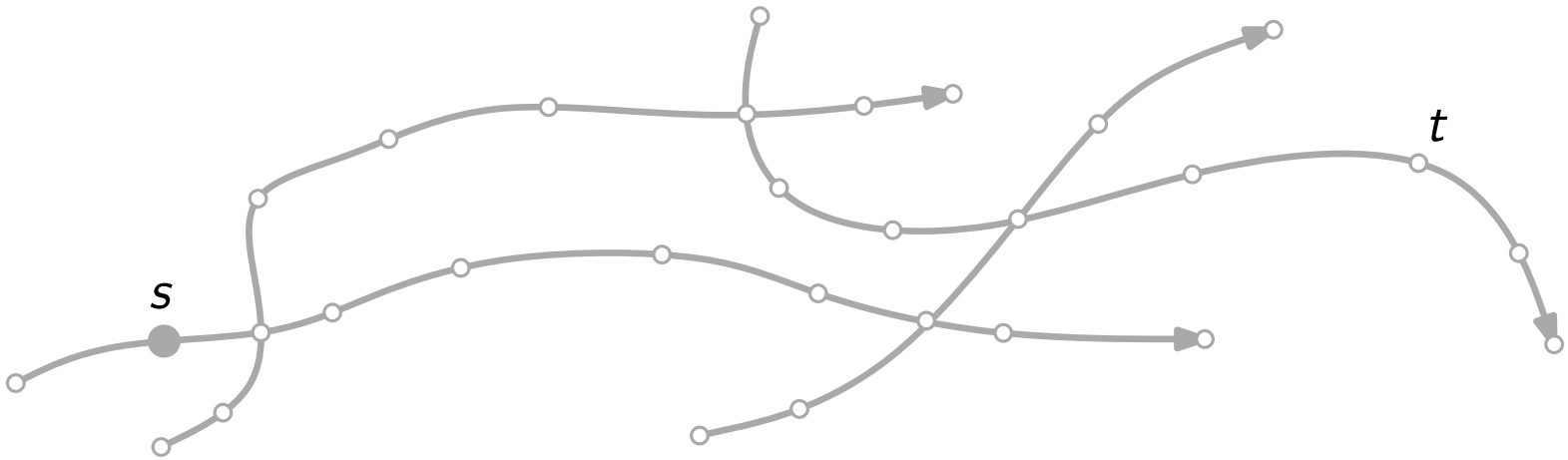
- Eine Linie ist aktiv wenn sich $d(x)$ in der vorherigen Route für ein Stop x auf der Linie verändert hat

Erster Stop

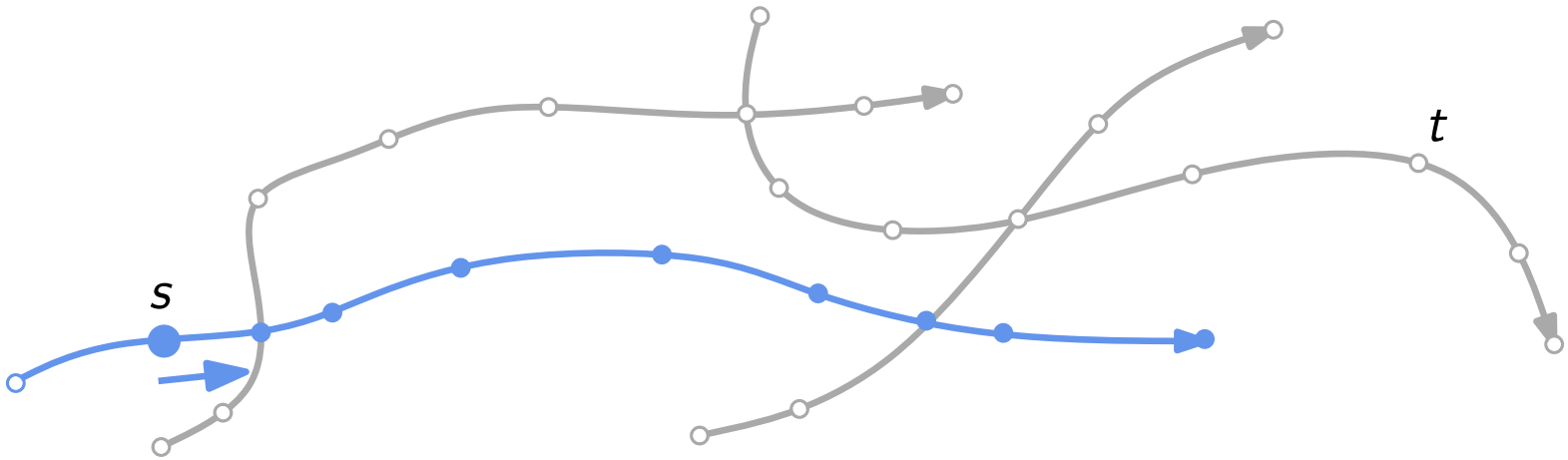
- Speichere für jede Linie die früheste Einstiegsmöglichkeit
- Fahre Linie erst ab da ab

- Benötigt eine Abbildung von Stop s auf (L_i, p_i) Paare
- L_i Linie durch s
- s ist der p_i-te Stop auf Linie L_i

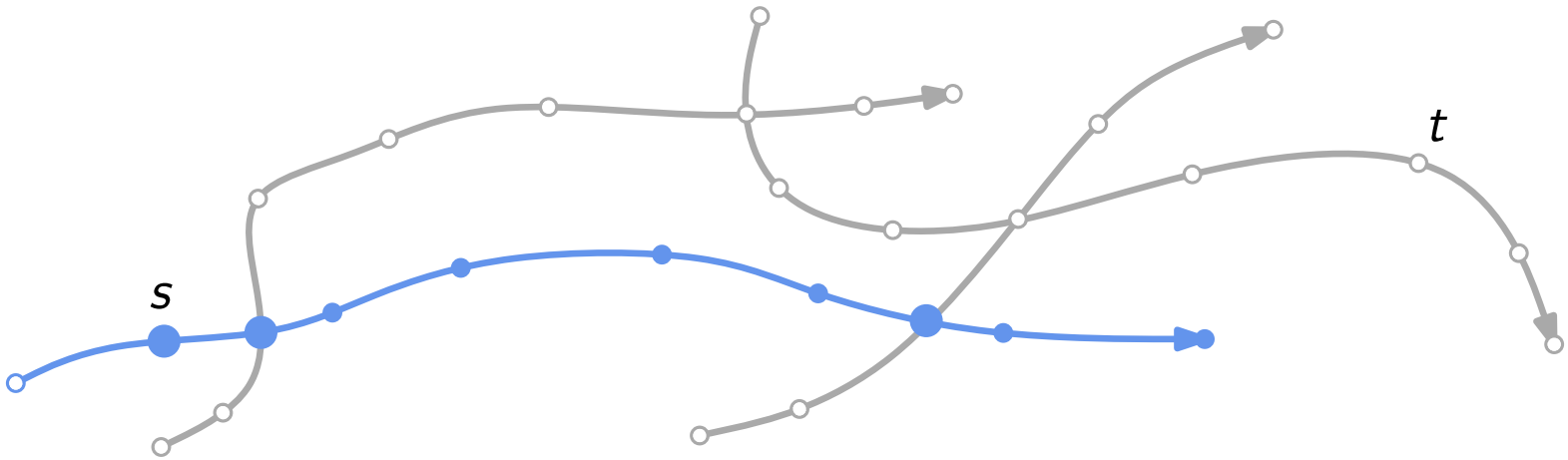
Optimierungen



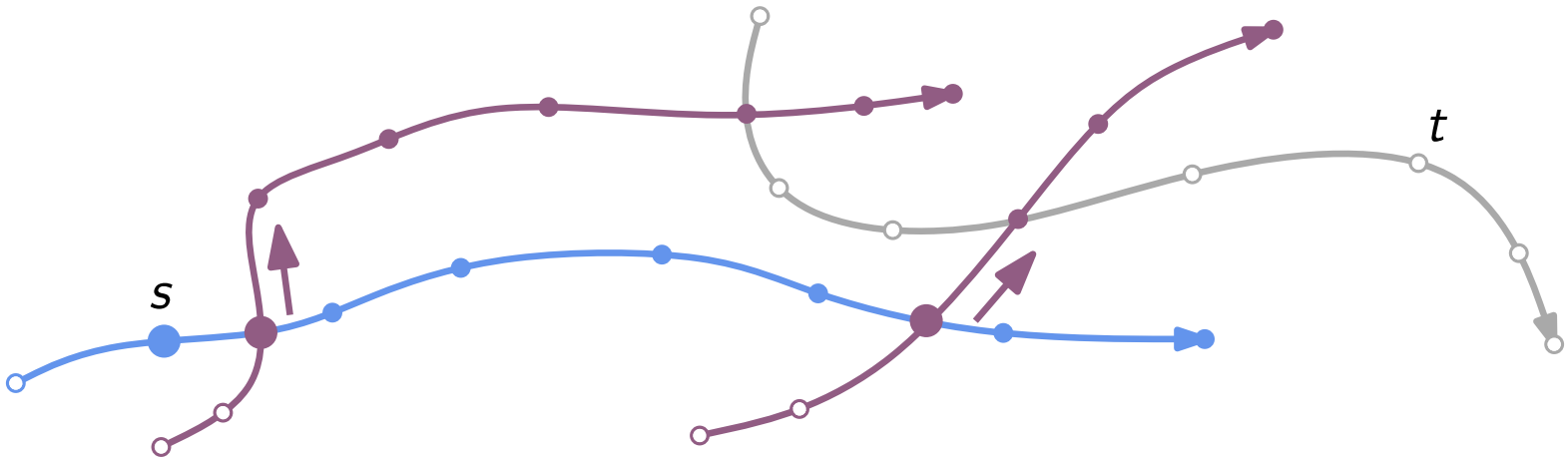
Optimierungen



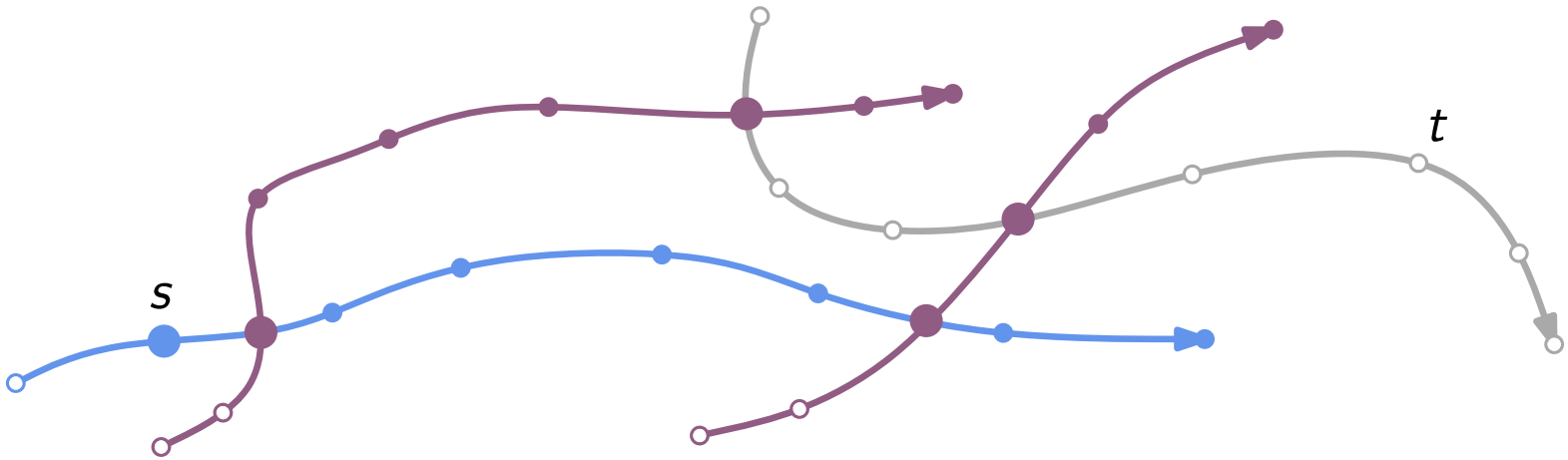
Optimierungen



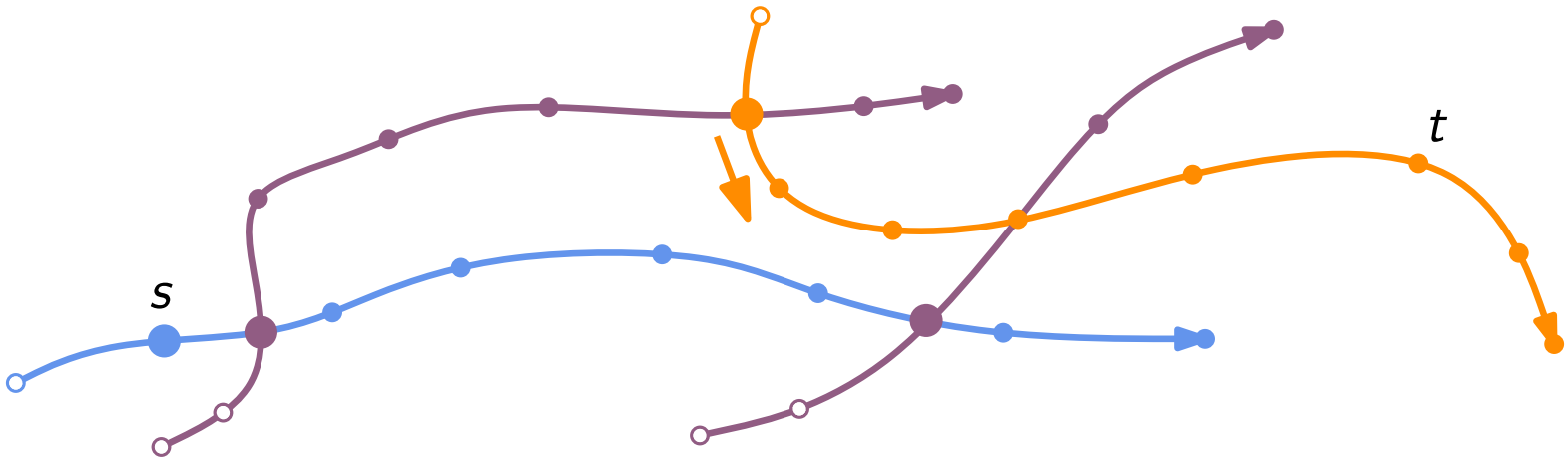
Optimierungen



Optimierungen



Optimierungen



$d[x] \leftarrow \infty$ für alle Stops x; $d[s] \leftarrow \tau$;
$A \leftarrow \emptyset$; $e[l] \leftarrow \infty$ für alle Linien k;
`/* Für diese Runde */`
$A' \leftarrow \emptyset$; $e'[l] \leftarrow \infty$ für alle Linien k;
`/* Für nächste Runde */`
for *Linien* (L, p) *durch* s **do** $A.add(L)$; $e[L] \leftarrow p$; ;

while $A \neq \emptyset$ **do**
  for *alle Linien* L *aus* A **do**
    Laufe L ab $e[L]$ ab;
    for *für alle Stops* x *wo* $d[x]$ *sich geändert hat* **do**
      for *Linien* (L', p) *durch* x **do**
        if $L \neq L'$ **then**
          $A.add(L')$;
          $e'[L'] \leftarrow \min\{e'[L'], p\}$;
    $e[L] \leftarrow \infty$;
    $A \leftarrow \emptyset$;
  Tausche e und e';
  Tausche A und A';

Erweiterungen

Profile:

- Self-Pruning lässt sich direkt anwenden

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Erweiterungen

Profile:

- Self-Pruning lässt sich direkt anwenden

Mehere Kriterien:

- Umstiege können im Pareto-Sinn optimiert werden indem nach jeder Runde $d[t]$ ausgegeben wird
- Weitere Kriteren bedürfen Bags

Das Londoner Netzwerk

Das vollständige Londoner Netzwerk

- Ein Dienstag.
- Beinhaltet Tube, Bus, DLR, Tram. . .
- 20 843 Stops,
- 2 225 Routen mit 133 011 Trips,
- 5 132 672 einzelne Abfahrten pro Tag.

Das Londoner Netzwerk

Das vollständige Londoner Netzwerk

- Ein Dienstag.
- Beinhaltet Tube, Bus, DLR, Tram. . .
- 20 843 Stops,
- 2 225 Routen mit 133 011 Trips,
- 5 132 672 einzelne Abfahrten pro Tag.

Experimente: 10 000 zufällige s–t-Anfragen.

Vergleich der Algorithmen

(Hardware: Intel Xeon X5680 mit 3.33 GHz und 96 GiB DDR3-1333 RAM)

Algorithm	Ar	R	Tr	Fz	Rounds	Journeys	[ms]
TD-Dijkstra	●	○	○	○	—	0.9	14.2
RAPTOR	●	○	●	○	8.4	1.9	7.3
TD-Dijkstra+Bags [DMS08]	●	○	●	○	—	1.9	67.2
RAPTOR+Bags	●	○	●	●	10.8	9.0	107.4
TD-Dijkstra+Bags [DMS08]	●	○	●	●	—	9.0	399.5
RAPTOR+Bags	●	●	●	○	9.5	16.3	259.8
RAPTOR+SP	●	●	●	○	138.5	16.3	87.0
TD-Dijkstra+SP [DKP12]	●	●	○	○	—	7.8	183.6

(**Ar**: Arrival Time, **R**: Range, **Tr**: Transfers, **Fz**: Fare Zones, **SP**: Profile mit Self-Pruning für 2h Intervall)

Literatur I

Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck.

Route planning in transportation networks.

Technical Report abs/1504.05140, ArXiv e-prints, 2016.

Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann.

Accelerating time-dependent multi-criteria timetable information is harder than expected.

In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, OpenAccess Series in Informatics (OASIcs), 2009.

Daniel Delling, Bastian Katz, and Thomas Pajor.

Parallel computation of best connections in public transportation networks.

ACM Journal of Experimental Algorithmics, 17(4):4.1–4.26, July 2012.

Literatur II


Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee.
Multi-criteria shortest paths in time-dependent train networks.
In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.


Daniel Delling, Thomas Pajor, and Renato F. Werneck.
Round-based public transit routing.
Transportation Science, 49(3):591–604, 2014.


H V Jagadish.
A compression technique to materialize transitive closure.
ACM Transactions on Database Systems, 15(4):558–598, December 1990.


Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis.
Efficient models for timetable information in public transportation systems.
ACM Journal of Experimental Algorithmics, 12(2.4):1–39, 2008.