Traffic Assignment in Transportation Networks

Dorothea Wagner - September 12, 2019
Shortest-Path Applications

Important applications, e.g.,

- Navigation systems for cars
- Apple Maps, Google Maps, Bing Maps, OpenStreetMap, ...
- Timetable information
- Transportation and urban planning
Core Problem

Request:
- Find the best connection in a transportation network w.r.t. some metric

Idea:
- Network as graph \( G = (V, E) \)
- Edge weights are according to metric
- Shortest paths in \( G \) equal best connections
- Classic problem (Dijkstra 1959)

Problems:
- Transport networks are huge
- Dijkstra too slow (> 1 second)
Speed-Up Techniques

Observations:
- Dijkstra visits all nodes closer than the target
- Unnecessary computations
- Many requests in a hardly changing network

Idea:
- Two-phase algorithm:
  - Offline: compute additional data during preprocessing
  - Online: speed-up query with this data
- 3 criteria: preprocessing time and space, speed-up over Dijkstra
Showpiece of Algorithm Engineering

Falsifiable Hypotheses

Design

Experiment

Analyze

Implement
Showpiece of Algorithm Engineering

Falsifiable Hypotheses

Design

Experiment

Analyze

Implement

Performance guarantees & practical algorithms

Realistic scenarios

Real-world data
Many techniques tuned for continent-sized road networks:

- Reach (2004, 2007)

Timetable information:

- Transfer Patterns (2010, 2016)
- RAPTOR (2013)
- Trip-Based Public Transit Routing (2015, 2016)

Survey on “Route Planning in Transportation Networks” (Bast et al. 2016)
Next Steps

State of the art:

- Portfolio of fast shortest-path algorithms
- Different trade-offs between:
  - Preprocessing time and space
  - Query time
  - Implementation complexity
  - Versatility

⇒ Leverage these in transportation applications

Case study in this talk: traffic assignment

- Major problem in transport and urban planning
- Goal: analyze utilization of roads, trains, buses
- Requires many shortest-path computations
Joint Work with

Valentin Buchhold

Moritz Baum

Peter Sanders

Jonas Sauer

Ben Strasser

Tobias Zündorf
In Road Networks
Traffic Assignment in Road Networks

**Input:**
- Urban road network
- Set of origin–destination pairs

**Output:**
- Equilibrium flow pattern
- i.e. flow on each segment
Traffic Assignment in Road Networks

**Input:**
- Urban road network
- Set of origin–destination pairs

**Output:**
- Equilibrium flow pattern
- i.e. flow on each segment

**Assumption:**
- Motorists choose path with *minimum travel time*...
- ... but travel time changes with flow (*congestion*)
Relation between Flow and Travel Time

travel time $t(f)$

free-flow travel time $t^0$

capacity $f_{\text{max}}$

link flow $f$

Link cost function: $t(f) = t^0 \left( 1 + \alpha \left( \frac{f}{f_{\text{max}}} \right)^\beta \right)$

(Bureau of Public Roads 1964)
**Solution Algorithms**

---

**Link-based methods:**

- Represent solution by link flows $f_e$ (flow on link $e$)
- Feasible-direction methods
  - Start from initial solution
  - Generate feasible direction of descent
  - Shift current solution along descent direction
Solution Algorithms

Path-based methods:

- Represent solution by path flows $F_k$ (flow on path $k$)
- Maintain set $K_p^+$ of promising paths between each O-D pair $p$
- In each iteration, process O-D pairs $p$ one by one
  1. Update $K_p^+$ (remove unpromising paths, insert new promising paths)
  2. Equilibrate $K_p^+$ (shift flow between paths in $K_p^+$)
Solution Algorithms

Bush-based methods:
- Represent solution by origin flows $f_{eo}$ (flow on link $e$ that originates at origin $o$)
- Maintain bush $B_o$ for each origin $o$
- $B_o$ is DAG that comprises promising paths from $o$ to all destinations
- In each iteration, process origins $o$ one by one
  1. Update $B_o$ (remove zero-flow links, insert new links giving rise to cheaper paths)
  2. Equilibrate $B_o$ (shift flow on $B_o$)
Frank-Wolfe Algorithm

- Represents solution (before iteration $i$) by link flows $f_i = \left( f_1^i, \ldots, f_{|E|}^i \right)$
- Main subroutine is all-or-nothing (AON) assignment
  - Process O-D pairs one by one
  - Assign one flow unit to each link on shortest path

```
FrankWolfe
1 Generate initial solution by performing free-flow AON assignment
2 while convergence criterion is not satisfied do
3     Update link costs based on current link flows
4     Perform AON assignment based on current link costs, yielding $y_i$
5     Let descent direction $d_i$ be $y_i - f_i$
6     Determine how far current solution must be moved along descent direction
7     Move current solution along descent direction, i.e., set $f_{i+1} = f_i + \lambda_i d_i$
```
Frank-Wolfe Algorithm

- Represents solution (before iteration $i$) by link flows $f^i = (f^i_1, \ldots, f^i_{|E|})$
- Main subroutine is all-or-nothing (AON) assignment
  - Process O-D pairs one by one
  - Assign one flow unit to each link on shortest path

---

FrankWolfe

1. Generate initial solution by performing free-flow AON assignment
2. while convergence criterion is not satisfied do
   3. Update link costs based on current link flows
   4. Perform AON assignment based on current link costs, yielding $y^i$
   5. Let descent direction $d^i$ be $y^i - f^i$
   6. Determine how far current solution must be moved along descent direction
   7. Move current solution along descent direction, i.e., set $f^{i+1} = f^i + \lambda^i d^i$

⇒ Benefits particularly from recent advances in route planning
State of the Art in Routing \cite{Bast:2016}

![Graph showing the state of the art in routing algorithms, comparing preprocessing time and query time.](image)

- **Arc Flags**: HH, HH*, SHARC, TNR with Arc Flags
- **Table Lookup**: PHAST
- **Hub Labels**: Hub Labels
- **Dijkstra’s Algorithm**: Bidirectional Search
- **Preprocessing time [min]**
- **Query time [ms]**
Speedup Techniques

Two-phase:
- Preprocessing (slow): compute additional data
- Query (fast): answer $s-t$ queries using data from preprocessing
Speedup Techniques

Two-phase:
- Preprocessing (slow): compute additional data
- Query (fast): answer $s-t$ queries using data from preprocessing

Three-phase:
- Preprocessing (slow): compute additional weight-independent data
- Customization (reasonably fast): introduce weights
- Query (fast): answer $s-t$ queries using data from preprocessing and customization
Shortest-Path Algorithm for Frank-Wolfe?

Requirements:
- Fast point-to-point shortest-path computations
- Easy retrieval of actual shortest paths (not only distances)
- Edge weights change in each iteration → dynamic scenario
Shortest-Path Algorithm for Frank-Wolfe?

Requirements:
- Fast point-to-point shortest-path computations
- Easy retrieval of actual shortest paths (not only distances)
- Edge weights change in each iteration → dynamic scenario

Best fit: customizable contraction hierarchies
- Uses metric-independent nested dissection order
- Customization: compute shortcut weights
- Elimination tree query (requires no queue)
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:

![Graph with vertices and edges labeled with weights]
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Preprocessing:
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices at end of order

![Graph](image-url)
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:

- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order

![Graph with nodes 0, 1, 2, 3, 4, 5, 6, 7]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- Partitioning: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Preprocessing:

- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices at **end** of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order

![Diagram of a graph with vertices and edges labeled 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, with vertex 6 highlighted.]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order

![Graph Diagram]

1. Recursive partitioning
2. Place separator vertices
3. Create shortcut edges

---

Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order

![Graph Diagram]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order

![Graph with nodes and edges]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices *at end* of order
Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices at *end* of order

![Graph Diagram]

- **Contraction**: shortcut vertices in this order
  - Temporarily remove vertex from graph
  - Add shortcut edges between its neighbors

---

---
Preprocessing:

- **Partitioning**: compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices *at end* of order

![Diagram of graph partitioning](image-url)
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Preprocessing:

- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
- **Contraction**: shortcut vertices in this order
  - Temporarily remove vertex from graph
  - Add shortcut edges between its neighbors
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices *at end* of order
- **Contraction**: shortcut vertices in this order
  - Temporarily *remove* vertex from graph
  - Add *shortcut edges* between its neighbors
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively **split** graph into two parts
  - Place separator vertices **at end** of order
- **Contraction**: shortcut vertices in this order
  - Temporarily **remove** vertex from graph
  - Add **shortcut edges** between its neighbors
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices at *end* of order
- **Contraction**: shortcut vertices in this order
  - Temporarily *remove* vertex from graph
  - Add *shortcut edges* between its neighbors
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices *at end* of order
- **Contraction**: shortcut vertices in this order
  - Temporarily *remove* vertex from graph
  - Add *shortcut edges* between its neighbors
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning:** compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices *at end* of order
- **Contraction:** shortcut vertices in this order
  - Temporarily *remove* vertex from graph
  - Add *shortcut edges* between its neighbors
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Preprocessing:
- **Partitioning:** compute nested dissection order
  - Recursively *split* graph into two parts
  - Place separator vertices *at end* of order
- **Contraction:** shortcut vertices in this order
  - Temporarily *remove* vertex from graph
  - Add *shortcut edges* between its neighbors
Preprocessing:
- **Partitioning**: compute nested dissection order
  - Recursively split graph into two parts
  - Place separator vertices at end of order
- **Contraction**: shortcut vertices in this order
  - Temporarily remove vertex from graph
  - Add shortcut edges between its neighbors

![Graph Example](image-url)
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:

- Assign orig edges their input weight
- Process edges in bottom-up fashion
- Enumerate all lower triangles
- Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks

Graph:
- Node 1 connected to nodes 2, 4, and 3
- Node 2 connected to nodes 1 and 3
- Node 3 connected to node 2
- Node 4 connected to node 1
- Node 5 connected to node 6
- Node 6 connected to nodes 5, 7, and 1
- Node 7 connected to node 6
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks

Diagram:

```
10 5 0
10 1 8
10 2 7
10 3 5
```

Diagram nodes and edges with weights.
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Diagram:
```
  7
 /\  \
/   \ /
2-----6-----7
|      |
|      |
1-----5-----4
|  8   |
|      |
|      |
3-----3
```

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
  - Only relax edges to higher ranks
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

```
10
5
0

6
5
1

2
3
2

1

4

5

8
7

3
```
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Diagram:

- Nodes: 1, 2, 3, 4, 5, 6, 7, 8
- Edges with weights: 2, 3, 5, 7, 8, 10
- Red line indicates a specific edge weight of 1
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

![Graph with edge weights]

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

![Graph diagram]

- Nodes represent network entities
- Edges represent network connections with weights
- Red edges indicate contractions

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks

D. Wagner - Traffic Assignment in Transportation Networks
Institute of Theoretical Informatics
Research Group Algorithmics
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

![Graph Diagram]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

![Diagram of a contraction hierarchy with weights and ranks highlighted.](image-url)
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies

(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Customization:
- Assign orig edges their input weight
- Process edges in bottom-up fashion
  - Enumerate all lower triangles
  - Check if it improves edge weight

Query algorithm:
- Bidirectional Dijkstra
- Only relax edges to higher ranks
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree

```
1. Compute LCA x of s and t
2. Scan all vertices on s–x path
3. Scan all vertices on t–x path
4. Scan all vertices on x–r path
5. Reset labels on s–r and t–r path
```

\[ \mu = \infty \]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:

\[ \mu = \infty \]
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA $x$ of $s$ and $t$
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA \( x \) of \( s \) and \( t \)
2. Scan all vertices on \( s-x \) path
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA $x$ of $s$ and $t$
2. Scan all vertices on $s$–$x$ path

\[ \mu = \infty \]
**Customizable Contraction Hierarchies**

(Dibbelt et al. 2016)

**Alternative query algorithm:**
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

**Elimination tree search:**
1. Compute LCA $x$ of $s$ and $t$
2. Scan all vertices on $s$–$x$ path
3. Scan all vertices on $t$–$x$ path

$$\mu = \infty$$
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA \( x \) of \( s \) and \( t \)
2. Scan all vertices on \( s \rightarrow x \) path
3. Scan all vertices on \( t \rightarrow x \) path
Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA $x$ of $s$ and $t$
2. Scan all vertices on $s$–$x$ path
3. Scan all vertices on $t$–$x$ path

$\mu = \infty$
Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA $x$ of $s$ and $t$
2. Scan all vertices on $s$–$x$ path
3. Scan all vertices on $t$–$x$ path
4. Scan all vertices on $x$–$r$ path

$\mu = 13$
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA $x$ of $s$ and $t$
2. Scan all vertices on $s$–$x$ path
3. Scan all vertices on $t$–$x$ path
4. Scan all vertices on $x$–$r$ path
Customizable Contraction Hierarchies
(Dibbelt et al. 2016)

Alternative query algorithm:
- Based on elimination tree
- Elimination tree efficiently encodes CCH search space of each vertex

Elimination tree search:
1. Compute LCA $x$ of $s$ and $t$
2. Scan all vertices on $s$–$x$ path
3. Scan all vertices on $t$–$x$ path
4. Scan all vertices on $x$–$r$ path
5. Reset labels on $s$–$r$ and $t$–$r$ path

\[ \mu = 13 \]
Faster Batched One-to-One Shortest Paths
(Buchhold et al. 2018)

Observation:
- Processing similar OD-pairs in succession improves locality
- Size of sym. diff between search spaces of u and v is equal to u–v distance in elimination tree
Faster Batched One-to-One Shortest Paths
(Buchhold et al. 2018)

Observation:
- Processing similar OD-pairs in succession improves locality
- Size of sym. diff between search spaces of \( u \) and \( v \) is equal to \( u-v \) distance in elimination tree

Idea:
- Partition elimination tree into few cells with bounded diameter
- Assign IDs according to DFS order
- Reorder OD-pairs by src and dst cell
Centralized Elimination Tree Searches

(Buchhold et al. 2018)

Bundling together multiple runs:

- $k$ distance labels for each vertex
- $i$-th label is distance from $i$-th src
- Relaxation updates all labels at once

**Diagram**

![Diagram of an elimination tree with labeled vertices and edges.](attachment:image.png)

\[ \mu_0 = \infty \]
Centralized Elimination Tree Searches
(Buchhold et al. 2018)

Bundling together multiple runs:
- $k$ distance labels for each vertex
- $i$-th label is distance from $i$-th src
- Relaxation updates all labels at once

\[
\mu_0 = \infty \\
\mu_1 = \infty
\]
Exploiting Parallelism  
*(Buchhold et al. 2018)*

**Instruction-level parallelism:**
- 128-/256-bit registers
- Basic operations on multiple data items *simultaneously*
- We use *SSE* and *AVX* instructions

**Core-level parallelism:**
- SP computations are *independent*
- Assign OD-pairs to *distinct* cores
- Cumulate flow units locally, aggregate after computing all paths
## Single-Threaded Traffic Assignment

<table>
<thead>
<tr>
<th>algo</th>
<th>sorted</th>
<th>$k$</th>
<th>SIMD</th>
<th>S-morn</th>
<th>S-even</th>
<th>S-day</th>
<th>L-peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dij</td>
<td>o</td>
<td>1</td>
<td>–</td>
<td>5753.22</td>
<td>8239.57</td>
<td>106687.46</td>
<td>1648.98</td>
</tr>
<tr>
<td>Bi-Dij</td>
<td>o</td>
<td>1</td>
<td>–</td>
<td>2459.27</td>
<td>3265.95</td>
<td>44078.13</td>
<td>907.85</td>
</tr>
<tr>
<td>CH</td>
<td>o</td>
<td>1</td>
<td>–</td>
<td>90.89</td>
<td>120.83</td>
<td>1048.10</td>
<td>86.58</td>
</tr>
<tr>
<td>CCH</td>
<td>o</td>
<td>1</td>
<td>–</td>
<td>41.50</td>
<td>55.02</td>
<td>698.16</td>
<td>49.01</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>1</td>
<td>–</td>
<td>26.98</td>
<td>35.45</td>
<td>372.34</td>
<td>32.23</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>4</td>
<td>–</td>
<td>31.73</td>
<td>42.10</td>
<td>452.73</td>
<td>40.03</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>4</td>
<td>SSE</td>
<td>18.29</td>
<td>23.95</td>
<td>230.18</td>
<td>20.47</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>8</td>
<td>–</td>
<td>34.39</td>
<td>45.32</td>
<td>472.77</td>
<td>42.69</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>8</td>
<td>SSE</td>
<td>17.45</td>
<td>22.74</td>
<td>211.26</td>
<td>18.65</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>8</td>
<td>AVX</td>
<td>15.30</td>
<td>19.94</td>
<td>175.72</td>
<td>15.89</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>16</td>
<td>AVX</td>
<td>14.46</td>
<td>18.68</td>
<td>153.06</td>
<td>13.52</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>32</td>
<td>AVX</td>
<td>14.12</td>
<td>18.20</td>
<td>132.54</td>
<td>11.44</td>
</tr>
<tr>
<td>CCH</td>
<td>•</td>
<td>64</td>
<td>AVX</td>
<td>18.83</td>
<td>24.27</td>
<td>160.51</td>
<td>13.07</td>
</tr>
</tbody>
</table>
## Multi-Threaded Traffic Assignment

<table>
<thead>
<tr>
<th>algo</th>
<th>cores</th>
<th>S-morn</th>
<th></th>
<th>S-day</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cust</td>
<td>query</td>
<td>total</td>
<td>cust</td>
<td>query</td>
</tr>
<tr>
<td>CH</td>
<td>1</td>
<td>36.12</td>
<td>54.06</td>
<td>90.89</td>
<td>49.52</td>
<td>997.60</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>36.46</td>
<td>3.95</td>
<td>40.48</td>
<td>50.24</td>
<td>67.66</td>
</tr>
<tr>
<td>CCH</td>
<td>1</td>
<td>1.77</td>
<td>11.77</td>
<td>14.12</td>
<td>2.40</td>
<td>129.34</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.13</td>
<td>6.58</td>
<td>8.02</td>
<td>1.54</td>
<td>68.96</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.61</td>
<td>3.85</td>
<td>4.62</td>
<td>0.83</td>
<td>36.42</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.32</td>
<td>2.53</td>
<td>2.94</td>
<td>0.43</td>
<td>19.28</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0.28</td>
<td>2.09</td>
<td>2.44</td>
<td>0.38</td>
<td>13.42</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.38</td>
<td>1.99</td>
<td>2.43</td>
<td>0.42</td>
<td>10.60</td>
</tr>
</tbody>
</table>
Traffic Assignment in Road Networks

Summary:
- Traffic assignment in only 2.4 sec.
- Makes interactive apps practical
  - Road traffic centers
  - Monitoring and controlling road traffic in real time

Ongoing and future research:
- Sample demand in early iterations
- Realistic demand data generation
- Time-dependent travel-time profiles
In Timetable Networks
Assignments for Timetable Networks

Objective:
- Determine the utilization of vehicles in the network
- For optimizing existing networks
- For planning new lines

Data Basis:
- Set of O-D pairs (as before)
- Timetable network
  - Consisting of lines and stops
  - Not represented as graph
Timetable networks

Network components:
- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips
Timetable networks

Network components:
- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips
Network components:
- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips
Timetable networks

Network components:
- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips

![Diagram of a timetable network with stops and connections marked at 8:30 and 9:00]
Timetable networks

Network components:
- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips

Trip 1:
Timetable networks

Network components:

- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips
Timetable networks

Network components:
- Set of stops (representing stops, stations, platforms, ...)
- Set of elementary connections
- Partition of the set of connections into trips

Trip 3:
(different line)
Route Planning on Timetable Networks

Types of Algorithms:

- Graph based
  - Transform timetable into time-dependent or time-expanded graph
  - Graph algorithms are applicable
  - But: Graphs get huge, special structure of timetable is lost

- Timetable based
  - Operate directly on timetable
  - Exploit knowledge of the network (chronological order, repetition of trips, ...)

Special Algorithms for timetables:
- RAPTOR
- CSA
- Transfer Patterns
- Trip-Based
Route Planning on Timetable Networks

Types of Algorithms:

- Graph based
  - Transform timetable into time-dependent or time-expanded graph
  - Graph algorithms are applicable
  - But: Graphs get huge, special structure of timetable is lost

- Timetable based
  - Operate directly on timetable
  - Exploit knowledge of the network (chronological order, repetition of trips, ...)

Special Algorithms for timetables:

- RAPTOR
- CSA
- Transfer Patterns
- Trip-Based
Route Planning on Timetable Networks

Types of Algorithms:
- Graph based
  - Transform timetable into time-dependent or time-expanded graph
  - Graph algorithms are applicable
  - But: Graphs get huge, special structure of timetable is lost
- Timetable based
  - Operate directly on timetable
  - Exploit knowledge of the network (chronological order, repetition of trips, ...)

Special Algorithms for timetables:
- RAPTOR
- CSA
- Transfer Patterns
- Trip-Based
Methods for Public Transit Traffic Assignments?

Requirements:

- Fast shortest-path computations
- Easy retrieval of actual shortest paths
- Realistic assessment of a journey's quality: Perceived Travel Time
  - Time in vehicle
  - Time spent waiting
  - Number of transfers
  - Delay robustness
  - ...

Best fit: CSA respectively MEAT

Fast one-to-many queries

Natural integration of delay robustness
Methods for Public Transit Traffic Assignments?

Requirements:
- Fast shortest-path computations
- Easy retrieval of actual shortest paths
- Realistic assessment of a journey's quality: Perceived Travel Time
  - Time in vehicle
  - Time spent waiting
  - Number of transfers
  - Delay robustness
  - ...

Best fit: CSA respectively MEAT
- Fast one-to-many queries
- Natural integration of delay robustness
Connection Scan (CSA) *(Dibbelt et al. 2013, 2018)*

**Basic idea:**
- Maintain earliest arrival times per stop
- Sort connections by their departure time
- Scan through the connections once

**Special properties:**
- Does not require a queue
- Uses chronological order of connections instead
Connection Scan (CSA) (Dibbelt et al. 2013, 2018)

Given: Timetable as array of connections, departure stop, departure time

Objective: Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>...</td>
</tr>
</tbody>
</table>

Connections sorted by departure time

| dep. stop | arr. stop | dep. time | arr. time | dep. stop | arr. stop | dep. time | arr. time | dep. stop | arr. stop | dep. time | arr. time | ... |
**Connection Scan (CSA) (Dibbelt et al. 2013, 2018)**

**Given:** Timetable as array of connections, departure stop, departure time

**Objective:** Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>...</td>
</tr>
</tbody>
</table>

Connections sorted by departure time

<table>
<thead>
<tr>
<th>from 1</th>
<th>to 3</th>
<th>9:00</th>
<th>9:25</th>
<th>from 3</th>
<th>to 4</th>
<th>9:15</th>
<th>9:45</th>
<th>from 3</th>
<th>to 4</th>
<th>9:40</th>
<th>9:55</th>
<th>...</th>
</tr>
</thead>
</table>
Connection Scan (CSA)  

(Dibbelt et al. 2013, 2018)

**Given:** Timetable as array of connections, departure stop, departure time

**Objective:** Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>$\infty$</td>
<td>8:00</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Connections sorted by departure time:

<table>
<thead>
<tr>
<th>from 1</th>
<th>to 3</th>
<th>9:00</th>
<th>9:25</th>
<th>from 3</th>
<th>to 4</th>
<th>9:15</th>
<th>9:45</th>
<th>from 3</th>
<th>to 4</th>
<th>9:40</th>
<th>9:55</th>
<th>...</th>
</tr>
</thead>
</table>

from 1 to 3
from 3 to 4
from 3 to 4

$\infty$ $\infty$ $\infty$
Connection Scan (CSA) (Dibbelt et al. 2013, 2018)

Given: Timetable as array of connections, departure stop, departure time

Objective: Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>$\infty$</td>
<td>8:00</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>...</td>
</tr>
</tbody>
</table>

Connections sorted by departure time

- from 1 to 3
- 9:25

- from 3 to 4
- 9:45

- from 3 to 4
- 9:55

...
**Connection Scan (CSA)** (Dibbelt et al. 2013, 2018)

**Given:** Timetable as array of connections, departure stop, departure time

**Objective:** Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>$+\infty$</td>
<td>8:00</td>
<td>$+\infty$</td>
<td>9:25</td>
<td>$+\infty$</td>
</tr>
</tbody>
</table>

Connections sorted by departure time:
- from 1 to 3: 9:00, 9:25
- from 3 to 4: 9:15, 9:45
- from 3 to 4: 9:40, 9:55
Connection Scan (CSA) (Dibbelt et al. 2013, 2018)

Given: Timetable as array of connections, departure stop, departure time
Objective: Earliest arrival time at the destination

Given:

<table>
<thead>
<tr>
<th>stop-id</th>
<th>earliest arrival time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>1</td>
<td>8:00</td>
</tr>
<tr>
<td>2</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>3</td>
<td>9:25</td>
</tr>
<tr>
<td>4</td>
<td>$+\infty$</td>
</tr>
</tbody>
</table>

Objective: Earliest arrival time at the destination

Connections sorted by departure time

<table>
<thead>
<tr>
<th>from 1</th>
<th>to 3</th>
<th>9:00</th>
<th>9:25</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>to 4</td>
<td>9:15</td>
<td>9:45</td>
</tr>
<tr>
<td>from 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>from 4</td>
<td>9:40</td>
<td>9:55</td>
<td></td>
</tr>
</tbody>
</table>
Connection Scan (CSA) \cite{Dibbeltetal2013,Dibbeltetal2018}

**Given:** Timetable as array of connections, departure stop, departure time

**Objective:** Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>+∞</td>
<td>8:00</td>
<td>+∞</td>
<td>9:25</td>
<td>+∞</td>
<td>...</td>
</tr>
</tbody>
</table>

Connections sorted by departure time

<table>
<thead>
<tr>
<th>from 1</th>
<th>to 3</th>
<th>9:00</th>
<th>9:25</th>
<th>from 3</th>
<th>to 4</th>
<th>9:15</th>
<th>9:45</th>
<th>from 3</th>
<th>to 4</th>
<th>9:40</th>
<th>9:55</th>
<th>...</th>
</tr>
</thead>
</table>
Connection Scan (CSA) (Dibbelt et al. 2013, 2018)

**Given:** Timetable as array of connections, departure stop, departure time

**Objective:** Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>earliest arrival time</td>
<td>+∞</td>
<td>8:00</td>
<td>+∞</td>
<td>9:25</td>
<td>9:55</td>
</tr>
</tbody>
</table>

Connections sorted by departure time:

<table>
<thead>
<tr>
<th>from 1</th>
<th>to 3</th>
<th>9:00</th>
<th>9:25</th>
<th>from 3</th>
<th>to 4</th>
<th>9:15</th>
<th>9:45</th>
<th>from 3</th>
<th>to 4</th>
<th>9:40</th>
<th>9:55</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Connection Scan (CSA) (Dibbelt et al. 2013, 2018)

**Given:** Timetable as array of connections, departure stop, departure time

**Objective:** Earliest arrival time at the destination

<table>
<thead>
<tr>
<th>stop-id</th>
<th>earliest arrival time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>1</td>
<td>8:00</td>
</tr>
<tr>
<td>2</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>3</td>
<td>9:25</td>
</tr>
<tr>
<td>4</td>
<td>9:55</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

High efficiency since modern processors are optimized for linear memory scans
Minimum Expected Arrival Time (MEAT)
(Dibbelt et al. 2013, 2018)

Extension of CSA:
- Can handle probabilistic delays of public transit vehicles
- Enables delay robust journey planning
- Computes expected arrival times instead of absolute arrival times
Minimum Expected Arrival Time (MEAT)  
(Dibbelt et al. 2013, 2018)

Extension of CSA:
- Can handle probabilistic delays of public transit vehicles
- Enables delay robust journey planning
- Computes expected arrival times instead of absolute arrival times

Interpretation of the result:
- Consider all journeys that contribute to the expected value
- These journeys represent fall back plans:
Minimum Expected Arrival Time (MEAT)  
(Dibbelt et al. 2013, 2018)

Extension of CSA:
- Can handle probabilistic delays of public transit vehicles
- Enables delay robust journey planning
- Computes expected arrival times instead of absolute arrival times

Interpretation of the result:
- Consider all journeys that contribute to the expected value
- These journeys represent fall back plans:
Minimum Expected Arrival Time (MEAT)
(Dibbelt et al. 2013, 2018)

Extension of CSA:
- Can handle probabilistic delays of public transit vehicles
- Enables delay robust journey planning
- Computes expected arrival times instead of absolute arrival times

Interpretation of the result:
- Consider all journeys that contribute to the expected value
- These journeys represent fall back plans:
Minimum Expected Arrival Time (MEAT)
(Dibbelt et al. 2013, 2018)

Extension of CSA:
- Can handle probabilistic delays of public transit vehicles
- Enables delay robust journey planning
- Computes expected arrival times instead of absolute arrival times

Interpretation of the result:
- Consider all journeys that contribute to the expected value
- These journeys represent fall back plans:
Perceived Arrival Time (PAT)

Further extending CSA:

- Represents the perceived cost of a journey
- Builds upon MEAT
- Also includes weighted costs for
  - Walking
  - Changing vehicles
  - Waiting at a stop
Perceived Arrival Time (PAT)

Further extending CSA:
- Represents the perceived cost of a journey
- Builds upon MEAT
- Also includes weighted costs for
  - Walking
  - Changing vehicles
  - Waiting at a stop

Properties:
- As efficient as plain CSA
- Requires only a single scan of the connection array
- Builds the foundation of an efficient CSA based assignment algorithm
CSA Based Assignment (Briem et al. 2017)

Algorithm overview:
- Partition O-D pairs by destination
- Handle destinations independently of each other
- For each destination:
  1. Compute PATs from everywhere to the destination
  2. Simulate Passenger movements through the network
  3. Refine the resulting journeys
Algorithm overview:

- Partition O-D pairs by destination
- Handle destinations independently of each other
- For each destination:
  1. Compute PATs from everywhere to the destination
     - Using a single scan of all connections
     - In reverse (descending order of arrival time, starting from the destination)
  2. Simulate Passenger movements through the network
  3. Refine the resulting journeys
Algorithm overview:

- Partition O-D pairs by destination
- Handle destinations independently of each other
- For each destination:
  1. Compute PATs from everywhere to the destination
     - Using a single scan of all connections
     - In reverse (descending order of arrival time, starting from the destination)
  2. Simulate Passenger movements through the network
     - Also using a single scan of all connections
     - In normal order (ascending order of arrival time)
     - Use PATs to decide if passengers use a connection or not
  3. Refine the resulting journeys
Passenger Movement Simulation:

- PAT of each connection is known
- Passengers are generated at their origin
- Passengers move towards their destination
  (One connection at a time)
- Whether a connection is used, depends on the connections PAT
- While getting closer to the destination:
  - Paths of individual passengers converge
  - More and more passengers collect at the same stops
  - All passengers at stop can use the same connections
  - Computation for this connection is only performed once

⇒ Synergy effects as more passengers gather at the same stops
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  1. Generate passengers with origin at the departure stop of $c$
Passenger Movement Simulation Example:
- Process connections in ascending order by departure time
- For each connection $c$:
  1. Decide which passengers enter the connection
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  1. Decide which passengers leave the trip

![Diagram of passenger movement simulation example](image-url)
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  1. Move disembarking passengers to their next stop

Time: 9:00
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection \( c \):
  1. Generate passengers with origin at the departure stop of \( c \)
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  1. Decide which passengers enter the connection
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  3. Decide which passengers leave the trip
CSA Based Assignment (Briem et al. 2017)

Passenger Movement Simulation Example:
- Process connections in ascending order by departure time
- For each connection $c$:
  4 Move disembarking passengers to their next stop

Time: 9:35
Passenger Movement Simulation Example:
- Process connections in ascending order by departure time
- For each connection $c$:
  1. Generate passengers with origin at the departure stop of $c$
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  - Decide which passengers enter the connection

![Diagram of passenger movement simulation example]

Time: 9:40
Passenger Movement Simulation Example:

- Process connections in ascending order by departure time
- For each connection $c$:
  
  3...
Journey Refinement: (Remove unwanted cycles)

- Cycle definition: Visiting a stop more than once
- Assigning cycles might be undesirable
- Journey with cycle can have minimum PAT
- High waiting cost leads to cycles
Journey Refinement: (Remove unwanted cycles)
- Cycle definition: Visiting a stop more than once
- Assigning cycles might be undesirable
- Journey with cycle can have minimum PAT
- High waiting cost leads to cycles
Journey Refinement: (Remove unwanted cycles)
- Cycle definition: Visiting a stop more than once
- Assigning cycles might be undesirable
- Journey with cycle can have minimum PAT
- High waiting cost leads to cycles
Benchmark Instance:
- Greater region of Stuttgart
- Reaching as far as Frankfurt, Basel or Munich
- Comprises the traffic of one day

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of vertices</td>
<td>15,115</td>
</tr>
<tr>
<td>Number of stops</td>
<td>13,941</td>
</tr>
<tr>
<td>Number of edges</td>
<td>33,890</td>
</tr>
<tr>
<td>Number of edges without loops</td>
<td>18,775</td>
</tr>
<tr>
<td>Number of connections</td>
<td>780,042</td>
</tr>
<tr>
<td>Number of trips</td>
<td>47,844</td>
</tr>
<tr>
<td>Number of passenger</td>
<td>1,249,910</td>
</tr>
</tbody>
</table>
Running Time and Passenger Multiplier:

- Algorithm assigns only one journey per O-D pair
- However probabilistic distribution of journeys is desired
- Solution: simulate multiple passengers per O-D pair
Running Time and Passenger Multiplier:

- Algorithm assigns only one journey per O-D pair
- However probabilistic distribution of journeys is desired
- Solution: simulate multiple passengers per O-D pair
CSA Based Assignment (Briem et al. 2017)

Comparison with Visum:
- Commercial tool for traffic planning

<table>
<thead>
<tr>
<th></th>
<th>Visum</th>
<th>CSA Based Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total travel time</td>
<td>46.885</td>
<td>47.199</td>
</tr>
<tr>
<td>Time spent in vehicle</td>
<td>21.059</td>
<td>21.231</td>
</tr>
<tr>
<td>Time spent walking</td>
<td>22.394</td>
<td>22.476</td>
</tr>
<tr>
<td>Time spent waiting</td>
<td>3.432</td>
<td>3.492</td>
</tr>
<tr>
<td>Trips per passenger</td>
<td>1.771</td>
<td>1.746</td>
</tr>
<tr>
<td>Connections per passenger</td>
<td>9.396</td>
<td>9.474</td>
</tr>
<tr>
<td>Passengers per connection</td>
<td>12.740</td>
<td>12.847</td>
</tr>
</tbody>
</table>
CSA Based Assignment (Briem et al. 2017)

Comparison with Visum:
- Commercial tool for traffic planning
- The computation in Visum takes ~30 minutes (8 threads)
- The CSA based assignment takes 39 seconds (4 threads)
- Both assignments look similar

<table>
<thead>
<tr>
<th>Quantity</th>
<th>VISUM</th>
<th>CSA Based Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>mean</td>
</tr>
<tr>
<td>Total travel time [min]</td>
<td>2.98</td>
<td>46.885</td>
</tr>
<tr>
<td>Time spent in vehicle [min]</td>
<td>0.02</td>
<td>21.059</td>
</tr>
<tr>
<td>Time spent walking [min]</td>
<td>2.00</td>
<td>22.394</td>
</tr>
<tr>
<td>Time spent waiting [min]</td>
<td>0.00</td>
<td>3.432</td>
</tr>
<tr>
<td>Trips per passenger</td>
<td>1.00</td>
<td>1.771</td>
</tr>
<tr>
<td>Connections per passenger</td>
<td>1.00</td>
<td>9.396</td>
</tr>
<tr>
<td>Passengers per connection</td>
<td>0.00</td>
<td>12.740</td>
</tr>
</tbody>
</table>
Multimodal Assignments:
- Goal: Consider multiple modes of transportation at once
- Problem: Combining timetable and non-timetable networks is hard
Multimodal Assignments:

- Goal: Consider multiple modes of transportation at once
- Problem: Combining timetable and non-timetable networks is hard
- ULTRA: (Baum et al. 2019)
  - Enables UnLimited TRAnsfers for many Public Transit algorithms
  - Is also combinable with the CSA based assignment (Sauer et al. 2019)
  - First efficient assignment for public transit with secondary transfer mode
Multimodal Assignments:
- Goal: Consider multiple modes of transportation at once
- Problem: Combining timetable and non-timetable networks is hard
- ULTRA: (Baum et al. 2019)
  - Enables UnLimited TRA nsfers for many Public Transit algorithms
  - Is also combinable with the CSA based assignment (Sauer et al. 2019)
  - First efficient assignment for public transit with secondary transfer mode

Consider Vehicle capacities:
- Similar to assignments on road networks
- PAT depends on utilization
- Iterative approach
In Combined Networks
Next steps: True Multimodal Assignments

State of the art:
- Applications already handle different modes of transportation
- However: mode choice and assignment sequentially
  1. Choose travel mode
  2. Select route for chosen mode of transportation

Integrate mode choice with route assignment:
- Integrate both assignment types
- Combine O-D for road networks and for public transit
- Algorithms assigns both: journey and travel mode
Thank you for your attention!


References III


