# In Transit to Constant Time Shortest-Path Queries in Road Networks

Holger Bast  Stefan Funke  Peter Sanders  Dominik Schultes

Domagoj Matijevic

Max-Planck-Institut f. Informatik  Universität Karlsruhe

Germany

# The Shortest Path Problem

- showcase problem for the power of algorithmics
- for general graphs with non-negative edge weights, exact solution given by Dijkstra's algorithm in $O(m + n \log n)$ where $n = \#$ nodes, $m = \#$ edges

# The Shortest Path Problem

- showcase problem for the power of algorithmics
- for general graphs with non-negative edge weights, exact solution given by Dijkstra's algorithm in $O(m + n \log n)$ where $n =\#$ nodes, $m =\#$ edges
- Example: roadmap of the US, ($n = 24$ Mio $m = 58$ Mio)

# The Shortest Path Problem

- showcase problem for the power of algorithmics
- for general graphs with non-negative edge weights, exact solution given by Dijkstra's algorithm in $O(m + n \log n)$ where $n = \#$ nodes, $m = \#$ edges
- Example: roadmap of the US, ($n = 24$ Mio $m = 58$ Mio)
- Good Dijkstra implementation takes around 10secs to answer a random source-target query
- $\Rightarrow$ infeasible for a web-based route planner

# The Shortest Path Problem

- showcase problem for the power of algorithmics
- for general graphs with non-negative edge weights, exact solution given by Dijkstra's algorithm in $O(m + n \log n)$ where $n =\#$ nodes, $m =\#$ edges
- Example: roadmap of the US, ($n = 24$ Mio $m = 58$ Mio)
- Good Dijkstra implementation takes around 10secs to answer a random source-target query
- $\Rightarrow$ infeasible for a web-based route planner
- $\Rightarrow$ need to exploit the special structure of roadmaps

# The Shortest Path Problem

- showcase problem for the power of algorithmics
- for general graphs with non-negative edge weights, exact solution given by Dijkstra's algorithm in $O(m + n \log n)$ where $n =\#$ nodes, $m =\#$ edges
- Example: roadmap of the US, ($n = 24$ Mio $m = 58$ Mio)
- Good Dijkstra implementation takes around 10secs to answer a random source-target query
- $\Rightarrow$ infeasible for a web-based route planner
- $\Rightarrow$ need to exploit the special structure of roadmaps
- so far, best solutions allow for a query time in the order of *milliseconds* (with preprocessing)

# Extremely Useful Insight No.1 (Gutman'04)

Only few roads appear in "the middle" of long shortest paths!

# Extremely Useful Insight No.1 (Gutman'04)

Only few roads appear in "the middle" of long shortest paths!

- used to prune the set of edges Dijkstra has to look at
- two metrics: one determines what *shortest paths* are, the other what *in the middle* means

# Extremely Useful Insight No.1 (Gutman'04)

Only few roads appear in "the middle" of long shortest paths!

- used to prune the set of edges Dijkstra has to look at
- two metrics: one determines what *shortest paths* are, the other what *in the middle* means
- extensively exploited e.g. in papers by Gutman, Goldberg et al. or Sanders/Schultes (the latter also use network contraction for increased efficiency)

# Extremely Useful Insight No.1 (Gutman'04)

Only few roads appear in "the middle" of long shortest paths!

- used to prune the set of edges Dijkstra has to look at
- two metrics: one determines what *shortest paths* are, the other what *in the middle* means
- extensively exploited e.g. in papers by Gutman, Goldberg et al. or Sanders/Schultes (the latter also use network contraction for increased efficiency)
- yields shortest path queries in the order of *milliseconds* on the US roadmap (after preprocessing)

# Extremely Useful Insight No.1 (Gutman'04)

Only few roads appear in "the middle" of long shortest paths!

- used to prune the set of edges Dijkstra has to look at
- two metrics: one determines what *shortest paths* are, the other what *in the middle* means
- extensively exploited e.g. in papers by Gutman, Goldberg et al. or Sanders/Schultes (the latter also use network contraction for increased efficiency)
- yields shortest path queries in the order of *milliseconds* on the US roadmap (after preprocessing)
- any sensible reason to aim for faster query times ?
  $\Rightarrow$YES! Web services, traffic simulations, logistics ...

# First Contribution: Extremely Useful Insight No.2

Imagine you aim to travel 'far'
– let's say more than 50 miles –

how many different routes would you potentially use to leave your local neighborhood ?

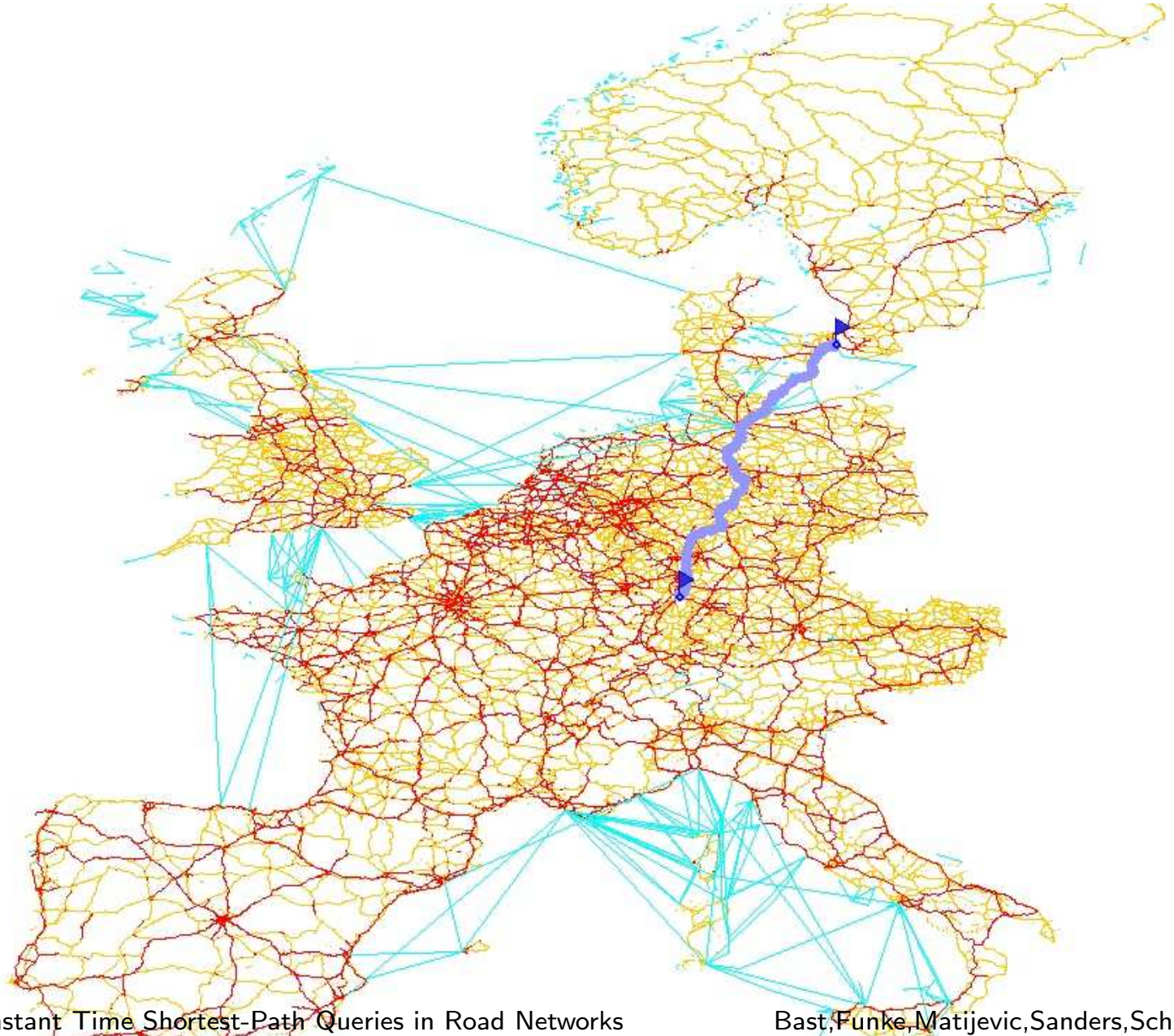
# First Contribution: Extremely Useful Insight No.2

Imagine you aim to travel 'far'
– let's say more than 50 miles –

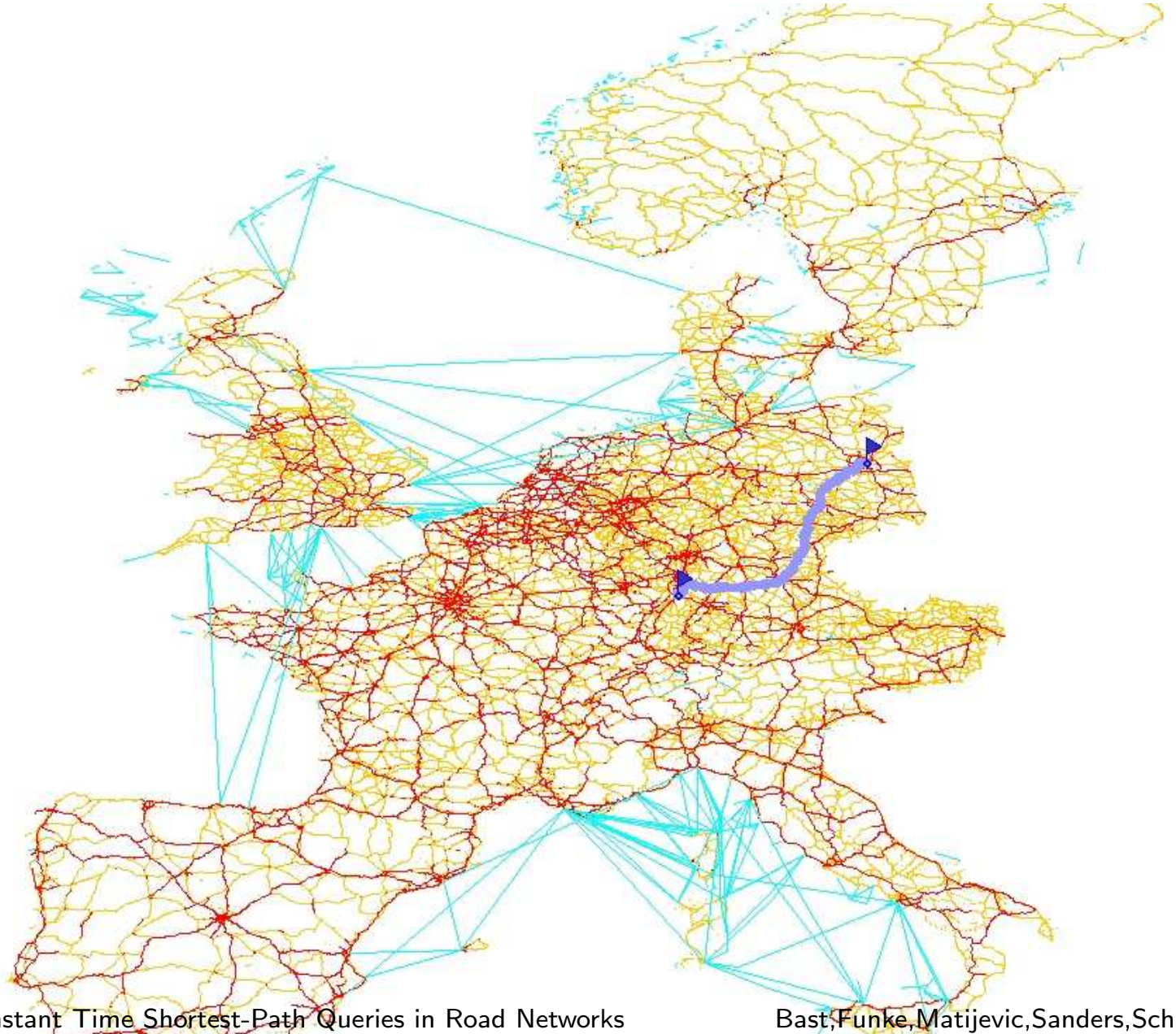
how many different routes would you potentially use to leave your local neighborhood ?

Only VERY few!

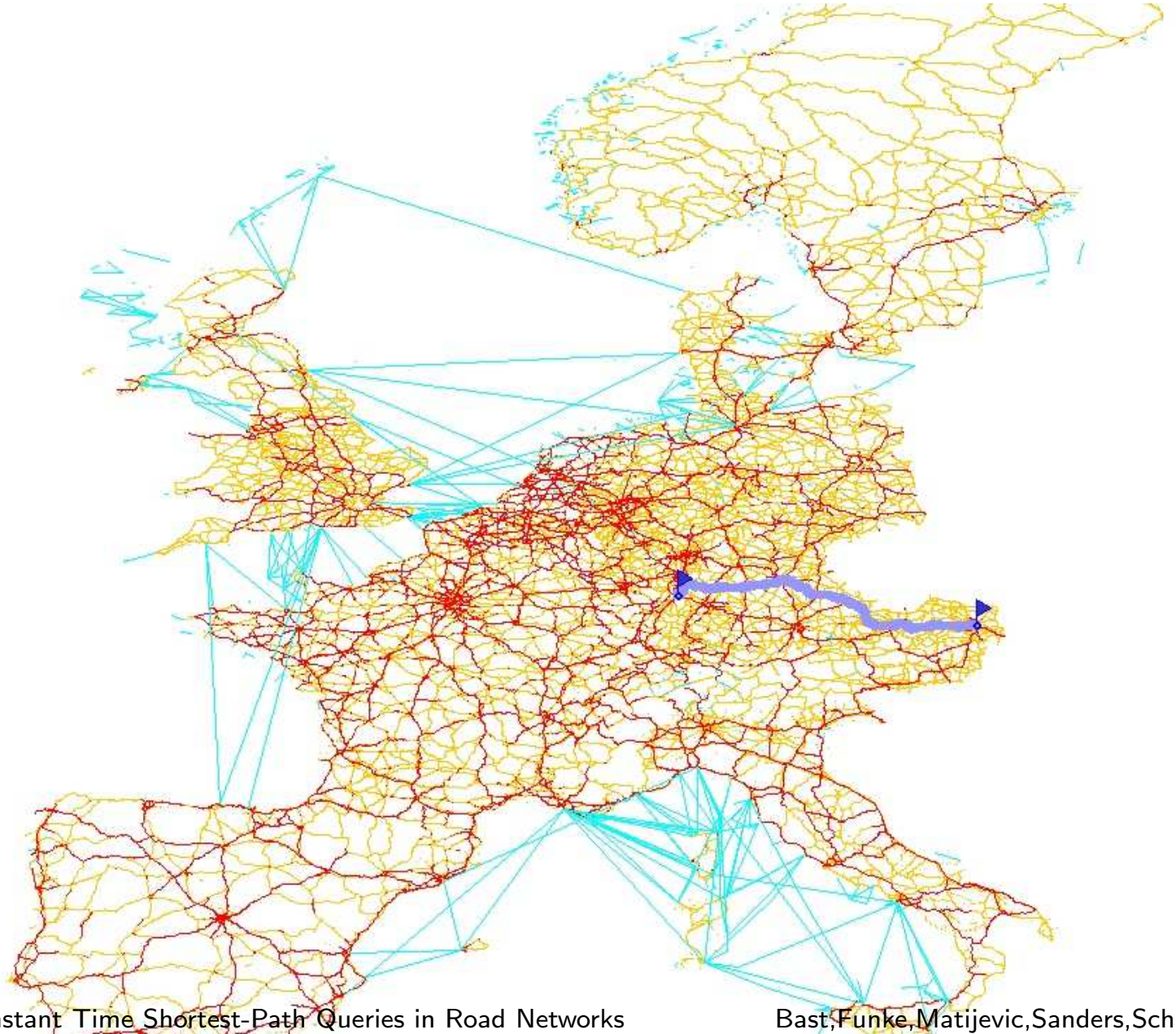
# Example: Karlsruhe → Copenhagen



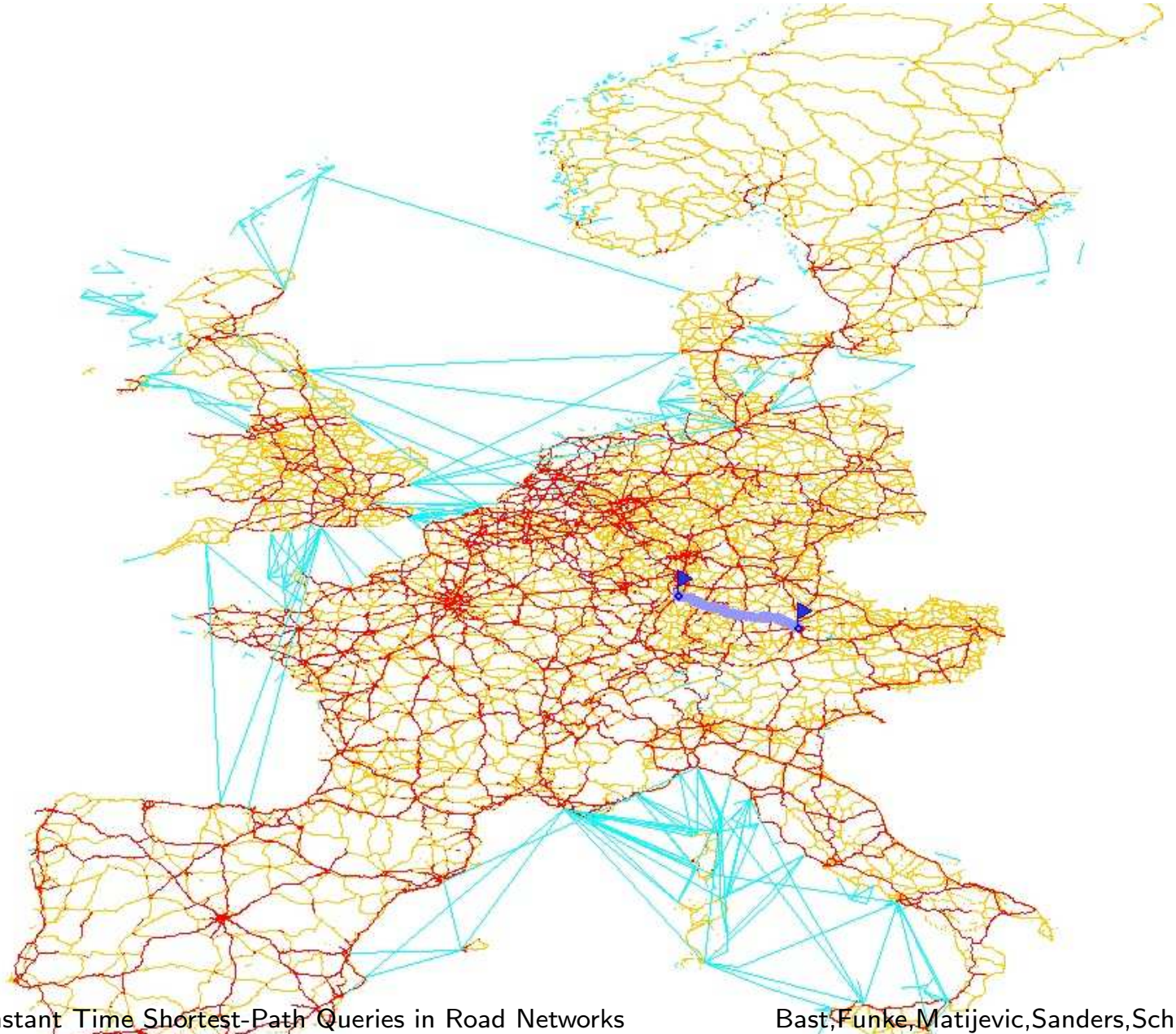
# Example: Karlsruhe → Berlin



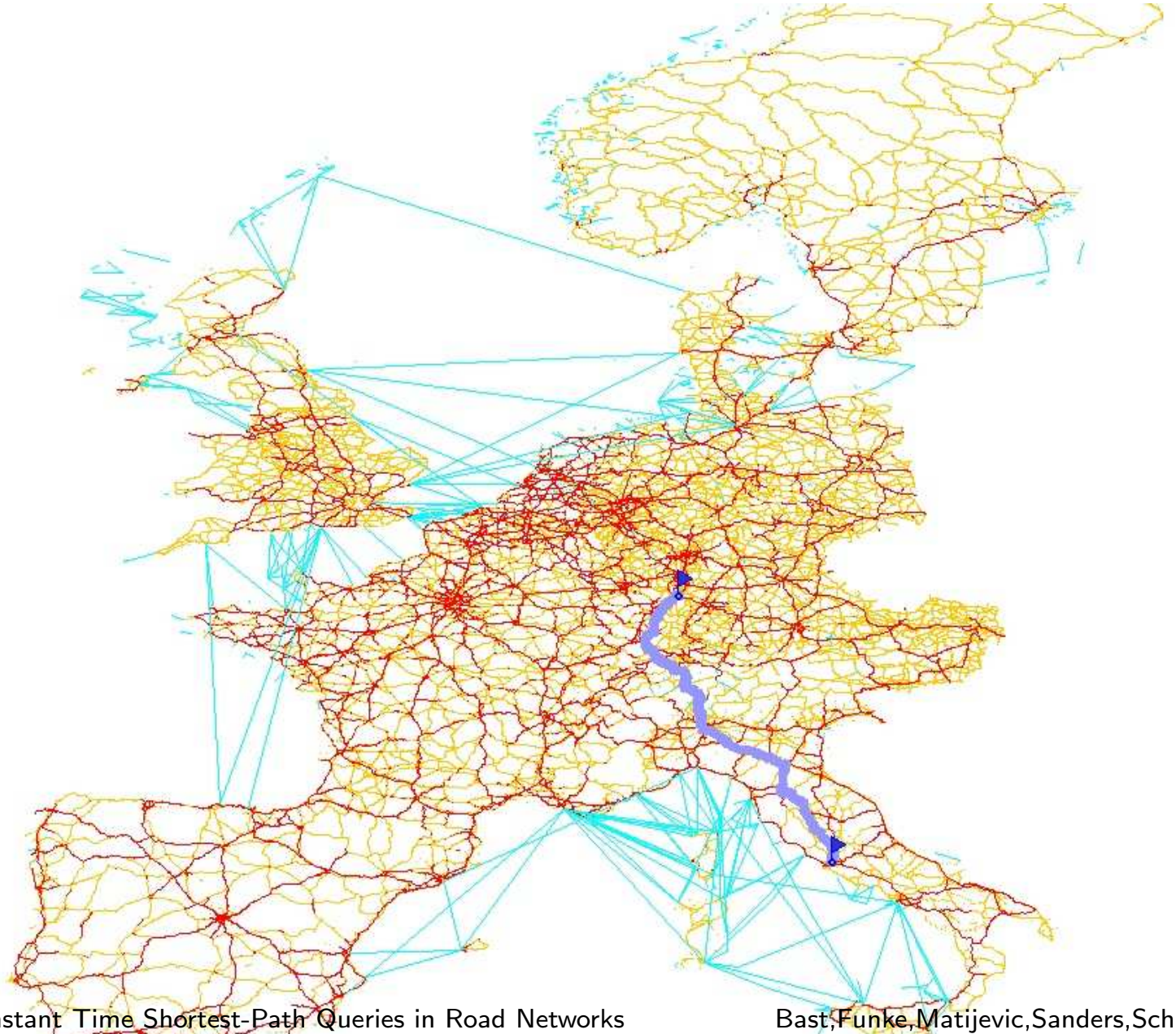
# Example: Karlsruhe → Vienna



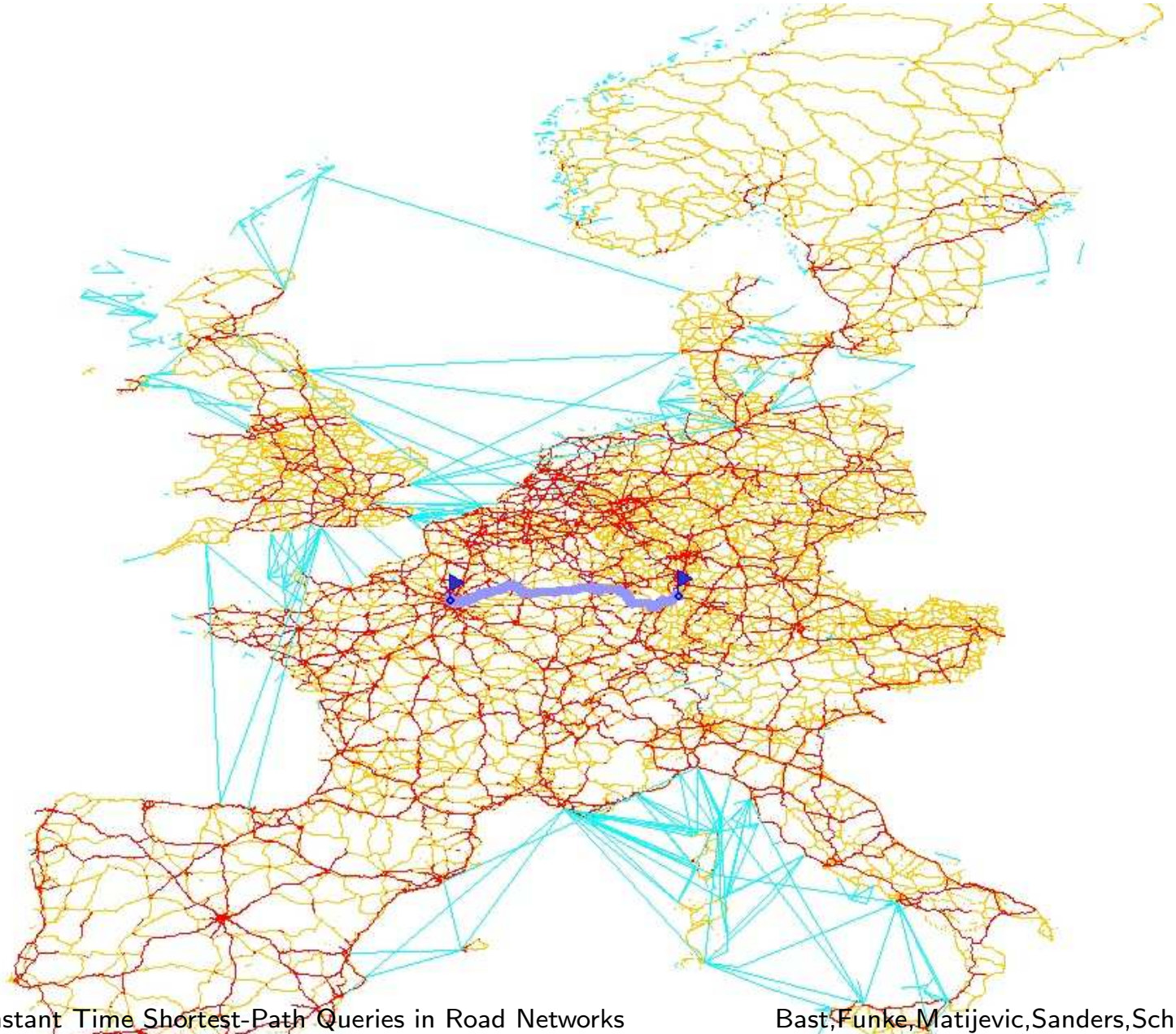
# Example: Karlsruhe → Munich



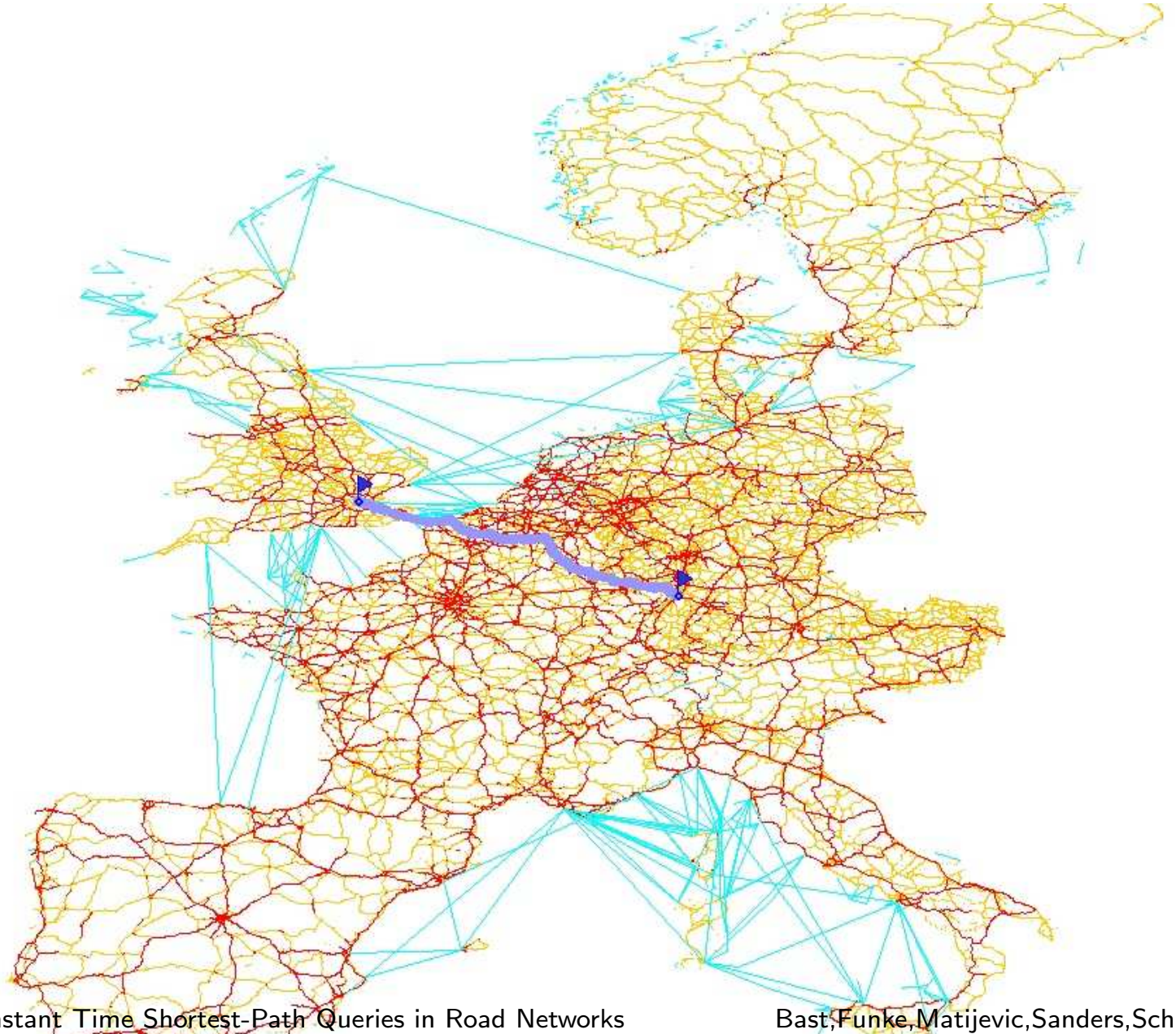
# Example: Karlsruhe → Rome



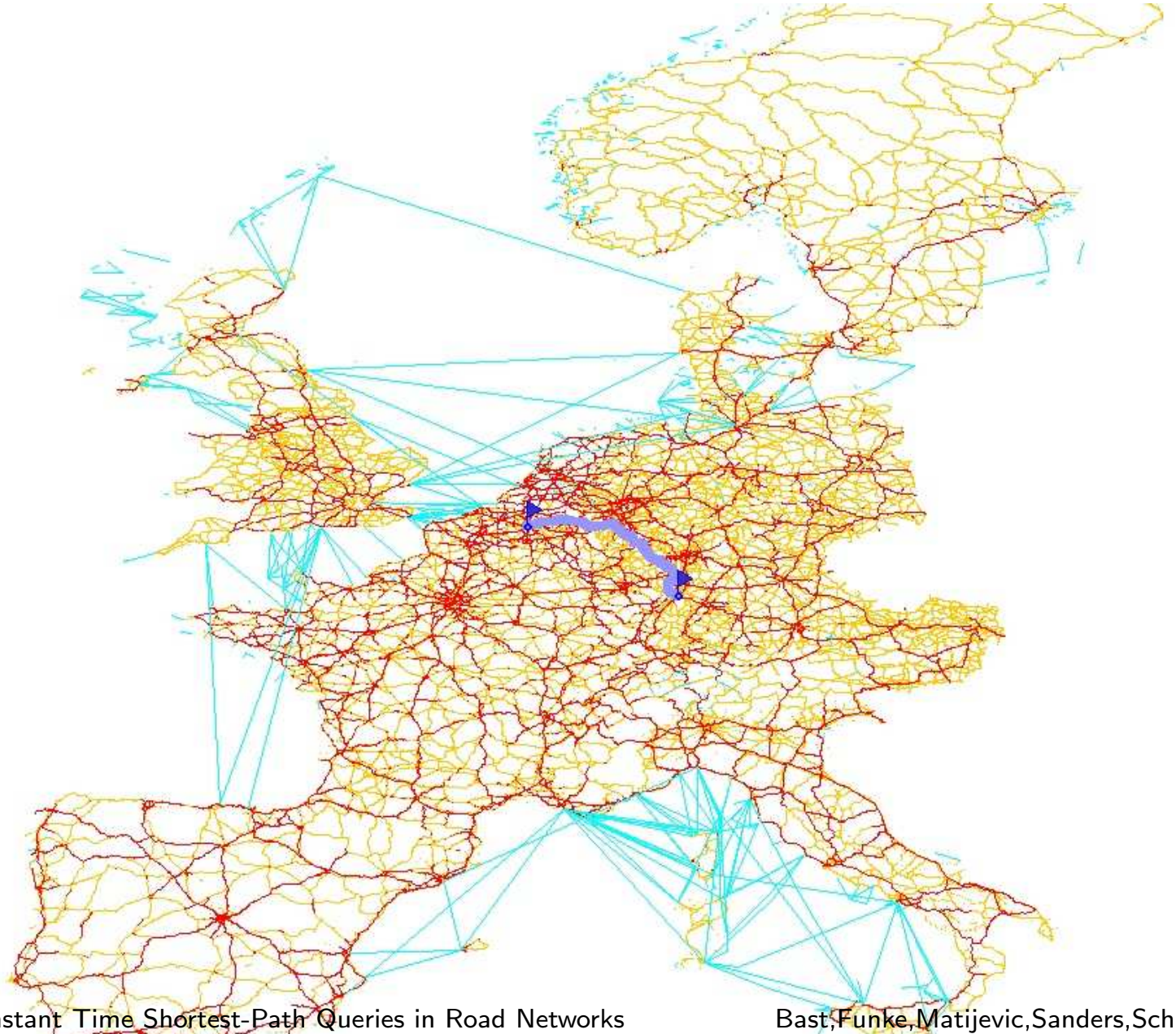
# Example: Karlsruhe → Paris



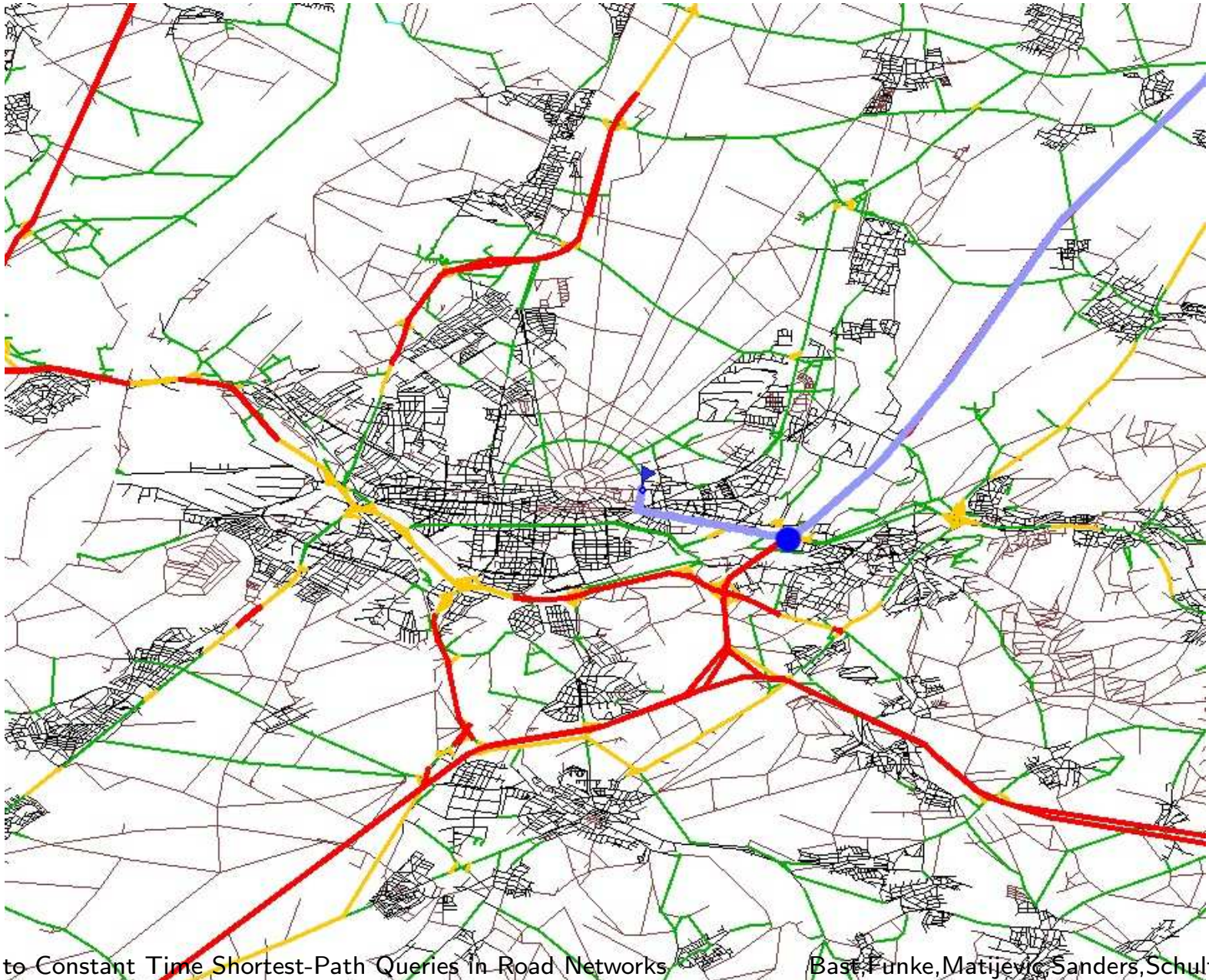
# Example: Karlsruhe → London



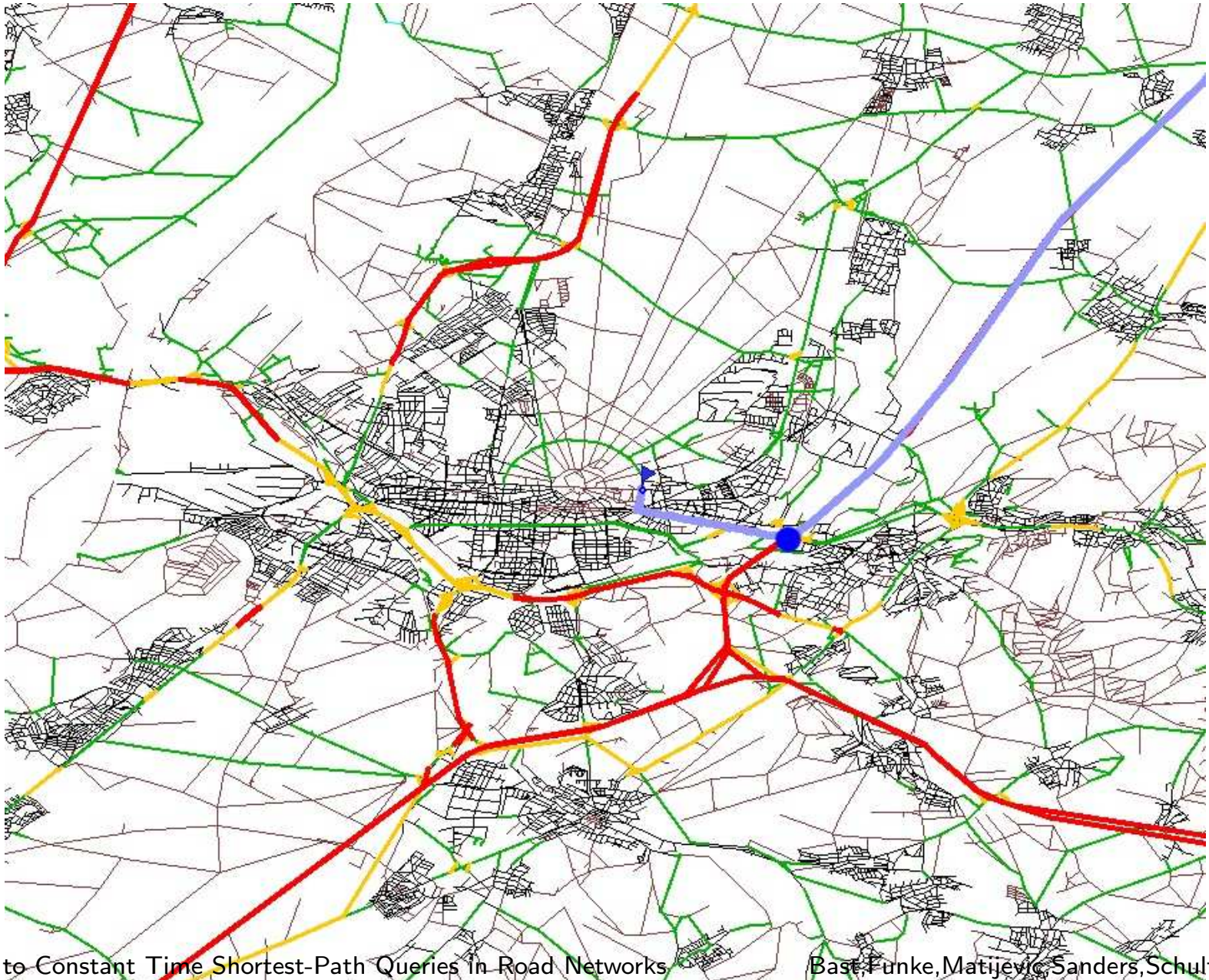
# Example: Karlsruhe → Brussels



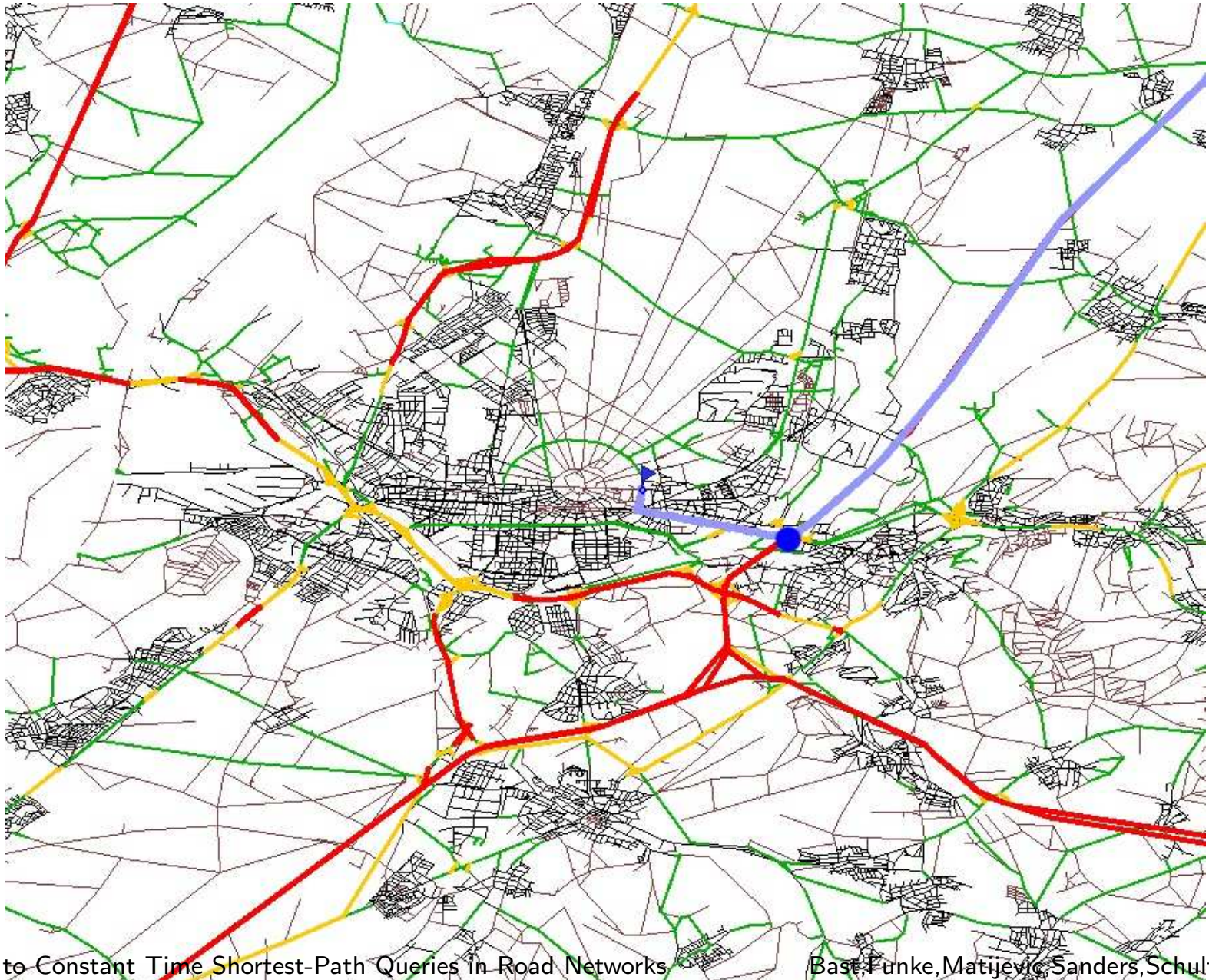
# Example: Karlsruhe → Copenhagen



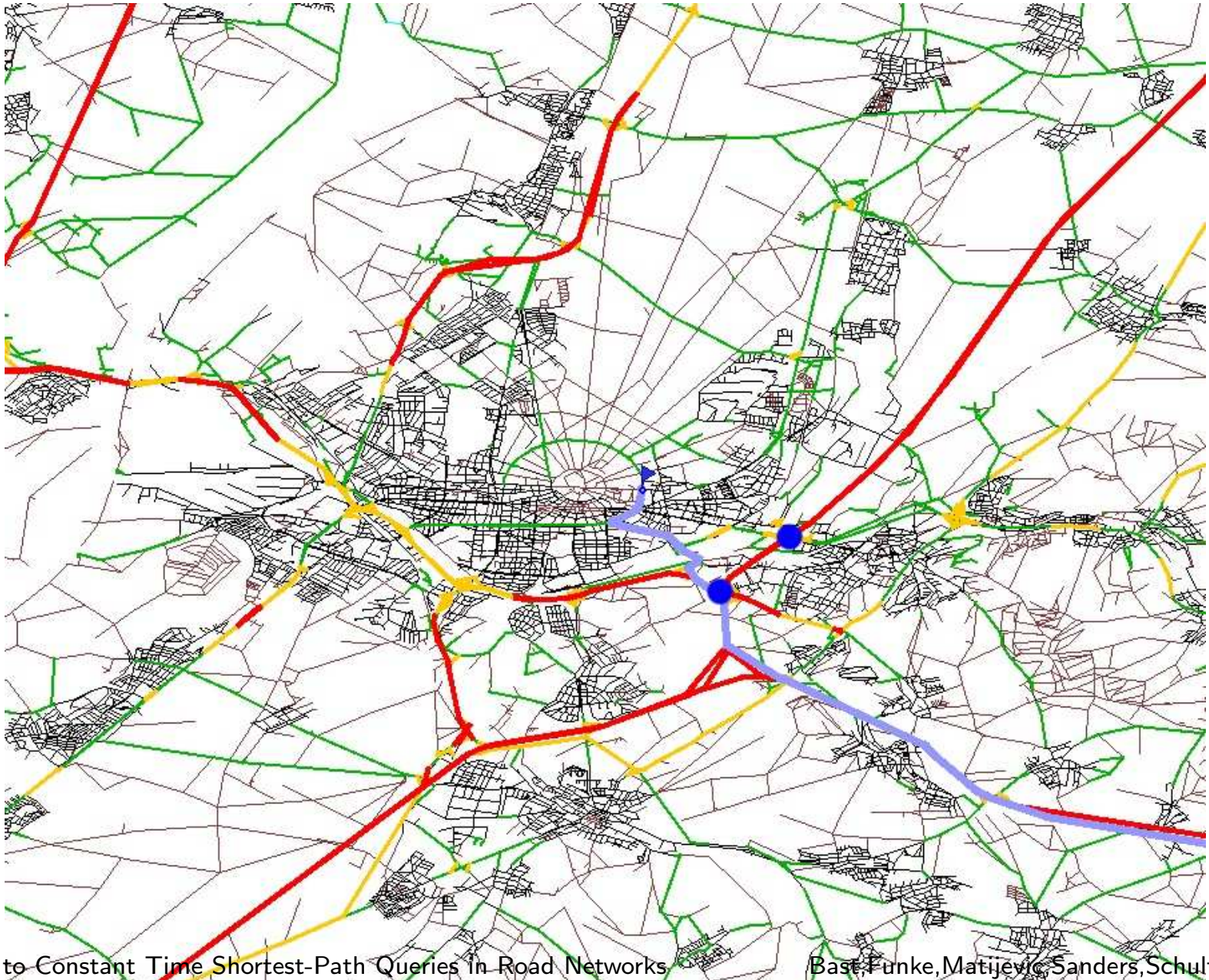
# Example: Karlsruhe → Berlin



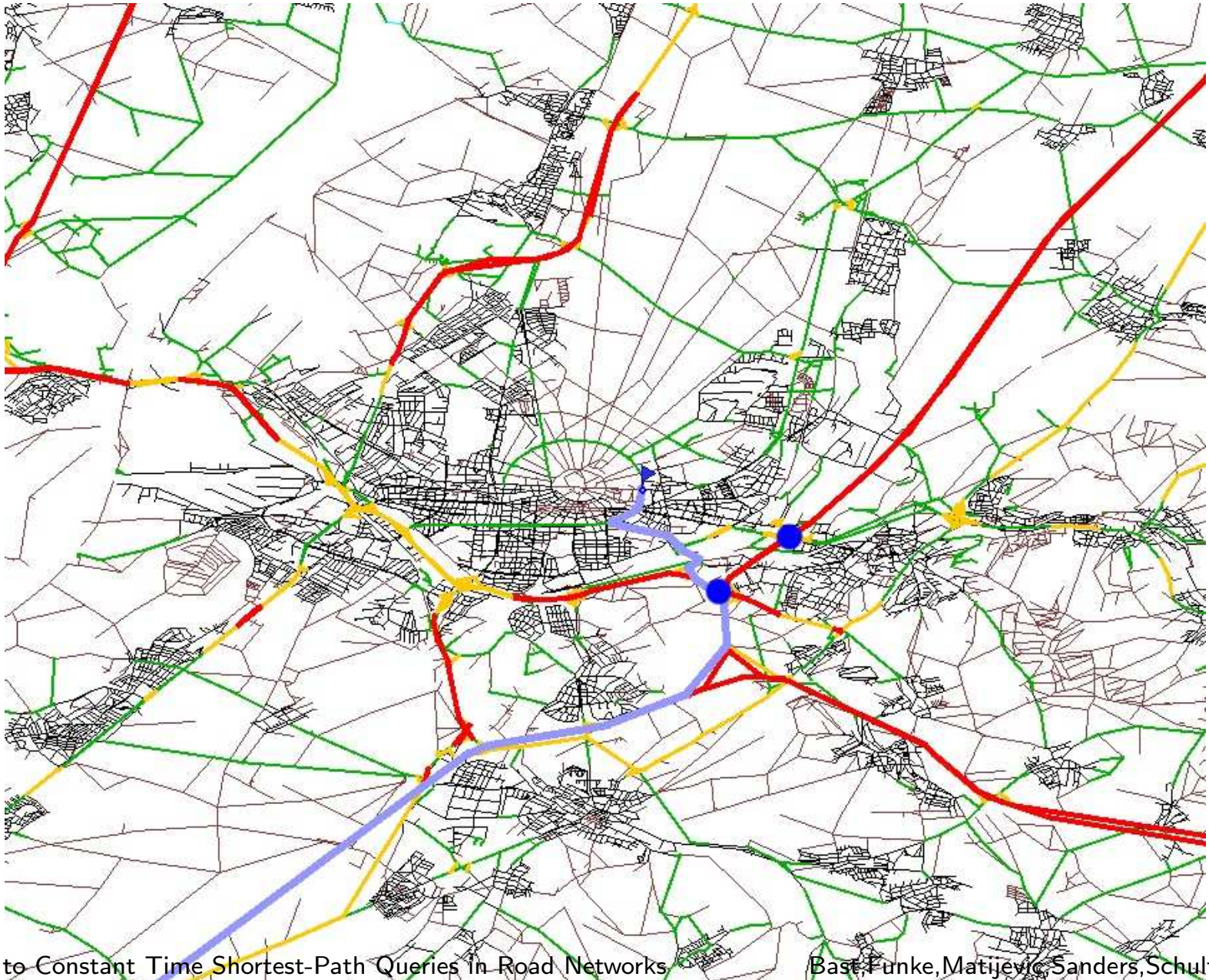
# Example: Karlsruhe → Vienna



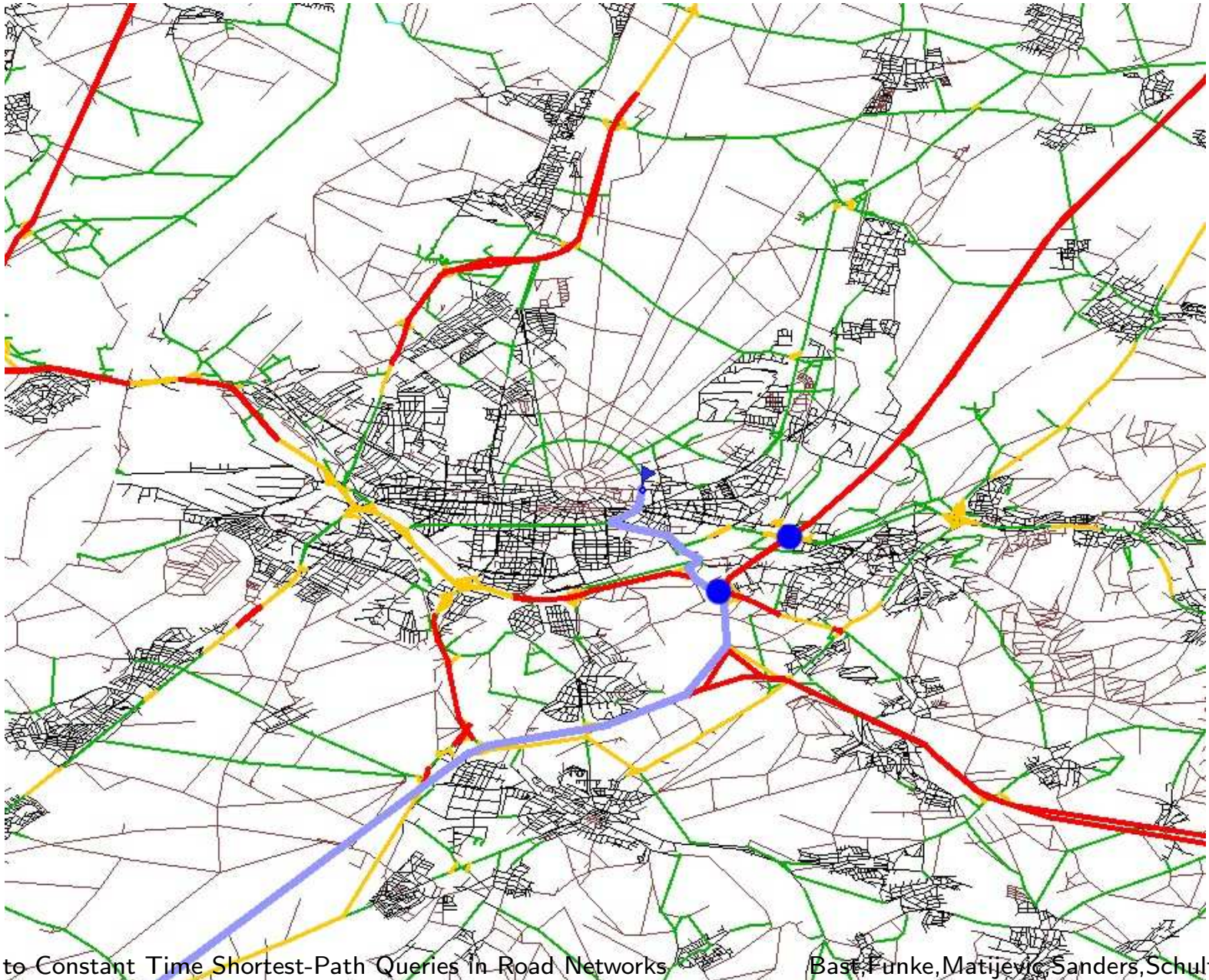
# Example: Karlsruhe → Munich



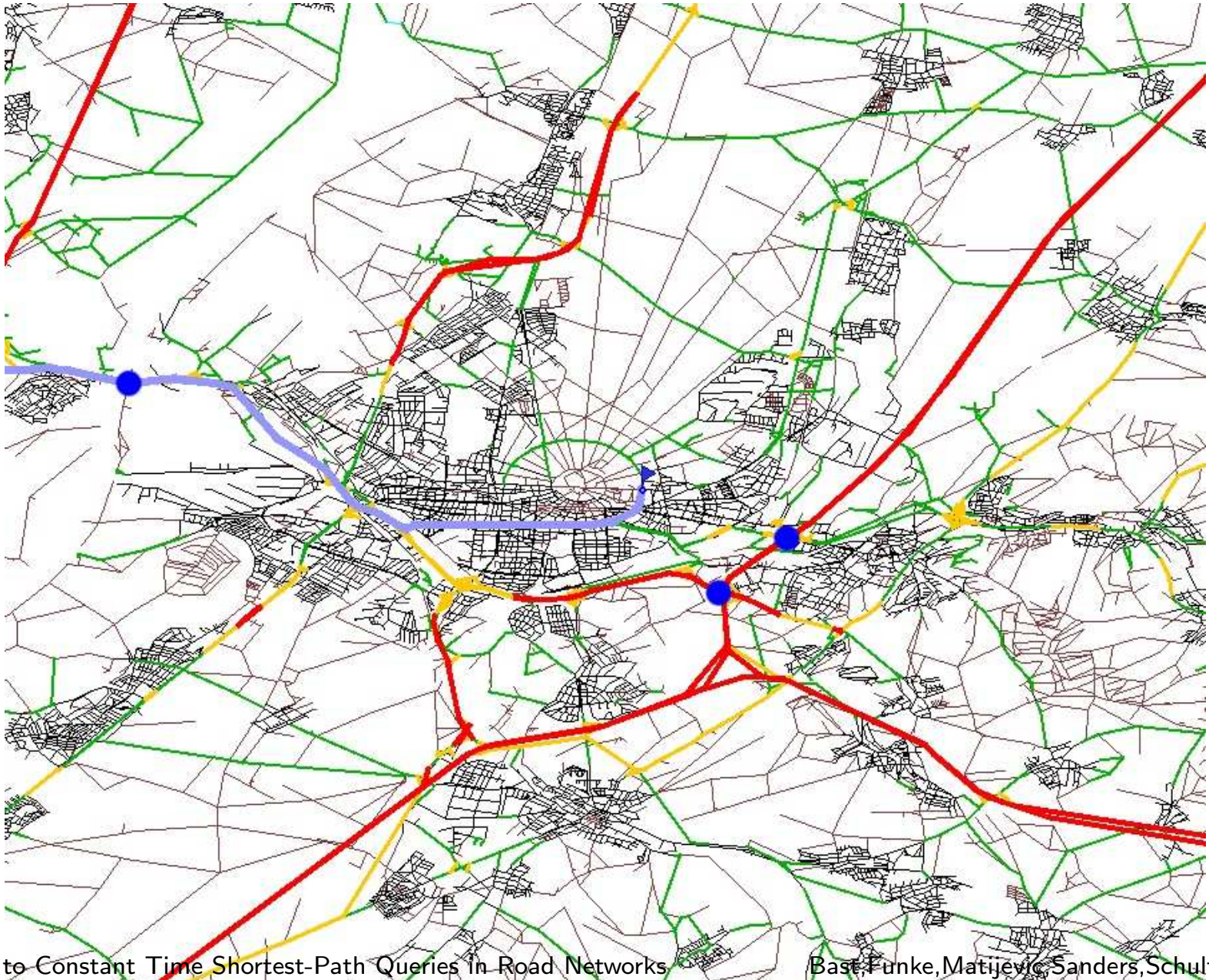
# Example: Karlsruhe → Rome



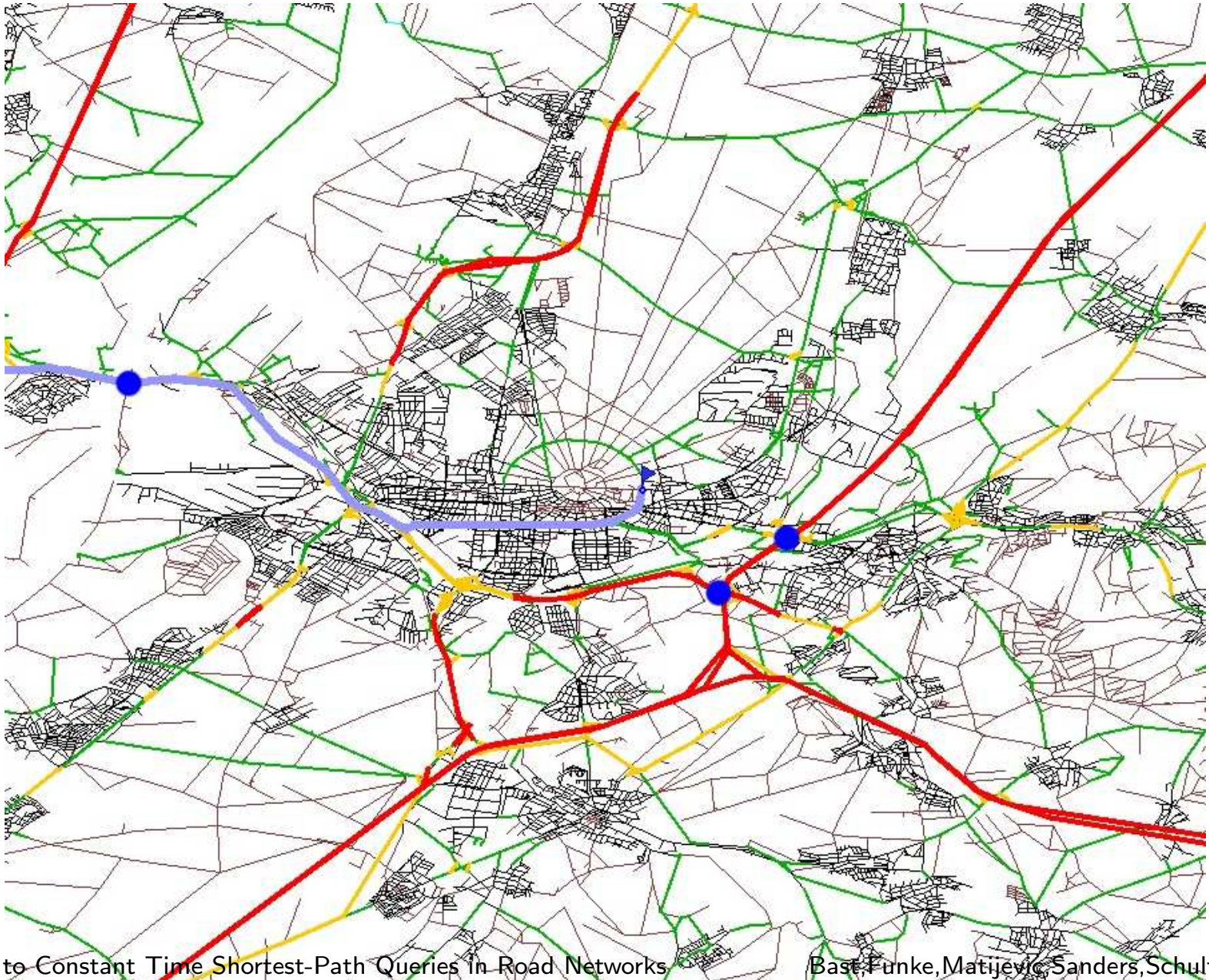
# Example: Karlsruhe → Paris



# Example: Karlsruhe → London



# Example: Karlsruhe → Brussels



# Second Contribution: Exploitation of our Insight

Based on our new insight we propose **Transit Node Routing**, a highly efficient scheme which allows for

# Second Contribution: Exploitation of our Insight

Based on our new insight we propose **Transit Node Routing**, a highly efficient scheme which allows for

- reduction of shortest path distance queries to a constant number of table-lookups

# Second Contribution: Exploitation of our Insight

Based on our new insight we propose **Transit Node Routing**, a highly efficient scheme which allows for

- reduction of shortest path distance queries to a constant number of table-lookups
- various trade-offs between space and query-/preprocessing times:
  - avg. query times between $5\mu s$ and $63\mu s$ (on the US road map)
  - preprocessing times between 1h and 20h
  - a per node space overhead of 21 to 244 bytes

# Second Contribution: Exploitation of our Insight

Based on our new insight we propose **Transit Node Routing**, a highly efficient scheme which allows for

- reduction of shortest path distance queries to a constant number of table-lookups
- various trade-offs between space and query-/preprocessing times:
  - avg. query times between $5\mu s$ and $63\mu s$ (on the US road map)
  - preprocessing times between 1h and 20h
  - a per node space overhead of 21 to 244 bytes
- $\Rightarrow$ Query times *orders of magnitudes* better than previously reported results

Milliseconds ($10^{-3}$) vs. Microseconds ($10^{-6}$)!

# Transit Nodes: Formalization

Consider the set $\Pi$ of all 'long' shortest paths within the network.
We want to find a set of *Transit Nodes* $\mathcal{T}$ such that

# Transit Nodes: Formalization

Consider the set $\Pi$ of all 'long' shortest paths within the network.
We want to find a set of *Transit Nodes* $\mathcal{T}$ such that

- every $\pi \in \Pi$ contains a node from $\mathcal{T}$ (global hitting set)

# Transit Nodes: Formalization

Consider the set $\Pi$ of all 'long' shortest paths within the network.
We want to find a set of *Transit Nodes* $\mathcal{T}$ such that

- every $\pi \in \Pi$ contains a node from $\mathcal{T}$ (global hitting set)
- $|\mathcal{T}| = O(\sqrt{|V|})$

# Transit Nodes: Formalization

Consider the set $\Pi$ of all 'long' shortest paths within the network. We want to find a set of *Transit Nodes* $\mathcal{T}$ such that

- every $\pi \in \Pi$ contains a node from $\mathcal{T}$ (global hitting set)
- $|\mathcal{T}| = O(\sqrt{|V|})$

That comes as no surprise given the previous results, but with the new insight we might even want the following property:

# Transit Nodes: Formalization

Consider the set $\Pi$ of all 'long' shortest paths within the network. We want to find a set of *Transit Nodes* $\mathcal{T}$ such that

- every $\pi \in \Pi$ contains a node from $\mathcal{T}$ (global hitting set)
- $|\mathcal{T}| = O(\sqrt{|V|})$

That comes as no surprise given the previous results, but with the new insight we might even want the following property:

- for every node $v$ there is a set of *Access Nodes* $\mathcal{A}(v) \subset \mathcal{T}$ which hits all 'long' paths starting at $v$ and $|\mathcal{A}(v)|$ is constant

# Transit Nodes: Formalization

Consider the set $\Pi$ of all 'long' shortest paths within the network. We want to find a set of *Transit Nodes* $\mathcal{T}$ such that

- every $\pi \in \Pi$ contains a node from $\mathcal{T}$ (global hitting set)
- $|\mathcal{T}| = O(\sqrt{|V|})$

That comes as no surprise given the previous results, but with the new insight we might even want the following property:

- for every node $v$ there is a set of *Access Nodes* $\mathcal{A}(v) \subset \mathcal{T}$ which hits all 'long' paths starting at $v$ and $|\mathcal{A}(v)|$ is constant

What to do with the transit/access nodes ?

# Transit Node Routing

## Preprocessing

- determine $\mathcal{T}$ and $\mathcal{A}(v)$

# Transit Node Routing

## Preprocessing

- determine $\mathcal{T}$ and $\mathcal{A}(v)$
- compute and store :
  - for each node $v$ its distances to $\mathcal{A}(v)$ – $O(n)$ space

# Transit Node Routing

## Preprocessing

- determine $\mathcal{T}$ and $\mathcal{A}(v)$
- compute and store :
  - for each node $v$ its distances to $\mathcal{A}(v)$ – $O(n)$ space
  - all pair-wise distances for $\mathcal{T} \times \mathcal{T}$ – $O(n)$ space

# Transit Node Routing

## Preprocessing

- determine $\mathcal{T}$ and $\mathcal{A}(v)$
- compute and store :
  - for each node $v$ its distances to $\mathcal{A}(v)$ – $O(n)$ space
  - all pair-wise distances for $\mathcal{T} \times \mathcal{T}$ – $O(n)$ space

## Query(s,t)

- decide whether path from $s$ to $t$ is 'long'/non-local

# Transit Node Routing

## Preprocessing

- determine $\mathcal{T}$ and $\mathcal{A}(v)$
- compute and store :
  - for each node $v$ its distances to $\mathcal{A}(v)$ – $O(n)$ space
  - all pair-wise distances for $\mathcal{T} \times \mathcal{T}$ – $O(n)$ space

## Query(s,t)

- decide whether path from $s$ to $t$ is 'long'/non-local
- YES $\rightarrow$ for every $(a_s, a_t)$, $a_s \in \mathcal{A}(s), a_t \in \mathcal{A}(t)$ evaluate

$$\text{dist} = \underbrace{d(s, a_s)}_{\text{stored with s}} + \overbrace{d(a_s, a_t)}^{\text{in dist.-table } \mathcal{T}\times\mathcal{T}} + \underbrace{d(a_t, t)}_{\text{stored with t}}$$

# Transit Node Routing

## Preprocessing

- determine $\mathcal{T}$ and $\mathcal{A}(v)$
- compute and store :
  - for each node $v$ its distances to $\mathcal{A}(v)$ – $O(n)$ space
  - all pair-wise distances for $\mathcal{T} \times \mathcal{T}$ – $O(n)$ space

## Query(s,t)

- decide whether path from $s$ to $t$ is 'long'/non-local
- YES $\rightarrow$ for every $(a_s, a_t)$, $a_s \in \mathcal{A}(s), a_t \in \mathcal{A}(t)$ evaluate

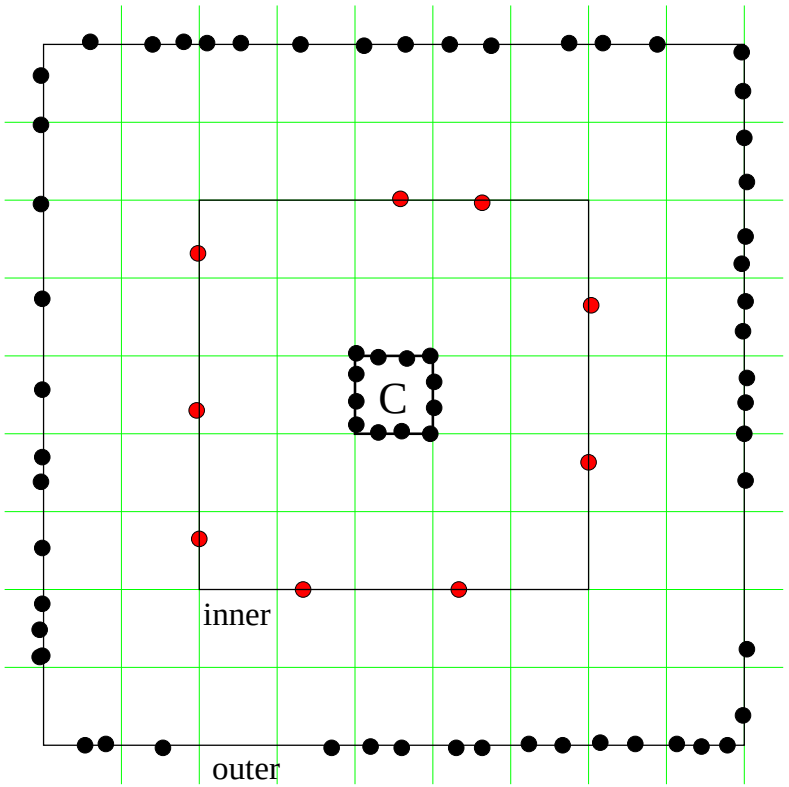
$$\text{dist} = \underbrace{d(s, a_s)}_{\text{stored with s}} + \overbrace{d(a_s, a_t)}^{\text{in dist.-table } \mathcal{T}\times\mathcal{T}} + \underbrace{d(a_t, t)}_{\text{stored with t}}$$

- NO $\rightarrow$ use favourite SP data structure – HH, edge reach ...

# Grid-based Implementation

First attempt – adhoc realization of the core idea.

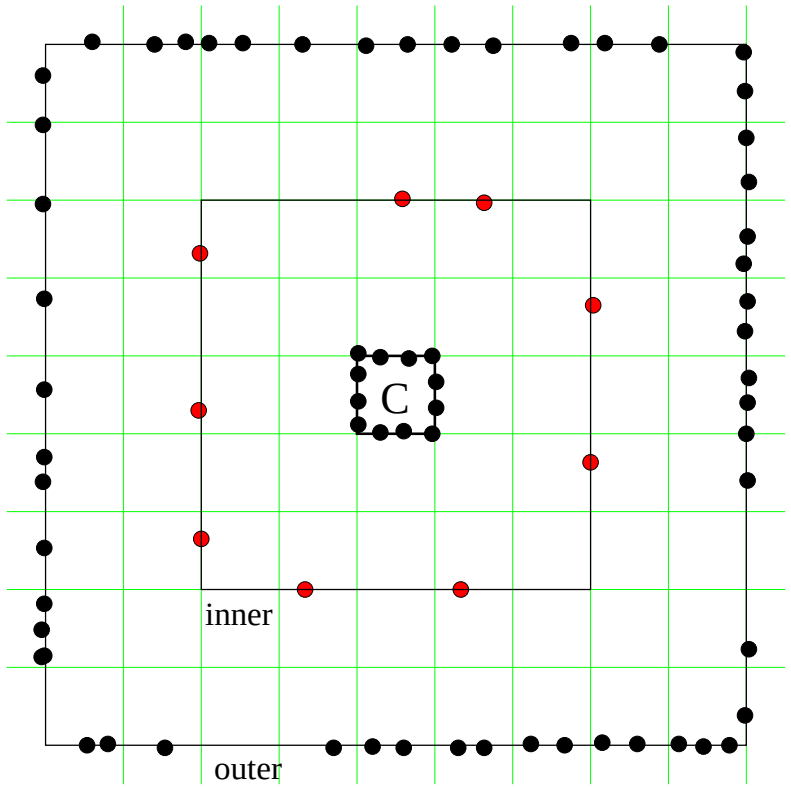
- impose a grid, e.g. $128 \times 128$ over the network



# Grid-based Implementation

First attempt – adhoc realization of the core idea.

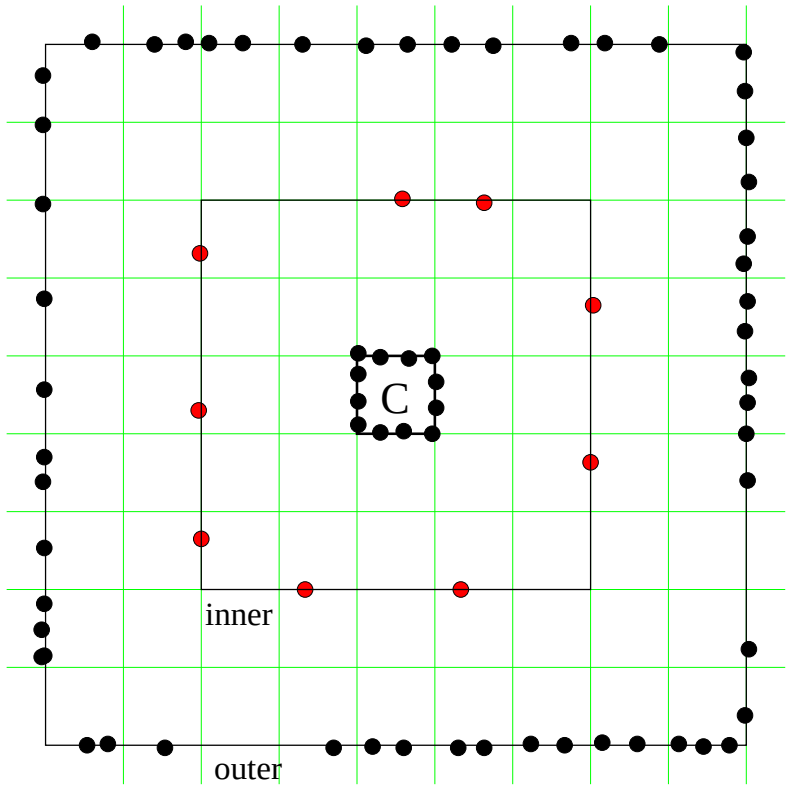
- impose a grid, e.g. $128 \times 128$ over the network
- for every cell $\mathcal{C}$ determine its *access nodes* $\mathcal{C}(\mathcal{A})$ which
  - lie on the inner boundary
  - appear on a shortest path from some node in $\mathcal{C}$ to some node on the outer boundary



# Grid-based Implementation

First attempt – adhoc realization of the core idea.

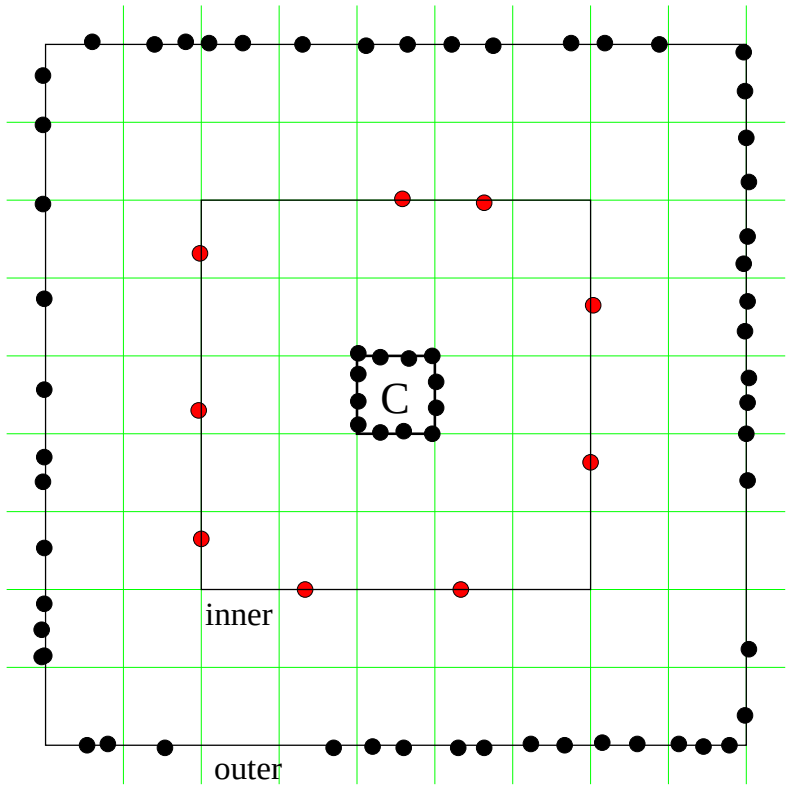
- impose a grid, e.g. $128 \times 128$ over the network
- for every cell $\mathcal{C}$ determine its *access nodes* $\mathcal{C}(\mathcal{A})$ which
  - lie on the inner boundary
  - appear on a shortest path from some node in $\mathcal{C}$ to some node on the outer boundary
- $\forall v \in \mathcal{C}$ we set $\mathcal{A}(v) = \mathcal{A}(\mathcal{C})$



# Grid-based Implementation

First attempt – adhoc realization of the core idea.

- impose a grid, e.g. $128 \times 128$ over the network
- for every cell $\mathcal{C}$ determine its *access nodes* $\mathcal{C}(\mathcal{A})$ which
  - lie on the inner boundary
  - appear on a shortest path from some node in $\mathcal{C}$ to some node on the outer boundary
- $\forall v \in \mathcal{C}$ we set $\mathcal{A}(v) = \mathcal{A}(\mathcal{C})$
- $\mathcal{T} = \cup \mathcal{A}(v)$



# Grid-based Implementation: What are 'long' paths?

Path/Query between source $s$ and target $t$ 'long'/non-local

$\Leftrightarrow$

$s$ and $t$ at least 4 grid cells (horiz./vert.) apart

# Grid-based Implementation: Remarks

- The construction as described would take days to weeks on the US roadmap $\Rightarrow$ more efficient construction via *sweep algorithm* takes around 10 h

# Grid-based Implementation: Remarks

- The construction as described would take days to weeks on the US roadmap $\Rightarrow$ more efficient construction via *sweep algorithm* takes around 10 h
- Queries for pairs $(s, t)$ less than 4 grid cells apart are handled by a variation of reach-based Dijkstra

# Grid-based Implementation: Remarks

- The construction as described would take days to weeks on the US roadmap $\Rightarrow$ more efficient construction via *sweep algorithm* takes around 10 h
- Queries for pairs $(s, t)$ less than 4 grid cells apart are handled by a variation of reach-based Dijkstra
- multi-layered implementation also possible

# Grid-based Implementation: Remarks

- The construction as described would take days to weeks on the US roadmap $\Rightarrow$ more efficient construction via *sweep algorithm* takes around 10 h
- Queries for pairs $(s, t)$ less than 4 grid cells apart are handled by a variation of reach-based Dijkstra
- multi-layered implementation also possible
- can be made very space-efficient

# Experiments: US roadmap ($n = 24$ Mio, $m = 58$ Mio)

| Grid | $\lvert\mathcal{T}\rvert$ | $\frac{\lvert\mathcal{T}\rvert \times \lvert\mathcal{T}\rvert}{\text{node}}$ | avg. $\lvert\mathcal{A}\rvert$ | % 'long' queries | construction of transit nodes |
|---|---|---|---|---|---|
| $64 \times 64$ | 2 042 | 0.1 | 11.4 | 91.7% | 498 min |
| **$128 \times 128$** | **7 426** | **1.1** | **11.4** | **97.4%** | **525 min** |
| **$256 \times 256$** | **24 899** | **12.8** | **10.6** | **99.2%** | **638 min** |
| $512 \times 512$ | 89 382 | 164.6 | 9.7 | 99.8% | 859 min |
| $1\,024 \times 1\,024$ | 351 484 | 2 545.5 | 9.1 | 99.9% | 964 min |

Statistics for several grid sizes.

# Experiments: US roadmap ($n = 24$ Mio, $m = 58$ Mio)

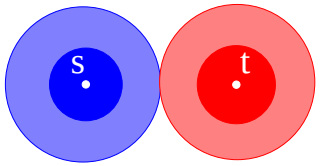
| Grid | $\lvert\mathcal{T}\rvert$ | $\frac{\lvert\mathcal{T}\rvert \times \lvert\mathcal{T}\rvert}{\text{node}}$ | avg. $\lvert\mathcal{A}\rvert$ | % 'long' queries | construction of transit nodes |
|---|---|---|---|---|---|
| $64 \times 64$ | 2 042 | 0.1 | 11.4 | 91.7% | 498 min |
| **$128 \times 128$** | **7 426** | **1.1** | **11.4** | **97.4%** | **525 min** |
| **$256 \times 256$** | **24 899** | **12.8** | **10.6** | **99.2%** | **638 min** |
| $512 \times 512$ | 89 382 | 164.6 | 9.7 | 99.8% | 859 min |
| $1\,024 \times 1\,024$ | 351 484 | 2 545.5 | 9.1 | 99.9% | 964 min |

Statistics for several grid sizes.

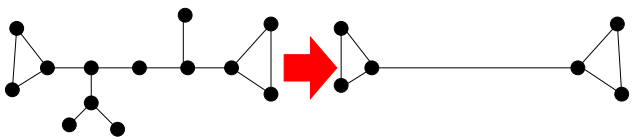
| non-local (99%) | local (1%) | all queries | preproc. | space/node |
|---|---|---|---|---|
| $12\mu s$ | $5112\mu s$ | $63\mu s$ | 20h | **21 bytes** |

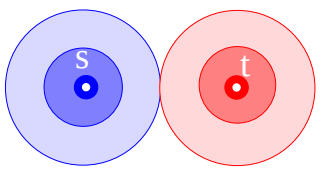
Results for 2-layer data structure.

# Highway Hierarchies [Sanders/Schultes ESA'05/'06]




- Gutman's insight with 2nd metric = Dijkstra rank
- complete search within a local area
- identify *highway network* = minimal subgraph that preserves all 'long' shortest paths
- contract network
- iterate $\Rightarrow$ *highway hierarchy*





# HH-based Transit Node Routing

Second attempt – a more sophisticated realization of the transit node routing idea.

- Long paths have to use higher levels in the HH
  $\Rightarrow$ high level in HH canonical choice for transit nodes $\mathcal{T}$

# HH-based Transit Node Routing

Second attempt – a more sophisticated realization of the transit node routing idea.

- Long paths have to use higher levels in the HH
  $\Rightarrow$ high level in HH canonical choice for transit nodes $\mathcal{T}$
- Using several levels from HH induce multi-layer solution in a very natural way

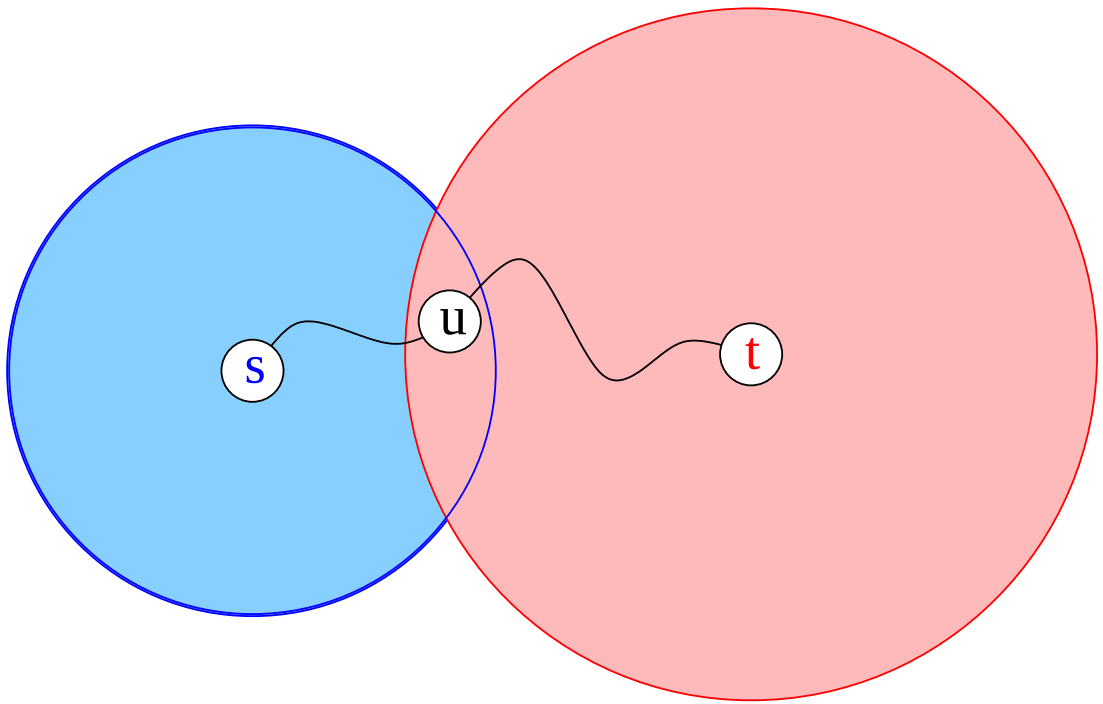
# HH-based Transit Node Routing

Second attempt – a more sophisticated realization of the transit node routing idea.

- Long paths have to use higher levels in the HH
  $\Rightarrow$ high level in HH canonical choice for transit nodes $\mathcal{T}$
- Using several levels from HH induce multi-layer solution in a very natural way
- Local queries can be handled very efficiently by HH ($\Leftrightarrow$ grid-based approach)

# HH-based Transit Node Routing: What are 'long' paths?

Compute for each node $v$ a radius $r(v)$ such that a query $(s, t)$ is non-local/the path is considered long, if respective disks with radii $r(s)$ and $r(t)$ do **not** overlap.



# HH-based Implementation: Remarks

- For the fastest variant in terms of query times, levels 4, 2, 1 of the HH are used as 3 layers

# HH-based Implementation: Remarks

- For the fastest variant in terms of query times, levels 4, 2, 1 of the HH are used as 3 layers
- A more economical version in terms of storage space and preprocessing times uses levels 5 and 3 in a 2-layer structure

# HH-based Implementation: Remarks

- For the fastest variant in terms of query times, levels 4, 2, 1 of the HH are used as 3 layers
- A more economical version in terms of storage space and preprocessing times uses levels 5 and 3 in a 2-layer structure
- Implemented and benchmarked also for directed graphs, e.g. roadmap of Europe, and *distances* instead of travel times

# Experiments: US roadmap ($n = 24$ Mio, $m = 58$ Mio)

Preprocessing:

| | variant | layer 1 $\lvert\mathcal{T}\rvert$ | $\lvert\mathcal{A}\rvert$ | layer 2 $\lvert\mathcal{T}_2\rvert$ | $\lvert\mathcal{A}_2\rvert$ | space [B/node] | time [h] |
|---|---|---|---|---|---|---|---|
| USA | eco | 12 111 | 6.1 | 184 379 | 4.9 | 111 | *0:59* |
| | gen | 10 674 | 5.7 | 485 410 | 4.2 | 244 | 3:25 |

# Experiments: US roadmap ($n = 24$ Mio, $m = 58$ Mio)

Preprocessing:

| | variant | layer 1 $\lvert\mathcal{T}\rvert$ | layer 1 $\lvert\mathcal{A}\rvert$ | layer 2 $\lvert\mathcal{T}_2\rvert$ | layer 2 $\lvert\mathcal{A}_2\rvert$ | space [B/node] | time [h] |
|---|---|---|---|---|---|---|---|
| | eco | 12 111 | 6.1 | 184 379 | 4.9 | 111 | *0:59* |
| USA | gen | 10 674 | 5.7 | 485 410 | 4.2 | 244 | 3:25 |

Query:

| | variant | layer 1 [%] wrong | layer 1 [%] cont'd | layer 2 [%] wrong | layer 2 [%] cont'd | time |
|---|---|---|---|---|---|---|
| | eco | 0.14 | 1.13 | 0.0064 | 0.2780 | 11.5 $\mu$s |
| USA | gen | 0.11 | 0.80 | 0.0014 | 0.0138 | *4.9 $\mu$s* |

# Conclusions

- new, extremely useful insight which improves avg. query times by *orders of magnitude*: Microseconds vs. Milliseconds!

# Conclusions

- new, extremely useful insight which improves avg. query times by *orders of magnitude*: Microseconds vs. Milliseconds!
- we presented two ways to exploit this insight:
  - simple, grid-based implementation tuned for space efficiency ($63\mu s$ avg.query; 20h preprocessing; 21 bytes/node)
  - sophisticated, extremely fast implementation based on HH ($5\mu s$; $< 4$h; 244 bytes/node); fast for *all* types of queries

# Conclusions

- new, extremely useful insight which improves avg. query times by *orders of magnitude*: Microseconds vs. Milliseconds!
- we presented two ways to exploit this insight:
  - simple, grid-based implementation tuned for space efficiency ($63\mu s$ avg.query; 20h preprocessing; 21 bytes/node)
  - sophisticated, extremely fast implementation based on HH ($5\mu s$; $< 4$h; 244 bytes/node); fast for *all* types of queries
- not mentioned: efficient ways to output actual paths

# Conclusions

- new, extremely useful insight which improves avg. query times by *orders of magnitude*: Microseconds vs. Milliseconds!
- we presented two ways to exploit this insight:
  - simple, grid-based implementation tuned for space efficiency ($63\mu s$ avg.query; 20h preprocessing; 21 bytes/node)
  - sophisticated, extremely fast implementation based on HH ($5\mu s$; $< 4$h; 244 bytes/node); fast for *all* types of queries
- not mentioned: efficient ways to output actual paths
- room for improvement in both realizations: grid-based in terms of query/preprocessing, HH-based in terms of space consumption

# Conclusions

- new, extremely useful insight which improves avg. query times by *orders of magnitude*: Microseconds vs. Milliseconds!
- we presented two ways to exploit this insight:
  - simple, grid-based implementation tuned for space efficiency ($63\mu s$ avg.query; 20h preprocessing; 21 bytes/node)
  - sophisticated, extremely fast implementation based on HH ($5\mu s$; $< 4$h; 244 bytes/node); fast for *all* types of queries
- not mentioned: efficient ways to output actual paths
- room for improvement in both realizations: grid-based in terms of query/preprocessing, HH-based in terms of space consumption
- One of the main challenges: deal with dynamics of real-world networks

# Thank you for your attention!