



# **Transit Node Routing** **based on Highway Hierarchies**

**Peter Sanders**

**Dominik Schultes**

Institut für Theoretische Informatik – Algorithmik II

Universität Karlsruhe (TH)

<http://algo2.iti.uka.de/schultes/hwy/>

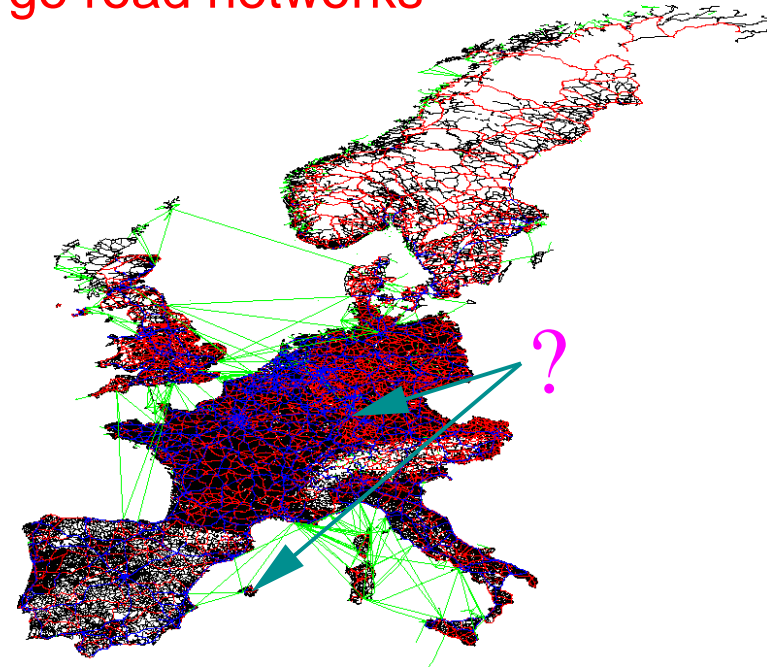
9th DIMACS Challenge, November 14, 2006



# Route Planning

## Goals:

- exact** shortest (i.e. fastest) paths in **large road networks**
- fast queries**
- fast preprocessing**
- low space** consumption



## Applications:

- route planning systems in the internet
- car navigation systems
- ...



## Motivation

### ‘Problem’:

existing solutions are already ‘too fast’.

**Example:** perform a **query** (using hwy. hierarchies):  $\approx 1$  ms

**visualise** the path (using our Java application):  $\approx 400$  ms

### Counter-Argument:

applications that require **a lot of queries** (and only a few paths)

- massive traffic **simulations**
- optimisations in **logistics** systems

*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Copenhagen

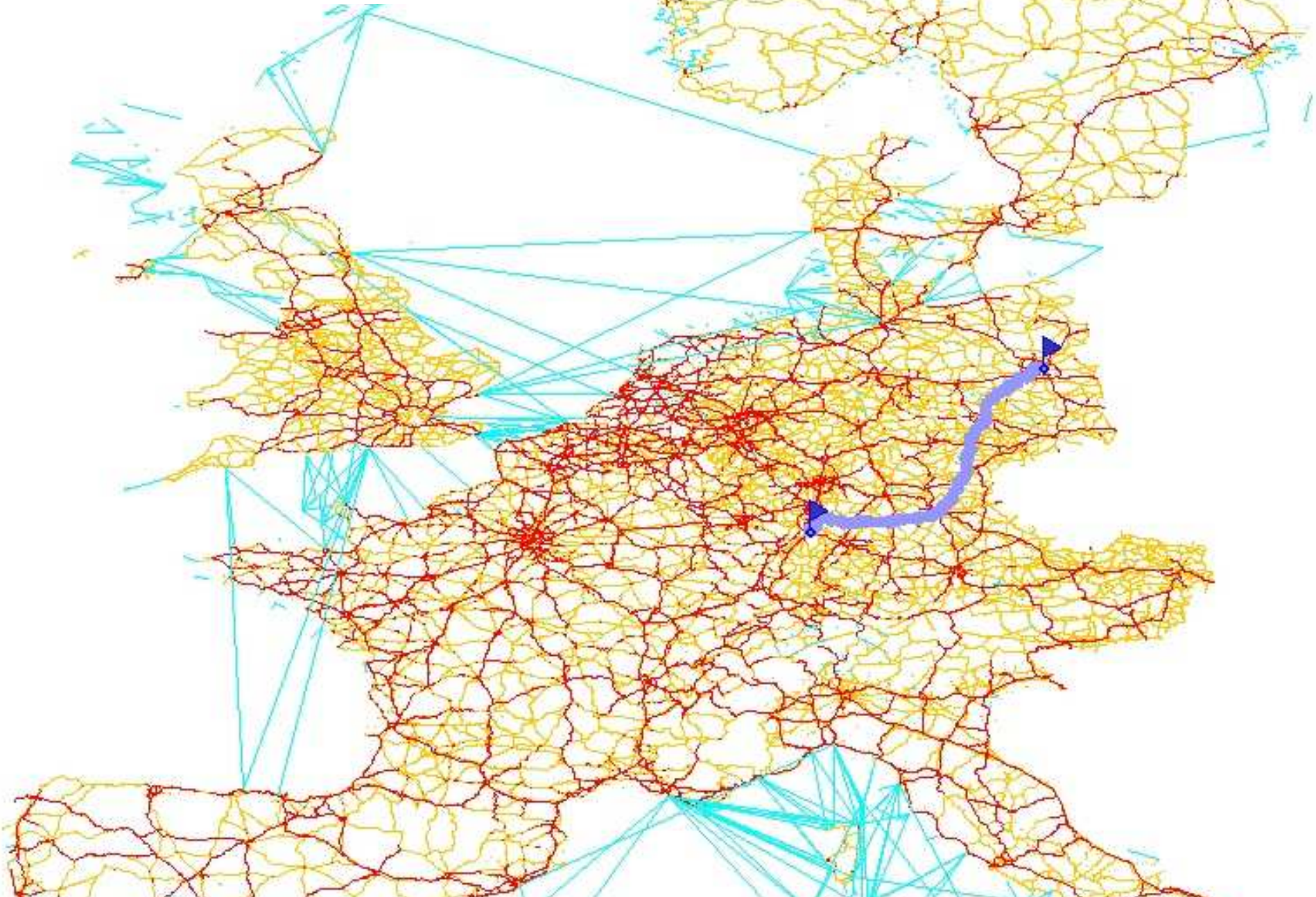




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Berlin





*Sanders/Schultes: Transit Node Routing*

**Example:**

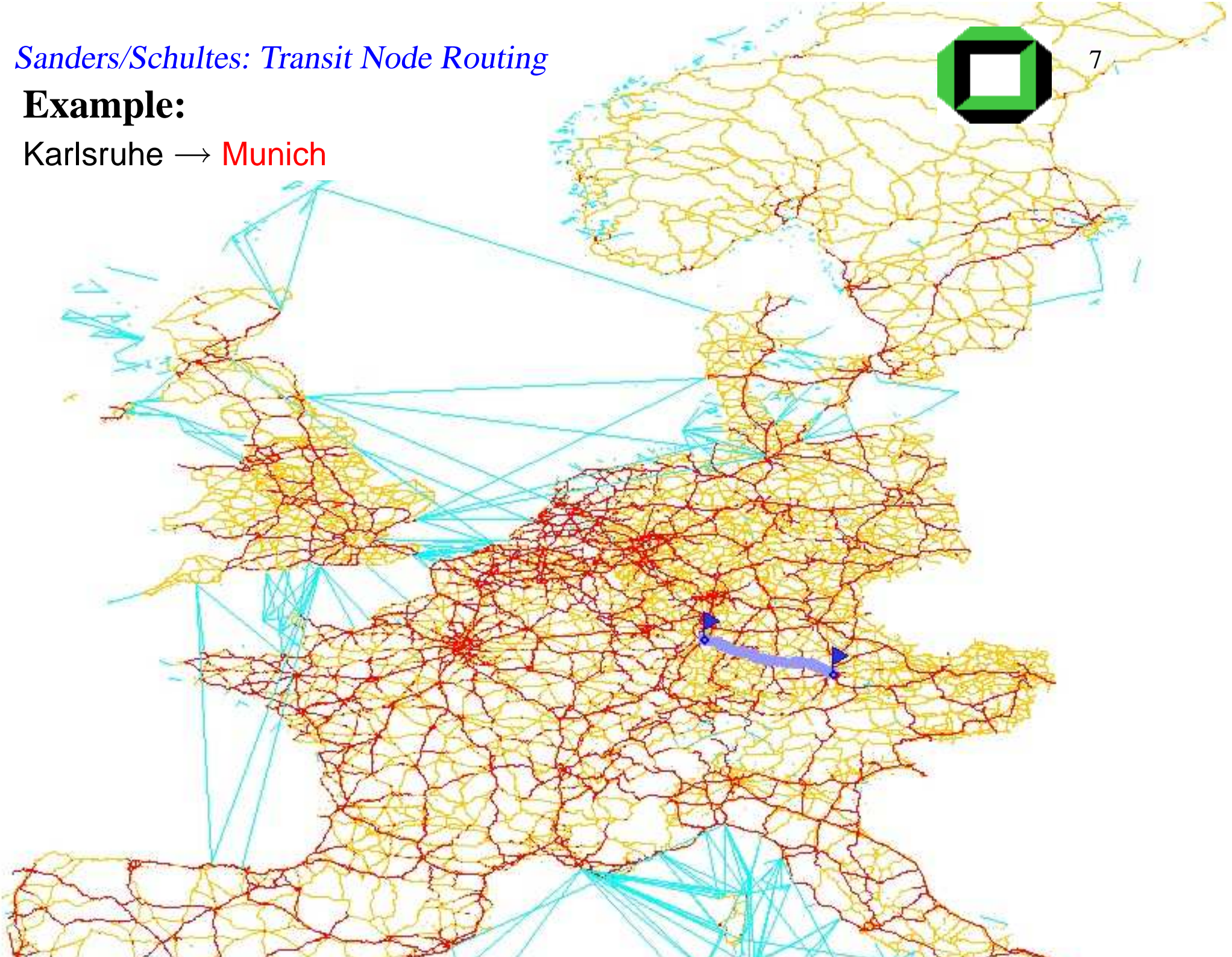
Karlsruhe → Vienna





**Example:**

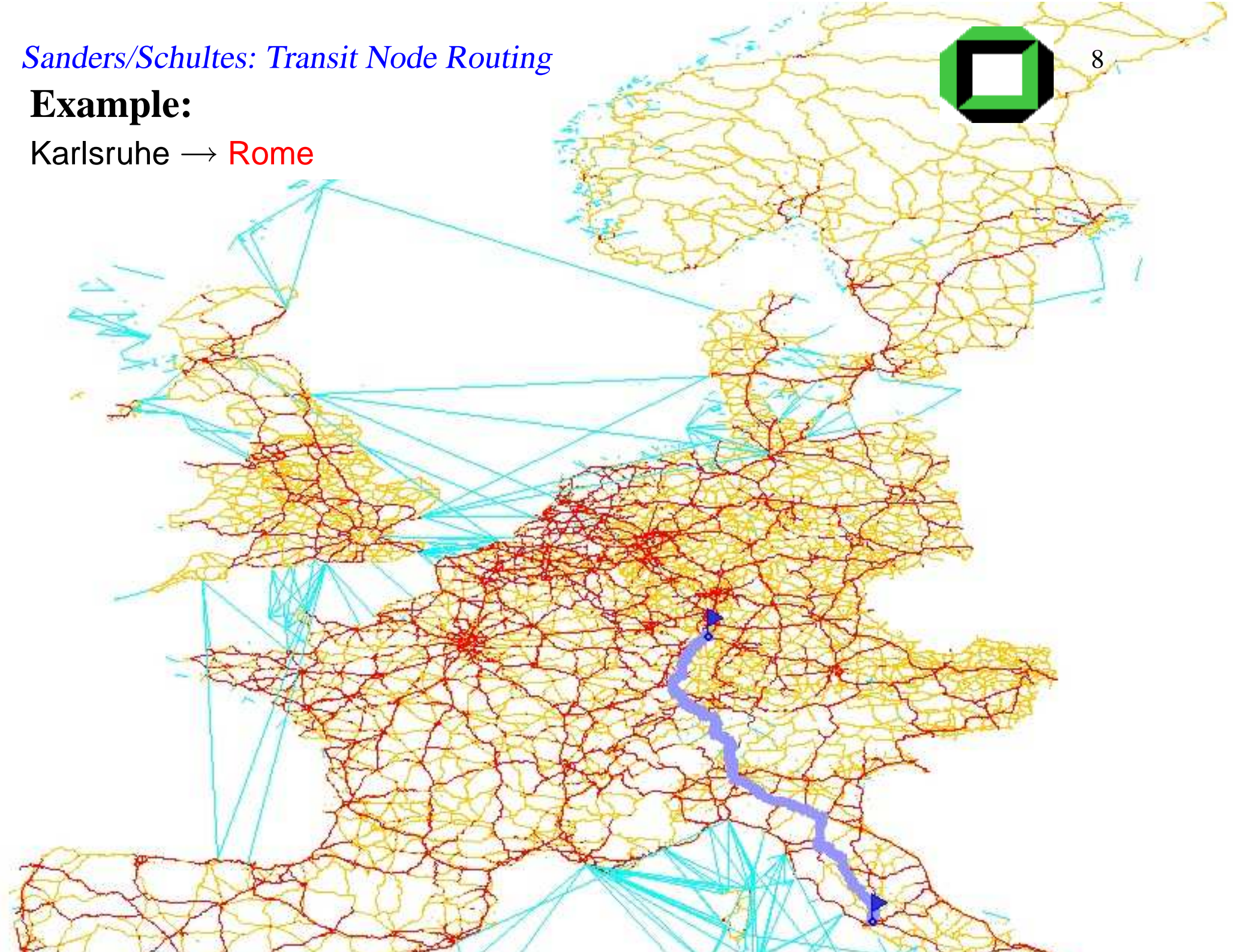
Karlsruhe → **Munich**





**Example:**

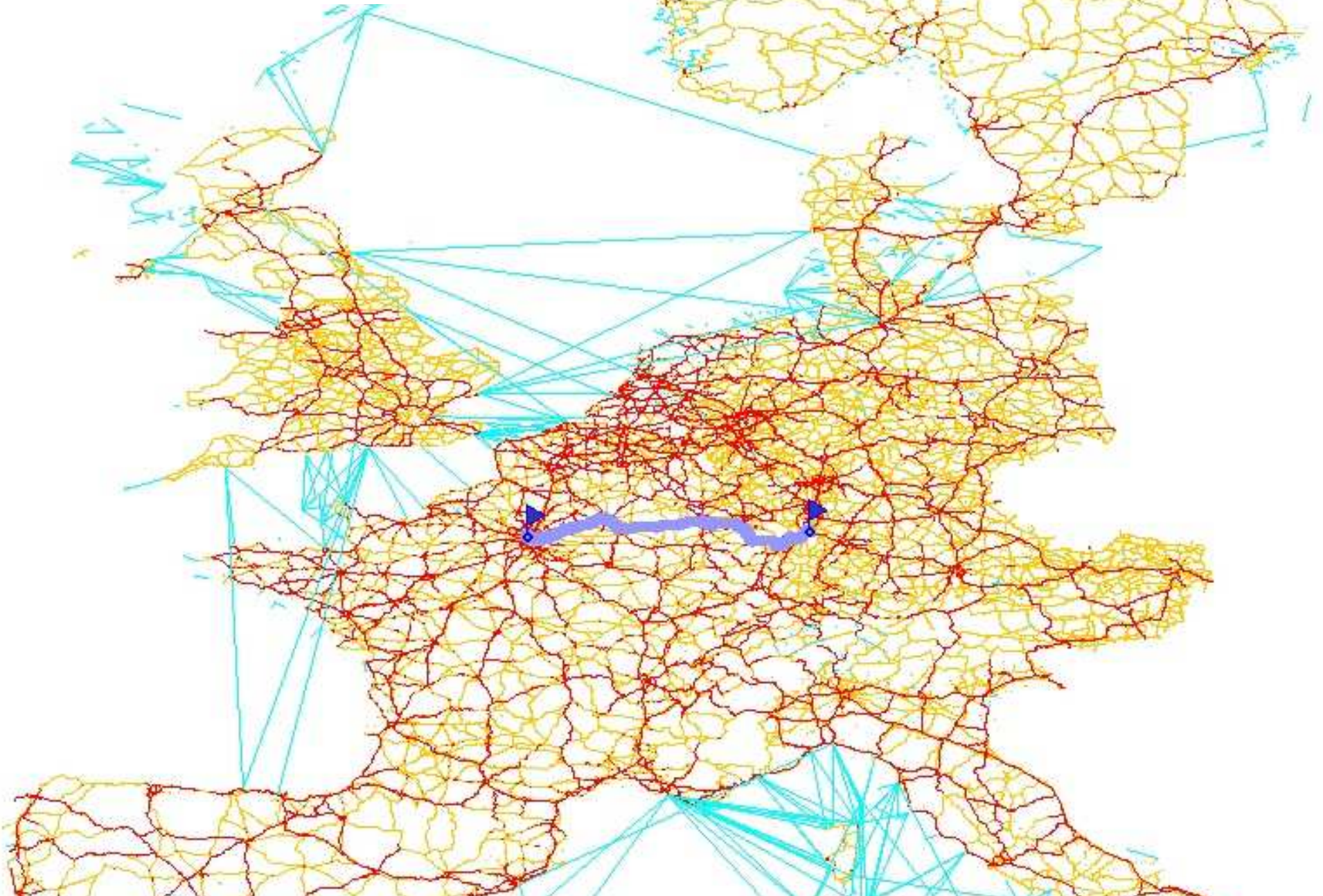
Karlsruhe → Rome





**Example:**

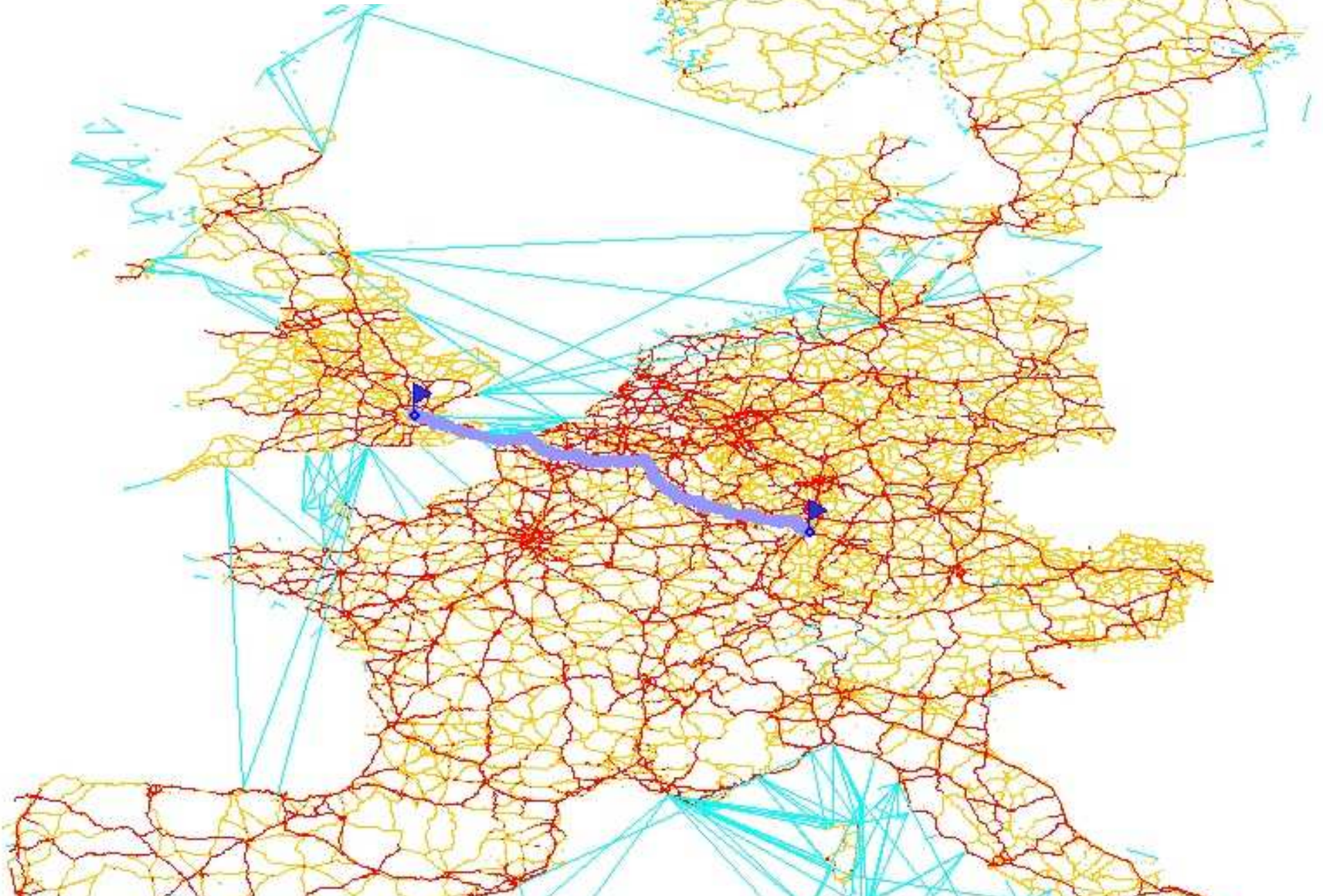
Karlsruhe → Paris





**Example:**

Karlsruhe → London

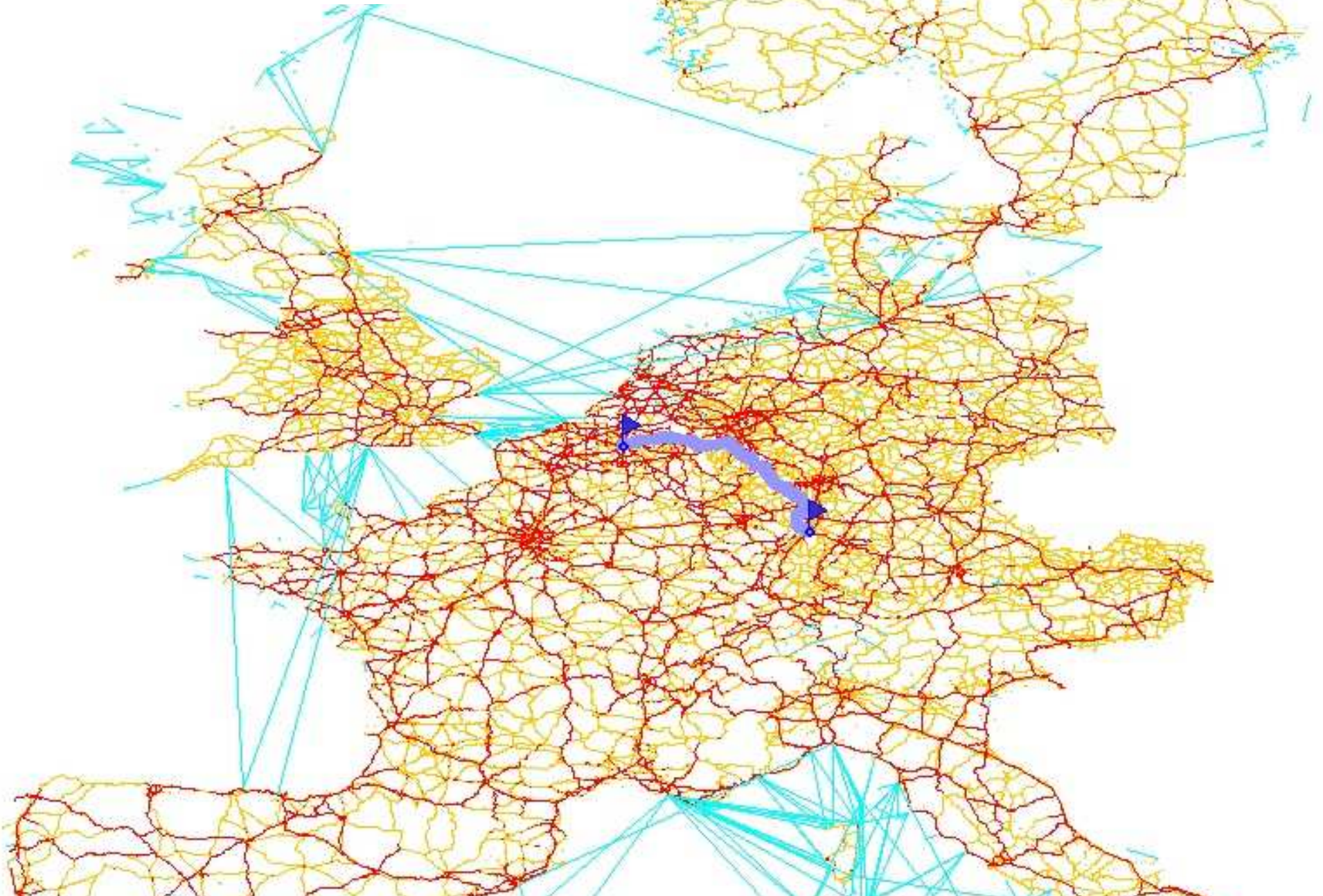






**Example:**

Karlsruhe → **Brussels**

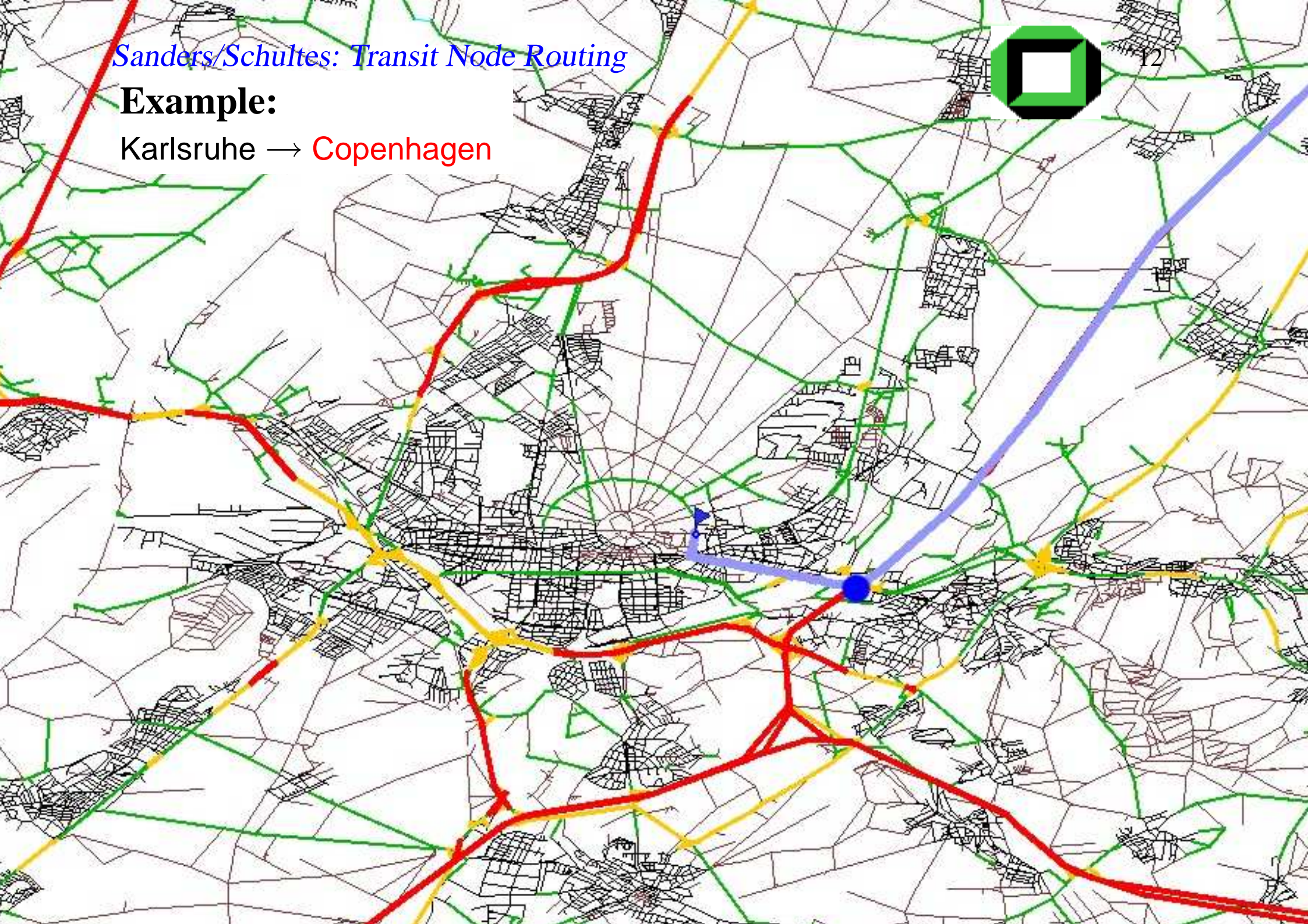




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Copenhagen

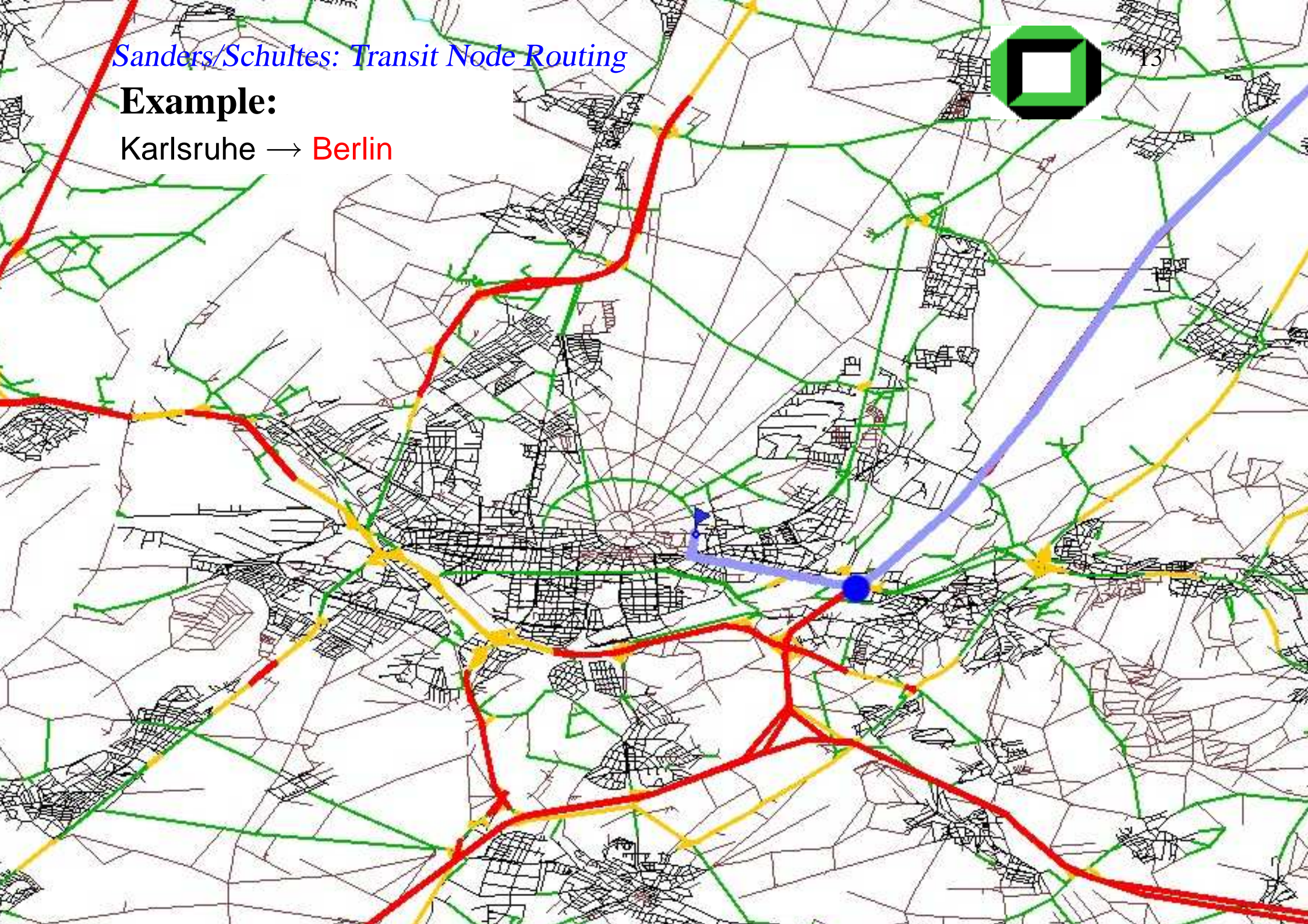




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Berlin

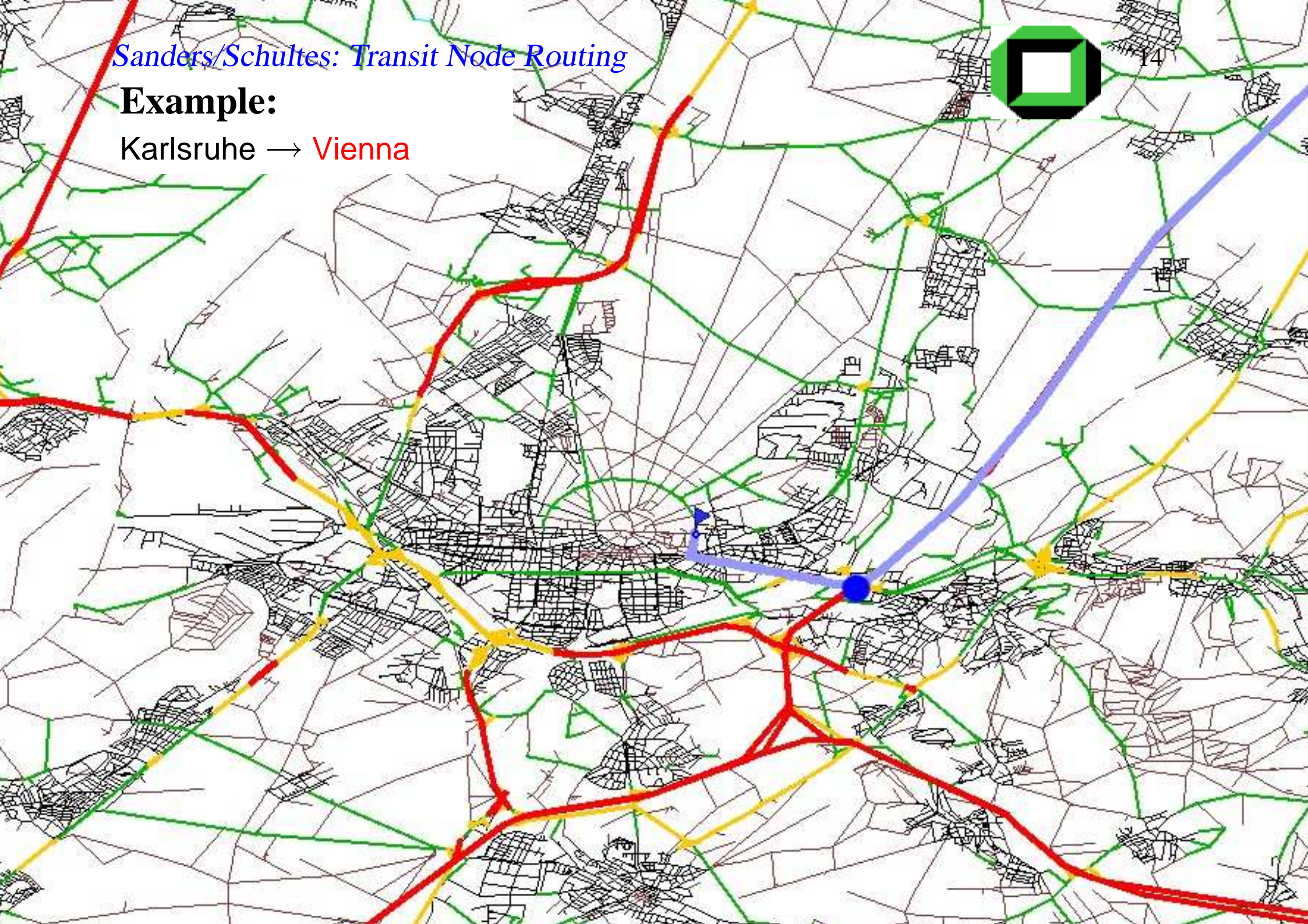




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Vienna

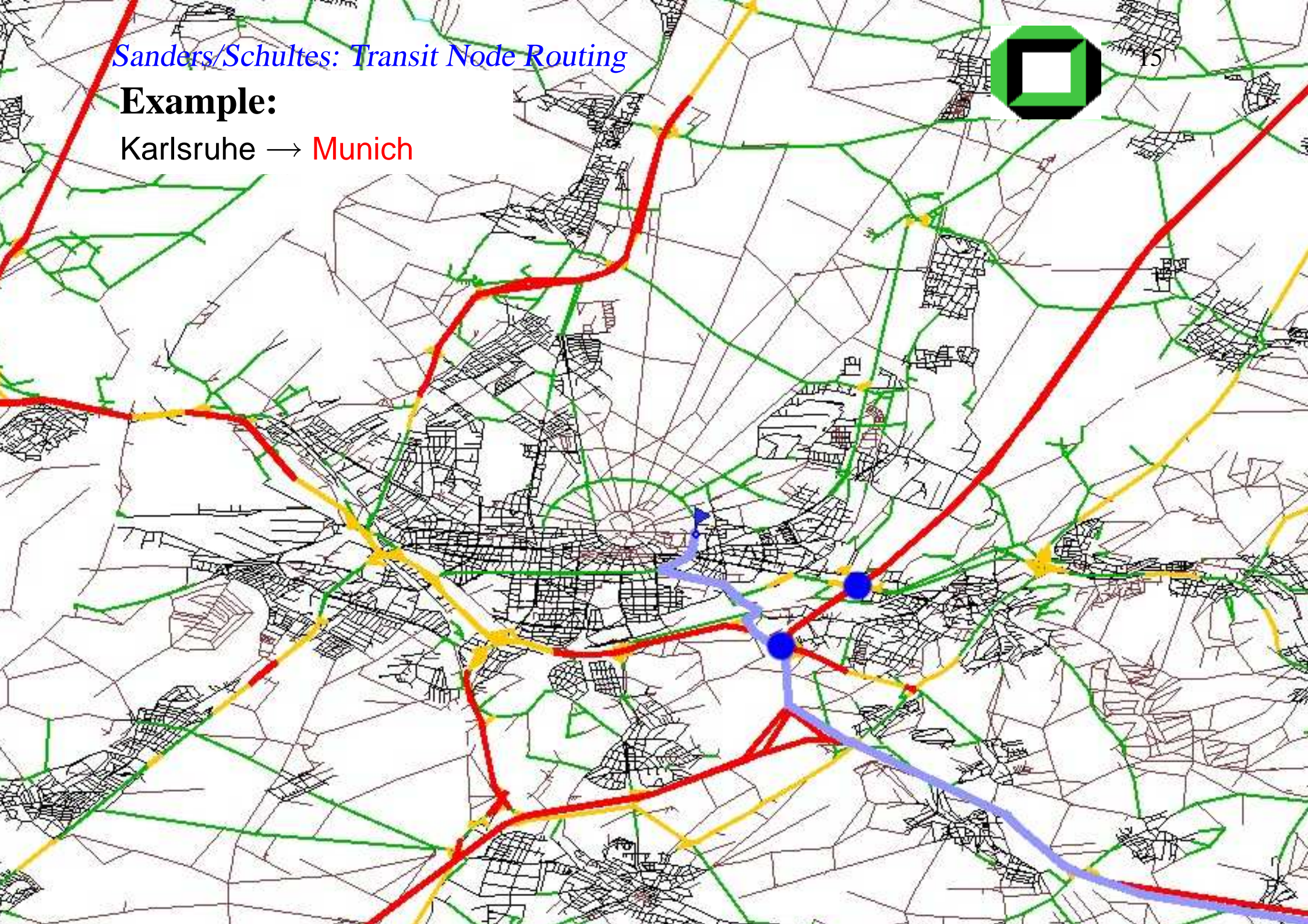




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Munich

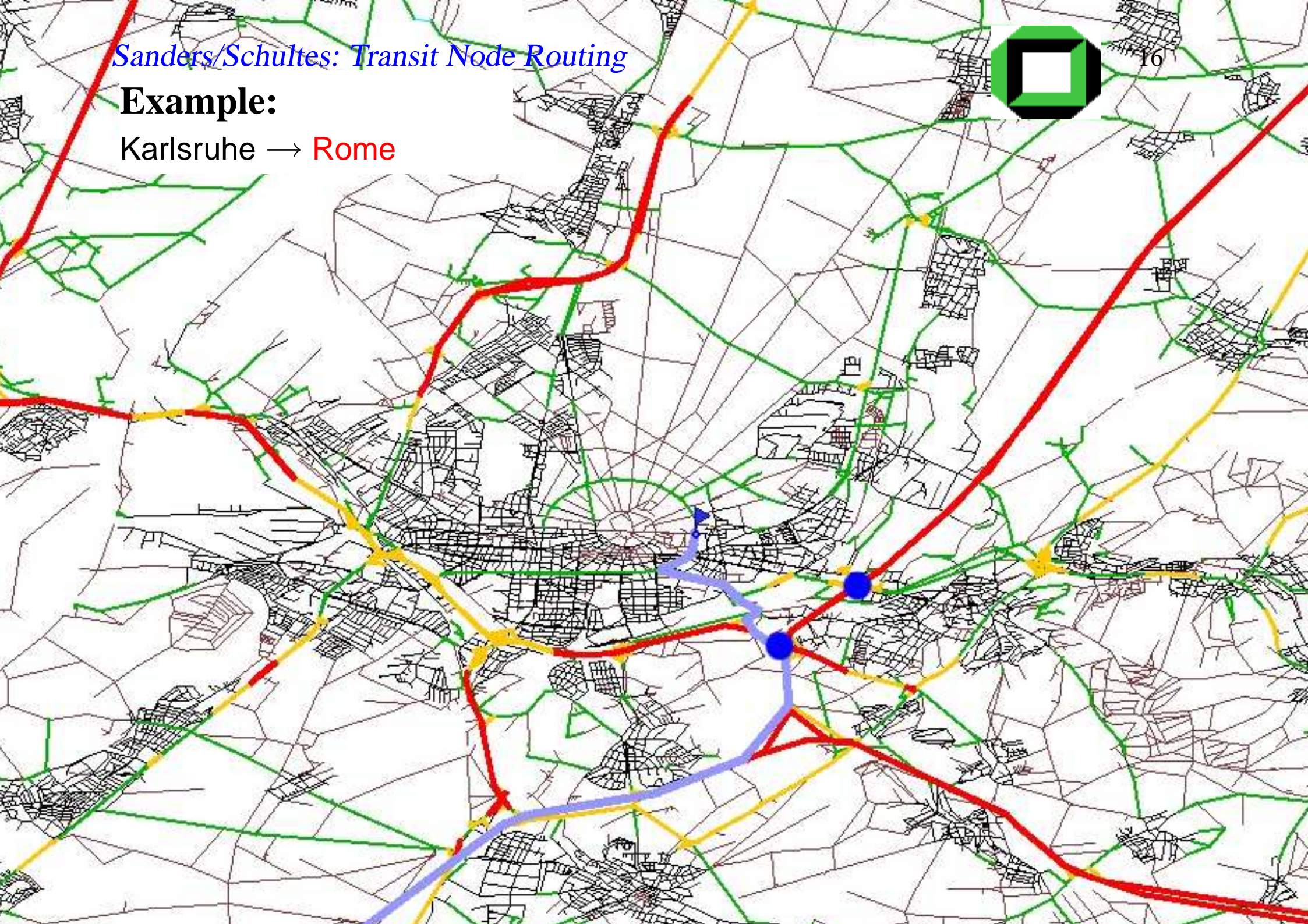




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Rome

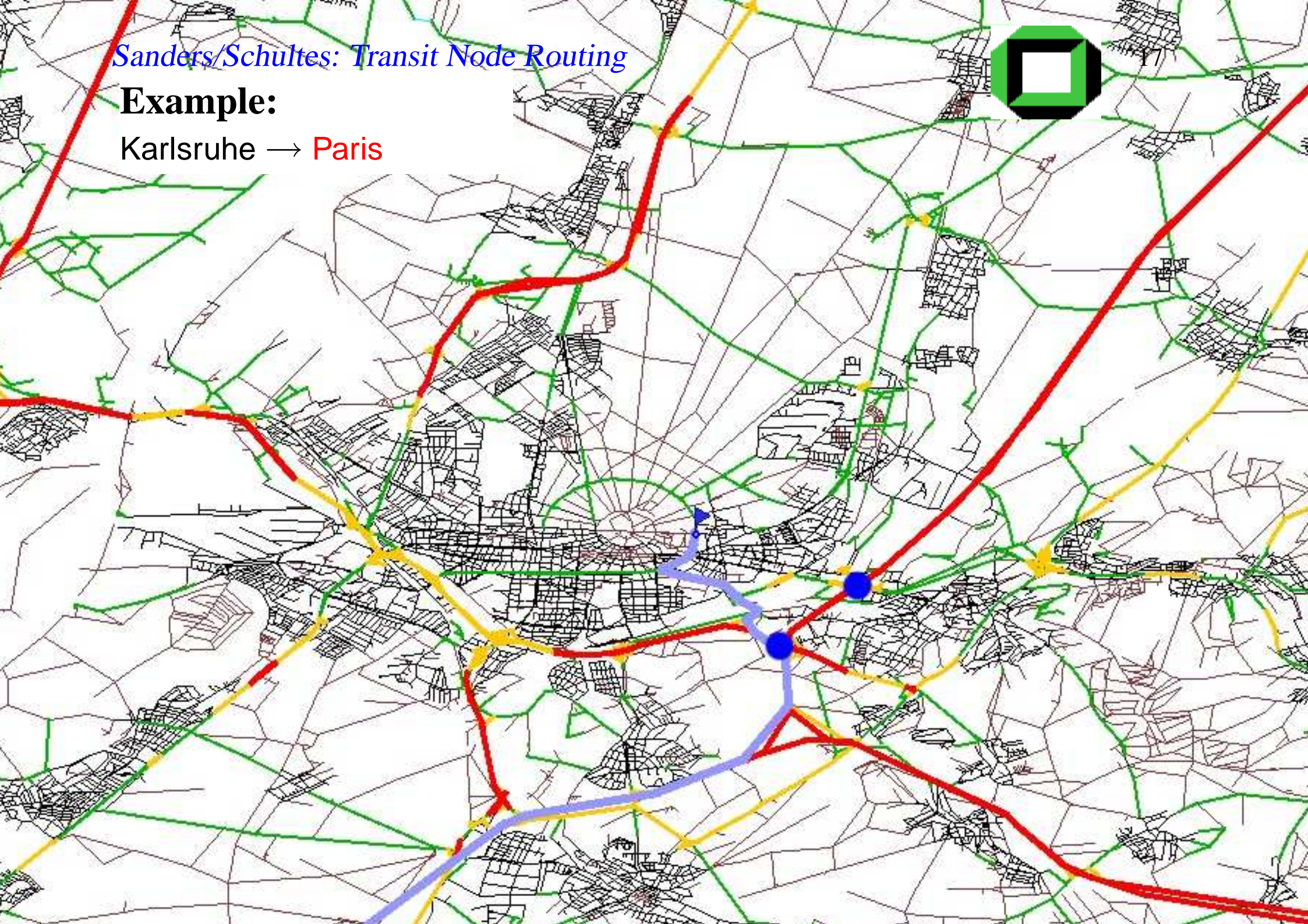




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Paris

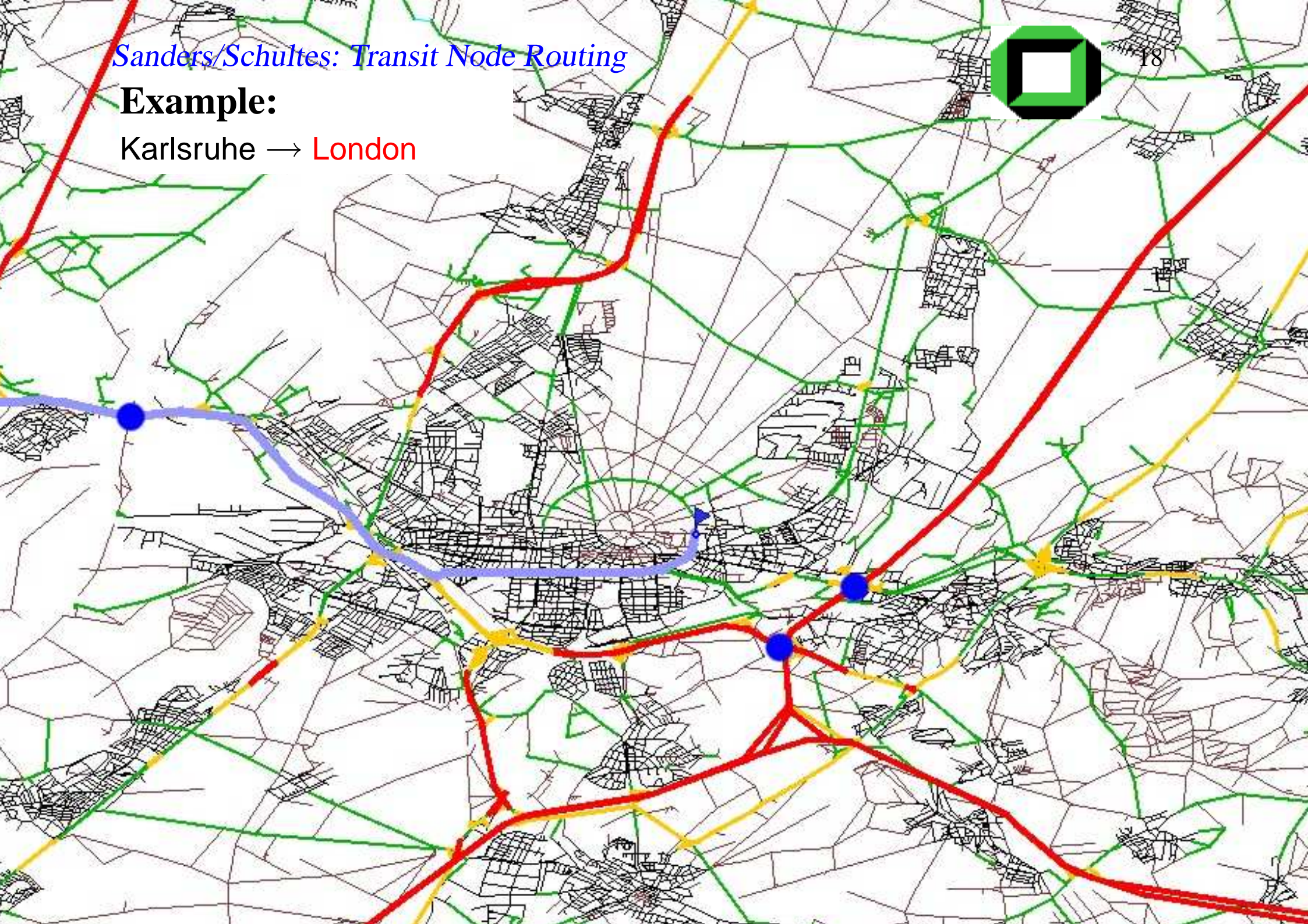




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → London

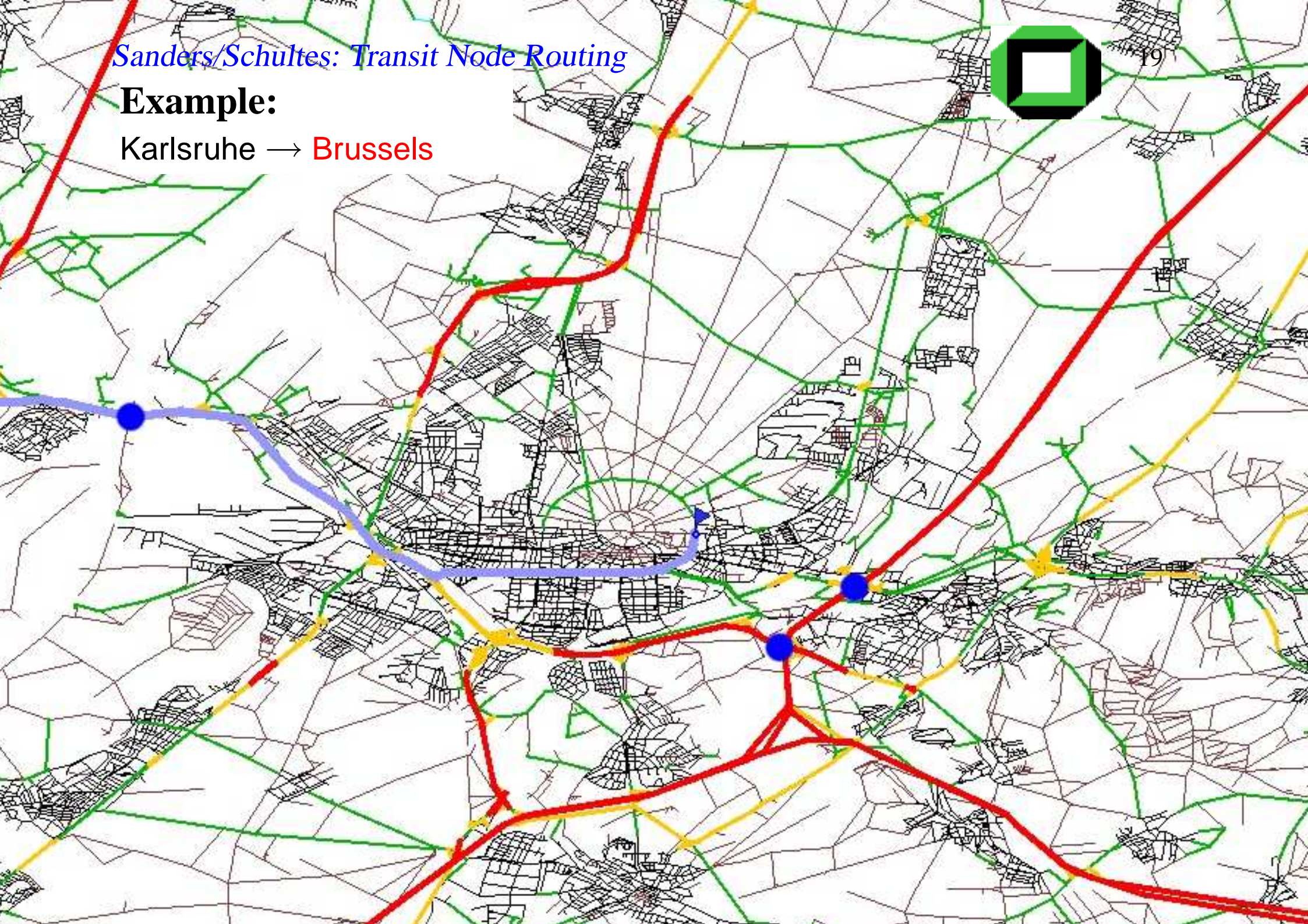




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → **Brussels**







## **First Observation**

**For long-distance travel:** leave current location

via one of only a **few 'important' traffic junctions**,  
called **access points**

( $\rightsquigarrow$  we can afford to store all access points for each node)

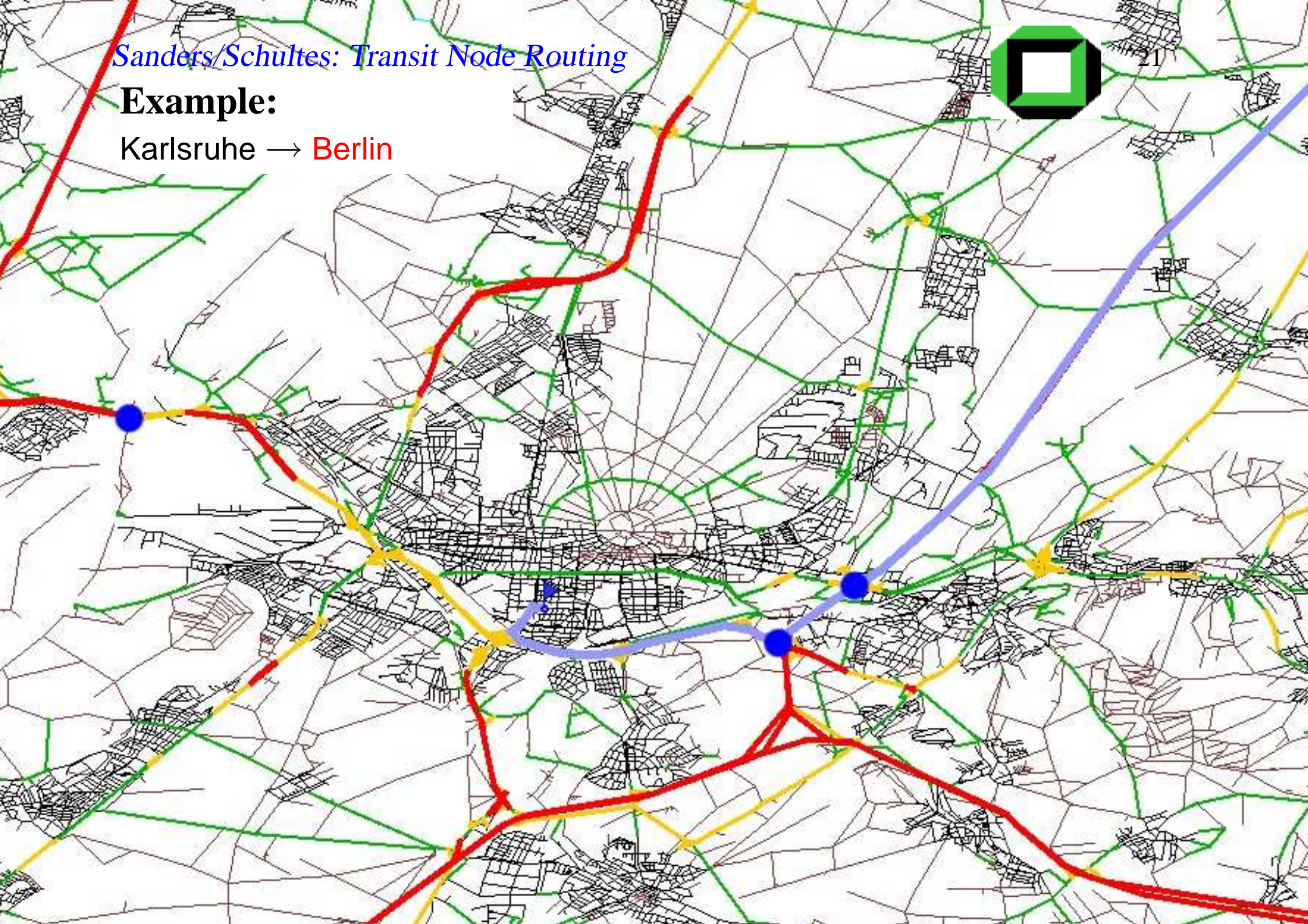
[in Europe: about 10 access points per node on average]



*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Berlin

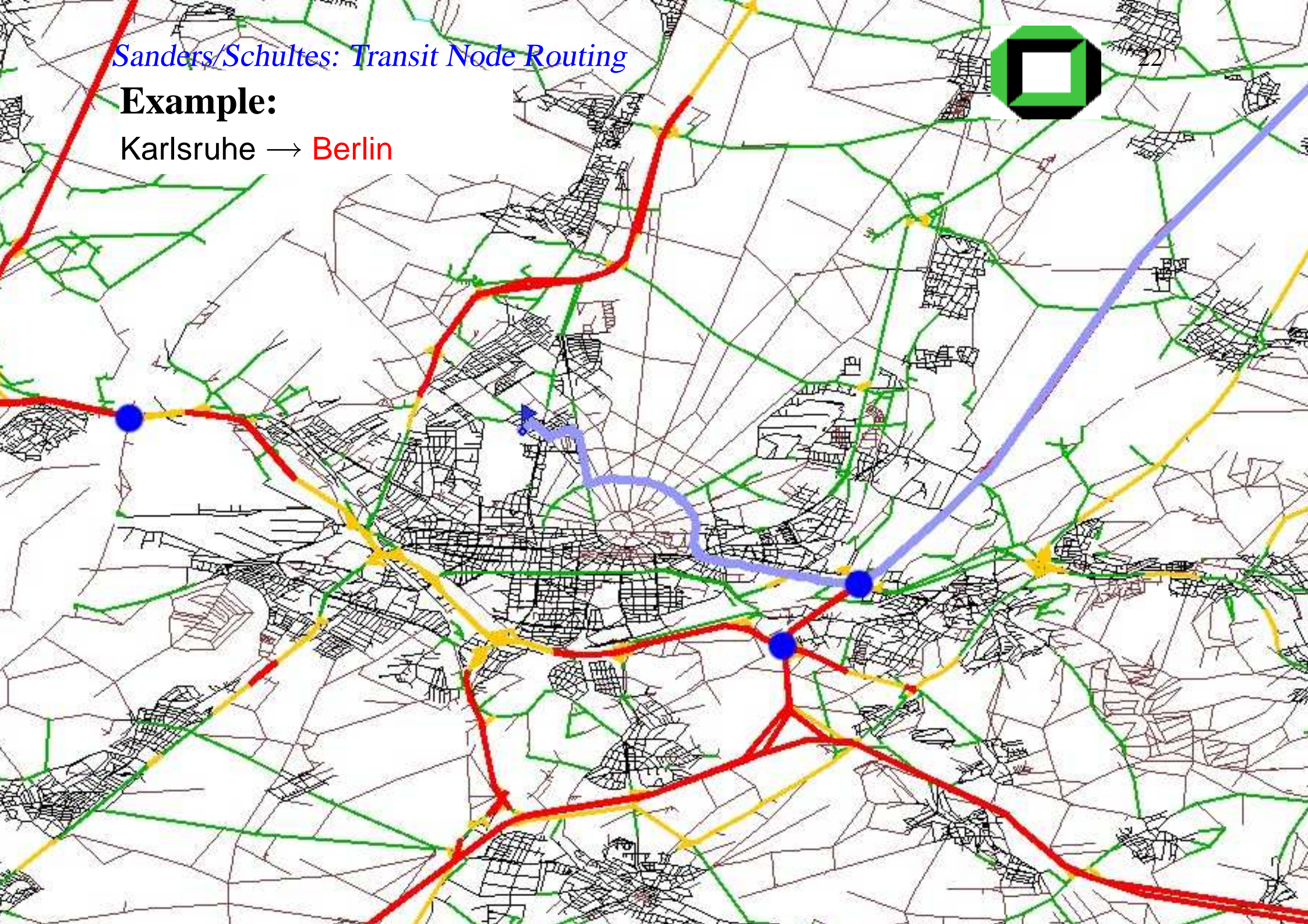




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Berlin

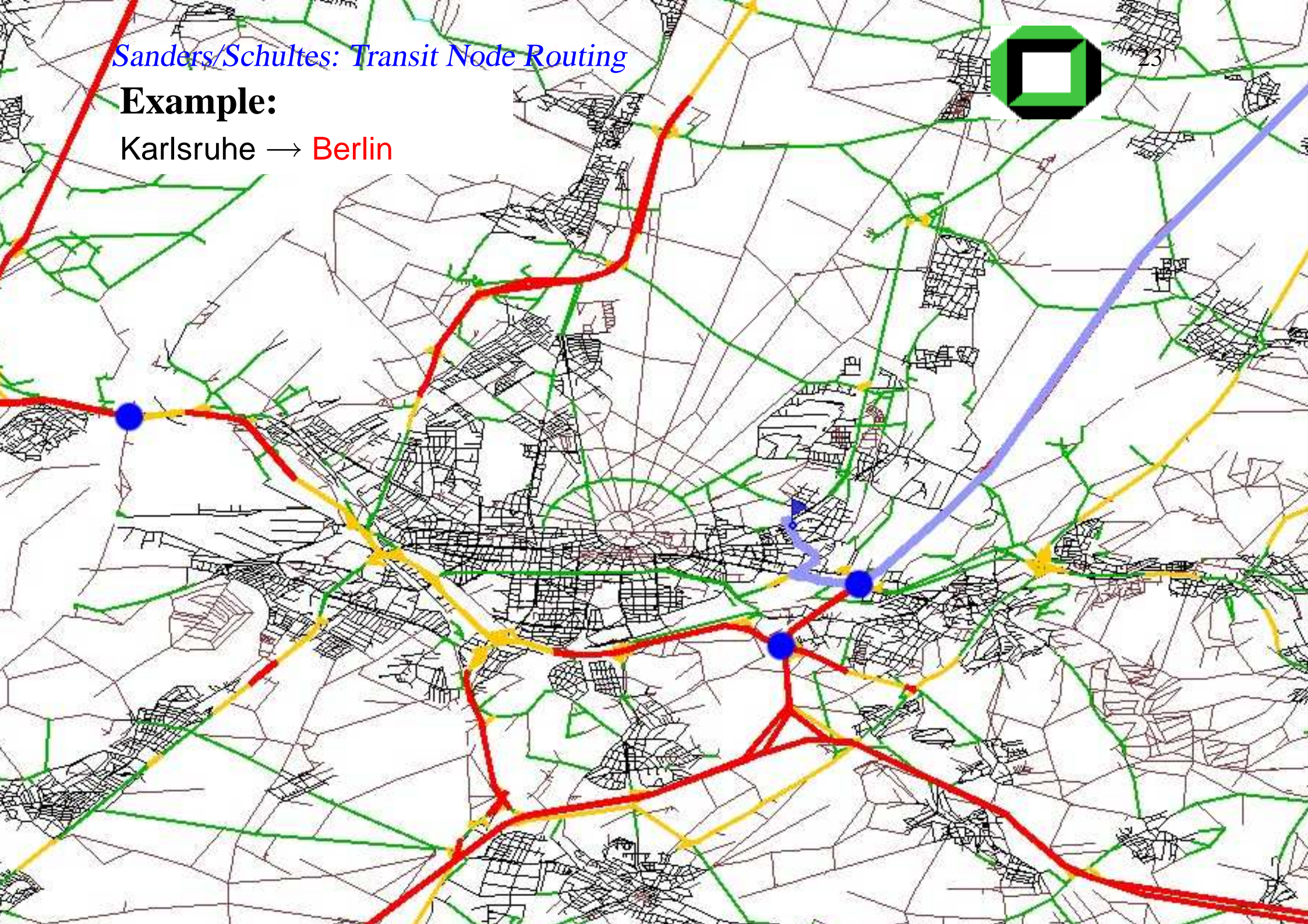




*Sanders/Schultes: Transit Node Routing*

**Example:**

Karlsruhe → Berlin







## **Second Observation**

Each access point is relevant for several nodes.  $\rightsquigarrow$

**union** of the access points of all nodes is **small**,  
called **transit node set**

( $\rightsquigarrow$  we can afford to store the distances between all transit node pairs)

[in Europe: about 10 000 transit nodes]





# Transit Node Routing

## Preprocessing:

- identify **transit node** set  $\mathcal{T} \subseteq V$
- compute complete  $|\mathcal{T}| \times |\mathcal{T}|$  **distance table**
- for each node: identify its **access points** (mapping  $A : V \rightarrow 2^{\mathcal{T}}$ ),  
store the **distances**

**Query** (source  $s$  and target  $t$  given): compute

$$d_{\text{top}}(s, t) := \min \{d(s, u) + d(u, v) + d(v, t) : u \in A(s), v \in A(t)\}$$





## Transit Node Routing

### Locality Filter:

**local** cases must be filtered ( $\rightsquigarrow$  special treatment)

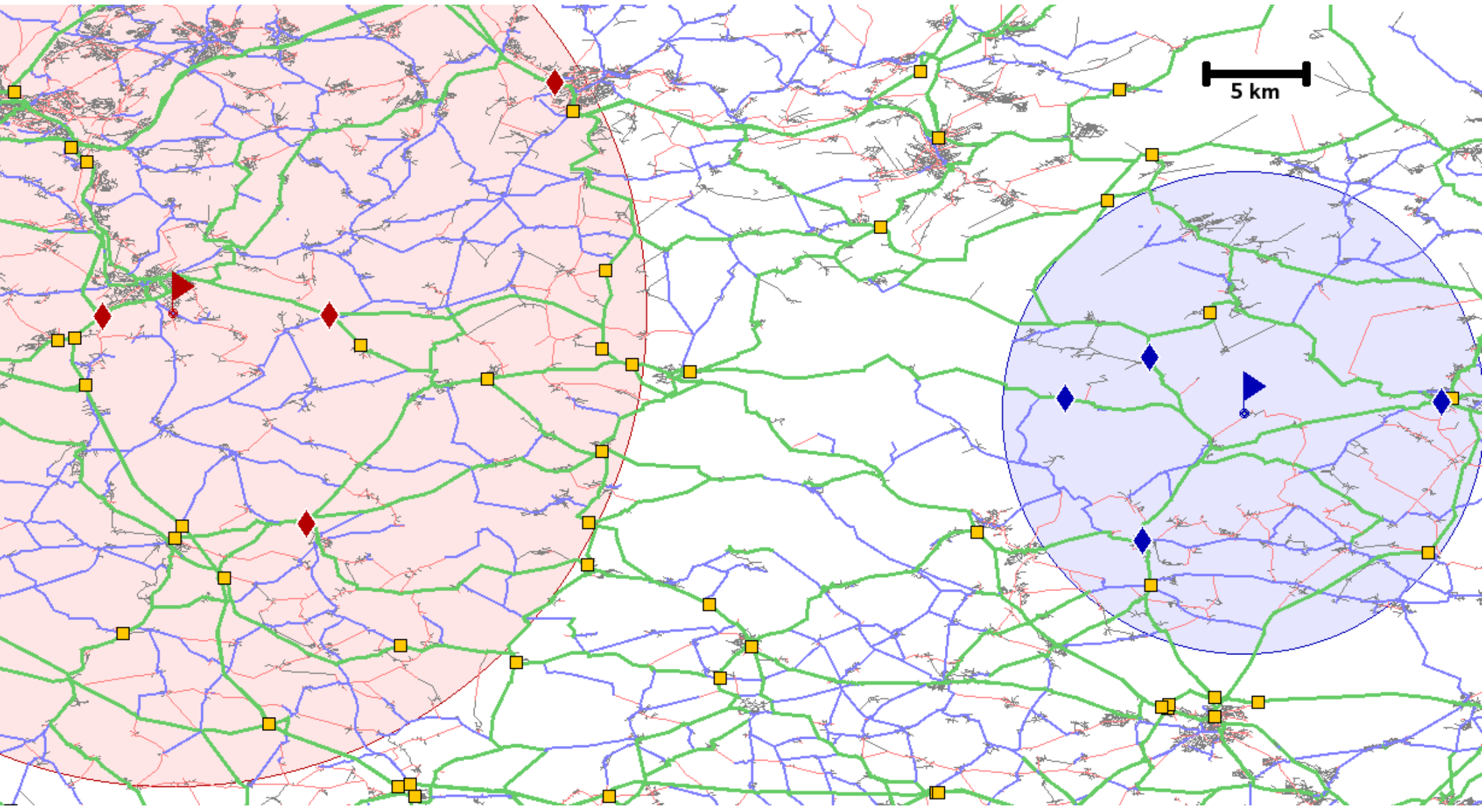
$$L : V \times V \rightarrow \{\text{true}, \text{false}\}$$

$$\neg L(s, t) \text{ implies } d(s, t) = d_{\text{top}}(s, t)$$





# Example







## Related Work

- **separator-based** implementation [Müller et al. 2006]
  - determine separator nodes (= **transit nodes**)
  - partition the graph into small components
  - **access points** of node  $u$ : border nodes of  $u$ 's component
  - **locality filter**: “same component?”
  
- **grid-based** implementation [Bast, Funke, Matijevic 2006]
  - compute geometric subdivision of the network into cells
  - **access points**: border nodes needed for ‘long-distance’ travel
  - **transit nodes**: union of all access points
  - **locality filter**: “less than a certain number of grid cells apart?”

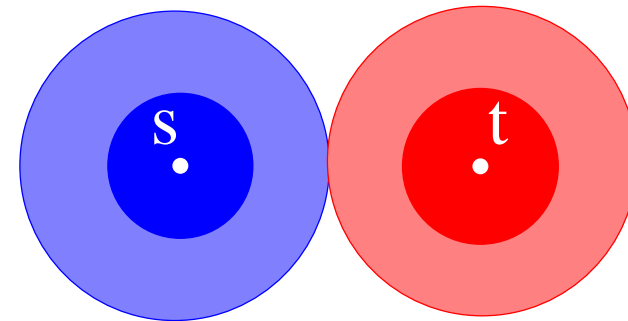




# Our Approach: Highway Hierarchies<sup>1</sup>

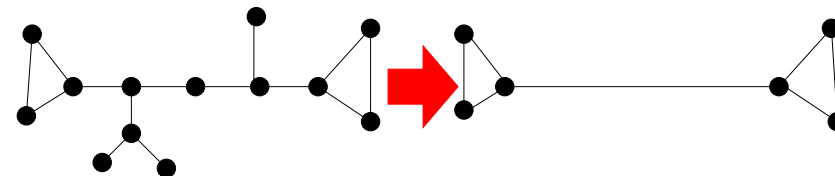
complete search within a local area

search in a (thinner) highway network

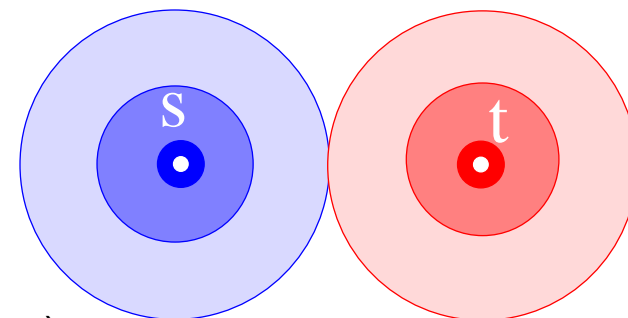


= minimal graph that preserves all shortest paths

contract network, e.g.,



iterate  $\rightsquigarrow$  highway hierarchy



---

<sup>1</sup>presented at ESA 2005 and ESA 2006 (and yesterday)





## Distance Table

- Compute an all-pairs distance table for the core of the topmost level  $\ell$ .  $13\,465 \times 13\,465$  entries
- Abort the search when all entrance points in the core of level  $\ell$  have been encountered.  $\approx 55$  for each direction
- Use the distance table to bridge the gap.  $\approx 55 \times 55$  entries





## HH-based Transit Node Routing

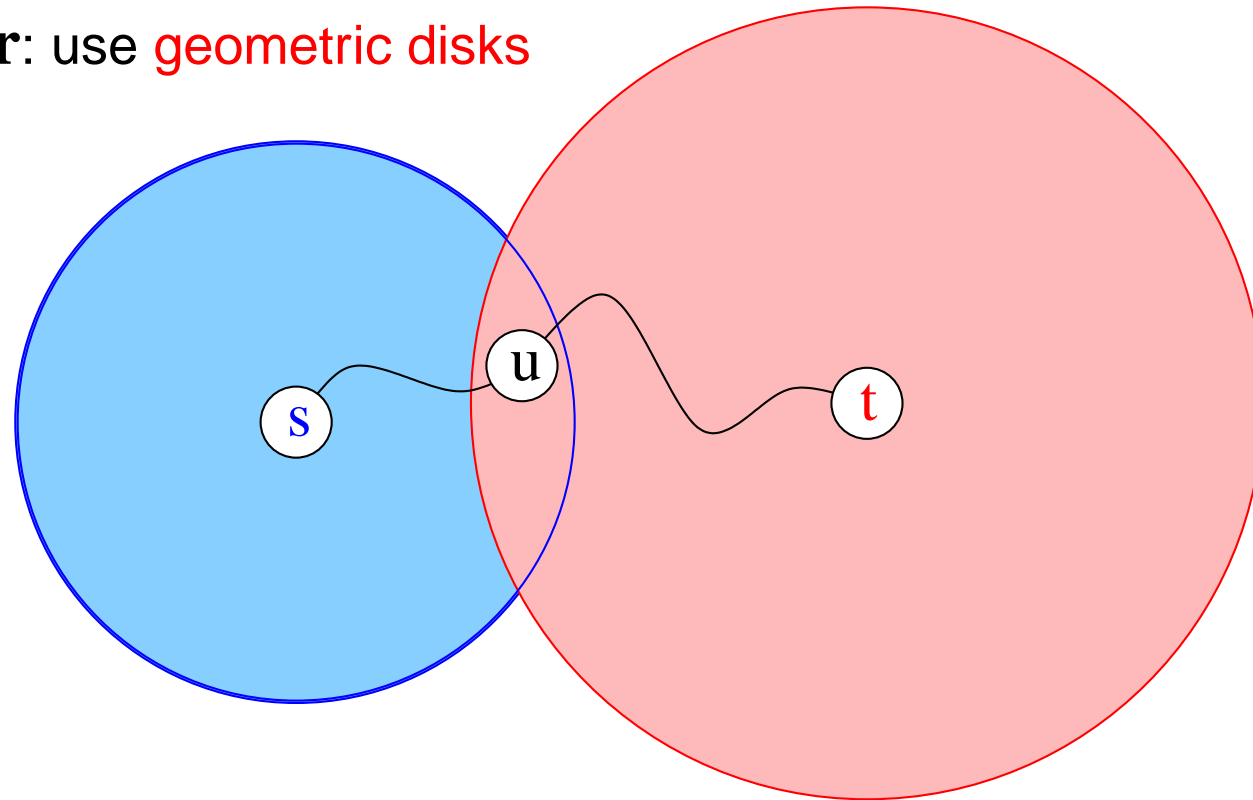
- Compute an all-pairs distance table for the core of the topmost level  $\ell$ . 13 465 × 13 465 entries  
transit node set  $\mathcal{T}$
- Abort the search when all entrance points in the core of level  $\ell$  have been encountered.  $\approx 55$  for each direction  
do not 'search', just perform look-ups
- Use the distance table to bridge the gap.  $\approx 55 \times 55$  entries





# Missing Pieces

1. **locality filter**: use **geometric disks**



$$d(s, t) = d_u(s, t) < d_{\text{top}}(s, t)$$

$L(s, t) :=$  “disks of  $s$  and  $t$  overlap”



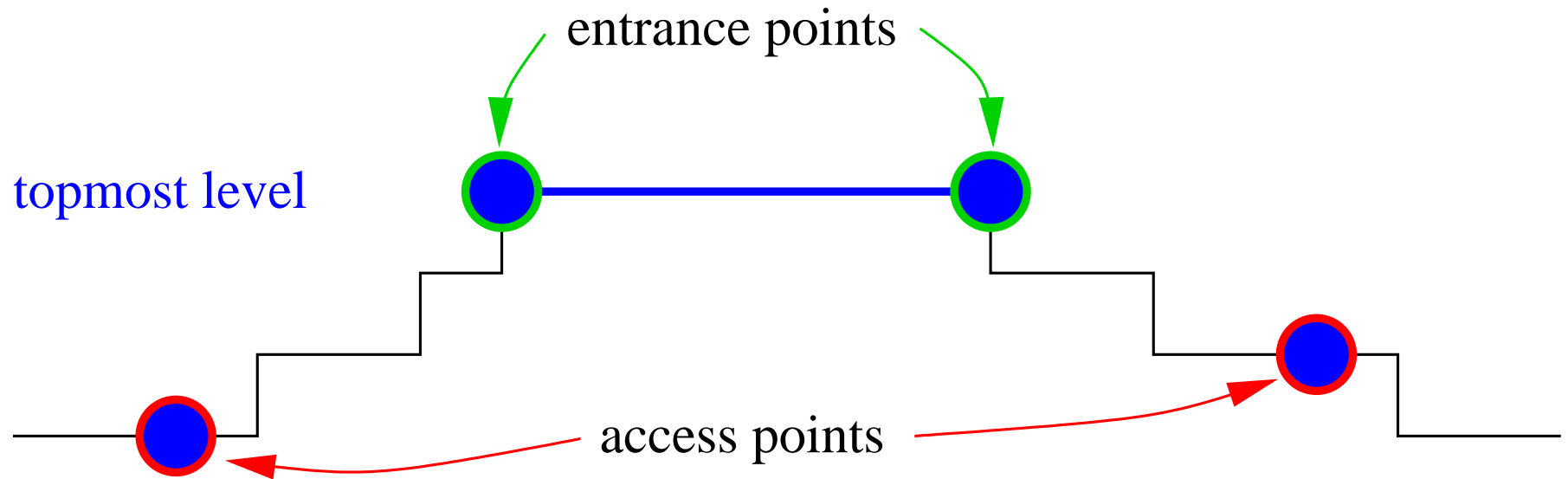


# Missing Pieces

2. **too many** 'entrance points' (55)

**solution:** fall back on comparatively few '**access points**' (10)

(motivated by the observations from [Bast, Funke, Matijevic 2006])







## Missing Pieces

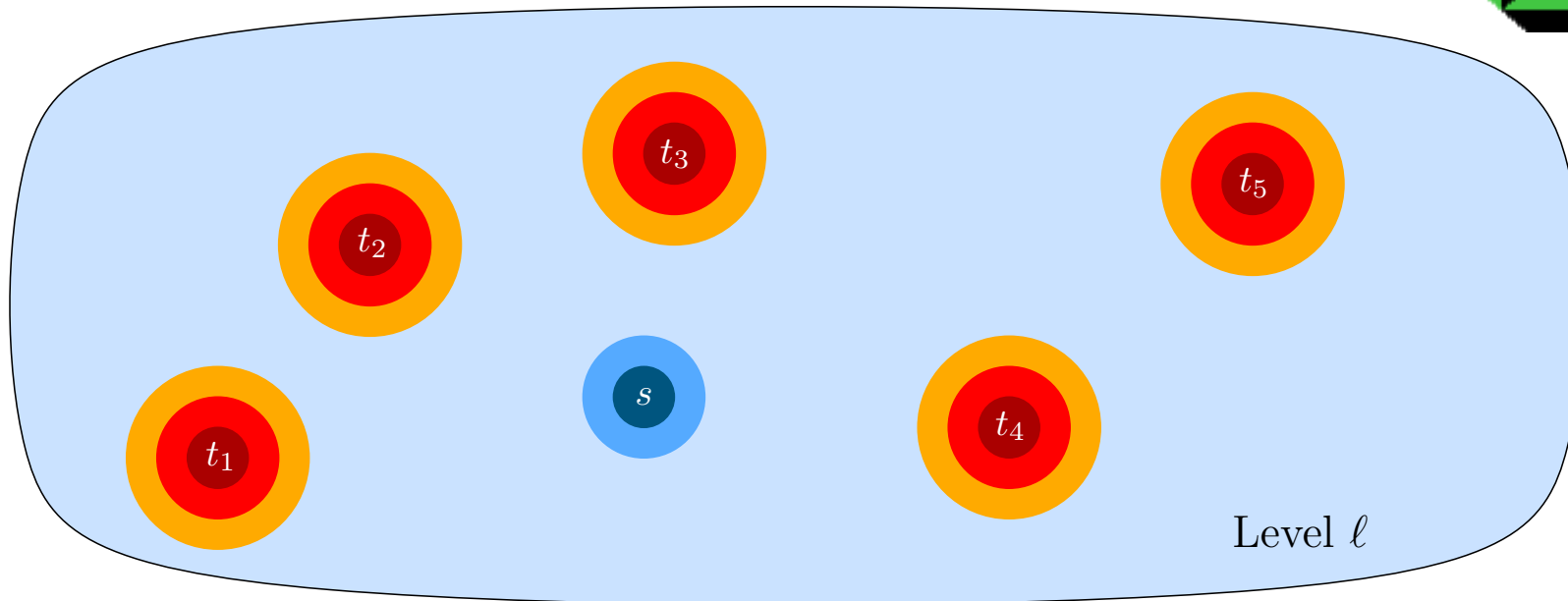
3. compute **top distance table** in the **original** graph

**solution:** [Knopp, S, S, Schulz, Wagner 2007]

“Computing Many-to-Many Shortest Paths  
Using Highway Hierarchies”

(e.g.  $10\,000 \times 10\,000$  table in **one minute**)





- for each  $t \in \mathcal{T}$ , perform **backward search** up to top level  $\ell$ ,  
store search space entries  $(t, u, d(u, t))$
- arrange search spaces: group entries by  $u$
- for each  $s \in \mathcal{T}$ , perform **forward search**,  
at each node  $u$ , **scan all entries**  $(t, u, d(u, t))$  and  
compute  $d(s, u) + d(u, t)$





## Second Layer

(to deal with **medium range queries**)

- secondary transit node set  $\mathcal{T}_2 \supset \mathcal{T}$
- secondary access mapping  $A_2 : V \rightarrow 2^{\mathcal{T}_2}$
- secondary dist. table  $\{d(u, v) : u, v \in \mathcal{T}_2 \wedge d(u, v) \neq d_{\text{top}}(u, v)\}$
- secondary locality filter  $L_2$

**$\neg L_2(s, t)$  implies**

$$d(s, t) = d_{\text{top}}(s, t)$$

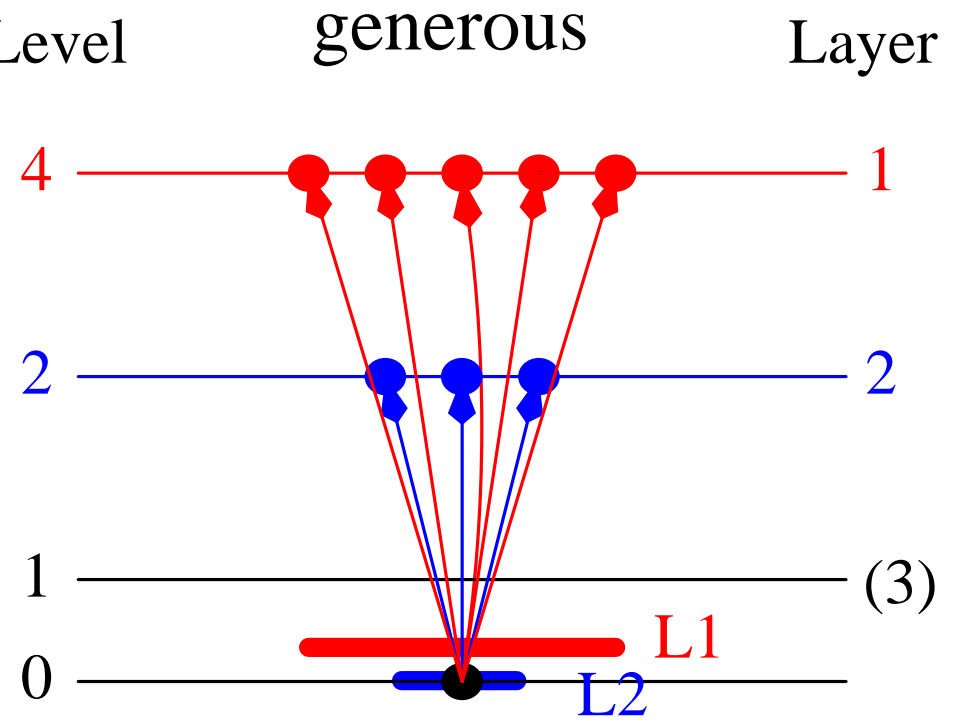
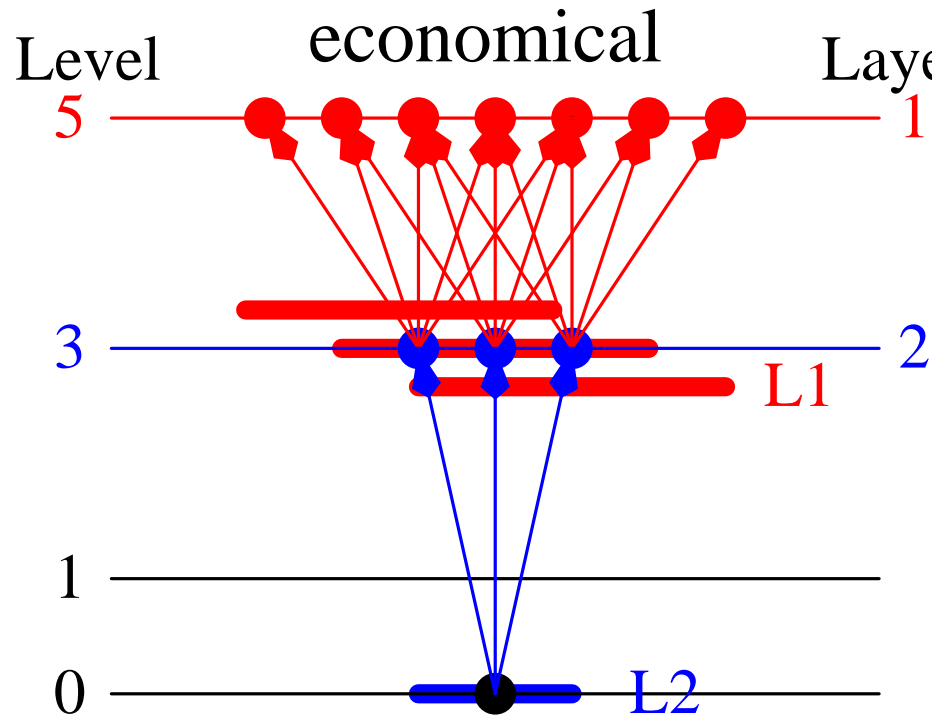
**OR**

$$d(s, t) = \min \{d(s, u) + d(u, v) + d(v, t) : u \in A_2(s), v \in A_2(t)\}$$





# Two Concrete Variants







# Preprocessing

- for each secondary transit node  $t \in \mathcal{T}_2$ :
  - perform **backward highway search**
  - **stop** at primary transit nodes ( $\rightsquigarrow$  set of **backward access points**)
  - **store** search space entries  $(t, u, d(u, t))$
  
- arrange search spaces



## Preprocessing

- for each secondary transit node  $s \in \mathcal{T}_2$ :
  - perform **forward highway search**
  - **stop** at primary transit nodes ( $\rightsquigarrow$  set of **forward access points**)
  - at each node  $u$  and for each search space entry  $(t, u, d(u, t))$ :
    - \* compute distance  $d_u(s, t) := d(s, u) + d(u, t)$  **via  $u$**
    - \* compute distance  $d_{\text{top}}(s, t)$  **via top layer**
    - \* if  $d_u(s, t) < d_{\text{top}}(s, t)$  then
      - **add entry  $d_u(s, t)$**  to the secondary distance table
      - ensure that the **disks** around  $s$  and  $t$  **contain  $u$**

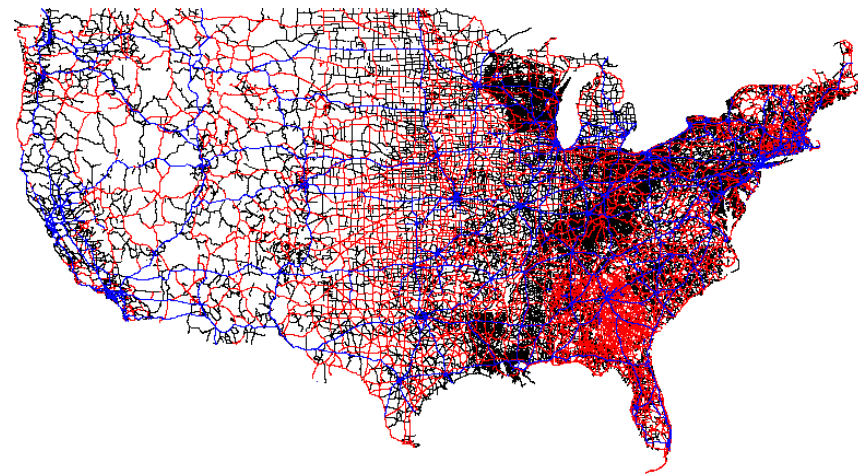
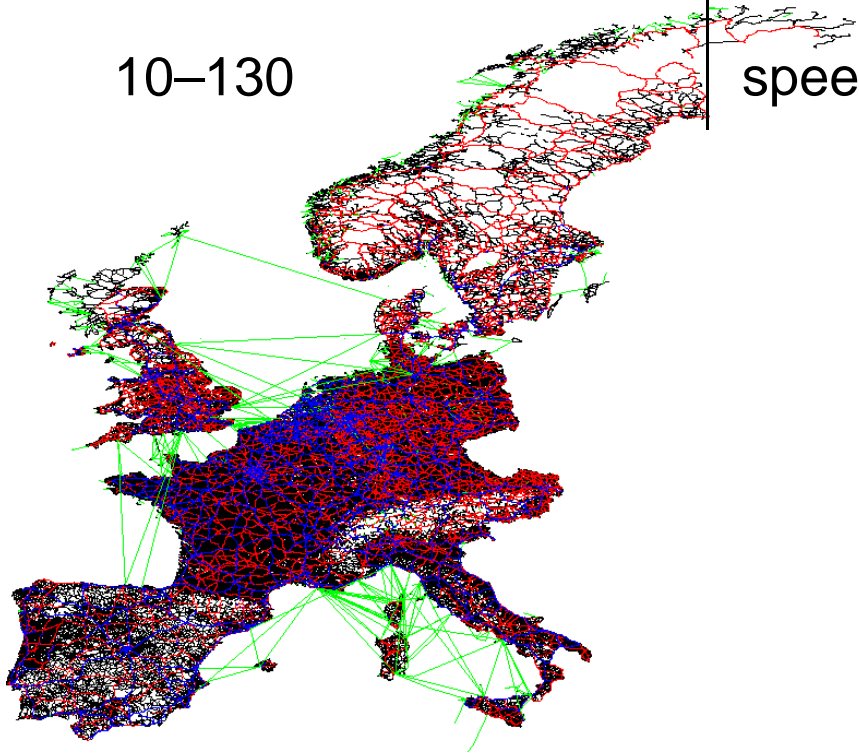
(use similar procedure for lower layers)





# Experiments

W. Europe (PTV)	Our Inputs	USA (TIGER/Line)
18 010 173	#nodes	23 947 347
42 560 279	#directed edges	58 333 344
13	#road categories	4
10–130	speed range [km/h]	40–100





# Preprocessing

metric	variant	layer 1		layer 2		space [B/node]	time [h]	
		$ \mathcal{T} $	$ A $	$ \mathcal{T}_2 $	$ A_2 $			
USA	time	eco	12 111	6.1	184 379	4.9	111	0:59
		gen	10 674	5.7	485 410	4.2	244	3:25
	dist	eco	15 399	17.0	102 352	10.9	171	8:58
EUR	time	eco	8 964	10.1	118 356	5.5	110	0:46
		gen	11 293	9.9	323 356	4.1	251	2:44
	dist	eco	11 610	20.3	69 775	13.1	193	7:05



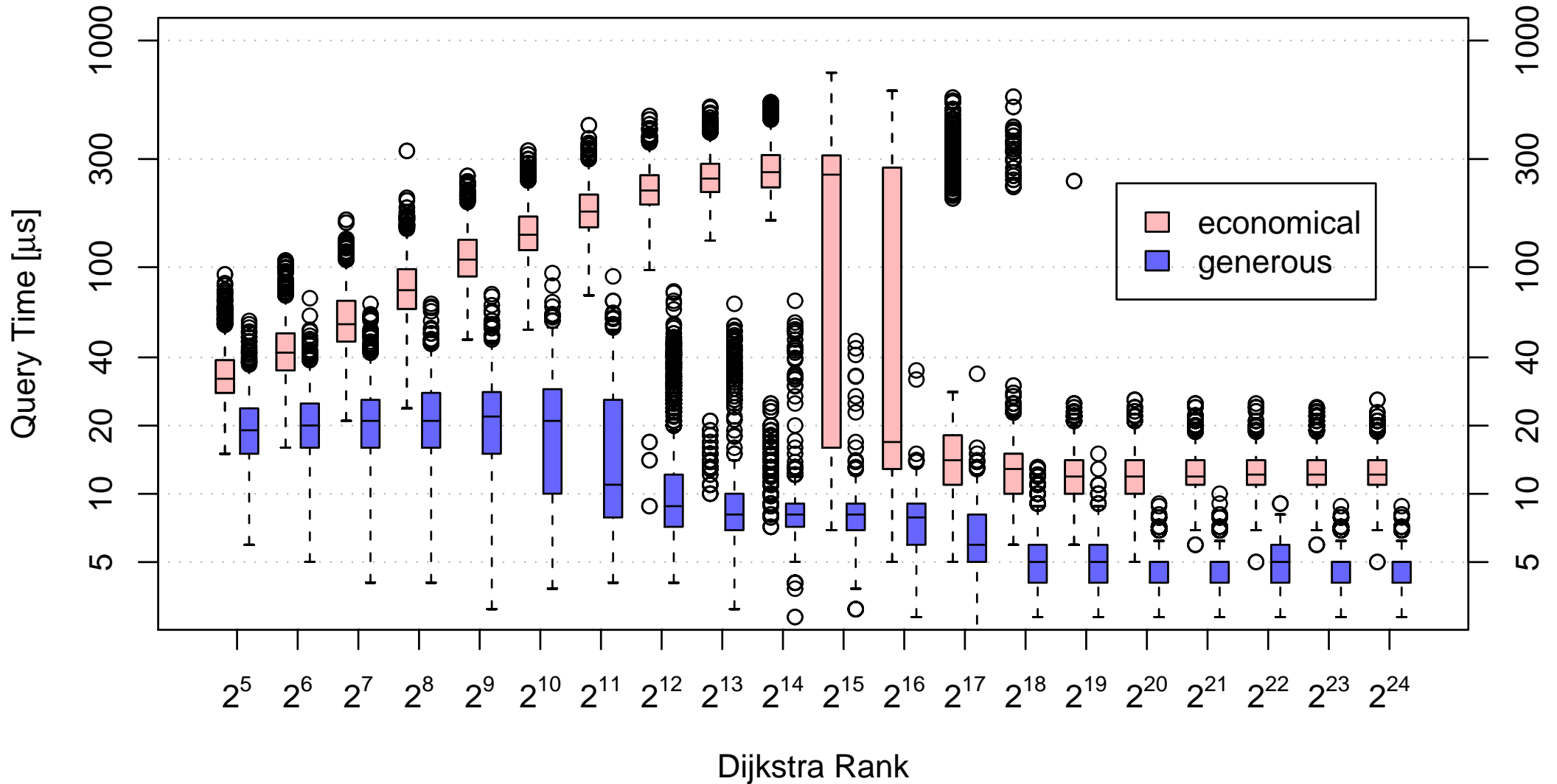


**Query**

		<b>layer 1 [%]</b>		<b>layer 2 [%]</b>			
<b>metric</b>	<b>variant</b>	wrong	cont'd	wrong	cont'd	<b>time</b>	
USA	time	eco	0.14	1.13	0.0064	0.2780	11.5 $\mu$ s
		gen	0.11	0.80	0.0014	0.0138	4.9 $\mu$ s
	dist	eco	1.57	8.10	0.0489	2.2352	87.5 $\mu$ s
EUR	time	eco	0.54	2.87	0.0092	0.5843	13.4 $\mu$ s
		gen	0.26	1.35	0.0015	0.0190	5.6 $\mu$ s
	dist	eco	4.68	18.32	0.1761	4.2764	107.4 $\mu$ s



# Local Queries: USA, travel time metric



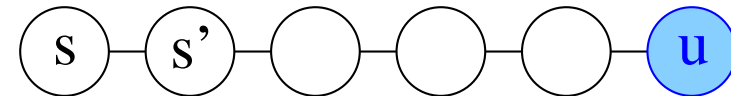




## Determine Shortest Paths

- from source  $s$ , look iteratively for the next adjacent node  $s'$  that leads to fwd acc pnt  $u$

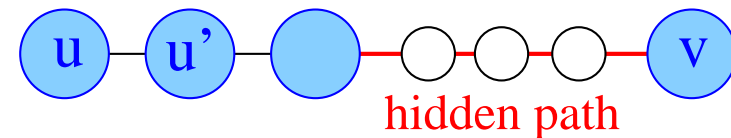
$$d(s, s') + d(s', u) = d(s, u)$$



- analogously, from target to bwd acc pnt

- from fwd acc pnt  $u$ , look iteratively for the next adjacent node  $u'$  in the topmost level that leads to bwd acc pnt  $v$

$$d(u, u') + d(u', v) = d(u, v)$$



(if necessary, use a hidden path instead)

- unpack shortcuts ( $\rightarrow$  yesterday's talk)
- if path not through top layer, fall back on highway query ( $\rightarrow$  yesterday's talk)



# Determine Shortest Paths

	preproc.	space	query	# hops
	[min]	[MB]	[ $\mu$ s]	(avg.)
USA	4:04	193	258	4 537
EUR	7:43	188	155	1 373



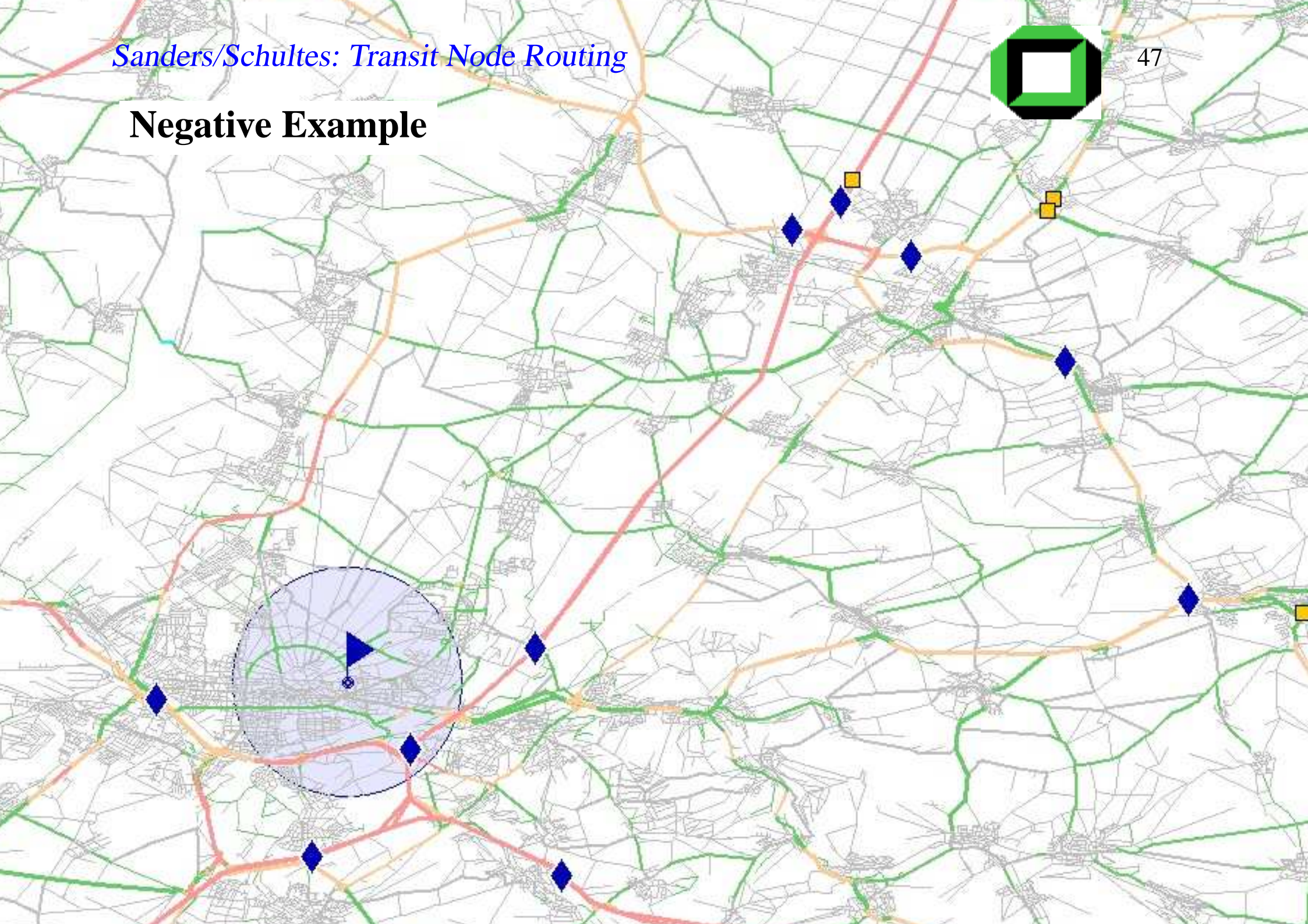


## Summary

- **insight: human intuition** is applicable in this case
- extremely **fast queries** (down to  $5 \mu s$ )
- handles **all kinds** of queries (local/global) ( $5-20 \mu s$ )
- **moderate preprocessing times** (down to  $46 \text{ min}$ )
- **output** shortest paths quickly (down to  $155 \mu s$ )



# Negative Example







## Future Work

- select better transit node set**  
(e.g. some combination with the separator-based approach)
- more **flexible** implementation (to better deal with the distance metric)
- more **space-efficient** implementation  
(e.g. store access points only at the core-1 nodes)
- use access points as landmarks to **guide local search**
- improve **locality filter** (get rid of geometry?)