Route Planning in Road Networks
– simple, flexible, efficient –

Peter Sanders Dominik Schultes

Institut für Theoretische Informatik – Algorithmik II
Universität Karlsruhe (TH)

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

Berlin, December 10, 2007
Route Planning

Task:
In a given road network, determine an optimal route from a given source to a given target.

Applications:
- route planning systems in the internet, car navigation systems,
- traffic simulation, logistics optimisation
**DIJKSTRA’s Algorithm**

**the classic solution [1959]**

\[ O(n \log n + m) \text{ (with Fibonacci heaps)} \]

Dijkstra

not practicable

for large graphs

(e.g. European road network:

\[ \approx 18\,000\,000 \text{ nodes} \])

bidirectional Dijkstra

improves the running time,

but still too slow
Speedup Techniques

that are faster than Dijkstra’s algorithm

- require additional data (e.g., node coordinates)
  not always available!

AND / OR

- preprocess the graph and generate auxiliary data (e.g., ‘signposts’)
  can take a lot of time; assume many queries;
  assume static graph or require update operations!

AND / OR

- exploit special properties of the network (e.g., planar, hierarchical)
  fail when the given graph has not the desired properties!

⇒ not a solution for general graphs,

but can be very efficient for many practically relevant cases
**Speedup Techniques**

- Require additional data (e.g., node coordinates)

AND / OR

- Preprocess the graph and generate auxiliary data (e.g., ‘signposts’)

AND / OR

- Exploit special properties of the network (e.g., planar, hierarchical)
Goals

- fast queries
- accurate results
- scale invariant / support all types of queries
- fast preprocessing / deal with large networks
- low space consumption
- fast update operations
- simple
Overview

Transit Node Routing
very fast queries
[DIMACS 06, ALENEX 07, Science 07]

HH Star
goal–directed
[DIMACS 06]

Highway Hierarchies
foundation
[ESA 05, ESA 06]

Many–to–Many
compute distance tables
[ALLENEX 07]

Hwy–Node Routing
allow edge weight changes
[WEA 07]
Highway Hierarchies

**Construction:** iteratively alternate between

- removal of low degree nodes
- removal of edges that only appear on shortest paths close to source or target

yields a hierarchy of highway networks in a sense, classify roads / junctions by ‘importance’
Highway Hierarchies

- Foundation for our other methods
- Directly allows point-to-point queries
- 13 min preprocessing
- 0.61 ms to determine the path length
- (0.80 ms to determine a complete path description)
- Reasonable space consumption (48 bytes/node) can be reduced to 17 bytes/node
Highway Hierarchies Star

joint work with D. Delling, D. Wagner

- combination of highway hierarchies with goal-directed search
- slightly reduced query times (0.49 ms)
- more effective
  - for approximate queries or
  - when a distance metric instead of a travel time metric is used
Many-to-Many Shortest Paths

joint work with S. Knopp, F. Schulz, D. Wagner

[ALENEX 07]

Given:

- graph $G = (V, E)$
- set of source nodes $S \subseteq V$
- set of target nodes $T \subseteq V$

Task: compute $|S| \times |T|$ distance table containing the shortest path distances

- e.g., 10 000 $\times$ 10 000 table in 23 seconds
Transit-Node Routing

joint work with H. Bast, S. Funke, D. Matijevic

- very fast queries
  (down to $4 \mu s$, > 1 000 000 times faster than Dijkstra)
- winner of the 9th DIMACS Implementation Challenge
- more preprocessing time (1:15 h) and space (247 bytes/node) needed
Transit Node Routing

Brussels, London

Munich, Rome, Paris

Copenhagen, Berlin, Vienna
Transit-Node Routing

First Observation:
For long-distance travel: leave current location via one of only a few ‘important’ traffic junctions, called access points [in Europe ≈ 10]
(⇒ we can afford to store all access points for each node)

Second Observation:
Each access point is relevant for several nodes. ⇒
union of the access points of all nodes is small, called transit node set [in Europe ≈ 10 000]
(⇒ we can afford to store the distances between all transit node pairs)
Transit-Node Routing

Query: usually only a few table lookups
Highway-Node Routing

1. **basic concepts**: overlay graphs, covering nodes

2. lightweight, efficient **static approach**

3. **dynamic** version
1. Basic Concepts
Overlay Graph: Definition


- graph $G = (V, E)$ is given
- select node subset $S \subseteq V$
Overlay Graph: Definition


- graph $G = (V, E)$ is given
- select node subset $S \subseteq V$

- overlay graph $G' := (S, E')$
  
  determine edge set $E'$ s.t. shortest path distances are preserved
Minimal Overlay Graph


- Graph $G = (V, E)$ is given
- Select node subset $S \subseteq V$

- Minimal overlay graph $G' := (S, E')$ where

$$E' := \{(s, t) \in S \times S \mid \text{no inner node of the shortest } s-t\text{-path belongs to } S\}$$
Covering Nodes

Definitions:

- **covered branch**: contains a node from \( S \)
- **covered tree**: all branches covered
- **covering nodes**: on each branch, the node \( u \in S \) closest to the root \( s \)
Query: Intuition

- bidirectional

- perform search in $G$ till search trees are covered by nodes in $S$
Query: Intuition

- bidirectional
- perform search in $G$ till search trees are covered by nodes in $S$
- continue search only in $G'$
for each node $u \in S$

- perform a local search from $u$ in $G$
- determine the covering nodes
- add an edge $(u, v)$ to $E'$ for each covering node $v$
Covering Nodes

Conservative Approach:

- stop searching in $G$ when all branches are covered

- can be very inefficient
Covering Nodes

Aggressive Approach:

- do not continue the search in $G$ on covered branches

- can be very inefficient
Covering Nodes

Compromise:

- Introduce parameter $p$

- Do not continue the search in $G$ on branches that already contain $p$ nodes from $S$

- In addition: stop when all branches are covered

- $p = 1 \rightarrow$ aggressive

- $p = \infty \rightarrow$ conservative

- Works very well in practice
2. Static Highway-Node Routing
Static Highway-Node Routing

- extend ideas from
  - multi-level overlay graphs [HolzerSchulzWagnerWeiheZaroliagis00–07]
  - highway hierarchies [SS05–06]
  - transit node routing [BastFunkeMatijevicSS06–07]

- use highway hierarchies to classify nodes by ‘importance’
  i.e., select node sets $S_1 \supseteq S_2 \supseteq S_3 \ldots \supseteq S_L$
  (crucial distinction from previous separator-based approach)

- construct multi-level overlay graph

\[
G_0 = G = (V, E), \quad G_1 = (S_1, E_1), \quad G_2 = (S_2, E_2), \ldots, \quad G_L = (S_L, E_L)
\]

(just iteratively construct overlay graphs)
Static Highway-Node Routing

- extend ideas from
  - multi-level overlay graphs [HolzerSchulzWagnerWeiheZaroliagis00–07]
  - highway hierarchies [SS05–06]
  - transit node routing [BastFunkeMatijevicSS06–07]

- use highway hierarchies to classify nodes by ‘importance’
  i.e., select node sets $S_1 \supseteq S_2 \supseteq S_3 \ldots \supseteq S_L$
  (crucial distinction from previous separator-based approach)

- construct multi-level overlay graph
  $G_0 = G = (V, E), G_1 = (S_1, E_1), G_2 = (S_2, E_2), \ldots, G_L = (S_L, E_L)$
  (just iteratively construct overlay graphs)

(experiments with a European road network with $\approx 18$ million nodes)
Query: Aggressive Variant

- node level $\ell(u) := \max \{\ell \mid u \in S_\ell\}$
- forward search graph $\vec{G} := (V, \{(u, v) \mid (u, v) \in \bigcup_{i=\ell(u)}^L E_i\})$
- backward search graph $\overleftarrow{G} := (V, \{(u, v) \mid (v, u) \in \bigcup_{i=\ell(u)}^L E_i\})$
- perform one plain Dijkstra search in $\vec{G}$ and one in $\overleftarrow{G}$
Proof of Correctness

Level 0

Level 1

Level 2

shortest path from $s$ to $t$ in $G = G_0$
Proof of Correctness

overlay graph $G_1$ preserves distance from $s_1 \in S_1$ to $t_1 \in S_1$
Proof of Correctness

Overlay graph $G_2$ preserves distance from $s_2 \in S_2$ to $t_2 \in S_2$
Proof of Correctness

Let \( \vec{G} := (V, \{(u, v) \mid (u, v) \in \bigcup_{i=\ell(u)}^{L} E_i\}) \) and \( \vec{G} := (V, \{(u, v) \mid (v, u) \in \bigcup_{i=\ell(u)}^{L} E_i\}) \).

Level 0

Level 1

Level 2
Stall-on-Demand

- A node $v$ can 'wake' an already settled node $u$

- $u$ can 'stall' $v$ (if $\delta(u) + w(u, v) < \delta(v)$)
  i.e., search is not continued from $v$

- Stalling can propagate to adjacent nodes

- Does not invalidate correctness (only suboptimal paths are stalled)
Karlsruhe → Bertinoro
NO Stall-on-Demand
search space size: 31,756
Karlsruhe → Bertinoro
Stall-on-Demand
search space size: 1179
const NodeID index = isReached(searchID, v);
if (edge.isDirected(1-dir) && index) {
    const PQData& data = pqData(searchID, index);
    EdgeWeight vKey = data.stalled() ? data.stallKey() : pqKey(searchID, index);
    if (vKey + edge.weight() < parentDist) {
        pqData(searchID, parent.index).stallKey(vKey + edge.weight());
        queue<pair<NodeID, EdgeWeight>> _stallQueue;
        _stallQueue.push(pair<NodeID, EdgeWeight>(parent.nodeID, vKey + edge.weight()));
        while (!_stallQueue.empty()) {
            u = _stallQueue.front().first;
            key = _stallQueue.front().second;
            _stallQueue.pop();
            for (EdgeID e = _graph->firstEdge(u); e < _graph->lastEdge(u); e++) {
                const Edge& edge = _graph->edge(e);
                if (! edge.isDirected(searchID)) continue;
                NodeID index = isReached(searchID, edge.target());
                if (index) {
                    const EdgeWeight newKey = key + edge.weight();
                    if (newKey < pqKey(searchID, index)) {
                        PQData& data = pqData(searchID, index);
                        if (! data.stalled()) {
                            data.stallKey(newKey);
                            _stallQueue.push(pair<NodeID, EdgeWeight>(edge.target(), newKey));
                        }
                    }
                }
            }
        }
    }
    return;
}
Example: Berlin → Karlsruhe
Example: Berlin → Karlsruhe
Example: Berlin → Karlsruhe
Example: Berlin → Karlsruhe
Example: Berlin → Karlsruhe
Example: Berlin → Karlsruhe
Local Queries

Dijkstra Rank

Query Time [ms]
Per-Instance Worst-Case Guarantee

max = 2,148 nodes
different trade-offs between memory consumption and query time

for example:

- 9.5 bytes per node overhead $\rightarrow$ 0.89 ms
  store complete multi-level overlay graph

- 0.7 bytes per node overhead $\rightarrow$ 1.44 ms
  store only forward and backward search graph $\vec{G}$ and $\vec{G}$
  ($\vec{G}$ and $\vec{G}$ are independent of $s$ and $t$)

numbers refer to the Western European road network with 18 million nodes
3. Dynamic Highway-Node Routing
Dynamic Scenarios

- change entire **cost function**
  (e.g., use different speed profile)

- change a **few edge weights**
  (e.g., due to a traffic jam)
### Constancy of Structure

**Assumption:**

- **structure of road network does not change**
  
  (no new roads, road removal = set weight to \(\infty\))

  \(\Rightarrow\) **not a significant restriction**

- **classification of nodes by ‘importance’** might be slightly perturbed, but not completely changed

  (e.g., a sports car and a truck both prefer motorways)

  \(\Rightarrow\) **performance** of our approach relies on that

  (not the correctness)
change entire cost function

keep the node sets $S_1 \supseteq S_2 \supseteq S_3 \ldots$

recompute the overlay graphs

<table>
<thead>
<tr>
<th>speed profile</th>
<th>default</th>
<th>fast car</th>
<th>slow car</th>
<th>slow truck</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>constr. [min]</td>
<td>1:40</td>
<td>1:41</td>
<td>1:39</td>
<td>1:36</td>
<td>3:56</td>
</tr>
<tr>
<td>query [ms]</td>
<td>1.17</td>
<td>1.20</td>
<td>1.28</td>
<td>1.50</td>
<td>35.62</td>
</tr>
<tr>
<td>#settled nodes</td>
<td>1 414</td>
<td>1 444</td>
<td>1 507</td>
<td>1 667</td>
<td>7 057</td>
</tr>
</tbody>
</table>
Dynamic Highway-Node Routing

change a few edge weights

- server scenario: if something changes,
  - update the preprocessed data structures
  - answer many subsequent queries very fast

- mobile scenario: if something changes,
  - it does not pay to update the data structures
  - perform single ‘prudent’ query that takes changed situation into account
Dynamic Highway-Node Routing

change a few edge weights, server scenario

- keep the node sets \( S_1 \supseteq S_2 \supseteq S_3 \ldots \)
- recompute only possibly affected parts of the overlay graphs
  - the computation of the level-\( \ell \) overlay graph consists of \(|S_\ell|\) local searches to determine the respective covering nodes
  - if the initial local search from \( v \in S_\ell \) has not touched a now modified edge \((u, x)\), that local search need not be repeated
  - we manage sets \( A^\ell_u = \{ v \in S_\ell \mid v's \level-\ell \text{ preprocessing might be affected when an edge } (u, x) \text{ changes} \} \)
Dynamic Highway-Node Routing

change a few edge weights, server scenario

<table>
<thead>
<tr>
<th>Road Type</th>
<th>Update Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>0.1 1 10 100</td>
</tr>
<tr>
<td>motorway</td>
<td>0.1 1 10 100</td>
</tr>
<tr>
<td>national</td>
<td>0.1 1 10 100</td>
</tr>
<tr>
<td>regional</td>
<td>0.1 1 10 100</td>
</tr>
<tr>
<td>urban</td>
<td>0.1 1 10 100</td>
</tr>
</tbody>
</table>

- add traffic jam
- cancel traffic jam
- block road
Dynamic Highway-Node Routing

change a **few edge weights**, mobile scenario

1. keep the node sets \( S_1 \supseteq S_2 \supseteq S_3 \ldots \)

2. keep the overlay graphs

3. \( C := \) all changed edges

4. use the sets \( A^\ell_u \) (considering edges in \( C \)) to determine for each node \( v \) a **reliable level** \( r(v) \)

5. during a query, at node \( v \)
   - do not use edges that have been created in some level \( > r(v) \)
   - instead, **downgrade** the search to level \( r(v) \) (forward search only)
Dynamic Highway-Node Routing

change a few edge weights, mobile scenario

reliable levels: $r(x) = 0$, $r(s_2) = r(t_2) = 1$
Dynamic Highway-Node Routing

change a few edge weights, mobile scenario

iterative variant (provided that only edge weight increases allowed)

1. keep everything (as before)

2. $C := \emptyset$

3. use the sets $A^\ell_u$ (considering edges in $C$) to determine for each node $v$ a reliable level $r(v)$ (as before)

4. ‘prudent’ query (as before)

5. if shortest path $P$ does not contain a changed edge, we are done

6. otherwise: add changed edges on $P$ to $C$, repeat from 3.
Dynamic Highway-Node Routing

change a few edge weights, mobile scenario

<table>
<thead>
<tr>
<th>change set (motorway edges)</th>
<th>affected queries</th>
<th>single pass query time [ms]</th>
<th>iterative query time [ms]</th>
<th>#iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4 %</td>
<td>2.3</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>5.8 %</td>
<td>8.5</td>
<td>1.7</td>
<td>1.1</td>
</tr>
<tr>
<td>100</td>
<td>40.0 %</td>
<td>47.1</td>
<td>3.6</td>
<td>1.4</td>
</tr>
<tr>
<td>1000</td>
<td>83.7 %</td>
<td>246.3</td>
<td>25.3</td>
<td>2.7</td>
</tr>
</tbody>
</table>
Unidirectional Queries

1. keep everything (as before)

2. \( C := \{\text{some edge } (t, x) \} \)

3. use the sets \( A_{\ell}^u \) (considering edges in \( C \)) to determine for each node \( v \) a reliable level \( r(v) \) (as before)

4. ‘prudent’ query (as before)
Unidirectional Queries

$\in \vec{G}$

Level 2

Level 1

Level 0

reliable levels: $r(t_1) = 0$, $r(t_2) = 1$
Summary

☐ efficient static approach

- fast preprocessing / fast queries  
  15 min / 0.9 ms
- outstandingly low memory requirements  
  0.7 bytes/node $\Rightarrow$ 1.4 ms

☐ can handle practically relevant dynamic scenarios

- change entire cost function  
  typically $< 2$ minutes
- change a few edge weights
  * update data structures  
    2–40 ms per changed edge
  OR
  * iteratively bypass traffic jams  
    e.g., 3.6 ms in case of 100 traffic jams

_____________________________

numbers refer to the Western European road network with 18 million nodes and  
to our 2.0 GHz AMD Opteron machine
find simpler / better ways to determine the node sets

\[ S_1 \supseteq S_2 \supseteq S_3 \ldots \]

parallelise the preprocessing

implementation for a mobile device
Future Work

☐ handle a massive amount of updates

☐ deal with time-dependent scenarios
  (where edge weights depend on the time of day)

☐ allow multi-criteria optimisations