TRANSIT
Ultrafast Shortest-Path Queries with Linear-Time Preprocessing

Ferienakademie im Sarntal — Course 2
Distance Problems: Theory and Praxis

Andreas Heider

Fakultät für Informatik
TU München

26. September 2010

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Universität München
Universität Stuttgart

Outline

Overview

Goal

- Faster Shortest-Path Queries
- Application: Navigation Systems

Overview

Goal

- Faster Shortest-Path Queries
- Application: Navigation Systems

Example

- US Road Network: 24 million nodes, 58 million edges
- Traditional Dijkstra too slow: worst case $O(m + nlogn)$
- Query time:
  - Dijkstra: seconds
  - Best other algorithms: milliseconds

Overview

Goal

- Faster Shortest-Path Queries
- Application: Navigation Systems

Example

- US Road Network: 24 million nodes, 58 million edges
- Traditional Dijkstra too slow: worst case $O(m + nlogn)$
- Query time:
  - Dijkstra: seconds
  - Best other algorithms: milliseconds
- Do we really need even faster algorithms?
- Yes: Web services, Traffic simulation, etc.

Overview

Solution

- Split the work into a preprocessing step and fast queries
- Considerations: Query time, preprocessing time, space usage, etc.

Overview

Solution

- Split the work into a preprocessing step and fast queries
- Considerations: Query time, preprocessing time, space usage, etc.

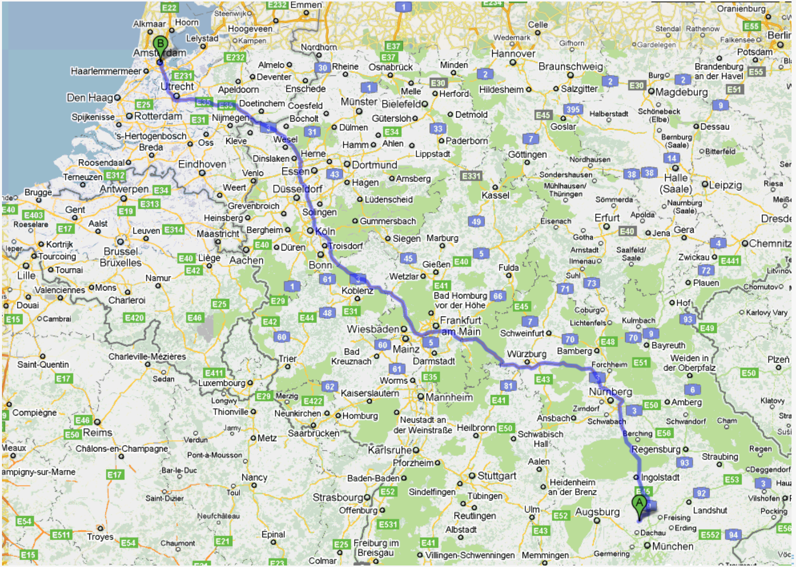
Special properties of road networks

- Optimize for the special structure of the problem
- Nodes have a small degree (US road network: 2.4)
- There is a hierachy of more and more important roads
- The graph is relatively static
- Much more...

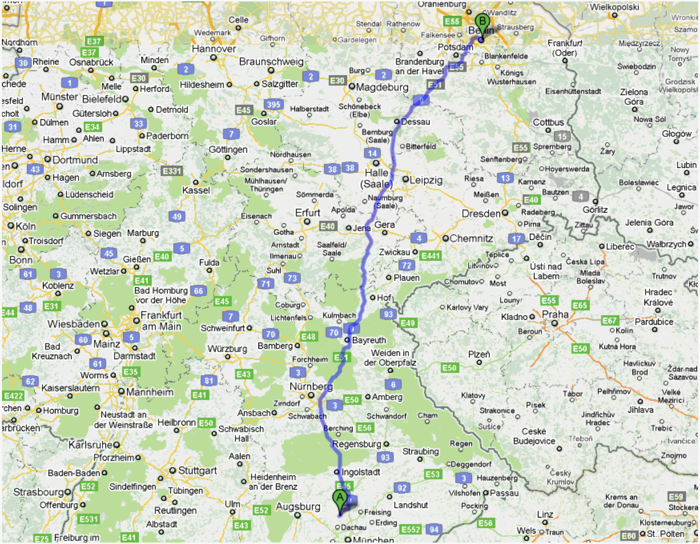
The key observation

- When travelling far there are only a few points you will leave your neighborhood through
- Those will be called *Transit Nodes*

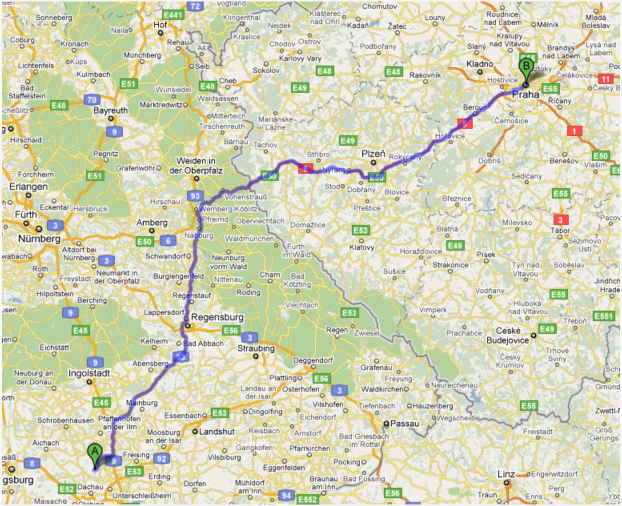
Vierkirchen - Amsterdam



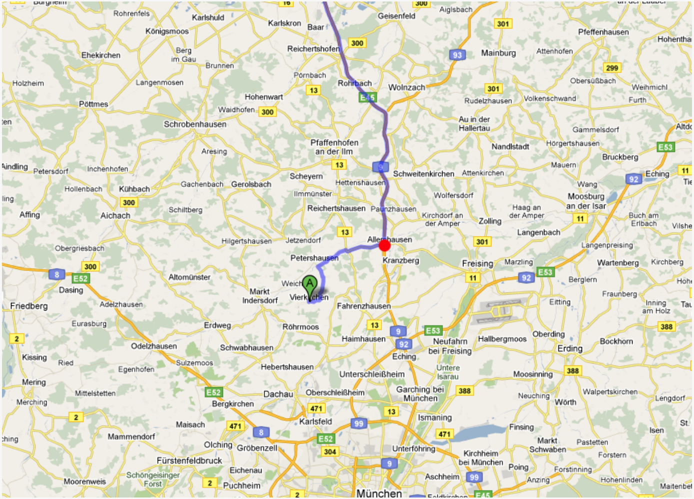
Vierkirchen - Berlin



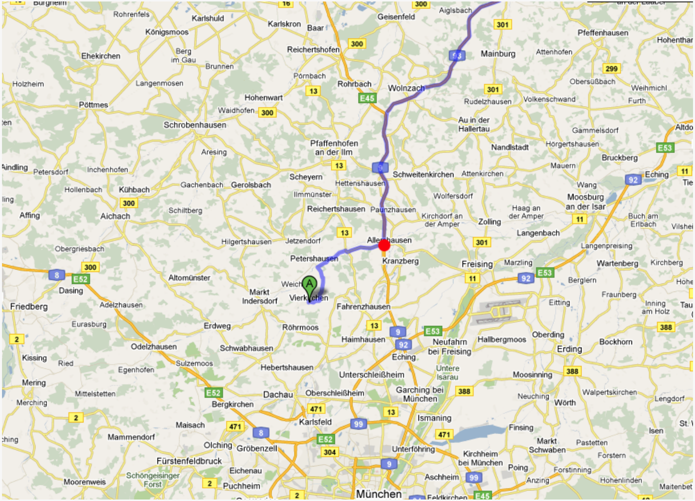
Vierkirchen - Prague



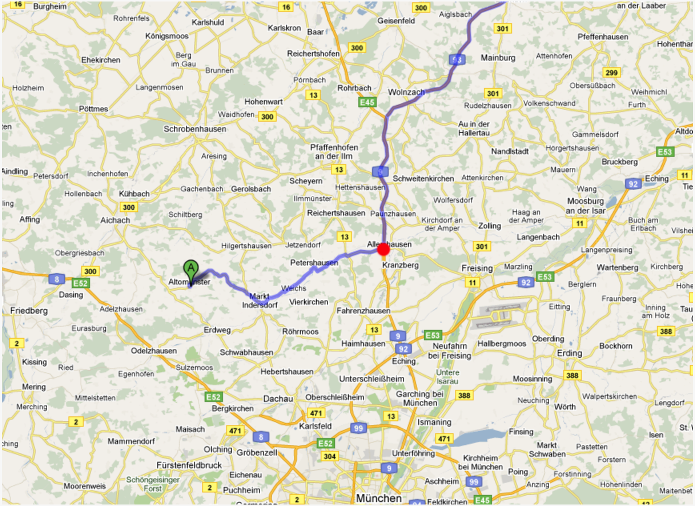
Vierkirchen - Amsterdam/Berlin



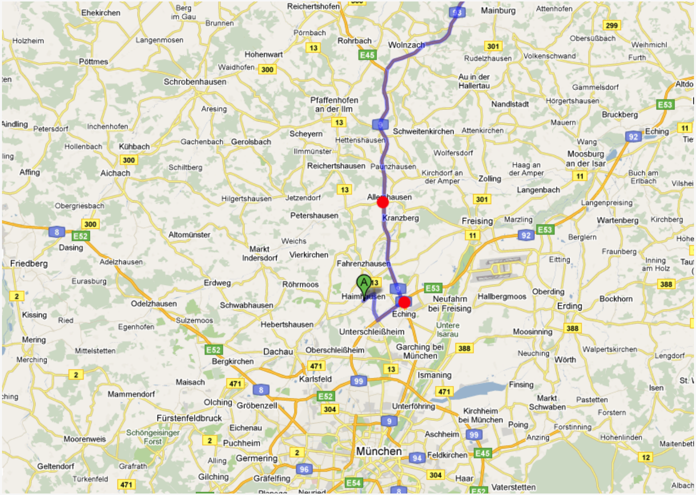
Vierkirchen - Prague



Altomünster - Prague



Haimhausen - Prague



The key observation

- When travelling far there are only a few points you will leave your neighborhood through
- Those will be called *Transit Nodes*

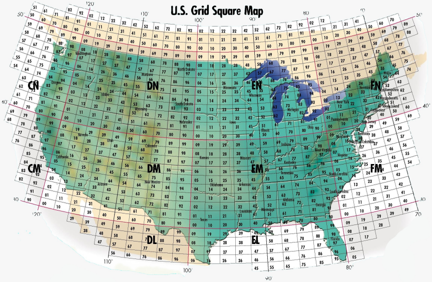
Algorithm outline

- Precomputation step:
  - For each neighborhood: find a set of Transit Nodes
  - Calculate distance from each node to its neighborhoods Transit Nodes
  - Run APSP (distances) between all Transit Nodes
- Shortest distance query: Find $t1$, $t2$ so that $dist(src, t1) + dist(t1, t2) + dist(t2, trg)$ is minimal

Formalization

How to implement ‘far’

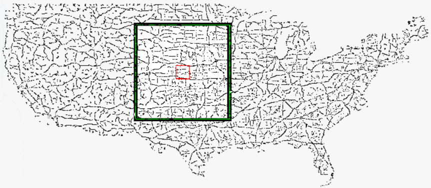
- Some metric is needed to determine wether a trip is far enough
- One possibility: Subdivide the map into a grid of cells



Formalization

How to implement ‘far’

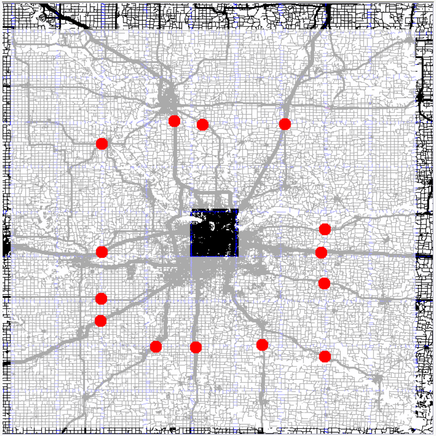
- Some metric is needed to determine wether a trip is far enough
- One possibility: Subdivide the map into a grid of cells
- A trip is long enough if the start and destination points are more than 4 cells apart
- To determine: best grid size



Formalization

How to implement ‘far’

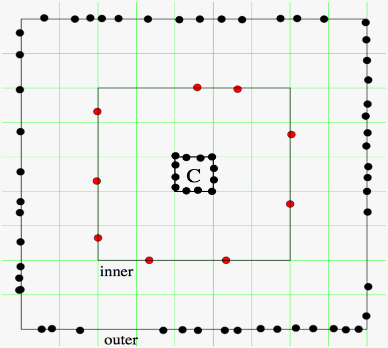
- Some metric is needed to determine wether a trip is far enough
- One possibility: Subdivide the map into a grid of cells
- A trip is long enough if the start and destination points are more than 4 cells apart
- To determine: best grid size



Formalization

Definitions

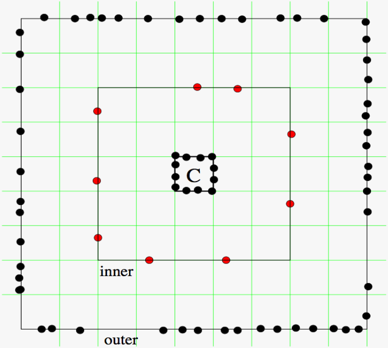
- C: The cell for which we want to compute the Transit Nodes



Formalization

Definitions

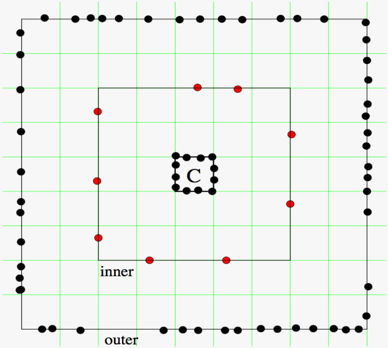
- C: The cell for which we want to compute the Transit Nodes
- S_{outer}: Square with C at it's center, everything outside is 'far away'



Formalization

Definitions

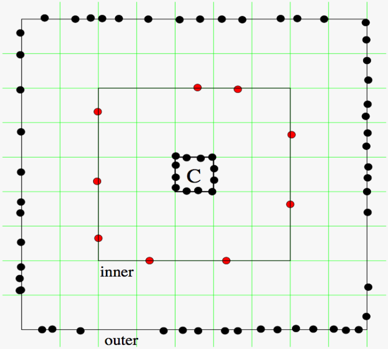
- C: The cell for which we want to compute the Transit Nodes
- S_{outer}: Square with C at it's center, everything outside is 'far away'
- S_{inner}: Between C and S_{outer}, all Transit Nodes cross S_{inner}



Formalization

Definitions

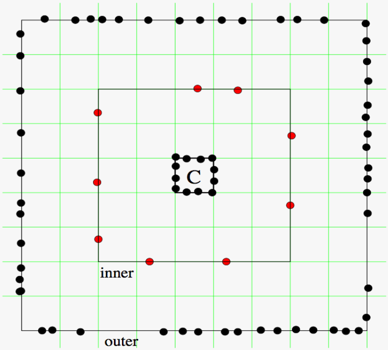
- $E_{C/inner/outer}$: Edges that cross a square



Formalization

Definitions

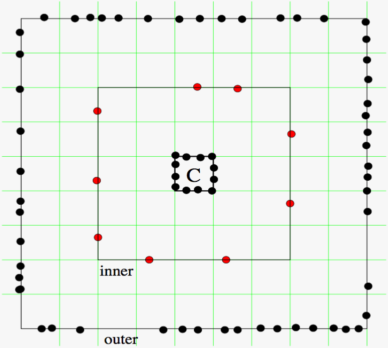
- $E_{C/inner/outer}$: Edges that cross a square
- $V_{C/inner/outer}$: For each edge in E: pick the node with the lower id



Formalization

Definitions

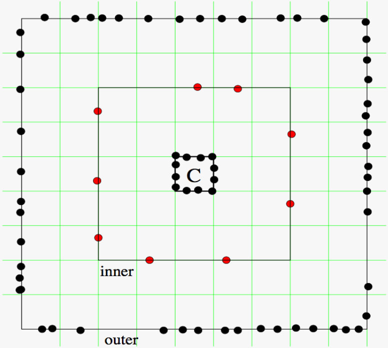
- $E_{C/inner/outer}$: Edges that cross a square
- $V_{C/inner/outer}$: For each edge in E: pick the node with the lower id
- All far trips starting inside C always first pass a node in V_C, then V_{inner}, then V_{outer}



Naive approach

Computing the Transit Nodes

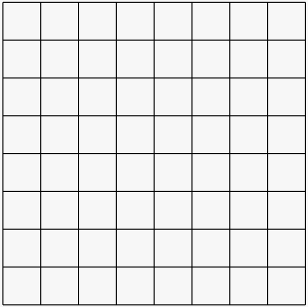
- For each cell: Compute all shortest paths between V_C and V_{outer}
- Mark all nodes in V_{inner} that lie on such a path, these are the Transit Nodes
- All paths starting inside V_C and ending outside V_{outer} will pass one of the Transit Nodes
- This requires a shortest paths run with a radius of 5 cells



Sweep-line algorithm

Sweep-line algorithm

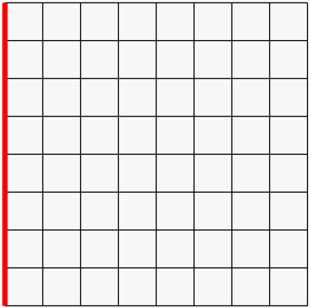
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

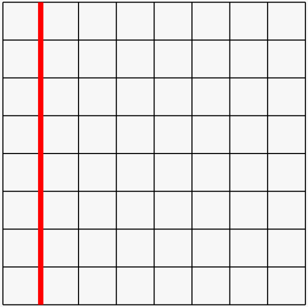
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

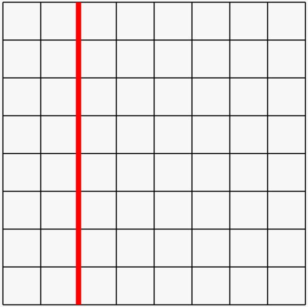
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

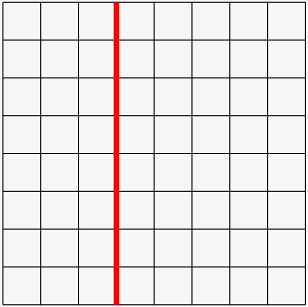
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

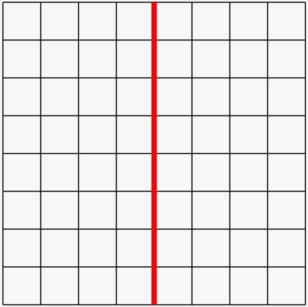
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

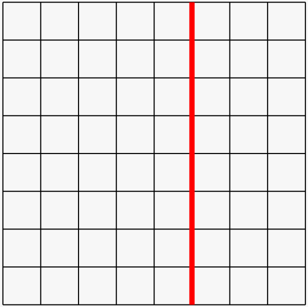
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

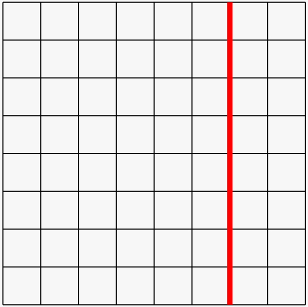
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

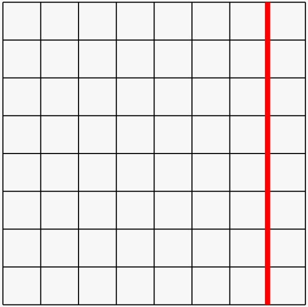
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

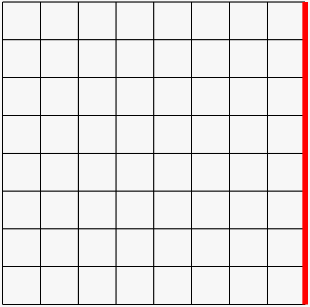
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

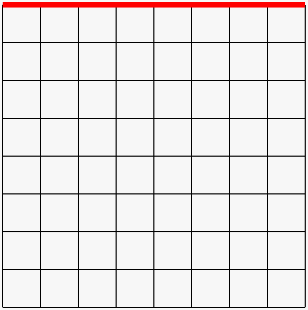
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

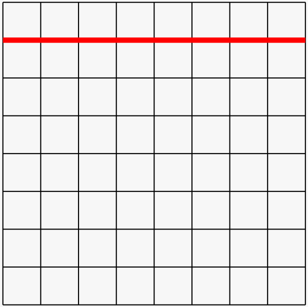
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

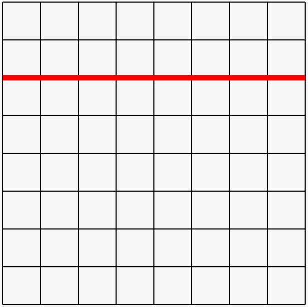
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

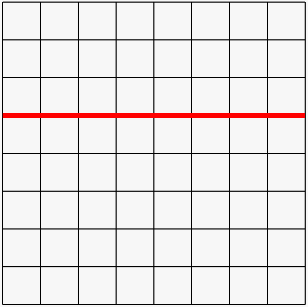
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

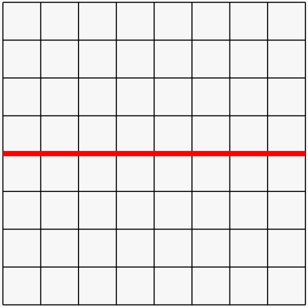
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

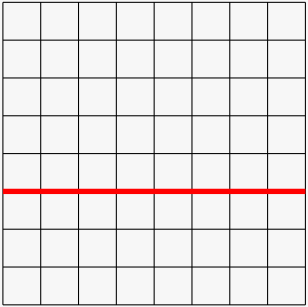
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

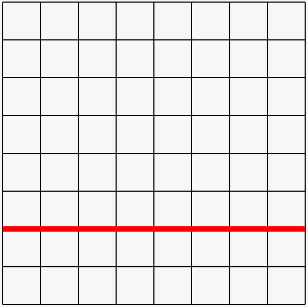
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

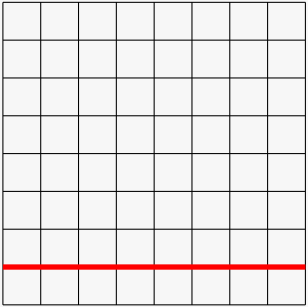
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

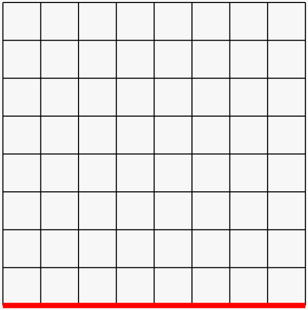
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Sweep-line algorithm

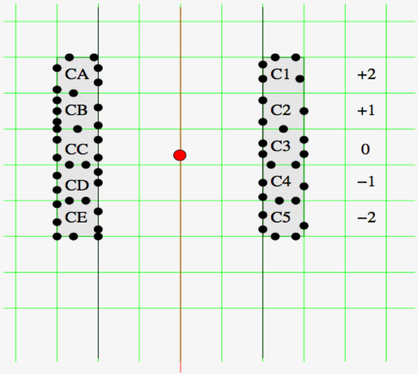
- A line is moved across the whole grid
- All roads that cross the line get processed
- When the line reaches the other end, the solution is available



Sweep-line algorithm

Computing the Transit Nodes

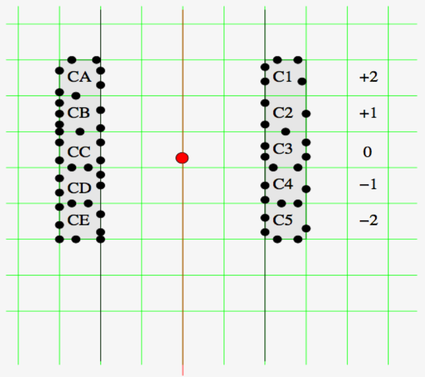
- For all roads intersecting the sweep line:
    - Choose one endpoint v
    - C_{left}, C_{right}: Cells two grid units left/right
    - Find all boundary nodes v_L, v_R on C_{left}, C_{right}
    - Run Dijkstra starting at v until we know the distance $d(v, v_{L/R})$ for all boundary nodes
    - To do this we mostly need to look at nodes no more than 3 cells away



Sweep-line algorithm

Computing the Transit Nodes

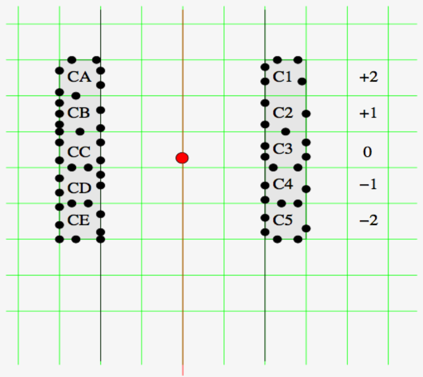
- We now know all $d(v, v_{L/R})$
- Look at all combinations of boundary nodes in (v_L, v_R) with a vertical distance of $<= 4$
- And determine v so that $d(v_L, v) + d(v, v_R)$ is minimal
- This v is a Transit Node for the cells containing v_L and v_R



Sweep-line algorithm

Computing the Transit Nodes

- We now know all $d(v, v_{L/R})$
- Look at all combinations of boundary nodes in (v_L, v_R) with a vertical distance of $<= 4$
- And determine v so that $d(v_L, v) + d(v, v_R)$ is minimal
- This v is a Transit Node for the cells containing v_L and v_R
- After one horizontal and one vertical sweep we computed exactly the Transit Nodes as defined before



Computing the Distance Tables

- For each node inside C: store the distance to all of Cs Transit Nodes

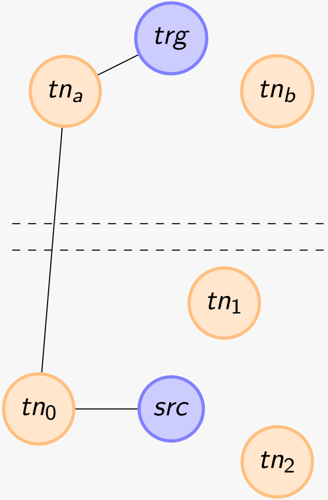
Computing the Distance Tables

- For each node inside C: store the distance to all of Cs Transit Nodes

- For each Transit Node: compute and the distance to all other Transit Nodes
- This is possible because only a few vertices are Transit Nodes
- Most cells only have about 10 Transit Nodes
- Transit Nodes are often shared between adjacent cells
- Ballpark figure: US road network using a 128x128 grid: 8000 Transit Nodes

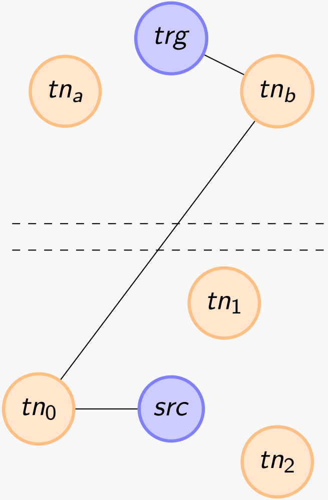
Shortest-distance queries

- Transit Nodes also work in reverse: Every 'far' trip entering a cell will do it through one of the Transit Nodes
- All 'far' trips can be split up into three parts: $src - transit_{src} - transit_{dest} - dest$
- Try all possible combinations of transit nodes to find the minimum of $d(src, transit_{src}) + d(transit_{src}, transit dest) + d(transit_{dest}, dest)$



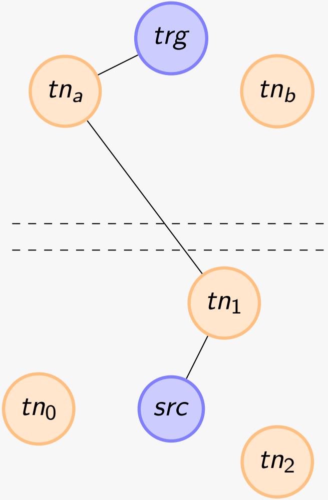
Shortest-distance queries

- Transit Nodes also work in reverse: Every 'far' trip entering a cell will do it through one of the Transit Nodes
- All 'far' trips can be split up into three parts: $src - transit_{src} - transit_{dest} - dest$
- Try all possible combinations of transit nodes to find the minimum of $d(src, transit_{src}) + d(transit_{src}, transit dest) + d(transit_{dest}, dest)$



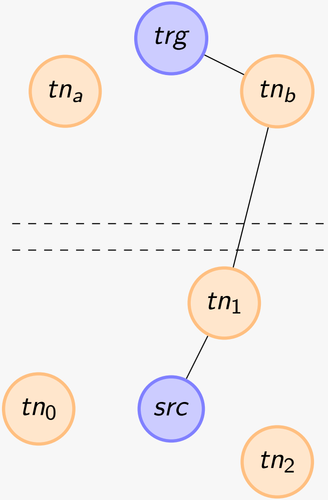
Shortest-distance queries

- Transit Nodes also work in reverse: Every 'far' trip entering a cell will do it through one of the Transit Nodes
- All 'far' trips can be split up into three parts: $src - transit_{src} - transit_{dest} - dest$
- Try all possible combinations of transit nodes to find the minimum of $d(src, transit_{src}) + d(transit_{src}, transit dest) + d(transit_{dest}, dest)$



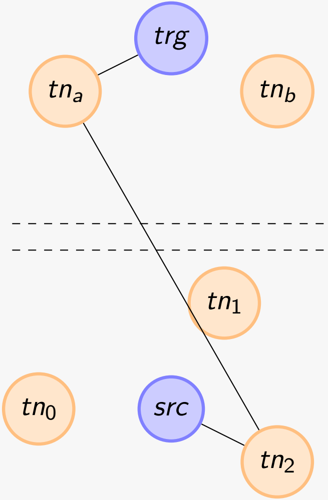
Shortest-distance queries

- Transit Nodes also work in reverse: Every 'far' trip entering a cell will do it through one of the Transit Nodes
- All 'far' trips can be split up into three parts: $src - transit_{src} - transit_{dest} - dest$
- Try all possible combinations of transit nodes to find the minimum of $d(src, transit_{src}) + d(transit_{src}, transit dest) + d(transit_{dest}, dest)$



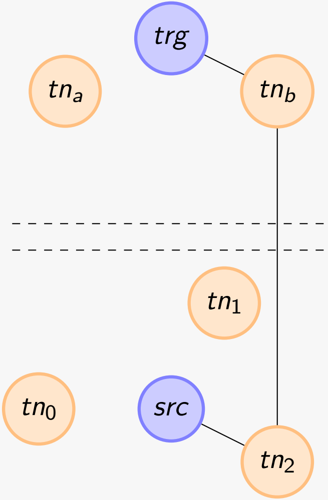
Shortest-distance queries

- Transit Nodes also work in reverse: Every 'far' trip entering a cell will do it through one of the Transit Nodes
- All 'far' trips can be split up into three parts: $src - transit_{src} - transit_{dest} - dest$
- Try all possible combinations of transit nodes to find the minimum of $d(src, transit_{src}) + d(transit_{src}, transit dest) + d(transit_{dest}, dest)$



Shortest-distance queries

- Transit Nodes also work in reverse: Every 'far' trip entering a cell will do it through one of the Transit Nodes
- All 'far' trips can be split up into three parts: $src - transit_{src} - transit_{dest} - dest$
- Try all possible combinations of transit nodes to find the minimum of $d(src, transit_{src}) + d(transit_{src}, transit dest) + d(transit_{dest}, dest)$



Shortest-path queries (with edges)

- Gradually find all nodes along the path
- Split it up into an already known part and the unknown rest
- Suppose we already know the path from src to a node u (initially $src = u$)
- To find the next step, find the neighbor v of u so that $d(u, dest) = d(u, v) + d(v, dest)$

Shortest-path queries (with edges)

- Problem: When approaching *dest* the path is no longer long enough

Shortest-path queries (with edges)

- Problem: When approaching *dest* the path is no longer long enough
- Two Solutions:
  - Reverse the search: start from *dest* instead of *src*
  - Only possible if the overall path is not too short
  - Just use another algorithm to find the shortest path

Shortest-path queries (with edges)

- Problem: When approaching *dest* the path is no longer long enough
- Two Solutions:
  - Reverse the search: start from *dest* instead of *src*
  - Only possible if the overall path is not too short
  - Just use another algorithm to find the shortest path
- It's possible to just fetch the next few steps instead of the whole path
- E.g. to just display the current region in navigation systems

Local queries

- If *src* and *dest* are less than 4 cells apart the shortest distance wasn't precomputed
- In such cases often the small roads are faster
- Use another shortest-path algorithm instead: Dijkstra, Highway Hierachies, etc.
- Most other algorithms are faster if the distance is very short

Multi-Level Grid

- Open Question: What grid size to choose?

Size	$\|T\|$	$\|T\| \times \|T\| / node$	% global queries	preprocessing
64×64	2042	0.1	91.7%	498 min
128×128	7426	1.1	97.4%	525 min
256×256	24899	12.8	99.2%	638 min
512×512	89382	164.6	99.8%	859 min
1024×1024	351484	2545.5	99.9%	964 min

- Still the same goal: Not too many Transit Nodes, almost no local queries

Multi-Level Grid

- Solution: Precompute multiple grids of different sizes
- Query: Use the coarsest grid for which the query is still non-local
- Few Transit nodes, faster query time

Multi-Level Grid

- Solution: Precompute multiple grids of different sizes
- Query: Use the coarsest grid for which the query is still non-local
- Few Transit nodes, faster query time
- Precomputation: Start with a coarse grid, do normal precomputation
- Add finer grids: Compute Transit Nodes like before, but only compute distances beween Transit Nodes if they are in the local region of the parent grid

Conclusion

- Most work done in a preprocessing step
- Shortest-path queries reduced to a few table lookups
- Query time reduced from milliseconds to microseconds
- Exact responses, not an approximation
- Other stuff: Compress preprocessed data, ...

Conclusion

- Most work done in a preprocessing step
- Shortest-path queries reduced to a few table lookups
- Query time reduced from milliseconds to microseconds
- Exact responses, not an approximation
- Other stuff: Compress preprocessed data, ...

- Interesting Problems:
- Directed graphs
- Best algorithm for local queries
- Graph changes require full recomputation

Thank you!