Efficient Route Planning

SS 2012

Lecture 8, Wednesday July 27th, 2012

(Transit Node Routing)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

UNI FREIBURG

Overview of this lecture

- Organizational
  - Feedback and results for Exercise Sheet 7 (CH, part 2)
- Transit Node Routing (TNR)
  - Last algorithm (in this course) for routing on **road** networks
  - One of the (algorithmically) fastest one to date
  - Very simple idea + very simple query algorithm
  - Various possibilities for the pre-computation … we will look at one based on Contraction Hierarchies
  - Historically TNR came two years before CH
  - Exercise Sheet 8: Implement a part of TNR

Feedback on Ex. Sheet 7 (CH, Part 2)

- Summary / excerpts   last checked June 27, 10:38
  - The way how and why Contraction Hierarchies works became much clearer in the last lecture
  - Again, not a lot of code
  - Easy to make lots of small mistakes
    - which don't show in the unit tests on small examples
    - which cause the number of shortcuts to explode in the end
  - Many could not fix all those mistakes ... frustrating

Results for Ex. Sheet 7 (CH, Part 2)

- Summary
  - Very fast precomputation: ~ 1 minute even on BaWü
  - Number of shortcuts ~ 2 million for BaWü
    - that's about the order of the number of arcs in the original graph, which is ok
  - Query times around 1 millisecond
  - All in all, clearly the best algorithm so far
  - BUT: also the hardest to implement
    - not a lot of code, but small mistakes can make everything fail ... and are hard to find (because they don't show in simple test cases)

Transit Node Routing 1/6

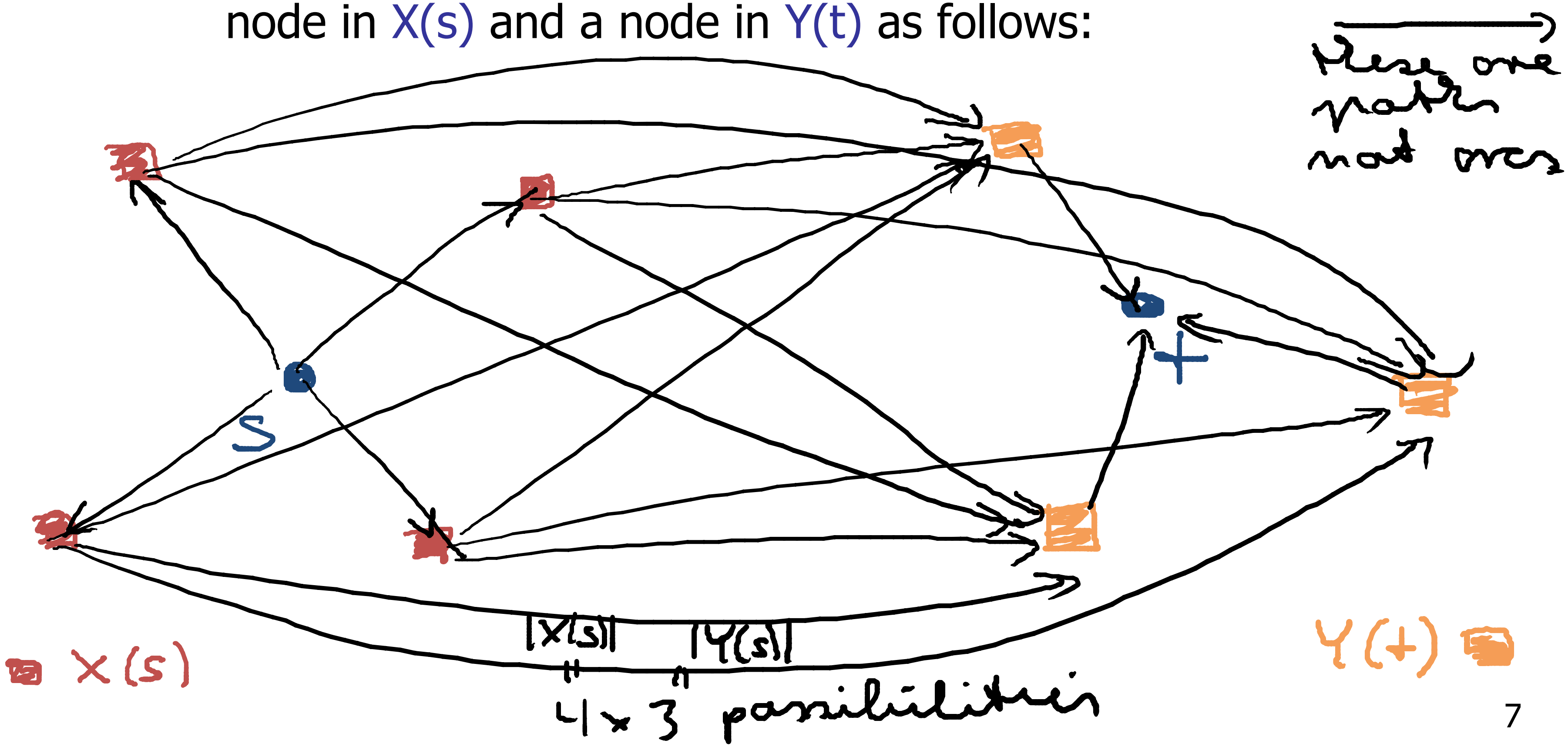
- The underlying very simple observation
  - When you go from your home to somewhere far away:
    - then the initial portion of your route will be one of a few standard routes
  - Let's look at a few examples on Google Maps
  - How can we use this to speed up shortest path queries?

Transit Node Routing 2/6

- We want to have the following
  - For each pair of nodes u and v a **"far-away" criterion** Far(u, v) that yields true or false
    - Intuitively, if Far(u, v) = true then u and v are "far away"
  - For each node u sets X(u) and Y(u) of **access nodes** such that
    - For all v with Far(u, v) = true → exists x ∊ X(u) on SP(u,v)
    - For all w with Far(w, u) = true → exists y ∊ Y(u) on SP(w, u)
    - Intuitively: when you go **from** u to somewhere "far away", you will pass through one of the X(u) ... and same for Y(u) when you go **to** u from somewhere "far away"
  - Note: for symmetric graphs, going from and going to is the same and X(u) = Y(u)

Transit Node Routing 3/6

- Processing a query from s to t — Basic idea:
  - If Far(s, t) = false (close together) ... use any algorithm
  - If Far(s, t) = true (far away) ... try all combinations of a node in X(s) and a node in Y(t) as follows:



Transit Node Routing 4/6

- Precomputation — Basic idea
  - Compute "something" such that $Far(u, v)$ can be evaluated quickly for given u and v
  - Compute and store the $X(u)$ and $Y(u)$ for each node u, as well as $dist(u, x)$ for each $x \in X(u)$ and $dist(y, u)$ for each $y \in Y(u)$
    - These are $\Sigma_u (|X(u)| + |Y(u)|)$ nodes and distances
  - Compute and store the unions $X = \bigcup_u X(u)$ and $Y = \bigcup_u Y(u)$ and the $dist(x, y)$ for all pairs x and y with $x \in X$ and $y \in Y$
    - These are $|X| \cdot |Y|$ distances
    - Our goal will be that both $|X|$ and $|Y|$ are on the order of $\sqrt{n}$ and not n, so that $|X| \cdot |Y| = O(n)$

Transit Node Routing 5/6

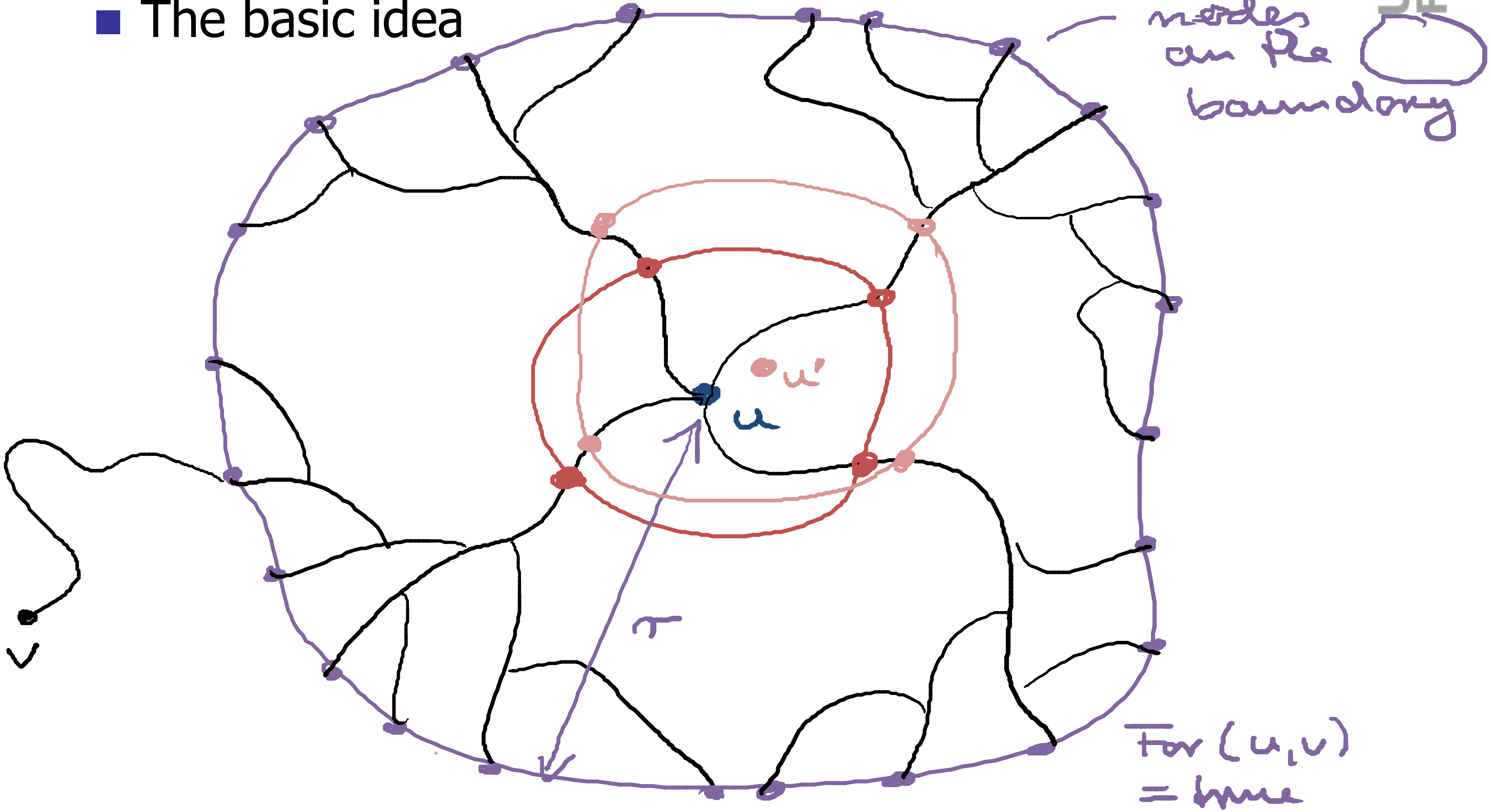
- Processing a query from s to t — Details
  - If Far(s, t) = false, compute dist(s, t) with another algorithm, for example ordinary Dijkstra; otherwise:
  - Fetch the set X(s) and the dist(s, x) for all $x \in X(s)$
  - Fetch the set Y(t) and the dist(y, t) for all $y \in Y(t)$
  - Fetch the d(x, y) for all x, y with $x \in X(s)$ and $y \in Y(t)$
  - Compute the minimum dist(s, x) + dist(x, y) + dist(y, t) over all x, y with $x \in X(s)$ and $y \in Y(t)$
    - this is the minimum over $|X(s)| \cdot |Y(t)|$ terms
    - in practice $|X(s)|$ and $|Y(t)|$ can be made as small as **5** on average ... this gives extremely fast query times

Transit Node Routing 6/6

- Efficiency
  - **Goal 1:** Far(u, v) should be very cheap to evaluate, and if Far(u, v) = false then SP(u, v) should be very cheap to compute
    - Then we can easily determine whether we have to resort to the fallback algorithm, and if so, it will be very cheap
  - **Goal 2:** X(u) and Y(u) should be ≤ a small C for (almost) all u
    - Then the X(u) and Y(u) and the distances to / from them can be stored in $\sim C \cdot n$ space, and queries can be processed in time C^2
  - **Goal 3:** $|X = \cup_u X(u)|$ and $|Y = \cup_v Y(v)|$ are $O(\sqrt{n})$
    - then the pairwise distances dist(x, y) with $x \in X$ and $Y \in Y$ can be stored in $O(n)$ space

Geometric Precomputation 1/3

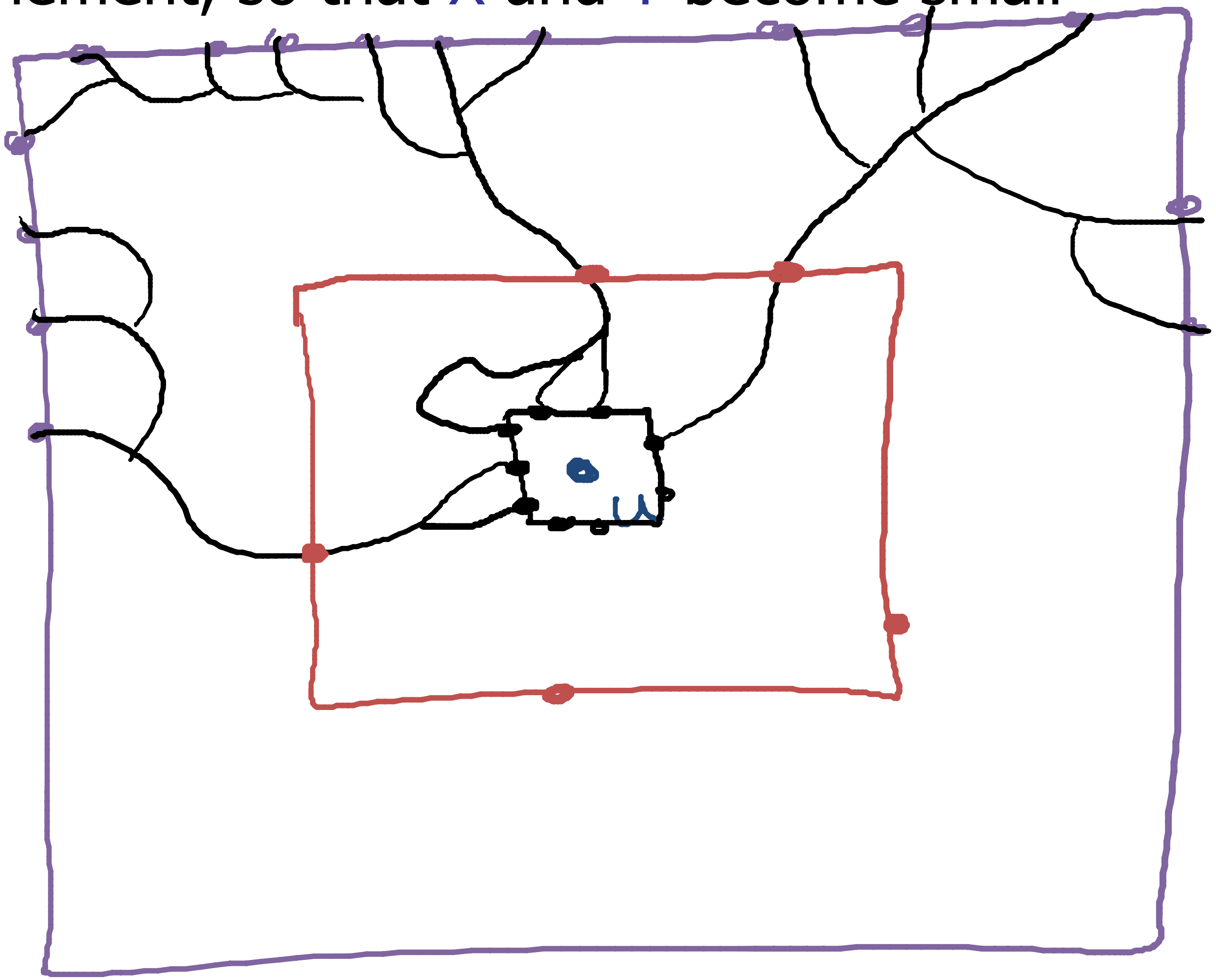
- The basic idea



For (u,v) = true
⟺ |uv| > r

Geometric Precomputation 2/3

- Refinement, so that X and Y become small



Geometric Precomputation 3/3

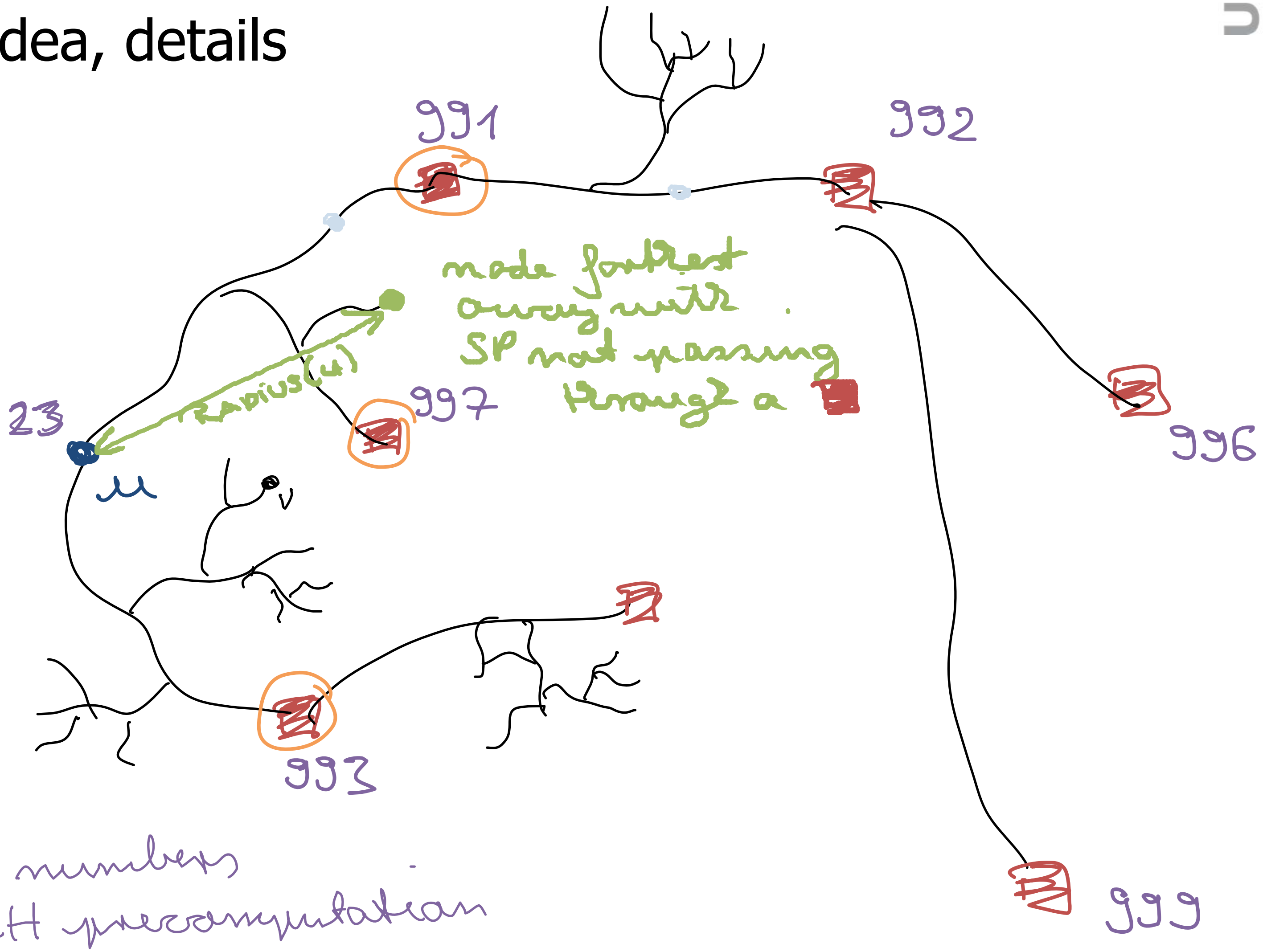
- Resource requirements
  - Small sets of access and transit nodes
  - But precomputation time comparable to that for arc flags
    (we need a Dijkstra for each boundary node of each cell)
  - There are various tricks to make this faster
  - And we can also make it hierarchical ... see later slide
  - See the references for details
  - But let's now look at a precomputation based on CH

Precomputation based on CHs 1/4

- Basic idea
  - Do the CH precomputation on the given graph
  - Let $X = Y$ be the set of nodes with ordering number above a certain threshold T (we want $|X| = |Y| \sim \sqrt{n}$)
  - For each node u in the graph do a forward search in the upward graph, and for each settled node v compute the first node $x \in X$ on $SP(u, v)$ if any; let $X(u)$ be the union of all these x
  - Similarly, compute $Y(v)$ for each node v in the graph via a backward search in the downward graph

Precomputation based on CHs 2/4

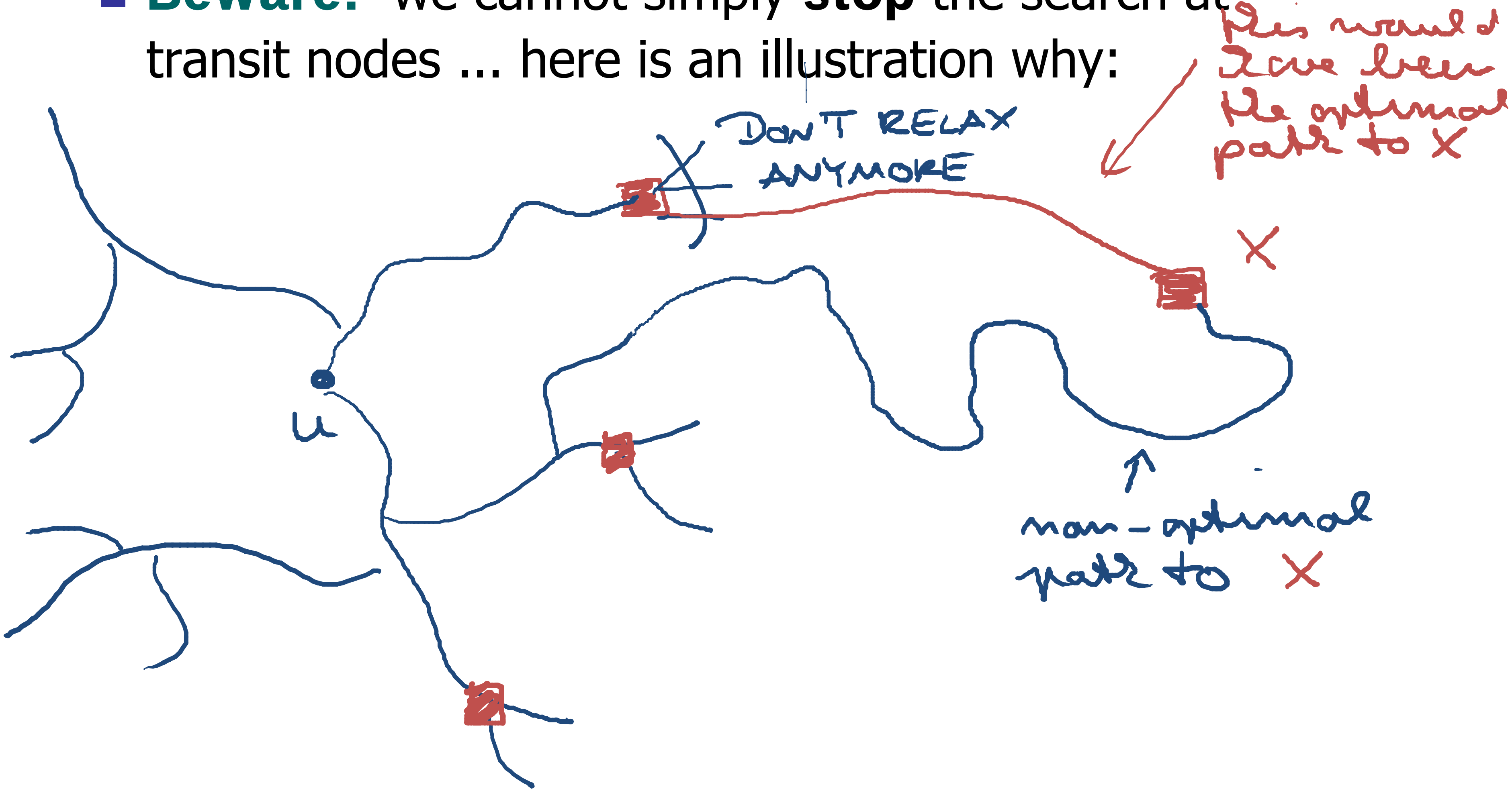
- Basic idea, details



ordering numbers
of the CH precomputation

Precomputation based on CHs 3/4

- **Beware:** we cannot simply **stop** the search at transit nodes ... here is an illustration why:



Precomputation based on CHs 4/4

- "Far-away" criterion
  - Along with the computation of X(u) ... see picture on slide 15
    - Compute the maximal geometric distance Radius(u) of a node v where SP(u, v) does **not** contain a node from X(u)
  - Define Far(u, v) = true if and only if the geometric distance from u to v is > Radius(u)
  - We can also do the same for Y(v) and thus possibly further improve our "far-away" criterion
  - For more refined "far-away" criteria, see papers in references
    - **Note:** the "far-away" criterion is called locality criterion there with exactly the opposite meaning ... quite confusing

Implementation advice 1/2

- Precomputation
  - Just do the CH precomputation and pick as transit nodes the T nodes contracted last ... it's really that simple
- Access nodes
  - For symmetric graph, X(u) = Y(u), that is we only need to compute **one** set of access nodes per set
  - For each u, you need to find the transit nodes on **all** shortest paths starting at u (in the upwards graph)
  - For each settled label, just **backtrack** the parent pointers
  - Beware: no need to backtrack further from a node which you have already seen before in the backtracking ... complexity should be #arcs in the SP tree

Implementation advice 2/2

- Simplification for Ex. Sheet 8
  - For a fully functional TNR you would need to precompute and store **all** $X(u)$ and **all** $dist(u, x), x \in X(u)$ to them
  - Similarly you would need to precompute and store **all** $dist(t_1, t_2)$ for **all** pairs of transit nodes t_1 and t_2
  - For BaWü you would probably run into memory problems
  - Instead do the following at query time, for given s and t
    - compute $X(s)$ and all $dist(s, x), x \in X(s)$
    - compute $X(t)$ and all $dist(x, t), x \in X(t)$
    - compute all $dist(x_1, x_2)$ where $x_1 \in X(s)$ and $x_2 \in X(t)$
  - But ignore the time for these three items when you benchmark the query time

Hierarchical TNR (sketch only) 1/2

- TNR can be made hierarchical, too
  - Here is an explanation for two levels of transit nodes
  - For each node, precompute and store the distances to the "closest" level-1 transit nodes (that is, the first level-1 transit nodes on paths to anywhere else)
  - For each level-1 transit node, precompute and store the distances to the "closest" level-2 transit nodes
  - Precompute and store the distances between all pairs of level-2 transit nodes
  - For a query from s to t, now try all combination of $(s, x_1, x_2, y_2, y_1, t)$, where x_1 and y_1 are the level-1 access nodes of s and t, respectively, and x_2 and y_2 are the level-2 access nodes of the respective x_1 and y_1

Hierarchical TNR (sketch only) 2/2

- Why does this make sense?
  - We need the pairwise distances only for the level-2 transit nodes
  - Therefore we can have more level-1 transit nodes and hence a better locality criterion = local searches needed only when s and t are very close together
  - But we have to try out more combinations at query time
  - Can be generalized to an arbitrary number of levels
  - Experiments suggest 5 levels for the road network of a whole continent (Western Europe or the US)
  - See the references for details

References

- Transit Node Routing, original paper

  Ultrafast Shortest-Path Queries Via Transit Nodes

  Bast, Funke, Matijevic, DIMACS Shortest Path Challenge

  http://www.mpi-inf.mpg.de/~bast/papers/transit-dimacs.pdf

- Transit Node Routing, based on HH and CH

  PhD thesis from Dominik Schultes (HH), Chapter 6

  http://algo2.iti.kit.edu/schultes/hwy/schultes_diss.pdf

  Master thesis from Robert Geisberger (CH), Section 4.2

  http://algo2.iti.kit.edu/documents/routeplanning/geisberger_dipl.pdf

- Transit Node Routing, article in Science Magazine

  Fast Routing in Road Networks with Transit Nodes

  http://www.sciencemag.org/content/316/5824/566.short