Efficient Route Planning

SS 2012

Lecture 2, Wednesday May 2nd, 2012
(Dijkstra's algorithm, Connected Components)

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

UNI FREIBURG

Overview of this lecture

- Organizational
  - Your feedback and results on Exercise Sheet 1
  - Course Systems: Jenkins
- Dijkstra's Algorithm
  - Idea + Example
  - Correctness proof
  - Implementation advice
  - Connected Components (CCs) using Dijkstra
  - Exercise Sheet 2: Implement Dijkstra + use it to compute the largest CC + use it for some random shortest path queries on Saarland and BaWü

Your Feedback on Exercise Sheet 1

- Summary / excerpts last checked May 2, 15:42
  - Interesting / entertaining, but also quite time-consuming
    - 6 hours for some, up to 15 / 20 / 30 hours for others
    - lack of programming practice
    - setup problems: gtest, SVN, Linux, IDE, etc.
  - Implementation advice from the lecture was useful
  - Memory problems with Java and the BaWü dataset
  - Parsers like Xerces are slow and use a lot of memory
  - Some fights with checkstyle / cpplint
  - How to compute with latitude-longitude coordinates?
  - Let's look at your results ...

Computing with lat-lng coordinates

- Distance in meters between two such coordinates
  - You can use the following approximations
    - one degree of latitude = 111,229 meters
    - one degree of longitude = 71,695 meters
  - Using this, you can easily compute

```
int diffLat = ...;  // Difference of latitude in meters.
int diffLng = ...;  // Difference of longitude in meters.
```

  - From that you can compute the distance in meters

```
int dist = sqrt(diffLat * diffLat + diffLng * diffLng);
```

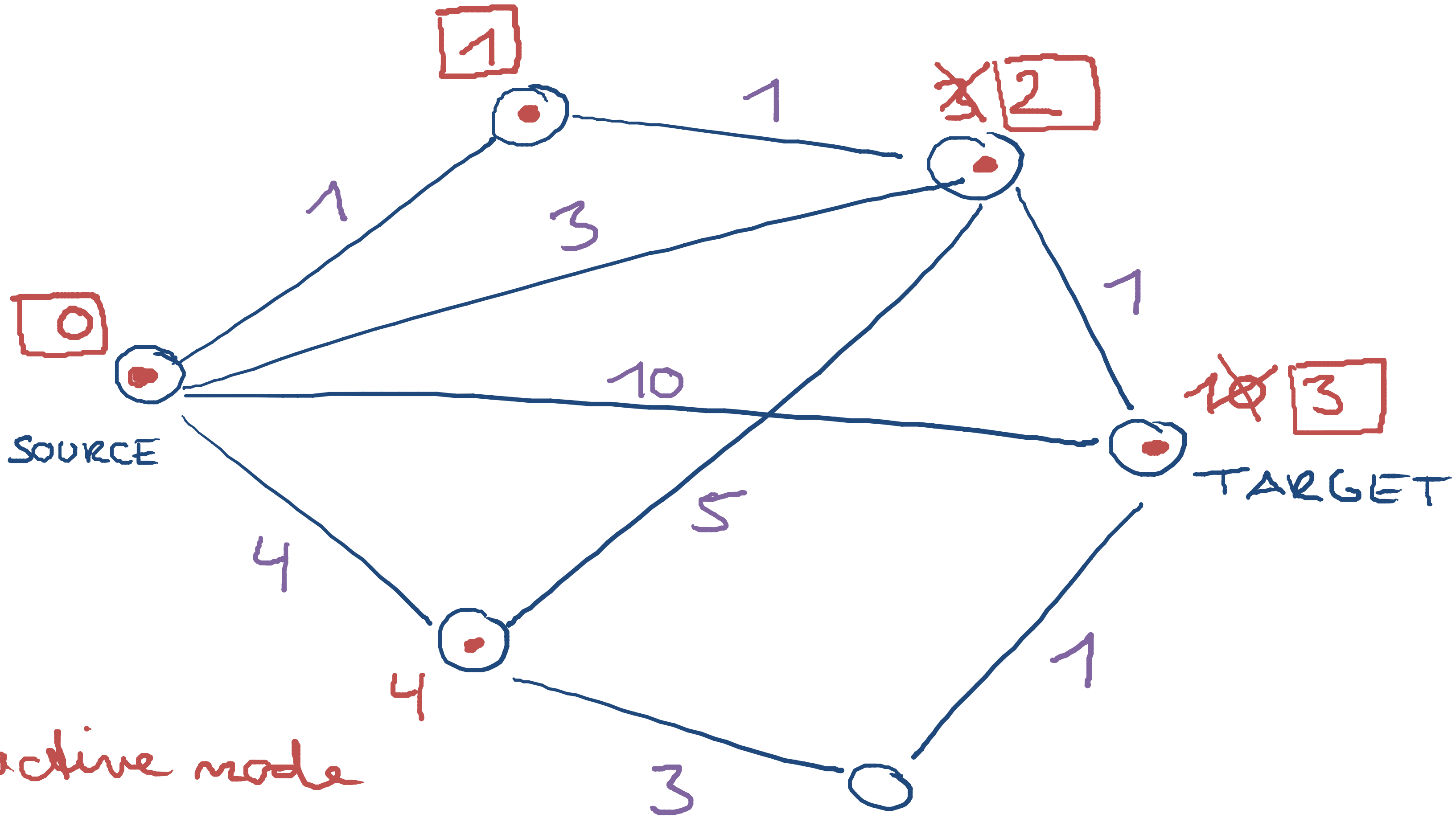
Shortest Path Queries

- Point to point queries
  - For most of this lecture, we are interested in finding the shortest path (path of minimal cost) between two given nodes A and B, called source and target node
  - The cost of a path is simply the sum of the costs of the arcs along the graph
  - The standard algorithm for this task is Dijkstra's algorithm

Dijkstra's algorithm

- You have probably heard it before, here is a recap:
  - Maintains a priority queue of active nodes with tentative distances
  - Initially only the start node is active, with tentative distance 0, all other tentative distances are ∞
  - In each iteration, pick the active node with the **smallest** tentative distance and change its status from active to settled
    - if all arc costs are non-negative, the tentative distance of each settled node is guaranteed to be the correct distance
  - Relax the outgoing arcs = see if the tentative distances of the adjacent nodes can be improved, if yes do so
  - Stop when the target node is settled

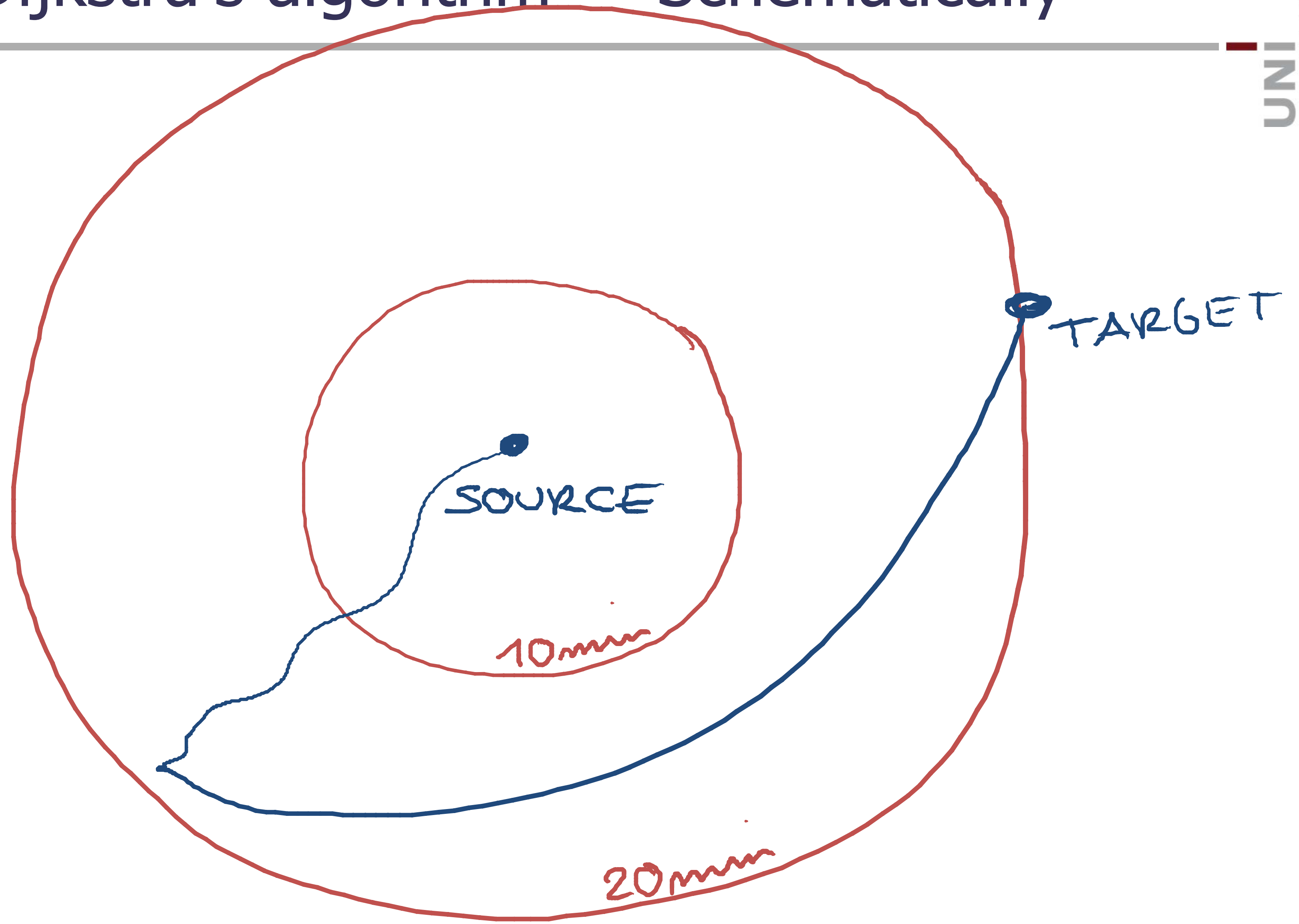
Dijkstra's algorithm — Example



• active node

☐ settled node

Dijkstra's algorithm — Schematically



Dijkstra's algorithm — Properties

- Some basic properties
  - When the target node has been settled, with cost c, than **all** other nodes with cost $< c$ have been settled, too
    - wost case: all nodes reachable from source are settled
  - Running time is $O((m + n) \cdot \log n)$, where
    - m = number of relaxed arcs (worst case: all arcs)
    - n = number of settled nodes (worst case: all nodes)
  - The $\log n$ is the cost of a priority queue (PQ) operation
    - one (potential) insert per arc, one deleteMin per node
    - for a state-of-the-art PQ: 1 µs / deleteMin dominates
    - hence Dijkstra can settle ≈ 1 million nodes / second

Dijkstra — Correctness proof 1/3

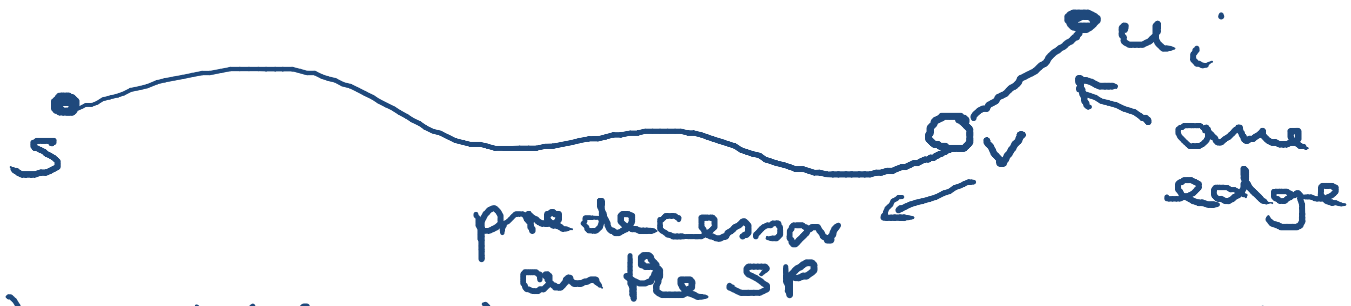
- Let s be our source node
  - Let's first make the simplifying assumptions that the $\mathrm{dist}(s, u)$ are **distinct** for all nodes u
  - Then we can order the nodes $u_1, u_2, u_3, \ldots$

    such that $\mathrm{dist}(s, u_1) < \mathrm{dist}(s, u_2) < \mathrm{dist}(s, u_3) < \ldots$
  - We want to prove that, at the end of the computation,
    - the tentative distance $\mathrm{dist}[u_i]$ for each node u_i satisfies $\mathrm{dist}[u_i] = \mathrm{dist}(s, u_i)$
  - More specifically, we can show that in the i-th iteration
    - Dijkstra's algorithm settles node u_i
    - And at that point $\mathrm{dist}[u_i] = \mathrm{dist}(s, u_i)$

Dijkstra — Correctness proof 2/3

- We show by induction over i
  - that in the i-th iteration, we have $dist[u_j] = dist(s, u_j)$ for all $j \leq i$, and node u_i will be settled in that iteration

Let's look at the SP from s to u_i



$dist(s,v) < dist(s,u_i)$ assuming $c(v,u_i) > 0$

Then, by assumption, v is one of the $u_1, \dots, u_{i-1}$

Let $j < i$: $v = u_j$

Dijkstra — Correctness proof 3/3

By induction hypothesis, v was settled in round $j < i$ and $dist[v] = dist(s, v)$.

But when v was settled, $dist[u_i]$ was set to $dist(s,v) + c(v, u_i) = dist(s, u_i)$.

Let's look at u_j with $j > i$.

For them $dist(u_j) \geq dist(s, u_j) > dist(s, u_i)$

Therefore u_i settled before each such u_j.

$\Longrightarrow$ u_i is settled in round i
with $dist[u_i] = dist(s, u_i)$ ∎

Dijkstra — Implementation advice 1/3

- Where to implement it in your code, two options:
  - As another method in your class RoadNetwork
  - In a separate class DijkstrasAlgorithm
  - I recommend the second option, reasons include:
    - gives you more freedom to extend it later
    - has or will have quite some complexity on its own, and so merits a class on its own
    - each of the more sophisticated algorithms to come will also have a class on their own
    - enables base class for all shortest path algorithms
  - You find a skeleton for the second option on the Wiki

Dijkstra — Implementation advice 2/3

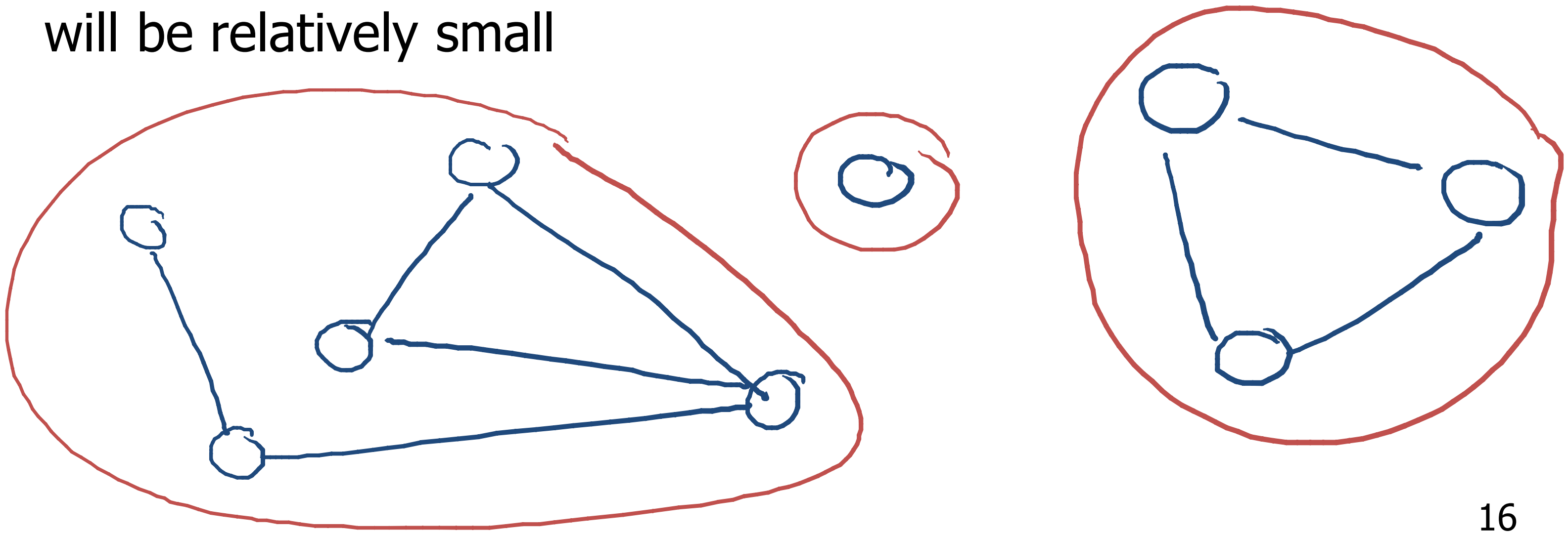
- Stopping criterion
  - It will be useful to support two modes of operation
    - stop when **a given target node** is settled
    - stop when **all reachable nodes** are settled
  - You can easily support both of these by always passing two arguments, sourceNode and targetNode, and for the second mode call with a value -1 for targetNode

Dijkstra — Implementation advice 3/3

- Standard Dijkstra requires a decrease-key operation
  - The tentative distance of a node in the priority queue (PQ) can decrease several times over the course of the execution
  - Requires an operation to decrease the key of a given PQ item
  - But PQs like the std::priority_queue, don't support this
- There is a simple trick to avoid this operation
  - Instead of a decrease-key, insert the node (again) with the smaller tentative distance
  - Whenever a node with key larger than the already known tentative distance is removed from the PQ, **ignore it**
  - Works fine as long as there are relatively few decrease-key operations, which is the case for road networks **why?**

Connected Components 1/2

- Definition
  - On an undirected graph, a connected component (CC) is a maximal subset C of nodes such that for all pairs x, y ϵ C there is a path between x and y
  - Our two OSM graphs are likely to have more than one connected component
  - But one will contain most of the nodes, and the other CCs will be relatively small



Connected Components 2/2

- Easy to compute using Dijkstra
  - Add a member variable Array<int> visitedNodes to your class DijkstrasAlgorithm, with one entry per node, and all entries initialized to 0
  - Proceed in rounds 1, 2, ... and in round i do:
    - If no more nodes are marked 0 we are done
    - Pick any node still marked 0 and run Dijkstra from that node until all nodes are settled
    - Mark all nodes visited on the way with i
  - Now it's easy to identify the connected components, and in particular the largest one

visitedNodes

1	1	2	1	3	2	1	1	1	2	2	3	1	1	3	3

0 1 2 3 4 5 6 ... 15

Jenkins

- Jenkins is a continuous build systems
  - Checks out your code from our repository
  - Does compile, test, and checkstyle
  - Makes sure that you committed all the necessary files and that everything works fine
  - Triggered by every SVN change or manually
  - If an error occurs, an email will be sent to you
  - You find the link to Jenkins on your Daphne page
  - From now on check that whatever you commit passes through Jenkins without errors, and if not correct it

References

- Dijkstra's Algorithm
  – http://en.wikipedia.org/wiki/Dijkstra's_algorithm
- Connected Components
  – http://en.wikipedia.org/wiki/Connected_component_(graph_theory)
- Jenkins
  – https://daphne.informatik.uni-freiburg.de/jenkins