Efficient Route Planning

SS 2011

Lecture 7, Friday July 1st, 2011
(Contraction Hierarchies again, implementation advice)

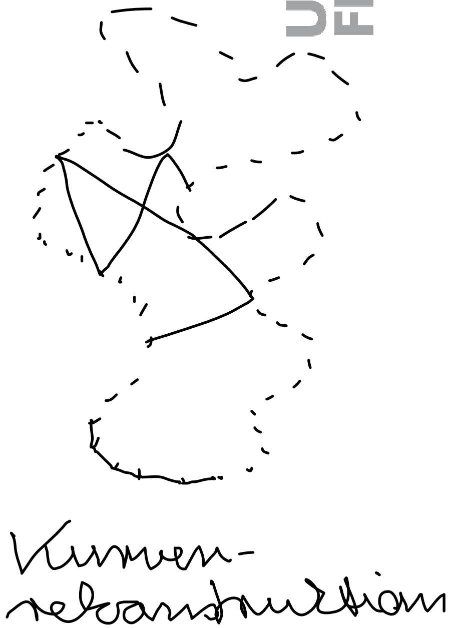
Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

UNI FREIBURG

Overview of this lecture

- Previous stuff
  - Your web applications from Exercise Sheet #4
  - Don't be shy to give us feedback!
  - How will the exam look like?
- Contraction Hierarchies again
  - Recapitulation of the main parts of the algorithm
  - A non-trivial working example
  - All kinds of implementation advice
  - Please ask questions when something is not 100% clear!

  - No new exercise sheet, please finish the last one (#5) until Friday next week (July 8, 2 pm)



Feedback

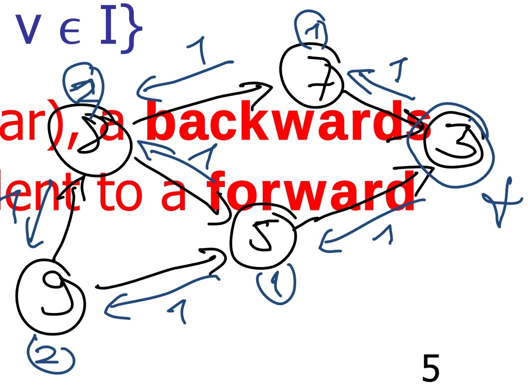
- Don't be shy to give us feedback
  - If you have any questions, ask!
  - If you think it's too much, say it!
  - Don't complain afterwards, complain before!

Exam

- The exam will be on Monday, August 15 at 2 pm
  - It will last (only) 90 minutes
  - There will be **4 tasks**, out of which you can select **3 tasks**
  - Three **kinds** of tasks are possible
    - Execute an algorithm from the lecture, or some variant of it, on a given example (on paper)
    - Write a small program to solve a variant of a problem we have seen in the lecture
    - Compute, reason about, or prove a non-trivial (but also not very difficult) property of an algorithm or data structure from the lecture, or some variant of it
  - See the Search Engines WS 2009/2010 exam for examples

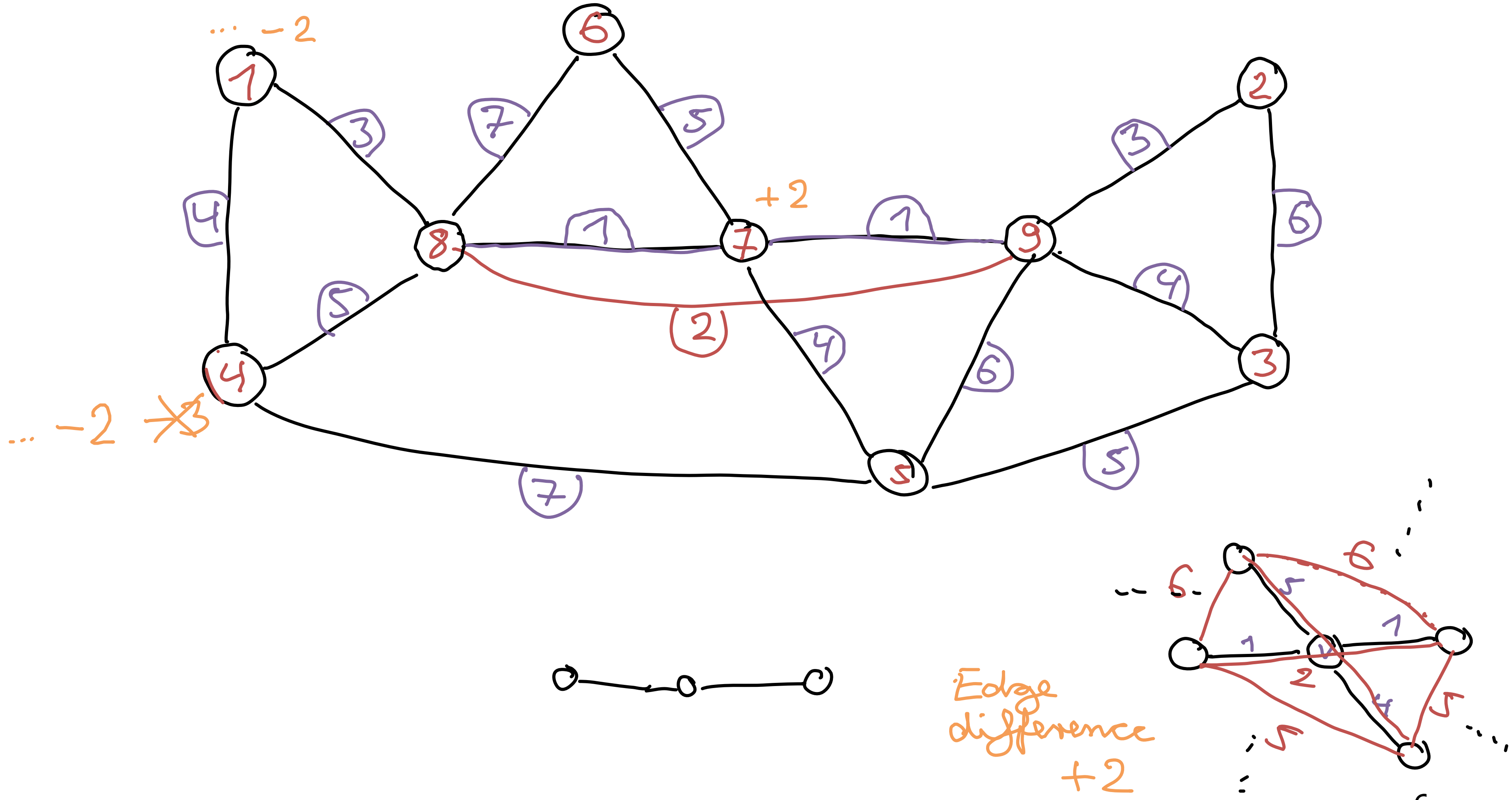
Contraction Hierarchies — Reprise

- Query algorithm, for given source s and target t
  - Do a full Dijkstra computation from s **forwards**, considering only arcs (u, v) with u < v
    - we call G↑ = (V, {(u, v) : u < v}) the **upward graph**
  - Do a full Dijkstra computation from t **backwards**, considering only arcs (u, v) with u > v
    - we call G↓ = (V, {(u, v): u > v}) the **downward graph**
  - Let I be the set of nodes settled in both Dijkstras
  - Take dist(s, t) = min {dist(s, v) + dist(v, t) : v ϵ I}
  - NOTE: for **symmetric** graphs (like ours so far), a **backwards search in the downward graph** is equivalent to a **forward search in the upward graph**



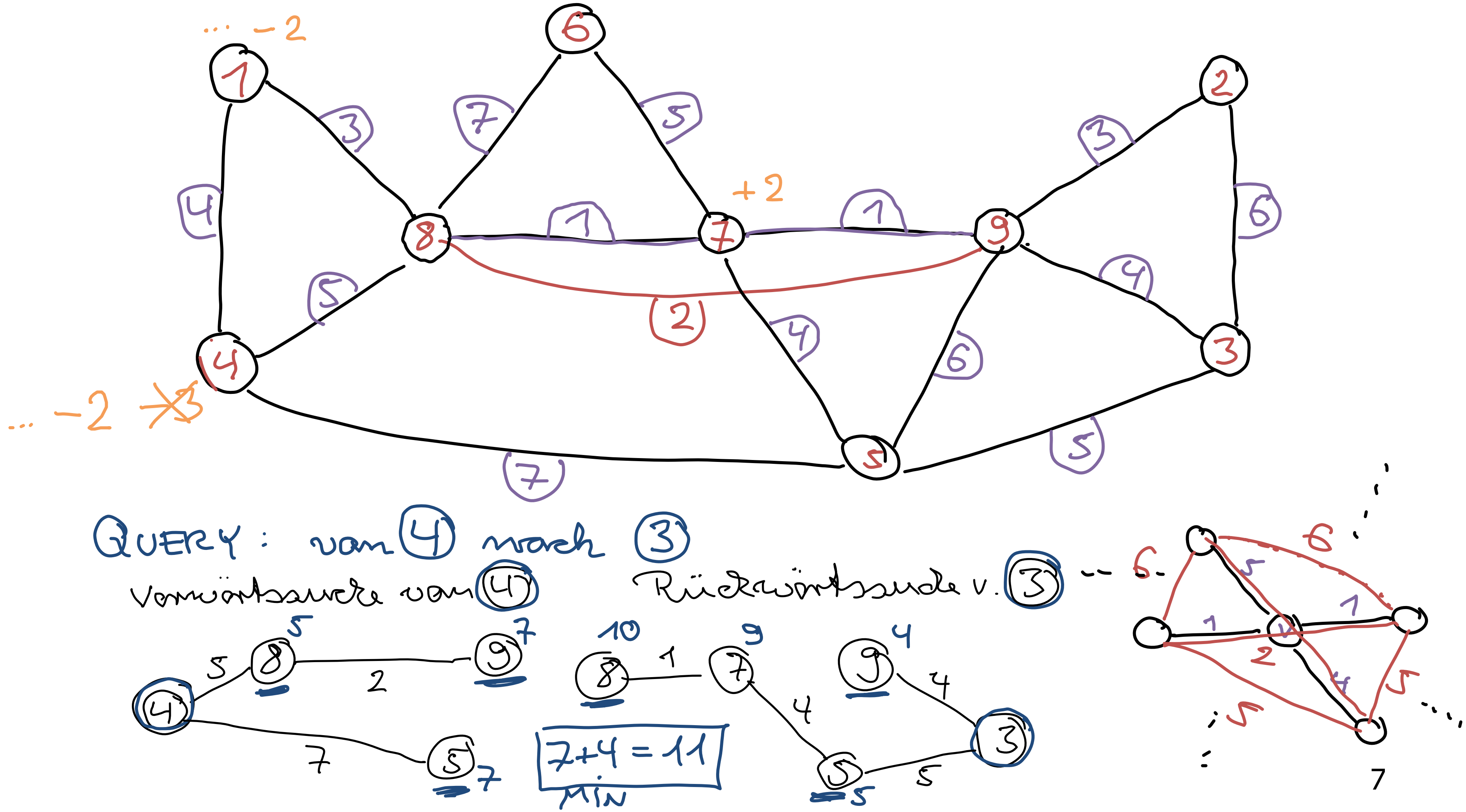
Contraction Hierarchies — Example 1/2

- The **full** pre-processing for a small graph



Contraction Hierarchies — Example 2/2

- A query on the fully pre-processed graph



Implementation Advice 1

- Disclaimer:
  - The following is just advice
  - You don't have to do it that way
  - It's a good and clean way to do things, however
    - based on quite some experience
    - with writing code in general
    - and with writing code for route planners in particular

Implementation Advice 2

- The canonical classes to have (so far)
  - class RoadNetwork
  - class RoutePlanningAlgorithm
  - class Dijkstra : public RoutePlanningAlgorithm
  - class Landmarks : public RoutePlanningAlgorithm
  - class ArcFlags : public RoutePlanningAlgorithm
  - class ContractionHierarchies : public RoutePlanningAlgorithm
  - class RoutePlanner
  - Let's have a look at my code ...

Implementation Advice 3

- The RoadNetwork class (or however you call it)
  - A good and simple implementation is with

    ```
    vector<Node> _nodes;
    vector<vector<Arc>> _adjacencyLists;
    ```

  - Have a DebugString method or something like that for writing the whole graph to a human-readable strins
  - Have a method for addings nodes and arcs
  - Or even better: a method for parsing a graph from a string in the same format as output by DebugString
  - That will be a great asset in testing and debugging
  - Let's have a look at my code ...

Implementation Advice 4a

- Dijkstra's algorithm and its **many** variants
  - Have a separate Dijkstra class with member variables like

    _estimatedCostsToTarget (for A*, landmarks, ...)

    _nodesToBeIgnored (for arc flags, contr. hierar., ...)

    _nodesToBeSettled (for contraction hierarchies, ...)

    _costUpperBound (for contraction hiearchies, ...)
  - Have setters for these member variables
  - Have a **single** implementation of computeShortestPath
    - containing various if statements that are executed conditional on the values of the above member variables

Implementation Advice 4b

- Dijkstra's algorithm and its **many** variants
  - Similarly, return anything else but the cost via member variables like

    size_t _numNodesSettled

    size_t _numArcsRelaxed

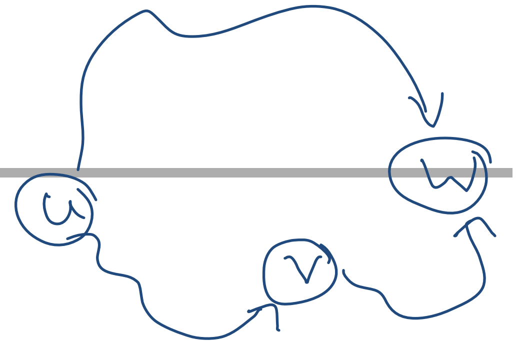
    vector<Arc> _arcsOnShortestPath
  - That avoids having a computeShortestPath method with an exorbitant number of arguments (or a complex result object, which is only partly used by most applications)
  - Let's have a look at my code

Implementation Advice 5

- Have a separate class ContractionHierarchies
  - Which has a RoadNetwork as a member variable
  - Better **not** to have CH-specific information (like the node ordering or which nodes are contracted already) in the RoadNetwork class, but as member variables in the ContractionHiearchies class

    vector<int> _orderingNumbers;
    vector<bool> _nodesContractedMarks;
  - Otherwise your RoadNetwork class (or your Node and Arc class) will get cluttered up with information from all kinds of difference algorithms
  - Let's have a look at my (preliminary) class ...

Implementation Advice 6



- How to contract a node v
  - For each pair of u and w with arcs (u, v) and (v, w), we need to figure out whether dist(u, w) in the graph **with** v is **strictly** better than the dist(u, w) in the graph **without** v
  - Compute d_{uv} = dist(u, v) in the graph **with** v
    - will be fast because $d_{uv} \leq$ cost of the arc (u, v)
  - Compute d_{vw}= dist(v, w) in the graph **with** v
    - will be fast because $d_{vw} \leq$ cost of the arc (v, w)
  - Then compute a lower bound d'_{uw} on dist(u, w) in the graph **without** v, stopping the search after cost $d_{uv} + d_{vw}$
    - without such a bound, this search could take **very** long

Implementation Advice 7

- How to contract a node v, optimizations
  - Do the shortest path computations for all pairs (u, w) simultaneously, in two parts
  - Do a single Dijkstra from u until all neighbours are settled
  - Do a single Dijkstra from each neighbour of v until all the other neighbours are settled (in the graph without v)
  - Use something like the _nodesToBeSettled explained earlier
  - Note: we might add slightly more shortcuts that way, because shortcuts introduced due to one pair (u, w) can now no longer influence the computation for another pair (u', w')
    - I don't think this is a big issue in practice though

Implementation Advice 8

- How to ignore nodes in a Dijkstra search
  - Use a vector<bool> with a mark for each node

    (more efficient than a hash map!)
  - In the Dijkstra class have one member variable which is a pointer to such a vector<bool>, as explained before
  - That way, you do not need to clutter up the Dijkstra class with information / code that is very algorithm-specific ... like arc flags or contracted nodes

Implementation Advice 9

- Augmenting the graph
  - Pay attention when you add an arc by just appending it to one of the elements from your vector<vector<Arc>>
  - It could happen that you already have an arc (u, w) with cost c1, and a contraction will add a shortcut (u, w) with cost c2 < c1
  - It depends on your implementation of Dijkstra, whether two arcs between the same pair of nodes is a problem or not, but be aware of the issue
  - To be on the safe side, search the adjacency list before you add a new arc, and just update the cost, if the arc is already there

Implementation Advice 10

- How to compute the edge difference
  - You can also use the method contractNode for that
  - Again, use member variables to change the calling mode and return result values

    bool _computeEdgeDifferenceOnly

    int _lastEdgeDifferenceComputed