# Efficient Route Planning

## SS 2012

### Lecture 11, Wednesday July 18th, 2012

### (Transfer Patterns, Course Evaluation)


Prof. Dr. Hannah Bast

Chair of Algorithms and Data Structures

Department of Computer Science

University of Freiburg

# Overview of this lecture

- Organizational
  - Your results from Ex. Sheet #10 (Multi-Criteria Costs)
  - This is the **next to last** lecture → Course Evaluation
  - Reminder: exam date is **Monday, August 20, 2:00pm**
- Transfer Patterns Routing
  - An algorithm that works well on transit networks
  - That's also the algorithm at work behind **Google Maps**
- Exercise sheet … the last one!
  - Fill out the **Evaluation Sheet** for this course → 20 points
  - Compute #transfer patterns for a subset of all station pairs

# Feedback from ES#10 (Multi-Criteria)

- Summary / excerpts last checked July 18, 14:54
  - Nice and relaxing exercise
  - Good for understanding the concept of Pareto sets in detail
  - Good to have a mandatory proof
  - First time it was indeed just a few lines of code

# Official Course Evaluation

- Please submit until the **end of this week**
  - Because I would like to discuss the feedback together with you in the next (=last) lecture
  - You get **20 points** for this ... with which you can replace the points from your worst exercise sheet
  - Just write in your feedback-exercise-sheet-11.txt that you submitted the form (provided you did)
  - Please **take your time** to fill out the form
  - The free text comments are of particular interest to us
  - Don't forget to comment on the tutors as well
  - Please by **honest** and **concrete**

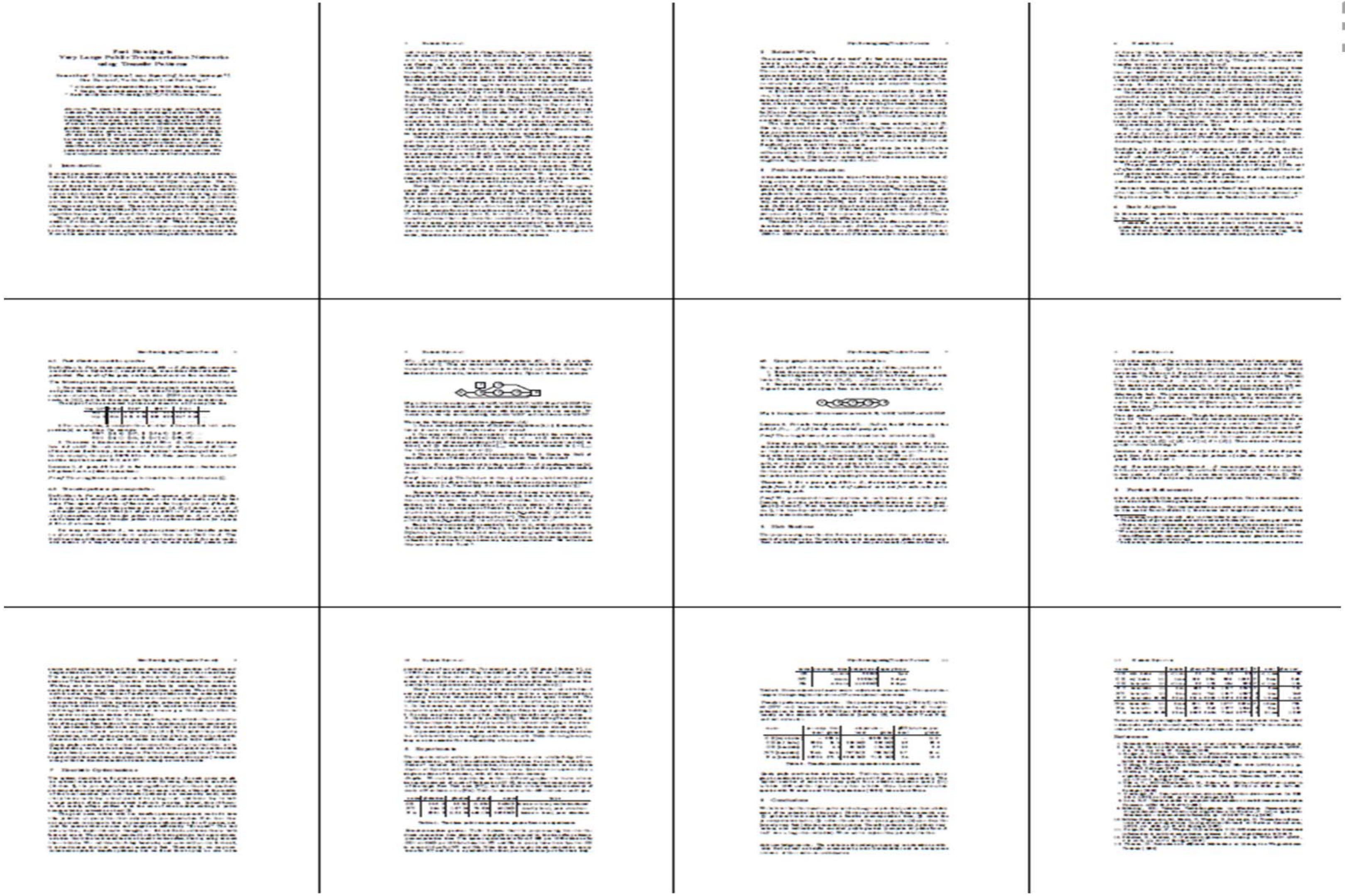
# Transfer Patterns 1/4

- An algorithm designed for transit networks
  - Trying to exploit what is special about transit networks
  - But what could this be? So far we have only seen things which are harder on transit networks than on road networks
  - Here is one thing special about transit networks:

    **transfers**

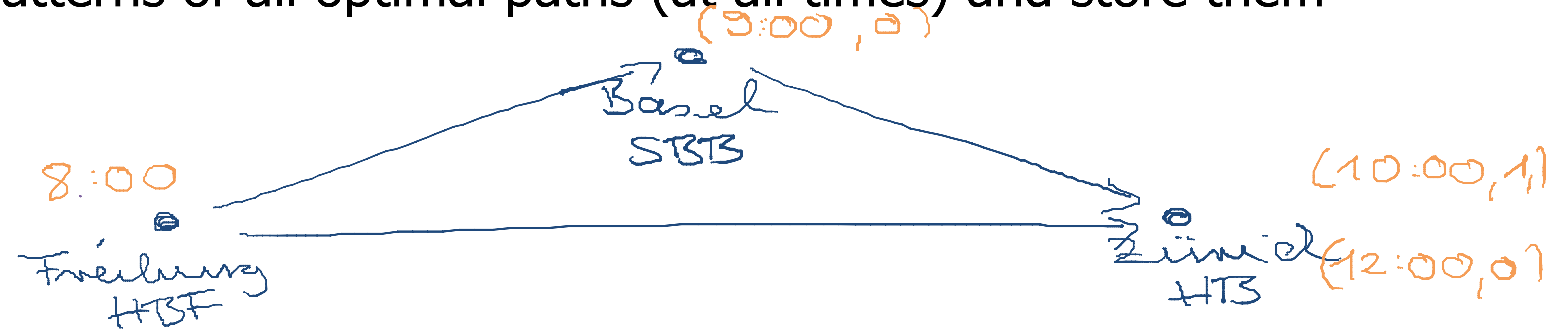
  - Even when you take a very long trip, the number of transfers is almost always a very small number
  - More than that, for a given source and destination, there is only a small number of "**patterns**" of where you transfer

# Transfer Patterns 2/4



# Transfer Patterns 3/4

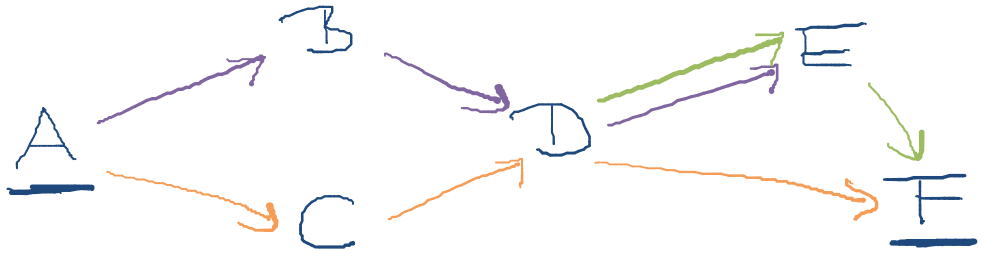
- The basic idea on one slide
  - The **transfer pattern** of a path = the sequence of stations on the path where one **transfers**, including start and end
  - **Idea:** for each pair of stations, precompute all transfer patterns of all optimal paths (at all times) and store them




  - Then, at query time, do a time-dependent Dijkstra computation on this so-called **query graph**, where each arc evaluation is again a shortest path query, but restricted to **no transfers**
  - Such **direct-connection** queries are easy to compute fast

# Transfer Patterns 4/4

- A more complex example




→ LINE 1
8:10, 8:40, 9:10, ...
(starting times from A)

→ LINE 2
8:00, 8:30, 9:00, ...
(starting times from A)

→ LINE 3
8:30, 9:30, 10:30, ...
(starting times from D)

Time / leg: 5 min.
Transfer buff.: 5 min.

All optimal transfer patterns between A and F:

AF (proof: 8:00 A $\xrightarrow{\text{LINE 2}}$ 8:15 F)

ADF (proof: 8:10 A $\xrightarrow{\text{LINE 1}}$ 8:20 D; 8:30 D $\xrightarrow{\text{LINE 3}}$ 8:40 F)

AEF (proof: 8:10 A $\xrightarrow{\text{LINE 1}}$ 8:25 E; 8:35 E $\xrightarrow{\text{LINE 3}}$ 8:40 F)

Both of these patterns are optimal but one of them would be enough!

# Components of a Transfer Pattern Router

- Transfer patterns precomputation
  - Compute (parts of) all transfer patterns of all optimal paths
- Direct-connection tables precomputation
  - Compute data structure for fast direct connection queries
- Query Graph Construction
  - Build the query graph of all transfer patterns between A and B
- Query Graph Evaluation
  - Dijkstra search on query graph, with arcs = direct connections
- Various Refinements / Optimizations
  - For example: filter out rare transfer patterns, …

# Direct-Connection Queries

- One table per "line", let us call this one L17

| | | | | | | |
|---|---|---|---|---|---|---|
| Stations: | S154 | S97 | S987 | S111 | … | |
| Time from start: | 0min | 7min | 12min | 21min | … | |
| Start times: | 8:15 | 9:15 | 10:15 | 11:20 | 12:20 | … |

- Lines per station (with positions in the respective line table)

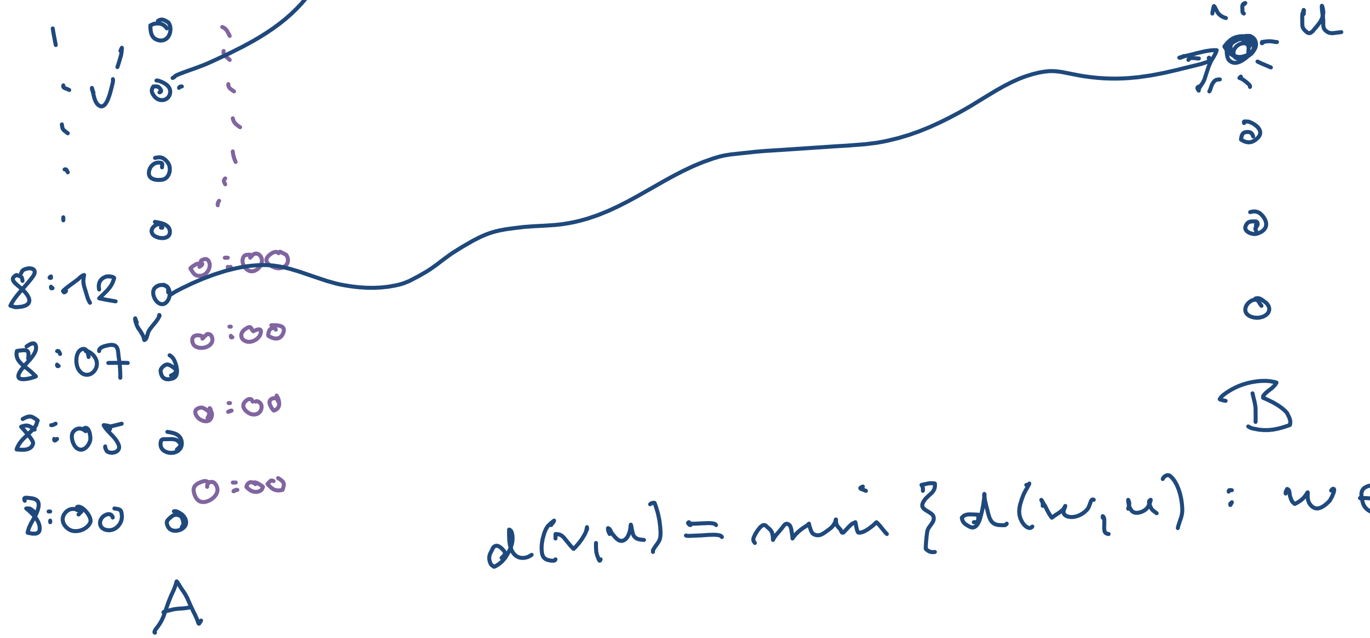
  Station S97: (L8, 4) (L17, 2) (L34, 5) (L87, 17) …

  Station S111: (L9, 1) (L13, 5) (L17, 4) (L55, 16) …

- Example query from S97 @ 10:20 to S111
  - Intersect the lists of the two stations : (L17, 2 → 4) …
  - Find time from start to S97 and to S111 : 7min and 21min
  - Find first start time after 10:20 – 7min : 10:15 → **depart 10:22**
  - Compute arrival time at S111 : 10:15 + 21min → **arrive 10:36**

# Transfer patterns precomputation 1/4

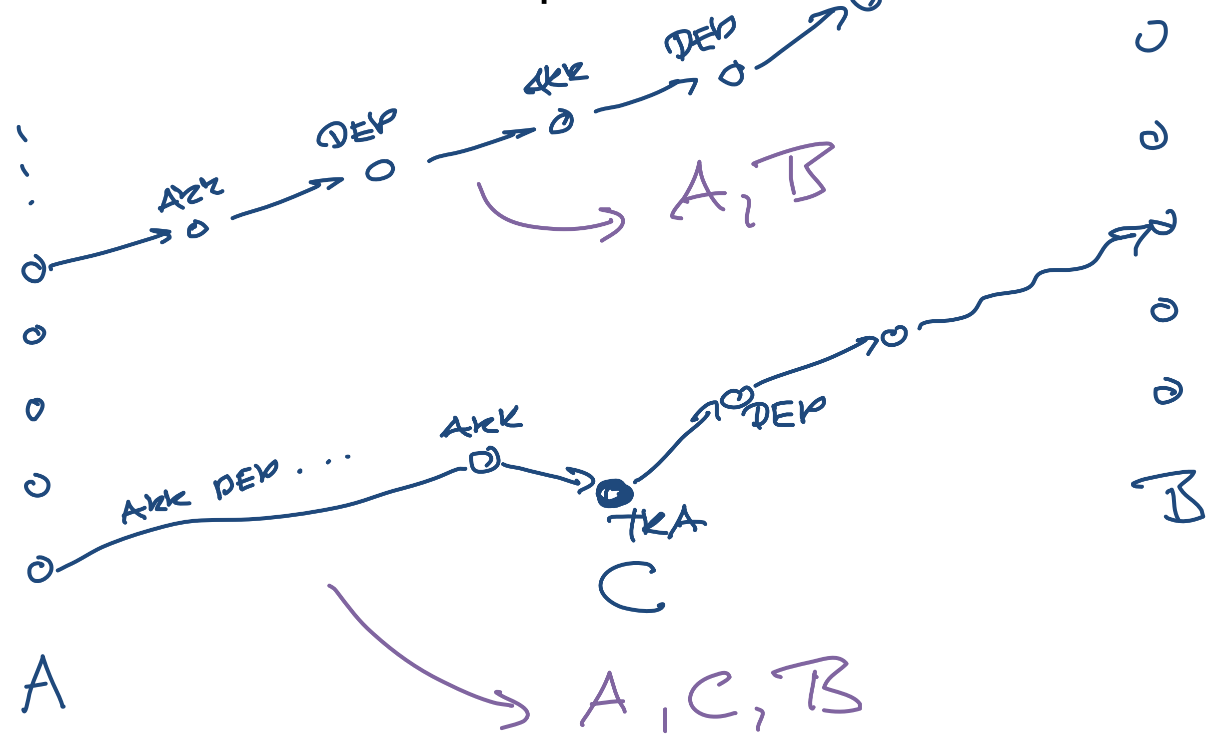
- Can be done via a Set-Dijkstra search
  - For each station A, do a Dijkstra starting from **all nodes** at that station (all with cost = travel time zero)
  - For each other node u in the graph, this will give us the path from the latest node at A so that u can still be reached




$$d(v,u) = \min\{d(w,u) : w \in A\}$$

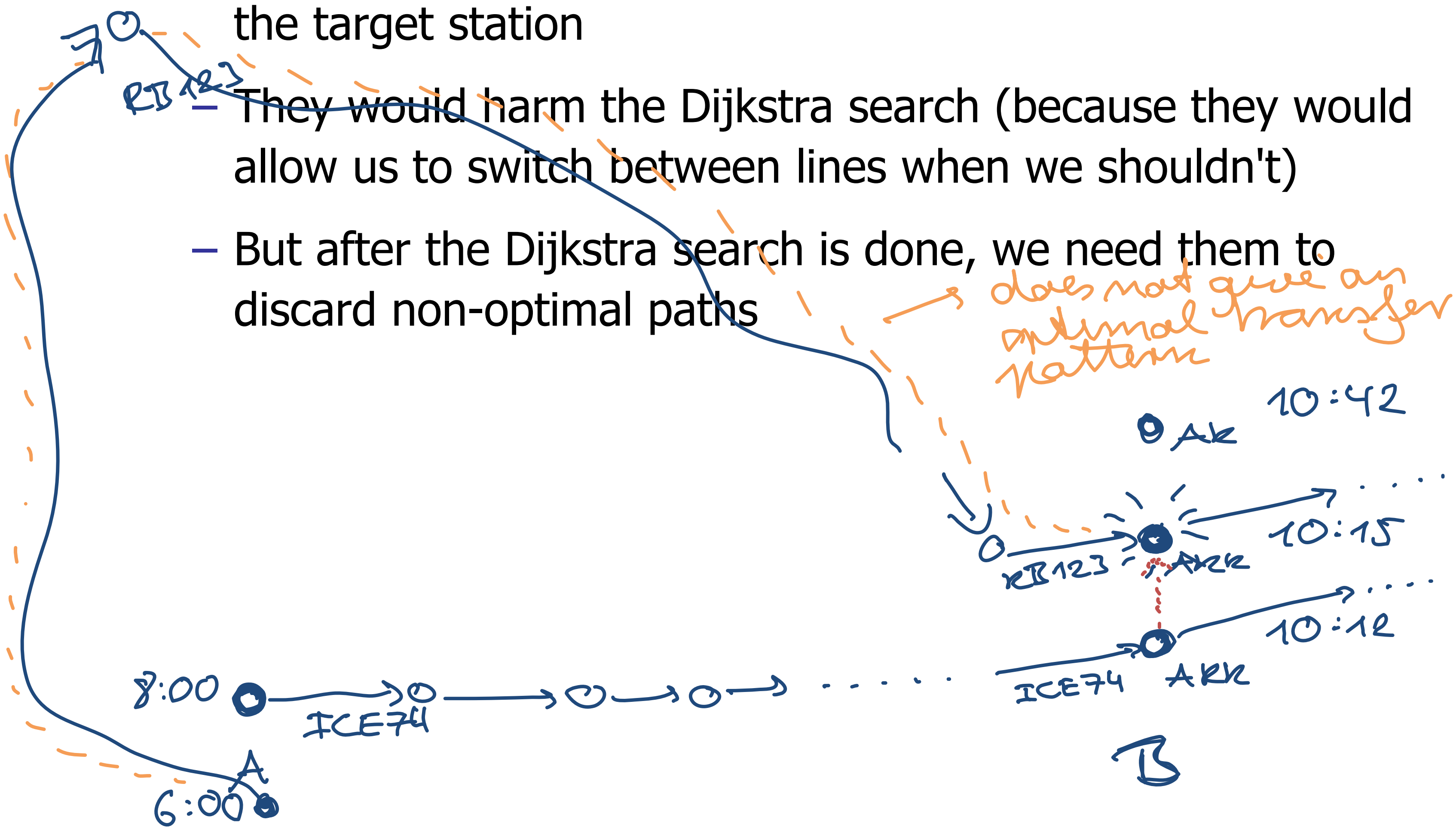
# Transfer patterns precomputation 2/4

- Now we have all optimal paths at all times
  - To obtain the transfer patterns for a station pair (A, B), simply trace back, in the Dijkstra search from A, the paths from all nodes in B and keep track of the transfers



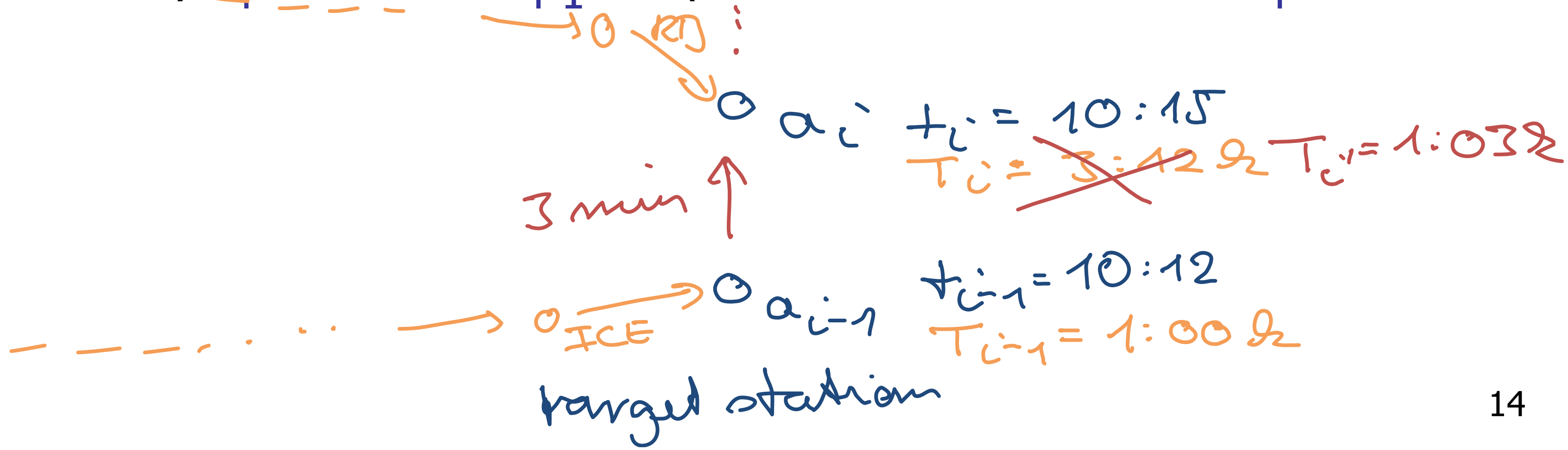
# Transfer patterns precomputation 3/4

- Beware of non-optimal paths to arrival nodes
  - Note that there are no arcs between the arrival nodes at the target station
  - They would harm the Dijkstra search (because they would allow us to switch between lines when we shouldn't)
  - But after the Dijkstra search is done, we need them to discard non-optimal paths



# Transfer patterns precomputation 4/4

- Arrival-loop algorithm for a target station B
  - Order the arrival nodes by time $t_1 \leq t_2 \leq t_3 \leq \ldots$ and call the corresponding arrival nodes $a_1, a_2, a_3, \ldots$
  - Do the following in the order of increasing time
  - Let $T_{i-1}$ and $T_i$ be the travel time of the shortest path to ai-1 and ai, respectively
  - If $T_i' := T_{i-1} + (t_i - t_{i-1}) \leq T_i$, replace the travel time at $a_i$ by $T_i'$ and make $a_{i-1}$ the predecessor on the SP to $a_i$



# Important Stations 1/3

- The pre-computation so far is quadratic
  - Full Dijkstra to the whole graph for every station
  - Let m = #stations and n = #nodes
  - This amounts to a total of ~ m · n · L Dijkstra iterations

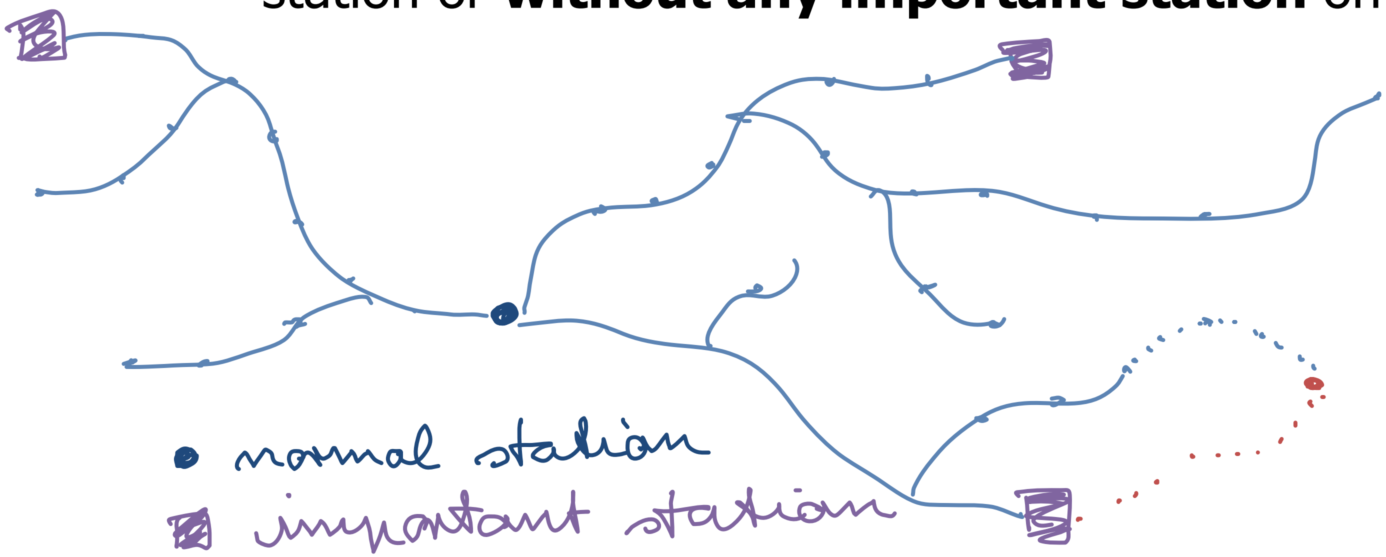
    where L is the average number of labels per node
  - A multi-label Dijkstra is ≈ 10 times slower per iteration than an ordinary Dijkstra (due to label set maintenance)
  - Example 1: m = 10K, n = 1M, L = 3, 10 μs / Dijkstra iter.

    30K seconds ≈ **80 hours**
  - Example 2: m = 1M, n = 1G, L = 3, 10 μs / Dijkstra iter.

    3G seconds ≈ **8 million hours ≈ 1000 years**

# Important Stations 2/3

- How to improve on this?
  - Idea: Select 1% of all stations as "important"
  - Heuristic: where many paths transfer + geographic diversity
  - For each important station compute a global Dijkstra as before
  - For each non-important station, compute a local Dijkstra, that is, compute all **local paths** = all paths **until an important** station or **without any important station** on them



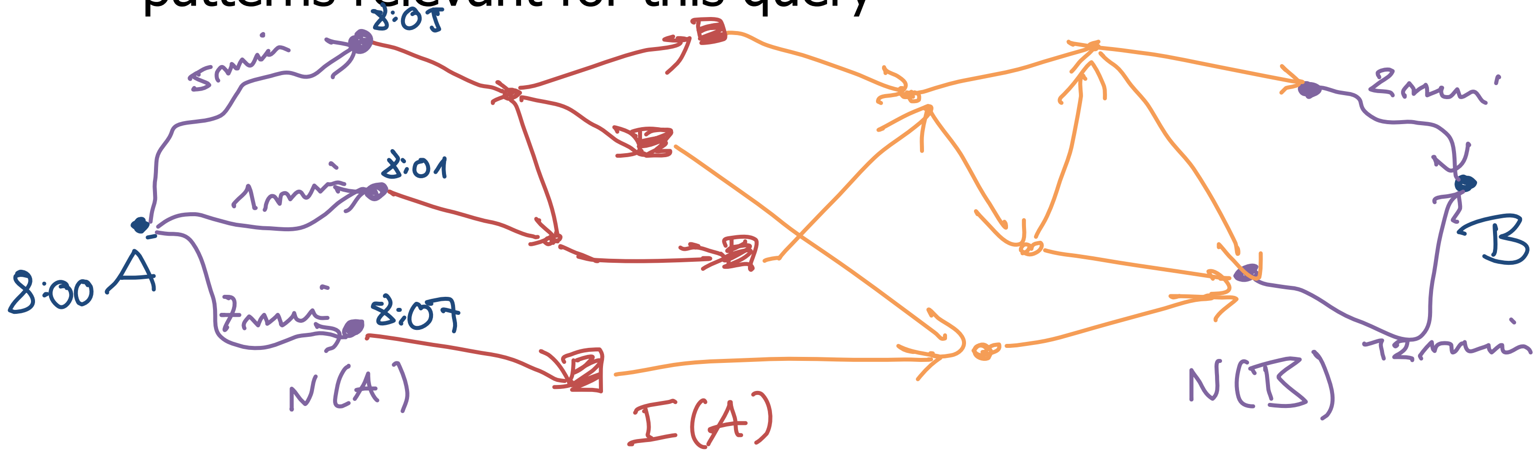
# Important Stations 3/3

- Local Dijkstra search from a station s ... problem:
  - The number of (nodes on the) local paths is indeed small
  - But we have the usual "15 hours to the next village problem":

    If only one of the local paths has a large cost, say 15 hours, then the Dijkstra computation needs to search everything that can be reached from s within 15 hours
  - Unfortunately, almost every station has at least one local path of high cost, and hence our local Dijkstra searches end up being no less expensive than the global Dijkstra searches
  - Simple heuristic remedy: only consider local paths **up to two transfers**, that is, paths where more than two transfers are needed to get to an important station will be lost
  - Experience shows that these are **very rare** in practice

# Query graph construction (sketch)

- For given source and target **location** A and B
  - Compute the sets N(A) and N(B) of stations near A and B
  - Get the precomp. local transfer patterns of these stations
  - Get the set I(A) of important stations, where the local paths from N(A) end
  - Get the global transfer patterns for each pair of stations (a, b) where a ∊ I(A) and b ∊ N(B)
  - Assemble this to form the query graph of all transfer patterns relevant for this query



# Query graph search

- Time-dependent Dijkstra search
  - Start at the source location
  - For arcs from the source location to nearby station launch road network query (or have these precomputed)

    Same for arcs to the target location
  - For arcs between stations, ask direct-connection table

# Implementation Advice 1/4

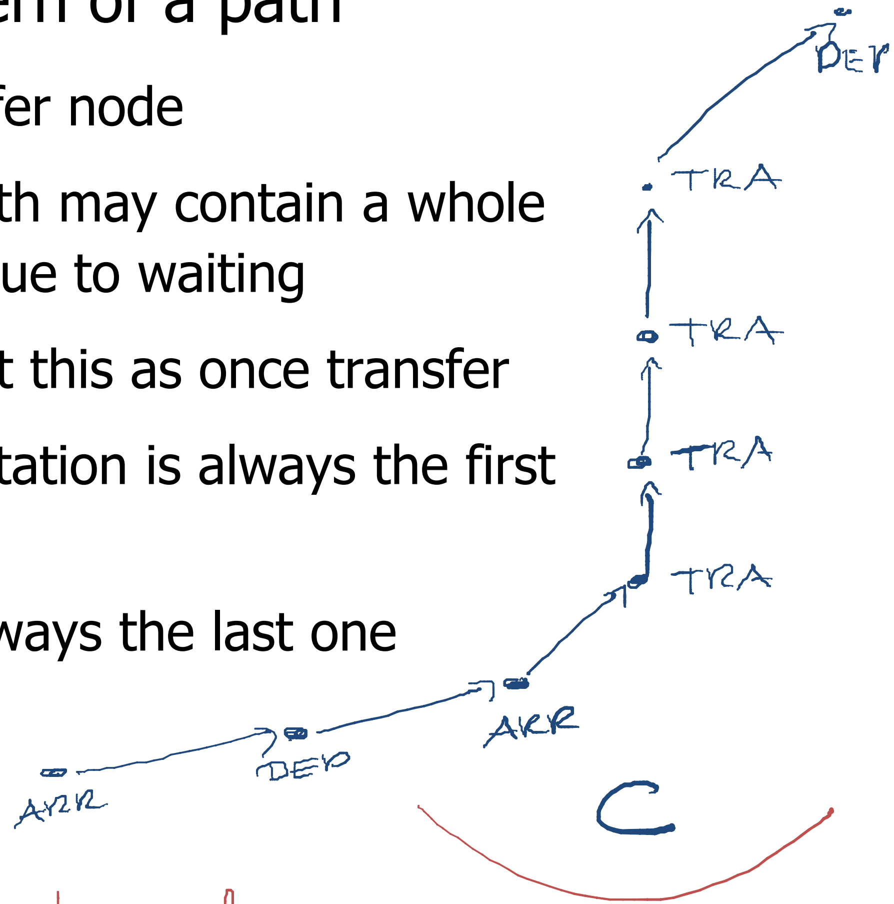
- Set Dijkstra
  - Just add an additional member sourceNodeSet
  - If non-empty, then in DijkstrasAlgorithm::computeShortestPath put all nodes from sourceNodeSet in the PQ with cost 0
  - And simply ignore the source node argument
- Arrival loop computation
  - For a given station, sort the nodes of that station by time
  - Then a single scan over the sorted sequence is enough

# Implementation Advice 2/4



- Computing the transfer pattern of a path
  - A transfer happens at a transfer node
  - However, at a transfer the path may contain a whole sequence of transfer nodes, due to waiting
  - Make sure that you only count this as once transfer
  - Don't forget that the source station is always the first station of a transfer pattern

    ... and the target station is always the last one

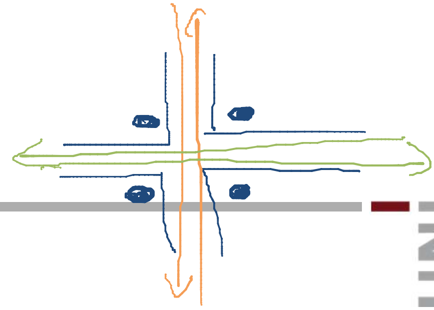



this is 1 transfer
and not four

# Implementation Advice 3/4

- Storing the transfer patterns for a station pair
  - For a (set) Dijkstra from a given source station, each node gives exactly one transfer patterns
    - Note: for single-criteria, we have **one label** per node
  - A transfer pattern can be stored as an Array<int>
    - that is, the sequence of station ids
  - **No need** to store the transfer patterns of all paths
  - Enough to remember which transfer patters occur at all
  - For a given source-target station pair, hence maintain the set of distinct transfer patterns in a Set<Array<int>>

# Implementation Advice 4/4



- Parsing Hawaii instead of Manhattan
  - You find the GTFS data for Hawaii on the Wiki
  - We checked that for Hawaii 80% of a set of random queries has a solution
  - Recall: for Manhattan it was 20% because of several station ids for bascially the same station
  - Beware: column order is not fixed in the GTFS standard, and different for Hawaii than for Manhattan
  - So your parser should consider the column headers, and not rely on a fixed position of the columns you need
  - You find an easy fix for this in the SVN, lectures/lecture-11

# References

- Transfer Patterns

  Fast Routing in Very Large Transportation Networks using Transfer Patterns

  Bast, Carlsson, Eigenwillig, Geisberger, Harrelson, Rachyev, Viger ESA 2010

  http://www.springerlink.com/content/c873271685124v42/