A Dynamic Multimodal Route Planner for Rome

Luca Allulli
Damiano Morosi
Roma Servizi per la Mobilità

ROMA mobilità

Public transport (PT) in Rome

Roma Servizi per la Mobilità: transport agency, in charge of

- Planning (transport network, timetable, PT contructions, etc.)
- Providing information to users (news, real-time info, etc.)
- Other services

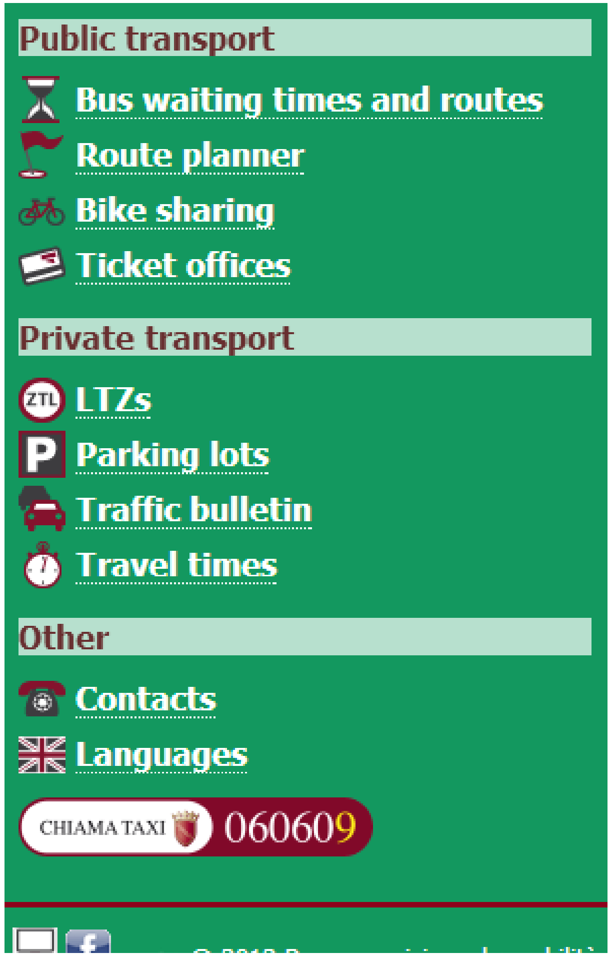
Atac: operator of PT lines:

- Most bus lines
- Tram, trolleybus
- Underground
- Urban railways
- Ticket

RomaTPL: operator of PT lines (privately-owned):

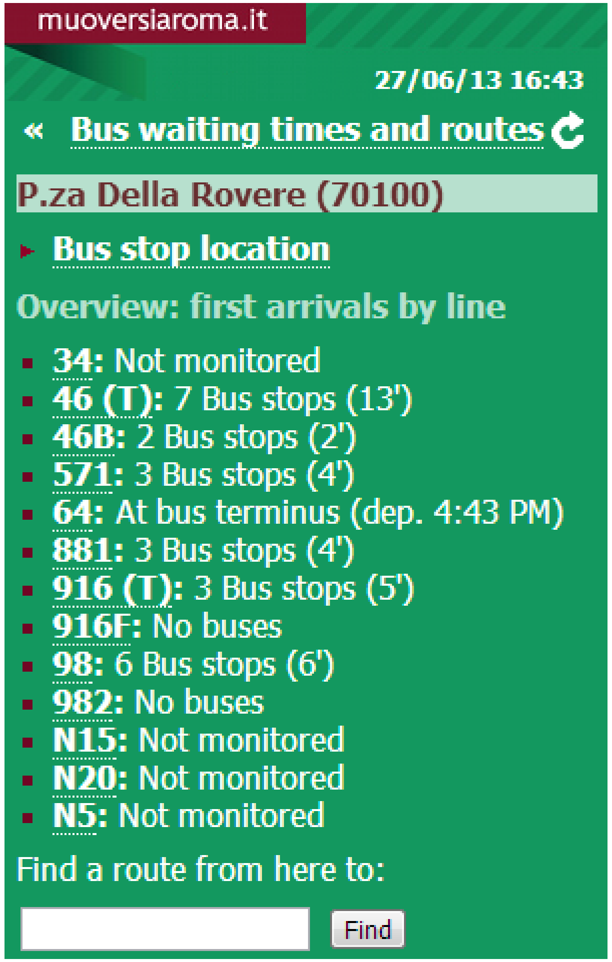
- Bus lines

Muoversi a Roma/1



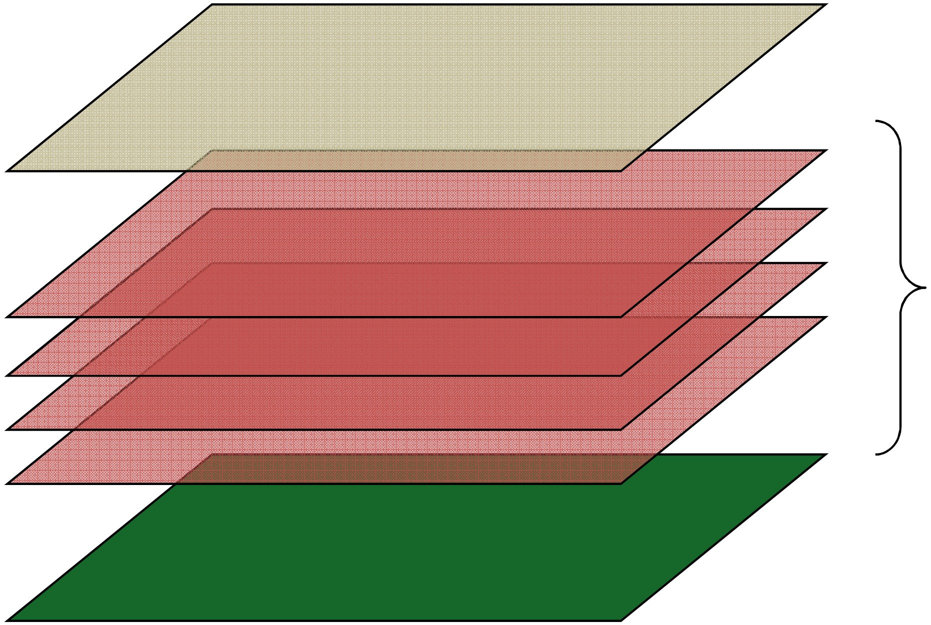
- **"Moving in Rome":** service operated by the Mobility Center of the Agency
- Website for **mobile phones**, since 2007
- **Real-time** information about public and private transport in Rome

Muoversi a Roma/2



- **Waiting times at bus stops**: our killer application
- **Real-time** data from GPS bus trackers
- "**This service would be** (almost) **useless in Germany**", where buses run on schedule
- Fact: PT in Rome is **different**
  - ...what about route planner?

Dijkstra's algorithm on a layered graph



Car pooling (experimental)

Public transport network

- Bus/tram/trolleybus
- Underground
- Urban railways
- Regional railways

Road network (walking and biking)

Every layer is connected to and from road network

Road network (walking and biking)

- Road graph:
  - Old: OpenStreetMap with CC-BY-SA license
    - But: license changed to ODbL: "If you publicly use [...] works produced from an adapted database, you must also offer that adapted database under the ODbL"
  - New: TomTom MultiNet (Tele Atlas)
- Cost model for walking:

  - Old: walking time
    - But: walking is often competitive wrt PT

  - New: unit cost increases as the user gets "tired“ (How?)

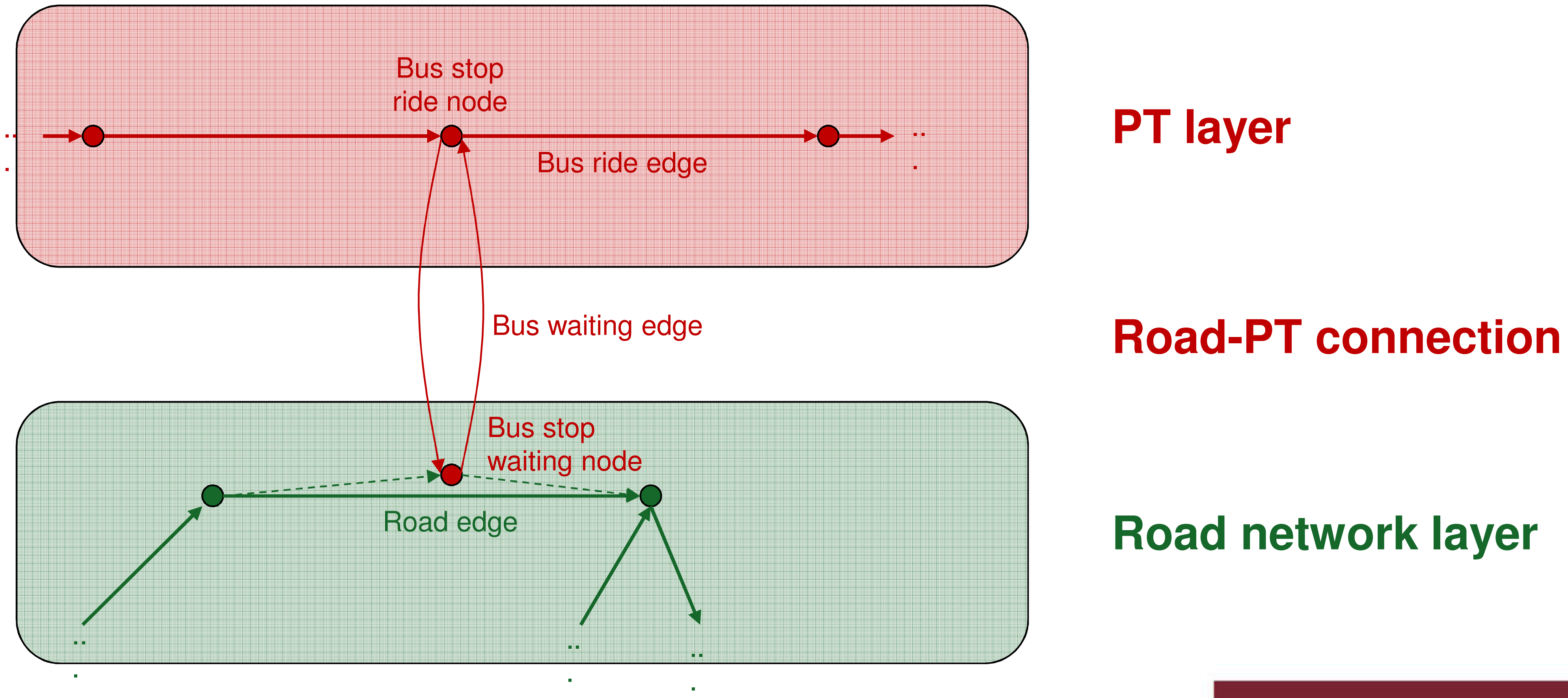
- Cost model for biking: biking time + number of turns; user-defined maximum biking distance

Node context

- Each visited node n has a **context** c_n
- c_n: dictionary containing additional information about the shortest path up to node n; such as:
  - **walking distance** (so far)
  - **biking distance** (so far)
  - modal switches (bike left), etc.
- When n is visited, context is «propagated» and updated from its predecessor pred(n). Let e = (pred(n), n):
  - ***e*.update_context(options)**

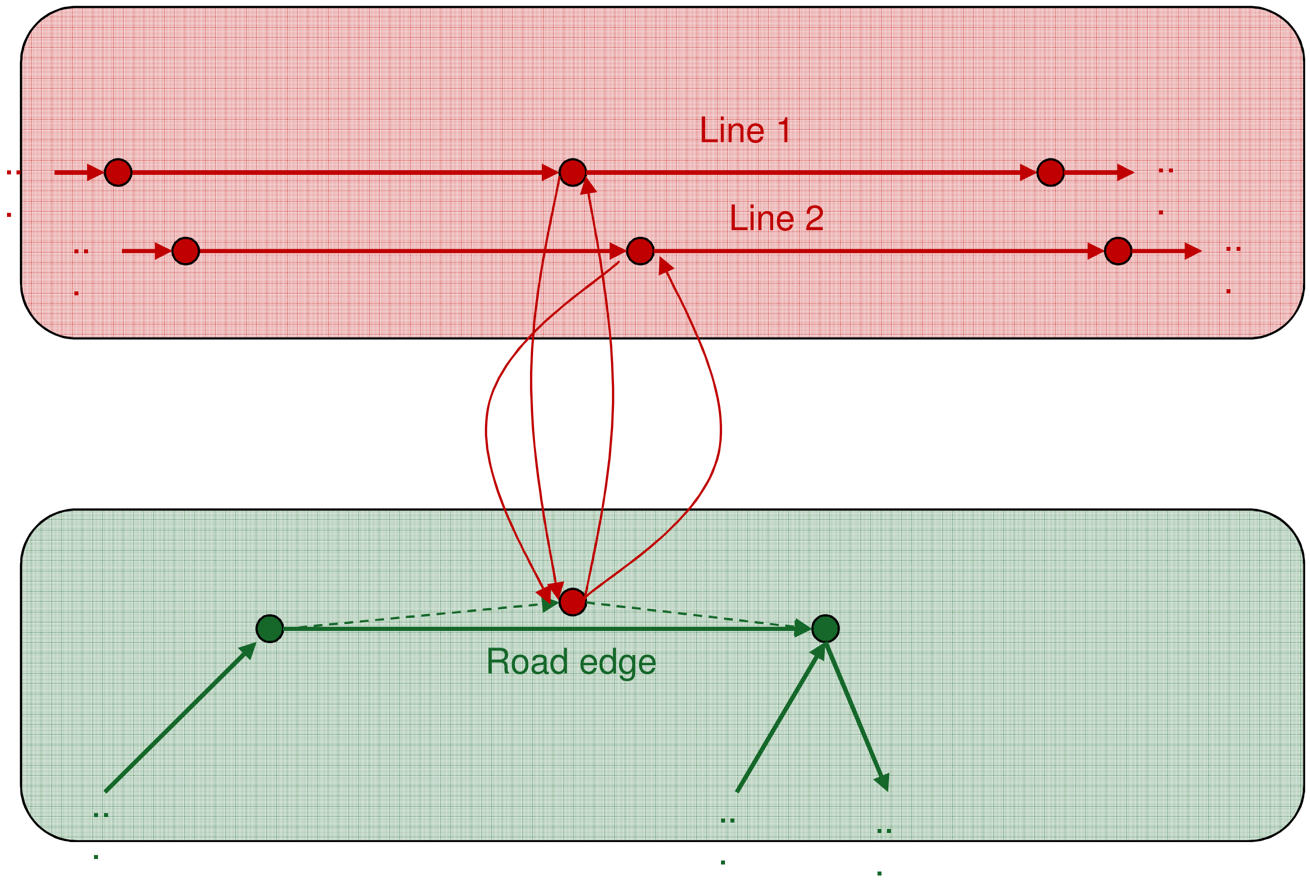
Public Transport - Road connection

Time-dependant model



Public Transport – Several lines

Time-dependant model



PT layer

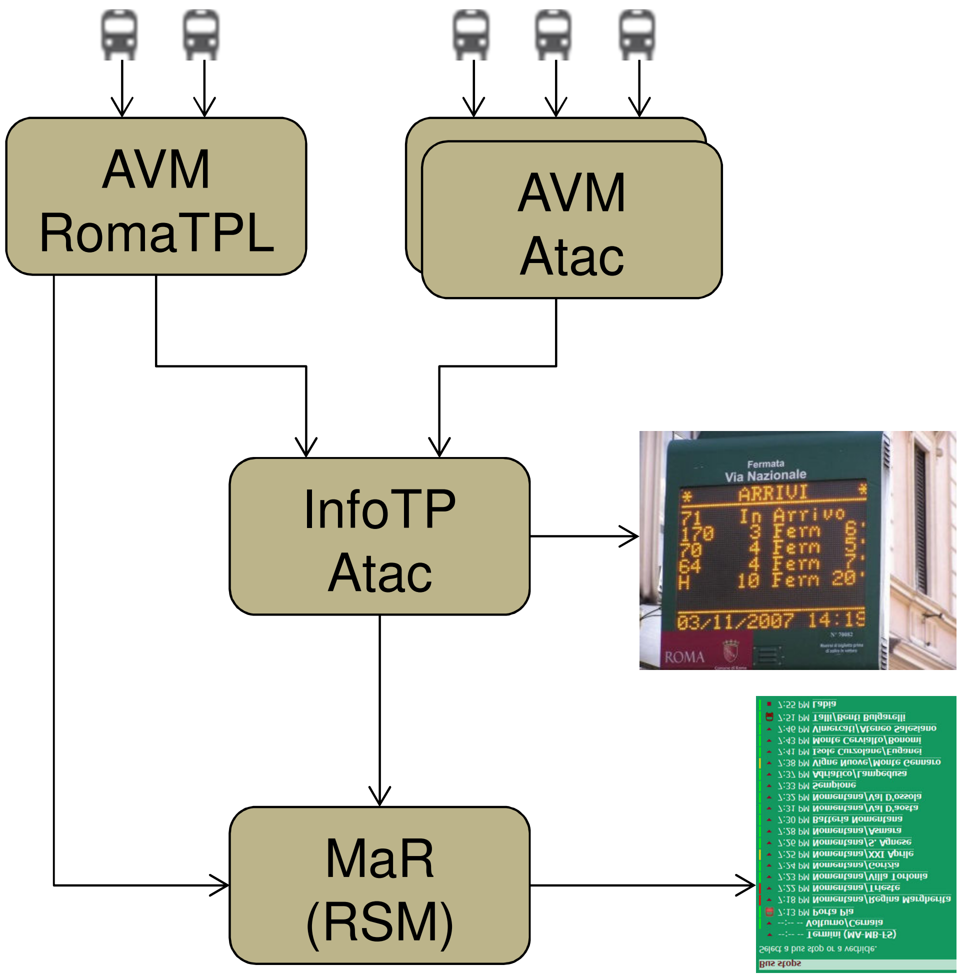
Road-PT connection

Road network layer

Public Transport - Dynamic costs

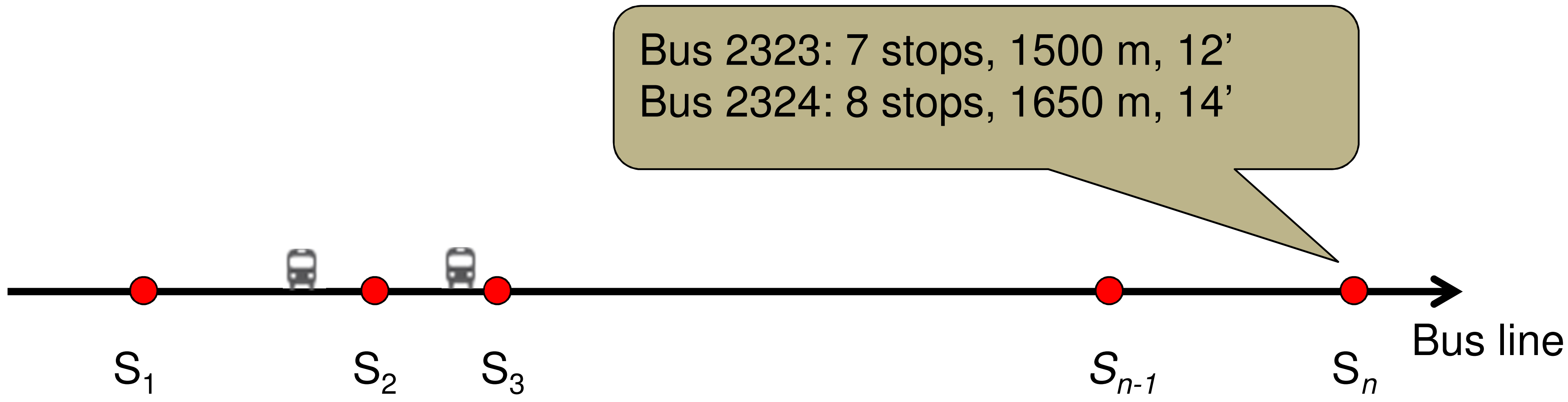
- Each edge object has a method: **get_cost(time, options)**
  - **time**: arrival time at source node (current tentative "distance" by Dijkstra's algo)
  - **options**: parameters for the route planner (e.g., walking speed)
- Cost for **bus waiting** edges:
  - Waiting time for catching first arriving bus, if real-time data available
  - Average waiting time from historic data or schedule, otherwise
- Cost for **bus ride** edges:
  - Use traffic speed, if real-time data
  - Use historic speed, otherwise

Bus trackers in Rome (AVM)



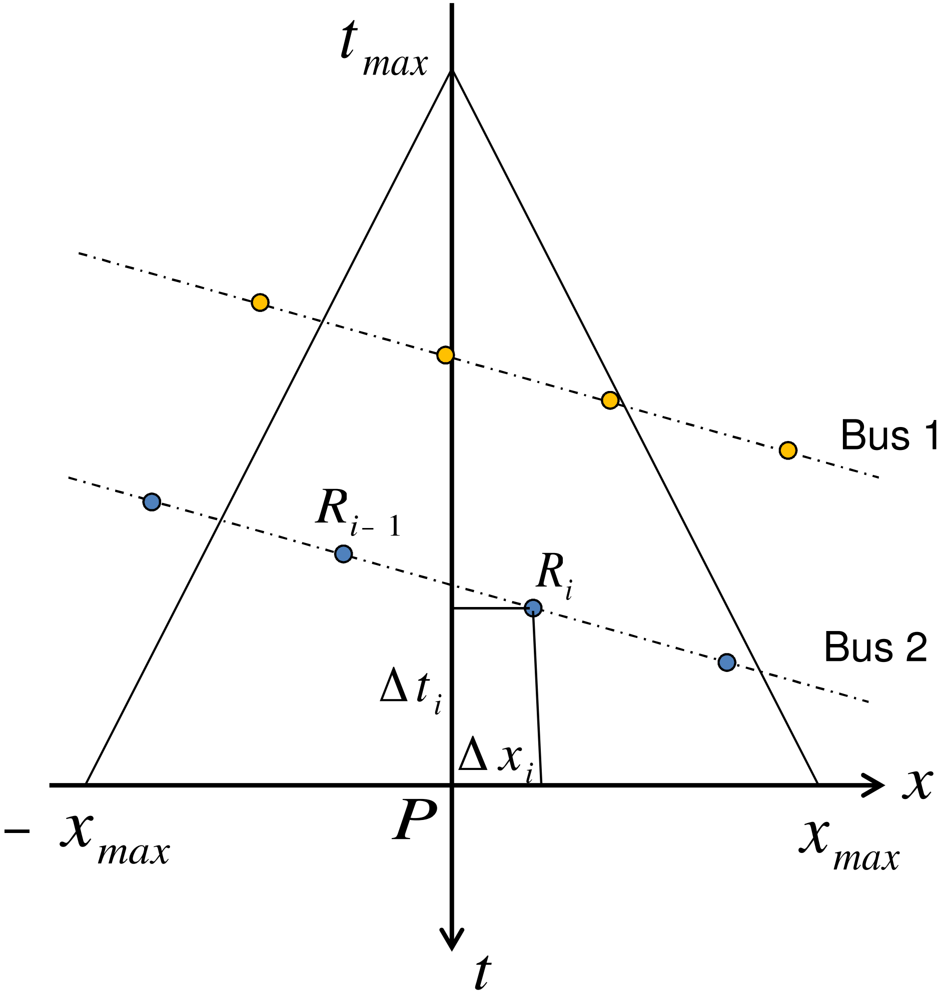
- Originally installed to monitor operators
  - Atac: agency, owner of the system
  - Trambus, Roma TPL: operators
- Later, extended to provide waiting times at phisical bus stops
  - Black box: only one method: get_arrivals(stop_id)
- Now:
  - RSM is the new agency
  - Atac is an operator, but still owns the system
  - Roma TPL sends data to Atac

Querying last stop to determine bus positions



What’s next:
Determine «average speed» of each edge
in order to forecast ride duration
and arrivals at bus stops

From AVM samples to edge speed



$$\bar{v} = \frac{\sum_{r \in R} w_i v_i}{\sum_{r \in R} w_i}$$

$$v_i = \frac{x_i - x_{i-1}}{t_i - t_{i-1}}$$

$$w_i = 1 - \frac{x_i}{x_{max}} - \frac{t_i}{t_{max}}$$

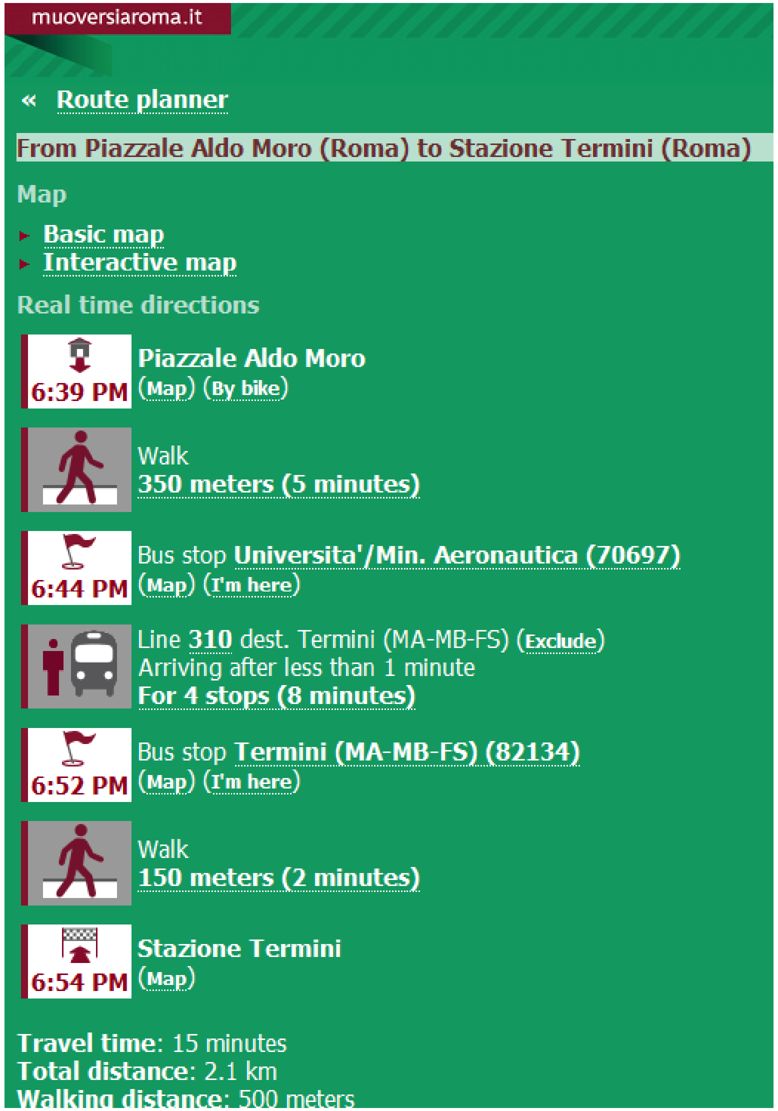
Interesting byproducts…

- Development of an internal **real-time view of PT state**
  - Recompute waiting times (better quality predictions than InfoTP) and
  - Give them in other forms (such as, schedule-like form)
  - Provide traffic information
  - Collect historical data, compute statistics
- Now RomaTPL sends (high quality) GPS data directly to the Agency

Public Transport – Several cost models

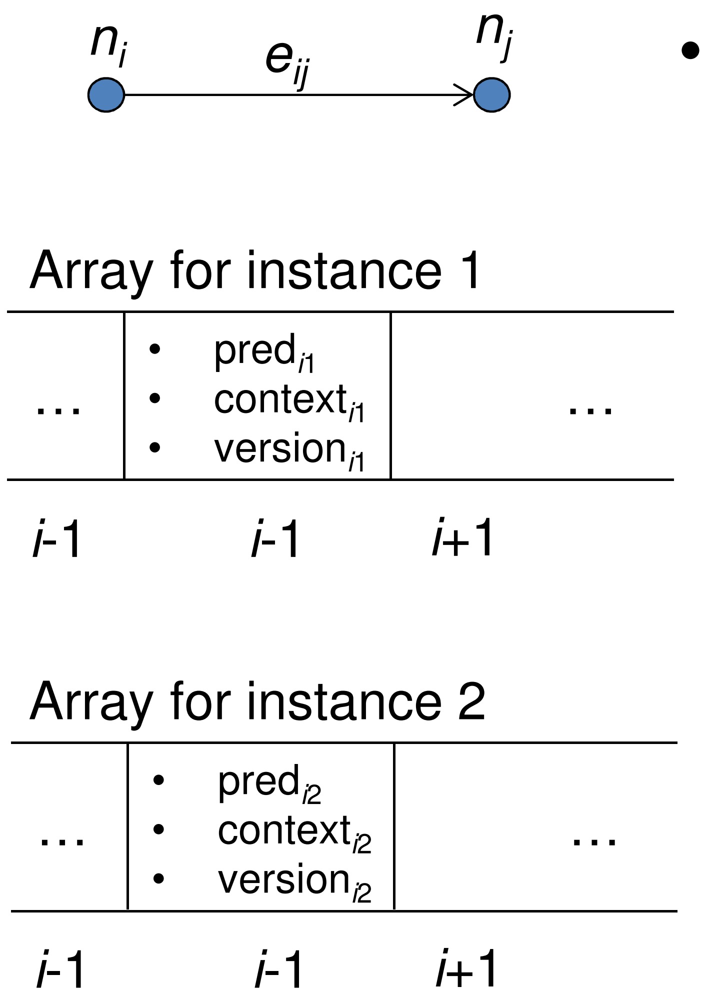
- **Bus, tram, trolleybus**
  - Data from bus trackers
  - Statistics (in each time band)
  - Frequency from schedule
- **Underground, urban railways**
  - Frequency from schedule
  - Journey time from schedule/heuristics
- **Regional railways**
  - "Classic" schedule
- **Cost != time**
  - Penalization for each modal switch
  - Smaller cost if user gets on bus at bus terminus
  - Walking: increasing cost factor when user is tired

From a Prototype to a Service



- Very small development team
  - In charge of several projects
- Solution: incremental approach
  - Working prototype
  - High-level programming language (Python)
  - Refactor often, never throw away
  - Profile and Optimize: core of Dijkstra's algorithm in Cython
    - Python partially compiled in C
    - Priority queue completely compiled
    - Main loop partially compiled
    - Cost functions: Pure Python

Dijkstra’s implementation: tips and tricks/1



- Separate graph representation from Dijkstra’s data structures
  - Each node has an index *i*
  - Keep variables for Dijkstra’s algorithm (tentative distance, predecessor etc.) in an array
  - Several instances of Dijkstra’s algorithm running in parallel with small memory overhead
  - “Emulate” several, connected copies of the same graph in a single computation (see later)

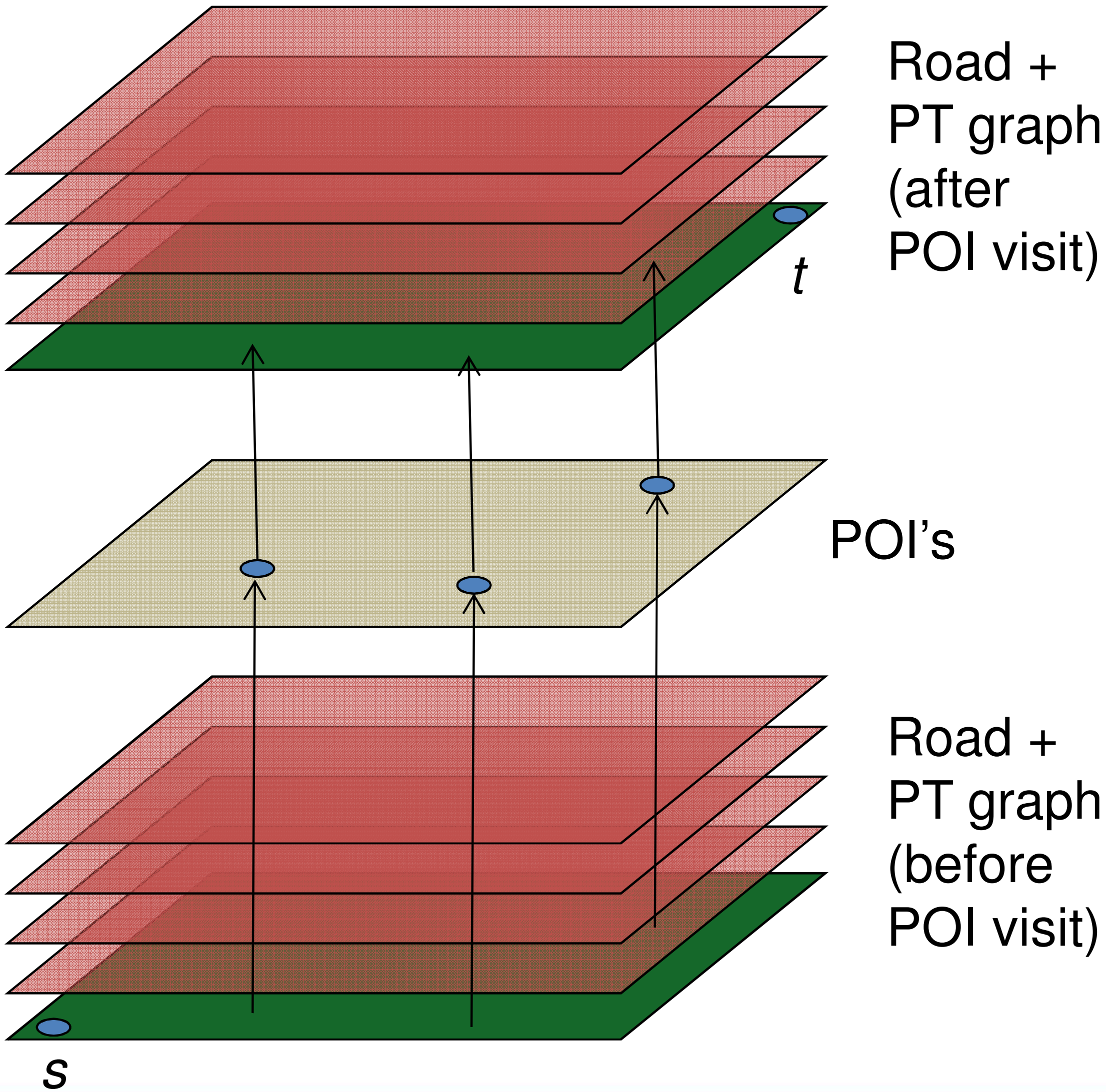
Dijkstra’s implementation: tips and tricks/2

- Don’t reset variables attached to nodes
  - add an extra variable to each node *n*: version[*n*]
  - version: global counter of Dijkstra’s computations
  - when a node *n* is reached for the first time during a computation, version[*n*] < version

Car Pooling

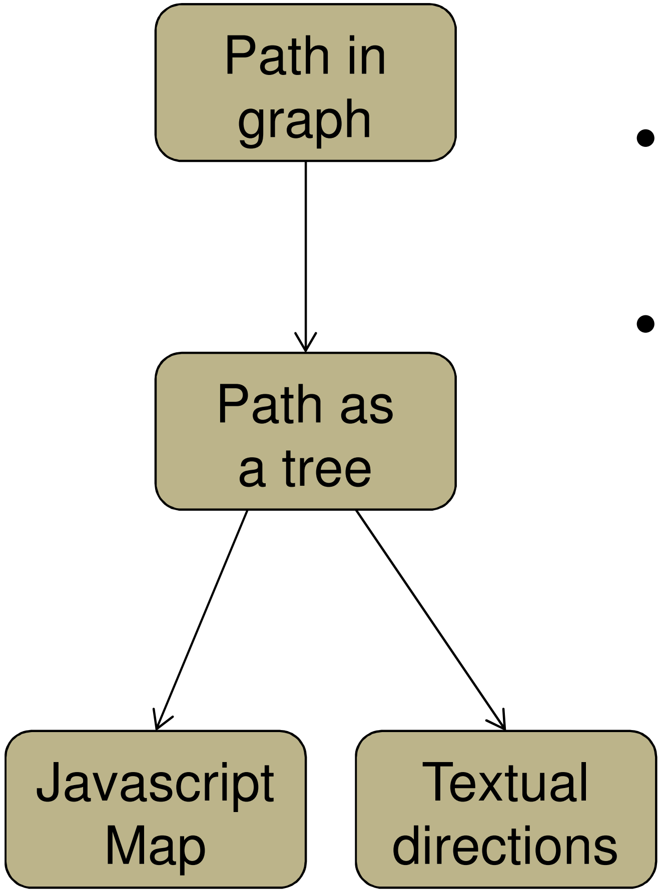
- Idea: when a user offers a ride, his path is inserted in the graph, in the carpooling layer
  - Path is computed through an (adjustable) private transport route plan
- When a user looks for a ride, he performs a route planner query. Route planner uses all the graph layers: walking/biking, car pooling and (optionally) public transport (intermodal car pooling)

s-t path through a POI



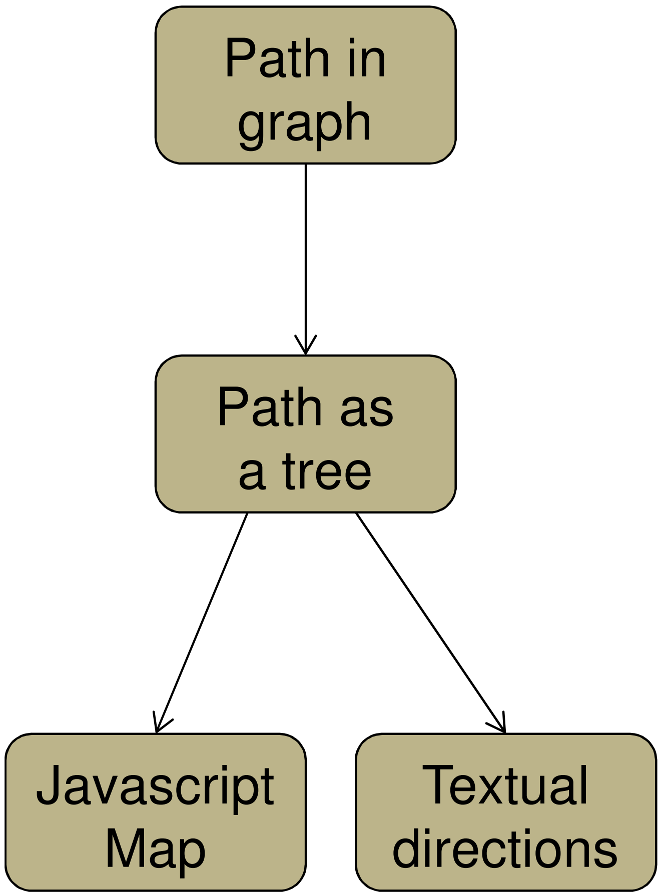
- «I want to buy a CD on the way home: find the most convenient music store»
- Instead of building 2 copies of the graph, use two PQ's (and 2 sets of Dijkstra's variables) to keep track of

Building output/1



- Several kinds of output: textual directions, javascript map, etc.
- Build an abstract tree representation of the path:
- RootNode
  - WalkingNode 1
    - WalkingEdgeNode 1
    - WalkingEdgeNode 2
  - BusNode 1
    - BusWaitingNode
    - BusRideNode 1
    - BusRideNode 2
    - …

Building output/2



- From graph to path tree
  - Traverse s-t path. Each node and each edge provides a method: build_path(tree_node, path_options) -> tree_node
  - Start from RootNode.
- From path tree to final output
  - Register «formatters» for each type of tree node (BusNode, WalkingNode etc.) and kind of output
  - Perform a DFS of the tree, invoking appropriate formatters