Θέματα Αλγορίθμων

Αλγόριθμοι και Εφαρμογές στον Πραγματικό Κόσμο

Μεταπτυχιακό Μάθημα

4η Εβδομάδα: Βέλτιστες Διαδρομές & σε Χρονοεξαρτώμενα Δίκτυα

Σπύρος Κοντογιάννης

kontog@cse.uoi.gr

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

Τετάρτη, 15-22 Μαρτίου 2017

Shortest Paths

... a fundamental problem in Computer Science

- **INPUT:**
  - Directed graph $G = (V, E)$.
  - Arc costs (distance, travel-time, fuel consumption, etc.): $\forall uv \in E,\ c[uv] \geq 0$.
  - Origin-destination pair: $(o, d) \in V \times V$.
  - $P_{o,d}$: Set of od-paths in G.
  - Additive path costs: $c[p] = \sum_{e \in p} c[e]$.

- INPUT:
  - Directed graph $G = (V, E)$.
  - Arc costs (distance, travel-time, fuel consumption, etc.): $\forall uv \in E,\ c[uv] \geq 0$.
  - Origin-destination pair: $(o, d) \in V \times V$.
  - $P_{o,d}$: Set of od-paths in G.
  - Additive path costs: $c[p] = \sum_{e \in p} c[e]$.
- OUTPUT: $\pi^* \in \arg\max_{\pi \in P_{o,d}} \{ c[p] \}$

- **INPUT:**
  - Directed graph $G = (V, E)$.
  - Arc costs (distance, travel-time, fuel consumption, etc.): $\forall uv \in E,\ c[uv] \geq 0$.
  - Origin-destination pair: $(o, d) \in V \times V$.
  - $P_{o,d}$: Set of *od*-paths in G.
  - Additive path costs: $c[p] = \sum_{e \in p} c[e]$.
- **OUTPUT:** $\pi^* \in \arg\max_{\pi \in P_{o,d}} \{ c[p] \}$
- **GOAL:** Route planning in **road networks**.

Shortest Paths Problem

Statement

- **INPUT:**
  - Directed graph $G = (V, E)$.
  - Arc costs (distance, travel-time, fuel consumption, etc.): $\forall uv \in E,\ c[uv] \geq 0$.
  - Origin-destination pair: $(o, d) \in V \times V$.
  - $P_{o,d}$: Set of *od*-paths in G.
  - Additive path costs: $c[p] = \sum_{e \in p} c[e]$.

- **OUTPUT:** $\pi^* \in \arg\max_{\pi \in P_{o,d}} \{\ c[p]\ \}$

- **GOAL:** Route planning in **road networks**.
  - V is the set of road junctions.
  - E is the set of uninterrupted road segments.

Shortest Paths Problem

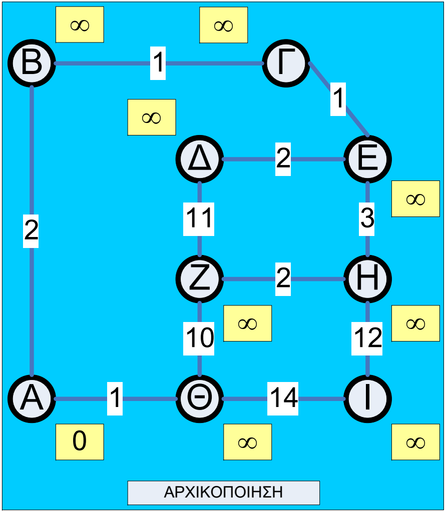
Statement

- **INPUT:**
  - Directed graph $G = (V, E)$.
  - Arc costs (distance, travel-time, fuel consumption, etc.): $\forall uv \in E,\ c[uv] \geq 0$.
  - Origin-destination pair: $(o, d) \in V \times V$.
  - $P_{o,d}$: Set of od-paths in G.
  - Additive path costs: $c[p] = \sum_{e \in p} c[e]$.
- **OUTPUT:** $\pi^* \in \arg\max_{\pi \in P_{o,d}} \{\ c[p]\ \}$
- **GOAL:** Route planning in **road networks**.
  - V is the set of road junctions.
  - E is the set of uninterrupted road segments.
    - ★ Sparse netwrok: $|E| \in \mathrm{O}(|V|)$.
    - ★ HUGE size: $|V| =$ tens of millions of nodes.

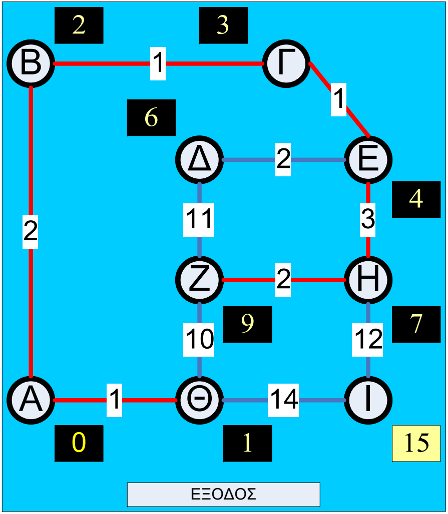
- **INPUT:**
  - Directed graph $G = (V, E)$.
  - Arc costs (distance, travel-time, fuel consumption, etc.): $\forall uv \in E,\ c[uv] \geq 0$.
  - Origin-destination pair: $(o, d) \in V \times V$.
  - $P_{o,d}$: Set of od-paths in G.
  - Additive path costs: $c[p] = \sum_{e \in p} c[e]$.

- **OUTPUT:** $\pi^* \in \arg\max_{\pi \in P_{o,d}} \{ c[p] \}$

- **GOAL:** Route planning in **road networks**.
  - V is the set of road junctions.
  - E is the set of uninterrupted road segments.
    - ★ Sparse netwrok: $|E| \in \mathrm{O}(|V|)$.
    - ★ HUGE size: $|V|$ = tens of millions of nodes.
  - Arc costs usually represent *travel-times*.



∞
∞
B
1
Γ
1
∞
Δ
2
E
∞
2
11
3
Z
2
H
10
∞
12
∞
A
1
Θ
14
I
0
∞
∞
ΑΡΧΙΚΟΠΟΙΗΣΗ



2
3
Β
1
Γ
1
6
Δ
2
Ε
4
2
11
3
Ζ
2
Η
9
7
10
12
Α
1
Θ
14
Ι
0
1
15
ΕΞΟΔΟΣ

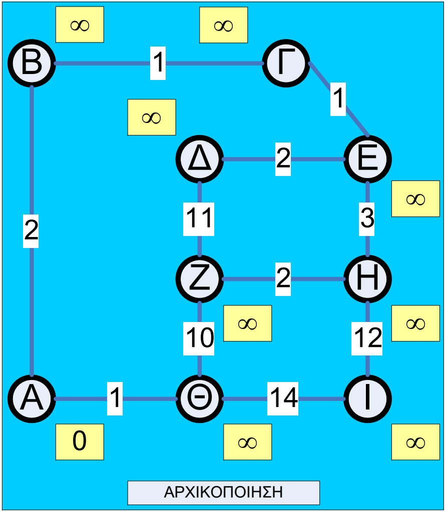
Dijkstra's Algorithm

Pseudocode

```
Dijkstra( G = (V,E), o ∈ V, d ∈ V, c : E → ℝ≥0 )
1.        for all v ∈ V do D[v] = ∞;
2.        D[o] = 0;
3.        Q.Insert(o, D[o]);                                   /* Q: priority queue */
4.        while !Q.IsEmpty() do
4.1.          v = Q.ExtractMin();                              /* v is the node with min tentative label */
4.2.          for all vw ∈ E(G) do                             /* scanning of node v */
4.2.1.            if D[w] > D[v] + c[vw]
4.2.2.            then                                         /* relaxation of arc vw */
4.2.2.1.              D[w] = D[v] + c[vw];
4.2.2.2.              if w ∈ Q then Q.DecreaseKey(w, D[w]);
4.2.2.3.              else Q.Insert(w, D[w]);
```

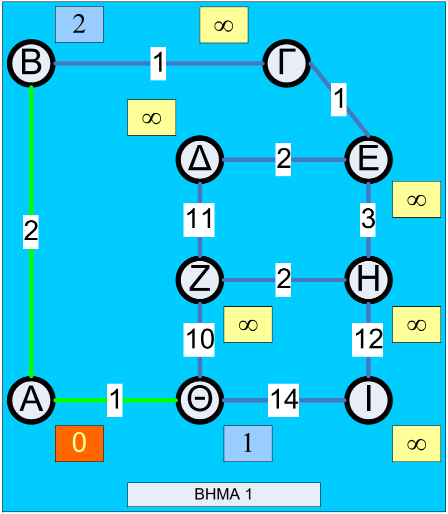
Dijkstra's Algorithm

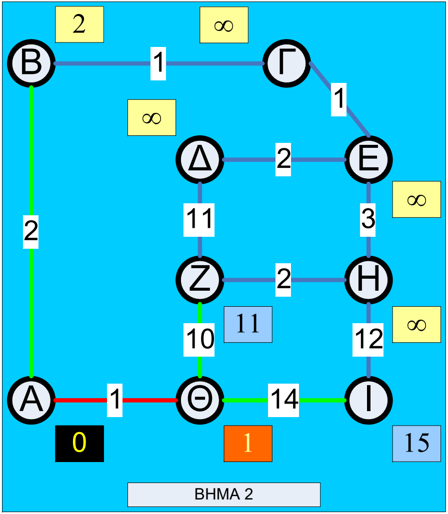
Execution Example



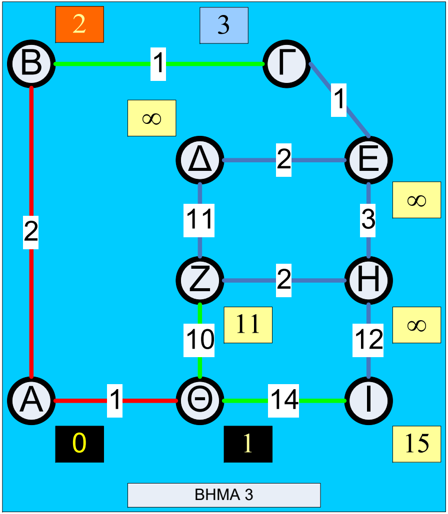
Dijkstra's Algorithm

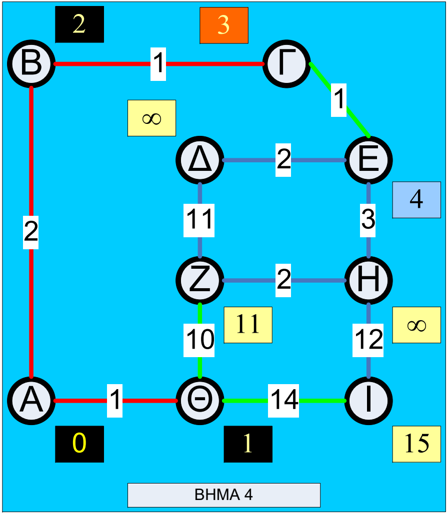
Execution Example

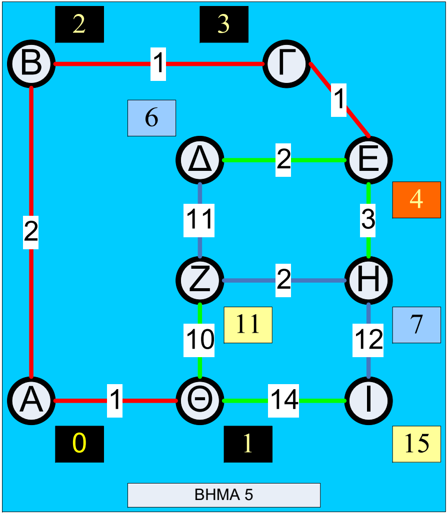




2
∞
Β
1
Γ
1
∞
Δ
2
Ε
∞
2
11
3
Ζ
2
Η
11
10
12
∞
Α
1
Θ
14
Ι
0
1
15
ΒΗΜΑ 2


2
3
B
1
Γ
1
∞
Δ
2
E
∞
2
11
3
Z
2
H
11
∞
10
12
A
1
Θ
14
I
0
1
15
BHMA 3

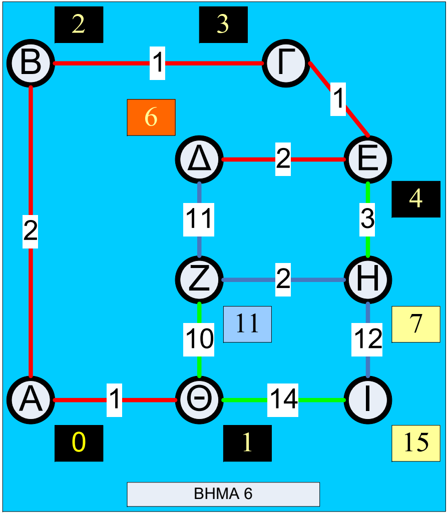

2
3
B
1
Γ
1
∞
Δ
2
E
4
11
3
Z
2
H
11
10
12
∞
A
1
Θ
14
I
0
1
15
BHMA 4

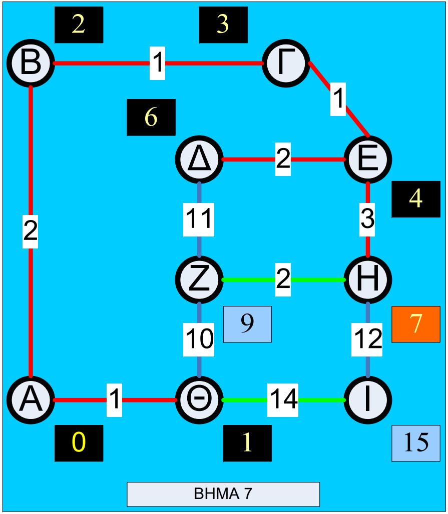


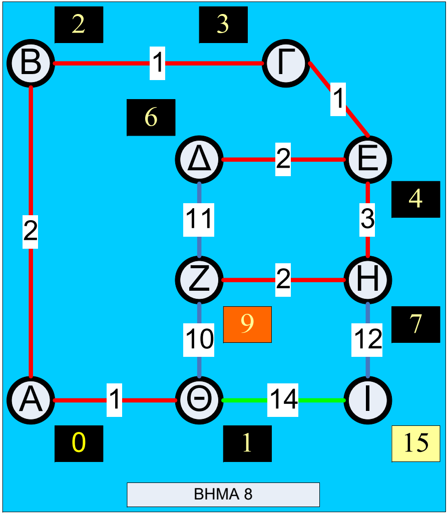
2
3
B
1
Γ
1
6
Δ
2
E
4
2
11
3
Z
2
H
11
10
12
7
A
1
Θ
14
I
0
1
15
BHMA 5

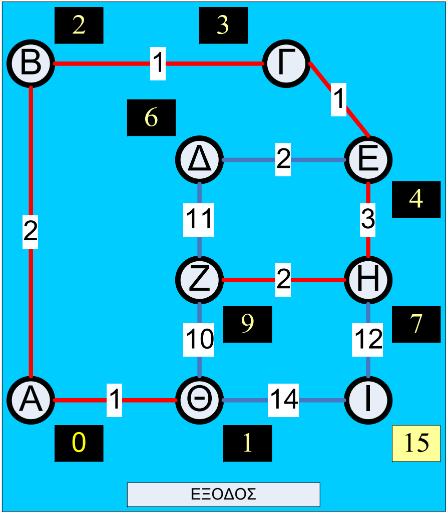
Dijkstra's Algorithm

Execution Example




2
3
B
1
Γ
1
6
Δ
2
E
4
2
11
3
Z
2
H
9
10
7
12
A
1
Θ
14
I
0
1
15
BHMA 7


2
3
B
1
Γ
1
6
Δ
2
E
4
2
11
3
Z
2
H
9
10
12
7
A
1
Θ
14
I
0
1
15
BHMA 8


2
3
Β
1
Γ
1
6
Δ
2
Ε
4
2
11
3
Ζ
2
Η
9
7
10
12
Α
1
Θ
14
Ι
0
1
15
ΕΞΟΔΟΣ

- Correctness
  - **Labels:** Represent **upper bounds** on total costs (travel-times) from the origin towards each destination.
  - In each round the node v with *minimum tentative label* $D[v]$ is chosen for **finalization** of its label (not tentative anymore).
  - Non-negative arc-costs $\Rightarrow$ The label $D[v]$ to be finalized in each round is the exact min-cost from o to v (cannot be further improved).

Dijkstra's Algorithm

Analysis

- Correctness
  - **Labels:** Represent **upper bounds** on total costs (travel-times) from the origin towards each destination.
  - In each round the node v with *minimum tentative label* $D[v]$ is chosen for **finalization** of its label (not tentative anymore).
  - Non-negative arc-costs $\Rightarrow$ The label $D[v]$ to be finalized in each round is the exact min-cost from o to v (cannot be further improved).
- Time Complexity /* depends on the choice of the priority queue */
  - $O(n)$ queue-insertion operations.
  - $O(n)$ queue-extract-minimum operations.
  - $O(m)$ queue-label correction operations (upon arc relaxations).

- Implementation of the priority queue with **Fibonacci Heaps**
  - $O(\log(n))$ *elementary operations* per extract-minimum operation.
  - $O(1)$ *elementary operations* per queue-insertion / queue-label correction operation.

$\therefore$ $O(m + n\log(n))$ *elementary operations* in total.

- Implementation of the priority queue with **Fibonacci Heaps**
  - $O(\log(n))$ *elementary operations* per extract-minimum operation.
  - $O(1)$ *elementary operations* per queue-insertion / queue-label correction operation.

  $\therefore$ $O(m + n\log(n))$ *elementary operations* in total.
- Implementation of priority queue with **Binary Heaps**
  - $O(\log(n))$ *elementary operations* per extract-minimum / insertion / label-correction operation of the queue.

  ☹ $O(m\log(n))$ *elementary operations* in total.

  ☺ Extremely simpler data structure than Fibonacci Heaps.

  ☺ Usually **faster in practice** (for large-scale, real-world instances).

  ☺ For **road netwroks**, $m \in O(n)$.

- Implementation of the priority queue with **Fibonacci Heaps**
  - $O(\log(n))$ *elementary operations* per extract-minimum operation.
  - $O(1)$ *elementary operations* per queue-insertion / queue-label correction operation.

  $\therefore$ $O(m + n\log(n))$ *elementary operations* in total.

- Implementation of priority queue with **Binary Heaps**
  - $O(\log(n))$ *elementary operations* per extract-minimum / insertion / label-correction operation of the queue.

  ☹ $O(m\log(n))$ *elementary operations* in total.

  ☺ Extremely simpler data structure than Fibonacci Heaps.

  ☺ Usually **faster in practice** (for large-scale, real-world instances).

  ☺ For **road netwroks**, $m \in O(n)$.

- Implementation of priority queue with **k-ary Heaps**
  - Each internal node has k children.
  - Fewer tree levels (than binary / fibonacci heaps), more nodes per level.

  ☺ Better exploitation of data locality.

  - Same time-complexity with Binary Heaps.

- Execution of Dijkstra for **Europe's road netwrok**, with respect to *arc-travel-times* metric:

Data Structure	Response to Queries (sec)
2-heap	12.38
4-heap	11.53
8-heap	11.52

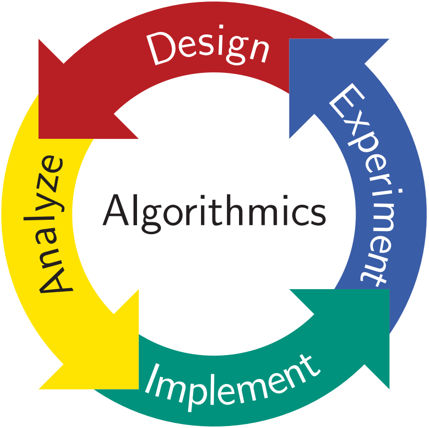
- Execution times on a *2.4GHz* AMD Opteron, with *16GB* RAM

(Microsoft Data Structures and Algorithms School (MIDAS), St. Petersburg (2010))

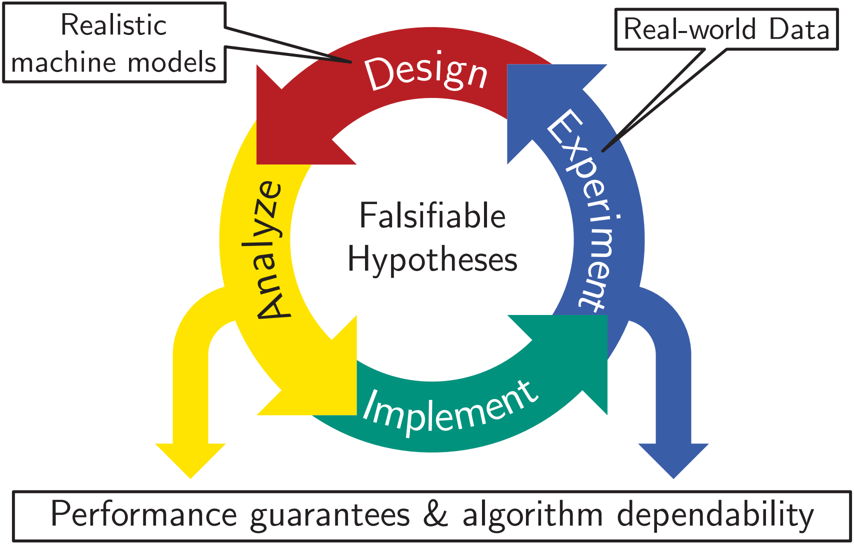
- Query times are for construction of a complete shortest-paths tree (SPT) from the origin towards **all reachable destinations**.

- Roughly half time for responding to **random** (o, d)-queries and *interrupting* Dijkstra upon scanning the destination vertex d.

Why Algorithm Engineering?



Why Algorithm Engineering?



Challenge of Scale

... shortest paths in large-scale road networks

Algorithm Engineering

Shortest paths in road networks: A successful showcase (mostly) in static graphs...

Continent-sized road networks: Millions of intersections

Continent-sized road networks: Millions of intersections

- `Dijkstra`: Responds within **a few seconds**.

Continent-sized road networks: Millions of intersections

- `Dijkstra`: Responds within **a few seconds**.
- **Speedup Techniques:** Shortest-Path *heuristics*, tailored especially for road networks.
  - Respond in **less than a millisecond** (or even **a few microseconds**).

Algorithm Engineering

Shortest paths in road networks: A successful showcase (mostly) in static graphs...

Continent-sized road networks: Millions of intersections

- `Dijkstra`: Responds within **a few seconds**.
- **Speedup Techniques:** Shortest-Path *heuristics*, tailored especially for road networks.
  - Respond in **less than a millisecond** (or even **a few microseconds**).

Most Popular Speedup Techniques

- Arc Flags (Lauther (2004), Köhler et al. (2006), Bauer & Delling (2008))
- A^* with Landmarks (Goldberg & Harrelson (2005))
- Reach (Gutman (2004), Goldberg et al. (2006))
- Highway Hierarchies (Sanders & Schultes (2005))
- Contraction Hierarchies (Geisberger et al. (2008))
- Transit Node Routing (Bast et al. (2006))

Distance Oracles

Another success story in static graphs...

Distance Oracles: Create (**offline**) data structures that require *reasonable space* requirements and allow answering in **real-time** to arbitrary queries *efficiently*, with provable approximation guarantees (stretch).

Distance Oracles

Another success story in static graphs...

Distance Oracles: Create (**offline**) data structures that require *reasonable space* requirements and allow answering in **real-time** to arbitrary queries *efficiently*, with provable approximation guarantees (stretch).

- Trivial solution (I): Preprocess by executing and storing APSP.
  - ☹ $O(n^2)$ size.
  - ☺ $O(1)$ query time.
  - ☺ $1-$stretch.

Distance Oracles

Another success story in static graphs...

Distance Oracles: Create (**offline**) data structures that require *reasonable space* requirements and allow answering in **real-time** to arbitrary queries *efficiently*, with provable approximation guarantees (stretch).

- Trivial solution (I): Preprocess by executing and storing APSP.
  - $O(n^2)$ size.
  - $O(1)$ query time.
  - $1-$stretch.

- Trivial solution (II): No preprocessing, respond to queries by running `Dijkstra`.
  - $O(n+m)$ size.
  - $O(m+n\log(n))$ query time.
  - $1-$stretch.

Distance Oracles

Another success story in static graphs...

Distance Oracles: Create (**offline**) data structures that require *reasonable space* requirements and allow answering in **real-time** to arbitrary queries *efficiently*, with provable approximation guarantees (stretch).

- Trivial solution (I): Preprocess by executing and storing APSP.
  - $O(n^2)$ size.
  - $O(1)$ query time.
  - $1-$stretch.

- Trivial solution (II): No preprocessing, respond to queries by running `Dijkstra`.
  - $O(n+m)$ size.
  - $O(m+n\log(n))$ query time.
  - $1-$stretch.

Provide smooth tradeoffs among space / query time / stretch!!!

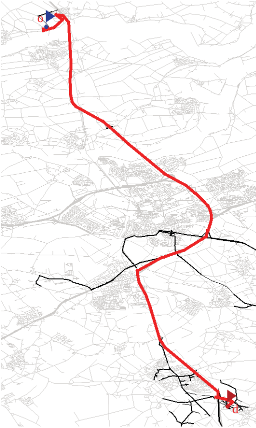
Distance Oracles

Theoretical bounds for static graphs...

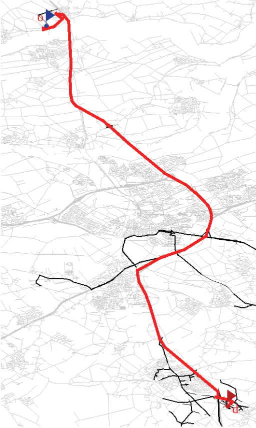
Reference	Setting	Stretch	Query	Space
(TZ05)	weighted graph	$2k-1, k \geq 2$	$O(k)$	$O\left(kn^{1+1/k}\right)$
(WN13)	weighted graph	$2k-1, k \geq 2$	$O(\log(k))$	$O\left(kn^{1+1/k}\right)$
(Che13)	weighted graph	$2k-1, k \geq 2$	$O(1)$	$O\left(kn^{1+1/k}\right)$
(AG13)	sparse weighted graph	$1+\epsilon$	$o(n)$	$o(n^2)$
(Kle02) (Tho04)	planar weighted digraph	$1+\epsilon$	$O(\epsilon^{-1})$	$O\left(\frac{n\log(n)}{\epsilon}\right)$
(MN06)	metric	$O(k)$	$O(1)$	$O\left(kn^{1+1/k}\right)$
(BGKRL11)	Doubling metric, dynamic	$1+\epsilon$	$O(1)$	$\epsilon^{-O(ddim)}n + 2^{O(ddim\log(ddim))}n$

- ☹ `Dijkstra` visits **all nodes** closer to o than d.

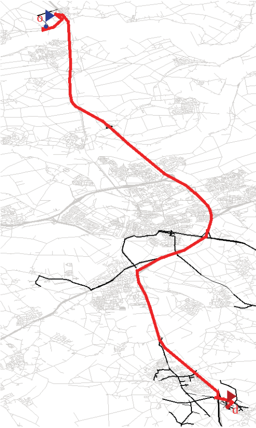
- Dijkstra visits **all nodes** closer to o than d.
- **Unnecessary** computations towards (eventually) irrelevant directions.



- Dijkstra visits **all nodes** closer to o than d.
- **Unnecessary** computations towards (eventually) irrelevant directions.
- Too many **shortest path requests** in networks that change very slowly (or, not at all) over time.



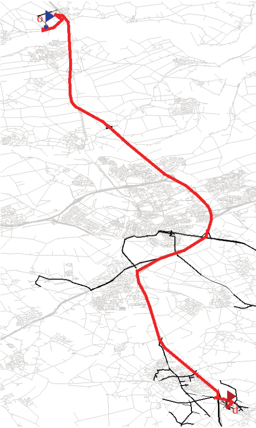
- Dijkstra visits **all nodes** closer to *o* than *d*.
- **Unnecessary** computations towards (eventually) irrelevant directions.
- Too many **shortest path requests** in networks that change very slowly (or, not at all) over time.
  - Exploit **preprocessing**: Compute offline selected *distance summaries* that will later allow, in *real-time*, responses to arbitrary shortest path requests.



Speedup Techniques / Distance Oracles

Goal...

- `Dijkstra` visits **all nodes** closer to *o* than *d*.
- **Unnecessary** computations towards (eventually) irrelevant directions.
- Too many **shortest path requests** in networks that change very slowly (or, not at all) over time.
  - Exploit **preprocessing**: Compute offline selected *distance summaries* that will later allow, in *real-time*, responses to arbitrary shortest path requests.



- *Assessment Criteria* of Speedup Techniques / Distance Oracles:
  - Preprocessing time / space.
  - Query (response) time to arbitrary requests.
  - Stretch (approximation guarantee).

Speedup Techniques / Distance Oracles

Generic idea...

1. Metric-independent preprocessing: Pick a **small** subset of **crucial** vertices in the graph, possibly ignoring the metric. E.g.:

Generic idea...

1. **Metric-independent preprocessing:** Pick a **small** subset of **crucial** vertices in the graph, possibly ignoring the metric. E.g.:
   - Consider (small) sets of *boundary vertices* in a partition of the graph into roughly equal-sized cells.
   - Randomly select *landmark* vertices.
   - Consider nearest *access points* (hubs) per vertex.
   - ...

Speedup Techniques / Distance Oracles

Generic idea...

1. **Metric-independent preprocessing:** Pick a **small** subset of **crucial** vertices in the graph, possibly ignoring the metric. E.g.:
   - Consider (small) sets of *boundary vertices* in a partition of the graph into roughly equal-sized cells.
   - Randomly select *landmark* vertices.
   - Consider nearest *access points* (hubs) per vertex.
   - ...
2. **Metric-dependent preprocessing:** Equip the network with selective **distance summaries**, e.g., boundary-to-boundary, hub-to-cell, landmark-to-all distances, etc.

Speedup Techniques / Distance Oracles

Generic idea...

1. **Metric-independent preprocessing:** Pick a **small** subset of **crucial** vertices in the graph, possibly ignoring the metric. E.g.:
   - Consider (small) sets of *boundary vertices* in a partition of the graph into roughly equal-sized cells.
   - Randomly select *landmark* vertices.
   - Consider nearest *access points* (hubs) per vertex.
   - ...
2. **Metric-dependent preprocessing:** Equip the network with selective **distance summaries**, e.g., boundary-to-boundary, hub-to-cell, landmark-to-all distances, etc.
3. **Query Algorithm:** Respond fast to queries, based on the (possibly metric-independent) preprocessing and/or the precomputed metric-dependent distance summaries.

- Extremely successful theme in static graphs.
  - In theory (oracles):
    - PRE-Space: Subquadratic (sometimes quasi-linear).
    - QUE-Time: Constant / sublinear in graph size.
    - Stretch: Small (sometimes PTAS).
  - In practice (speedups):
    - PRE-Space: A few GBs (sometimes less than 1 GB).
    - QUE-Time: Milliseconds (sometimes microseconds).
    - Stretch: Exact distances (in most cases).

Time Dependent

Shortest Path

... a more realistic but also more involved problem

Why Time-Dependent Shortest Paths?

Real-life networks: Elements demostrate temporal behavior.

Why Time-Dependent Shortest Paths?

Real-life networks: Elements demostrate temporal behavior.

- Graph elements **added/removed** in real-time. /* Dynamic Shortest Path */
- Metric demonstrates **stochastic behavior**. /* Sthochastic Shortest Path */
- Graph is **fixed**, metric **changes with the value of a parameter** $\gamma \in [0, 1]$ in a predetermined fashion. /* Parametric Shortest Path */
- Graph is **fixed**, metric **changes over time** in a predetermined fashion. /* Time-Dependent Shortest Path */

Why Time-Dependent Shortest Paths?

Real-life networks: Elements demostrate temporal behavior.

- Graph is **fixed**, metric **changes over time** in a predetermined fashion. /*

Time-Dependent Shortest Path */

Why Time-Dependent Shortest Paths?

Real-life networks: Elements demostrate temporal behavior.

- Graph is **fixed**, metric **changes over time** in a predetermined fashion. /*

  Time-Dependent Shortest Path */

  - Arcs are allowed to become **occasionally unavailable** (e.g., due to periodic maintenance, saving consumption of resources, etc), for predetermined unavailability time-intervals (discrete domain).
  - Arc lengths (e.g., traversal-time / consumption) **change with departure-time from tail** which is treated as a real-valued variable (functions with continuous domain, but not necessarily continuous range).

Why Time-Dependent Shortest Paths?

Real-life networks: Elements demostrate temporal behavior.

- Graph is fixed, metric changes with the value of a parameter $\gamma \in [0, 1]$ in a predetermined fashion. /* Parametric Shortest Path */
- Graph is fixed, metric changes over time in a predetermined fashion. /* Time-Dependent Shortest Path */
    - Arc lengths (e.g., traversal-time / consumption) change with departure-time from tail which is treated as a real-valued variable (functions with continuous domain, but not necessarily continuous range).

Why Time-Dependent Shortest Paths?

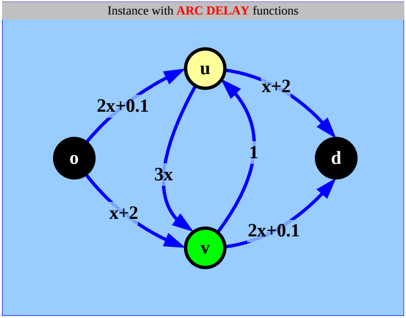
Real-life networks: Elements demostrate temporal behavior.

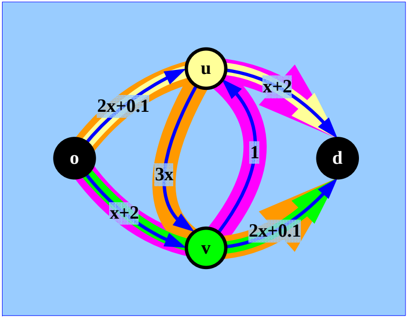
- Graph is **fixed**, metric **changes over time** in a predetermined fashion. /*

  Time-Dependent Shortest Path */

  - Arc lengths (e.g., traversal-time / consumption) **change with departure-time from tail** which is treated as a real-valued variable (functions with continuous domain, but not necessarily continuous range).

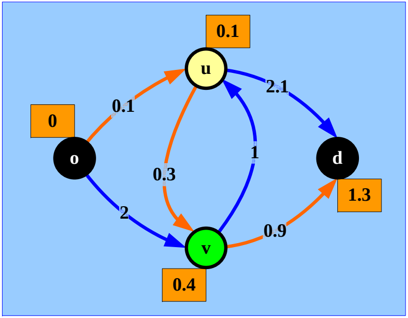
...working with earliest arrivals...

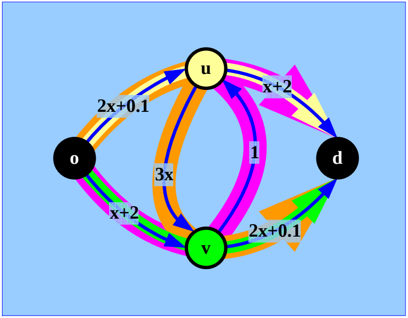




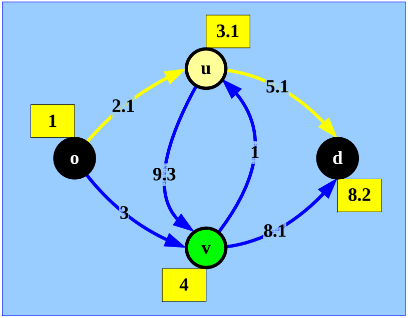
Q1 How would you commute **as fast as possible** from *o* to *d*, for a given departure time (from *o*)?

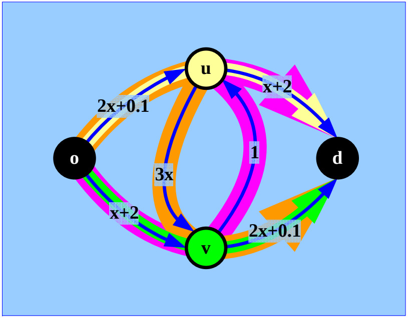
...working with earliest arrivals...





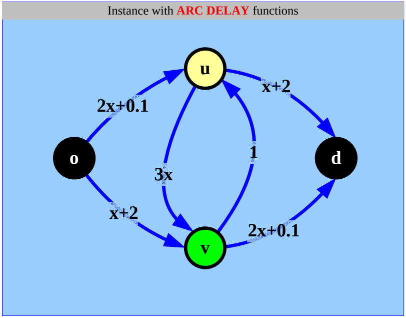
Q1 How would you commute **as fast as possible** from o to d, for a given departure time (from o)? Eg: $\boxed{t_o = 0}$

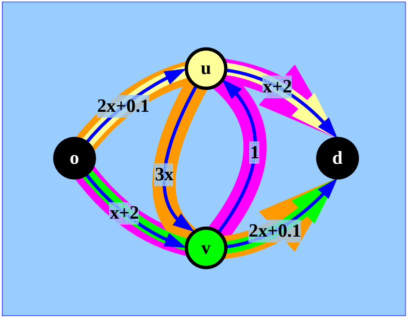




Q1 How would you commute **as fast as possible** from *o* to *d*, for a given departure time (from *o*)? Eg: $\boxed{t_o = 1}$

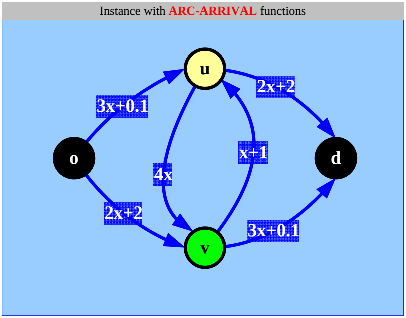
...working with earliest arrivals...

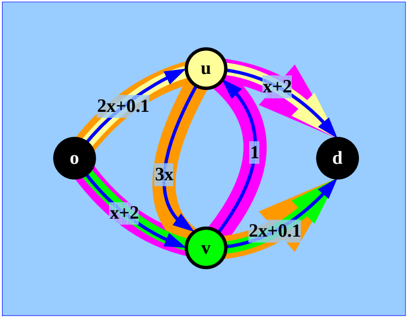




Q1 How would you commute **as fast as possible** from *o* to *d*, for a given departure time (from *o*)?

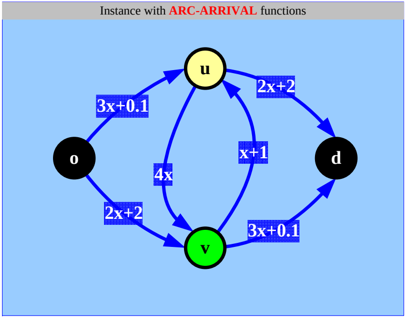
Q2 What if you are not sure about the departure time?

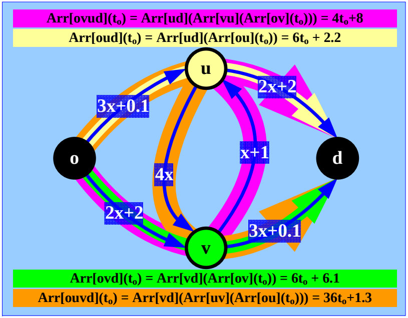




Q1 How would you commute **as fast as possible** from *o* to *d*, for a given departure time (from *o*)?

Q2 What if you are not sure about the departure time?

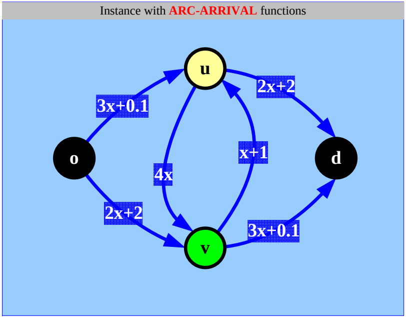


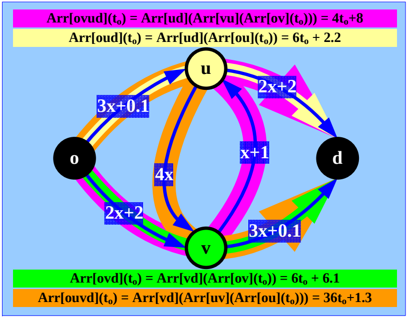


Q1 How would you commute **as fast as possible** from *o* to *d*, for a given departure time (from *o*)?

Q2 What if you are not sure about the departure time?

...working with earliest arrivals...



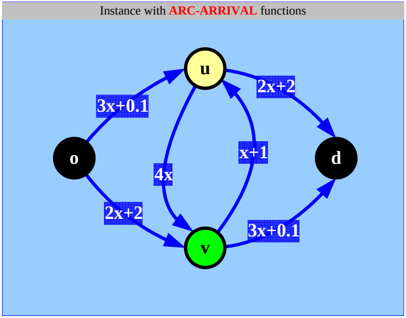


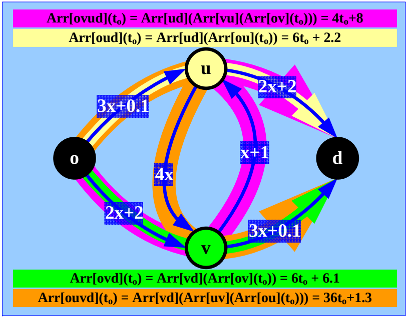
Q1 How would you commute **as fast as possible** from *o* to *d*, for a given departure time (from *o*)?

Q2 What if you are not sure about the departure time?

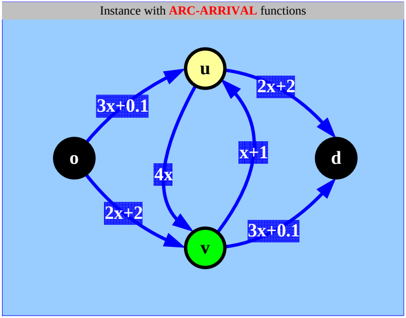
A2

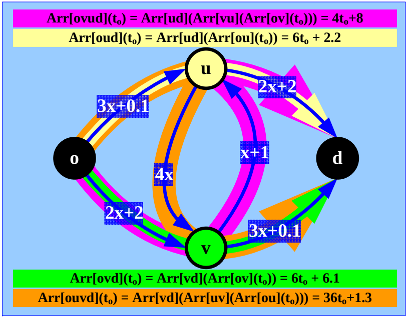
$$\text{shortest } od\text{–path} = \begin{cases} \textbf{orange path}, \text{ if} & t_o \in [0, 0.03] \\ \textbf{yellow path}, \text{ if} & t_o \in [0.03, 2.9] \\ \textbf{purple path}, \text{ if} & t_o \in [2.9, +\infty) \end{cases}$$





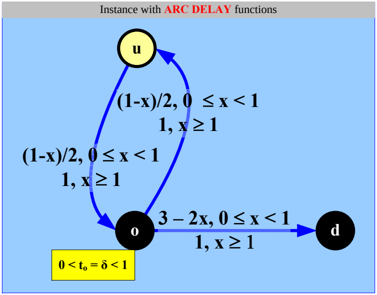
Q3 Would **waiting-at-nodes** be worth it?





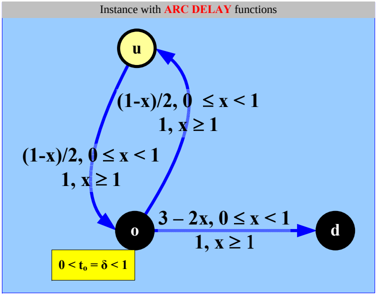
Q3 Would **waiting-at-nodes** be worth it?

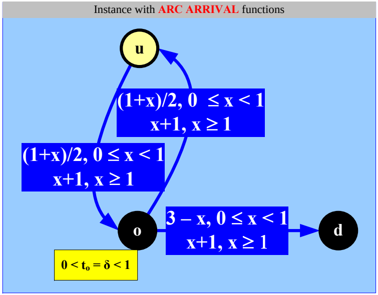
A3 NO, since arrival-time functions are *non-decreasing* functions of departure-time from origin.



Q4 Would **waiting-at-nodes** be worth it in this case?

...waiting at nodes...





Q4 Would **waiting-at-nodes** be worth it in this case?

A4 YES, because arrival-time function is *decreasing* in x: Wait until time 1 and then traverse od, if already present at o at time $t_o < 1$. Otherwise, traverse od immediately.

Waiting Policies

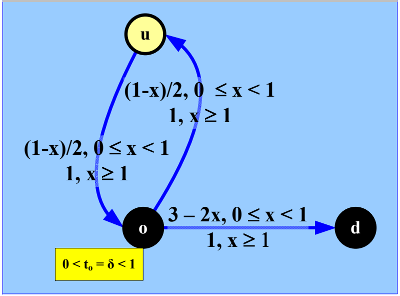
Unrestricted Waiting (UW) Unlimited waiting is allowed at every node along an *od*-path.

Origin Waiting (OW) Unlimited waiting is only allowed at the origin node of each *od*-path.

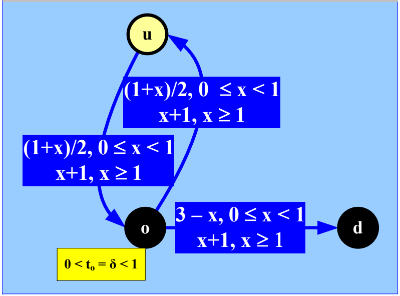
Forbidden Waiting (FW) No waiting is allowed at any node of each *od*-path.

> Depending on the *waiting policy*, the scheduler has to decide not only for an optimal connecting path (that assures the earliest arrival at the destination), but also for the appropriate optimal waiting times at the nodes along this path.

Instance with **ARC DELAY** functions

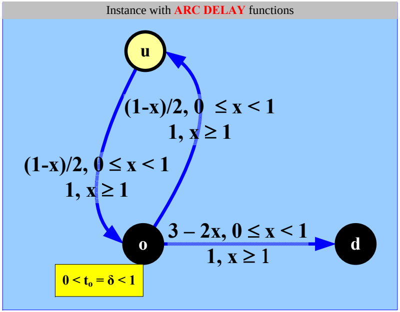


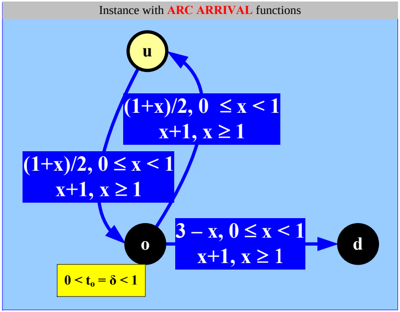
Instance with **ARC ARRIVAL** functions



Q5 What if **waiting-at-nodes** is **forbidden**?

....forbidden waiting times....

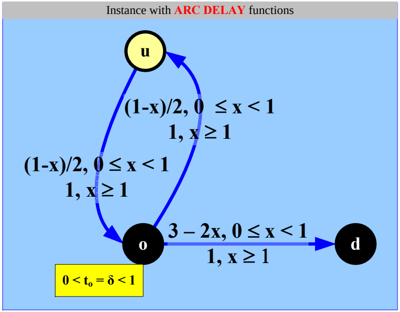


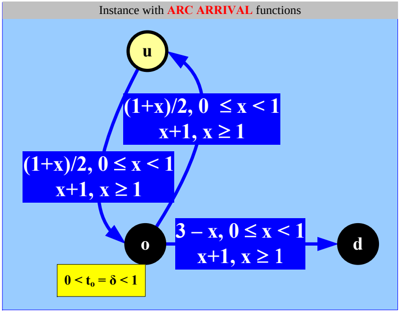


Q5 What if waiting-at-nodes is forbidden?

A5 An infinite, non-simple TD shortest *od*-path with finite delay.

o					d
δ					$3-\delta$





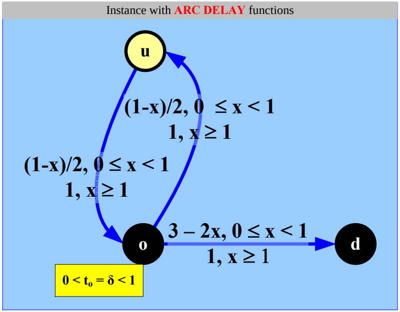
Q5 What if waiting-at-nodes is forbidden?

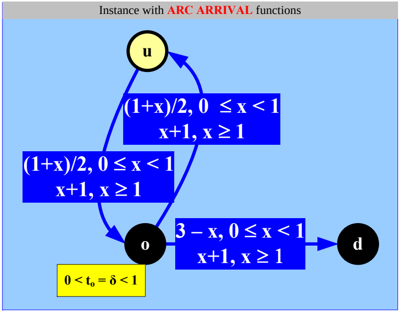
A5 An infinite, non-simple TD shortest *od*-path with finite delay.

o	u	o			d
δ	$\frac{1+\delta}{2}$	$\frac{3+\delta}{4}$			$3 - \frac{3+\delta}{4} > 2$

TDSP :: EXAMPLE 3

....forbidden waiting times....

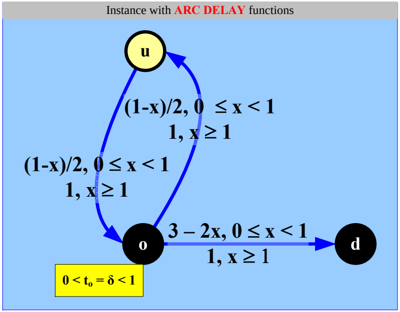


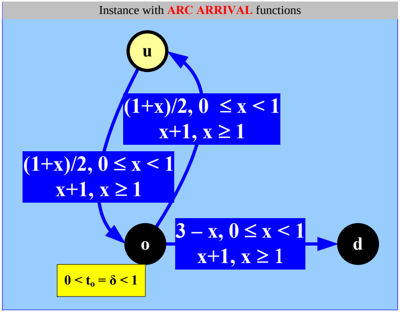


Q5 What if waiting-at-nodes is forbidden?

A5 An infinite, non-simple TD shortest *od*-path with finite delay.

o	u	o	u	o	d
δ	$\frac{1+\delta}{2}$	$\frac{3+\delta}{4}$	$\frac{7+\delta}{8}$	$\frac{15+\delta}{16}$	$3 - \frac{15+\delta}{16} > 2$





Q5 What if **waiting-at-nodes** is forbidden?

A5 An infinite, non-simple TD shortest *od*-path with finite delay.

o	u	o	presence at o after $k \uparrow \infty$ visits of u	d
δ	$\frac{1+\delta}{2}$	$\frac{3+\delta}{4}$	$\lim_{k\uparrow\infty} \frac{2^{2k}-1+\delta}{2^{2k}} = 1$	$t_d \downarrow 2$

Subpath optimality and *shortest path simplicity* are **not guaranteed** for TDSP, if waiting-at-nodes is forbidden.

Inexistence of Optimal Waiting Times

Q Do optimal waiting times at nodes always exist?

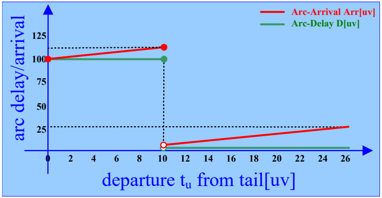
Inexistence of Optimal Waiting Times

Q Do optimal waiting times at nodes always exist?

A Unfortunately NOT! Bad Example:

$$D[uv](t_u) = \begin{cases} 100, & t_u \leq 10, \\ 1, & t_u > 10 \end{cases}$$

$$Arr[uv](t_u) = \begin{cases} t_u + 100, & t_u \leq 10, \\ t_u + 1, & t_u > 10 \end{cases}$$



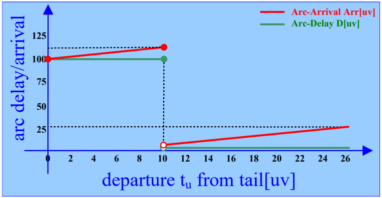
Inexistence of Optimal Waiting Times

Q Do optimal waiting times at nodes always exist?

A Unfortunately NOT! Bad Example:

$$D[uv](t_u) = \begin{cases} 100, & t_u \leq 10, \\ 1, & t_u > 10 \end{cases}$$

$$Arr[uv](t_u) = \begin{cases} t_u + 100, & t_u \leq 10, \\ t_u + 1, & t_u > 10 \end{cases}$$



- **Reason:** *Pathological discontinuity* of the delay / arrival-time function.

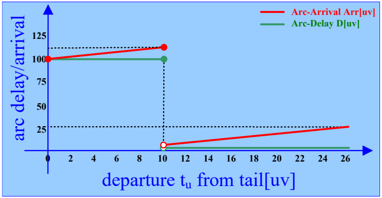
Inexistence of Optimal Waiting Times

Q Do optimal waiting times at nodes always exist?

A Unfortunately NOT! Bad Example:

$$D[uv](t_u) = \begin{cases} 100, & t_u \leq 10, \\ 1, & t_u > 10 \end{cases}$$

$$Arr[uv](t_u) = \begin{cases} t_u + 100, & t_u \leq 10, \\ t_u + 1, & t_u > 10 \end{cases}$$



- **Reason:** *Pathological discontinuity* of the delay / arrival-time function.
- **Solution:** Optimal waiting times always exist for continuous functions, and for (possibly discontinuous) pwl functions for which

  if $\lim_{t \downarrow t_u} D[uv](t) < \lim_{t \uparrow t_u} D[uv](t)$
  then $D[uv](t_u) = \lim_{t \downarrow t_u} D[uv](t)$

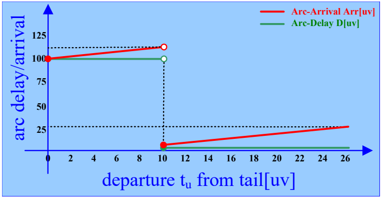
Inexistence of Optimal Waiting Times

Q Do optimal waiting times at nodes always exist?

A Unfortunately NOT! Bad Example:

$$D[uv](t_u) = \begin{cases} 100, & t_u \le 10, \\ 1, & t_u > 10 \end{cases}$$

$$Arr[uv](t_u) = \begin{cases} t_u + 100, & t_u \le 10, \\ t_u + 1, & t_u > 10 \end{cases}$$



- **Reason:** *Pathological discontinuity* of the delay / arrival-time function.
- **Solution:** Optimal waiting times always exist for continuous functions, and for (possibly discontinuous) pwl functions for which

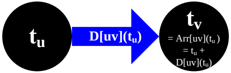
  if $\lim_{t \downarrow t_u} D[uv](t) < \lim_{t \uparrow t_u} D[uv](t)$
  then $D[uv](t_u) = \lim_{t \downarrow t_u} D[uv](t)$

From now on we assume that *optimal waiting times* at nodes always exist and are polynomial-time computable.

FIFO vs non-FIFO Arc Delays

- **(Strict) FIFO Arc-Delays:** The slopes of all the *arc-delay* functions are at least equal to (greater than) -1.

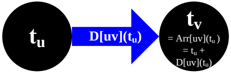
Equivalently: *Arc-arrival* functions are **non-decreasing** (aka **no-overtaking** property).



FIFO vs non-FIFO Arc Delays

- **(Strict) FIFO Arc-Delays:** The slopes of all the *arc-delay* functions are at least equal to (greater than) -1.

Equivalently: *Arc-arrival* functions are **non-decreasing** (aka **no-overtaking** property).

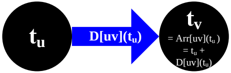


- **Non-FIFO Arc-Delays:** Possibly preferrable to wait for some period at the tail of an arc, before trespassing it. E.g.:
  - Wait for the next *(faster) IC train*, than use the (immediately available) *(slower) local train*.

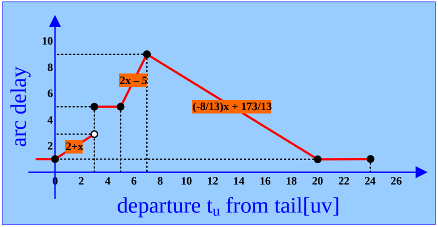
FIFO vs non-FIFO Arc Delays

- **(Strict) FIFO Arc-Delays:** The slopes of all the *arc-delay* functions are at least equal to (greater than) -1.

Equivalently: *Arc-arrival* functions are **non-decreasing** (aka **no-overtaking** property).



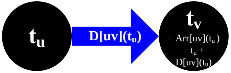
- **Non-FIFO Arc-Delays:** Possibly preferrable to wait for some period at the tail of an arc, before trespassing it. E.g.:
  - Wait for the next *(faster) IC train*, than use the (immediately available) *(slower) local train*.



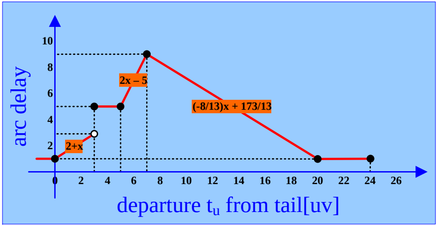
FIFO arc delay example

- **(Strict) FIFO Arc-Delays:** The slopes of all the *arc-delay* functions are at least equal to (greater than) -1.

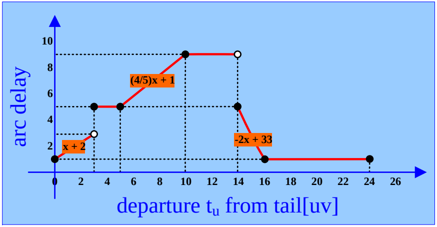
Equivalently: *Arc-arrival* functions are **non-decreasing** (aka **no-overtaking** property).



- **Non-FIFO Arc-Delays:** Possibly preferrable to wait for some period at the tail of an arc, before trespassing it. E.g.:
  - Wait for the next *(faster) IC train*, than use the (immediately available) *(slower) local train*.

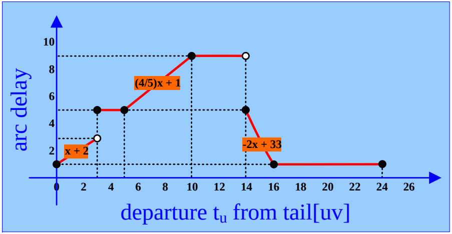


FIFO arc delay example

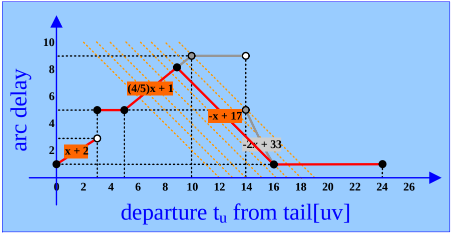


Non-FIFO arc delay example

Non-FIFO+UW Arc ⇔ FIFO Arc

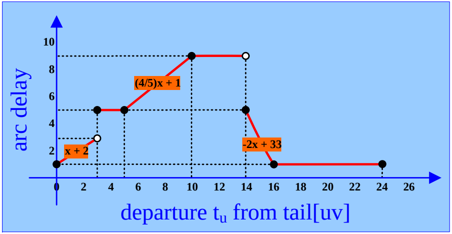


Non-FIFO+UW arc delay function

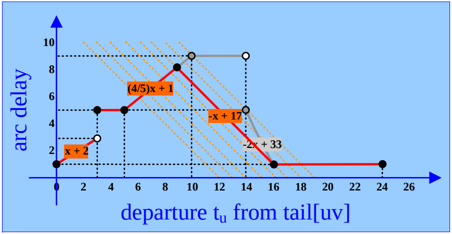


Equivalent FIFO (+FW) arc delay function

Non-FIFO+UW Arc ⇔ FIFO Arc



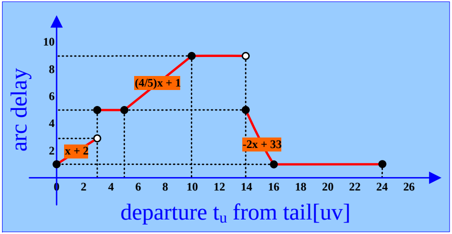
Non-FIFO+UW arc delay function



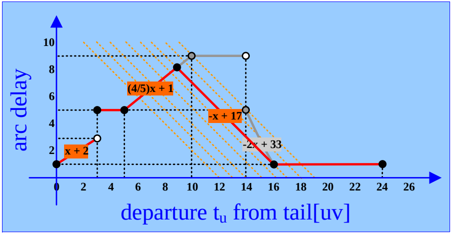
Equivalent FIFO (+FW) arc delay function

- A ``scan'' of the line with slope -1 *from right to left* suffices.

Non-FIFO+UW Arc ⇔ FIFO Arc

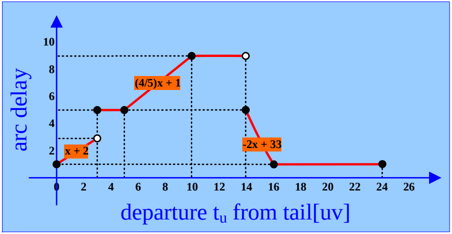


Non-FIFO+UW arc delay function

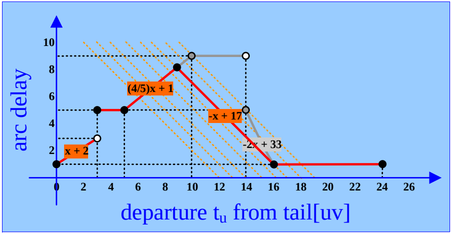


Equivalent FIFO (+FW) arc delay function

- A ``scan'' of the line with slope -1 *from right to left* suffices.
  - **Shortcircuit** pieces of the arc-delay function lying above the line of slope -1.



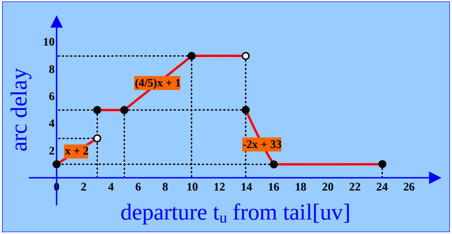
Non-FIFO+UW arc delay function



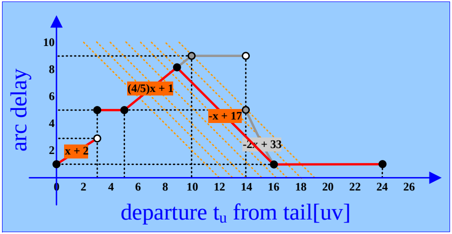
Equivalent FIFO (+FW) arc delay function

- A ``scan'' of the line with slope -1 *from right to left* suffices.
  - **Shortcircuit** pieces of the arc-delay function lying above the line of slope -1.
- *Identical arrival-times* in **Non-FIFO+UW** and **FIFO** instances.

Non-FIFO+UW Arc ⇔ FIFO Arc



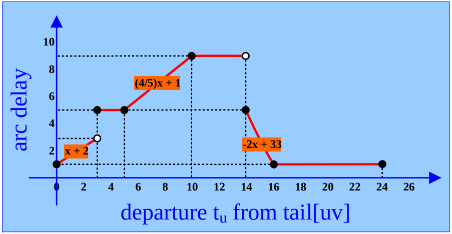
Non-FIFO+UW arc delay function



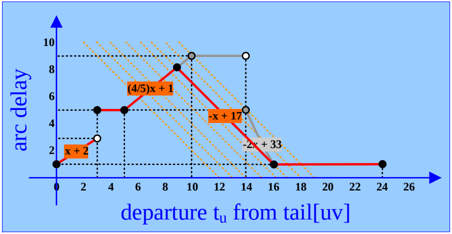
Equivalent FIFO (+FW) arc delay function

- A \`\`scan'' of the line with slope -1 *from right to left* suffices.
  - **Shortcircuit** pieces of the arc-delay function lying above the line of slope -1.
- *Identical arrival-times* in **Non-FIFO+UW** and **FIFO** instances.
- Need to consider *latest departures* given the arrival times, in order to compute the optimal waiting times in the original **Non-FIFO+UW** instance.

Non-FIFO+UW Arc ⇔ FIFO Arc



Non-FIFO+UW arc delay function



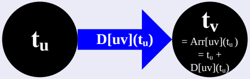
Equivalent FIFO (+FW) arc delay function

- A ``scan'' of the line with slope -1 *from right to left* suffices.
  - **Shortcircuit** pieces of the arc-delay function lying above the line of slope -1.
- *Identical arrival-times* in **Non-FIFO+UW** and **FIFO** instances.
- Need to consider *latest departures* given the arrival times, in order to compute the optimal waiting times in the original **Non-FIFO+UW** instance.
- Interested in programming the transformation? Let me know!

DEFINITION: Time-Dependent Shortest Paths

INPUT:

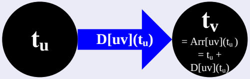
- *Directed* graph $G = (V, A)$ with **succinctly represented arc-travel-time** functions $(D[a])_{a\in A}$. $(Arr[a] = ID + D[a])_{a\in A}$.



Variants of Time-Dependent Shortest Path

DEFINITION: Time-Dependent Shortest Paths

INPUT:



- *Directed* graph $G = (V, A)$ with **succinctly represented** **arc-travel-time** functions $(D[a])_{a\in A}$. $(Arr[a] = ID + D[a])_{a\in A}$.

DEFINITIONS:

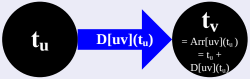
- **Path arrival / travel-time** functions: $\forall p = (a_1, \ldots, a_k) \in P_{o,d}$, $Arr[p] = Arr[a_k] \circ \cdots \circ Arr[a_1]$ (*composition* of the involved arc-arrivals). $D[p] = Arr[p] - ID$.

- **Earliest-arrival / Shortest-travel-time** functions: $Arr[o, d] = \min_{p\in P_{o,d}} \{ Arr[p] \}$, $D[o, d] = Arr[o, d] - ID$.

DEFINITION: Time-Dependent Shortest Paths

INPUT:

- *Directed* graph $G = (V, A)$ with **succinctly represented arc-travel-time** functions $(D[a])_{a\in A}$. $(Arr[a] = ID + D[a])_{a\in A}$.



DEFINITIONS:

- **Path arrival / travel-time** functions: $\forall p = (a_1, \ldots, a_k) \in P_{o,d}$, $Arr[p] = Arr[a_k] \circ \cdots \circ Arr[a_1]$ (*composition* of the involved arc-arrivals). $D[p] = Arr[p] - ID$.
- **Earliest-arrival / Shortest-travel-time** functions: $Arr[o,d] = \min_{p\in P_{o,d}} \{ Arr[p] \}$, $D[o,d] = Arr[o,d] - ID$.

GOAL1: For departure-time t_o from o, determine $t_d = Arr[o,d](t_o)$.

GOAL2: Provide a **succinct representation** of $Arr[o,d]$ (or $D[o,d]$).

Why Care for Both Goals?

1. Not always sure when to depart (still think about it)! Possessing the entire *distance function* $D[o,d]$ allows for easy answers (e.g., via look-ups) in several queries for varying departure times, or even finding the *minimum travel / ealriest-arrival time* within a window of possible departure times.

Why Care for Both Goals?

1. Not always sure when to depart (still think about it)! Possessing the entire *distance function* $D[o,d]$ allows for easy answers (e.g., via look-ups) in several queries for varying departure times, or even finding the *minimum travel / ealriest-arrival time* within a window of possible departure times.
2. Need to respond **efficiently** (theory: in sublinear time; practice: in micro/milliseconds) to arbitrary queries in *large-scale nets*, for arbitrary departure-times and $od-$pairs.

Why Care for Both Goals?

1. Not always sure when to depart (still think about it)! Possessing the entire *distance function* $D[o, d]$ allows for easy answers (e.g., via look-ups) in several queries for varying departure times, or even finding the *minimum travel / ealriest-arrival time* within a window of possible departure times.

2. Need to respond **efficiently** (theory: in sublinear time; practice: in micro/milliseconds) to arbitrary queries in *large-scale nets*, for arbitrary departure-times and $od-$pairs.

- **Preprocess** (offline) towards GOAL2 (succinct representations of *selected* $D[o, d]$ functions) in order to support *real-time* responses to **queries** of GOAL1.

Why Care for Both Goals?

1. Not always sure when to depart (still think about it)! Possessing the entire *distance function* $D[o, d]$ allows for easy answers (e.g., via look-ups) in several queries for varying departure times, or even finding the *minimum travel / ealriest-arrival time* within a window of possible departure times.

2. Need to respond **efficiently** (theory: in sublinear time; practice: in micro/milliseconds) to arbitrary queries in *large-scale nets*, for arbitrary departure-times and $od-$pairs.

☺ **Preprocess** (offline) towards GOAL2 (succinct representations of *selected* $D[o, d]$ functions) in order to support *real-time* responses to **queries** of GOAL1.

☹ **Preprocessing** of distance summaries (as in static case) requires to precompute functions instead of scalars.

Consequencies of Different Network Models

- (Dreyfus (1969)) Prefix-subpath optimality holds in **Non-FIFO+UW** networks (given that optimal waiting times *exist*). The same applies for **FIFO** networks.

Consequencies of Different Network Models

- (Dreyfus (1969)) Prefix-subpath optimality holds in **Non-FIFO+UW** networks (given that optimal waiting times *exist*). The same applies for **FIFO** networks.
- (OR (1990)) Prefix-subpath optimality does NOT hold in **non-FIFO+FW** networks (cf. EXAMPLE of Slide 7).

Consequencies of Different Network Models

- (Dreyfus (1969)) Prefix-subpath optimality holds in **Non-FIFO+UW** networks (given that optimal waiting times *exist*). The same applies for **FIFO** networks.
- (OR (1990)) Prefix-subpath optimality does NOT hold in **non-FIFO+FW** networks (cf. EXAMPLE of Slide 7).
- (OR (1990)) If arc-delay functions are *continuous*, or *piecewise continuous with negative discontinuities*[1], then the solution (path+waiting policy) in non-FIFO+UW network induces a solution in **non-FIFO+OW** network using the **same path** and appropriate waiting time **only at the origin**.

[1] This means that: $\forall t_u, D[uv](t_u) \geq \lim_{t \downarrow t_u} D[uv](t)$)

Consequencies of Different Network Models

- (Dreyfus (1969)) Prefix-subpath optimality holds in **Non-FIFO+UW** networks (given that optimal waiting times *exist*). The same applies for **FIFO** networks.
- (OR (1990)) Prefix-subpath optimality does NOT hold in **non-FIFO+FW** networks (cf. EXAMPLE of Slide 7).
- (OR (1990)) If arc-delay functions are *continuous*, or *piecewise continuous with negative discontinuities*[1], then the solution (path+waiting policy) in non-FIFO+UW network induces a solution in **non-FIFO+OW** network using the **same path** and appropriate waiting time **only at the origin**.
- (KZ (2014)) In **strict-FIFO** networks, (general) subpath optimality holds also in the time-dependent case.

[1]This means that: $\forall t_u, D[uv](t_u) \geq \lim_{t \downarrow t_u} D[uv](t)$)

Consequencies of Different Network Models

- (Dreyfus (1969)) Prefix-subpath optimality holds in **Non-FIFO+UW** networks (given that optimal waiting times *exist*). The same applies for **FIFO** networks.
- (OR (1990)) Prefix-subpath optimality does NOT hold in **non-FIFO+FW** networks (cf. EXAMPLE of Slide 7).
- (OR (1990)) If arc-delay functions are *continuous*, or *piecewise continuous with negative discontinuities*[1], then the solution (path+waiting policy) in non-FIFO+UW network induces a solution in **non-FIFO+OW** network using the **same path** and appropriate waiting time **only at the origin**.
- (KZ (2014)) In **strict-FIFO** networks, (general) subpath optimality holds also in the time-dependent case.
- (FHS (2011)) In **(strict) FIFO** networks, $Arr[o, d]$ is non-decreasing (increasing).

[1] This means that: $\forall t_u, D[uv](t_u) \geq \lim_{t \downarrow t_u} D[uv](t)$)

Algorithms for TDSP

- For arbitrary (o, d, t_o) queries (**GOAL1**):

Algorithms for TDSP

- For arbitrary (o, d, t_o) queries (**GOAL1**):
  - TD variants of Dijkstra and Bellman-Ford algorithms work correctly in **FIFO** networks, and in **non-FIFO+UW** networks. Time complexity slightly worse (when updating arc labels, some arc-delay *functions* are evaluated).
  - TD variants of Dijkstra and Bellman-Ford algorithms do NOT work correctly in **non-FIFO+FW** networks. Determining existence of a *finite-hop solution* is $\mathcal{NP}$−hard.

Algorithms for TDSP

- For arbitrary (o, d, t_o) queries (**GOAL1**):
  - TD variants of Dijkstra and Bellman-Ford algorithms work correctly in **FIFO** networks, and in **non-FIFO+UW** networks. Time complexity slightly worse (when updating arc labels, some arc-delay *functions* are evaluated).
  - TD variants of Dijkstra and Bellman-Ford algorithms do NOT work correctly in **non-FIFO+FW** networks. Determining existence of a *finite-hop solution* is $\mathcal{NP}$−hard.
- For arbitrary (o, d) queries (**GOAL2**):
  - (OR (1990)) Propose a TD-variant of Bellman-Ford, for **non-FIFO+UW** networks.

Algorithms for TDSP

- For arbitrary (o, d, t_o) queries (**GOAL1**):
  - TD variants of Dijkstra and Bellman-Ford algorithms work correctly in **FIFO** networks, and in **non-FIFO+UW** networks. Time complexity slightly worse (when updating arc labels, some arc-delay *functions* are evaluated).
  - TD variants of Dijkstra and Bellman-Ford algorithms do NOT work correctly in **non-FIFO+FW** networks. Determining existence of a *finite-hop solution* is $\mathcal{NP}-$hard.
- For arbitrary (o, d) queries (**GOAL2**):
  - (OR (1990)) Propose a TD-variant of Bellman-Ford, for **non-FIFO+UW** networks.
    - ☹ Complexity is **polynomial** in the number of ``elementary'' *functional operations*. i.e., (EVAL, LINEAR COMBINATION, MIN, COMPOSITION)
    - ☹ Not so ``elementary'' operations after all (see next slides)!!!

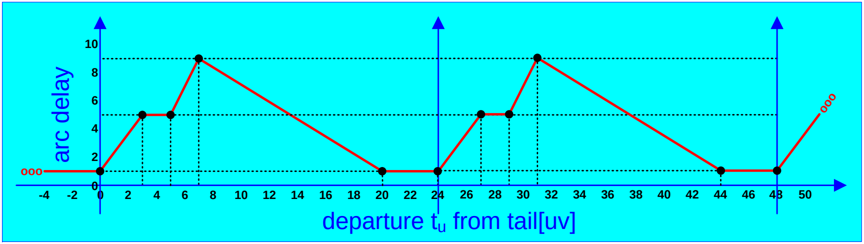
Algorithms for TDSP

... in FIFO, continuous, pwl instances

Input/Output Data

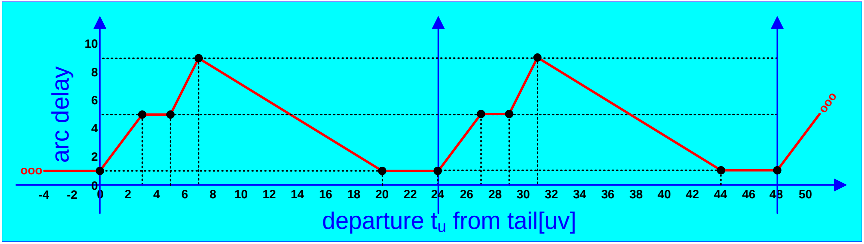
PWL Arc Delays

Forward Description (as function of departure times from origin)

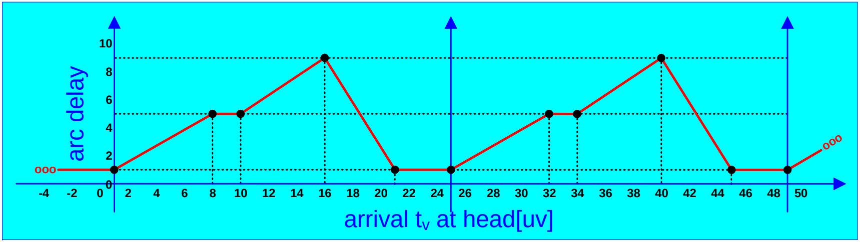


PWL Arc Delays

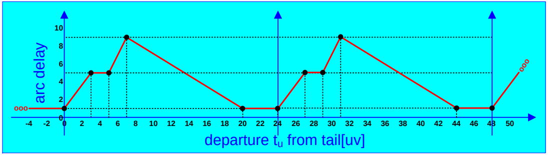
Forward Description (as function of departure times from origin)



Reverse Description (as function of arrival times at destination)



How to Store/Access PWL Arc Delays



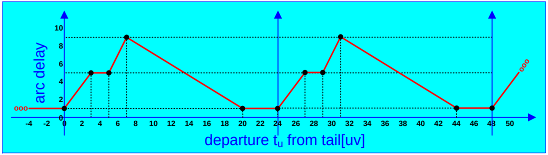
- Exploit *periodicity* and *piecewise-linearity*:

$$\forall t_u \in \mathbb{R},\ \overrightarrow{D}[uv](t_u) \;=\; \begin{cases} \frac{4}{3}t_u + 1, & 0 \le t_u \textbf{mod}\ T \le 3 \\ 5, & 3 \le t_u \textbf{mod}\ T \le 5 \\ 2t_u - 5, & 5 \le t_u \textbf{mod}\ T \le 7 \\ -\frac{8}{13}t_u + \frac{173}{13}, & 7 \le t_u \textbf{mod}\ T \le 20 \\ 1, & 20 \le t_u \textbf{mod}\ T \le 24 \end{cases}$$

- Representation: Array of **(slope-constant-dep.time UB) triples** equipped with advanced (binary/predecessor) *search capabilities*.

$\left(\frac{4}{3}, 1, 3\right)$	$(0, 5, 5)$	$(2, -5, 7)$	$\left(-\frac{8}{13}, \frac{173}{13}, 20\right)$	$(0, 1, 24)$

How to Store/Access PWL Arc Delays

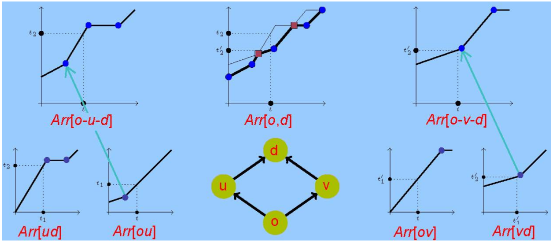


- Exploit *periodicity* and *piecewise-linearity*:

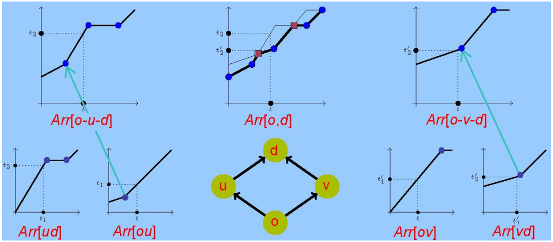
$$\forall t_u \in \mathbb{R},\ \overrightarrow{D}[uv](t_u) \;=\; \begin{cases} \frac{4}{3}t_u + 1, & 0 \le t_u \textbf{mod } T \le 3 \\ 5, & 3 \le t_u \textbf{mod } T \le 5 \\ 2t_u - 5, & 5 \le t_u \textbf{mod } T \le 7 \\ -\frac{8}{13}t_u + \frac{173}{13}, & 7 \le t_u \textbf{mod } T \le 20 \\ 1, & 20 \le t_u \textbf{mod } T \le 24 \end{cases}$$

- Representation: Array of **(dep.time-delay) pairs** equipped with advanced (binary/predecessor) *search capabilities*.

$(0,1)$	$(3,5)$	$(5,5)$	$(7,9)$	$(20,1)$

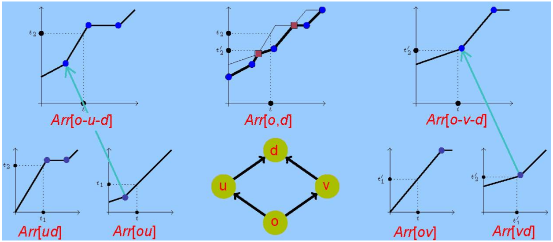


- **Primitive Breakpoint (PB):** Departure-time b'_e from $head[e]$ at which $D[e]$ changes slope (assume $K \in O(m)$ PBs in total).



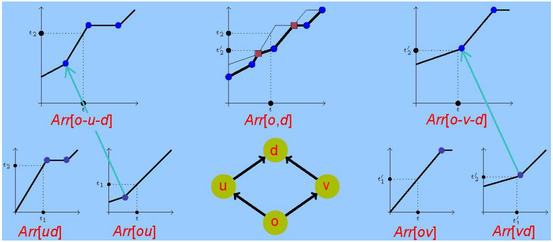
- **Primitive Breakpoint (PB):** Departure-time b'_e from $head[e]$ at which $D[e]$ changes slope (assume $K \in \mathrm{O}(m)$ PBs in total).
- **Primitive Image (PI):** **Latest departure-time** b_e from origin o s.t. earliest-arrival-time $b'_e = Arr[o, tail(e)](b_e)$ coincides with a breakpoint for $D[e]$.

Piecewise Linearity of Path / Earliest Arrivals



- **Primitive Breakpoint (PB):** Departure-time b'_e from $head[e]$ at which $D[e]$ changes slope (assume $K \in \mathrm{O}(m)$ PBs in total).
- **Primitive Image (PI):** **Latest departure-time** b_e from origin o s.t. earliest-arrival-time $b'_e = Arr[o, tail(e)](b_e)$ coincides with a breakpoint for $D[e]$.
- **Minimization Breakpoint (MB):** Departure-time b_v from origin o s.t. $Arr[o, v]$ **changes slope** due to application of MIN.

Piecewise Linearity of Path / Earliest Arrivals



- **Primitive Breakpoint (PB):** Departure-time b'_e from $head[e]$ at which $D[e]$ changes slope (assume $K \in O(m)$ PBs in total).
- **Primitive Image (PI):** **Latest departure-time** b_e from origin o s.t. earliest-arrival-time $b'_e = Arr[o, tail(e)](b_e)$ coincides with a breakpoint for $D[e]$.
- **Minimization Breakpoint (MB):** Departure-time b_v from origin o s.t. $Arr[o, v]$ **changes slope** due to application of MIN.
- Periodicity of arc-delays implies periodicity of earliest-arrival function $Arr[o, d]$.

Known Issues wrt Representations

- **Same representation** both for arc-arrival (or delay) functions and earliest-arrival (or shortest-travel-time) functions.
  - Convenient for handling artificial arcs (representing shortest-travel-time functions) in *overlay abstractions* of the road network.

Known Issues wrt Representations

- **Same representation** both for arc-arrival (or delay) functions and earliest-arrival (or shortest-travel-time) functions.
  - Convenient for handling artificial arcs (representing shortest-travel-time functions) in *overlay abstractions* of the road network.

- Too many (worst case: $n^{\Theta(\log(n))}$) breakpoints to store $Arr[o,d]$ (or $D[o,d]$), even for **linear** arc-delays and **very sparse** graphs.

Known Issues wrt Representations

- **Same representation** both for arc-arrival (or delay) functions and earliest-arrival (or shortest-travel-time) functions.
  - Convenient for handling artificial arcs (representing shortest-travel-time functions) in *overlay abstractions* of the road network.

- Too many (worst case: $n^{\Theta(\log(n))}$) breakpoints to store $Arr[o,d]$ (or $D[o,d]$), even for **linear** arc-delays and **very sparse** graphs.

- We need only $O\left(\frac{1}{\varepsilon} \cdot \log\left(\frac{D_{\max}[o,d]}{D_{\min}[o,d]}\right)\right)$ breakpoints for a $(1+\varepsilon)$ **upper approximation** $\overline{D}[o,d]$ of $D[o,d]$, for the case of continuous, piecewise-linear arc-delays.

Complexity of TDSP

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (I)

A Useful Observation (L2.1-2.2 in FHS11)

For any pair of **monotone, pwl** functions f and g, both their composition $f \circ g$ and their minimum $\min\{f, g\}$ are also **monotone, pwl** functions.

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (I)

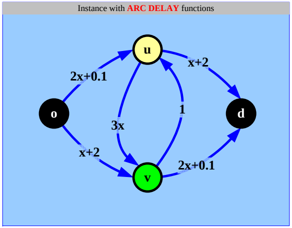
A Useful Observation (L2.1-2.2 in FHS11)

For any pair of **monotone, pwl** functions f and g, both their composition $f \circ g$ and their minimum $\min\{f, g\}$ are also **monotone, pwl** functions.

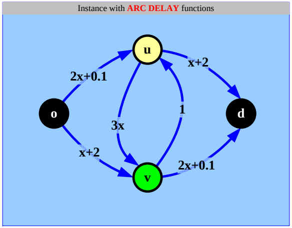
Parametric Shortest Path (PSP): A Similar (but different) Problem

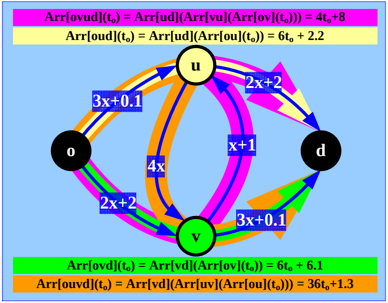
- INPUT: $G = (V, A)$, $o, d \in V$. A *linear length function* $\ell[a](\gamma) = \lambda[a] \cdot \gamma + \mu[a]$ per edge $a \in A$ (negative lengths are **allowed**).
- DEFINITIONS:
  - **Path-length:** $\forall p \in G, L[p](\gamma) = \sum_{a \in p} \ell[a](\gamma)$.
  - **Min-length:** $\forall o, d \in V, L[o,d](\gamma) = \min_{p \in P_{o,d}}\{L[p](\gamma)\}$.
- GOAL1: Compute $L[o,d]$ for a **given value** of γ.
- GOAL2: Succinctly represent $L[o,d]$ for **all (real) values** of γ.

TDSP vs PSP?

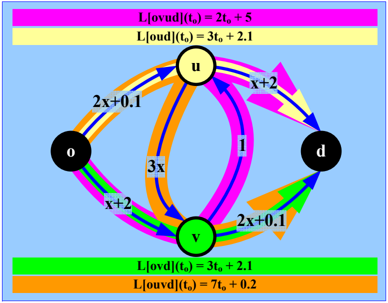


TDSP vs PSP?





TDSP: Arc-arrival composition along paths



PSP: Arc-length addition along paths

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (II)

Known Fact (Carstensen (1984), Mulmuley-Shah (2000))

There exists (linear) PSP-instance with $n^{\Omega(\log n)}$ BPs in $L[o,d]$.

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (II)

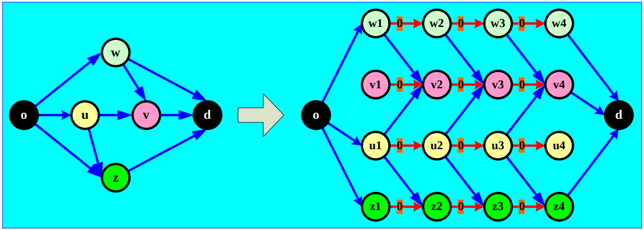
Known Fact (Carstensen (1984), Mulmuley-Shah (2000))

There exists (linear) PSP-instance with $n^{\Omega(\log n)}$ BPs in $L[o,d]$.

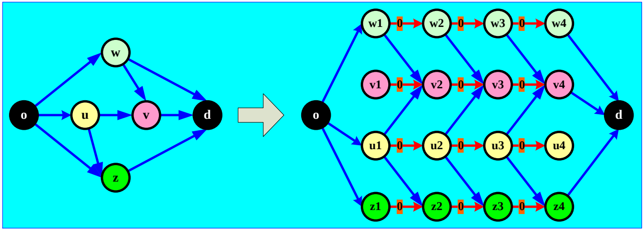
Main Steps for TDSP Lower Bound:

1. Assure *non-negativity of lengths* in the PSP instance, in the **departure-time interval of interest**.
2. Scale properly the PSP instance.
3. Consider the corresponding TDSP instance, with parameter γ handled as departure time from the origin o.
4. Prove that $L[o,d]$ (for PSP instance) and $D[o,d]$ (for TDSP instance) have (almost) the **same number** of BPs.

1. Construct a **layered-graph**, in a **path-length-preserving** manner:



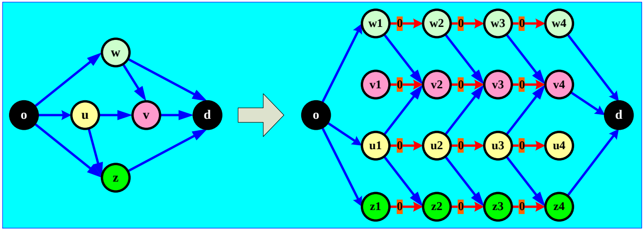
1. Construct a **layered-graph**, in a **path-length-preserving** manner:



Assure non-negativity of arc-lengths in PSP: For the sequence $\langle \gamma_1, \gamma_2, \ldots, \gamma_N \rangle$ of **breakpoints** (BPs) wrt $L[o,d]$, shift arc lengths by $\max\{0, -L_{\min}\}$, $\boxed{L_{\min} = \min_{\gamma \in [\gamma_1, \gamma_N], a \in A(G)} \{L[a](\gamma)\}}$.

Lower Bound: $|BP(Arr_{\text{pwl}}[o, d])| = n^{\Omega(\log n)}$ (III)

1. Construct a **layered-graph**, in a **path-length-preserving** manner:

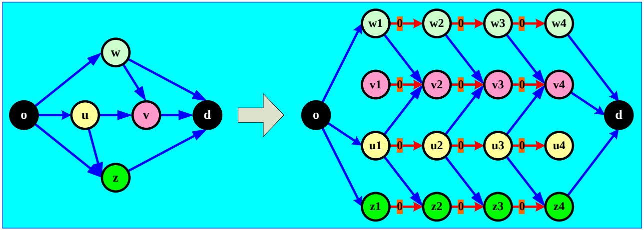


Assure non-negativity of arc-lengths in PSP: For the sequence $\langle \gamma_1, \gamma_2, \ldots, \gamma_N \rangle$ of **breakpoints** (BPs) wrt $L[o, d]$, shift arc lengths by $\max\{0, -L_{\min}\}$, $\boxed{L_{\min} = \min_{\gamma \in [\gamma_1, \gamma_N], a \in A(G)} \{L[a](\gamma)\}}$.

2. Scale arc-length functions in PSP by a proper positive constant μ.

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (III)

1. Construct a **layered-graph**, in a **path-length-preserving** manner:



Assure non-negativity of arc-lengths in PSP: For the sequence $\langle \gamma_1, \gamma_2, \ldots, \gamma_N \rangle$ of **breakpoints** (BPs) wrt $L[o,d]$, shift arc lengths by $\max\{0, -L_{\min}\}$, $\boxed{L_{\min} = \min_{\gamma\in[\gamma_1,\gamma_N], a\in A(G)}\{L[a](\gamma)\}}$.

2. Scale arc-length functions in PSP by a proper positive constant μ.

3. For the TDSP resulting from the scaled PSP when considering γ as departure time, prove that $\forall j \in \{1,\ldots,N-1\}$, at **"time"** $\bar{\gamma}_j \equiv \frac{\gamma_j+\gamma_{j+1}}{2}$ both instances return *the same* shortest $od-$path p_j.

How it works: At given $j \in \{1, \ldots, N-1\}$:

- $\bar{\gamma}_j = \frac{\gamma_j + \gamma_{j+1}}{2}$, $\bar{L}_j = L[p_j](\bar{\gamma}_j) = L[o,d](\bar{\gamma}_j)$.
- $L'_j = \min_{q \in P_{s,d} - \{p_j\}} \{L[q](\bar{\gamma}_j), \Delta_j = L'_j - \bar{L}_j > 0$.
$\Delta_{\min} = \min_{j \in [N-1]} \Delta_j$	$\varepsilon^* = \frac{\Delta_{\min}}{2n}$	$\delta^* = \min_{a \in A : \lambda[a] \neq 0} \left\{ \frac{\Delta_{\min}}{2n\lvert\lambda[a]\rvert} \right\}$

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (IV)

How it works: At given $j \in \{1,\ldots,N-1\}$:

- $\bar{\gamma}_j = \frac{\gamma_j+\gamma_{j+1}}{2}$, $\bar{L}_j = L[p_j](\bar{\gamma}_j) = L[o,d](\bar{\gamma}_j)$.
- $L'_j = \min_{q \in P_{s,d}-\{p_j\}}\{L[q](\bar{\gamma}_j)$, $\Delta_j = L'_j - \bar{L}_j > 0$.
-

$\Delta_{\min} = \min_{j\in[N-1]} \Delta_j$	$\varepsilon^* = \frac{\Delta_{\min}}{2n}$	$\delta^* = \min_{a \in A:\lambda[a]\neq 0}\left\{\frac{\Delta_{\min}}{2n\lvert\lambda[a]\rvert}\right\}$

- **Arc-delay perturbations:** Small-enough so as *not to affect optimality* of p_j in PSP instance: $\forall \varepsilon_a \in (0, \varepsilon^*]$,

$$\boxed{\sum_{a\in p_j} \ell[a](\bar{\gamma}_j + \varepsilon_a) \leq \bar{L}_j + \frac{\Delta_j}{2} < \frac{\bar{L}_j + L'_j}{2} < L'_j \leq \sum_{a\in q} \ell[a](\bar{\gamma}_j),\ \forall q \neq p_j}$$

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (IV)

How it works: At given $j \in \{1,\ldots,N-1\}$:

- $\bar{\gamma}_j = \frac{\gamma_j+\gamma_{j+1}}{2}$, $\bar{L}_j = L[p_j](\bar{\gamma}_j) = L[o,d](\bar{\gamma}_j)$.
- $L'_j = \min_{q\in P_{s,d}-\{p_j\}}\{L[q](\bar{\gamma}_j), \Delta_j = L'_j - \bar{L}_j > 0$.
$\Delta_{\min} = \min_{j\in[N-1]} \Delta_j$	$\varepsilon^* = \frac{\Delta_{\min}}{2n}$	$\delta^* = \min_{a\in A:\lambda[a]\neq 0}\left\{\frac{\Delta_{\min}}{2n\lvert\lambda[a]\rvert}\right\}$
- **Arc-delay perturbations:** Small-enough so as *not to affect optimality* of p_j in PSP instance: $\forall \varepsilon_a \in (0,\varepsilon^*]$,

  $$\boxed{\sum_{a\in p_j}\ell[a](\bar{\gamma}_j+\varepsilon_a) \le \bar{L}_j + \frac{\Delta_j}{2} < \frac{\bar{L}_j+L'_j}{2} < L'_j \le \sum_{a\in q}\ell[a](\bar{\gamma}_j),\ \forall q \neq p_j}$$

- **Departure-time perturbations:** Small-enough so as to cause *not too large arc-delay perturbations*: $\forall a \in A,\ \forall \delta_a \in (0,\delta^*]$,

  $$\boxed{D[a](\bar{\gamma}_j+\delta_a) = \ell[a](\bar{\gamma}_j+\delta_a) \le \ell[a](\bar{\gamma}_j)+\varepsilon^*}$$

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (V)

How it works (continued): At given $j \in \{1, \ldots, N-1\}$:

- Scale-invariance of time-perturbations: **Scaling** of all arc-delays by a **positive** number $\mu > 0$ does not affect at all the range of allowed **time-perturbations** $\delta^* = \min_{a \in A : \lambda[a] \neq 0} \left\{ \frac{\Delta_{\min}}{2n|\lambda[a]|} \right\}$.

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (V)

How it works (continued): At given $j \in \{1, \ldots, N-1\}$:

- Scale-invariance of time-perturbations: **Scaling** of all arc-delays by a **positive** number $\mu > 0$ does not affect at all the range of allowed **time-perturbations** $\delta^* = \min_{a \in A : \lambda[a] \neq 0} \left\{ \frac{\Delta_{\min}}{2n|\lambda[a]|} \right\}$.
- TDSP-instance: Scale the PSP-instance by $\mu = \frac{\delta^*}{2(L_{\max} + \Delta_{\min})}$. Handle the PSP-parameter γ as time.

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (V)

How it works (continued): At given $j \in \{1, \ldots, N-1\}$:

- Scale-invariance of time-perturbations: **Scaling** of all arc-delays by a **positive** number $\mu > 0$ does not affect at all the range of allowed **time-perturbations** $\delta^* = \min_{a \in A : \lambda[a] \neq 0} \left\{ \frac{\Delta_{\min}}{2n|\lambda[a]|} \right\}$.
- TDSP-instance: Scale the PSP-instance by $\mu = \frac{\delta^*}{2(L_{\max} + \Delta_{\min})}$. Handle the PSP-parameter γ as time.
- **Proper scaling** guarantees sufficiently small **departure-time perturbations**: $Arr[p_j](\bar{\gamma}_j) = \bar{\gamma}_j + D[p_j](\bar{\gamma}_j) < \bar{\gamma}_j + \delta^*$.

Lower Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = n^{\Omega(\log n)}$ (V)

How it works (continued): At given $j \in \{1, \ldots, N-1\}$:

- Scale-invariance of time-perturbations: **Scaling** of all arc-delays by a **positive** number $\mu > 0$ does not affect at all the range of allowed **time-perturbations** $\delta^* = \min_{a \in A: \lambda[a] \neq 0} \left\{ \frac{\Delta_{\min}}{2n|\lambda[a]|} \right\}$.
- TDSP-instance: Scale the PSP-instance by $\mu = \frac{\delta^*}{2(L_{\max} + \Delta_{\min})}$. Handle the PSP-parameter γ as time.
- **Proper scaling** guarantees sufficiently small **departure-time perturbations**: $Arr[p_j](\bar{\gamma}_j) = \bar{\gamma}_j + D[p_j](\bar{\gamma}_j) < \bar{\gamma}_j + \delta^*$.

$\therefore$ Small time-perturbations guarantee sufficiently small **arc-delay perturbations**, and thus, optimality of p_j:

$$\boxed{\begin{array}{rcl} D[p_j](\bar{\gamma}_j) & \leq & \mu \cdot \bar{L}_j + \mu \cdot \frac{(n-1)\Delta_{\min}}{2n} \\ & < & \mu \cdot L'_j - \mu \frac{(n-1)\Delta_{\min}}{2n} \leq D[q](\bar{\gamma}_j), \ \forall q \neq p_j \end{array}}$$

QED

Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o, d])| = K \cdot n^{O(\log n)}$ (I)

Observation: (L4.1 in FHS11)

Between any two consecutive Primitive Images (PIs) $t_j < t_{j+1}$, $Arr[o, d]$ forms a concave chain.

Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (I)

Observation: (L4.1 in FHS11)

Between any two consecutive Primitive Images (PIs) $t_j < t_{j+1}$, $Arr[o,d]$ forms a concave chain.

EXPLANATION:

- Any arc-delay is **linear** (no primitive breakpoints occur at edges), if the departure-time domain is restricted to (t_j, t_{j+1}).
- Any path-arrival $Arr[p](t)$ function is a *composition* of linear functions, thus **linear**.
- $Arr[o,d]$ is the application of the $\min$ operator among linear functions, thus **concave**.

Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (I)

Observation: (L4.1 in FHS11)

Between any two consecutive Primitive Images (PIs) $t_j < t_{j+1}$, $Arr[o,d]$ forms a concave chain.

EXPLANATION:

- Any arc-delay is **linear** (no primitive breakpoints occur at edges), if the departure-time domain is restricted to (t_j, t_{j+1}).
- Any path-arrival $Arr[p](t)$ function is a *composition* of linear functions, thus **linear**.
- $Arr[o,d]$ is the application of the $\min$ operator among linear functions, thus **concave**.

- Corollary: $|BP(Arr_{\mathrm{pwl}}[o,d])| \leq$ #different path slopes

Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (I)

Observation: (L4.1 in FHS11)

Between any two consecutive Primitive Images (PIs) $t_j < t_{j+1}$, $Arr[o,d]$ forms a concave chain.

EXPLANATION:

- Any arc-delay is **linear** (no primitive breakpoints occur at edges), if the departure-time domain is restricted to (t_j, t_{j+1}).
- Any path-arrival $Arr[p](t)$ function is a *composition* of linear functions, thus **linear**.
- $Arr[o,d]$ is the application of the $\min$ operator among linear functions, thus **concave**.

- Corollary: $|BP(Arr_{\mathrm{pwl}}[o,d])| \leq$ #different path slopes

- Is this enough?

Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (I)

Observation: (L4.1 in FHS11)

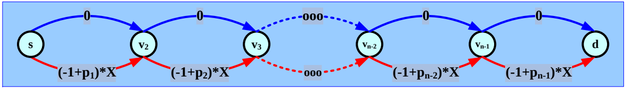
Between any two consecutive Primitive Images (PIs) $t_j < t_{j+1}$, $Arr[o,d]$ forms a concave chain.

EXPLANATION:

- Any arc-delay is linear (no primitive breakpoints occur at edges), if the departure-time domain is restricted to (t_j, t_{j+1}).
- Any path-arrival $Arr[p](t)$ function is a *composition* of linear functions, thus linear.
- $Arr[o,d]$ is the application of the $\min$ operator among linear functions, thus concave.

- Corollary: $|BP(Arr_{\mathrm{pwl}}[o,d])| \le$ #different path slopes

- Is this enough?
- *NO!!!*



Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (II)

OBSERVATION II: (L4.2 in FHS11)

$|BP(Arr_{\mathrm{pwl}}[o,d])| \leq K \cdot |BP(Arr_{\mathrm{lin}}[o,d])|$.

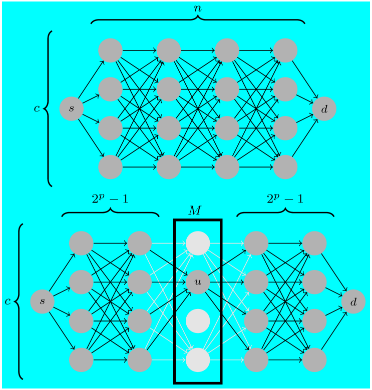
Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (II)

OBSERVATION II: (L4.2 in FHS11)

$|BP(Arr_{\mathrm{pwl}}[o,d])| \leq K \cdot |BP(Arr_{\mathrm{lin}}[o,d])|$.

Lemma 4.3 (FHS11)

$|BP(Arr_{\mathrm{lin}}[o,d])| \leq \frac{(2n+1)^{1+\log c}}{2}$ in a **layered graph** with c layers of n nodes each.



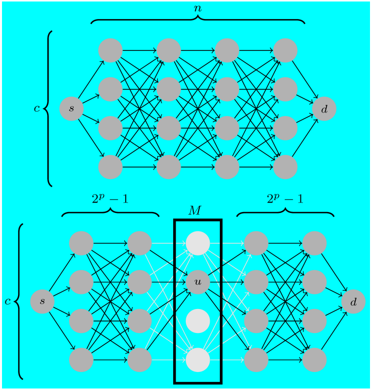
Upper Bound: $|BP(Arr_{\mathrm{pwl}}[o,d])| = K \cdot n^{O(\log n)}$ (II)

OBSERVATION II: (L4.2 in FHS11)

$|BP(Arr_{\mathrm{pwl}}[o,d])| \le K \cdot |BP(Arr_{\mathrm{lin}}[o,d])|$.

Lemma 4.3 (FHS11)

$|BP(Arr_{\mathrm{lin}}[o,d])| \le \frac{(2n+1)^{1+\log c}}{2}$ in a **layered graph** with c layers of n nodes each.



THM4.4 (FHS11)

$|BP(Arr_{\mathrm{lin}}[o,d])| = n^{O(\log n)}$ in any graph G and pair of nodes $o, d \in V(G)$.

(Exact) Output Sensitive Algorithm for Earliest-Arrival Functions

Why Do We Need the Output Sensitive Algorithm?

Why Do We Need the Output Sensitive Algorithm?

1. It gives exactly the distance functions in question, ie, **functional descriptions of earliest-arrivals**, that we would ideally like to have from/to any origin/destination vertex.

Why Do We Need the Output Sensitive Algorithm?

1. It gives exactly the distance functions in question, ie, **functional descriptions of earliest-arrivals**, that we would ideally like to have from/to any origin/destination vertex.
2. We may need to compute *exact distance summaries* for **special pairs of vertices** (eg, from/to hubs, all superhub-to-superhub connections, etc).

Why Do We Need the Output Sensitive Algorithm?

1. It gives exactly the distance functions in question, ie, **functional descriptions of earliest-arrivals**, that we would ideally like to have from/to any origin/destination vertex.

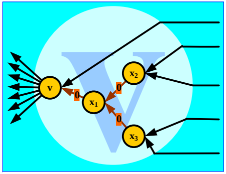
2. We may need to compute *exact distance summaries* for **special pairs of vertices** (eg, from/to hubs, all superhub-to-superhub connections, etc).

3. Interesting to discover whether the complexity of the earliest-arrival functions is indeed so bad **in real (e.g., road) networks**.

The Output-Sensitive Algorithm (I)

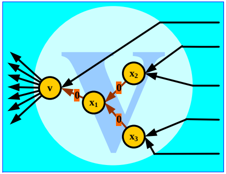
The Output-Sensitive Algorithm (I)

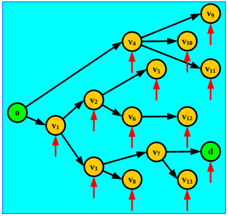
- **ASSUMPTION:** The in-degree of every node in the graph is at most 2.



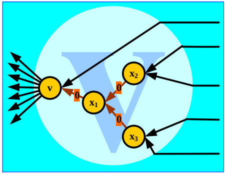
The Output-Sensitive Algorithm (I)

- **ASSUMPTION:** The in-degree of every node in the graph is at most 2.
- Given an arbitrary point in time (*``current time´´*) $t_0 \geq 0$ as departure time from origin o, compute a **TDSP tree**.

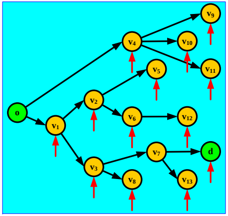




The Output-Sensitive Algorithm (I)

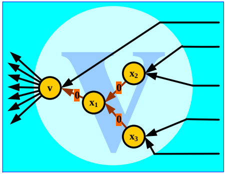


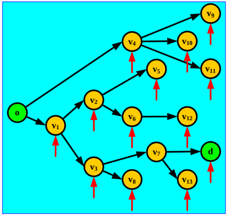
- ASSUMPTION: The in-degree of every node in the graph is at most 2.
- Given an arbitrary point in time (*``current time''*) $t_0 \geq 0$ as departure time from origin o, compute a **TDSP tree**.
- Discover *until when* the TDSP tree is **valid**.
  - $\forall v \in V$, two short alternatives when departing from o at time t_0: Earliest-arrival to each parent, plus delay of corresponding incoming arc.



  - **Minimization (vertex) Certificate** $t_{fail}[v]$: Earliest departure time from o at which the two alternatives of v become **equivalent**.
  - **Primitive (arc) Certificate** $t_{fail}[e]$: Primitive image of the next (ie, after t_0) breakpoint of the arc to come.

The Output-Sensitive Algorithm (I)



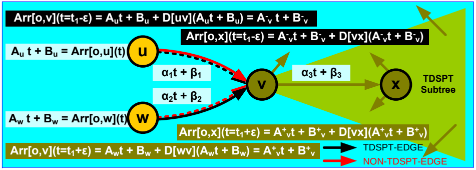


- ASSUMPTION: The in-degree of every node in the graph is at most 2.
- Given an arbitrary point in time (*``current time''*) $t_0 \geq 0$ as departure time from origin o, compute a **TDSP tree**.
- Discover *until when* the TDSP tree is **valid**.
  - $\forall v \in V$, two short alternatives when departing from o at time t_0: Earliest-arrival to each parent, plus delay of corresponding incoming arc.
  - **Minimization (vertex) Certificate** $t_{fail}[v]$: Earliest departure time from o at which the two alternatives of v become **equivalent**.
  - **Primitive (arc) Certificate** $t_{fail}[e]$: Primitive image of the next (ie, after t_0) breakpoint of the arc to come.
- All $(m + n)$ certificates temporarily stored in a *priority queue*.

The Output-Sensitive Algorithm (II)

When current time $t_1 > t_0$ matches the earliest failure-time of a certificate in the queue:

if minimization-certificate failure, at node $v \in V$:

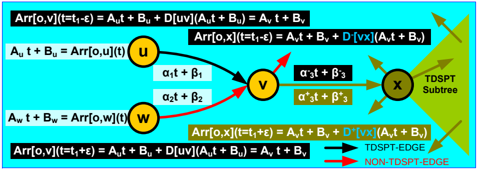


then (1) Update shortest ov–path

/* **ONE-BIT** change in combinatorial structure */

(2) Update $Arr[o,x]$ and $t_{fail}[x]$, $\forall x \in T_v$.

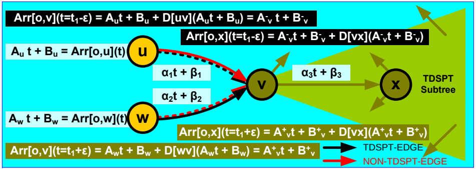
(3) Update $t_{fail}[e]$, $\forall e \in E : x = tail[e] \in T_v$.



The Output-Sensitive Algorithm (II)

When current time $t_1 > t_0$ matches the earliest failure-time of a certificate in the queue:

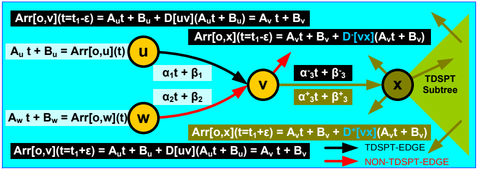
if minimization-certificate failure, at node $v \in V$:



then (1) Update shortest ov–path

/∗ **ONE-BIT** change in combinatorial structure ∗/

(2) Update $Arr[o, x]$ and $t_{fail}[x]$, $\forall x \in T_v$.



(3) Update $t_{fail}[e]$, $\forall e \in E : x = tail[e] \in T_v$.

else /∗ primitive-certificate failure, at arc $e = vx \in E$ ∗/

(1) Update $Arr[o, y]$ and $t_{fail}[y]$, $\forall y \in T_x$.

(2) Update $t_{fail}[e']$, $\forall e' \in E : tail[e'] \in T_x$.

The Output-Sensitive Algorithm (III)

- What to keep in memory:
  - Breakpoint triples for earliest-arrival functions, plus ONE bit (indicating the parent).
  - Advanced search structures, if number of BPs is large.
  - Only temporarily store certificates in a priority queue.

The Output-Sensitive Algorithm (III)

- What to keep in memory:
  - Breakpoint triples for earliest-arrival functions, plus ONE bit (indicating the parent).
  - Advanced search structures, if number of BPs is large.
  - Only temporarily store certificates in a priority queue.
- *Response-time* per certificate failure at $c \in V \cup E$:
  - In the *in-degrees-2 graph* (or any constant-in-degree graph): $\mathrm{O}(|E_c| \cdot \log n)$. E_c is the set of arcs whose tails are in T_c, or $T_{head[c]}$. Logarithmic factor is due to **priority-queue operations**.
  - In the *original graph* (in worst-case): $\mathrm{O}\left(m \times \log^2 n\right)$. Second logarithmic factor is due to **updates of tournament trees** implementing the MIN operator at a particular node, upon emergence of a single certificate failure.

The Output-Sensitive Algorithm (III)

- What to keep in memory:
  - Breakpoint triples for earliest-arrival functions, plus ONE bit (indicating the parent).
  - Advanced search structures, if number of BPs is large.
  - Only temporarily store certificates in a priority queue.
- *Response-time* per certificate failure at $c \in V \cup E$:
  - In the *in-degrees-2 graph* (or any constant-in-degree graph): $\mathrm{O}\left(|E_c| \cdot \log n\right)$. E_c is the set of arcs whose tails are in T_c, or $T_{head[c]}$. Logarithmic factor is due to **priority-queue operations**.
  - In the *original graph* (in worst-case): $\mathrm{O}\left(m \times \log^2 n\right)$. Second logarithmic factor is due to **updates of tournament trees** implementing the MIN operator at a particular node, upon emergence of a single certificate failure.
- *Worst-case time-complexity* of output-sensitive algorithm:

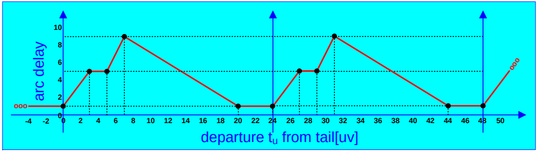
$$\mathrm{O}\left(m \times \log^2 n \times (\mathrm{PRIMBPs} + \mathrm{MINBPs})\right)$$

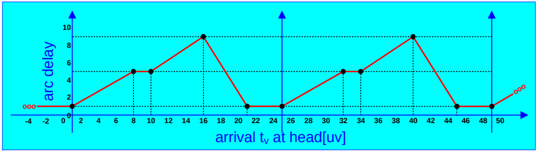
Poly-time Approximation Algorithms

$(1+\varepsilon)-$approximation of $D[o,d]$: Preliminaries

- Why focus on **shortest-travel-time** (delays) functions, and not on **earliest-arrival-time** functions?

- Arc/Path Delay Reversal: Easy task!!!

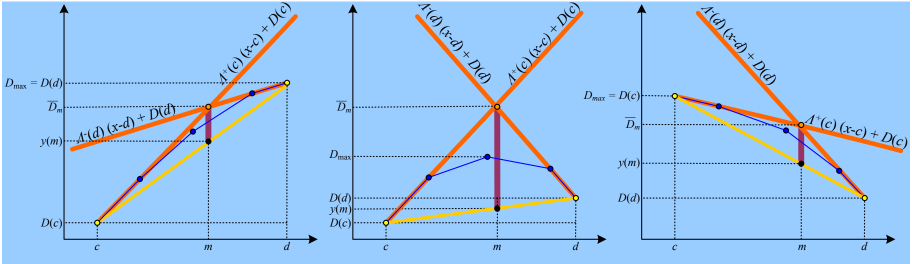




- $t_o = \overleftarrow{Arr}[o,v](t_v) = t_v - \overleftarrow{D}[o,v](t_v)$: **Latest-departure-time** from o to v, as a function of the arrival time t_v at v.

Approximating $D[o,d]$: Quality

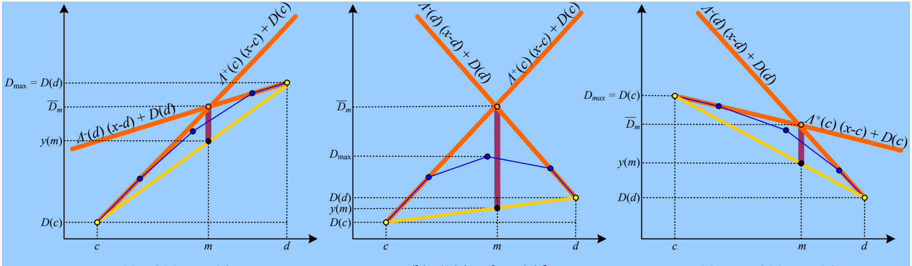
- **Maximum Absolute Error:** A crucial quantity both for the time-complexity and for the space-complexity of the algorithm:



(a) $\Lambda^+(c) > \Lambda^-(d) \geq 0$ (b) $\Lambda^+(c) > 0 > \Lambda^-(d)$ (c) $0 \geq \Lambda^+(c) > \Lambda^-(d)$

Approximating $D[o, d]$: Quality

- **Maximum Absolute Error:** A crucial quantity both for the time-complexity and for the space-complexity of the algorithm:



(a) $\Lambda^+(c) > \Lambda^-(d) \geq 0$ (b) $\Lambda^+(c) > 0 > \Lambda^-(d)$ (c) $0 \geq \Lambda^+(c) > \Lambda^-(d)$

LEMMA: Closed Form of Maximum Absolute Error (Kontogianis-Zaroliagis (2014))

$$MAE(c, d) = (\Lambda^+(c) - \Lambda^-(d)) \cdot \frac{(m-c)\cdot(d-m)}{L} \leq \frac{L\cdot(\Lambda^+(c) - \Lambda^-(d))}{4}$$

Approximating $D[o, d]$: Basic Idea (I)

- Approximations of $D[o, d]$: For given $\varepsilon > 0$, and $\forall t \in [0, T)$,

$$\boxed{\boxed{\underline{D}[o,d](t) \leq D[o,d](t) \leq \overline{D}[o,d](t) \leq (1+\varepsilon)\cdot \underline{D}[o,d](t)}}$$

Approximating $D[o,d]$: Basic Idea (I)

- Approximations of $D[o,d]$: For given $\varepsilon > 0$, and $\forall t \in [0,T)$,

$$\boxed{\boxed{\underline{D}[o,d](t) \leq D[o,d](t) \leq \overline{D}[o,d](t) \leq (1+\varepsilon)\cdot\underline{D}[o,d](t)}}$$

- FACT: **if** $D[o,d]$ was a priori known **then** a **linear scan** would give a space-optimal $(1+\varepsilon)$−upper-approximation (i.e., with the MIN #BPs).

Approximating $D[o,d]$: Basic Idea (I)

- Approximations of $D[o,d]$: For given $\varepsilon > 0$, and $\forall t \in [0,T)$,

$$\boxed{\boxed{\underline{D}[o,d](t) \leq D[o,d](t) \leq \overline{D}[o,d](t) \leq (1+\varepsilon)\cdot \underline{D}[o,d](t)}}$$

- FACT: **if** $D[o,d]$ was a priori known **then** a **linear scan** would give a space-optimal $(1+\varepsilon)-$upper-approximation (i.e., with the MIN #BPs).

- PROBLEM: Prohibitively expensive to compute/store $D[o,d]$ before approximating it. We must be based only on a few **samples** of $D[o,d]$.

Approximating $D[o, d]$: Basic Idea (I)

- Approximations of $D[o, d]$: For given $\varepsilon > 0$, and $\forall t \in [0, T)$,

$$\boxed{\underline{D}[o, d](t) \le D[o, d](t) \le \overline{D}[o, d](t) \le (1 + \varepsilon) \cdot \underline{D}[o, d](t)}$$

- FACT: **if** $D[o, d]$ was a priori known **then** a **linear scan** would give a space-optimal $(1 + \varepsilon)-$upper-approximation (i.e., with the MIN #BPs).

- PROBLEM: Prohibitively expensive to compute/store $D[o, d]$ before approximating it. We must be based only on a few **samples** of $D[o, d]$.

- FOCUS: **Linear** arc-delays. Later extend to pwl arc-delays.

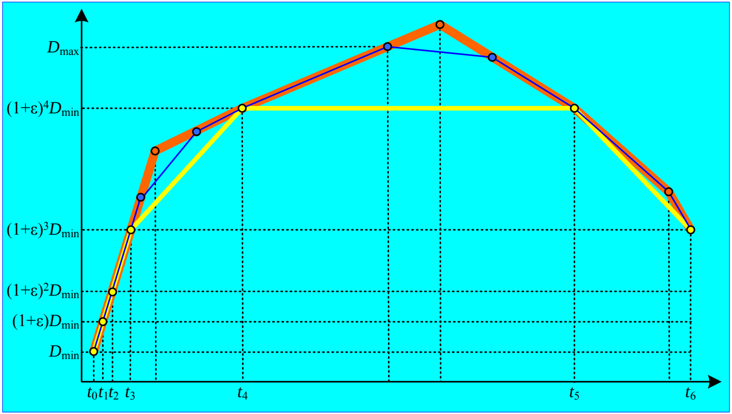
Approximating $D[o,d]$: Basic Idea (I)

- Approximations of $D[o,d]$: For given $\varepsilon > 0$, and $\forall t \in [0,T)$,

$$\boxed{\underline{D}[o,d](t) \le D[o,d](t) \le \overline{D}[o,d](t) \le (1+\varepsilon)\cdot \underline{D}[o,d](t)}$$

- FACT: **if** $D[o,d]$ was a priori known **then** a **linear scan** would give a space-optimal $(1+\varepsilon)$–upper-approximation (i.e., with the MIN #BPs).
- PROBLEM: Prohibitively expensive to compute/store $D[o,d]$ before approximating it. We must be based only on a few **samples** of $D[o,d]$.
- FOCUS: **Linear** arc-delays. Later extend to pwl arc-delays.
- $D[o,d]$ lies entirely in a **bounding box** that we can easily determine, with only 3 TD-Djikstra probes.

Approximating $D[o,d]$: Basic Idea (II)



- Make the sampling so that $\forall t \in [0, T],\ \overline{D}[o,d](t) \leq (1+\varepsilon) \cdot \underline{D}[o,d](t)$.
- Keep sampling always the **fastest-growing axis** wrt to $D[o,d]$.

One-To-One Approximation: PHASE-1

(Foschini-Hershberger-Suri (2011))

while slope of $D[o,d] \geq 1$ **do**

$$t_0 \xrightarrow{\text{Forward Dijkstra}} t_0 + D[o,d](t_0)$$

$$\downarrow$$

$$t_1 \xleftarrow{\text{Backward Dijkstra}} t_0 + (1+\varepsilon)^{1/2} D[o,d](t_0) = t_1 + D[o,d](t_1)$$

$$\downarrow$$

$$t_2 \xleftarrow{\text{Backward Dijkstra}} t_1 + (1+\varepsilon)^{1/2} D[o,d](t_1) = t_2 + D[o,d](t_2)$$

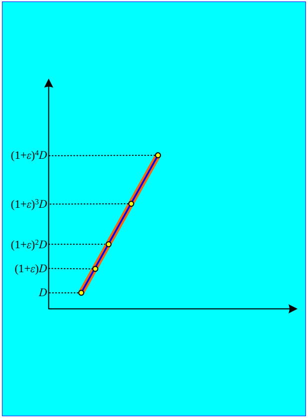
$$\vdots$$

One-To-One Approximation: PHASE-1

(Foschini-Hershberger-Suri (2011))

while slope of $D[o,d] \geq 1$ **do**

Bad Case for (Foschini-Hersberger-Suri (2011)) :



One-To-One Approximation: PHASE-1

(Foschini-Hershberger-Suri (2011))

while slope of $D[o,d] \geq 1$ **do**

ontogiannis-Zaroliagis (2013)) :

$t_0 \xrightarrow{\text{Forward Dijkstra}} t_0 + D[o,d](t_0)$

$t_{1,1} \xleftarrow{\text{Backward Dijkstra}} \mathrm{MAX}\{\, t_{fail},\ t_0 + (1+\varepsilon)\, D[o,d](t_0) \,\}$ *MaxAbsError* $< \varepsilon\, D[o,d](t_0)$

o
o
o

$t_1 = t_{1,k} \xleftarrow{\text{Backward Dijkstra}} t_0 + (1+\varepsilon)^k D[o,d](t_0) = t_1 + D[o,d](t_1)$ *MaxAbsError* $< \varepsilon\, D[o,d](t_0)$

$t_{1,k+1} \xleftarrow{\text{Backward Dijkstra}} t_0 + (1+\varepsilon)^{k+1} D[o,d](t_0)$ *MaxAbsError* $> \varepsilon\, D[o,d](t_0)$

$t_{2,1} \xleftarrow{\text{Backward Dijkstra}} \mathrm{MAX}\{\, t_{fail},\ t_1 + (1+\varepsilon)\, D[o,d](t_1) \,\}$ *MaxAbsError* $< \varepsilon\, D[o,d](t_1)$

o
o
o

One-To-One Approximation: PHASE-2

(Foschini-Hershberger-Suri (2011))

Slope of $D[o, d] \leq 1$:

repeat

Apply **BISECTION** to the remaining time-interval(s)

until desired approximation guarantee (wrt **Max Absolute Error**) is achieved.

One-To-All Approximaton via Bisection (I)

(Kontogiannis-Zaroliagis (2013))

ASSUMPTION 1: Concavity of arc-delays. /* to be removed later */

- Implies concavity of the *unknown* function $D[o, d]$.

One-To-All Approximaton via Bisection (I)

(Kontogiannis-Zaroliagis (2013))

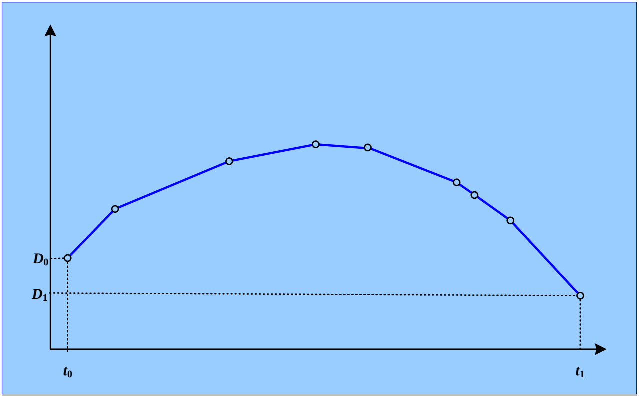
ASSUMPTION 1: Concavity of arc-delays. /* to be removed later */

- Implies concavity of the *unknown* function $D[o, d]$.

ASSUMPTION 2: Bounded Travel-Time Slopes. Small slopes of the (pwl) arc-delay functions.

- **Verified** by TD-traffic data for road network of Berlin (TomTom (February 2013)) that all arc-delay slopes are in $[-0.5, 0.5]$.
- Slopes of *shortest-travel-time* function $D[o, d]$ from $[-\Lambda_{min}, \Lambda_{max}]$, for some constants $\Lambda_{max} > 0,\ \Lambda_{min} \in [0, 1)$.

Under ASSUMPTIONS 1-2: Execute `Bisection` to *sample simultaneously* all distance values from *o*, at mid-points of time intervals, until required approximation guarantee is achieved *for each destination node*.

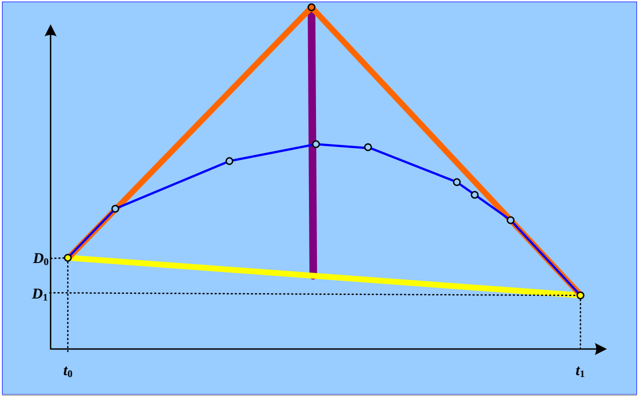


Example of Bisection Execution : INPUT = **UNKNOWN BLUE** function

One-To-All Approximaton via Bisection (II)

(Kontogiannis-Zaroliagis (2013))

Under ASSUMPTIONS 1-2: Execute `Bisection` to *sample simultaneously* all distance values from o, at mid-points of time intervals, until required approximation guarantee is achieved *for each destination node*.

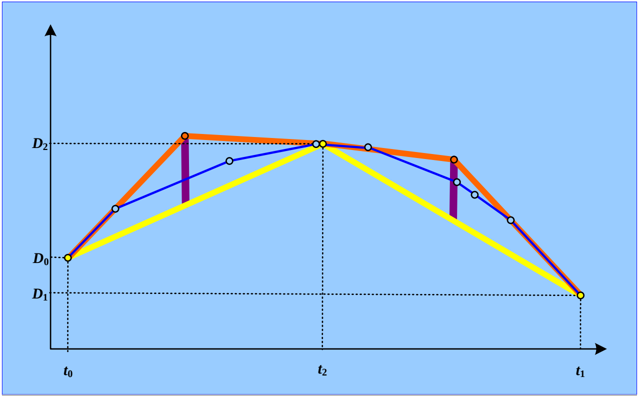


Example of Bisection Execution : **ORANGE** = Upper Bound, **YELLOW** = Lower Bound

One-To-All Approximaton via Bisection (II)

(Kontogiannis-Zaroliagis (2013))

Under ASSUMPTIONS 1-2: Execute `Bisection` to *sample simultaneously* all distance values from o, at mid-points of time intervals, until required approximation guarantee is achieved *for each destination node*.

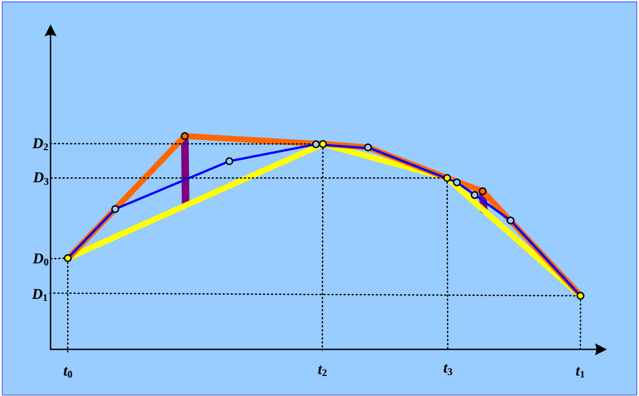


Example of Bisection Execution : Level-1 Recursion

One-To-All Approximaton via Bisection (II)

(Kontogiannis-Zaroliagis (2013))

Under ASSUMPTIONS 1-2: Execute `Bisection` to *sample simultaneously* all distance values from o, at mid-points of time intervals, until required approximation guarantee is achieved *for each destination node*.



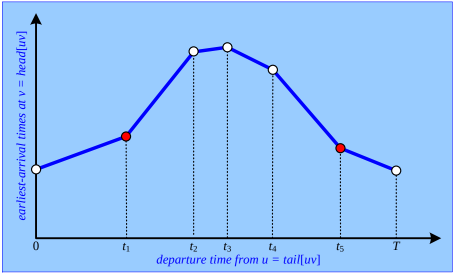
Example of Bisection Execution : Level-2 Recursion

One-To-All Approximaton via Bisection (III)

(Kontogiannis-Zaroliagis (2013))

Only under ASSUMPTION 2: For continuous, pwl arc-delays.

1. Call `Reverse TD-Dijkstra` to project each concavity-spoiling PB to a PI of the origin o.
2. For each pair of *consecutive PIs* at o, run `Bisection` for the corresponding departure-times interval.



3. Return the *concatenation* of approximate distance summaries.

Approximating $D[o,d]$: Space/Time Complexity

THEOREM: Space Complexity (KZ (2014))

Let K^* be the total number of concavity-spoiling BPs among all the arc-delay functions in the instance.

Space Complexity: For a *given orign* $o \in V$ and *all* possible destinations $d \in V$, the following complexity bounds hold for creating all the approximation functions $\overline{D}[o,\star] = (\overline{D}[o,d])_{d\in V}$:

1. $\mathrm{O}\left(\frac{K^*}{\mathrm{e}} \log\left(\frac{D_{\max}[o,\star](0,T)}{D_{\min}[o,\star](0,T)}\right)\right)$
2. In each interval of *consecutive* PIs,
   $|UBP[o,d]| \leq 4 \cdot$ (minimum #BPs for any $(1+\varepsilon)$−approximation.

Time Complexity: The number of *shortest-path probes* executed for the computation of the approximate distance functions is:

$$TDSP[o,d] \in \mathrm{O}\left(\log\left(\frac{T}{\varepsilon \cdot D_{\min}[o,d]}\right) \cdot \frac{K^*}{\varepsilon} \log\left(\frac{D_{\max}[o,\star](0,T)}{D_{\min}[o,\star](0,T)}\right)\right)$$

Implementation Issues wrt One-To-All Bisection

- `One-To-All Bisection` of (KZ (2014)) is a **label-setting** approximation method that provably works *space/time optimally* (within constant factors) wrt **concave** continuous pwl arc-delay functions.

Implementation Issues wrt One-To-All Bisection

- 🙂 `One-To-All Bisection` of (KZ (2014)) is a **label-setting** approximation method that provably works *space/time optimally* (within constant factors) wrt **concave** continuous pwl arc-delay functions.

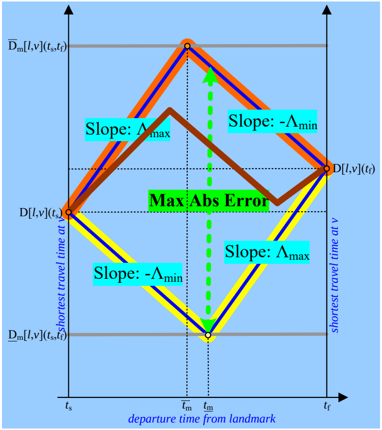
- 🙁 Both `One-To-One Approximation` of (FHS (2011)) and `One-To-All Bisection` of (KZ (2014)) suffer from **linear dependence** in the degree of disconcavity (value of K^*) in the TD Instance.

Implementation Issues wrt One-To-All Bisection

- `One-To-All Bisection` of (KZ (2014)) is a **label-setting** approximation method that provably works *space/time optimally* (within constant factors) wrt **concave** continuous pwl arc-delay functions.
- Both `One-To-One Approximation` of (FHS (2011)) and `One-To-All Bisection` of (KZ (2014)) suffer from **linear dependence** in the degree of disconcavity (value of K^*) in the TD Instance.
- A novel **one-to-all** (again **label-setting**) approximation technique, called the `Trapezoidal` method ((KWZ (2016))) avoids entirely the dependence of the required space from the network structure (and, of course, the degree of disconcavity).

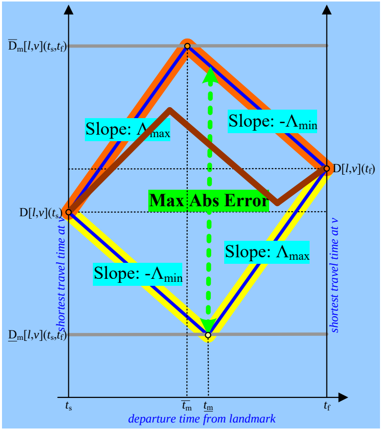
The Trapezoidal One-To-All Approximation Method

- **Sample travel-times** to all destinations, from coarser to finer departure-times from the (common) origin.
- Between consecutive samples of the same resolution, the unknown function is *bounded within a given trapezoidal*.
- ``Freeze'' destinations within intervals with satisfactory approximation guarantee.



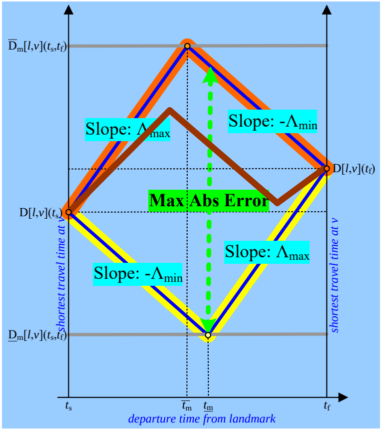
The Trapezoidal One-To-All Approximation Method

- **Sample travel-times** to all destinations, from coarser to finer departure-times from the (common) origin.
- Between consecutive samples of the same resolution, the unknown function is *bounded within a given trapezoidal*.
- ``Freeze'' destinations within intervals with satisfactory approximation guarantee.



Avoids dependence on concavity-spoiling BPs of the metric.

- **Sample travel-times** to all destinations, from coarser to finer departure-times from the (common) origin.
- Between consecutive samples of the same resolution, the unknown function is *bounded within a given trapezoidal*.
- ``Freeze'' destinations within intervals with satisfactory approximation guarantee.



☺ Avoids dependence on concavity-spoiling BPs of the metric.

☹ Cannot provide good approximations for ``nearby'' destinations around the origin.

Time-Dependent Oracles

Distance Oracles

- Extremely successful theme in static graphs.
  - In theory:
    - P-Space: Subquadratic (sometimes quasi-linear).
    - Q-Time: Constant.
    - Stretch: Small (sometimes PTAS).
  - In practice:
    - P-Space: A few GBs (sometimes less than 1 GB).
    - Q-Time: Miliseconds (sometimes microseconds).
    - Stretch: Exact distances (in most cases).

Distance Oracles

- Extremely successful theme in static graphs.
  - In theory:
    - ★ P-Space: Subquadratic (sometimes quasi-linear).
    - ★ Q-Time: Constant.
    - ★ Stretch: Small (sometimes PTAS).
  - In practice:
    - ★ P-Space: A few GBs (sometimes less than 1 GB).
    - ★ Q-Time: Miliseconds (sometimes microseconds).
    - ★ Stretch: Exact distances (in most cases).
- Some practical algorithms extended to time-dependent case.

Distance Oracles

- Extremely successful theme in static graphs.
  - In theory:
    - P-Space: Subquadratic (sometimes quasi-linear).
    - Q-Time: Constant.
    - Stretch: Small (sometimes PTAS).
  - In practice:
    - P-Space: A few GBs (sometimes less than 1 GB).
    - Q-Time: Miliseconds (sometimes microseconds).
    - Stretch: Exact distances (in most cases).
- Some practical algorithms extended to time-dependent case.

FOR THE REST OF THE TALK

The focus is on **time-dependent oracles**, with **provably good** preprocessing-space / query-time / stretch tradeoffs.

CHALLENGE: Given a *large scale* graph with continuous, pwl, FIFO arc-delay functions, create a data structure (**oracle**) that requires reasonable *(subquadratic)* space and allows answering **distance queries** efficiently (in *sublinear* time).

CHALLENGE: Given a *large scale* graph with continuous, pwl, FIFO arc-delay functions, create a data structure (**oracle**) that requires reasonable *(subquadratic)* space and allows answering **distance queries** efficiently (in *sublinear* time).

- **Trivial solution:** Precompute all the $(1+\epsilon)-$approximate distance summaries from every origin to every destination.
  - ☹ $O(n^3)$ size ($O(n^2)$, if all arc-delay functions concave).
  - ☺ $O(\log\log(n))$ query time.
  - ☺ $(1+\epsilon)-$stretch.

CHALLENGE: Given a *large scale* graph with continuous, pwl, FIFO arc-delay functions, create a data structure (**oracle**) that requires reasonable *(subquadratic)* space and allows answering **distance queries** efficiently (in *sublinear* time).

- **Trivial solution:** Precompute all the $(1+\epsilon)$–approximate distance summaries from every origin to every destination.
  - $O(n^3)$ size ($O(n^2)$, if all arc-delay functions concave).
  - $O(\log\log(n))$ query time.
  - $(1+\epsilon)$–stretch.
- **Trivial solution:** No preprocessing, respond to queries by running `TD-Dijkstra`.
  - $O(n+m+K)$ size (K = total number of PBs of arc-delays).
  - $O([m+n\log(n)]\times\log\log(K))$ query time.
  - 1–stretch.

Distance Oracles

Is it a Success Story in Time-Dependent Graphs?

CHALLENGE: Given a *large scale* graph with continuous, pwl, FIFO arc-delay functions, create a data structure (**oracle**) that requires reasonable *(subquadratic)* space and allows answering **distance queries** efficiently (in *sublinear* time).

- **Trivial solution:** Precompute all the $(1+\epsilon)-$approximate distance summaries from every origin to every destination.
  - $\mathrm{O}(n^3)$ size ($\mathrm{O}(n^2)$, if all arc-delay functions concave).
  - $\mathrm{O}(\log\log(n))$ query time.
  - $(1+\epsilon)-$stretch.
- **Trivial solution:** No preprocessing, respond to queries by running `TD-Dijkstra`.
  - $\mathrm{O}(n+m+K)$ size (K = total number of PBs of arc-delays).
  - $\mathrm{O}([m+n\log(n)]\times\log\log(K))$ query time.
  - $1-$stretch.

Is there a smooth tradeoff among space / query time / stretch?

FLAT TD-Oracle

Landmark Selection Policy

- Rationale: Identify a few ``important'' vertices (**landmarks**) in the network, which are assumed to be crucial for almost all shortest paths. Then compute approximate **travel-time summaries** (functions) $\Delta[\ell, v](t)$, $\forall(\ell, v) \in L \times V,\ \forall t \in [0, T)$ s.t.:

$$\boxed{D[\ell, v](t) \leq \Delta[\ell, v](t) \leq (1+\epsilon) \cdot D[\ell, v](t)}$$

Landmark Selection Policy

- **Rationale:** Identify a few ``important'' vertices (**landmarks**) in the network, which are assumed to be crucial for almost all shortest paths. Then compute approximate **travel-time summaries** (functions) $\Delta[\ell, v](t)$, $\forall(\ell, v) \in L \times V,\ \forall t \in [0, T)$ s.t.:

$$\boxed{D[\ell, v](t) \leq \Delta[\ell, v](t) \leq (1 + \epsilon) \cdot D[\ell, v](t)}$$

- **In theory:** Choose landmarks **independently and uniformly at random**.
- **In practice:** Several options.
  - Random Selection (R). (KMPPWZ (2015))
  - METIS Selection (M). (KMPPWZ (2015))
  - KaHIP Selection (K). (KMPPWZ (2015))

Landmark Selection Policy

- **Rationale:** Identify a few "important" vertices (**landmarks**) in the network, which are assumed to be crucial for almost all shortest paths. Then compute approximate **travel-time summaries** (functions) $\Delta[\ell, v](t)$, $\forall(\ell, v) \in L \times V,\ \forall t \in [0, T)$ s.t.:

$$\boxed{D[\ell, v](t) \leq \Delta[\ell, v](t) \leq (1+\epsilon) \cdot D[\ell, v](t)}$$

- **In theory:** Choose landmarks **independently and uniformly at random**.
- **In practice:** Several options.
  - Random Selection (R). (KMPPWZ (2015))
  - METIS Selection (M). (KMPPWZ (2015))
  - KaHIP Selection (K). (KMPPWZ (2015))
  - Important-Random Selection (IR). (KMPPWZ (2016))
  - Sparse-Random Selection (SR). (KMPPWZ (2016))
  - Hybrid Selection (H). (KMPPWZ (2016))

Landmark Selection Policy

- **Rationale:** Identify a few ``important'' vertices (**landmarks**) in the network, which are assumed to be crucial for almost all shortest paths. Then compute approximate **travel-time summaries** (functions) $\Delta[\ell, v](t)$, $\forall(\ell, v) \in L \times V,\ \forall t \in [0, T)$ s.t.:

$$\boxed{D[\ell, v](t) \le \Delta[\ell, v](t) \le (1+\epsilon) \cdot D[\ell, v](t)}$$

- **In theory:** Choose landmarks **independently and uniformly at random**.
- **In practice:** Several options.
  - Random Selection (R). (KMPPWZ (2015))
  - METIS Selection (M). (KMPPWZ (2015))
  - KaHIP Selection (K). (KMPPWZ (2015))
  - Important-Random Selection (IR). (KMPPWZ (2016))
  - Sparse-Random Selection (SR). (KMPPWZ (2016))
  - Hybrid Selection (H). (KMPPWZ (2016))
  - Betweeness-Centrality Selection (BC). (KPPWZ (2017))

Preprocessing of FLAT

(KZ (2014), KMPPWZ2015, KMPPWZ2016)

- Each landmark is **informed** about all destinations.
- **Subquadratic** preprocessing space/time.
- Query time **sublinear** in the network size.
- Constant approximation, or even **PTAS**.

Preprocessing of FLAT

(KZ (2014), KMPPWZ2015, KMPPWZ2016)

- Each landmark is **informed** about all destinations.
- **Subquadratic** preprocessing space/time.
- Query time **sublinear** in the network size.
- Constant approximation, or even **PTAS**.

Preprocessing Complexity of FLAT

When the landmark set $L \subset V$ is chosen **uniformly at random**:

(KZ (2014)) Subquadratic preprocessing time and space, when BIS is used and the *degree of disconcavity* K^* is not too large: $K^* \cdot |L| \in o(n)$.

Preprocessing of FLAT

(KZ (2014), KMPPWZ2015, KMPPWZ2016)

- Each landmark is **informed** about all destinations.
- **Subquadratic** preprocessing space/time.
- Query time **sublinear** in the network size.
- Constant approximation, or even **PTAS**.

Preprocessing Complexity of FLAT

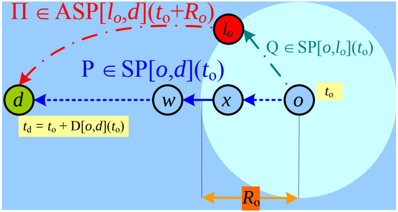
When the landmark set $L \subset V$ is chosen **uniformly at random**:

(KZ (2014)) Subquadratic preprocessing time and space, when BIS is used and the *degree of disconcavity* K^* is not too large: $K^* \cdot |L| \in \mathrm{o}(n)$.

(KWZ-2016) If each vertex becomes a landmark with probability $\rho = n^{-\delta}$, BIS is used for $F = \sqrt{n}$ *``nearby''* destinations and TRAP is used for the rest *``faraway''* destinations from each landmark, then the preprocessing space and time are $\mathrm{O}\left(n^{2-\delta} \cdot \mathrm{polylog}(n)\right)$.

Forward Constant Approximation (FCA)

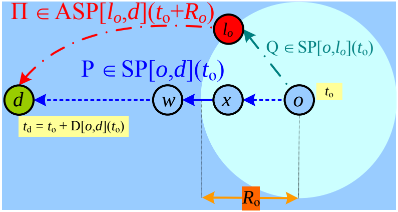
1. Grow TD-Dijkstra ball $B(o, t_o)$ until the closest landmark ℓ_o, or d, is settled
2. **return** $sol_o = D[o, \ell_o](t_o) + \Delta[\ell_o, d](t_o + D[o, \ell_o](t_o))$



FCA: A Constant-Approximation Query Algorithm (KZ (2014))

Forward Constant Approximation (FCA)

1. Grow TD-Dijkstra ball $B(o, t_o)$ until the closest landmark ℓ_o, or d, is settled
2. **return** $sol_o = D[o, \ell_o](t_o) + \Delta[\ell_o, d](t_o + D[o, \ell_o](t_o))$



Complexity of FCA for random landmarks

- Constant approximation guarantee: $sol_o \le (1 + \epsilon + \psi) \cdot D[o, d](t_o)$, for $\psi = 1 + \Lambda_{\max}(1 + \epsilon)(1 + 2\zeta + \Lambda_{\max}\zeta) + (1 + \epsilon)\zeta \in O(1)$.
- Sublinear Query-time: $O\left(\frac{1}{\rho} \cdot \ln\left(\frac{1}{\rho}\right) \log\log(K_{\max})\right)$

FCA+: A natural extension of FCA

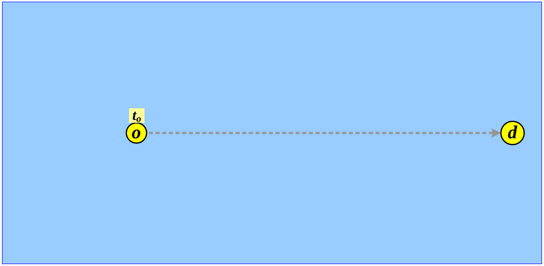
(KMPPWZ (2015-2016))

Extended Forward Constant Approximation (FCA+)

1. Grow TD-Dijkstra ball $B(o, t_o)$ until the N closest landmarks $\ell_o, \ldots, \ell_{N-1}$ (or d) are settled.
2. **return** $\min_{i \in \{0,1,\ldots,N-1\}} \{ sol_i = D[o, \ell_i](t_o) + \Delta[\ell_i, d](t_i + D[o, \ell_i](t_o)) \}$

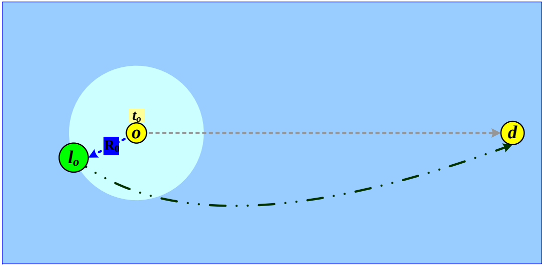
Extended Forward Constant Approximation (FCA+)

1. Grow TD-Dijkstra ball $B(o, t_o)$ until the N closest landmarks $\ell_o, \ldots, \ell_{N-1}$ (or d) are settled.
2. **return** $\min_{i \in \{0,1,\ldots,N-1\}} \{ sol_i = D[o, \ell_i](t_o) + \Delta[\ell_i, d](t_i + D[o, \ell_i](t_o)) \}$



Extended Forward Constant Approximation (FCA+)

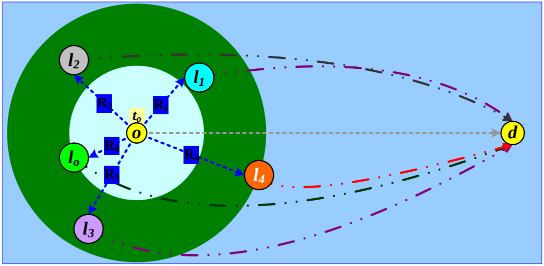
1. Grow TD-Dijkstra ball $B(o, t_o)$ until the N closest landmarks $\ell_o, \ldots, \ell_{N-1}$ (or d) are settled.
2. **return** $\min_{i \in \{0,1,\ldots,N-1\}} \{ sol_i = D[o,\ell_i](t_o) + \Delta[\ell_i, d](t_i + D[o,\ell_i](t_o)) \}$



FCA+: A natural extension of FCA

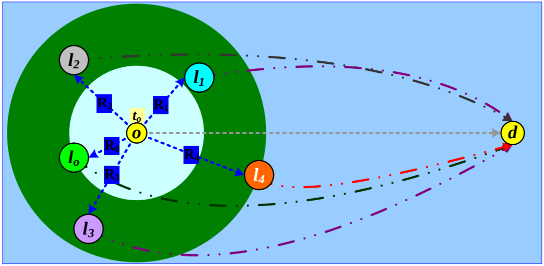
Extended Forward Constant Approximation (FCA+)

1. Grow TD-Dijkstra ball $B(o, t_o)$ until the N closest landmarks $\ell_o, \ldots, \ell_{N-1}$ (or d) are settled.
2. **return** $\min_{i \in \{0,1,\ldots,N-1\}} \{ sol_i = D[o, \ell_i](t_o) + \Delta[\ell_i, d](t_i + D[o, \ell_i](t_o)) \}$



Extended Forward Constant Approximation (FCA+)

1. Grow TD-Dijkstra ball $B(o,t_o)$ until the N closest landmarks $\ell_o,\ldots,\ell_{N-1}$ (or d) are settled.
2. **return** $\min_{i\in\{0,1,\ldots,N-1\}}\{\ sol_i = D[o,\ell_i](t_o) + \Delta[\ell_i,d](t_i + D[o,\ell_i](t_o))\ \}$



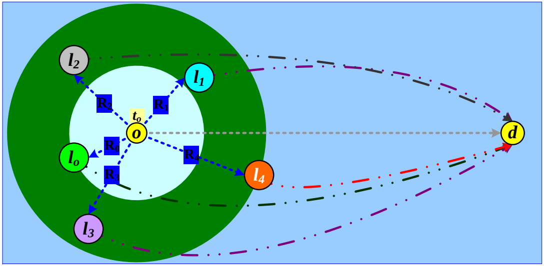
Performance of FCA+ for random landmarks

- In theory: Analogous to that of FCA.

FCA+: A natural extension of FCA

Extended Forward Constant Approximation (FCA+)

1. Grow TD-Dijkstra ball $B(o, t_o)$ until the N closest landmarks $\ell_o, \ldots, \ell_{N-1}$ (or d) are settled.
2. **return** $\min_{i \in \{0,1,\ldots,N-1\}} \{ sol_i = D[o,\ell_i](t_o) + \Delta[\ell_i, d](t_i + D[o,\ell_i](t_o)) \}$



Performance of FCA+ for random landmarks

- In theory: Analogous to that of FCA.
- In practice: Performance analogous to (indeed, better than) that of RQA.

RQA: A Sublinear-Time Approximation Scheme (KZ (2014))

Recursive Query Approximation (RQA)

while recursion budget R not exhausted **do**

  Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled

  $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$

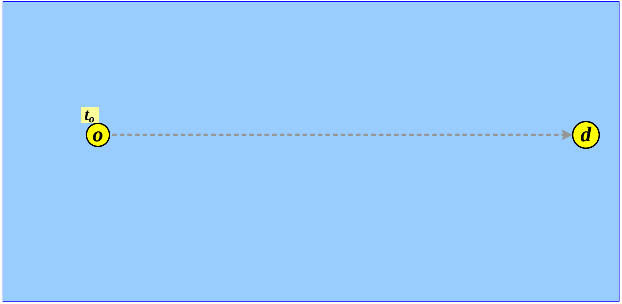
  Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$

end while

return best solution found

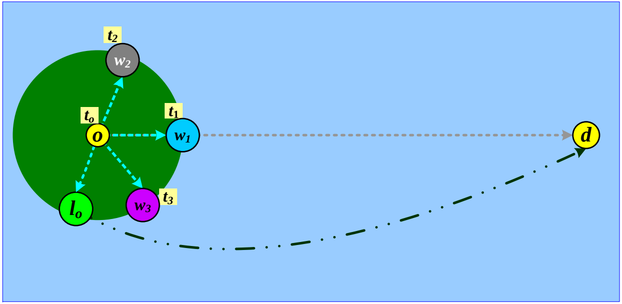
Recursive Query Approximation (RQA)

1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found



Recursive Query Approximation (RQA)

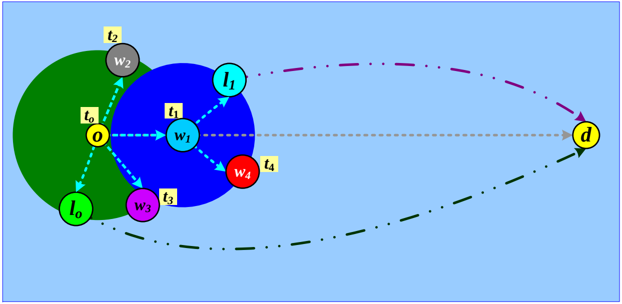
1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found



- Growing level-0 ball...

Recursive Query Approximation (RQA)

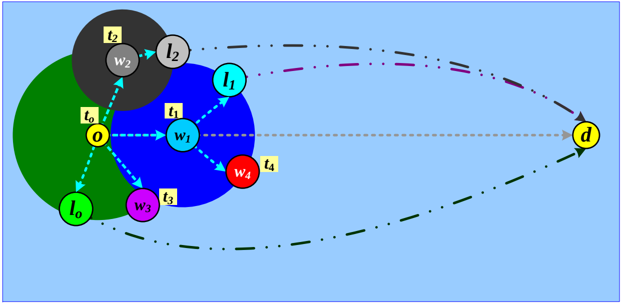
1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found



- Growing level-0 ball...
- Growing level-1 balls...

Recursive Query Approximation (RQA)

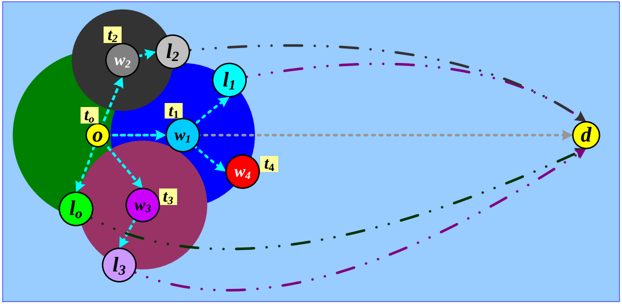
1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found



- Growing level-0 ball...
- Growing level-1 balls...

Recursive Query Approximation (RQA)

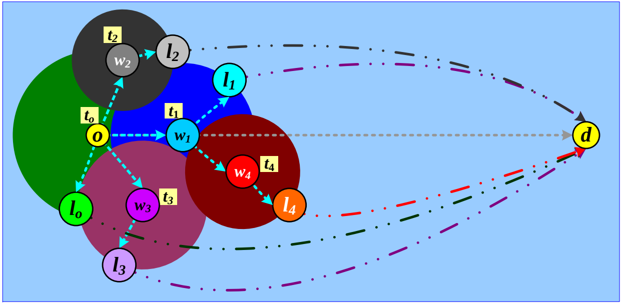
while recursion budget R not exhausted **do**
    Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
    $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
    Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
end while
return best solution found



- Growing level-0 ball...
- Growing level-1 balls...

Recursive Query Approximation (RQA)

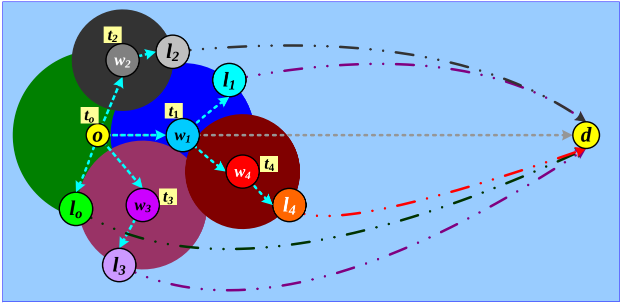
1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found



- Growing level-0 ball...
- Growing level-1 balls...
- Growing level-2 balls...

Recursive Query Approximation (RQA)

1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found



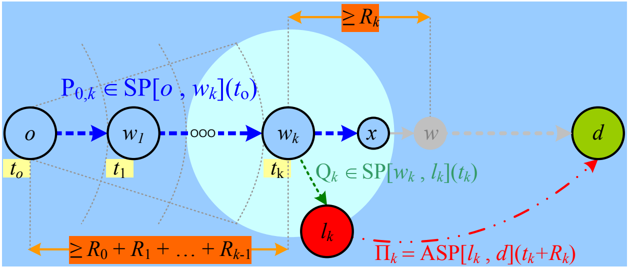
- Growing level-0 ball...
- Growing level-1 balls...
- Growing level-2 balls...
- ...

Recursive Query Approximation (RQA)

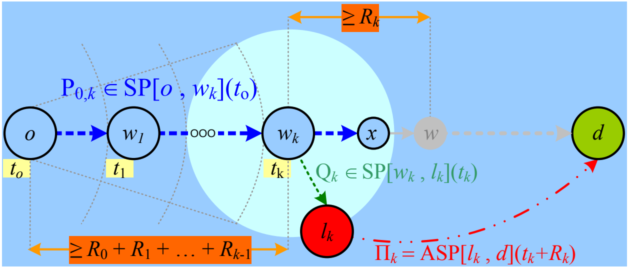
1. **while** recursion budget R not exhausted **do**
2. Grow TD-Dijkstra ball $B(w_i, t_i)$ until closest landmark ℓ_i is settled
3. $sol_i = D[o, w_i](t_o) + D[w_i, \ell_i](t_i) + \Delta[\ell_i, d](t_i + D[w_i, \ell_i](t_i))$
4. Run RQA at *each boundary node* of $B(w_i, t_i)$ with budget $R - 1$
5. **end while**
6. **return** best solution found

Complexity of RQA for random landmarks

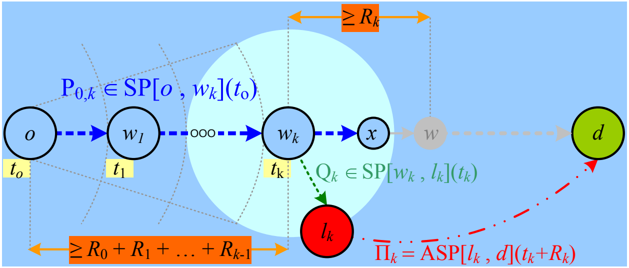
- PTAS: $sol \leq (1 + \sigma) \cdot D[o, d](t_o)$, for $\sigma = \varepsilon \cdot \frac{(1+\varepsilon/\psi)^{R+1}}{(1+\varepsilon/\psi)^{R+1}-1}$ and $R \in \mathrm{O}(1)$.
- Sublinear Query-time: $\mathrm{O}\left(\left(\frac{1}{\rho}\right)^{R+1} \cdot \ln\left(\frac{1}{\rho}\right) \log\log(K_{\max})\right)$


$\geq R_k$
$P_{0,k} \in SP[o , w_k](t_o)$
o
w_1
w_k
x
w
d
t_o
t_1
t_k
$Q_k \in SP[w_k , l_k](t_k)$
l_k
$\geq R_0 + R_1 + \ldots + R_{k-1}$
$\Pi_k = ASP[l_k , d](t_k+R_k)$

RQA: Boosting The Approximation Guarantee Of FCA (KZ (2014))



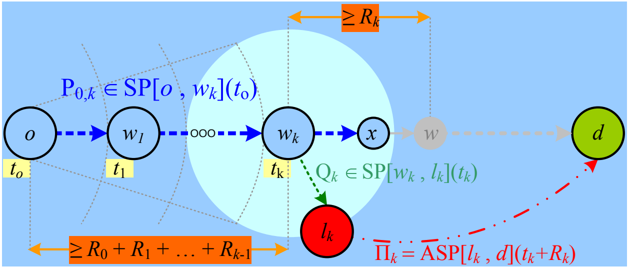
1. One of the discovered approximate *od*−paths has all its ball centers at *nodes of the (unknown) shortest od-path*.



1. One of the discovered approximate *od*−paths has all its ball centers at *nodes of the (unknown) shortest od-path*.
2. Optimal **prefix subpaths** improve approximation guarantee:

$$\boxed{\forall \beta > 1,\ \forall \lambda \in (0,1),\ \lambda \cdot OPT + (1-\lambda)\cdot \beta \cdot OPT < \beta \cdot OPT}$$

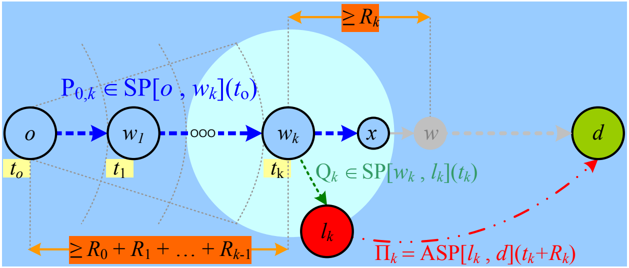
RQA: Boosting The Approximation Guarantee Of FCA (KZ (2014))



1. One of the discovered approximate *od*−paths has all its ball centers at *nodes of the (unknown) shortest od-path*.
2. Optimal **prefix subpaths** improve approximation guarantee:

$$\boxed{\forall\beta > 1,\ \forall\lambda \in (0,1),\ \lambda \cdot OPT + (1-\lambda)\cdot\beta\cdot OPT < \beta\cdot OPT}$$

3. Approximation guarantee for **suffix subpath** to destination depends on *last ball radius*.



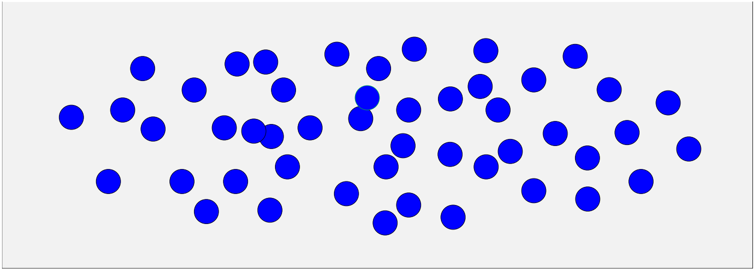
1. One of the discovered approximate *od*−paths has all its ball centers at *nodes of the (unknown) shortest od-path*.
2. Optimal **prefix subpaths** improve approximation guarantee:

$$\boxed{\forall\beta > 1,\ \forall\lambda \in (0,1),\ \lambda \cdot OPT + (1-\lambda)\cdot\beta\cdot OPT < \beta\cdot OPT}$$

3. Approximation guarantee for **suffix subpath** to destination depends on *last ball radius*.
4. $R = O(1)$ recursion budget suffices to ensure guarantee close to $1+\varepsilon$.

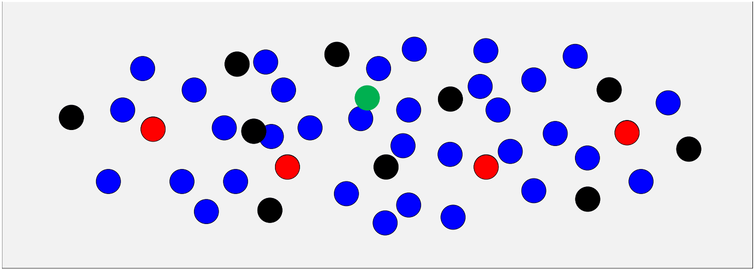
HORN Oracle

HORN Preprocessing: Hierarchy of Landmarks (I) (KWZ (2016))

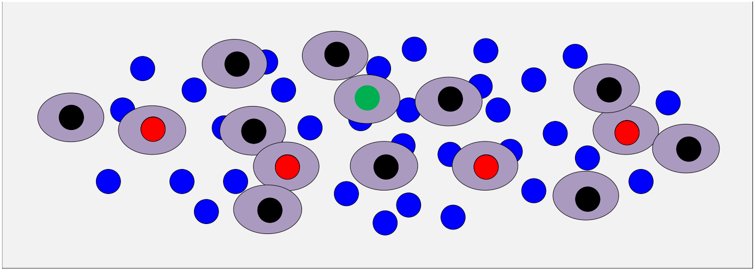


HORN Preprocessing: Hierarchy of Landmarks (I) (KWZ (2016))

- Selection of landmark sets (colors indicate sizes of coverages).

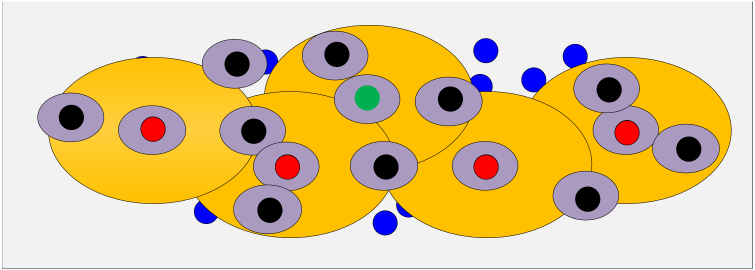


- Selection of landmark sets (colors indicate sizes of coverages).
- **Small-coverage** landmarks ``learn'' travel-time functions to their (only short-range) destinations.



- Selection of landmark sets (colors indicate sizes of coverages).
- **Small-coverage** landmarks ``learn´´ travel-time functions to their (only short-range) destinations.
- **Medium-coverage** landmarks ``learn´´ travel-time functions to their (up to medium-range) destinations.

...

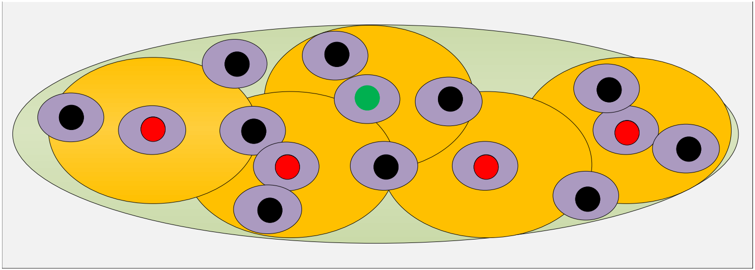


HORN Preprocessing: Hierarchy of Landmarks (I) (KWZ (2016))

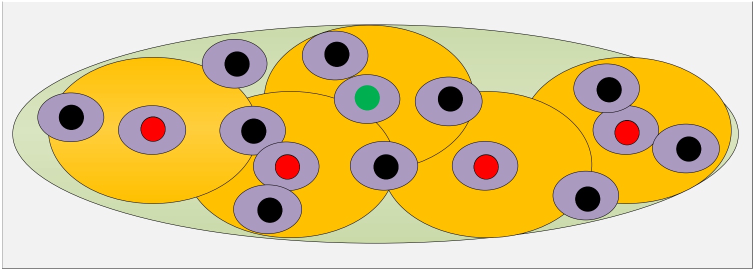
- Selection of landmark sets (colors indicate sizes of coverages).
- **Small-coverage** landmarks ``learn´´ travel-time functions to their (only short-range) destinations.
- **Medium-coverage** landmarks ``learn´´ travel-time functions to their (up to medium-range) destinations.

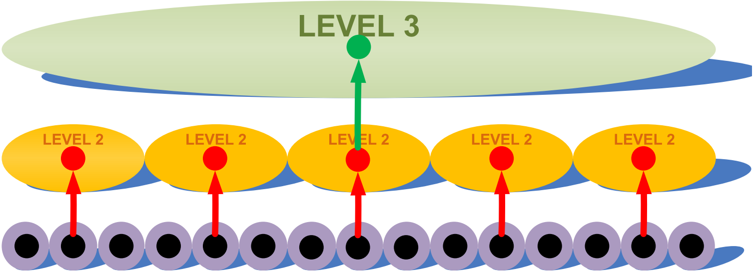
 . . .

- **Global-coverage** landmarks ``learn´´ travel-time functions to their (up to long-range) destinations.



HORN Preprocessing: Hierarchy of Landmarks (II) (KWZ (2016))





Creating Distance Summaries From Landmarks (KWZ (2016))

Preprocessing of HORN

- Depending on its level, each landmark has its own **coverage**, a given-size set of surrounding vertices for which it is *informed*.
- Exponentially decreasing sequence of *landmark set sizes*.
- exponentially increasing sequence of *coverages per landmark*

$\therefore$ $\boxed{O(\log\log(n)) \text{ levels}}$ $\Rightarrow$ **Subquadratic** preprocessing space/time.

- Depending on its level, each landmark has its own **coverage**, a given-size set of surrounding vertices for which it is *informed*.
- Exponentially decreasing sequence of *landmark set sizes*.
- exponentially increasing sequence of *coverages per landmark*

$\therefore$ $\boxed{O(\log\log(n)) \text{ levels}}$ $\Rightarrow$ **Subquadratic** preprocessing space/time.

Preprocessing Complexity of HORN (KWZ (2016))

An appropriate construction of the hierarchy assures preprocessing space and time $O\left(n^{2-\frac{\delta}{R+1}} \cdot \text{polylog}(n)\right)$, i.e., **subquadratic**. R is the *recursion budget* (depth), and $\delta \in (0,1)$ is the targeted exponent of sublinearity, for the query algorithm to be used (see next slides).

- Depending on its level, each landmark has its own **coverage**, a given-size set of surrounding vertices for which it is *informed*.
- Exponentially decreasing sequence of *landmark set sizes*.
- exponentially increasing sequence of *coverages per landmark*

$\therefore$ $\boxed{O(\log\log(n)) \text{ levels}}$ $\Rightarrow$ **Subquadratic** preprocessing space/time.

Preprocessing Complexity of HORN (KWZ (2016))

An appropriate construction of the hierarchy assures preprocessing space and time $O\left(n^{2-\frac{\delta}{R+1}} \cdot \mathrm{polylog}(n)\right)$, i.e., **subquadratic**. R is the *recursion budget* (depth), and $\delta \in (0, 1)$ is the targeted exponent of sublinearity, for the query algorithm to be used (see next slides).

NEXT: Query algorithm with constant approximation, or even **PTAS**, and query-time **sublinear** in the **Dijkstra Rank** of the query at hand.

HQA: The Query Algorithm of HORN

(KWZ (2016))

Rationale of the hierarchy...

level	targeted DR	Q-time	coverage	TRAP	Ring
1	$N_1 = n^{(\gamma-1)/\gamma}$	N_1^{δ}	$c_1 = N_1 \cdot n^{\xi_1}$	$\sqrt{c_1}$	$N_1^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
2	$N_2 = n^{(\gamma^2-1)/\gamma^2}$	N_2^{δ}	$c_2 = N_2 \cdot n^{\xi_2}$	$\sqrt{c_2}$	$N_2^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
$\vdots$					
k	$N_k = n^{(\gamma^k-1)/\gamma^k}$	N_k^{δ}	$c_k = N_k \cdot n^{\xi_k}$	$\sqrt{c_k}$	$N_k^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
k+1	$N_{k+1} = n$	n^{δ}	$c_{k+1} = n$	$\sqrt{n}$	$\left(N_k^{\delta/(R+1)} \cdot \ln(n), n\right]$

1. Mimic FLAT in each level i: All level-i landmarks are informed about c_i destinations around them.

Rationale of the hierarchy...

level	targeted DR	Q-time	coverage	TRAP	Ring
1	$N_1 = n^{(\gamma-1)/\gamma}$	N_1^δ	$c_1 = N_1 \cdot n^{\xi_1}$	$\sqrt{c_1}$	$N_1^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
2	$N_2 = n^{(\gamma^2-1)/\gamma^2}$	N_2^δ	$c_2 = N_2 \cdot n^{\xi_2}$	$\sqrt{c_2}$	$N_2^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
			$\vdots$		
k	$N_k = n^{(\gamma^k-1)/\gamma^k}$	N_k^δ	$c_k = N_k \cdot n^{\xi_k}$	$\sqrt{c_k}$	$N_k^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
k+1	$N_{k+1} = n$	n^δ	$c_{k+1} = n$	$\sqrt{n}$	$\left(N_k^{\delta/(R+1)} \cdot \ln(n), n\right]$

1. Mimic FLAT in each level i: All level-i landmarks are informed about c_i destinations around them.
2. The *density* of level-i landmarks is such that ALL queries of Dijkstra rank $\leq N_i$ can be answered by using ONLY level-i landmarks.

HQA: The Query Algorithm of HORN

(KWZ (2016))

Rationale of the hierarchy...

level	targeted DR	Q-time	coverage	TRAP	Ring
1	$N_1 = n^{(\gamma-1)/\gamma}$	N_1^δ	$c_1 = N_1 \cdot n^{\xi_1}$	$\sqrt{c_1}$	$N_1^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
2	$N_2 = n^{(\gamma^2-1)/\gamma^2}$	N_2^δ	$c_2 = N_2 \cdot n^{\xi_2}$	$\sqrt{c_2}$	$N_2^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
			$\vdots$		
k	$N_k = n^{(\gamma^k-1)/\gamma^k}$	N_k^δ	$c_k = N_k \cdot n^{\xi_k}$	$\sqrt{c_k}$	$N_k^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
k+1	$N_{k+1} = n$	n^δ	$c_{k+1} = n$	$\sqrt{n}$	$\left(N_k^{\delta/(R+1)} \cdot \ln(n), n\right]$

1. Mimic FLAT in each level i: All level-i landmarks are informed about c_i destinations around them.
2. The *density* of level-i landmarks is such that ALL queries of Dijkstra rank $\leq N_i$ can be answered by using ONLY level-i landmarks.
3. **FACT:** Running RQA at the **appropriate level** of the hierarchy would yield a good approximation.

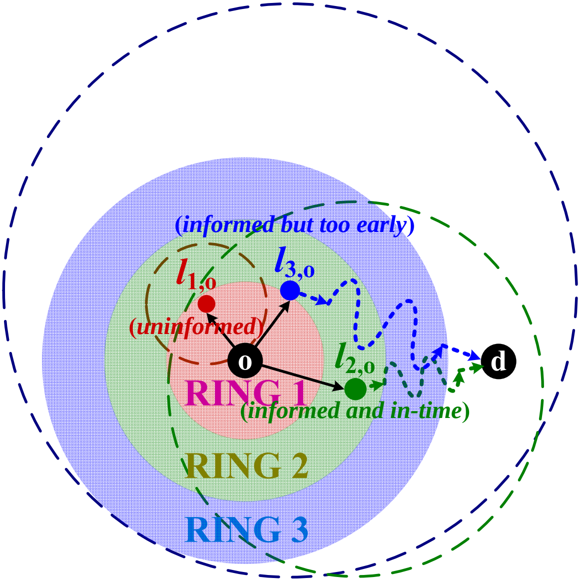
Rationale of the hierarchy...

level	targeted DR	Q-time	coverage	TRAP	Ring
1	$N_1 = n^{(\gamma-1)/\gamma}$	N_1^δ	$c_1 = N_1 \cdot n^{\xi_1}$	$\sqrt{c_1}$	$N_1^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
2	$N_2 = n^{(\gamma^2-1)/\gamma^2}$	N_2^δ	$c_2 = N_2 \cdot n^{\xi_2}$	$\sqrt{c_2}$	$N_2^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
			$\vdots$		
k	$N_k = n^{(\gamma^k-1)/\gamma^k}$	N_k^δ	$c_k = N_k \cdot n^{\xi_k}$	$\sqrt{c_k}$	$N_k^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, \ln(n)\right]$
k+1	$N_{k+1} = n$	n^δ	$c_{k+1} = n$	$\sqrt{n}$	$\left(N_k^{\delta/(R+1)} \cdot \ln(n), n\right]$

1. Mimic FLAT in each level i: All level-i landmarks are informed about c_i destinations around them.
2. The *density* of level-i landmarks is such that ALL queries of Dijkstra rank $\leq N_i$ can be answered by using ONLY level-i landmarks.
3. **FACT:** Running RQA at the **appropriate level** of the hierarchy would yield a good approximation.
4. **CHALLENGE:** *``Guess´´* the appropriate level, **whp** . Then, sublinearity in N_i (rather than n) can be achieved.

Guessing the appropriate level in the hierarchy...

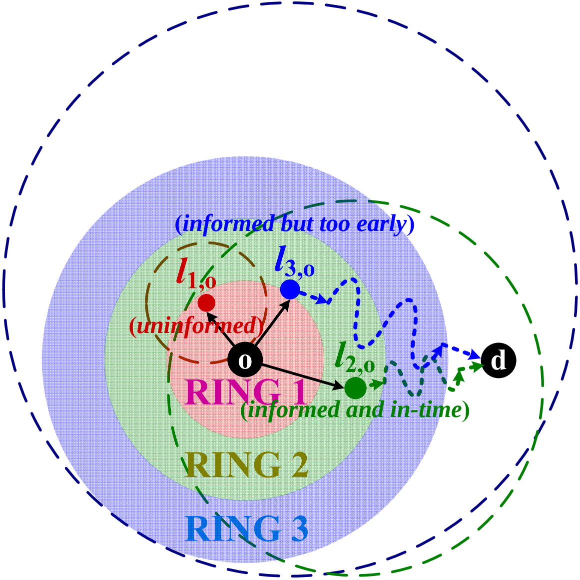
- level-1 landmark $\ell_{1,o}$ is **uninformed**.
- level-3 landmark $\ell_{3,o}$, although informed, came **too early**.
- level-2 landmark $\ell_{2,o}$ is **informed** and within the **right distance**.



Guessing the appropriate level in the hierarchy...

- level-1 landmark $\ell_{1,o}$ is **uninformed**.
- level-3 landmark $\ell_{3,o}$, although informed, came **too early**.
- level-2 landmark $\ell_{2,o}$ is **informed** and within the **right distance**.

∴ RQA will use only level-(≥ 2) landmarks from now on.



Hierarchical Query Algorithm (HQA)

1. Grow a unique TD-ball from (o, t_o), until the first **informed landmark** ℓ_o discovered *at the right distance* (not too close, not too far) from o.
2. (ESC) Interrupt the process if an informed landmark is discovered very close to the origin (already a good approximation).
3. (ALH) Execute an appropriate variant of RQA, using only landmarks of level at least as high as that of ℓ_o.
4. Return the best approximation, via all discovered informed landmarks.

Hierarchical Query Algorithm (HQA)

1. Grow a unique TD-ball from (o, t_o), until the first **informed landmark** ℓ_o discovered *at the right distance* (not too close, not too far) from o.
2. (ESC) Interrupt the process if an informed landmark is discovered very close to the origin (already a good approximation).
3. (ALH) Execute an appropriate variant of RQA, using only landmarks of level at least as high as that of ℓ_o.
4. Return the best approximation, via all discovered informed landmarks.

Performance of HQA for random landmarks

HORN can be fine-tuned so that it achieves **subquadratic** preprocessing space and time, and query-response time $\tilde{O}(N_i^\delta)$, i.e., **sublinear** in N_i, when $N_{i-1} < DR[o, d](t_o) \leq N_i$, with probability $1 - \mathrm{O}\left(\frac{1}{n}\right)$. The approximation guarantee is $1 + \varepsilon \cdot \frac{(1+\varepsilon/\psi)^{R+1}}{(1+\varepsilon/\psi)^{R+1}-1}$, where $R \leq \frac{2\delta}{\alpha} - 1$ is the recursion budget.

- ☹ Despite using only landmarks of the appropriate level (and above), RQA may **fail to provide approximate paths** via every landmark that it settles (some of them may be ``uninformed'').

HQA: The Query Algorithm of HORN (KWZ (2016))

Approximation guarantee of RQA (in FLAT) also holds for HQA...

- ☹ Despite using only landmarks of the appropriate level (and above), RQA may **fail to provide approximate paths** via every landmark that it settles (some of them may be ``uninformed'').

- ☺ By defining the **appropriate level** i according to the first landmark that is *both ``informed'' and at the ``right'' distance*, we can guarantee that the closest level-i landmark to subsequent ball centers along the unknown shortest path are **always informed**.

HQA: The Query Algorithm of HORN

(KWZ (2016))

Approximation guarantee of RQA (in FLAT) also holds for HQA...

- Despite using only landmarks of the appropriate level (and above), RQA may **fail to provide approximate paths** via every landmark that it settles (some of them may be ``uninformed'').

- By defining the **appropriate level** *i* according to the first landmark that is *both ``informed'' and at the ``right'' distance*, we can guarantee that the closest level-*i* landmark to subsequent ball centers along the unknown shortest path are **always informed**.

- Analysis of RQA's approximation guarantee still works, because it is based on the via-landmark paths corresponding only to balls centered at vertices of the unknown shortest *od*-path.

- The quality of approximation provided via an informed landmark is dependent on the *landmark's relative distance* from the origin.
- For the first *informed* level-i landmark, the probability of its distance from o NOT belonging to the $N_i^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, ln(n)\right]$ is $\mathrm{O}\left(\frac{1}{n}\right)$, where i is the appropriate level for (o, d, t_o).

∴ **Success of (ALH)** criterion, which happens **whp** , reveals *asymptotic bounds*, for the (unknown) distance (and Dijkstra rank) from o to d.

- The quality of approximation provided via an informed landmark is dependent on the *landmark's relative distance* from the origin.
- For the first *informed* level-i landmark, the probability of its distance from o NOT belonging to the $N_i^{\delta/(R+1)} \cdot \left(\frac{1}{\ln(n)}, ln(n)\right]$ is $\mathrm{O}\left(\frac{1}{n}\right)$, where i is the appropriate level for (o, d, t_o).

∴ **Success of (ALH)** criterion, which happens **whp** , reveals *asymptotic bounds*, for the (unknown) distance (and Dijkstra rank) from o to d.

- Given that (ESC) did not occur (which could only improve the performance), and that (ALH) succeeds in its ``guess'' of the appropriate level, the corresponding variant of RQA works fine.
- Level-$(k+1)$ landmarks would *always provide a solution*, in time $\mathrm{o}(n)$.

∴ **Failure-of-(ALH)** contribution to the expectation of the query-time is negligible.

Experimental Evaluation

Experimental Evaluation

Identities of Instances

PARAMETER \ INSTANCE	Berlin (TomTom)	Germany (PTV AG)
#Nodes	473,253	4,692,091
#Edges	1,126,468	11,183,060
Time Period	24h (Tue)	24h (Tue-Wed-Thu)
$\lambda_{\max}$	0.017	0.130
$-\lambda_{\min}$	-0.013	-0.130
#Arcs with constant traversal-times	924,254	10,310,234
#Arcs with non-constant traversal-times	20,2214	872,826
Min #Breakpoints	4	5
Avg #Breakpoints	10.4	16.3
Max #Breakpoints	125	52
Total #Breakpoints	3,234,213	25,424,506

(A) Three variants of **random selection** method:

- **RANDOM (R)**: Independent and uniform random selections.
- **IMPORTANT RANDOM (IR)**: Move each selection of (R) to the most important node within a small ball from the selection.
- **SPARSE RANDOM (SR)**: Sequential random selection. Each selected landmark excludes a small neighborhood around it from future selections.

(A) Three variants of **random selection** method:

- **RANDOM (R)**: Independent and uniform random selections.
- **IMPORTANT RANDOM (IR)**: Move each selection of (R) to the most important node within a small ball from the selection.
- **SPARSE RANDOM (SR)**: Sequential random selection. Each selected landmark excludes a small neighborhood around it from future selections.

(B) Partition-dependent selections: Given a graph partition, consider as candidate landmarks only the boundary nodes of the partition.

- **METIS (M)** / **KAHIP (K)**: Start from a METIS / KaHIP partition.
- **SPARSE KAHIP (SK)**: Start from a finer KaHIP partition. Choose randomly, assuring sparsity, landmarks from the boundary nodes.
- **HYBRID (H)**: In a KaHIP partition, half landmarks chosen randomly (and sparsely) from boundary nodes. Remaining nodes equi-distributed randomly in the cells.

(A) Three variants of **random selection** method:

- **RANDOM (R)**: Independent and uniform random selections.
- **IMPORTANT RANDOM (IR)**: Move each selection of (R) to the most important node within a small ball from the selection.
- **SPARSE RANDOM (SR)**: Sequential random selection. Each selected landmark excludes a small neighborhood around it from future selections.

(B) Partition-dependent selections: Given a graph partition, consider as candidate landmarks only the boundary nodes of the partition.

- **METIS (M)** / **KAHIP (K)**: Start from a METIS / KaHIP partition.
- **SPARSE KAHIP (SK)**: Start from a finer KaHIP partition. Choose randomly, assuring sparsity, landmarks from the boundary nodes.
- **HYBRID (H)**: In a KaHIP partition, half landmarks chosen randomly (and sparsely) from boundary nodes. Remaining nodes equi-distributed randomly in the cells.

(C) **BETWEENESS CENTRALITY (BC)**: Choose landmarks sequentially, assuring sparsity, according to an approximate BC order.

- Preprocessing of FLAT @ BERLIN:

	BERLIN		GERMANY	
Parallelism	1 thread	6 threads	1 thread	6 threads
Time per landmark	69.5sec	11.5sec	481sec	80.2sec
Space per landmark	13.8MB		25.7MB	

- Responsiveness to **live-traffic reporting**: Averaging $1,000$ random disruptions of 15-min duration.

	BERLIN		GERMANY	
	#Affected Landmarks	Update Time (sec)	#Affected Landmarks	Update Time (sec)
SR_{2000}	32	21.4	3	37.2
SK_{2000}	36	28.8	4	39.1

Experimental Evaluation

(KMPPWZ (2016))

Query-Time Performance: Speedup $> 1,146$ for Berlin and > 902 for Germany.

Berlin: $n = 473,253$ vertices, $m = 1,126,468$ arcs.

Germany: $n = 4,692,091$ vertices, $m = 11,183,060$ arcs.

- **BERLIN:** 1.32sec resolution and $10,000$ *random* queries.

	TDD		FCA		FCA$^+$(6)		RQA	
	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %
R_{2000}	92.906	0	0.100	0.969	0.527	0.405	0.519	0.679
K_{2000}			0.115	1.089	0.321	0.405	0.376	0.523
H_{2000}			0.102	0.886	0.523	0.332	0.445	0.602
IR_{2000}			0.086	0.923	0.489	0.379	0.473	0.604
SR_{2000}			0.081	0.771	0.586	0.317	0.443	0.611
SK_{2000}			0.083	0.781	0.616	0.227	0.397	0.464
R_{541}			0.326	1.854	1.887	0.693	1.904	1.610
SR_{541}			0.451	1.638	3.252	0.614	2.856	1.531
R_{270}			0.639	2.583	3.707	0.881	3.842	2.482
SR_{270}			0.730	2.198	4.491	0.745	4.271	2.336

- **GERMANY:** 8.82sec resolution and 10,000 *random* queries.

	TDD		FCA		FCA$^+$(6)		RQA	
	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %
R_{2000}	$1,145.060$	0	1.532	1.567	8.529	0.742	9.219	1.502
K_{2000}			10.455	2.515	15.209	1.708	30.577	2.343
SR_{2000}			1.275	1.444	9.952	0.662	9.011	1.412
SK_{2000}			1.269	1.534	9.689	0.676	7.653	1.475

Experimental Evaluation

(KMPPWZ (2016))

Dijkstra-Rank Performance: Speedup $> 1,570$ for Berlin and $> 1,531$ for Germany.

Berlin: $n = 473,253$ vertices, $m = 1,126,468$ arcs.

Germany: $n = 4,692,091$ vertices, $m = 11,183,060$ arcs.

- **BERLIN:** 1.32sec resolution and 10, 000 *random* queries.

	TDD		FCA		FCA$^+$(6)		RQA	
	Rank	Speedup	Rank	Speedup	Rank	Speedup	Rank	Speedup
R_{2000}	146, 022	1	150	973.480	877	166.502	925	157.862
K_{2000}			190	768.537	866	168.616	670	217.943
H_{2000}			154	948.195	851	171.589	777	187.931
IR_{2000}			135	1, 081.644	823	177.426	839	174.043
SR_{2000}			119	1, 227.075	952	153.384	776	188.173
SK_{2000}			93	1, 570.129	755	193.406	501	291.461
R_{541}			545	267.930	3, 178	45.947	3, 406	42.872
SR_{541}			638	228.874	3, 684	39.637	3, 950	36.967
R_{270}			1, 075	135.834	6, 198	23.559	6, 702	21.788
SR_{270}			1, 195	122.194	7, 362	19.835	7, 398	19.738

- **GERMANY:** 8.82sec resolution and 10, 000 *random* queries.

	TDD		FCA		FCA$^+$(6)		RQA	
	Rank	Speedup	Rank	Speedup	Rank	Speedup	Rank	Speedup
R_{2000}	1, 717, 793	1	1, 659	1, 035.439	10, 159	169.091	11, 045	155.527
K_{2000}			9, 302	184.669	15, 373	111.741	30, 137	56.999
SR_{2000}			1, 277	1, 345.178	9, 943	172.764	9, 182	187.082
SK_{2000}			1, 122	1, 531.010	9, 000	190.866	7, 975	215.397

- Landmark hierarchies for HORN, with HR and HSR landmark sets:

Level	Size of Levels		Area of coverage	Excluded Ball Size (for *HSR*)	
	$\lvert L\rvert = 10,256$	$\lvert L\rvert = 20,513$		$\lvert L\rvert = 10,256$	$\lvert L\rvert = 20,513$
L_1	7,685	15,370	1,274	35	15
L_2	1,604	3,208	29,243	150	80
L_3	697	1,394	154,847	350	180
L_4	270	541	292,356	800	400

- Performance of HQA at 2.64sec resolution and 10,000 *random* queries:

	TDD				HQA			
	Time (msec)	Rel.Error %	Rank	Speedup	Time (msec)	Rel.Error %	Rank	Speedup
HR_{10256}	92.906	0	146,022	1	0.354	1.499	636	229.594
HSR_{10256}					0.436	1.409	721	202.527
HR_{20513}					0.217	1.051	324	450.685
HSR_{20513}					0.314	0.919	378	386.302

- HQA vs. FLAT/FCA in Berlin:

	Improvement in			Deterioration in
	Query Times (%)	Worst-case Relative Error (%)	Dijkstra Ranks (%)	Space (times)
R_{270} vs HR_{10256}	44.60	41.96	40.83	6.089
SR_{270} vs HSR_{10256}	40.27	35.89	39.66	6.407
R_{541} vs HR_{20513}	33.43	43.31	40.55	6.195
SR_{541} vs HSR_{20513}	30.37	43.89	40.75	6.438

Work In Progress (I)

(KPPWZ (2017))

- CFLAT -- A combinatorial oracle that:
  - Preprocesses and stores only **time-varying shortest-path trees**, rather than travel-time functions: Each vertex has a *time-dependent parent*, per landmark.
  - Avoids **duplicates** in preprocessed data, by storing common departure-time sequences only once and having all the relevant landmark-vertex pairs index them.

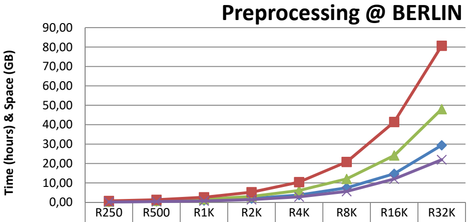
- CFCA -- A novel query algorithm that:
  1. Computes, in reverse order, many candidate paths from each discovered landmark to the destination.
  2. Runs TD-Dijkstra in the subgraph induced by the edges of these paths.

Work In Progress (II)

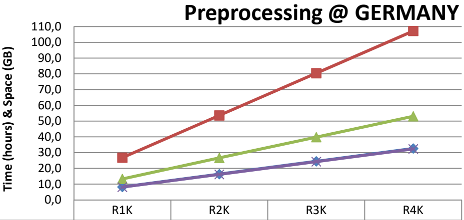
(KPPWZ (2017))

- Experimental Evaluation for CFLAT:
  - More detailed average-case statistics (50, 000 random queries).
  - Significant preprocessing space/time requirements.
  - Comparable query times with FLAT/FCA+, but now including the path reconstruction in the measurements.
  - Improved approximation guarantees.
  - Study the tails of the statistics (existence of outliers).

Preprocessing of CFLAT (RANDOM landmarks) (KPPWZ (2017))

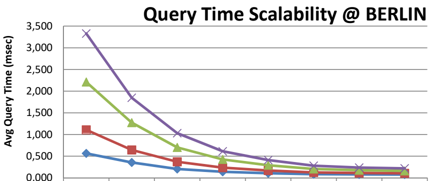


	R250	R500	R1K	R2K	R4K	R8K	R16K	R32K
Time (12 threads)	0,23	0,46	0,92	1,95	3,73	7,45	14,70	29,38
Time (6 threads)	0,38	0,75	1,53	3,02	6,08	12,05	24,12	48,02
Space (uncompr.)	0,70	1,30	2,60	5,20	10,40	20,80	41,40	80,66
Space (compr.)	0,17	0,34	0,69	1,40	2,80	5,60	12,00	21,94

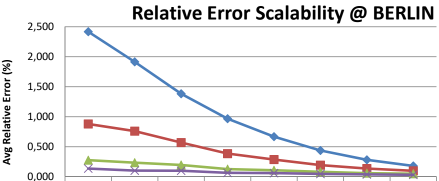


	R1K	R2K	R3K	R4K
Time (12 threads)	8,1	16,3	24,4	32,6
Time (6 threads)	13,3	26,6	39,8	53,0
Space (uncompr.)	26,8	53,6	80,4	107,2
Space (compr.)	8,1	16,1	24,2	32,3

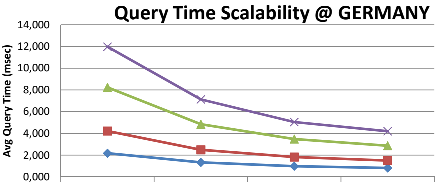
Query-time / Error Scalability CFCA (KPPWZ (2017))



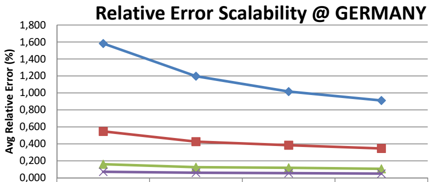
	R250	R500	R1K	R2K	R4K	R8K	R16K	R32K
CFCA(1)	0,565	0,356	0,205	0,140	0,106	0,085	0,079	0,077
CFCA(2)	1,109	0,644	0,371	0,236	0,169	0,125	0,112	0,105
CFCA(4)	2,212	1,274	0,702	0,425	0,291	0,204	0,177	0,161
CFCA(6)	3,330	1,848	1,031	0,613	0,411	0,281	0,237	0,219



	R250	R500	R1K	R2K	R4K	R8K	R16K	R32K
CFCA(1)	2,418	1,915	1,383	0,967	0,668	0,438	0,282	0,180
CFCA(2)	0,880	0,760	0,570	0,385	0,287	0,193	0,136	0,098
CFCA(4)	0,276	0,234	0,196	0,127	0,108	0,082	0,059	0,050
CFCA(6)	0,136	0,102	0,100	0,064	0,060	0,047	0,038	0,032



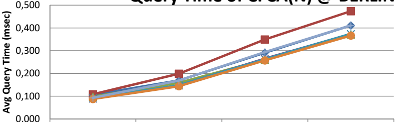
	R1K	R2K	R3K	R4K
CFCA(1)	2,175	1,333	0,981	0,819
CFCA(2)	4,219	2,492	1,824	1,503
CFCA(4)	8,238	4,845	3,477	2,864
CFCA(6)	11,974	7,130	5,045	4,201



	R1K	R2K	R3K	R4K
CFCA(1)	1,582	1,197	1,016	0,911
CFCA(2)	0,547	0,426	0,384	0,346
CFCA(4)	0,160	0,125	0,118	0,106
CFCA(6)	0,071	0,060	0,055	0,049

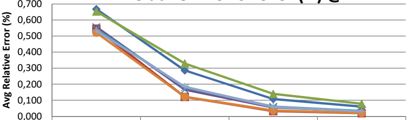
Query-time / Error of CFCA w.r.t. Landmark Sets (KPPWZ (2017))

Query Time of CFCA(N) @ BERLIN



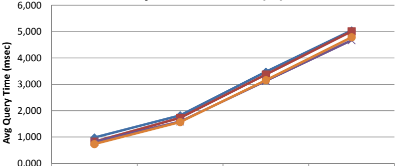
	N=1	N=2	N=4	N=6
R4K	0,106	0,169	0,291	0,411
SR4K	0,108	0,199	0,349	0,474
IR4K	0,098	0,159	0,261	0,371
SK4K	0,089	0,147	0,265	0,373
KC4K	0,091	0,148	0,262	0,374
BC4K	0,088	0,144	0,257	0,367
KB4K	0,093	0,169	0,294	0,411

Relative Error of CFCA(N) @ BERLIN



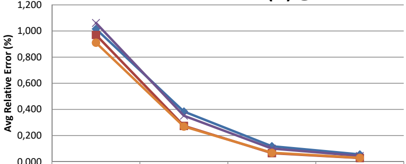
	N=1	N=2	N=4	N=6
R4K	0,668	0,287	0,108	0,060
SR4K	0,546	0,121	0,033	0,019
IR4K	0,653	0,329	0,140	0,078
SK4K	0,557	0,166	0,055	0,033
KC4K	0,544	0,181	0,060	0,033
BC4K	0,521	0,121	0,036	0,021
KB4K	0,534	0,184	0,058	0,031

Query Time of CFCA(N) @ GERMANY



	N=1	N=2	N=4	N=6
R3K	0,981	1,824	3,477	5,045
SR3K	0,825	1,727	3,373	5,010
SK3K	0,819	1,587	3,129	4,669
BC3K	0,733	1,568	3,158	4,787

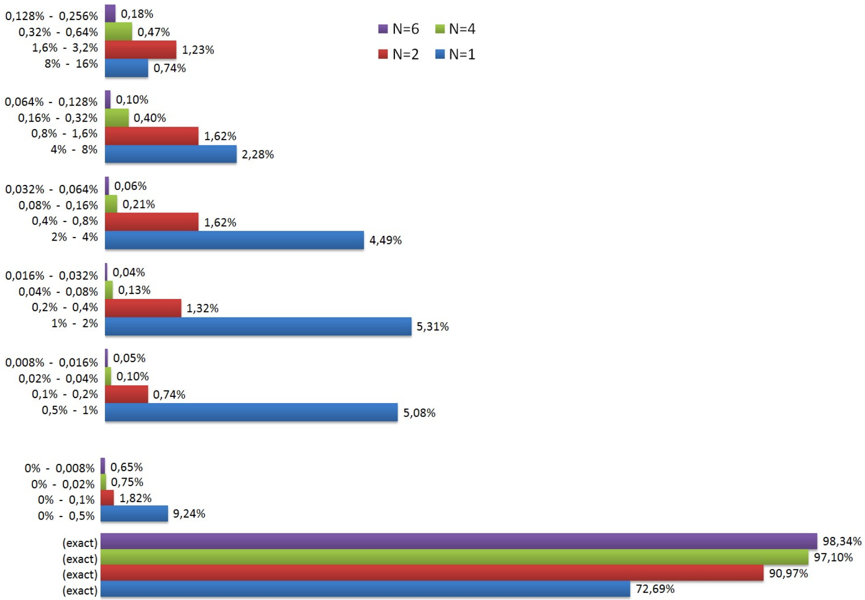
Relative Error of CFCA(N) @ GERMANY



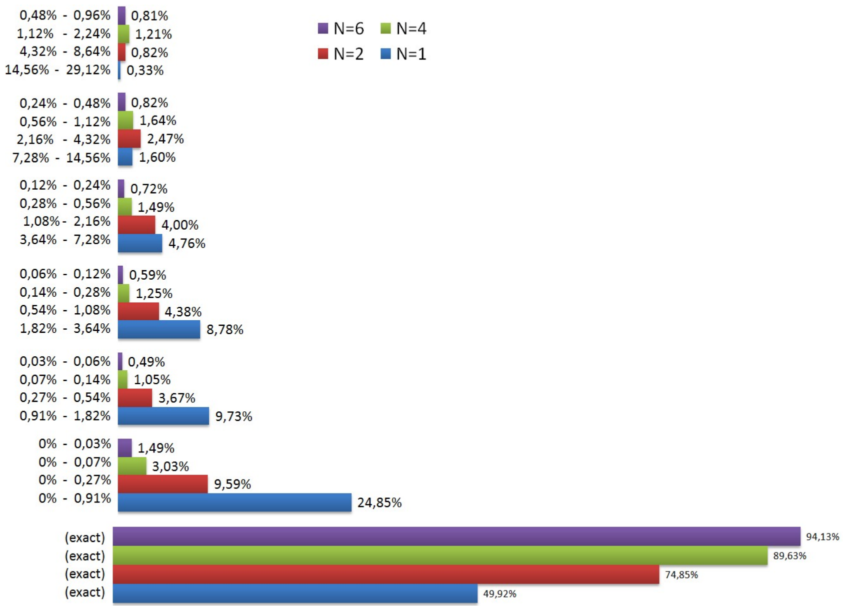
	N=1	N=2	N=4	N=6
R3K	1,016	0,384	0,118	0,055
SR3K	0,970	0,274	0,065	0,028
SK3K	1,062	0,352	0,100	0,044
BC3K	0,911	0,269	0,068	0,029

Exploring Outliers of Relative Error in BERLIN

(KPPWZ (2017))



Exploring Outliers of Relative Error in GERMANY (KPPWZ (2017))



Related Literature

1. (Dreyfus (1969)) S. E. Dreyfus. **An appraisal of some shortest-path algorithms**. *Operations Research*, 17(3):395--412, 1969.
2. (OR (2000)) A. Orda, R. Rom. S**hortest-path and minimum delay algorithms in networks with time-dependent edge-length**. *J. ACM*, 37(3):607--625, 1990.
3. (Dean (2004)) B. C. Dean. **Shortest paths in FIFO time-dependent networks: Theory and algorithms**. Technical report. MIT, 2004.
4. (DOS (2010)) F. Dehne, O. T. Masoud, J. R. Sack. Shortest paths in time-dependent FIFO networks. *ALGORITHMICA*, 62(1-2):416--435, 2012.
5. (FHS (2011)) L. Foschini, J. Hershberger, S. Suri. **On the complexity of time-dependent shortest paths**. *ALGORITHMICA*, 68(4), pp. 1075--1097, 2014.
6. (KZ (2014)) S. Kontogiannis, C. Zaroliagis. **Distance oracles for time dependent networks**. In *ALGORITHMICA*.
7. (KMPPWZ (2016)) S. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, C. Zaroliagis. **Engineering Oracles for Time-Dependent Road Networks**. *Algorithm Engineering and Experiments* (ALENEX 2016), SIAM, 2016.
8. (KPPWZ (2017)) S. Kontogiannis, G. Papastavrou, A. Paraskevopoulos, D. Wagner, C. Zaroliagis. **Improved Oracles for Time-Dependent Road Networks**. Submitted for publication.
9. (KWZ (2016)) S. Kontogiannis, D. Wagner, C. Zaroliagis. **Hierarchical Oracles for Time-Dependent Road Networks**. In ISAAC 2016. Invited to ALGORITHMICA (2017).