Distance Queries in Modern Large-Scale Complex Networks

Mattia D'Emidio

Gran Sasso Science Institute (GSSI), L'Aquila (Italy)
`mattia.demidio@gssi.it` – `www.mattiademidio.com`

DISTRIBUTED SYSTEMS @ Univaq

October 31, 2017

GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

Outline

Reporting Shortest Paths/Distances on Graphs

Problem

- Reporting shortest paths/distances between pairs of **vertices** of a graph is one of the **most fundamental problems** in graph theory and algorithmics
- **Pletora of applications**

Reporting Shortest Paths/Distances on Graphs

Classic Applications

- **Communication Networks**
  - Fundamental to **most of routing protocols**, efficient use of communication resources to forward data
- **Sensor Networks**
  - establish connections
- **Route/Journey Planning**
  - computing best connections in road networks and/or schedule-based transport systems, fundamental to planning **software** like e.g. Google Maps
- **Data Mining**
  - Find stronger relationships among data by their “closeness”
- **Graph Databases**
  - management and/or efficient querying of data

Reporting Shortest Paths/Distances on Graphs

Emerging Applications

- **Context-Aware Search**
  - give higher ranks to **web pages** more related to the currently visiting web page
  - Fundamental to **search engines**
- **Socially-Sensitive Search**
  - help users to find related users/contents
  - Fundamental to **social networks hosts**
- **Social Network Analysis / Social Engineering**
  - distance between users is a proxy for closeness, analyze **influential people and communities**
- **Biological Systems Analysis**
  - discovery of **optimal pathways** between compounds in metabolic **networks**
- **Distributed File Systems**
  - reflect changes onto **replicae** efficiently

Reporting Shortest Paths/Distances on Graphs

Problem

- Given a **graph** $G = (V, E)$ having $n = |V|$ vertices and $m = |E|$ edges, and a **pair** of vertices $s, t \in V$
- Report, upon query, **distance** $d(s, t)$, i.e. the **weight of a shortest path** between s and t in G
  - in the smallest possible amount of **time** (**efficiently**)
  - in a **reliable** way (we'll say what this means later)

(Highly) related problems:

- **Reachability:** report **yes** if there exist a path between s and t, **no** otherwise
- **Path-reporting:** report the **whole shortest path** (set of vertices and edges)

Reporting Shortest Paths/Distances on Graphs

Naive Approach 1: BFS (Dijkstra's) Algorithm

- Execute upon query, **no preprocessing**
- Space-efficient, **no additional storage**
- Best algorithm w.r.t. worst-case query time if no preprocessing is allowed
    - $O(m+n)$ in unweighted graphs
    - $O(m+n\log n)$ in weighted graphs

Reporting Shortest Paths/Distances on Graphs

Naive Approach 2: **Distance Table**

- **Time-consuming** preprocessing
- Compute and store **all pairs** distances via $|V|$ BFS (or Dijkstra's) executions
- $\Theta(|V|^2)$ space
- Optimal $O(1)$ query time, retrieve the value upon query

Reporting Shortest Paths/Distances on Graphs

Naive approaches fail at being practical in **modern networks** which tend to be

- **Large-Scale:** billion vertices networks (e.g. Twitter, Facebook, Road Networks, Internet)
- **Complex:** various topological features (non-regular, non-bounded-treewidth, non-uniform degree distributions, etc)

Reporting Shortest Paths/Distances on Graphs

Naive approaches worst-case performance

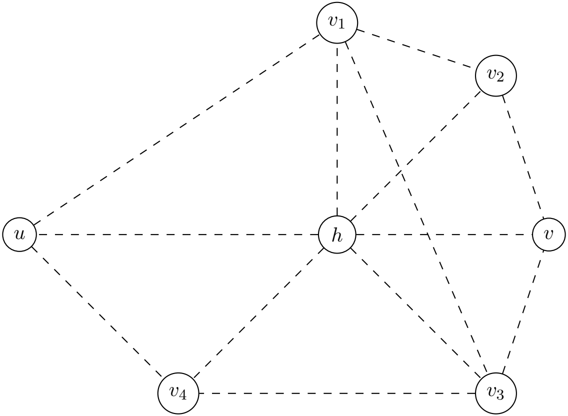
- Approach 1:
  - **unsustainable** query time (even linear per query can be too much)
- Approach 2:
  - **impractical** preprocessing effort and space occupancy
  - **Trade-offs are needed** to achieve scalability

Reporting Shortest Paths/Distances on Graphs

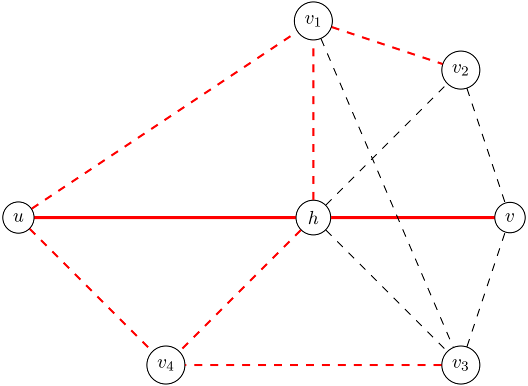
Trending Strategy

- Do **"something in the middle"** between the two extreme naive solutions
  - in terms of space, preprocessing, and query time
- "Suitably" **preprocess** the graph
  - Computational pre-processing effort in between $O(1)$ and $O(n(m+n))$
- **Store** "acceptable" amount of data
  - Space complexity in between $O(1)$ and $O(n^2)$
- Use data to **answer** queries "quickly"
  - Computational complexity of query algorithm in between $O(1)$ and $O(m+n)$
- There are a lot of **trade-offs**, let us discuss an example

Given a graph

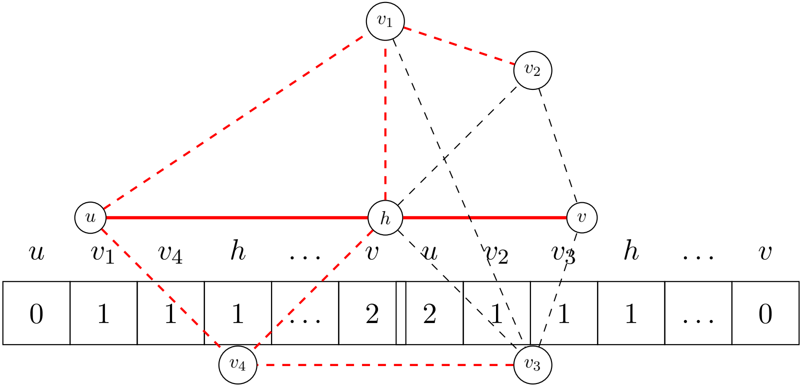


Naive 1: Dijkstra's (or BFS)



- **No preprocessing**, **do not store** anything
- **Query:** possibly access the whole graph (search space through all data)

Naive 2: Distance Table



u	v_1	v_4	h	$\dots$	v	u	v_2	v_3	h	$\dots$	v
0	1	1	1	$\dots$	2	2	1	1	1	$\dots$	0

- **Full preprocessing**, store explicitly **all solutions**
- **Query:** access **single** data entry of interest

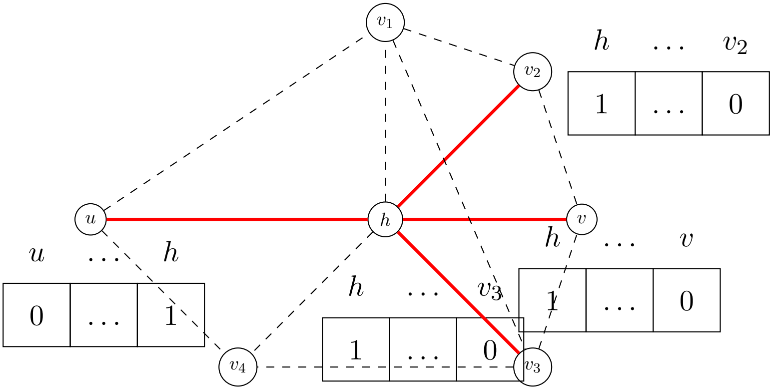
Reporting Shortest Paths/Distances on Graphs

Trade-Off: questions

- How can we **suitably preprocess** the graph (subquadratic time), store a **practical** amount of space (subquadratic) and be able to answer to queries in **reasonable time** (as close to constant time as possible)?
- **Preprocessing:** do less that all BFSs or Dijkstra's
- **Space:** store less than all pairs
- **Query:** access few data entries and do few operations

Reporting Shortest Paths/Distances on Graphs

Temptative answer: yes



- Exploit **optimal sub-structure** of shortest paths
- e.g. look at h, it is a sort of "center" of **many shortest paths**, it could be used for **encoding** many of them

Reporting Shortest Paths/Distances on Graphs

More detailed question

1. Can we **compress** the n^2 **solutions** into a more **compact** data structure that can answer distance queries **quickly** (in time not so far from $O(1)$)?
2. Can we **compute** such data structure by a preprocessing algorithm that is **practical** in terms of time?
3. Moreover, can we **distribute** the data in order to build a **more reliable system**?

Reporting Shortest Paths/Distances on Graphs

Several works on the matter

- Tree-decomposition based (Akiba+ EDBT 2012)
- Multi-level / Hierarchical Based (Abraham+ ESA 2012)
- Variety of **speed-up techniques** for Dijkstra's algorithm, tailored for special classes of graphs (e.g. graphs with low *highway dimension* like road networks)
- **Pruned Landmark Labeling (PLL)** (Akiba+ SIGMOD 2013)
- Landmark-based (Potamias+ CIKM 2009)
- Distance Sketch (Sarma+ WSDM 2010)
- Path Sketch (Gubichev+ CIKM 2010)
- Graph Spanners (Peleg+ JGT 1989, Baswana+ SODA 2008)
- "SP based" (Elkin+ SODA 2015, Thorup+ JACM 2015)

Outline

Pruned Landmark Labeling

State-of-the-art w.r.t.: Pruned Landmark Labeling (PLL)

- Best known **trade-off** for **complex general networks** (undirected/directed unweighted/weighted) [1]
- **Worst-case**
  - Naive 2 time and space
  - $O(n)$ query time
  - Awful
- However, in practice, it outperforms **all other methods**
  - **acceptable** preprocessing effort ($\sim$hours)
  - **practical** space occupancy ($\sim$gibibytes)
  - **small enough** query time ($\sim$milliseconds)
  - for **billion vertices graphs**
  - suitable to exploit **parallel architectures**
  - it allows **distribution of information**

[1] Experimentally speaking

Pruned Landmark Labeling

Two Main ingredients: 2–Hop Cover + Distance Labeling

- Based on the **intuition** we've seen before
- Rough idea is
  - to compute a **compact representation** of the shortest paths, namely the **2–Hop Cover**
  - to convert it to a **distance labeling**, a compact **label-based data structure** that can be used to answer query **quickly** (and in a **distributed fashion**)

Pruned Landmark Labeling

Some Notation

- Focus on **undirected unweighted graphs** and **distances**
- Given undirected unweighted graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges
- $N(v) = \{u \in V \mid \{u, v\} \in E\}$ denotes **set of neighbors** of v in G
- $d(u, v)$ denotes (hop) **distance** between u and v (number of edges in shortest path between u and v)
- If u and v not connected, then $d(u, v) = \infty$

2-Hop Cover (Cohen+ J. Comput. 2012)

Given a graph G

- For every $u, v \in$ let
  - P_{uv} be a **collection** of paths between u and v in G (e.g. P_{uv} can be the **shortest paths** between u and v)
- A **hop** is a pair (h, u)
  - where h is a **path** in G and u is **one of the endpoints** of h (for instance, h is a shortest path)

2-Hop Cover (Cohen+ J. Comput. 2012)

2–Hop Cover of G

- A **set of hops** $H(G)$ is a **2–Hop Cover** of the collection of paths $P = \bigcup_{u,v \in V} P_{uv}$ **if and only if**
  - **for every pair** $u, v \in V$ such that $P_{uv} \neq \emptyset$
  - there exists **at least one** $p \in P_{uv}$ and **two hops** $(h_1, u) \in H$ and $(h_2, v) \in H$ such that $p = h_1 \oplus h_2$
- Each pair is said to be **covered** (or to satisfy the **Cover Property**)

Pruned Landmark Labeling

Distance Labeling of a Graph G

- A **label** $L(v)$ is assigned to each vertex v of G
- The **labeling** $L(G)$ of G is given by $\{L(v)\}_{v \in V}$
- A **query** on the distance between two vertices s and t is answered by simply **looking** at the labels $L(s)$ and $L(t)$ **of the two vertices** i.e. $d_G(s,t) = f(L(s), L(t))$
- Main benefit: **distribution of information**
- Several methods to build distance labelings
  - Graph Embedding
  - Distance Sketches/Landmark based

Pruned Landmark Labeling

2–Hop Covers yield Distance Labelings

- A label is intended as a **set of pairs** (entries) (u, δ_{uv}), where u is a vertex in V and $\delta_{uv} = d_G(u, v)$
- Compute a **2–Hop Cover** $H(G)$ of the collection P of the shortest paths of G
- For each **hop** $(h, u) \in H$ add **entry** $v, w(h)$ to $L(u)$
- Where v is the other **endpoint** of h and $w(h)$ is its **weight**

Pruned Landmark Labeling

2–Hop Covers yield Distance Labelings

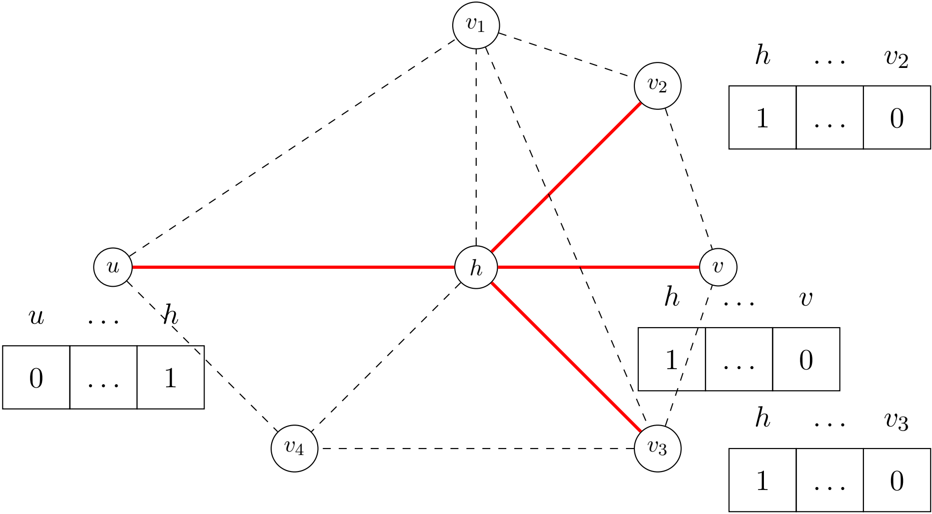
- **Labels** can the be used to answer to a **query** on the distance between two vertices s and t as follows:

$$\textsc{Query}(s,t,L) = \begin{cases} \min\{\delta_{vs} + \delta_{vt} \mid v \in L(s) \wedge v \in L(t)\} & \text{If } L(s) \cap L(t) \neq \emptyset \\ \infty & \text{Otherwise} \end{cases}$$

- $\arg\min\{\delta_{vs} + \delta_{vt} \mid v \in L(s) \wedge v \in L(t)\}$ is called **hub vertex** that covers the pair
- Clearly $|L| \approx |H|$

Pruned Landmark Labeling

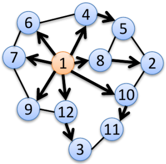
Example:



Pruned Landmark Labeling

Trivial Computation of 2–Hop Cover Labelings

1. $L(u) = \emptyset$ for all $u \in V$
2. BFS rooted at v, for all $v \in V$
3. When u is settled, add pair (v, δ_{vu}) to $L(u)$



After a BFS from 1

$L_1(1)$:

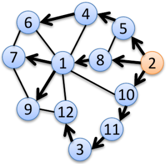
Vertex	1
Distance	0

$L_1(2)$:

Vertex	1
Distance	2

$L_1(3)$:

Vertex	1
Distance	2



After a BFS from 2

$L_2(1)$:

Vertex	1	2
Distance	0	2

$L_2(2)$:

Vertex	1	2
Distance	2	0

$L_2(3)$:

Vertex	1	2
Distance	2	3

Pruned Landmark Labeling

Trivial Preprocessing: resulting performance

- **Preprocessing** in $O(n(m+n))$ worst case time – **impractical** as Naive Approach 2
- $\Theta(n^2)$ resulting labeling **space occupancy** – **impractical** as Naive Approach 2
- For pair (s,t), **query** takes $O(|L(s)|+|L(t)|)$ – depends on labels' size
- Avg label size per vertex $O(n)$, hence query time $\in O(n)$ also **impractical**

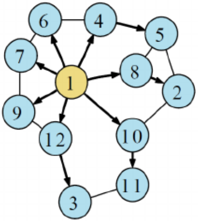
Pruned Landmark Labeling

Improved Preprocessing: Ordering and Pruning

1. **Order** vertices according to some “importance“ criterion $v_1, v_2, ...v_n$
2. Perform BFS rooted at v_i, for all $v_i \in V$, according to the computed **ordering**
3. Let L_{k-1} be the **status of the labeling** before the BFS rooted at a certain v_k
4. Initially $L_0(u) = \emptyset$ for all $u \in V$
5. During visit rooted at v_k, when vertex u is settled at distance d
   - 5.1 **If** $\textsc{Query}(v_k, u, L_{k-1}) \leq d \Rightarrow$ **Break! i.e. Prune**
   - 5.2 **Else** add pair (v_k, d) to $L_k(u)$ and continue

Pruned Landmark Labeling

Improved Preprocessing



First BFS from vertex 1

$L'_1(1)$:

Vertex	1
Distance	0

$L'_1(2)$:

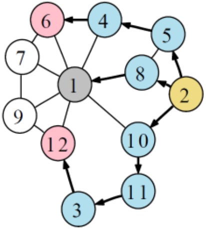
Vertex	1
Distance	2

⋮

$L'_1(6)$:

Vertex	1
Distance	1

⋮



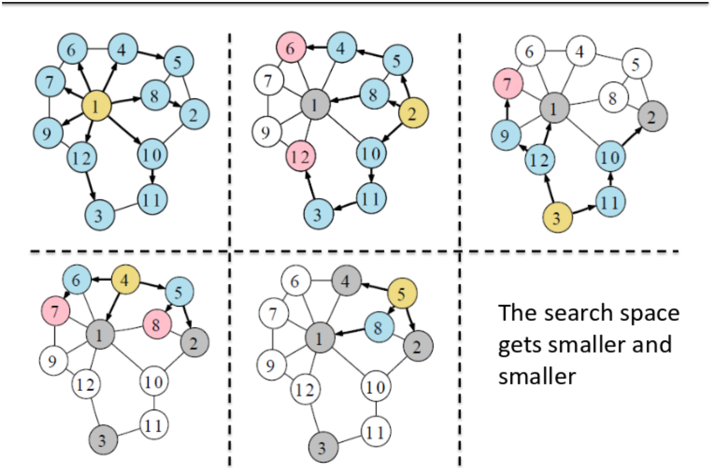
Second BFS from vertex 2

$\textsc{Query}(2, 6, L'_1) = 2 + 1 = 3 = d(2,6)$

→ Vertex 6 is pruned.

Pruned Landmark Labeling

Improved Preprocessing



The search space gets smaller and smaller

Improved Preprocessing

Performance and Theoretical Foundations

- **Same worst-case bounds** in terms of time and space of the trivial
- Correctness is **ordering-independent**
- Quality instead depends on **ordering**
- **Different orderings** yield **different labelings** (of different size, preprocessing and query time)
- Intuitively, the **more shortest paths we find (sooner), the better**
- Vertices should be **ordered** by some function of their **importance w.r.t. shortest paths**

Improved Preprocessing

Performance and Theoretical Foundations

- Computing an ordering that induce a **labeling of minimum size** (i.e. finding a 2–Hop Cover of **minimum cardinality**) is known to be **NP-Hard** (cast to greedy set cover)
- There exists a **polytime** $O(\log n)$ approx algo
- Requires several computations of **densest subgraph**, takes $O(mn\log(\frac{n^2}{m}))$, **impractical**

Improved Preprocessing

There is a way out

- All these methods yield **minimal labelings**
- **Minimal labelings** have been experimentally shown to exhibit good-performance
  - **Practical** Preprocessing
  - **Compact** Labeling (Size)
  - **Small** Query Time
- A **minimal labeling** is s.t. the removal of any single entry breaks the **cover property**
- Good minimal labelings can be computed via **centrality measures** (e.g. degree, betweenness centrality)

Pruned Landmark Labeling

PLL versus Modern Networks

- Modern networks are intrinsically **dynamic:** change over time
  - **On-line Social Networks:** new friends, removed friends/pages
  - **Web indexing graphs:** new pages/links, broken links, removed pages
  - **Blogging:** new replies/posts, removed users/posts/replies
  - **Collaboration networks:** new papers
  - **Infrastructure networks:** disruptions, new roads, new trains, cancelled flights
  - **Evolving data sets:** new entries, outdated entries

Pruned Landmark Labeling

PLL as it is, fails

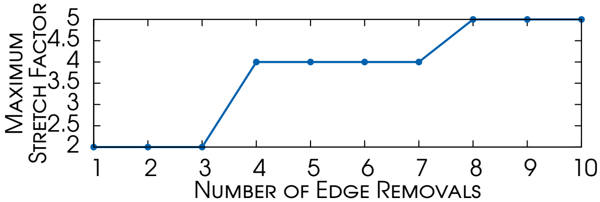
- Data (labels) become easily **outdated**
- The number of queries that become **incorrect** grows fast (even after few updates, preliminary experiments show)
- **Repeating** the preprocessing phase every time something changes is not **doable**
- Need for **efficient dynamic algorithms!**

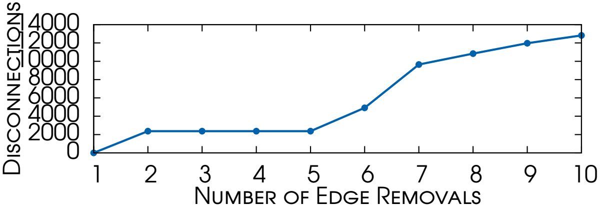
Further motivation

- There are a **lot of applications** that **inherently rely** on knowing **how distances and shortest paths** evolve over **time**
- Need for **efficient dynamic algorithms** also to efficiently support **historical** queries (ask distances at different times)

Pruned Landmark Labeling

Maximum stretch factor and number of disconnected pairs, on a ”Flight Data“ network subject to up to 10 edge removals.





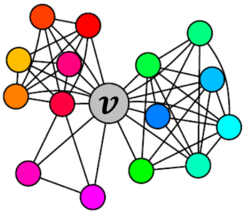
Some Applications

Graph Analytics

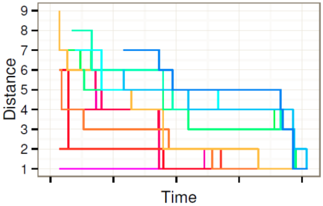
1. Dynamic Centrality Measures

Ego Network of v

(subgraph induced by v and neighbors)



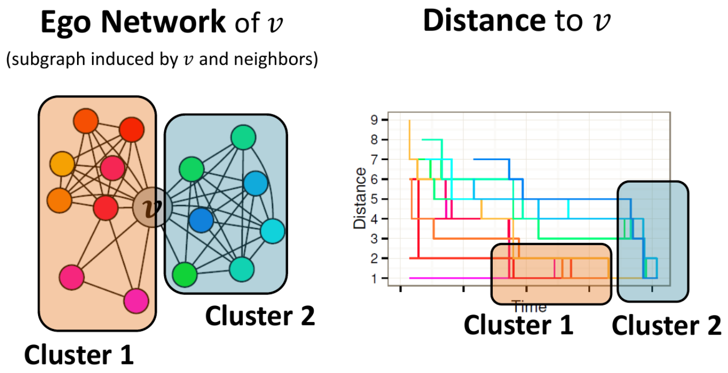
Distance to v



Some Applications

Graph Analytics

2. Dynamic Community Detection

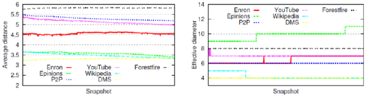


Some Applications

Graph Analytics

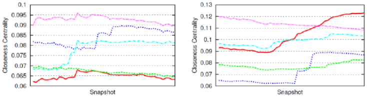
3. Property Evaluation over Time (e.g. Bioinformatics)

Average Distance
Effective Diameter



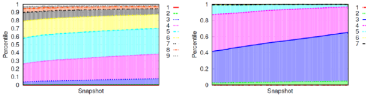
(a) Average distance (b) Effective diameter

Closeness Centrality



(a) Enron (b) Epinions

Temporal Hop Plot



Outline

Dynamic Algorithms for 2–Hop Cover Labelings

Very active field of research
Publications in basically all **top notch CS conferences**: ESA, SODA, WWW, SIGMOD, AAAI, KDD, VLDB

Incremental Case
An efficient algorithm for handling both **edge and vertex additions** is known (Akiba+ WWW 2014)

Decremental + Fully-Dynamic Case: work by our group
First algorithm for handling **edge and vertex removals** and for supporting **generic updates** (D'Angelo, D'Emidio, Frigioni – IWOCA 2016)

Dynamic Algorithms for 2–Hop Cover Labelings

Incremental Algorithm (IncPLL): intuition

- Let (u, v) be an **edge** to be **added to the graph**
- There might be some **label entries** that do not correspond to shortest paths **anymore** (insertions can induce **decreases** of distances only)
- **Lazy strategy: forget** outdated label entries, insert **new ones** only
- **Correctness:** query remains **correct** since the **minimum** is searched
- **Performance:** might **degrade** over time, perform from-scratch preprocessing **periodically**

Dynamic Algorithms for 2–Hop Cover Labelings

IncPLL algorithm

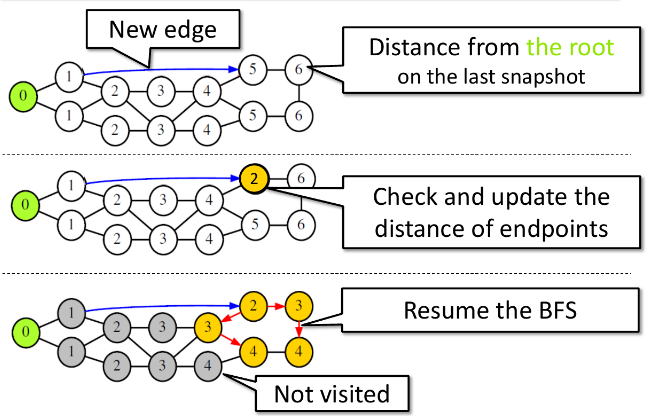
For all vertices $v_i \in L(u) \cup L(v)$

- resume the BFS, originally rooted at v_i, from vertices u and v
- **add new** pairs if the query test succeeds
- prune with the same **policy** of preprocessing

Vertex addition: add an isolated vertex, add its edges, perform IncPLL

Dynamic Algorithms for 2–Hop Cover Labelings

IncPLL at work[2]



[2] Thanks to Akiba+ for providing some of the figures

Dynamic Algorithms for 2–Hop Cover Labelings

Why the lazy strategy?

- Removing **outdated entries** is costly, **takes** $O(n)$ worst-case per update

Alternative

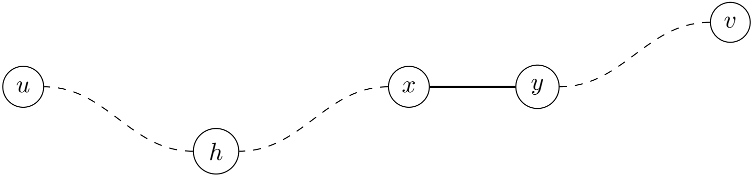
- Ignore **outdated** entries, simply **add new ones**
- "Exact" still **holds**, query looks for the **minimum**
- Does not guarantee minimality, requires **periodical reconstruction**
- Experimentally **behaves** pretty well (Akiba+ WWW 2014)
  - Performance degrades very slowly, **few** new entries **added**

Dynamic Algorithms for 2–Hop Cover Labelings

Decremental Algorithm (DecPLL)

- Outdated entries **must** be removed
- **Cannot be ignored** (as in the incremental case), they might lead to **underestimation** of distances
- Algorithm DecPLL works in **three phases**
  - **Detection** of so-called **affected** vertices
    - vertices whose label contains at least one **entry** that might be **out-of-date**
  - **Removal** of outdated entries by analyzing such affected vertices' labels
    - This might **break** the cover property
  - **Restore** the **cover property** for vertices that are **uncovered** by computing and adding new **label entries**

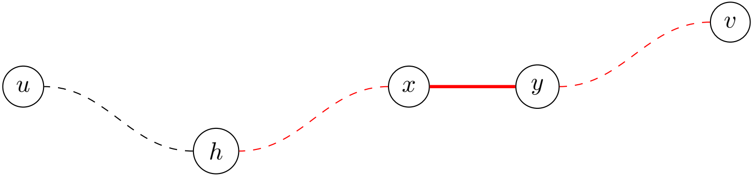
Affected Vertices



- Suppose we are given a **shortest path** between u and v
- **Solid line:** edge $\{x, y\}$
- **Dashed lines:** shortest paths
- **Assume** that $h \in L(u) \cap L(v)$ and h is a **hub** for pair (u, v)
- That is $(h, \delta_{vh}) \in L(v)$ and $(h, \delta_{uh}) \in L(u)$

Dynamic Algorithms for 2–Hop Cover Labelings

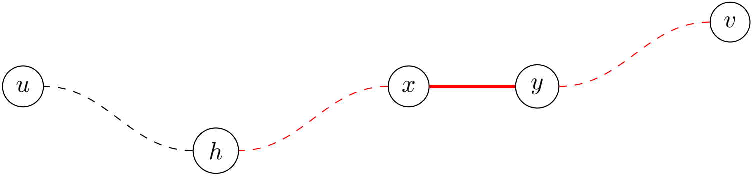
Affected Vertices



- If $\{x, y\}$ is **removed**
- Then pair (h, δ_{vh}) in $L(v)$ is not **correct** and must be **updated** (or **removed**)
  - v is said to be **affected**
  - **Formally,** v **affected** if there exists a shortest path **induced** by L between v and any other vertex u that passes through edge $\{x, y\}$
  - A shortest path is **induced** by L if it can be **obtained** by combining **two hops**
  - By analyzing such vertices we can **find and remove** obsolete labels

Dynamic Algorithms for 2–Hop Cover Labelings

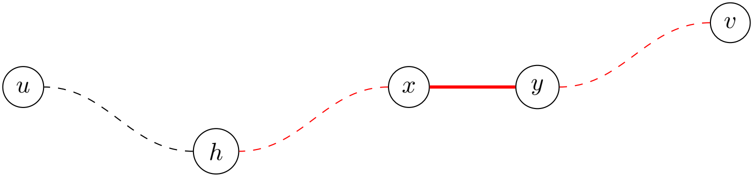
Detection of Affected Vertices: baseline



- Trivial computation of **all affected vertices** would require finding (and checking) the status of **all hubs** of **all pairs** (u, v) **of vertices** of G

Dynamic Algorithms for 2–Hop Cover Labelings

Detection of Affected Vertices: advanced



- A more **convenient** way of computing and storing them is that of **dividing them into two sets** `A`(x) and `A`(y)
- Set `A`(x) (`A`(y), resp.): vertices that are **affected w.r.t.** y (x, resp.)
- It can be **proved** that is sufficient to **test pairs**
  - (i, x) for all $i \in V$ and (y, j) for all $j \in V$
  - to determine **all affected vertices**
- **Intuition:** if v is affected **because** of the shortest path **toward** u if v is affected **because** of the shortest path **toward** x

Dynamic Algorithms for 2–Hop Cover Labelings

Detection of Affected Vertices

Different possible strategies for computing **affected vertices**

Common Intuition:

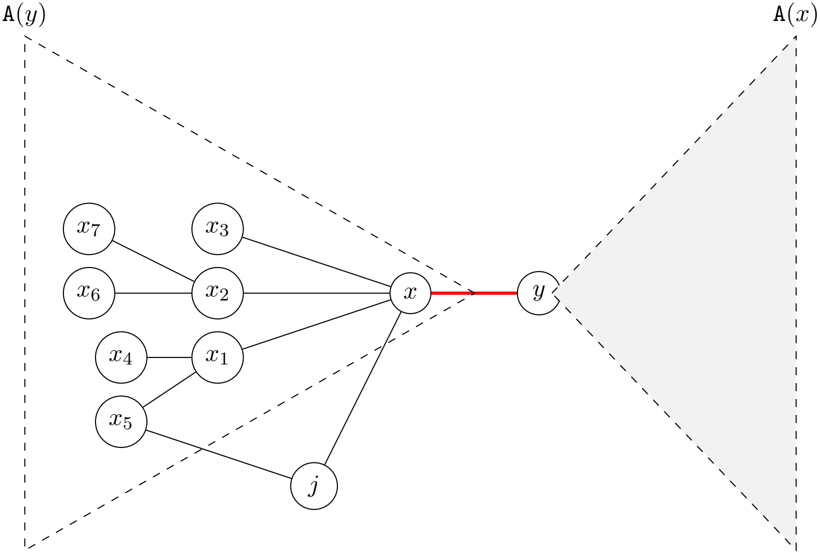
- Grow **two BFS-like visits** rooted at x and y
- During visit rooted at x (y, resp.), compute set A(y) (A(x), resp.) as follows:
  - Start the BFS visit by adding x to A(y)
  - For each **settled** vertex u, test the status of the corresp. **hub** w.r.t. y
  - Let h be the hub of pair u, y
    - If h is in A(y), add vertex u to A(y), i.e. u becomes **affected**
  - **If** u becomes affected, **visits** its neighbors and continue
  - **Else** break

Dynamic Algorithms for 2–Hop Cover Labelings

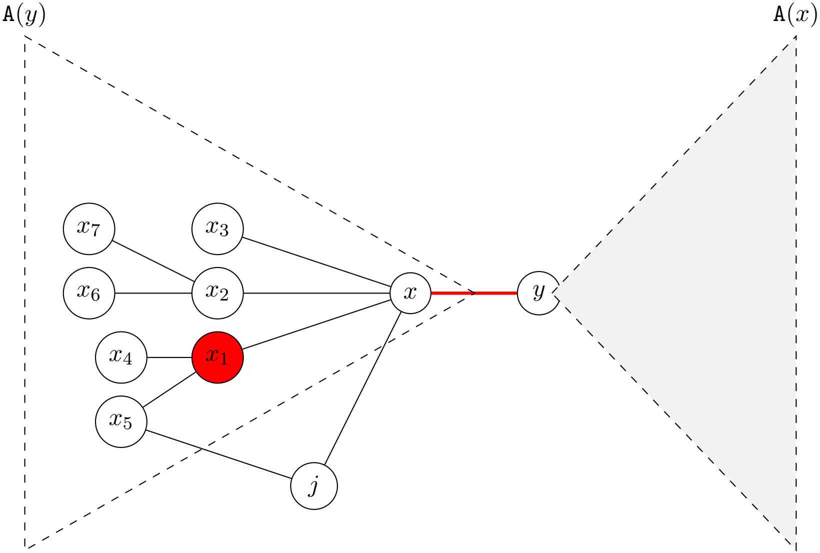
PseudoCode (some details omitted)

```
A ← ∅;
foreach v ∈ V do
    mark[v] ← false;
Q ← ∅;
Q.Enqueue(x);
while Q ≠ ∅ do
    v ← Q.Dequeue();
    mark[v] ← true;
    A(x) ← A(x) ∪ {v};
    foreach u ∈ N_i(v) such that ¬mark[u] do
        if d_i(u, y) ≠ d_{i−1}(u, y) then
            Q.Enqueue(u);
        else
            if h ∈ A(x) for some h in the set of hubs of pair (u, y) in G_{i−1} then
                Q.Enqueue(u);
```

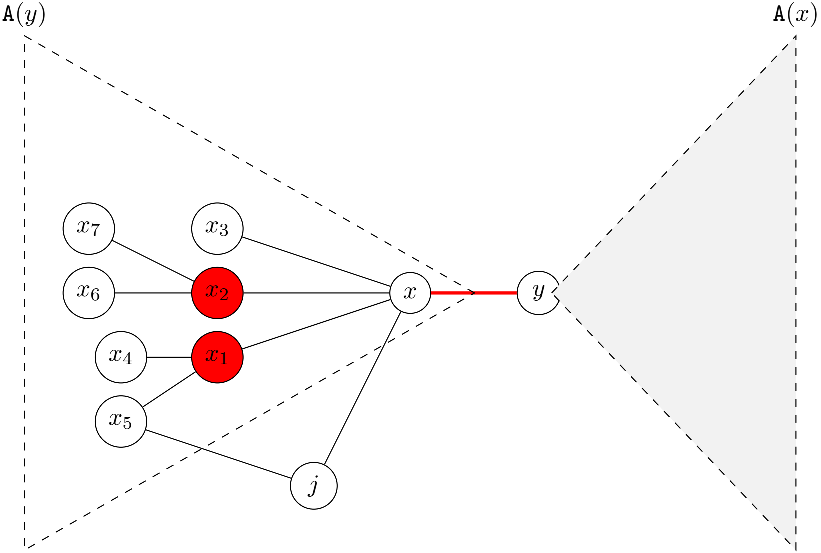
Dynamic Algorithms for 2–Hop Cover Labelings



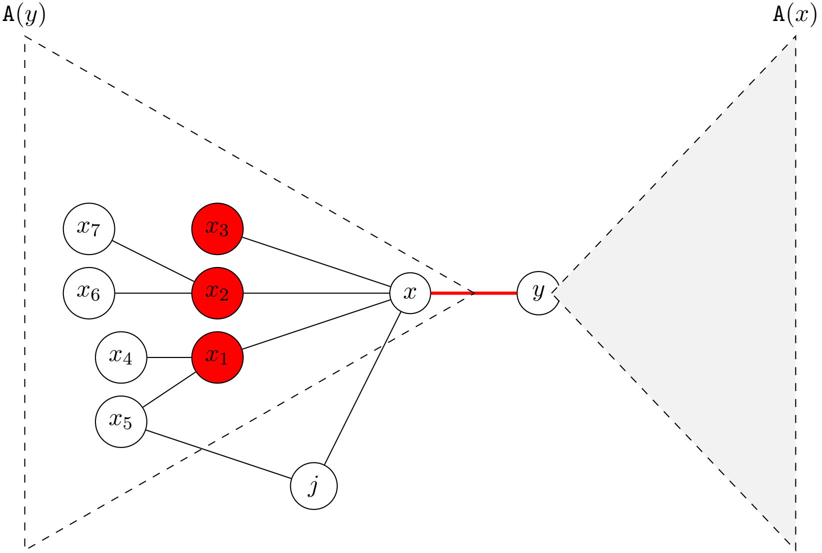
Dynamic Algorithms for 2–Hop Cover Labelings



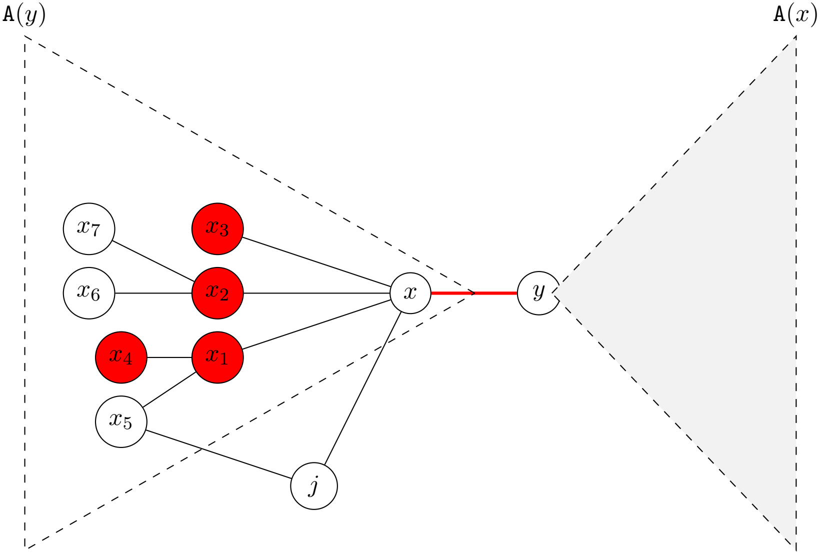
Dynamic Algorithms for 2–Hop Cover Labelings



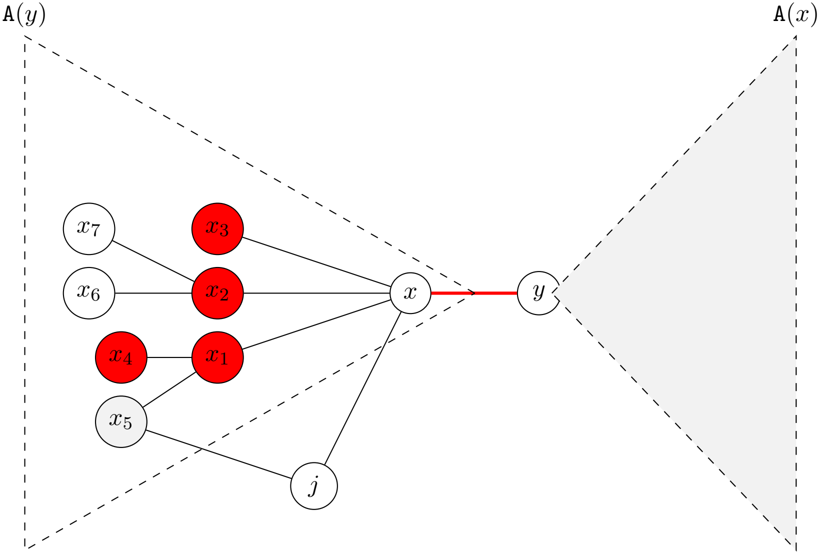
Dynamic Algorithms for 2–Hop Cover Labelings



Dynamic Algorithms for 2–Hop Cover Labelings



Dynamic Algorithms for 2–Hop Cover Labelings



Dynamic Algorithms for 2–Hop Cover Labelings

Theorem (Correctness)

*At the end of the above routine, all **affected** vertices are **found***

Proof.

By induction on the distance from x (y) $\square$

Dynamic Algorithms for 2–Hop Cover Labelings

Removal of Outdated Labels

1. For all vertices $v \in \texttt{A}(x)$
   - **Remove** from $L(v)$ any entry (u, δ_{uv}) such that $u \in \texttt{A}(y)$, if it exists
2. A symmetrical algorithm to **remove** labels of vertices in $\texttt{A}$(y)

PseudoCode

```
foreach v ∈ A(x) do
    foreach u ∈ L(v) such that u ∈ A(y) do
        Remove (u, δ_uv) from L(v);

foreach v ∈ A(y) do
    foreach u ∈ L(v) such that u ∈ A(x) do
        Remove (u, δ_uv) from L(v);
```

Theorem (Correctness)

At the end of the above routine, all ***outdated*** *entries are* ***removed***

Proof.

Trivial, we have shown that affected nodes are those which contain outdated entries □

Dynamic Algorithms for 2–Hop Cover Labelings

Restoring the cover property

1. A BFS-like visit rooted at **each vertex** $a \in \hat{\mathtt{A}}$ is **restarted**, where $\hat{\mathtt{A}}$ is the **smaller** in size between $\mathtt{A}(x)$ and $\mathtt{A}(y)$
2. **Restore the cover property**, by adding labels to vertices settled during the BFS
3. **Do not add redundant labels**, by performing queries during the visit
    - Guarantees minimality

PseudoCode

```
foreach a ∈ A(x) do
    Q ← ∅;
    mark[a] ← true; dist[a] ← 0;
    foreach v ∈ V \ {a} do
        mark[v] ← false;
        dist[v] ← ∞;
    foreach v ∈ N_i(a) do
        Q.Enqueue(v);
        dist[v] ← 1;
    while Q ≠ ∅ do
        v ← Q.Dequeue();
        mark[v] ← true;
        if dist[v] < QUERY(a, v, L) and v ∈ A(y) then
            if v < a then
                Insert (v, dist[v]) in L(a);
            else
                Insert (a, dist[v]) in L(v);
            foreach u ∈ N_i(v) such that ¬mark[u] do
                dist[u] ← dist[v] + 1;
                Q.Enqueue(u);
```

Dynamic Algorithms for 2–Hop Cover Labelings

Theorem (Worst-case Complexity)

Algorithm DecPLL ***takes*** $O(m_{\hat{\mathtt{A}}}\ell \log |\hat{\mathtt{A}}|) + |\hat{\mathtt{A}}|(m + n \log |\hat{\mathtt{A}}| + n\ell))$ *worst case time*[a]

[a] worse than PLL

Theorem (Minimality)

Algorithm DecPLL *computes* ***minimal*** *2–Hop Cover labelings*

Theorem (Fully Dynamic)

Algorithm DecPLL *can be* ***combined*** *with* IncPLL *to obtain a fully dynamic algorithm, namely* FulPLL, *that computes minimal 2–Hop Cover labelings under general updates occurring onto the graph*

Extensions

Weighted Graphs

- Use Dijkstra's **instead** of BFS
- Modify labels, priorities and comparisons over labels in order to consider **real-weighted edges**

Directed Graphs

- It is enough to define **two label sets** L_{in} and L_{out}
- The former stores a set of pairs (u, δ_{uv}), while the latter stores pairs (u, δ_{vu}), where $\delta_{uv} = d(u, v)$ and $\delta_{vu} = d(v, u)$
- A query from vertex s to vertex t is **answered** by

$$\textsc{Query}(s,t,L) = \begin{cases} \min\{\delta_{sv} + \delta_{vt} \mid v \in L_{out}(s) \cap L_{in}(t)\} & \\ & \text{If } L_{out}(s) \cap L_{in}(t) \neq \emptyset \\ \infty & \text{Otherwise} \end{cases}$$

Outline

Experiments

Setting & Executed Tests

- Real-World and Synthetic Dynamic Network **Instances**
- Real-World and Synthetic **Edge Modifications**[3]
- Wide combination of input parameters: various **densities**, **topologies**, number of **queries**, number of **modifications**, ...
- Basic Vertex Ordering: **Degree** + **Approx Betweenness**
- Dynamic Algorithms **against** PLL from scratch, to compare:
  - Computational **Effort** (i.e. time for building vs time for updating)
  - **Space** Occupancy (proxy for quality of labeling)
  - **Query** Time (proxy for quality of labeling)

[3] Known repositories Konect, SNAP, ...

Experiments – Inputs

Dataset	Network	|V|	|E|	AvgDeg	S	D	W
Eu-All (eua)	email	265 214	365 570	2.77	○	●	○
Twitter (twi)	social	465 017	834 797	3.59	○	●	○
Brightkite (bkt)	Location-based	58 228	214 078	7.35	○	○	○
Caida (cai)	Communication	32 000	40 204	2.51	○	○	●
Epinions (epn)	Social	131 828	841 372	12.76	○	●	○
Google (goo)	Web	875 713	4 322 051	9.87	○	●	○
BerkStan (wbs)	Web	685 230	7 600 595	22.18	○	●	○
WikiTalk (wtk)	Communication	2 394 385	4 659 565	4.19	○	●	○
Netherlands (nld)	Road	892 027	2 278 290	5.11	○	●	●
YouTube (ytb)	Social	1 134 890	2 987 624	5.26	○	○	○
FlickrImg (fli)	Meta-data	105 938	2 316 948	43.74	○	○	○
SimpWiki-En (swe)	Hyper-link	100 312	826 491	16.5	○	○	○
Wiki-It (itw)	Hyper-link	1 203 995	21 639 725	36.9	○	●	○
ForestFire-U (ffu)	Synthetic	2 000 000	14 908 267	14.91	●	○	○
ForestFire-D (ffd)	Synthetic	2 100 000	16 044 834	15.28	●	●	○
Gnutella (gnu)	p2p	36 682	88 328	4.82	○	●	○
AS-Skitter (ski)	Computer	1 696 415	11 095 298	13.08	○	○	○
FlickrLinks (fll)	Social	1 715 255	15 550 782	18.13	○	○	○
DBPedia (dbp)	Miscellaneous	3 966 924	13 820 853	6.97	○	●	○
Barabási-A. (baa)	Synthetic	631 912	1 000 772	3.17	●	○	●
Erdős-Rényi (erd)	Synthetic	50 000	6 252 811	250.11	●	○	●

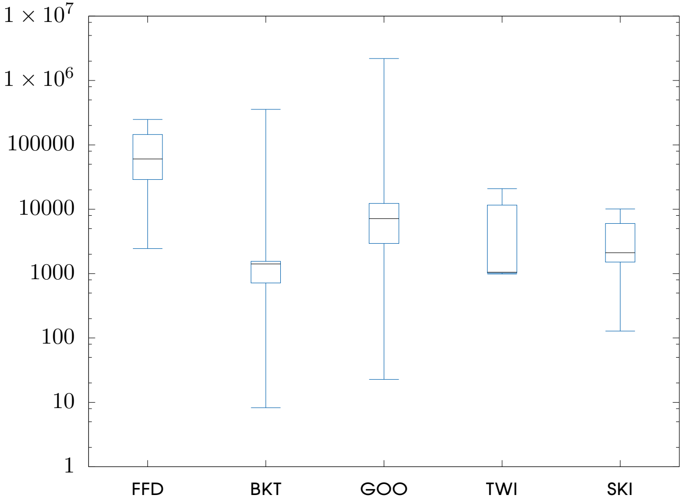
Experimental Results – DecPLL

Dataset	DEC workload						O
	CT (s)		LS (MB)		QT (μs)		
	PLL	DecPLL	PLL	DecPLL	PLL	DecPLL	
Eu-All	19.8	0.073	217	217	7.2	7.5	D
Twitter	25.4	0.018	390	390	7.3	7.3	D
Brightkite	98.7	0.328	81	81	23.1	25.7	D
Caida	1.1	0.497	24	23	39.5	40.1	D
Epinions	71.8	0.630	372	372	13.5	14.6	D
Google	3 950	4.27	3 862	3 862	39.9	57.2	D
BerkStan	2 510	0.639	1 659	1 659	31.4	28.9	D
WikiTalk	3 920	5.15	5 035	5 035	35.2	37.9	D
Netherlands	1 280	371	7 410	7 553	63.9	70.9	B
YouTube	2 720	104.0	2 899	2 899	43.9	60.4	D
FlickrImg	1 770	48.4	836	836	81.5	82.4	D
ForestFire-U	35 300	14.3	23 556	23 556	110	153	D
ForestFire-D	29 200	18.7	16 499	16 499	57.3	90.8	D
Gnutella	102	21.1	322	322	62.7	61.3	D
AS-Skitter	15 800	17.2	11 826	11 826	70.8	110.0	D
FlickrLinks	17 900	9.92	12 970	12 970	77.8	102.0	D
DBPedia	20 600	2.61	14 877	14 877	45.9	48.7	D
Barabási-A.	143	48.4	954	954	41.8	48.1	B
Erdős-Rényi	2 530	4.37	881	879	123	119	B

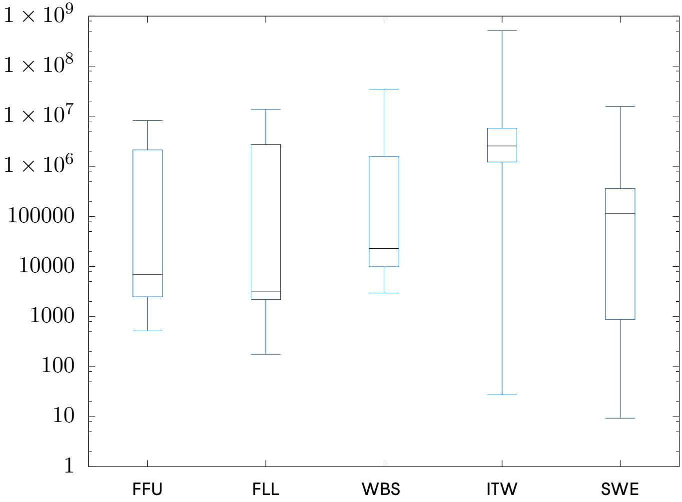
Experimental Results – FuLPLL

Dataset	FUL workload						O
	CT (s)		LS (MB)		QT (μs)		
	PLL	FuLPLL	PLL	FuLPLL	PLL	FuLPLL	
Eu-All	19.6	0.032	217	217	7.2	7.0	D
Twitter	25.6	0.007	390	390	7.3	7.3	D
Brightkite	95.3	0.217	81	81	22.7	26.4	D
Caida	1.21	0.331	24	25	39.9	41.5	D
Epinions	71.8	0.121	372	372	13.5	13.6	D
Google	3 950	0.566	3 862	3 872	39.4	52.3	D
BerkStan	2 480	0.176	1 659	1 659	30.9	27.3	D
WikiTalk	3 920	2.03	5 035	5 035	34.0	49.9	D
Netherlands	1 350	350	7 057	6 990	60.5	54.9	B
YouTube	2 650	79.3	2 899	2 899	40.6	55.6	D
FlickrImg	1 740	33.9	836	836	80.1	81.0	D
SimpWiki-En	78	0.232	181	180	47.4	49.2	D
Wiki-It	17 400	16.8	11 253	11 253	54.5	80.9	D
ForestFire-U	35 300	10.1	23 555	23 555	112	143	D
ForestFire-D	25 500	9.46	16 499	16 499	53.7	64.6	D
Gnutella	113	7.2	322	322	62.7	61.4	D
AS-Skitter	17 000	3.95	11 826	11 826	72.8	120.1	D
FlickrLinks	17 300	7.29	12 970	12 970	75.8	125.0	D
DBPedia	20 700	0.583	14 877	14 877	43.8	57.9	D
Barabási-A.	141	6.97	954	954	41.1	45.9	B
Erdős-Rényi	2 520	2.42	882	880	122	119	B

Experimental Results – Speed-up of DecPLL

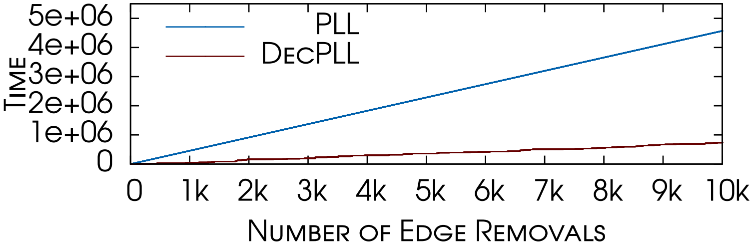


Experimental Results – Speed-up of FuLPLL



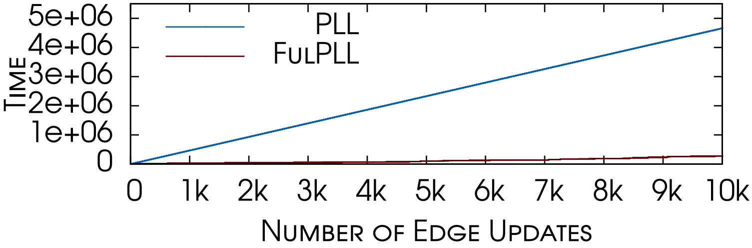
Experiments – Cumulative computational time of PLL vs DecPLL

Real-World network Google: increasing number of edge update operations



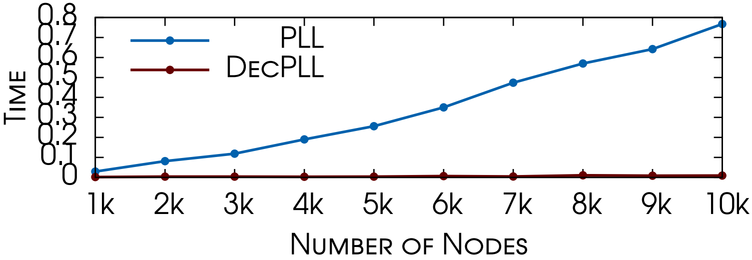
Experiments – Cumulative computational time of PLL vs FuLPLL

Real-World network Google: increasing number of edge update operations



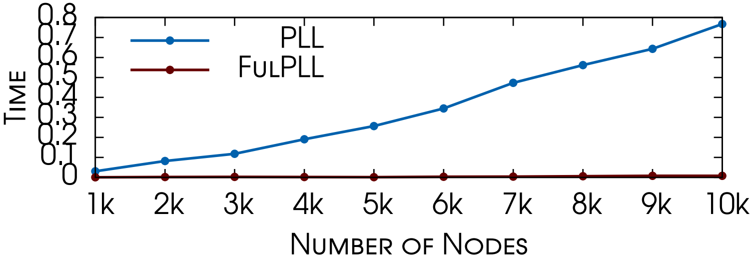
Experiments – Cumulative computational time of PLL vs DecPLL

Synthetic network ForestFire-U: increasing number of vertices



Experiments – Cumulative computational time of PLL vs FulPLL

Synthetic network ForestFire-U: increasing number of vertices



Outline

Conclusion #1 and Future Work

What it is done

- First non trivial **fully dynamic** scheme for distance queries on large-scale dynamic networks with **practical performance**
  - Answer queries in **microseconds** (no degradation)
  - Update indices in **few seconds**
  - No **increase** in **avg label size**

What it has to be done

- Improve practical **performance** of DecPLL in "bad instances"
- Build a **comprehensive theoretical background**
  - **Better characterize** trade-off approaches, such as PLL and its dynamic versions, from the **computational point of view**
  - **Fully distributed algorithms**

Future Work

What it has to be done

- **Extensions**
  - Support **historical queries** and **batches of updates**
  - **Parallel versions** of dynamic algorithms, if possible
  - More extensive **experimental** evaluation (weighted graphs)
- **Fault-Tolerant** Labelings?
  - Promptly react to transient failures by making the labeling **robust**
    - Add some "data" in advance
- **Stretched** labelings?
  - Relax optimality constraints

Many of the above stuffs are currently under investigation

Outline

2-Hop Cover Path-Reporting Labeling

A generalization, better suited for communication networks, WANs, MANET

Path-Reporting Labeling of a Graph G

- Given a graph $G = (V, E)$, let:
  - A **label** $P(v)$ is assigned to each vertex v of G
  - The **labeling** $P(G)$ of G is given by $\{P(v)\}_{v \in V}$
- A **path query** between two vertices s and t returns the **next hop on the shortest-path**
- Can be answered by simply **looking** at the labels $P(s)$ and $P(t)$ **of the two vertices** i.e. $\pi^G_{st} = f(P(s), P(t))$

2-Hop Cover Path-Reporting Labeling

Again 2–Hop Covers yield Path-Reporting Labelings

- A label is now intended as a **set of triples** (entries) $(u, \delta_{uv}, p(u, \pi^G_{uv}))$, where
  - u is a vertex in V
  - $\delta_{uv} = d_G(u, v)$
  - $p(u, \pi^G_{uv})$ is the *predecessor* of u within a shortest path π^G_{uv}
- Compute a **2–Hop Cover** $H(G)$ of the collection P of the shortest paths of G
- For each **hop** $(h, u) \in H$ add **entry** $(v, w(h), p(h))$ to $L(u)$
- Where v is the other **endpoint** of h, $w(h)$ is its weight, and $p(h)$ is the **predecessor** of u on h

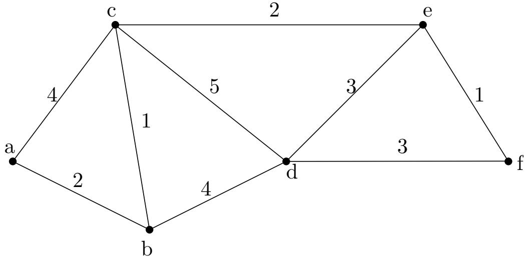
2-Hop Cover Path-Reporting Labeling

Path-Reporting Labeling of a Graph G

A **path query** from s to t is defined as follows:

$$\textsc{pQuery}(s,t,P) = \begin{cases} \langle h, \delta_{hs}, p(s, \pi^G_{hs})\rangle \mid h = \operatorname{argmin}\{\delta_{vs} + \delta_{vt} \mid v \in P(s) \wedge v \in P(t)\} & \text{If } P(s) \cap P(t) \neq \emptyset \\ \emptyset & \text{Otherwise} \end{cases}$$

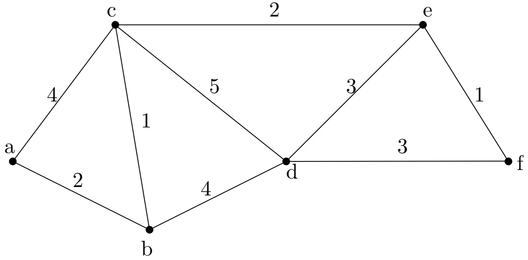
Example of Path-Reporting Labeling



Example of Path-Reporting Labeling

$L(a) = \{(b,2),(c,3),(d,6)\}$

$L(f) = \{(c,3),(d,3),(e,1)\}$



Example of Path-Reporting Labeling

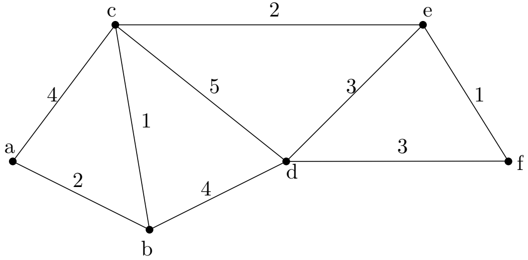
$L(a)$ $L(f)$

$(c, 3)$ $(c, 3)$
$+ \longrightarrow 6$

$(d, 6)$ $(d, 3)$
$+ \longrightarrow 9$

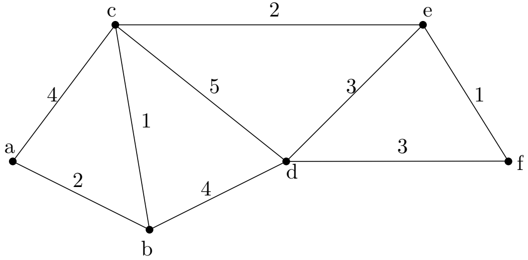
c is the *hub* and 6 is the distance between a and f.

$L(a) = \{(b, 2), (c, 3), (d, 6)\}$ $L(f) = \{(c, 3), (d, 3), (e, 1)\}$



Example of Path-Reporting Labeling

$L(a) = \{(b, 2, b), (c, 3, b), (d, 6, b)\}$ $L(f) = \{(c, 3, e), (d, 3, d), (e, 1, e)\}$



Example of Path-Reporting Labeling

$L(a)$ $\quad$ $L(f)$

$(c, 3, b)$ $\quad$ $(c, 3, e)$

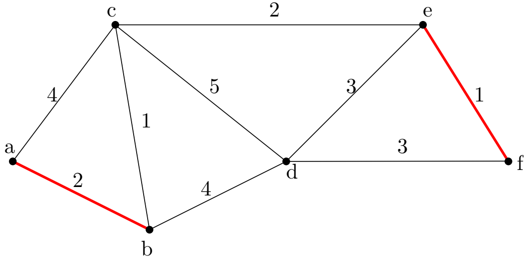
$+ \longrightarrow 6$

$(d, 6, b)$ $\quad$ $(d, 3, d)$

$+ \longrightarrow 9$

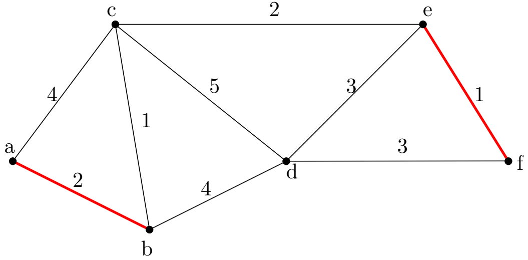
c is the *hub*, 6 is the distance between a and f, b and e are the next hops.

$L(a) = \{(b, 2, b), (c, 3, b), (d, 6, b)\}$ $\qquad$ $L(f) = \{(c, 3, e), (d, 3, d), (e, 1, e)\}$



Example of Path-Reporting Labeling

$L(b) = \{\ldots (c, 1, b), (d, 4, d), \ldots\}$ $\qquad$ $L(e) = \{\ldots, (c, 2, e), (d, 3, d), \ldots\}$

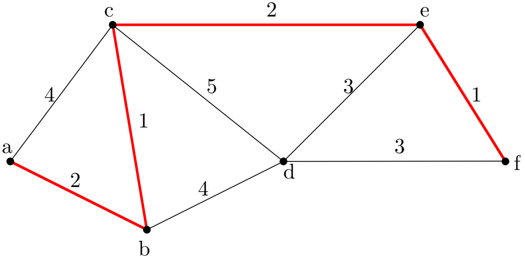


Example of Path-Reporting Labeling

$$\begin{array}{ll}
L(b) & L(e) \\
(c, 1, b) & (c, 2, e) \\
\quad + & \longrightarrow 3 \\
(d, 4, d) & (d, 3, d) \\
\quad + & \longrightarrow 7
\end{array}$$

c is the *hub*, 3 is the distance between a and f, c is the next hop.

$L(b) = \{\ldots (c, 1, b), (d, 4, d), \ldots\}$ $\qquad$ $L(e) = \{\ldots, (c, 2, e), (d, 3, d), \ldots\}$



State-tof-the-Art

Known Limits

- We already know that computing a **compact** 2-Hop Cover is hard
- An **approximation** algorithm is known, but it is not practical for large graphs
- PLL takes **cubic** time in the worst case
- Modern networks are **prone** to (often transient) **failures**
  - A link in a network can temporary be unavailable
  - A road can be blocked

State-tof-the-Art

Limits of Dynamic Algos

- Dynamic algorithms **update times** are still far to be used in a **real-time** applications (e.g. routing)
- **Fault-Tolerant** approaches are **advisable**

Fault-tolerant scheme

- An approach that **allows** to answer to queries **even in presence of a number of (transient) graph failure operations** (e.g., edge or vertex removals)
- Usually achieved by suitably **enriching** the underlying data structure and by accordingly **modifying** the query strategy to consider such enrichment

State-tof-the-Art

Limits of Exact Fault Tolerance

- Computing an **exact** fault-tolerant labeling is not **feasible** in terms of both **space** and **time**
  - If we want to tolerate the **failure of a single edge** at a time
  - It can be shown that we need to **store** in the worst case m times the space of **a single labeling**
  - And to spend m times the **time** taken by PLL in the worst case
- **Approximation**
  - Reasonable compromise: **relax the optimality constraint**
  - Devise **more compact** schemes that return **approximate** (a.k.a. stretched) distances (shortest paths, resp.)
  - **Stretch:** ratio between quality of optimal solution and quality of returned solution

Recent Results

Fault-Tolerant approach for 2-Hop Cover Labeling

- **Recently proposed** by our group (unweighted graphs)
- k**-Edge Fault-Tolerant Path-Reporting Labeling scheme** (k-EFTPL)
- Exhibits the following **properties** (should any set of k edges fail):
- **Time/Space overhead**
  - for $k = 1, 2, 3$ and G at least $(k+1)$-edge connected, the enrichment takes $O(m+n)$, $O(n^2)$ and $O(n^3)$ additional **time**, resp., and $O(n)$ additional **space**;
  - for $k > 3$ and G at least $(2k+2)$-edge connected, the enrichment takes $O(k^2n^2)$ additional **time** and $O(kn)$ additional **space**
- **Query time linear** in the length of the retrieved path (as non-fault-tolerant)
- **Linear stretch** (in n and k)

Independent Trees

How to make a labeling resistent to the failure of k edges

- Exploiting **Edge-Independent trees**
- A well-known concept from the 80's

Edge-Independent Trees

Given a graph $G = (V, E)$ and a distinguished **root** vertex $r \in V$, then $\mathtt{IT} = \{T_1, T_2, \ldots, T_q\}$ is a collection of q **edge-independent spanning trees** of G if and only if

- for each vertex $v \in V$, and for each $i \neq j$, $\pi_{rv}^{T_i}$ and $\pi_{rv}^{T_j}$ are **pairwise edge-disjoint paths**, i.e. they **do not share any edge**

Independent Trees

Theorem (Menger's Theorem)

Let $\mathtt{IT}$ *be a graph obtained by merging a collection of* $q+1$ *edge-independent spanning trees* $\{T_i\}_{i=1,2,\ldots,q+1}$ *of a graph* G*. Then,* $\mathtt{IT}$ *is* $(q+1)$*-edge connected*

k-Edge Fault-Tolerant Path-Reporting Labeling

How to build a k-EFTPL

1. Start from a 2-Hop Cover Path Labeling
2. Compute $k+1$ **independent trees**
3. Enrich each vertex's label by adding k **tree entries**
    - A tree entry contains the **parent (aka next-hop)** of the node in the corresponding k–th tree

On the availability of $k + 1$ Ind-Trees

- Depending on the value of k, **different approaches** can be used to build the $k + 1$ edge-independent trees
- Clearly, **a necessary condition** to guarantee that any pair of vertices remains **connected** even in presence of k edge failures is that G **is at least** $(k + 1)$-edge connected

On the computation of $k+1$ Ind-Trees

1. If $k \in \{1,2,3\}$ and G is $(k+1)$-edge connected, $k+1$ edge-independent trees **can be computed in polynomial time**, with a time complexity of $O(m+n)$, $O(n^2)$ and $O(n^3)$, resp.
    - A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. Inf. Comput., 79(1):43–59, 1988
    - J. Cheriyan and S. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. Journal of Algorithms 9(4):507–537, 1988
    - S. Curran, O. Lee, and X. Yu. Finding four independent trees. SIAM J. Comput., 35(5):1023–1058, 2006
2. If $k > 3$ and G is h-edge connected, with $k+1 \leq h \leq 2k+1$
    - To the best of our knowledge **it is not known** how to build $k+1$ edge-independent trees in polynomial time

On the availability of $k+1$ Ind-Trees

1. If $k > 3$ and G is h-edge connected, with $k+1 \leq h \leq 2k+1$ and G is at least $(2k+2)$-edge connected
   - We can build $k+1$ *edge-disjoint* spanning trees of G, which are clearly also **edge-independent**, in $O(k^2 n^2)$ time by using the approach
     - J. Roskind and R. E. Tarjan. A note on finding minimum-cost edge-disjoint spanning trees. Mathematics of Operations Research, 10(4):701–708, 1985

The k-EFTPL

How to query a k-EFTPL (note: it is distributed)

1. Let x and y be the **endpoints** of our query
2. Starting from x, compute the next-hop **via path-query**
3. If the next-hop is **not available**, start using **tree entries** to reach the root
4. Symmetrically, do the same from y

Theorem (From Menger's Th)

There always exists at least

- *a path from the root toward vertex* x
- *a path from the root toward vertex* y

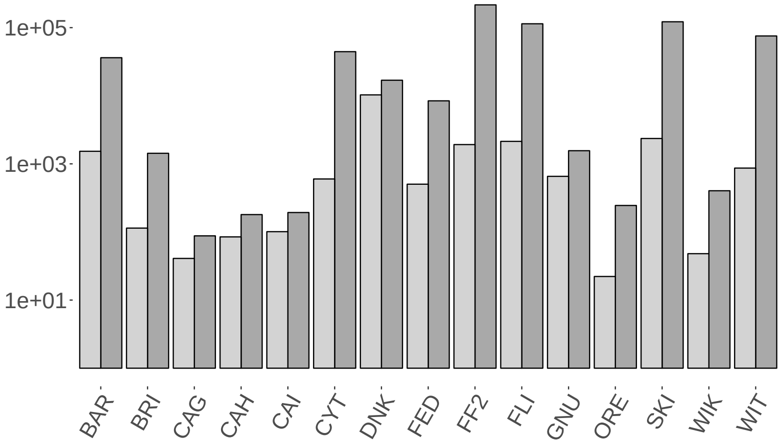
Experimental setting

- The above solutions have **replacement** paths which can be **linearly** (in n and k) **stretched** (as compared to new shortest paths in the surviving graph)
- What about in **practice**?
- Experiments to assess the **performance** of the approach:
  - Real and synthetic networks
  - Implemented and run both PLL and KHL for each networkm for $k = 1$
  - Performed $10k$ queries as follows
    - Randomly **remove** an edge e
    - Let e be on the shortest path with probability $p = 5/100$
    - Run a **POI rerouting scheme** to compare (the only known distributed fault-tolerant)
  - Measured query time and stretch

Results: Space and Preprocessing Time

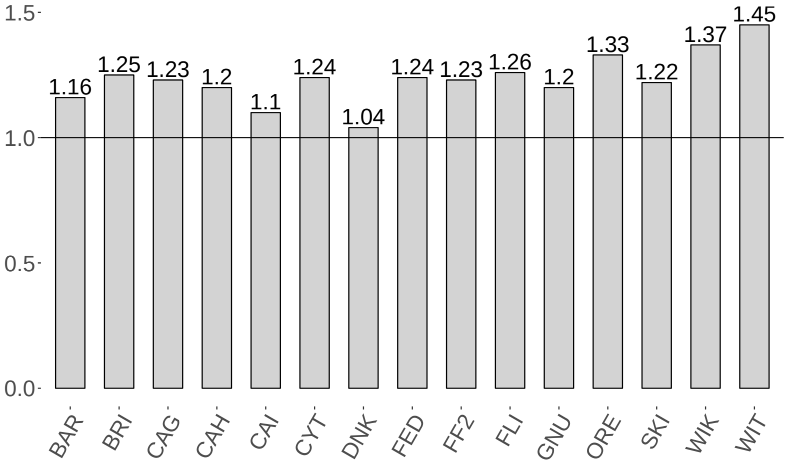
Network	$\lvert V \rvert$	$\lvert E \rvert$	**time** (seconds)		**space per vertex** (bytes)	
			PLL	KHL	$P(G)$	$T(G,1)$
BARABASI	365 488	734 347	68 500	6.41	5 198	18
BRIGHTKITE	33 187	188 577	5 680	1.75	2 326	18
CA-GRQC	2 651	10 480	499	0.03	1 271	18
CA-HEPTH	5 898	20 983	1 130	0.03	2 085	18
CAIDA	6 855	13 341	1 650	0.02	1 412	18
COM-YOUTUBE	452 060	2 295 072	66 200	5 090	4 136	18
DENMARK	252 416	320 914	147 000	0.75	13 152	18
FLICKREDGES	105 512	2 316 450	2 180	165	12 415	18
FORESTFIRE	1 178 888	13 849 776	212 000	16 100	17 983	18
FLICKRLINKS	704 985	14 501 930	125 000	17 700	12 525	18
OREGON	7 218	19 448	638	2.54	286	18
SKITTER	1 443 769	10 830 987	197 000	11 100	10 666	18
WIKIVOTE	4 786	98 456	751	1.12	1 890	18
WIKITALK	622 315	2 889 703	47 700	38 700	2 951	18

Results: Query Time



Query time of our approach (light gray) versus query time of POI rerouting.

Results: Stretch



Estimation of stretch factor based on 10.000 measures.

Outline

Conclusion #2 and Future Work

Conclusion

- First 2–Hop Cover distance/path-reporting labeling scheme in the **fault-tolerant setting**
- **Compact**, small query time, can be computed **quickly** and exhibits **worst case linear stretch**
- Practically effective (through an extensive experimental evaluation, **surprisingly small stretch**)

Future Work

- Implement the algorithms for $k = 2$ and $k = 3$
- Deeper investigation of the case of $k > 3$ using other techniques
- Can we design a scheme with a **better theoretical guarantee?**

Q&A

mattia.demidio@gssi.it
www.mattiademidio.com