A new fully dynamic algorithm for distributed shortest paths and its experimental evaluation

Serafino Cicerone **Gianlorenzo D'Angelo**
Gabriele Di Stefano Daniele Frigioni Vinicio Maurizio

Deptartment of Electrical and Information Engineering
University of L'Aquila, Italy
{serafino.cicerone, gianlorenzo.dangelo, gabriele.distefano, daniele.frigioni}@univaq.it, vinicio.maurizio@cc.univaq.it

Motivation

Update shortest paths in a graph representing a distributed asynchronous system (e.g. the Internet) when edge changes occur

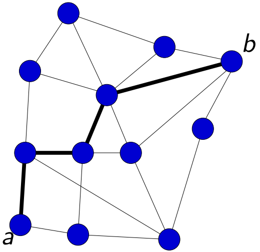
The changes can occur in an unpredictable way

- Concurrent updates
- Asynchronous networks

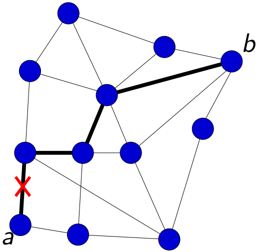
Admitted edge changes:

- weight increase/delete
- weight decrease/insert

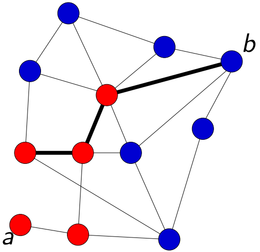
Concurrent updates



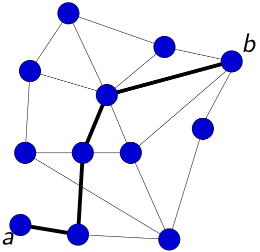
Concurrent updates



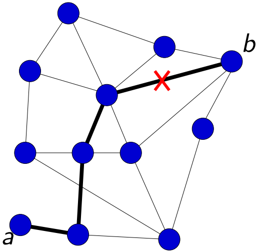
Concurrent updates



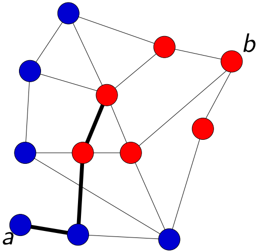
Concurrent updates



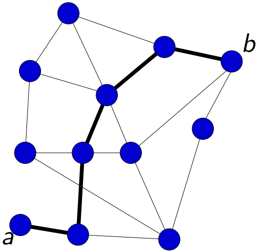
Concurrent updates



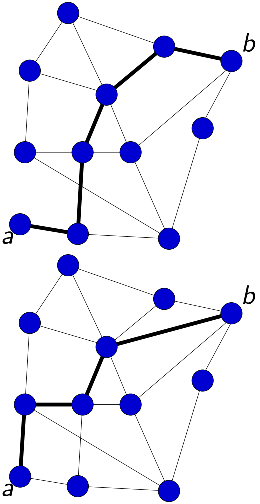
Concurrent updates



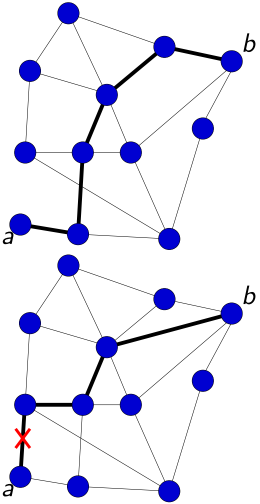
Concurrent updates



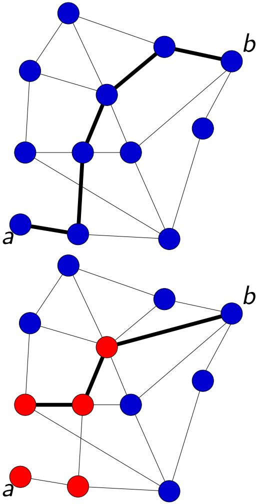
Concurrent updates



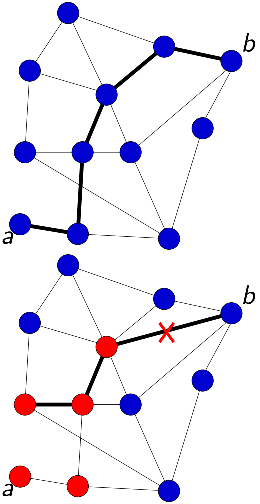
Concurrent updates



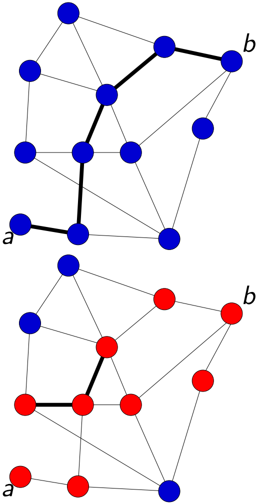
Concurrent updates



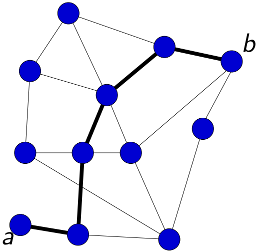
Concurrent updates

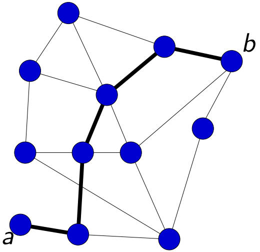


Concurrent updates



Concurrent updates





Outline

Previous works

New fully dynamic algorithm

Simulation environment

Input data

Experimental results

Conclusion and future research

Outline

All previous algorithms suffer of one of the following drawbacks

- Not concurrent [Cicerone et al. 2003, Garcia-Lunes-Aceves 1993, Italiano 1991, Ramarao et al. 1992]
- Looping or count-to-infinity or slow convergence in case of weight increase/delete operations [Humblet 1991]
- Partially Dynamic [Cicerone et al. 2010]

Algorithm in this paper

- Concurrent
- Heuristically reduces the cases where looping or count-to-infinity occurs
- Experimentally fast
- Fully Dynamic

Outline

Data Structure

Each node v stores a routing table:
For each source s:

- $\mathtt{D}[v, s]$: the distance to s
- $\mathtt{VIA}[v, s]$: the set of neighbors of v on a shortest path to s

Messages

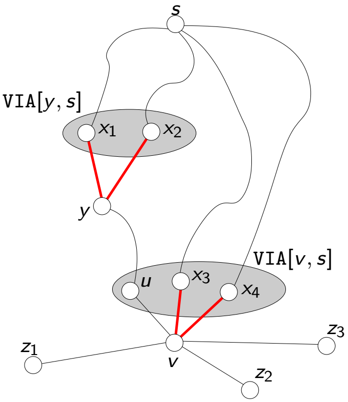
Three kind of messages

- *increase*: notifies that a distance increased
- *decrease*: notifies that a distance decreased
- *get-dist*: asks for a distance to a neighbor

We analyze the behavior of node v in three cases:

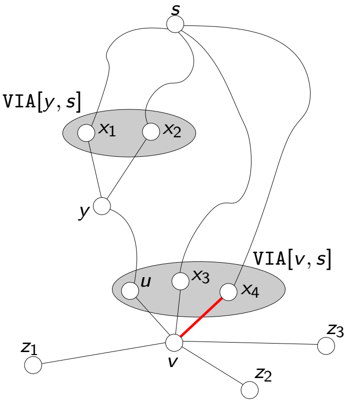
- The distance from v to s increases
- The distance from v to s decreases
- The distance from v to s does not change

The distance from v to s increases



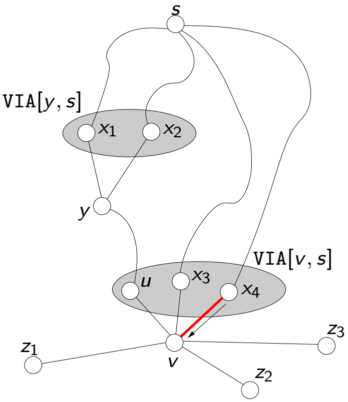
weight increase operations on the red edges

The distance from v to s increases



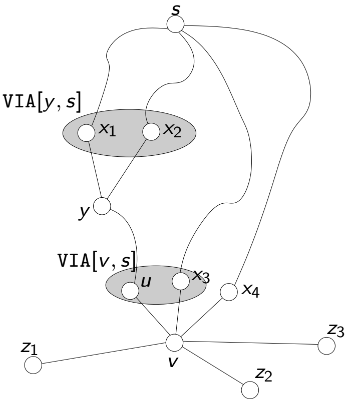
weight increase on (v, x_4) occurs

The distance from v to s increases



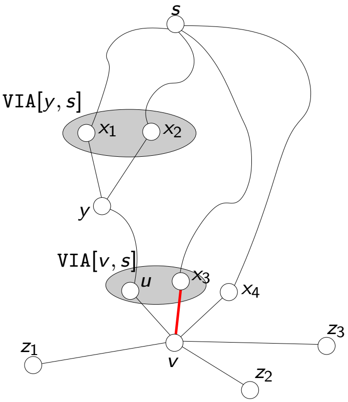
x_4 sends a *increase* message to v

The distance from v to s increases



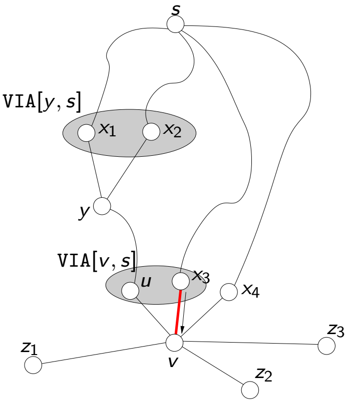
v removes x_4 from $\mathtt{VIA}[v, s]$

The distance from v to s increases



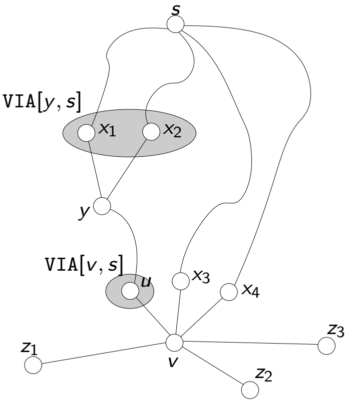
weight increase on (v, x_3) occurs

The distance from v to s increases



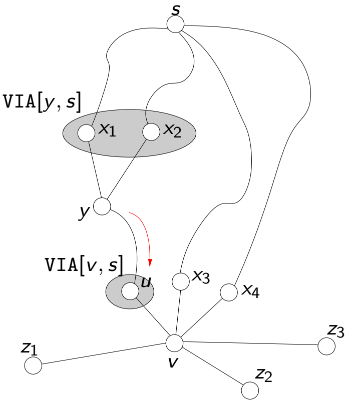
x_3 sends a *increase* message to v

The distance from v to s increases



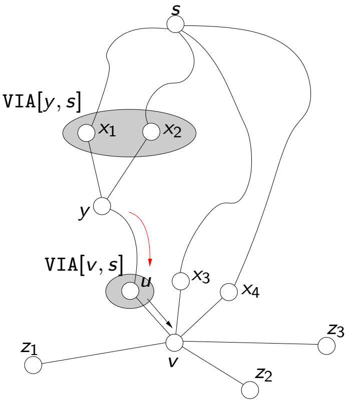
v removes x_3 from VIA[v, s]

The distance from v to s increases



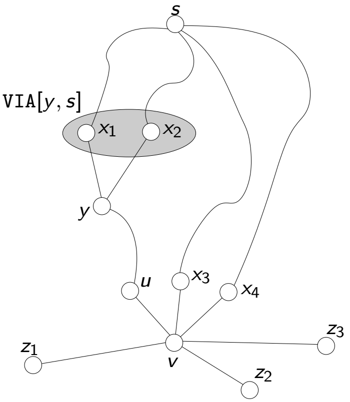
Weight increases on (x_1, y) and (x_2, y) are propagated to u by *increase* messages

The distance from v to s increases



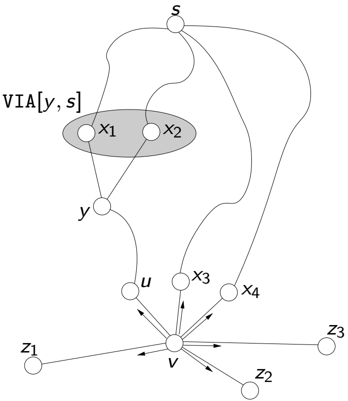
u sends a *increase* message to v

The distance from v to s increases



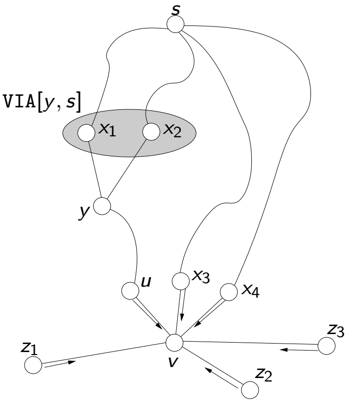
v removes u from $\texttt{VIA}[v, s]$: now $\texttt{VIA}[v, s]$ is empty

The distance from v to s increases



v asks its neighbors for their distances to s

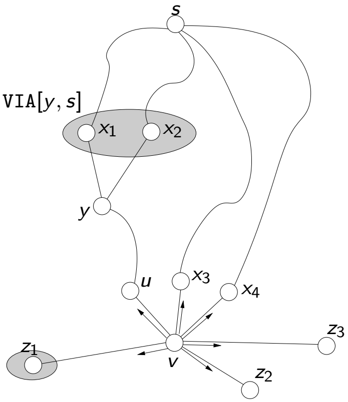
The distance from v to s increases



Neighbor z of v communicates its distance to v, eventually sending ∞ if

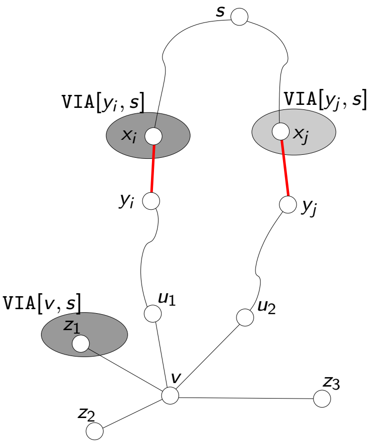
- $\texttt{VIA}[z, s] \equiv v$
- z is computing its distance to s (due to some other weight increase operations)

The distance from v to s increases



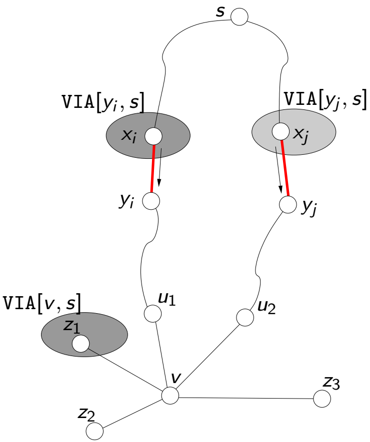
v computes its minimal distance to s and propagate the modification by sending *increase* messages to its neighbors

The distance from v to s decreases



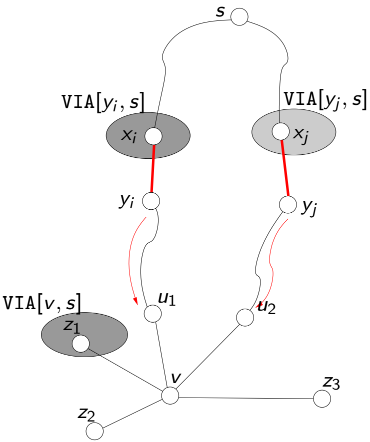
weight decrease operations on the red edges

The distance from v to s decreases



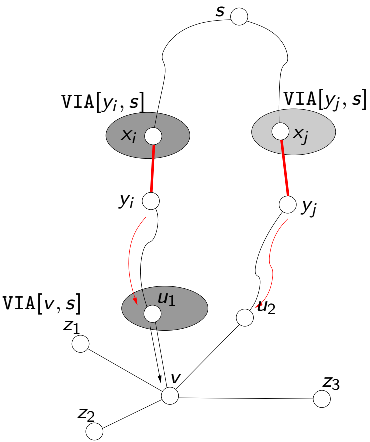
Nodes x_i and x_j sends *decrease* messages to y_i and y_j, respectively

The distance from v to s decreases



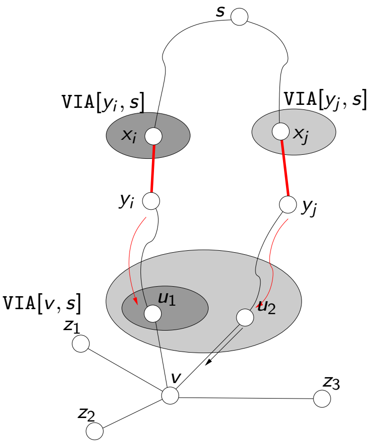
decrease messages are propagated to node v

The distance from v to s decreases



When v receives the *decrease* message from u_1, it sets $\texttt{VIA}[v, s] = \{u_1\}$

The distance from v to s decreases



When v receives the *decrease* message from u_2, it sets $\mathtt{VIA}[v,s] = \mathtt{VIA}[v,s] \cup \{u_2\}$

The distance from v to s does not change

Either the algorithm does not change the routing table or one of the two previous cases occur

Outline

- OMNeT++ environment version 4.0p1
- Implemetend algorithms:
  - New algorithm (ConFu)
  - Distributed Bellman-Ford algorithm (BF): a node v updates its distance to a node s, by executing the iteration

$$\mathtt{D}[v,s] := min_{u \in N(v)} \{ w(v,u) + \mathtt{D}[u,s] \}$$

    using the latest estimated distance $\mathtt{D}[u,s]$ received from a neighbor $u \in N(v)$ and the latest status of its links $w(v,u)$ Each node needs to store the latest estimated distance $\mathtt{D}[u,s]$ for each neighbor u and each node s

Outline

Two graph classes:

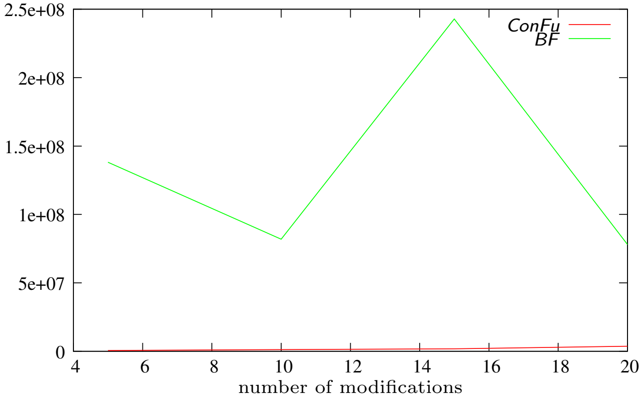
- CAIDA (Cooperative Association for Internet Data Analysis) topology dataset
- Erdös-Rényi random graphs

- The CAIDA dataset is collected by a globally distributed set of monitors which send probe messages to IP addresses chosen at random
- For each destination the path from the source monitor to the destination is collected: the set of IP addresses and the Round Trip Times (RTT) of each node in the path
- We obtained a weighted undirected graph G_{IP} where a node represents an IP address, edges represent links and weights are given by RTTs
- We selected subgraphs of 5000 nodes
- We evaluated the algorithms over sets of $5, 10, \ldots, 100$ concurrent edge weight updates, each weight updates change the weight of a random selected edge by a percentage value randomly chosen in $[50\%, 150\%]$

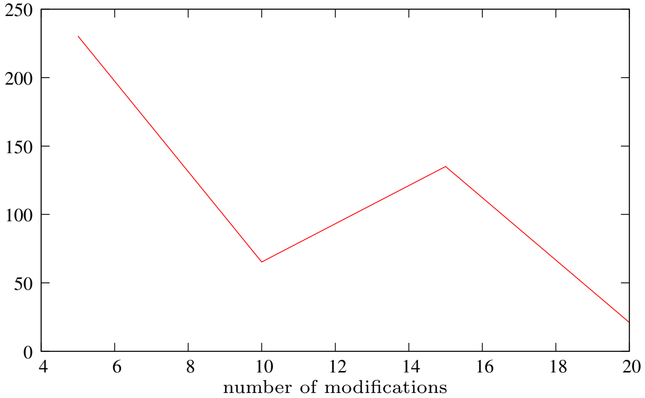
- G_{IP} is very sparse
- We generated Erdös-Rényi random graphs G_{rand} of 1000 nodes with variable number of edges and random edge weights
- We evaluated the algorithms over sets of $30, 100, 1000$ concurrent edge weight updates

Outline

G_{IP} number of messages



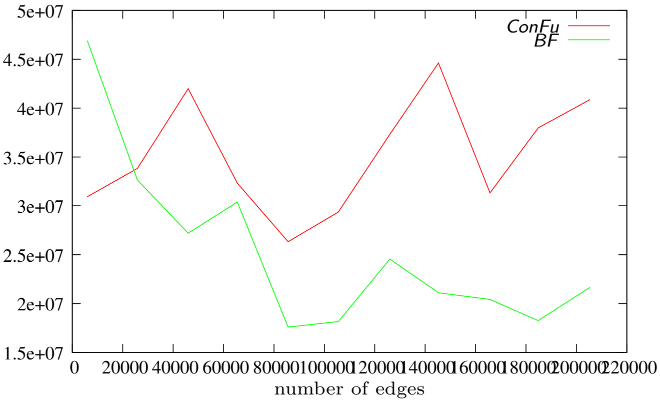
G_{IP} number of messages ratio



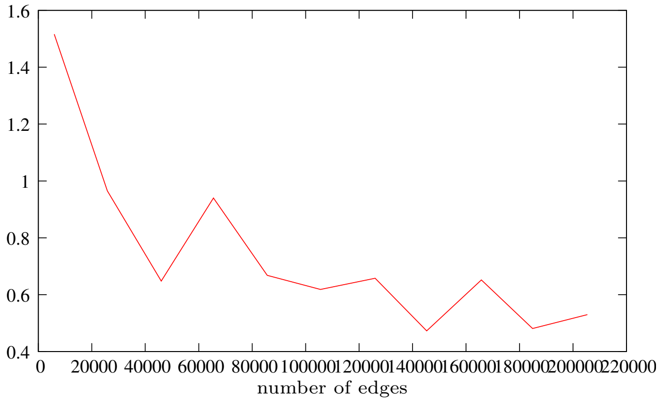
G_{IP} space occupancy per node

- Average:
  - ConFu requires 40000 bytes per node
  - Bellman-Ford requires 44436 bytes per node
- Worst case:
  - ConFu requires 40084 bytes per node
  - Bellman-Ford requires 4M bytes per node

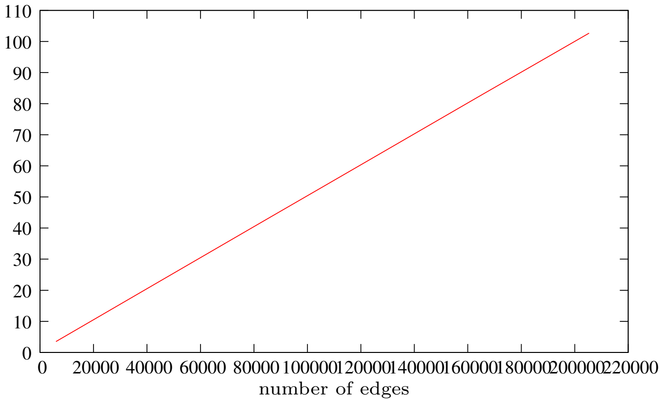
G_{rand} number of messages



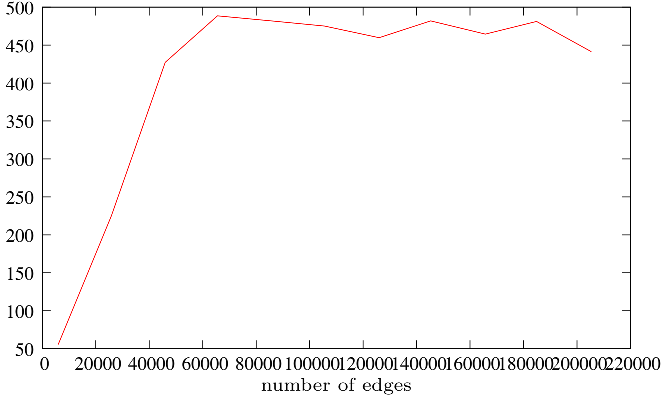
G_{rand} number of messages ratio



G_{rand} average space occupancy per node



G_{rand} worst case space occupancy per node



Outline

Previous works

New fully dynamic algorithm

Simulation environment

Input data

Experimental results

Conclusion and future research

We proposed an algorithm:

- Concurrent
- Fully Dynamic
- Experimentally better than the BellmanFord algorithm either in number of messages or in space

Further research:

- Store more information in the nodes in order to reduce the number of messages
- Identify the information to store by exploiting the structure of the Internet topology (core-like structure and power low distribution)
- Test the algorithms on random graphs that model the Internet topology