Powerful and Efficient Bulk Shortest-Path Queries:
Cypher language extension & Giraph implementation

Peter Rutgers, Claudio Martella, Spyros Voulgaris, Peter Boncz

VU University Amsterdam
Goal and Contributions

- Context: Shortest-path queries in Giraph

- Desired functionality
  - Edge weights (monotonic cost function!)
  - Multiple sources and destinations ("bulk" queries)
  - Top-N shortest paths for each pair
  - Filters on path edges and vertices
  - Provide both paths and their costs

- Our contributions are twofold:
  - Cypher language extension
  - Efficient top-N shortest path algorithm design & implementation on Giraph
Outline

Cypher Extension

Algorithms and Implementation

Evaluation

Conclusions
Shortest Paths in Cypher [1/2]

- No weighted paths!
- No top-N shortest paths!
- Conditions in WHERE applied after finding path
  - Could result in empty answer!
Shortest Paths in Cypher [1/2]

- No weighted paths!
- No top-N shortest paths!
- Conditions in WHERE applied after finding path
  - Could result in empty answer!

MATCH path=shortestPath( (a)-[*]->(b) )
WHERE none(x in nodes(path) WHERE x.danger)
RETURN path, length(path);
Shortest Paths in Cypher [2/2]

**MATCH** `path=(a)-[r*]->(b)`
**WHERE** `none(x in nodes(path) WHERE x.danger)`
**RETURN** `path,
    reduce(sum=0, x IN r | sum=sum+x.dist*x.speed) AS len`
**ORDER BY** `len DESC`
**LIMIT** `5`

- Matches *all* paths! Expensive!
- Orders all paths that remain after the WHERE condition
- Complex query for humans
- Complex query for the query planner
  - Hard to detect and optimize
Proposed language extension

MATCH path=(src)-[e* | sel(e)]->(dst)
CHEAPEST n SUM cost(e) AS d

- Selector applied before WHERE condition (optional)
- Multiple paths (top-N) for each pair
- Custom cost function
- AS keyword to bind cost to variable
- Supports bulk queries (multiple sources / multiple destinations)
Example

- Suppose you are building a navigation system
  - Some nodes are of type Src, some of type Dst
  - Some nodes have the property danger
  - The cost of each segment is the distance times the speed limit

- You can get the top-3 cheapest routes by the following simple query:

```sql
MATCH path=(a:Src)-[e* | not(endNode(e).danger)]->(b:Dst)
CHEAPEST 3 SUM e.dist * e.speed AS len
RETURN a, b, path, len
```
Outline

Cypher Extension

Algorithms and Implementation

Evaluation

Conclusions
The Lighthouse Project

- Cypher-based declarative language, query planning and execution, for Apache Giraph.

  - **Parser**
    - Turns Cypher query into query graph

  - **Planner**
    - Builds query plan (tree of operators)

  - **Execution engine**
    - Runs query plan on Giraph`
Top-N Shortest Path

- We need to compute both the **cost** and the **path** itself

- Basic algorithm
  - Each node maintains the top-N paths (and costs) found so far
  - In each step, each node propagates all its updates along all its outgoing edges
  - When a node has received no updates in a step, it votes to halt
  - The algorithm terminates when they all vote to halt
Top-N Shortest Path

N=5

A

1: AB
0: A
3: AD
7: AF
1: AC

B

C

D

E

F

G

N=5

1: AB
0: A
3: AD
7: AF
1: AC

1

1

3

1

3

1

2

2

1

3

1

1

7

3

1

2

1

3

2

1
Top-N Shortest Path

N=5

0: A

1: AB

2: ABE

3: ACE

1: AC

2: ABC

4: ACF

3: AD

6: ADF

7: AF

9: AFG
Top-N Shortest Path

N=5

Graph with nodes A, B, C, D, E, F, G and paths:

1: AB
2: ABE
3: ACF
4: ADF
5: AF
6: AFG

Path weights:

A to B: 1
B to A: 1
B to C: 1
C to A: 1
C to E: 2
E to A: 1
A to D: 3
D to A: 3
D to F: 3
F to A: 4
F to C: 3
C to E: 2
E to G: 1
G to A: 9
A to G: 1

N=5
Top-N Shortest Path

N=5

A
0: A
B
1: AB
C
1: AC
2: ABC
D
3: AD
E
2: ABE
3: ACE
4: ABCE
F
4: ACF
5: ABCF
6: ADF
7: AF
G
3: ABEG
4: ACEG
5: ABCEG
6: ACFG
7: ABCFG
Can we do better?!

- One problem:
  - Memory footprint is too high
  - Paths passed around are too long

- The solution:
  - No need to pass and store the entire path
  - Store only *predecessor node ID* and *cost to date* per path
  - Less communication, lower runtime!

- The price to pay?
  - An extra phase for path reconstruction
Top-N Shortest Path

N=5

A
0: A
3: AD

B
1: AB

C
1: AC
2: ABC

D

E
2: ABE
3: ACE
4: ABCE

F
4: ACF
5: ABCF
6: ADF
7: AF

G
3: ABEG
4: ACEG
5: ABCEG
6: ACFG
7: ABCFG

N=5
Top-N Shortest Path

N=5

A
0: A
1: AB
2: ABE
3: ACE
4: ABCE

B
1: AB

C
1: AC
2: ABC

D
3: AD

E
2: ABE
3: ACE
4: ABCE

F
4: ACF
5: ABCF
6: ADF
7: AF

G
3: ABEG
4: ACEG
5: ABCEG
6: ACFG
7: ABCFG
Top-N Shortest Path Reconstruction
Top-N Shortest Path Reconstruction

1: A
0: A

2: B

3: C

4: C

5: C

6: D

7: A

EG: 4,5

FG: 6,7

EG: 3

ED: 4,5

EG: 3

ED: 4

ED: 5

ED: 6

ED: 7
Top-N Shortest Path Reconstruction

ABEG: 3
ACEG: 4
ACFG: 6

A
0: A
3: A

B
1: A
1: A

C
1: A
2: B

D
3: A

E
2: B
3: C
4: C

F
4: C
5: C
6: D
7: A

G
3: E
4: E
5: E
6: F
7: F

CEG: 4
CFG: 6

BEG: 3
CEG: 5
CFG: 7
Top-N Shortest Path Reconstruction

ABEG: 3
ACEG: 4
ACFG: 6
ABCEG: 5
ABCFG: 7

Diagram:

- A
  - 0: A
  - 1: A
  - 2: B
  - 3: E
  - 4: C
  - 5: C
  - 6: F
  - 7: F

- B
  - 1: A

- C
  - 1: A
  - 2: B

- D
  - 3: A

- E
  - 2: B
  - 3: C
  - 4: C

- F
  - 4: C
  - 5: C
  - 6: D
  - 7: A

- G
  - 3: E
  - 4: E
  - 5: E
  - 6: F
  - 7: F
Can we do even better???

The problem:
- In the first few supersteps, some **expensive, yet short, paths** are propagated aggressively.
- Unnecessary resource consumption

Solution:
- **Postpone exploration!**
- Reduce the exponential growth of exploration in the first supersteps.
- Delay propagating paths that “appear” to be not-too-cheap.

How?
- Place paths in buckets \([0, \Delta]\), \([\Delta, 2\Delta]\), ... and suppress the propagation of paths of bucket \(i\) until superstep \(i\).
Pruning via Landmarks

- To further confine unnecessary exploration, we prune based on upper cost bounds.

- We use landmarks:
  - Selected nodes $X_i$,
  - For each src/dst pair AB, we compute $|AX_i|$ and $|X_iB|$.
  - $|AX_i| + |X_iB|$ forms an upper bound for $|AB|$.
Outline

Cypher Extension

Algorithms and Implementation

Evaluation

Conclusions
Overall scalability

- LDBC - SF10 trace
  - Scale factor 10, with 72,949 vertices and 4,641,430 edges

<table>
<thead>
<tr>
<th>#workers</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>&gt;1000</td>
<td>492</td>
<td>222</td>
<td>126</td>
<td>89</td>
<td>72</td>
</tr>
</tbody>
</table>
Postponing Path Exploration (Delta stepping)

- Rnd1K trace: Erdos-Renyi, 1000 vertices, 50K edges
- One-to-all, top-5 shortest paths

- Total runtime drops from 35sec to 25sec
- Total #bytes sent drops by 49%
Effect of Multiphase Approach

- Rnd1K trace: 1K nodes, 50K edges

<table>
<thead>
<tr>
<th></th>
<th>bytes</th>
<th>messages</th>
<th>supersteps</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>182,204,626</td>
<td>402628</td>
<td>18</td>
<td>35.92 sec</td>
</tr>
<tr>
<td>Multiphase</td>
<td>83,926,097</td>
<td>402749</td>
<td>28 (18+10)</td>
<td>27.132 sec</td>
</tr>
</tbody>
</table>
LDBC - SF1 trace: 10,993 vertices, 451K edges
25 random sources, all nodes as destinations
Top-5 shortest paths
2 landmarks (the highest degree nodes)

Actual computation drops by ~40%
Landmark estimation takes too long
Outline

Cypher Extension

Algorithms and Implementation

Evaluation

Conclusions
Conclusions

- We proposed new Cypher syntax that allows
  - Flexible edge weights
  - Flexible filter conditions over these
  - Top-N queries

- This syntax is concise, and guarantees that efficient (pruning) algorithms can be employed by the query planner

- We proposed efficient shortest path algorithms
  - Number of messages and data transferred are substantially reduced
  - Much improved memory footprint
  - However, they do not necessarily reduce runtime
  - Landmarks do not always improve runtime