OCITS-Seminar Implementation of link prediction on facebook data

using Dgraph for data storage

Konstantin Igin
Matr. Nr. 375455
31.07.2020
SS2020

Contents

- Motivation
- Introduction
- Environment setup
    - Dgraph setup
    - Dataset selection, processing and upload
- Implementation
- Results
- Conclusion/Outlook

Motivation

- Amount of graph structure applications grows continuously

Motivation

- Amount of graph structure applications grows continuously
- Processing of data stored using traditional approaches (SQL) into an actual graph structure might require extra ressources
- Data Storages able to store data in a more native format required

Motivation

- Amount of graph structure applications grows continuously
- Processing of data stored using traditional approaches (SQL) into an actual graph structure might require extra ressources
- Data Storages able to store data in a more native format required
- Link prediction used widely these days
- It is particularly interesting how existing solutions of graph databases are optimized for practical use cases *(like link prediction)*

Introduction

Link prediction overview [1]

- Used to predict what edges are most likely to appear in a given graph

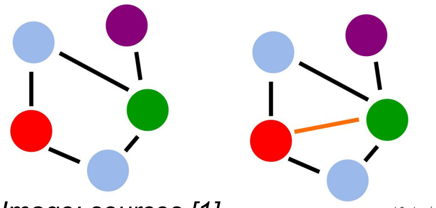
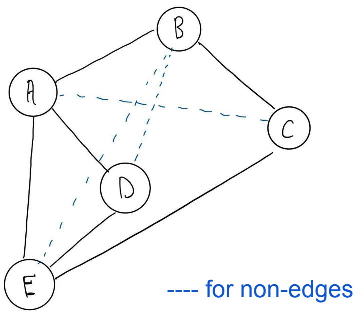


Image: sources [1]

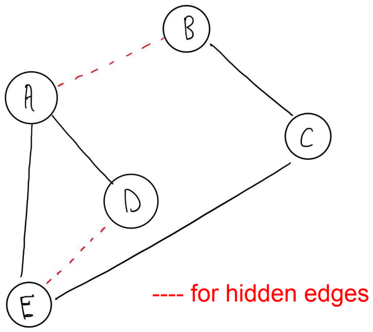
- Process usually consists of following steps:
  - Given a graph
  - Split data into a training set and a test set
  - Choose link prediction algorithm and use it on training set
  - Check accuracy compared to a test set
  - Compare with other link prediction algorithms

Link prediction overview [2] - Graph splitting

- Labels need to be defined - values of 1 or 0 (edge, non edge)
- Solution: randomly hide some of the edges from graph
- Hidden edges represent features with target variable of value 1
- Non-existent edges represent features with target variable of value 0



A-C	0
A-E	0
B-E	0
B-D	0



A-B	1
E-D	1

Link prediction overview [3] - Algorithms

The Jaccard coefficient of nodes X and Y is

$$\text{jacc\_coeff}(X,Y) = \frac{|N(X) \cap N(Y)|}{|N(X) \cup N(Y)|}$$

The Resource Allocation index of nodes X and Y is

$$\text{res\_alloc}(X,Y) = \sum_{u \in N(X) \cap N(Y)} \frac{1}{|N(u)|}$$

The Adamic-Adar index of nodes X and Y is

$$\text{adamic\_adar}(X,Y) = \sum_{u \in N(X) \cap N(Y)} \frac{1}{\log(|N(u)|)}$$

Preferential Attachment: the score *Sxy* depends on the degree of node *x* and *y* respectively

$$S_{xy} = k_x \cdot k_y$$

K-shortest-path prediction

$$S_k = \Sigma_{i=0}^{k-1} KSP(s,t,k)[i]$$

How to evaluate models/algorithms performance

- Precision

Precision = Total number of documents retrieved that are relevant/Total number of documents that are retrieved.

- Recall

Recall = Total number of documents retrieved that are relevant/Total number of relevant documents in the database

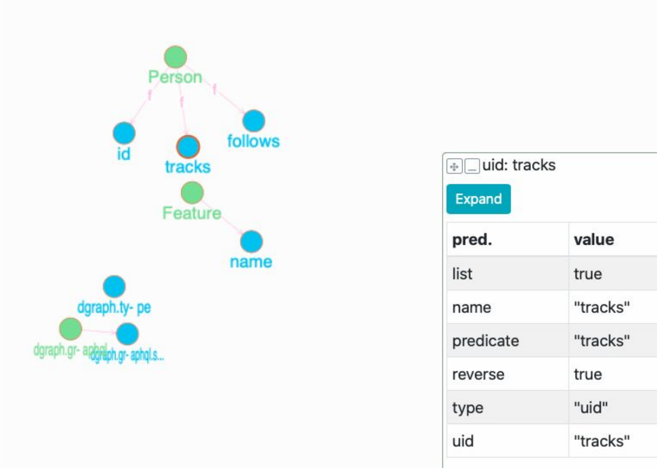
- AP

Average Precision is method to evaluate the precision and *ranking* of a predicted list of retrieved objects

$$AP(q) = \frac{\sum_{k=1}^{n} [P(k) \times rel(k)]}{n}$$

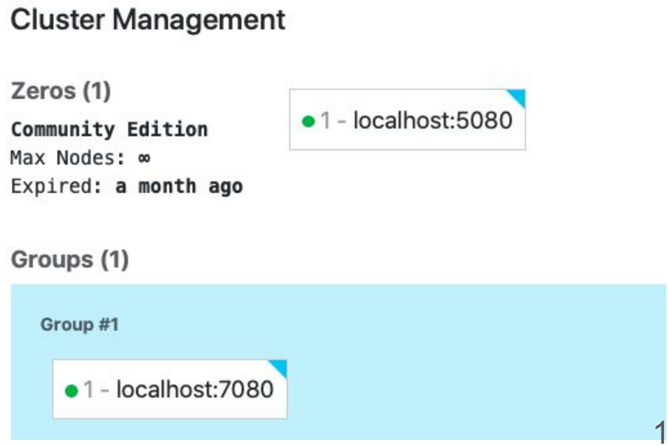
Dgraph setup

- Dgraph entity was setup locally using Docker
- Following schema initiated:



pred.	value
list	true
name	"tracks"
predicate	"tracks"
reverse	true
type	"uid"
uid	"tracks"

- Primitive dgraph setup used: single zero, single group with single alpha.



Dataset [1]

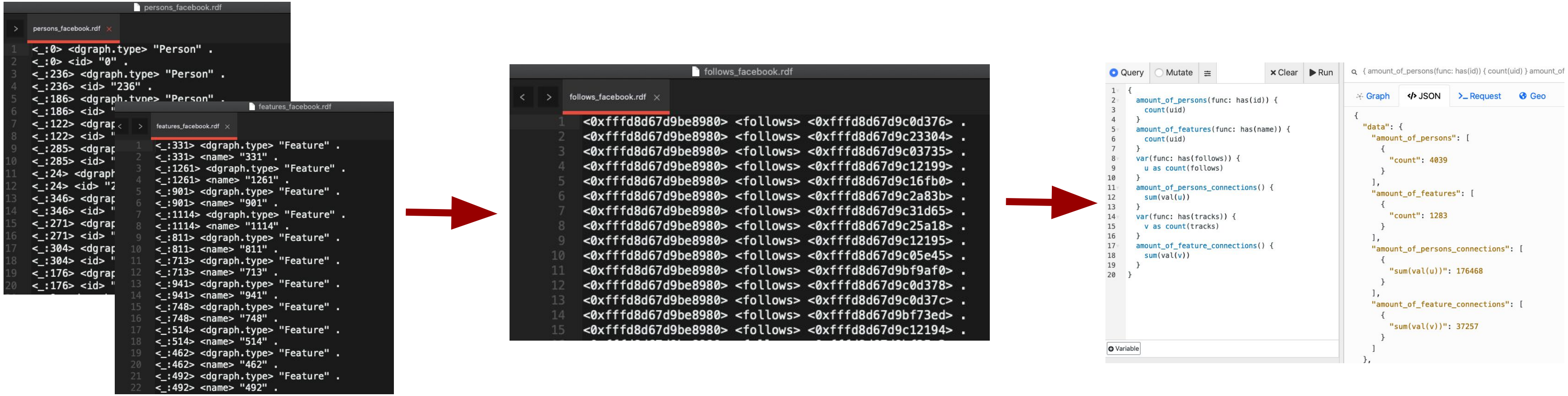
https://snap.stanford.edu/data/ego-Facebook.html

- Nodes represent persons, edges are undirected and represent relation (“friend”)
- Consists of 4 different file types for each file representing a single node’s perspective

Dataset statistics	
Nodes	4039
Edges	88234
Nodes in largest WCC	4039 (1.000)
Edges in largest WCC	88234 (1.000)
Nodes in largest SCC	4039 (1.000)
Edges in largest SCC	88234 (1.000)
Average clustering coefficient	0.6055
Number of triangles	1612010
Fraction of closed triangles	0.2647
Diameter (longest shortest path)	8
90-percentile effective diameter	4.7

Dataset [2]

- Data transformed into .rdf format using custom python script and written down into separate files
- Features were used as separate nodes, which allowed to increase amount of nodes and edges in graph
- Firstly nodes written to **features_facebook.rdf** and **persons_facebook.rdf** files using blank node ids
- This files were used by **live loader** to load data into dgraph
- Later loaded uids were used to add edges using **live loader**



Implementation

Data preparation & Details

- We will try to predict **[a]** possible new connections between persons
- & **[b]** new connections between persons and features
- Populate networkx Graph from Dgraph
- Average time required (*mean of 100 calculations on MacBook Air 13” mid 2012*): **[a]** 5.71 sec. **[b]** 5.21 sec.

```
def getAllPersons(self) -> str:
    query = """
            query {
                persons(func: has(id)) {
                    id
                    uid
                    tracks {
```

```
# prepare data
G = nx.Graph() # normal graph
G_persons = nx.Graph() # graph only with persons
for row in tqdm(features):
    feature = row['name']
    G.add_node(feature)
for row in tqdm(persons):
    person = row['uid']
    G.add_node(person)
    G_persons.add_node(person)
for row in tqdm(persons):
    person = row['uid']
    if 'tracks' in row:
        tracks = row['tracks']
        for feat in tracks:
            G.add_edge(person, feat['name'])
            # edges can be provided with additional attributes
            G[person][feat['name']]['type'] = 'tracks'
    if 'follows' in row:
        follows = row['follows']
        for pers in follows:
            G.add_edge(person, pers['uid'])
            G_persons.add_edge(person, pers['uid'])
            # edges can be provided with additional attributes
            G[person][pers['uid']]['type'] = 'follows'
            G_persons[person][pers['uid']]['type'] = 'follows'
```

Calculate 0's - negative samples - graph complement

- Non-existent edges can be taken from complement graph
- Take complement of graph containing only
  - [a] persons and their connections
  - [b] persons/features connections
- Networkx provides functionality to calculate complement graph:

complement

`complement` (G, name=None) [source]

Return the graph complement of G.

Parameters:
- G (*graph*) – A NetworkX graph
- name (*string*) – Specify name for new graph

Returns:
- GC (*A new graph.*)

- Results:
  - **[a] 8.066.507** non-existent person's connections edges
  - **[b] 5.144.780** non-existent connections between persons and features

Calculate 1’s - positive samples - split graph

- Not to disconnect the graph
- Amount of nodes should remain the same
- Results:
    - [a] **84.196** existent removable person’s connections edges
    - [b] **35.974** existent removable connections

```python
''' REMOVE LINKS FROM CONNECTED NODE PAIRS TO CREATE TRAINING SET BASIS '''
print("Working on removable edges in graph...")
omissible_links = [] # contains removable edges
if to_calculate_removable:
    G_temp = G.copy()
    for e in tqdm(G.edges):
        src = e[0]
        dst = e[1]
        if not filter_removable(src, dst, prediction_mode):
            continue
        # remove nodes pair
        G_temp.remove_edge(src, dst)
        # check if there is no splitting of graph and number of nodes is same
        if nx.number_connected_components(G_temp) == 1 and len(G_temp.nodes) == initial_node_count:
            # prepare as a line
            omissible_links.append(src + " " + dst + "\n")
        else:
            G_temp.add_edge(src, dst)

    # save removable links to some file
    with open(removable_links_file, "a") as f:
        f.writelines(omissible_links)
else:
    # read removables from file
    with open(removable_links_file, 'r') as f:
        for line in tqdm(f):
            edge = line.strip().split(" ")
            src, dst = edge[0], edge[1]
            if filter_removable(src, dst, prediction_mode):
                omissible_links.append((src, dst))
print("Removable edges calculated...")
G_train = G.copy()
```

Networkx provided algorithms

- Networkx provides functionality to compute many popular metrics
- We selected 4 of them: Jaccard Coefficients, Adamic Adar Index, Resource Allocation Index, Preferential Attachment
- **Plus**: k-shortest-paths-prediction custom implementation
- GraphQL+- has own k-shortest-paths query

```
def k_shortest_prediction(G, src, dst, k):
    shortests = nx.shortest_simple_paths(G,
src, dst)
    res = 0
    for c, path in enumerate(shortests):
        res += 1/sqrt(len(path))
        if c == k-1:
            break
    return res
```

Prediction set

[a] **8.066.507** non-existent person's connections edges

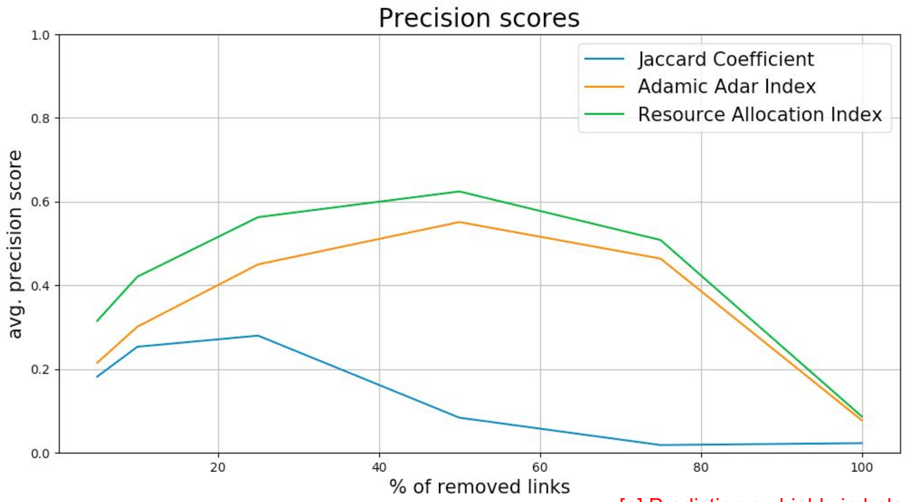
[b] **5.144.780** non-existent connections between persons and features

[a] **84.196** existent removable person's connections edges

[b] **35.974** existent removable connections

[a] The ratio link/no link is ~1%

[b] The ration link/no link is < 1%



[a] Prediction on highly imbalanced data

Prediction set

[a] **8.066.507** non-existent person's connections edges

[b] **5.144.780** non-existent connections between persons and features

[a] **84.196** existent removable person's connections edges

[b] **35.974** existent removable connections

[a] The ratio link/no link is ~1%

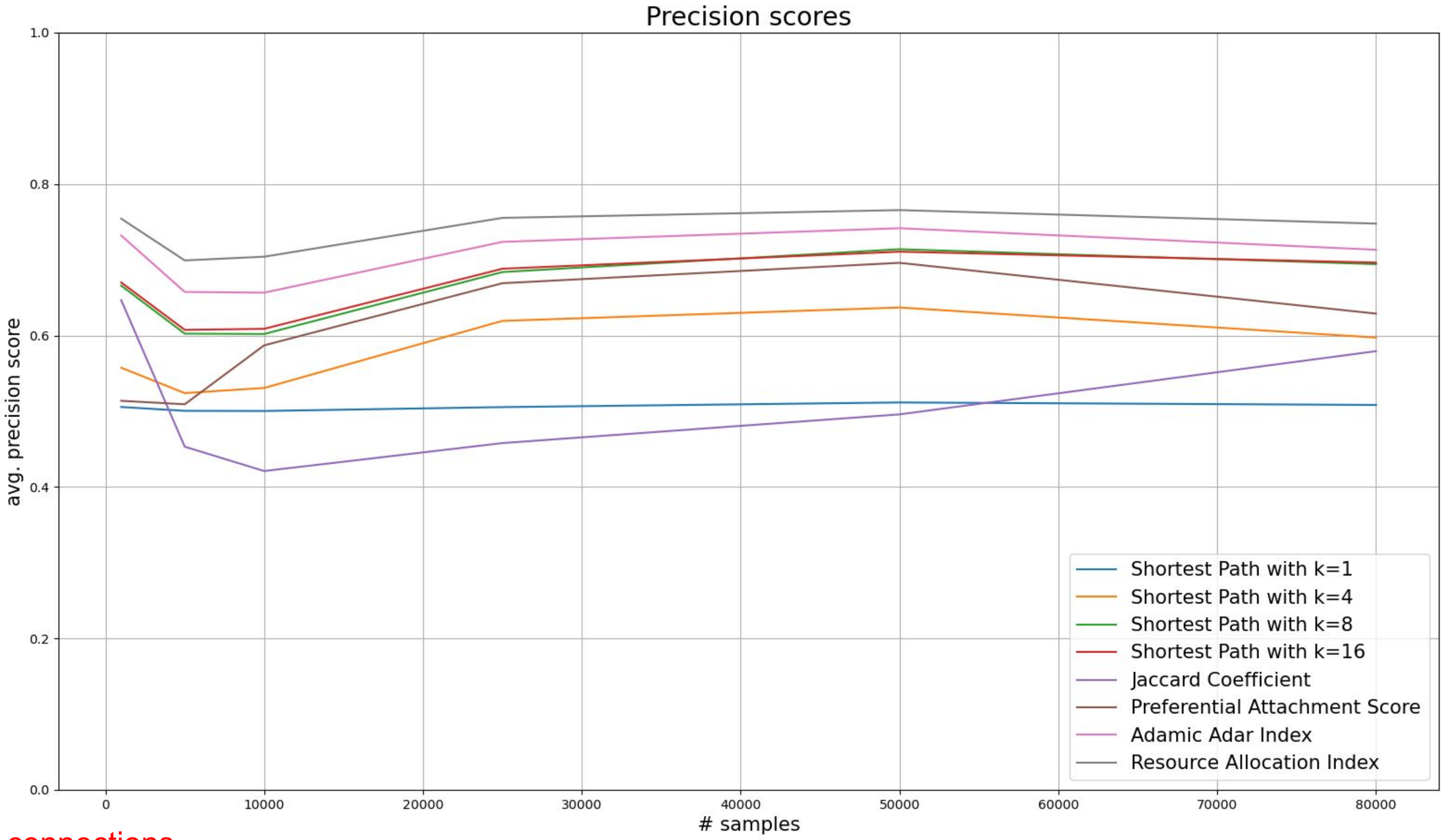
[b] The ration link/no link is < 1%

- Prediction performed on different numbers of samples between 1000 and **80.000** for [a] and **35.000** for [b]
- For each sample corresponding edges were removed from the graph

```
interface = DgraphInterface()

''' SETTINGS '''
numbers = [1000, 5000, 10000, 25000, 50000, 80000] # how many nodes to take for both labels (balanced sets)
predict_persons = True

interface = DgraphInterface()
G, _ = download_graph(predict_persons, interface)

for number in numbers:
```

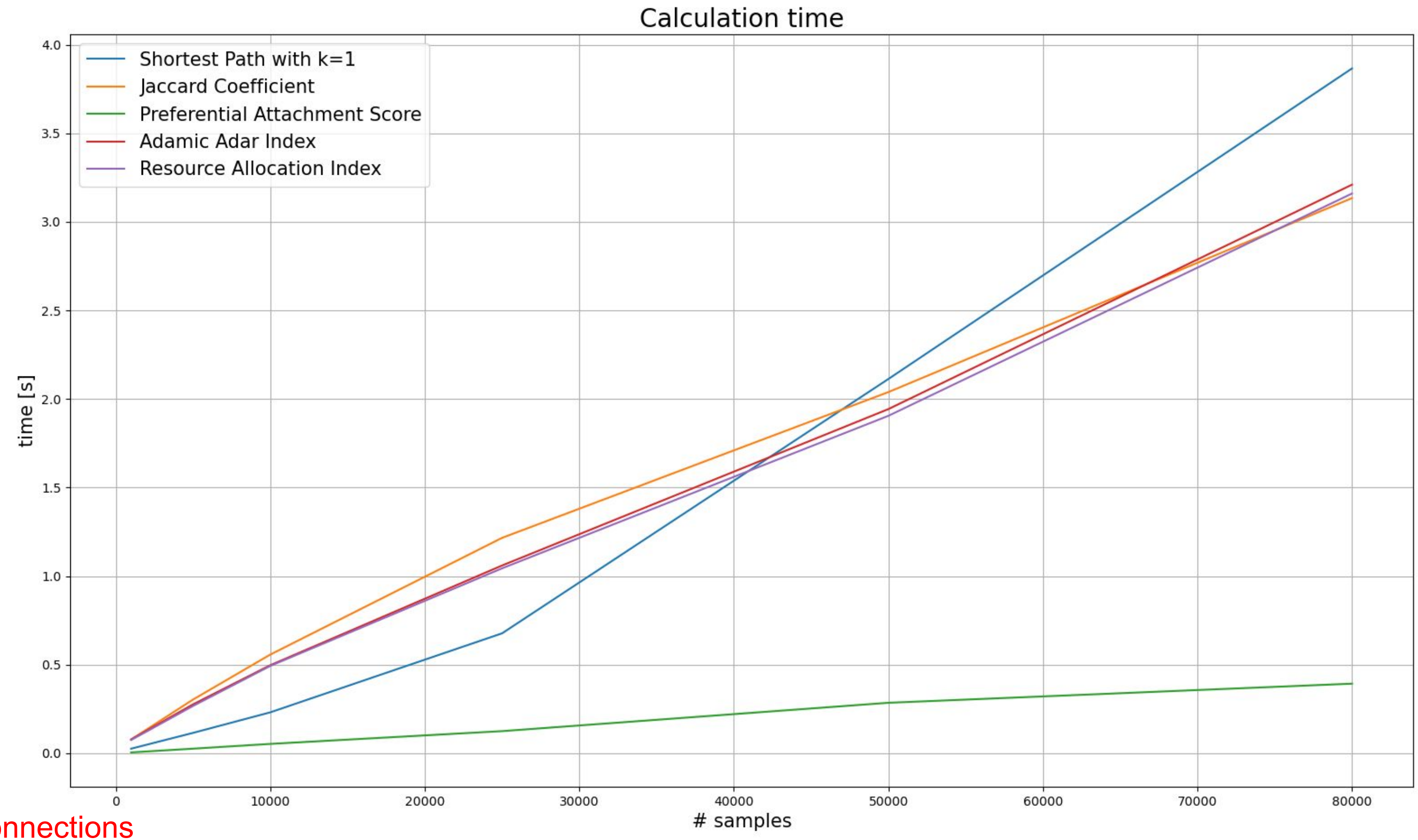
Results

[a] Performance measures



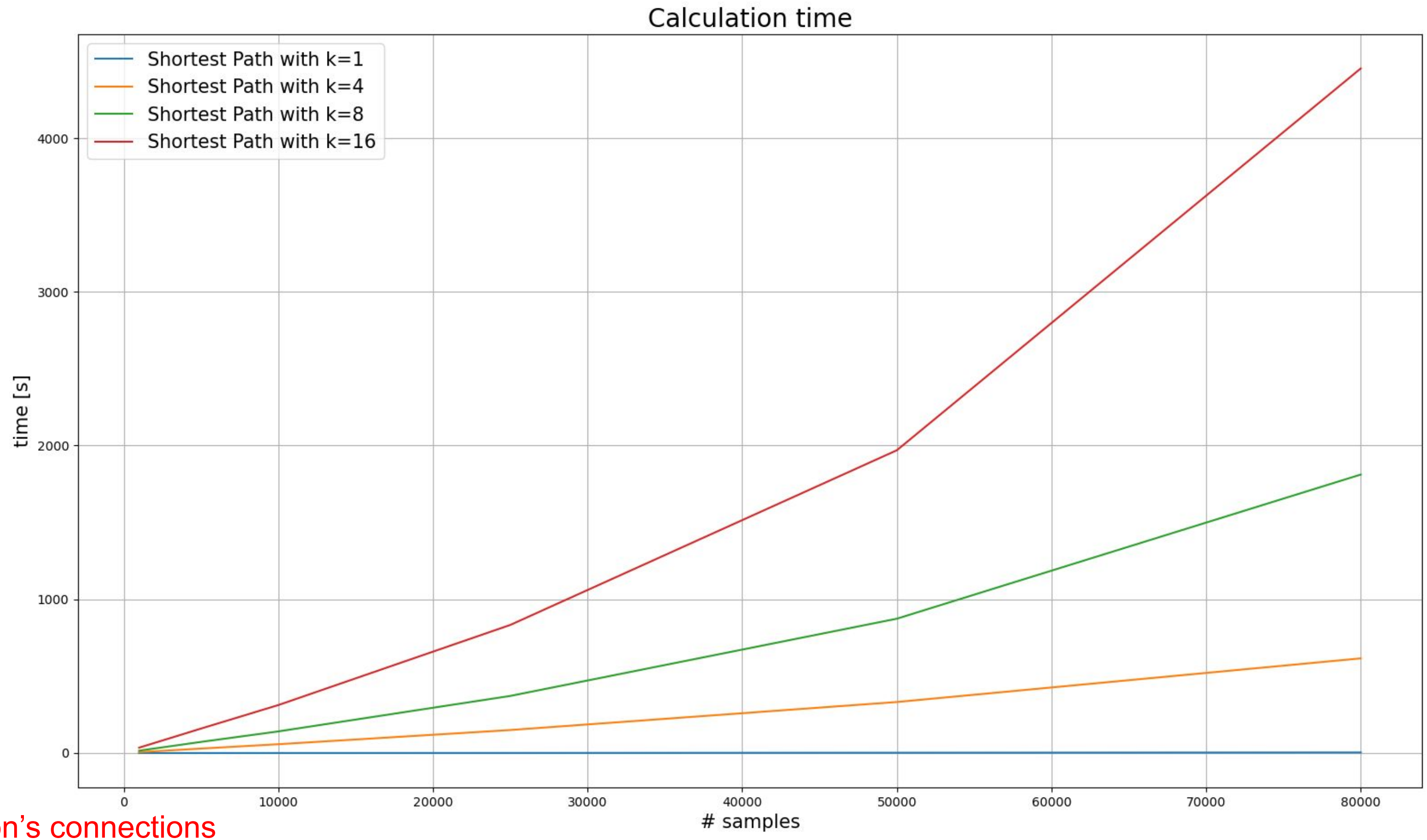
Person’s connections

[a] Calculation times measures



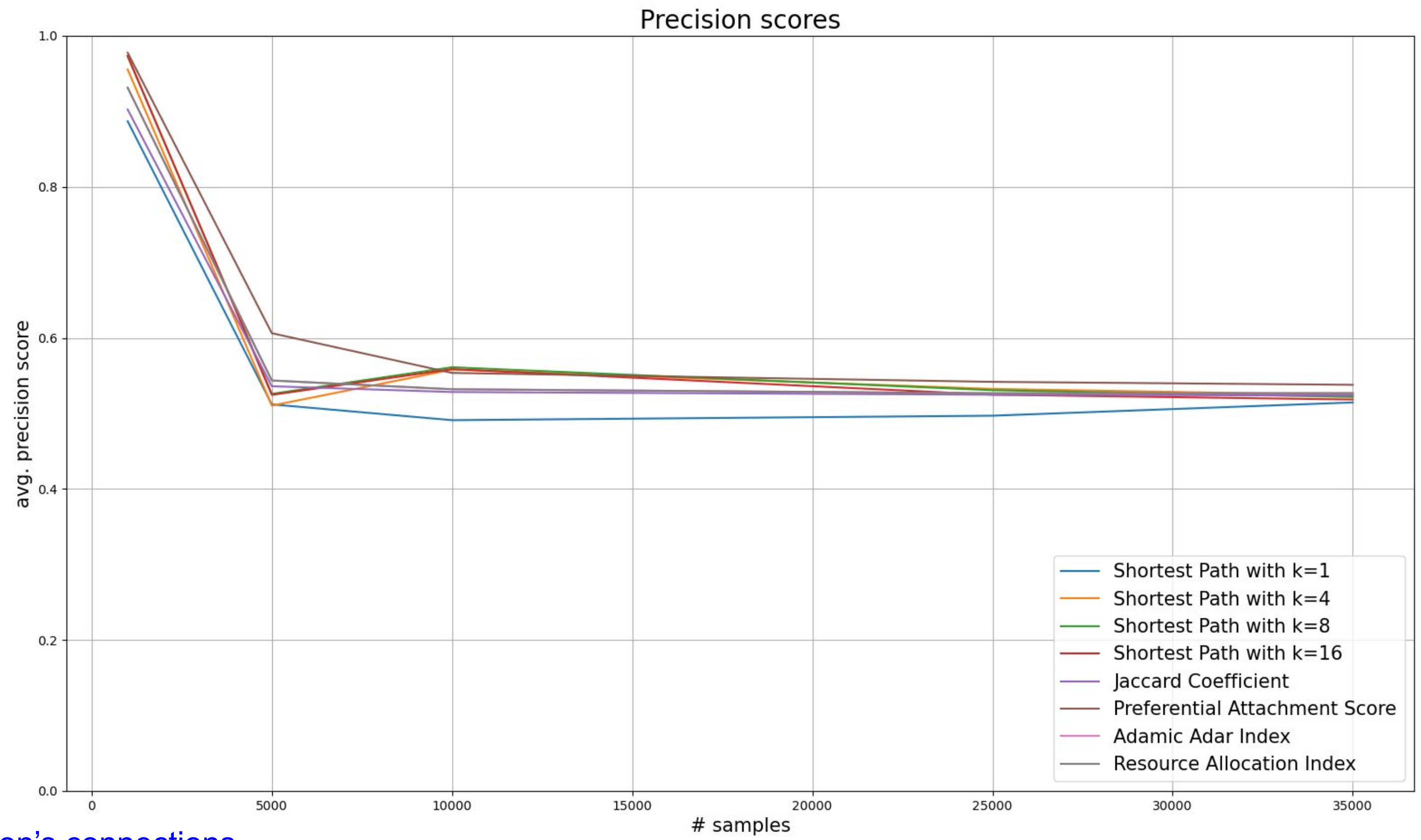
Person’s connections

[a] Calculation times measures between k-shortests



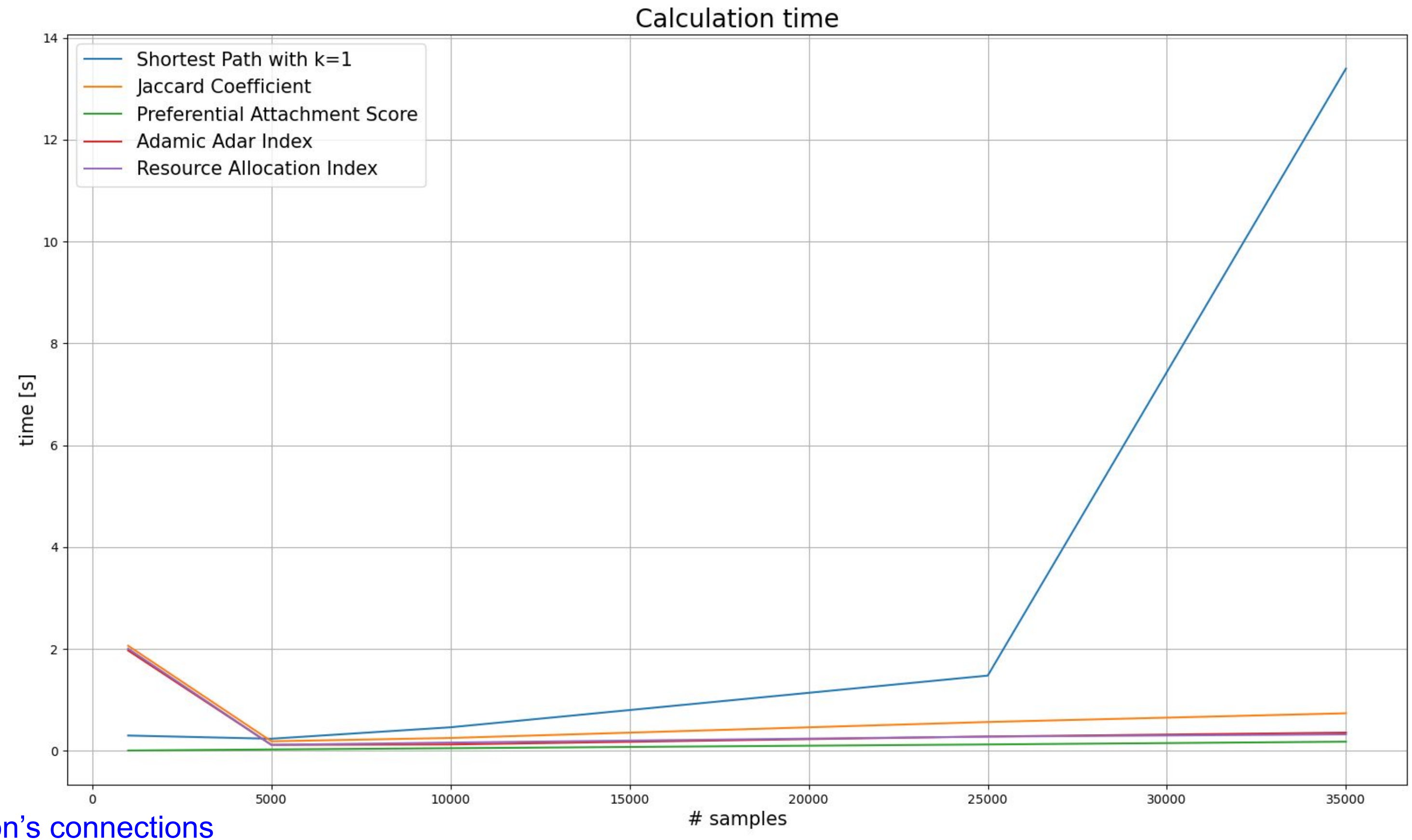
Person’s connections

[b] Performance measures



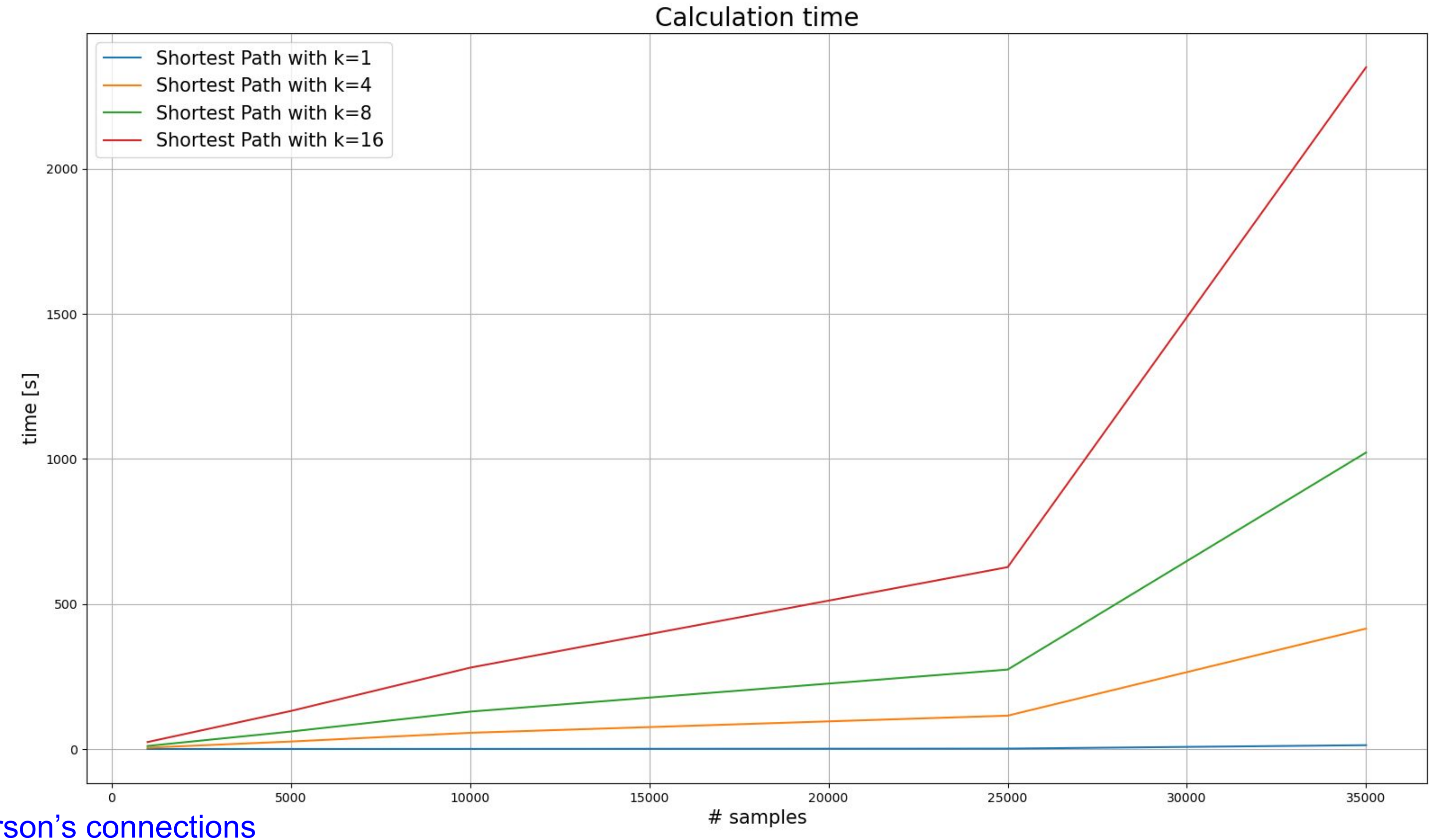
Person’s connections

[b] Calculation times measures



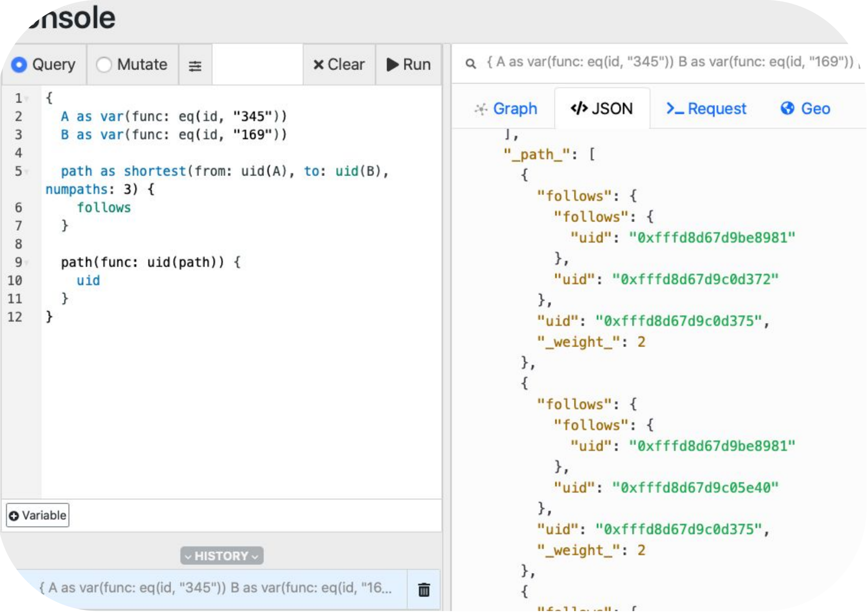
Person’s connections

[b] Calculation times measures between k-shortests



Person's connections

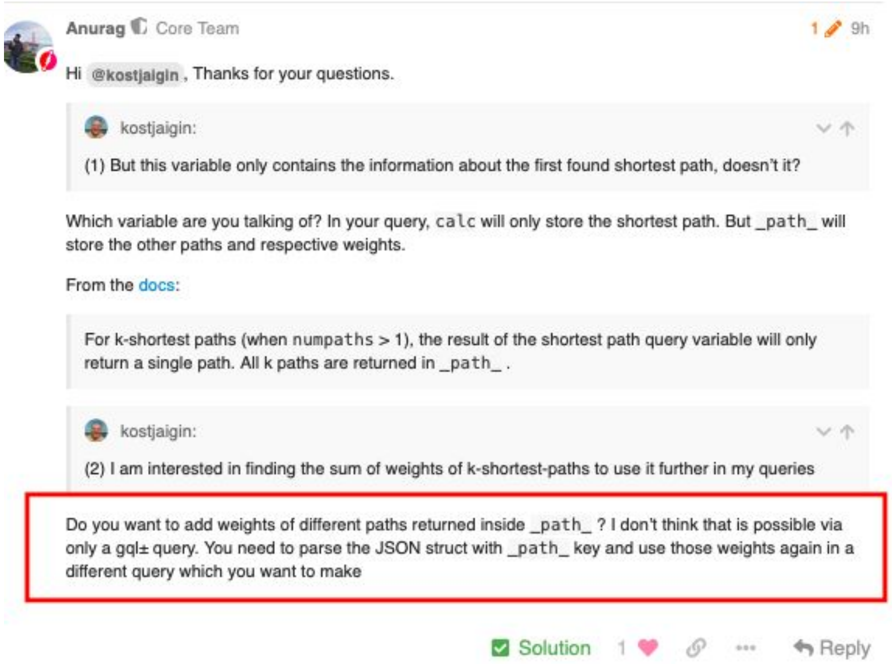
GraphQL+- Implementation k-shortest-path queries




```python
'''
    Find k shortest pathes in dgraph between @arg str and @arg dst
'''
def get_k_shortest_pathes(self, k: int, src: str, dst: str):
    query = """query all($src: string, $dst: string, $k: int) {
                source as var(func: uid($src))
                destination as var(func: uid($dst))
                shortest(from: uid(source), to: uid(destination), numpaths: $k) {
                    predictable
                    tracks
                    ~tracks
                }
            }
        """
    variables = {'$src': src, '$dst': dst, '$k': str(k)}
    res = self.client.txn(read_only=True).query(query, variables=variables)
    ppl = json.loads(res.json)
    latency = res.latency
    return ppl, latency
```

GraphQL+- Implementation k-shortest-path queries

- **Problem**: no way to sum weights using queries (GraphQL+-)



Anurag Core Team 1 9h

Hi @kostjaigin , Thanks for your questions.

> kostjaigin:
>
> (1) But this variable only contains the information about the first found shortest path, doesn't it?

Which variable are you talking of? In your query, `calc` will only store the shortest path. But `_path_` will store the other paths and respective weights.

From the docs:

> For k-shortest paths (when `numpaths` > 1), the result of the shortest path query variable will only return a single path. All k paths are returned in `_path_` .

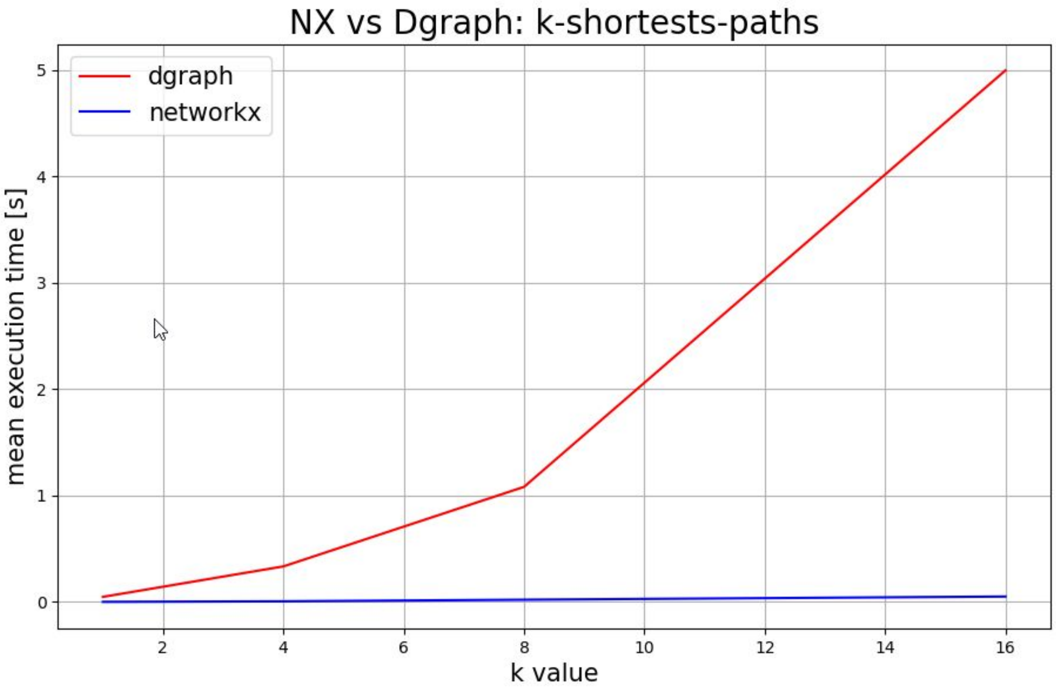
> kostjaigin:
>
> (2) I am interested in finding the sum of weights of k-shortest-paths to use it further in my queries

Do you want to add weights of different paths returned inside `_path_` ? I don't think that is possible via only a gql± query. You need to parse the JSON struct with `_path_` key and use those weights again in a different query which you want to make

Solution 1 Reply

- "Queries run concurrently, achieving low-latency and better throughput"
- Calculating weights for each requested edge manually requires a lot of resources

K-Shortests-Paths on Dgraph



- **Remark**: the simplest form of Dgraph deployment used
- Time of dgraph-networkx transformation not taken in count
- **But**: only processing delay taken in count for dgraph calculation

Outlook

- https://github.com/kostjaigin/OCITS_Dgraph implementations
- Algorithms/Scores we applied are usually used as features for ML models
- Simplest form of Dgraph deployment
- Basic knowledge in the field of link prediction gained
- GraphQL+- implementations of considered algorithms

Sources

1. A. Hogan, E. Blomqvist, M. Cochez, C. D'Amato, G. De Melo, C. Gutierrez, J. E. Labra Gayo, S. Kirrane, S. Neuemaier, A. Polleres, R. Navigli, A. Ngonga Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab and A.Zimmermann, "Knowledge Graphs", 2020.
2. A.Lebedev, J. Lee, V. Rivera, and M. Mazzara, "Link Prediction using Top-k Shortest Distances", Innnopolis University, Russia, 2017.
3. J. McAuley and J. Leskovec, "Learning to Discover Social Circles in Ego Networks", NIPS, 2012.
4. K.M. Ting, "Precision and Recall", in *Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning*. Springer, Boston, MA., 2010.
5. L. Sanders, O. Woolley, I. Moize and N. Antulov-Fantulin, "Introduction to Link Prediction. Machine Learning and Modelling for Social Networks", ETH Zürich, 2020.
6. Neo4j.Inc, "Overcoming SQL Strain and SQL Pain", *Go.neo4j.com*, 2018. [Online]. Available: https://go.neo4j.com/rs/710-RRC-335/images/WP-OvercomingSQLstrain.pdf?_ga=2.167365822.1441496647.1596201298-1505888731.1596201298. [Accessed: 31- Jul- 2020].
7. T. Akiba, T. Hayashi, N. Nori, Y. Iwata and Y. Yoshida ", "Efficient Top-k Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling" in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Austin, Texas, 2015.