fornax Documentation

Release 0.1.1

Daniel Staff

Apr 10, 2021

Contents

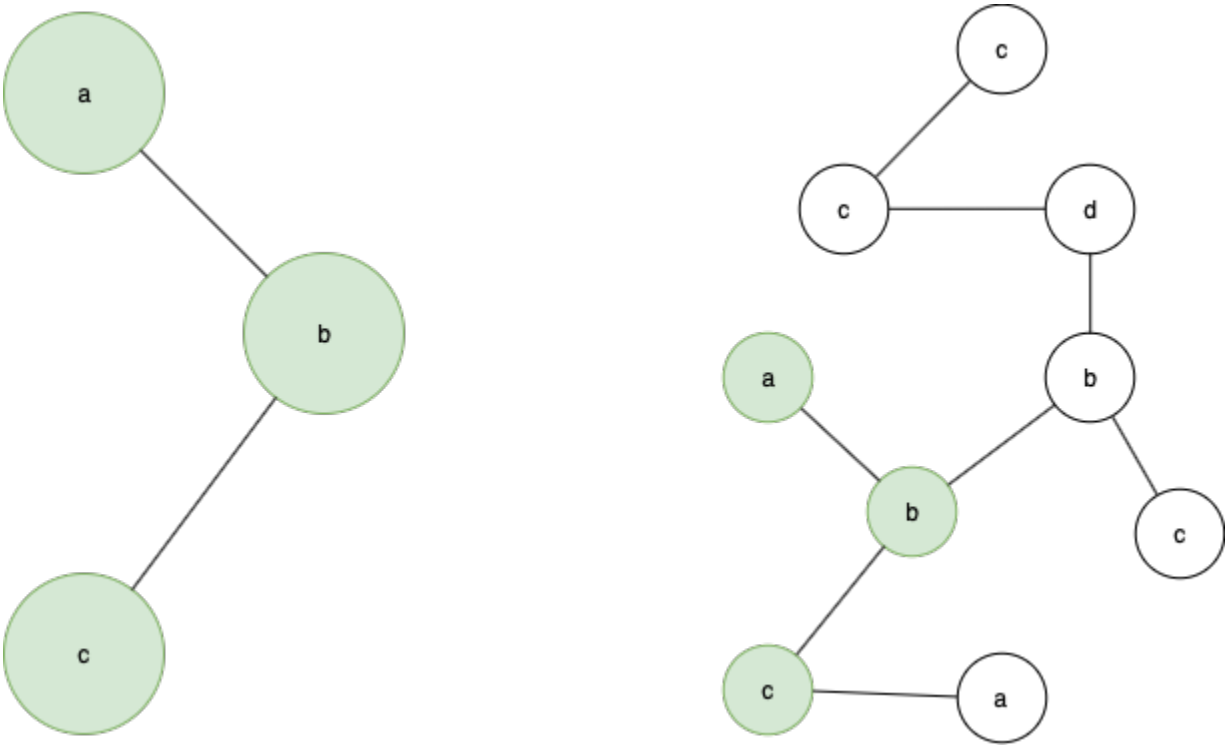
CHAPTER 1

Guide

Fornax is an open source library to perform fuzzy subgraph matching between labelled undirected graphs based on NeMa: Fast Graph Search with Label Similarity.

1.1 Subgraph Matching

A subgraph is any collection of node and edges that form some subset of a graph. For example in the image below the graph on the left is isomorphic to the green nodes in the graph on the right, hence they form a subgraph.



If we refer to the graph on the left as the *query graph* and the graph on the right as the *target graph* subgraph matching is the process of finding the *query graph* in the *target graph* such that the node labels and edges are strictly the same.

Fornax will kind the *n* most similar subgraphs in a *target graph* based on a user specified *query graph* using a user specified *label similarity function*.

Fornax will not only find exact subgraph isomorphisms but the *n* most similar subgraphs even if they are not exact isomorphisms of the query graph. Hence, fornax can be used for **fuzzy** subgraph matching.

For example, Fornax can be used to find subgraphs where labels are similar, yet different, based on a user specified definition. Additionally neighbours in the query graph may be absent, or are neighbours of neighbours in the target graph.

1.2 Example Problems

Common fuzzy subgraph matching problems include:

- searching knowledge graphs
- mining social networks
- searching geospation data as a graph
- searching text as a graph

1.3 Goals

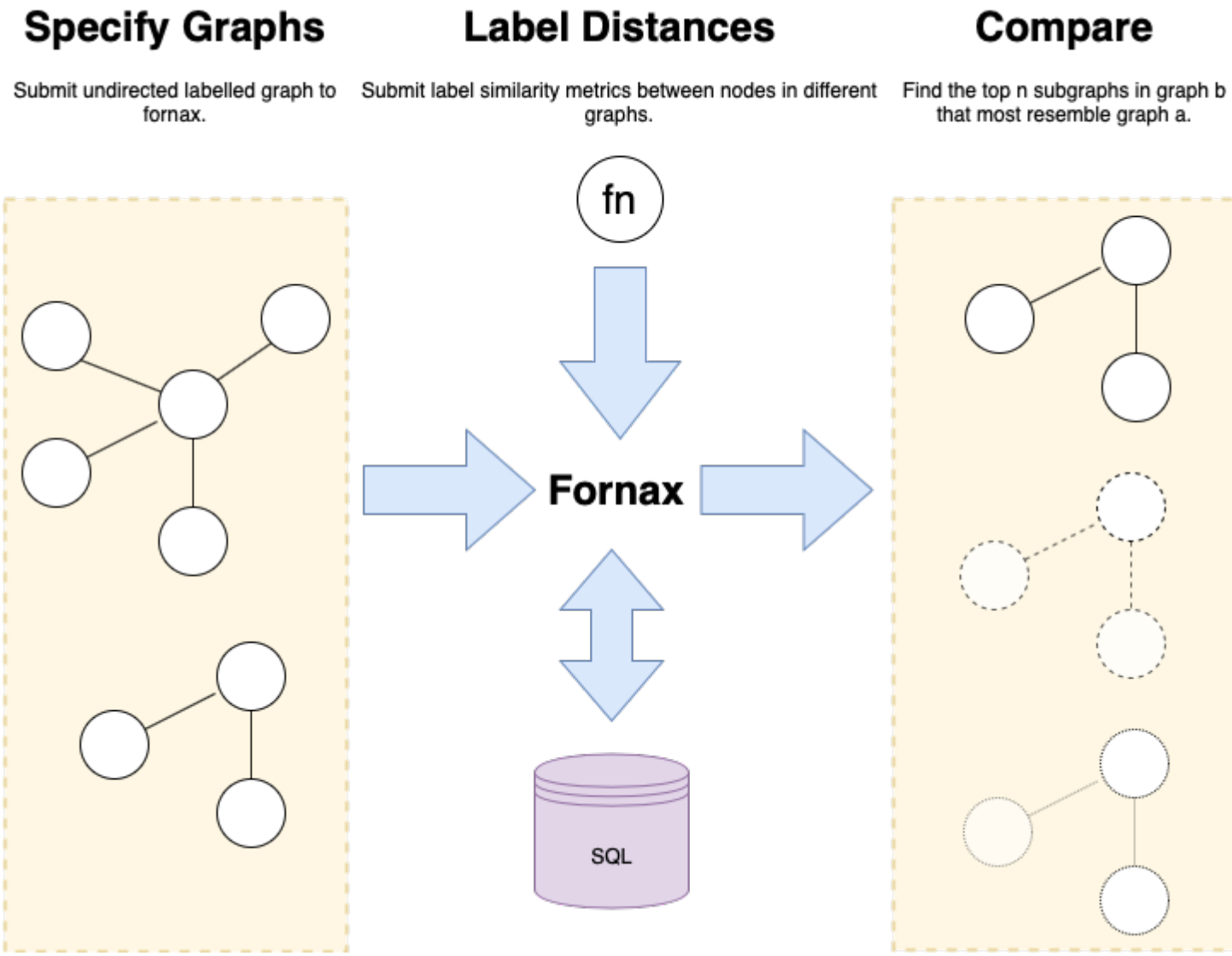
fornax was written with three primary goals in mind

- to demonstrate process and provide ease of use over performance
- to be flexible and accomidate the users notions of similarity
- to scale to large target graphs of millions of nodes and edges

1.4 Architecture

In order to support large graphs and persist them between python interpreter sessions fornax stores all data in a database.

To facilite ease of use fornax can use *sqlite* or *postgresql* as a back end. For more details see the API *Introduction*.


Specify Graphs
Submit undirected labelled graph to fornax.
Label Distances
Submit label similarity metrics between nodes in different graphs.
Compare
Find the top n subgraphs in graph b that most resemble graph a.
fn
Fornax
SQL

CHAPTER 2

Creating a Dataset

[1]:
```
from SPARQLWrapper import SPARQLWrapper, JSON
import pandas as pd
import hashlib
```

To install the use the dependencies for this notebook:

```
conda env create -f environment.yml
source activate fornax_tutorial
```

To run this notebook from the project root:

```
cd docs/tutorial
jupyter-notebook
```

2.1 Download

For the duration of this tutorial we will be using the social network of Marvel Comicbook characters.

Nodes will represent

- characters
- aliases
- groups of characters

edges will represent relationships between the nodes.

For example Wolverine, Logan and X-Men are all nodes. There is an edge between Wolverine and Logan because Logan is an alternative name for Wolverine. There is an edge between Wolverine and X-Men because Wolverine is a member of X-Men. There is no direct relationship between Logan and X-Men so there is no edge between them.

2.1.1 SPARQL

Below is a SPARQL query which will return data in the following format (using 'Wolverine' as an example):


```
{
    "group": {
        "type": "uri",
        "value": "http://www.wikidata.org/entity/Q2690825"
    },
    "character": {
        "type": "uri",
        "value": "http://www.wikidata.org/entity/Q186422"
    },
    "birthName": {
        "xml:lang": "en",
        "type": "literal",
        "value": "James Howlett"
    },
    "characterLabel": {
        "xml:lang": "en",
        "type": "literal",
        "value": "Wolverine"
    },
    "groupLabel": {
        "xml:lang": "en",
        "type": "literal",
        "value": "Horsemen of Apocalypse"
    },
    "characterAltLabel": {
        "xml:lang": "en",
        "type": "literal",
        "value": "Logan, Weapon X, Jim Logan, Patch, James Howlett, Agent Ten,␣
↪Experiment X, Weapon Ten"
    }
}
```


[2]:
```
sparql = SPARQLWrapper("https://query.wikidata.org/sparql")
sparql.setQuery("""
    SELECT ?character ?characterLabel ?group ?groupLabel ?birthName ?characterAltLabel
    WHERE {
        ?group wdt:P31 wd:Q14514600 ;  # group of fictional characters
               wdt:P1080 wd:Q931597.    # from Marvel universe
        ?character wdt:P463 ?group.    # member of group
        optional{ ?character wdt:P1477 ?birthName. }
      SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en".
↪}
    }
""")
sparql.setReturnFormat(JSON)
results = sparql.query().convert()
```

2.2 Loading with Pandas

We'll be using pandas to do some data manipulation so lets put the result inside a pandas dataframe.

[3]:
```
# load the results into a pandas DataFrame
records = []
for result in results["results"]["bindings"]:
    character_id = result['character']['value']
    group_id = result['group']['value']
    name = result['characterLabel']['value']
    group = result['groupLabel']['value']
    alt_names = None
    if 'characterAltLabel' in result:
        alt_names = result['characterAltLabel']['value']
    birth_name = None
    if 'birthName' in result:
        birth_name = result['birthName']['value']
    records.append((character_id, group_id, name, group, birth_name, alt_names))

frame = pd.DataFrame.from_records(records, columns=['character_id', 'group_id', 'name
↪', 'group', 'birth_name', 'alt_names'])
```

[4]:
```
frame.head()
```

[4]:

	character_id	group_id	name	group	birth_name	alt_names
0	http://www.wikidata.org/entity/Q60002	http://www.wikidata.org/entity/Q2603976	Colossus	Excalibur		Peter Rasputin, Piotr Nikolayevich Rasputin, P...
1	http://www.wikidata.org/entity/Q258015	http://www.wikidata.org/entity/Q2603976	Rachel Summers	Excalibur	Rachel Anne Summers	Phoenix, Prestige, Marvel Girl, Mother Askani,...
2	http://www.wikidata.org/entity/Q369197	http://www.wikidata.org/entity/Q2527918	Black Widow	Thunderbolts		Natalie Rushman, Natasha Romanoff, asesina rus...
3	http://www.wikidata.org/entity/Q388316	http://www.wikidata.org/entity/Q2527918	Bullseye	Thunderbolts	None	Lester, Hawkeye, Benjamin Poindexter
4	http://www.wikidata.org/entity/Q432272	http://www.wikidata.org/entity/Q2457162	Medusa	Frightful Four	None	None

2.2.1 Splitting into Tables

The dataframe above is unwieldy since it contains a list of values in the alt_names column. Most values also appear in many rows. Below we seperate the frame into a set of tables with unique rows much like we would for a relational database.

[5]:
```
names = frame[['character_id', 'name']].drop_duplicates()
groups = frame[['group_id', 'group']].drop_duplicates()
```

(continues on next page)

(continued from previous page)

```
character_group = frame[['character_id', 'group_id']].drop_duplicates()
birth_names = frame[
    frame['birth_name'].notna() # do not include a row for characters without a␣
↪birthname
][['character_id', 'birth_name']].drop_duplicates()
```

[6]:
```
records = []
for uid, alt_names in zip(frame['character_id'], frame['alt_names']):
    if alt_names is None:
        continue
    for name in alt_names.split(','):
        records.append({'character_id': uid, 'alt_name': name})
alt_names = pd.DataFrame.from_records(records).drop_duplicates()
```

2.3 Analysis

Lets quickly examine the data to check it still makes sense.

There are 399 unique characters in the dataset

[7]:
```
# count the number of unique character ids
len(names)
```

[7]:
```
400
```

The characters belong to 107 different groups

[8]:
```
# count the number of unique group ids
len(groups)
```

[8]:
```
107
```

Let's find Wolverine…

[9]:
```
names[names['name'] == 'Wolverine']
```

[9]:
```
Empty DataFrame
Columns: [character_id, name]
Index: []
```

What groups has he been in?

[10]:
```
character_group[
    character_group['character_id'] == 'http://www.wikidata.org/entity/Q186422'
].merge(groups, on='group_id')['group']
```

[10]:
```
0                     X-Men
1             Alpha Flight
2                 Avengers
3    Horsemen of Apocalypse
4                    Team X
Name: group, dtype: object
```

What are his alternative names?

[11]:
```
alt_names[alt_names['character_id'] == 'http://www.wikidata.org/entity/Q186422']
```

[11]:
```
           alt_name                              character_id
187           Logan  http://www.wikidata.org/entity/Q186422
188        Weapon X  http://www.wikidata.org/entity/Q186422
189       Jim Logan  http://www.wikidata.org/entity/Q186422
190           Patch  http://www.wikidata.org/entity/Q186422
191   James Howlett  http://www.wikidata.org/entity/Q186422
192       Agent Ten  http://www.wikidata.org/entity/Q186422
193    Experiment X  http://www.wikidata.org/entity/Q186422
194      Weapon Ten  http://www.wikidata.org/entity/Q186422
```

What is his birthname?

[12]:
```
birth_names[birth_names['character_id'] == 'http://www.wikidata.org/entity/Q186422']
```

[12]:
```
                              character_id     birth_name
103  http://www.wikidata.org/entity/Q186422  James Howlett
```

What is the biggest team?

[13]:
```
sorted(
    [
        (len(group), group[['group_id', 'group']].drop_duplicates())
        for uid, group
        in character_group.merge(groups, on='group_id').groupby('group_id')
    ],
    key=lambda x: x[0],
    reverse=True
)[0]
```

[13]:
```
(116,                                    group_id  group
 125  http://www.wikidata.org/entity/Q128452  X-Men)
```

Who has been in the most groups?

[14]:
```
sorted(
    [
        (len(group), group['name'].drop_duplicates())
        for uid, group
        in character_group.merge(names, on='character_id').groupby('character_id')
    ],
    key=lambda x: x[0],
    reverse=True
)[0]
```

[14]:
```
(8, 102    Cannonball
 Name: name, dtype: object)
```

2.4 Export to CSV

Let's write each node to a csv file, we need to record

- a unique ID for each node (we use a hash of the Wikidata URL since later we will need an integer ID)
- a label (such as the name of the character or the group)
- a type (0, 1, 2 for character, group or birthname)

Node that birth names don't have a Wikidata URL so we just use a hash of the name.

[15]:
```
def get_id(url):
    """ A function to map the python hash function onto 32-bit integers"""
    return int(hashlib.sha256(url.encode('utf-8')).hexdigest(), 16) % 2147483647
```

[16]:
```
nodes = pd.concat(
    [
        pd.DataFrame({'uid': [get_id(item) for item in names['character_id']], 'type':
↪ 0, 'label': names['name']}),
        pd.DataFrame({'uid': [get_id(item) for item in groups['group_id']], 'type': 1,
↪ 'label': groups['group']}),
        pd.DataFrame({'uid': [get_id(item) for item in birth_names['birth_name']],
↪'type': 2, 'label': birth_names['birth_name']}),
        pd.DataFrame({'uid': [get_id(item) for item in alt_names['alt_name']], 'type':
↪ 2, 'label': alt_names['alt_name']})
    ],
    sort=True
).drop_duplicates()
nodes.to_csv('./nodes.csv', index=False)
```

[17]:
```
nodes.head()
```

[17]:

	label	type	uid
0	Colossus	0	2105314676
1	Rachel Summers	0	298635603
2	Black Widow	0	1897346471
3	Bullseye	0	2027281781
4	Medusa	0	347320780

Edges connect characters to their birth names and their groups.

[18]:
```
# character_group_edges = frame[['character_id', 'group_id']].drop_duplicates()
# character_birth_name_edges = frame[['character_id', 'birth_name']].drop_duplicates()

edges = pd.concat([
    # character to group
    pd.DataFrame([
        {'start': get_id(start), 'end': get_id(end)}
        for start, end in zip(character_group['character_id'], character_group['group_
↪id'])
    ]),

    # character to alt name
    pd.DataFrame([
        {'start': get_id(start), 'end': get_id(end)}
        for start, end in zip(alt_names['character_id'], alt_names['alt_name'])
    ])
])
```

[19]:
```
edges.to_csv('./edges.csv', index=False)
```

CHAPTER 3

Tutorial

[1]:
```
import os
import sys
import pandas as pd
import json
import matplotlib.pyplot as plt
import networkx as nx
import fornax

%matplotlib inline
from IPython.core.display import SVG

# Add project root dir
ROOT_DIR = os.path.abspath("../../")
sys.path.append(ROOT_DIR)
```

To install the use the dependencies for this notebook:

```
conda env create -f environment.yml
source activate fornax_tutorial
```

To run this notebook from the project root:

```
cd docs/tutorial
jupyter-notebook
```

In this tutorial we will:

- Load a graph of superheros and their teams from csv files
- Search for nodes in the graph using a string similarity function
- Use fornax to search for nodes using string similarity and fuzzy graph matching

The data in this tutorial we be generated using the preceding notebook: `Tutorial1.ipynb`.

3.1 Introduction

`nodes.csv` and `edges.csv` contain a graph of superheros and their teams along with alternative names for those heros and groups (or aliases).

The image below uses the example of Iron Man, who is known as "Tony" to his friends. Iron man is a member of the Avengers, a.k.a. Earth's Mightiest Superheros. Other heros are also members of The Avengers, and they will also have aliases. Other heros will also be members of other teams and so and so forth.

All of these heros, teams and aliases together make our target graph, a graph which we will search using fornax.

[2]:
```
SVG('../img/iron_man.svg')
```

[2]:

Let's load the data into the notebook using pandas.

[3]:
```
# used for converting csv values in nodes.csv
mapping = {
    '0': 'hero',
    '1': 'team',
    '2': 'hero_alias',
    '3': 'team_alias'
}

nodes_df = pd.read_csv(
    './nodes.csv',
    # rename the columns as targets as this will form the target graph
    # (the graph which we will be searching)
    names=['target_label', 'target_type', 'target_id'],
    # ignore the header
    header=0,
    converters = {
        # convert target_type from numeric values to
        # literal string representations for ease of reading
        'target_type': lambda key: mapping.get(key)
    }
)

# contains pairs of target node ids
edges_df = pd.read_csv('./edges.csv')
```

We can see that the target nodes have a label (the hero's primary name). The target_type column will be one of `hero`, `team`, `hero alias`, `team alias`, the four types of nodes in the graph.

(Note that by hero we mean a person in a comic book who has superpowers regardless of them being good or bad)

[4]:
```
nodes_df['target_label'].head()
```

[4]:
```
0         Selene
1    Doctor Doom
2          Viper
3            Sin
4    David North
Name: target_label, dtype: object
```

Edges are pairs of `target_id` values. Note that fornax deals with undirected graphs so there is no need to add the edge in the reverse direction. Doing so will cause an exception as the edge will be considered a duplicate.

[5]:
```
edges_df.head()
```

[5]:
```
          end       start
0   839851079    87770955
1   685373387  2073821878
2  1988120854   396175249
3   608208951   396175249
4  1988120854  2062678112
```

3.2 Label similarity

For some motivation, before using fornax, let us search for nodes just using their labels. Let's search for nodes similar to `guardians`, `star` and `groot`.

We will create a function that given a pair of labels, it will return a score where:

$$0 <= score <= 1$$

Secondly we'll create a search function that returns rows from our table of target nodes that have a non zero similarity score.

[6]:
```
def node_scoring_function(first: str, second: str):
    """ node scoring function takes two strings and returns a
        score in the range 0 <= score <= 1
    """
    first_, second_ = sorted((first.lower(), second.lower()), key=len)
    # if first is not a substring of second: score = 0
    if not first_ in second_:
        return 0
    # otherwise use the relative difference between
    # the two lengths
    score = len(second_) - len(first_)
    score /= max(len(first_), len(second_))
    score = 1. - score
    return score
```

[7]:
```
def search(query_id: int, query_label: str):
    # compute all of the scores
    scores = nodes_df['target_label'].apply(
        node_scoring_function,
        args=(query_label,)
    )
    # create a boolean mask
    mask = scores > 0
    # graph the non zero scoring nodes
    matches = nodes_df[mask].copy()
    # add extra columns
    matches['score'] = scores[mask]
    matches['query_label'] = query_label
    matches['query_id'] = query_id
    return matches
```

3.2.1 Aside:

Note that these string search functions are not terribly efficient. They involve repeated full scans of the target nodes table. If we were searching a larger graph we could use a search tree as an index, an external sting matching service or database. However, since this is a tutorial, the above functions are simpler and more reproducible. This is important as we will be using these search results with fornax.

[8]:
```
query_labels = ['guardians', 'star', 'groot']
```

Examining the table below we can see that we have a conundrum. There are 22 nodes with varying similarity to `star` and 4 nodes similar to `galaxy`.

[9]:
```
# find the nodes similar to 'guardians', 'star' and 'groot'
matches = pd.concat(search(id_, label) for id_, label in enumerate(query_labels))
matches
```

[9]:

	target_label	target_type	target_id	score	query_label	query_id
285	Guardian	hero	1081675	0.888889	guardians	0
427	Guardians of the Galaxy	team	870807271	0.391304	guardians	0
507	Guardians of the Galaxy (1969 team)	team	1295400389	0.257143	guardians	0
1019	Guardian	hero_alias	2062791326	0.888889	guardians	0
10	Danielle Moonstar	hero	2083850919	0.235294	star	1
25	Darkstar	hero	1276753309	0.500000	star	1
71	Firestar	hero	274821742	0.500000	star	1
121	Star-Lord	hero	1061867605	0.444444	star	1
189	Northstar	hero	1260880284	0.444444	star	1
292	Starfox	hero	1594294259	0.571429	star	1
323	Ultimate Firestar	hero	1718026772	0.235294	star	1
338	Shatterstar	hero	1241925506	0.363636	star	1
401	Upstarts	team	839851079	0.500000	star	1
443	Starjammers	team	895117495	0.363636	star	1
474	Starforce	team	1605941117	0.444444	star	1
536	James Proudstar	hero_alias	268149375	0.266667	star	1
587	John Proudstar	hero_alias	880197081	0.285714	star	1
604	Anthony "Tony" Edward Carbonell Stark	hero_alias	2007806013	0.108108	star	1
661	Moonstar	hero_alias	294373473	0.444444	star	1
750	Star-Lord	hero_alias	92571479	0.400000	star	1
831	Starlord	hero_alias	1788314407	0.444444	star	1
832	Star Lord	hero_alias	925434646	0.400000	star	1
1010	Anthony Edward "Tony" Stark	hero_alias	2138996395	0.142857	star	1
1011	Tony Stark	hero_alias	182299133	0.363636	star	1
1014	The Star Spangled Man With A Plan	hero_alias	1915573563	0.117647	star	1
1069	Firestar	hero_alias	1580065367	0.444444	star	1
120	Groot	hero	74671434	1.000000	groot	2

(continues on next page)

(continued from previous page)

```
401          star          1
443          star          1
474          star          1
536          star          1
587          star          1
604          star          1
661          star          1
750          star          1
831          star          1
832          star          1
1010         star          1
1011         star          1
1014         star          1
1069         star          1
120         groot          2
```

Fornax enables a more powerful type of search. By specifying 'guardians', 'star', 'groot' as nodes in a graph, and by specifying the relationships between them, we can search for nodes in our target graph with the same relationships.

3.3 Creating a target graph

Fornax behaves much like a database. In fact it uses SQLite or Postgresql to store graph data and index it. To insert a new graph into fornax we can use the following three steps: 1. create a new graph 2. add nodes and node meta data 3. add edges and edge meta data

The object `fornax.GraphHandle` is much like a file handle. It does not represent the graph but it is an accessor to it. If the `GraphHandle` goes out of scope the graph will still persist until it is explicitly deleted, much like a file.

[10]:
```
with fornax.Connection('sqlite:///mydb.sqlite') as conn:
    target_graph = fornax.GraphHandle.create(conn)
    target_graph.add_nodes(
        # use id_src to set a custom id on each node
        id_src=nodes_df['target_id'],
        # use other keyword arguments to attach arbitrary metadata to each node
        label=nodes_df['target_label'],
        # the type keyword is reserved to we use target_type
        target_type=nodes_df['target_type']
        # meta data must be json serialisable
    )
    target_graph.add_edges(edges_df['start'], edges_df['end'])
```

We can use the `graph_id` to access our graph in the future.

[11]:
```
with fornax.Connection('sqlite:///mydb.sqlite') as conn:
    target_graph.graph_id
    another_target_graph_handle = fornax.GraphHandle.read(conn, target_graph.graph_id)
    print(another_target_graph_handle == target_graph)
```

```
True
```

3.4 Creating a query graph

Let's imagine that we suspect `groot` is directly related to `guardians` and `star` is also directly related to `guardians`. For example `groot` and `star` could both be members of a team called `guardians`. Let's create another small graph that represents this situation:

[12]:
```
with fornax.Connection('sqlite:///mydb.sqlite') as conn:
    # create a new graph
    query_graph = fornax.GraphHandle.create(conn)

    # insert the three nodes:
    #   'guardians' (id=0), 'star' (id=1), 'groot' (id=2)
    query_graph.add_nodes(label=query_labels)

    # alternatively:
    #    query_graph.add_nodes(id_src=query_labels)
    # since id_src can use any unique hashable items

    edges = [
        (0, 1), # edge between groot and guardians
        (0, 2)  # edge between star and guardians
    ]

    sources, targets = zip(*edges)
    query_graph.add_edges(sources, targets)
```

3.5 Search

We can create a query in an analogous way to creating graphs using a `QueryHandle`, a handle to a query stored in the fornax database. To create a useful query we need to insert the string similarity scores we computed in part 1. Fornax will use these scores and the graph edges to execute the query.

[13]:
```
with fornax.Connection('sqlite:///mydb.sqlite') as conn:
    query = fornax.QueryHandle.create(conn, query_graph, target_graph)
    query.add_matches(matches['query_id'], matches['target_id'], matches['score'])
```

Finally we can execute the query using a variety of options. We specify we want the top 5 best matches between the query graph and the target graph.

[14]:
```
with fornax.Connection('sqlite:///mydb.sqlite') as conn:
    %time results = query.execute(n=5)
```

```
CPU times: user 69.4 ms, sys: 2.35 ms, total: 71.8 ms
Wall time: 74.1 ms
```

```
/Users/dstaff/anaconda3/envs/fornax/lib/python3.6/site-packages/numpy/core/records.py:
↪513: FutureWarning: Numpy has detected that you may be viewing or writing to an␣
↪array returned by selecting multiple fields in a structured array.

This code may break in numpy 1.15 because this will return a view instead of a copy --
↪ see release notes for details.
  return obj.view(dtype=(self.dtype.type, obj.dtype))
```

3.6 Visualise

`query.execute` returns an object describing the search result. Of primary interest is the `graph` field which contains a list of graphs in `node_link_graph` format. We can use networkx to draw these graphs and visualise the results.

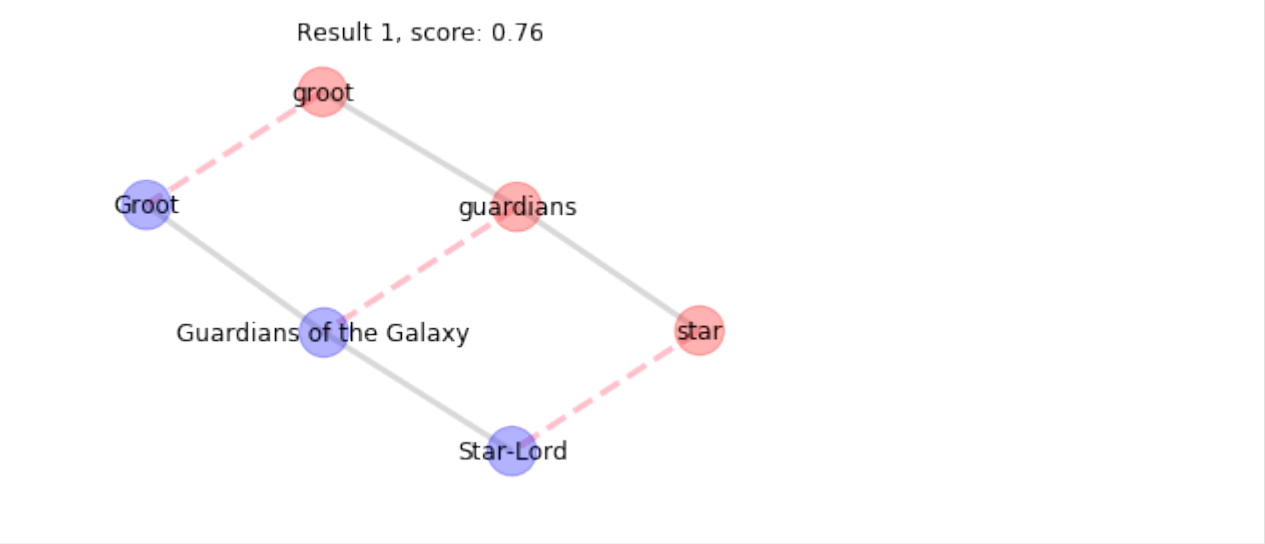
[15]:
```
def draw(graph):
    """ function for drawing a graph using matplotlib and networkx"""

    # each graph is already in node_link_graph format
    G = nx.json_graph.node_link_graph(graph)

    labels = {node['id']: node['label'] for node in graph['nodes']}
    node_colour = ['r' if node['type'] == 'query' else 'b' for node in graph['nodes']]
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, node_size=600, node_color=node_colour, alpha=.3)
    edgelist = [(e['source'], e['target']) for e in graph['links'] if e['type'] !=
↪'match']
    nx.draw_networkx_edges(G, pos, width=3, edgelist=edgelist, edge_color='grey',␣
↪alpha=.3)
    edgelist = [(e['source'], e['target']) for e in graph['links'] if e['type'] ==
↪'match']
    nx.draw_networkx_edges(G, pos, width=3, edgelist=edgelist, style='dashed', edge_
↪color='pink')
    nx.draw_networkx_labels(G, pos, font_size=12, font_family='sans-serif',␣
↪labels=labels)
```

Result 1 contains the best match. The three query nodes (in red) best match the three target nodes (in blue). The dashed lines show which pairs of query and target nodes matched each other. The blue nodes are a subgraph of the target graph. Note that the result does not describe the whole target graph because in principle it can be very large.

Here we can see that the blue subgraph has exactly the same shape as the red query graph. However, the labels are not exactly the same (e.g. `guardians != Guardians of the Galaxy`) so the result scores less than the maximum score of 1. However, we can see that our query graph is really similar to Groot and Star-Lord from Guardians of the Galaxy. Since this is the best match we know that

[16]:
```
for i, graph in enumerate(results['graphs'][:1]):
    plt.title('Result {0}, score: {1:.2f}'.format(1, 1. - graph['cost']))
    draw(graph)
    plt.xlim(-1.2,1.2)
    plt.ylim(-1.2,1.2)
    plt.axis('off')
    plt.show()
```

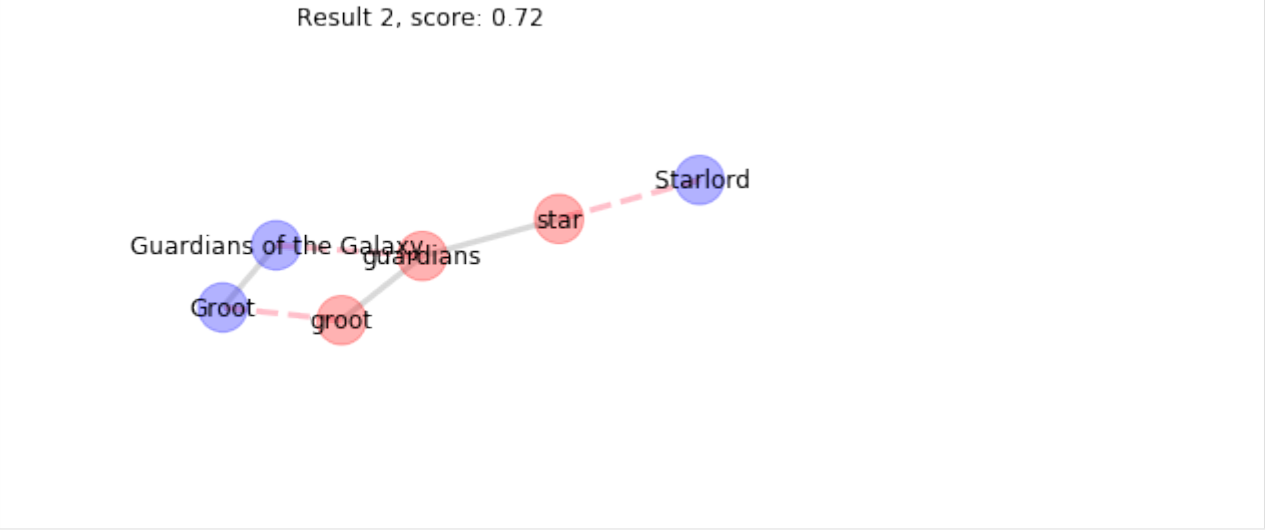
```
/Users/dstaff/anaconda3/envs/fornax/lib/python3.6/site-packages/networkx/drawing/nx_
↪pylab.py:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```

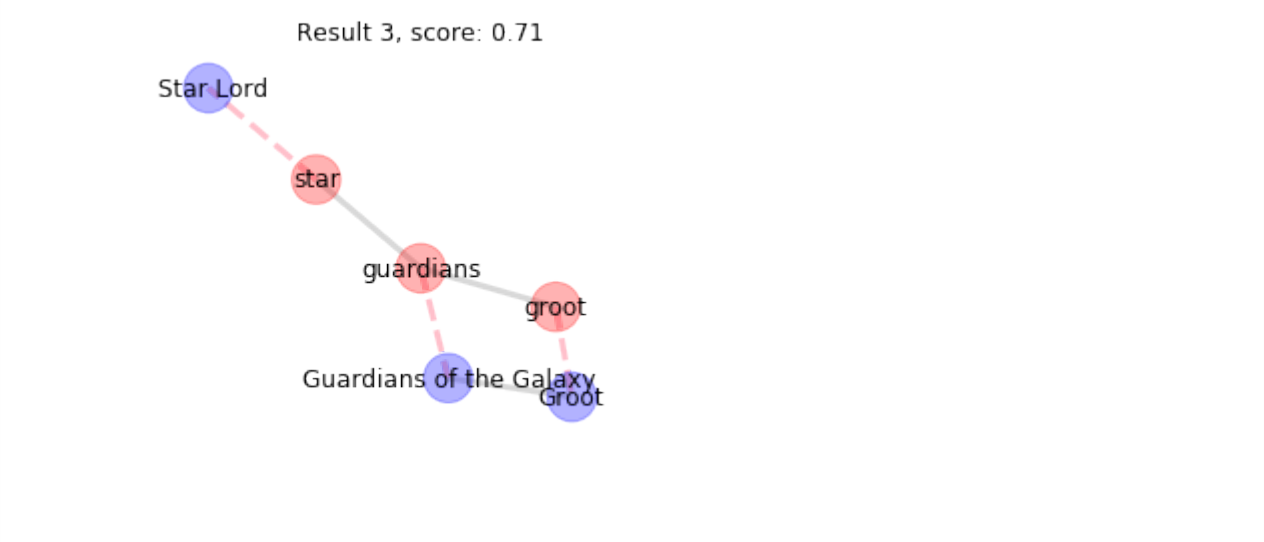


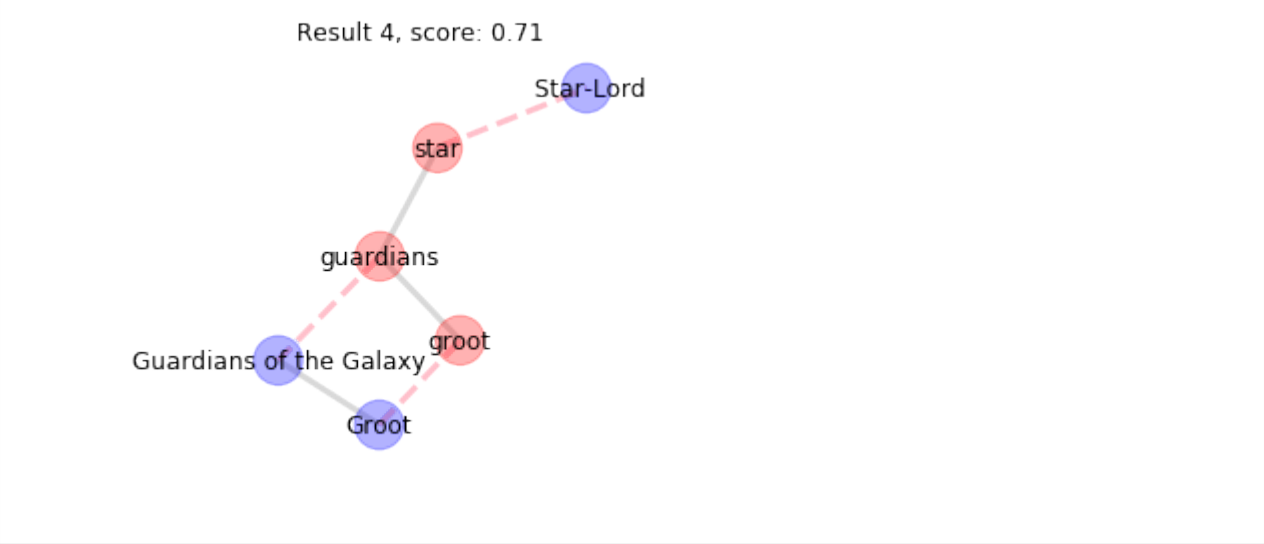
Results 2-4 have a lower score because `star` matches to a different node not adjacent to Guardians of the Galaxy. Further inspection would show that `star` has matched aliases of Star-Lord which are near Guardians of the Galaxy but not ajacent to it.

[17]:
```
for i, graph in enumerate(results['graphs'][1:4]):
    plt.title('Result {0}, score: {1:.2f}'.format(i+2, 1. - graph['cost']))
    draw(graph)
    plt.xlim(-1.2,1.2)
    plt.ylim(-1.2,1.2)
    plt.axis('off')
    plt.show()
```

```
/Users/dstaff/anaconda3/envs/fornax/lib/python3.6/site-packages/networkx/drawing/nx_
↪pylab.py:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```



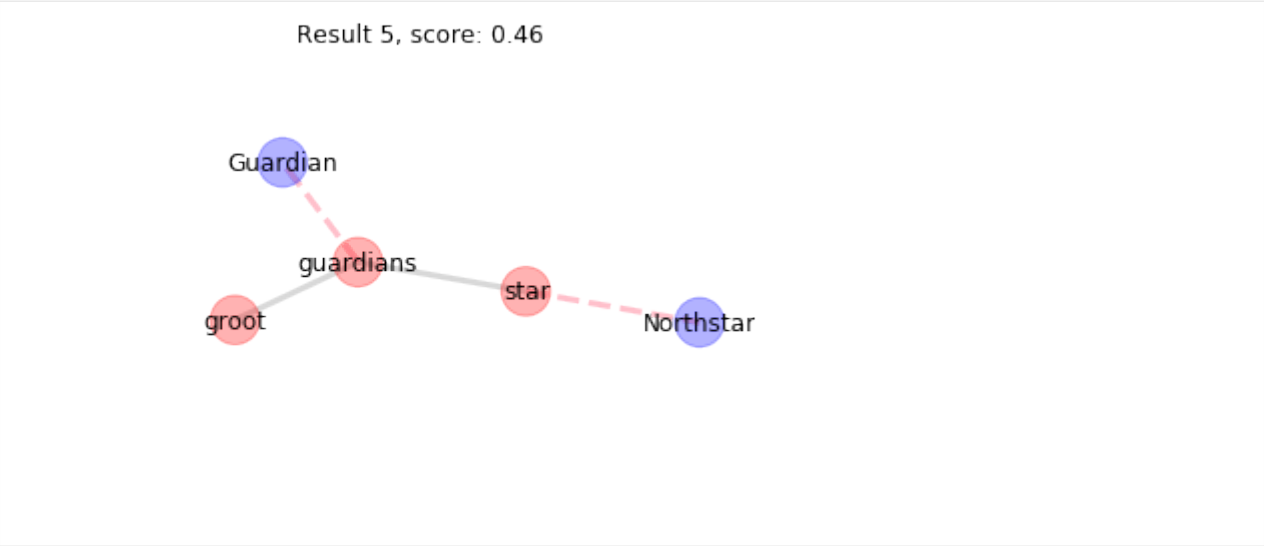




The final match pairs `guardians` and `star` to two nodes that do not have similar edges to the target graph. `groot` is not found in the target graph. The result gets a much lower score than the preceding results and we can be sure that any additional results will also be poor because the result are ordered.

[18]:
```
for i, graph in enumerate(results['graphs'][4:]):
    plt.title('Result {0}, score: {1:.2f}'.format(i+5, 1. - graph['cost']))
    draw(graph)
    plt.xlim(-1.2,1.2)
    plt.ylim(-1.2,1.2)
    plt.axis('off')
    plt.show()
```

```
/Users/dstaff/anaconda3/envs/fornax/lib/python3.6/site-packages/networkx/drawing/nx_
↪pylab.py:611: MatplotlibDeprecationWarning: isinstance(..., numbers.Number)
  if cb.is_numlike(alpha):
```


Result 5, score: 0.46
Guardian
guardians
star
groot
Northstar

CHAPTER 4

API

4.1 Introduction

This part of the documentation covers the the interface for creating an searching graphs using the fornax package. For the full documentation of the module api see *fornax.api module*.

All of the functionality in `fornax` can be accessed via the follwoing three classes.

- `Connection`
- `GraphHandle`
- `QueryHandle`

`Connection` is used to manage a connection to a SQL database. `GraphHandle` and `QueryHandle` are used to create, insert update and delete graphs and queries.

4.2 Connection API

Fornax stores and queries graphs using a database via a database connection. `Connection` manages the lifecycle of this database connection, the creation of database schema (if required) and any cleanup once the connection is closed.

class fornax.api.**Connection**(*url*, ***kwargs*)

Create a new database connection. If the database is empty `Connection` will create any missing schema.

Currrently sqlite and postgresql are activly supported as backend databases.

In addition to the open/close syntax, Connection supports the context manager syntax where the context is treaded as a transaction. Any changes will be automatically rolled back in the event of an exception:

```
with Connection("postgres:://user/0.0.0.0./mydb") as conn:
    graph = fornax.GraphHandle.create(conn)
```

Parameters **url** (*str*) – dialect[+driver]://user:password@host/dbname[?key=value..]

close()
Close the fornax database connection and free any connections in the connection pool

open()
Open the fornax database connection and create any absent tables and indicies

4.3 Graph API

Since Graphs are persisted in a database they are not represented directly by any object. Rather, graphs are accessed via a graph handle which permits the user to manipulate graphs via a *Connection* instance.

class fornax.api.**GraphHandle**(*connection: fornax.api.Connection*, *graph_id: int*)
Create a handle to an existing graph with id *graph_id* accessed via *connection*.

Parameters

- **connection** (*Connection*) – fornax database connection
- **graph_id** (*int*) – unique id for an existing graph

add_edges(*sources: Iterable*, *targets: Iterable*, ***kwargs*)
Append edges to a graph representing relationships between nodes

Parameters

- **sources** (*typing.Iterable*) – node id_src
- **targets** (*typing.Iterable*) – node id_src

Keyword arguments can be used to attach metadata to the edges. For example to add three edges with a relationship attribute friend or foe:

```
graph_handle.add_edges(
    sources=[0, 1, 2],
    targets=[1, 2, 0],
    relationship=['friend', 'friend', 'foe']
)
```

Keyword arguments can be used to attach any arbitrary JSON serialisable data to edges.

Note: The following reserved keywords are not reserved and will raise an exception

- *start*
- *end*
- *type*
- *weight*

---

add_nodes(***kwargs*)
Append nodes to a graph

Parameters **id_src** (*Iterable*) – An iterable of unique hashable identifiers, default None

Keyword arguments can be used to attached arbitrary JSON serialised metadata to each node:

```
#  create 3 nodes with ids: 0, 1, 2
#  and names 'Anne', 'Ben', 'Charles'
graph_handle.add_nodes(names=['Anne', 'Ben', 'Charles'])
```

By default, each node will be assigned a sequential integer id starting from 0. A custom id can be assigned using the *id_src* keyword provided that all of the ids are hashable:

```
#  create 3 nodes with ids: 'Anne', 'Ben', 'Charles'
#  and no explicit name field
graph_handle.add_nodes(id_src=['Anne', 'Ben', 'Charles'])
```

Note: *id* is a reserved keyword argument which will raise an exception

classmethod create(*connection: fornax.api.Connection*)
Create a new empty graph via *connection* and return a GraphHandle to it

Parameters **connection** (Connection) – a fornax database connection

Returns GraphHandle to a new graph

Return type *GraphHandle*

delete()
Delete this graph.

Delete the graph accessed through graph handle and all of the associated nodes and edges.

graph_id
Get the unique id for this graph

Graph id's are automaticly assigned at creation time.

classmethod read(*connection: fornax.api.Connection*, *graph_id: int*)
Create a new GraphHandle to an existing graph with unique identifier *graph_id*

Parameters

- **connection** (Connection) – a fornax database connection
- **graph_id** (*int*) – unique identifier for an existing graph

Returns A new graph handle to an existing graph

Return type *GraphHandle*

4.4 Query API

Like Graphs, queries exist in a database and a accessed via a handle. Queries are executed using the *QueryHandle.execute()* method.

A query brings together three important concenpts.

A **target graph** is the graph which is going to be searched.

A **query graph** is the subgraph that is being seached for in the target graph.

matches are label similarities between nodes in the query graph and target graph with a weight where $0 weight = 1$. Users are free to caculate label similarity scores however they like. Fornax only needs to know about non zero weights between matches.

Once a query has been created and executed it will return the *n* subgraphs in the target graph which are most similar to the query graph based on the similarity score between nodes and their surrounding neighbourhoods.

Note: Nodes in the target graph will only be returned from a query if they have a non zero similarity score to at least one node in the query graph.

class fornax.api.**QueryHandle**(*connection: fornax.api.Connection*, *query_id: int*)

Create a handle to an existing query via *connection* with unique id *query_id*.

Parameters

- **connection** (Connection) – a fornax database connection
- **query_id** (*int*) – unique id for an existing query

add_matches(*sources: Iterable[int], targets: Iterable[int], weights: Iterable[float], **kwargs*)

Add matches between the query graph and the target graph

Parameters

- **sources** (*typing.Iterable[int]*) – Iterable of *src_id* in the query graph
- **targets** (*typing.Iterable[int]*) – Iterable of *src_id* in the target graph
- **weights** (*typing.Iterable[float]*) – Iterable of weights between 0 and 1

For example, to add matches between

- node *0* in the query graph and node *0* in the target graph with weight *.9*
- node *0* in the query graph and node *1* in the target graph with weight *.1*

then:

```
query.add_matches([0, 0], [0, 1], [.9, .1])
```

Note: Adding weights that compare equal to zero will raise an exception.

classmethod create(*connection: fornax.api.Connection*, *query_graph: fornax.api.GraphHandle*, *target_graph: fornax.api.GraphHandle*)

Create a new query and return a QueryHandle for it

Parameters

- **connection** (Connection) – a fornax database connection
- **query_graph** (GraphHandle) – subgraph to find target graph
- **target_graph** (GraphHandle) – Graph to be searched

Returns new QueryHandle

Return type *QueryHandle*

delete()

Delete this query and any associated matches

execute(*n=5*, *hopping_distance=2*, *max_iters=10*)

Execute a fuzzy subgraph matching query finding the top *n* subgraph matches between the query graph and the target graph.

Parameters

- **n** (*int, optional*) – number of subgraph matches to return
- **hopping_distance** (*int, optional*) – lengthscale hyperparameter, defaults to 2

- **max_iters** (*int, optional*) – maximum number of optimisation iterations

Returns query result

Return type dict

query_graph() → fornax.api.GraphHandle
Get a QueryHandle for the query graph

Returns query graph

Return type *GraphHandle*

classmethod read(*connection: fornax.api.Connection*, *query_id: int*)
Create a new QueryHandle to an existing query with unique id *query_id* via *connection*.

Parameters

- **connection** (Connection) – a fornax database connection
- **query_id** (*int*) – unique identifier for a query

Returns new QueryHandle

Return type *QueryHandle*

target_graph() → fornax.api.GraphHandle
Get a QueryHandle for the target graph

Returns target graph

Return type *GraphHandle*

CHAPTER 5

Modules

5.1 Submodules

5.1.1 fornax.api module

class fornax.api.**Connection**(*url*, ***kwargs*)
Bases: object

Create a new database connection. If the database is empty *Connection* will create any missing schema.

Currrently sqlite and postgresql are activly supported as backend databases.

In addition to the open/close syntax, Connection supports the context manager syntax where the context is treaded as a transaction. Any changes will be automatically rolled back in the event of an exception:

```
with Connection("postgres:://user/0.0.0.0./mydb") as conn:
    graph = fornax.GraphHandle.create(conn)
```

Parameters **url** (*str*) – dialect[+driver]://user:password@host/dbname[?key=value..]

SQLITE_MAX_SIZE = 9223372036854775807

close()
Close the fornax database connection and free any connections in the connection pool

open()
Open the fornax database connection and create any absent tables and indicies

class fornax.api.**Edge**(*start: int*, *end: int*, *edge_type: str*, *meta: dict*, *weight=1.0*)
Bases: object

Representation of an Edge used internally be QueryHandle

Parameters

- **start** (*int*) – id of start node

- **end** (*int*) – id of end node
- **edge_type** (*str*) – either query target or match
- **meta** (*dict*) – dictionary of edge metadata to be json serialised
- **weight** – weight between 0 and 1, defaults to 1.

Raises **ValueError** – Raised if type is not *query*, *target* or *match*

end

meta

start

type

weight

class fornax.api.**GraphHandle**(*connection: fornax.api.Connection*, *graph_id: int*)

Bases: object

Create a handle to an existing graph with id *graph_id* accessed via *connection*.

Parameters

- **connection** (Connection) – fornax database connection
- **graph_id** (*int*) – unique id for an existing graph

add_edges(*sources: Iterable*, *targets: Iterable*, ***kwargs*)

Append edges to a graph representing relationships between nodes

Parameters

- **sources** (*typing.Iterable*) – node id_src
- **targets** (*typing.Iterable*) – node id_src

Keyword arguments can be used to attach metadata to the edges. For example to add three edges with a relationship attribute friend or foe:

```
graph_handle.add_edges(
    sources=[0, 1, 2],
    targets=[1, 2, 0],
    relationship=['friend', 'friend', 'foe']
)
```

Keyword arguments can be used to attach any arbitrary JSON serialisable data to edges.

Note: The following reserved keywords are not reserved and will raise an exception

- *start*
- *end*
- *type*
- *weight*

---

add_nodes(***kwargs*)

Append nodes to a graph

Parameters **id_src** (*Iterable*) – An iterable of unique hashable identifiers, default None

Keyword arguments can be used to attached arbitrary JSON serialised metadata to each node:

```
#  create 3 nodes with ids: 0, 1, 2
#  and names 'Anne', 'Ben', 'Charles'
graph_handle.add_nodes(names=['Anne', 'Ben', 'Charles'])
```

By default, each node will be assigned a sequential integer id starting from 0. A custom id can be assigned using the *id_src* keyword provided that all of the ids are hashable:

```
#  create 3 nodes with ids: 'Anne', 'Ben', 'Charles'
#  and no explicit name field
graph_handle.add_nodes(id_src=['Anne', 'Ben', 'Charles'])
```

Note: *id* is a reserved keyword argument which will raise an exception

classmethod create(*connection: fornax.api.Connection*)
Create a new empty graph via *connection* and return a GraphHandle to it

Parameters **connection** (Connection) – a fornax database connection

Returns GraphHandle to a new graph

Return type *GraphHandle*

delete()
Delete this graph.

Delete the graph accessed through graph handle and all of the associated nodes and edges.

graph_id
Get the unique id for this graph

Graph id's are automaticly assigned at creation time.

classmethod read(*connection: fornax.api.Connection*, *graph_id: int*)
Create a new GraphHandle to an existing graph with unique identifier *graph_id*

Parameters

- **connection** (Connection) – a fornax database connection
- **graph_id** (*int*) – unique identifier for an existing graph

Returns A new graph handle to an existing graph

Return type *GraphHandle*

exception fornax.api.**InvalidEdgeError**(*message: str*)
Bases: Exception

exception fornax.api.**InvalidMatchError**(*message: str*)
Bases: Exception

exception fornax.api.**InvalidNodeError**(*message: str*)
Bases: Exception

class fornax.api.**Node**(*node_id: int*, *node_type: str*, *meta: dict*)
Bases: object

Representation of a Node use internally by QueryHandle

Parameters

- **node_id** (*int*) – unique id of a node
- **node_type** (*str*) – either *source* or *target*
- **meta** (*dict*) – meta data to attach to a node to be json serialised

Raises **ValueError** – Raised is type is not either *source* or *target*

id

meta

type

class fornax.api.**NullValue**

Bases: object

A dummy nul value that will cause an exception when serialised to json

class fornax.api.**QueryHandle**(*connection: fornax.api.Connection*, *query_id: int*)

Bases: object

Create a handle to an existing query via *connection* with unique id *query_id*.

Parameters

- **connection** (Connection) – a fornax database connection
- **query_id** (*int*) – unique id for an existing query

add_matches(*sources: Iterable[int], targets: Iterable[int], weights: Iterable[float], **kwargs*)

Add matches between the query graph and the target graph

Parameters

- **sources** (*typing.Iterable[int]*) – Iterable of *src_id* in the query graph
- **targets** (*typing.Iterable[int]*) – Iterable of *src_id* in the target graph
- **weights** (*typing.Iterable[float]*) – Iterable of weights between 0 and 1

For example, to add matches between

- node *0* in the query graph and node *0* in the target graph with weight *.9*
- node *0* in the query graph and node *1* in the target graph with weight *.1*

then:

```
query.add_matches([0, 0], [0, 1], [.9, .1])
```

Note: Adding weights that compare equal to zero will raise an exception.

classmethod create(*connection: fornax.api.Connection*, *query_graph: fornax.api.GraphHandle*, *target_graph: fornax.api.GraphHandle*)

Create a new query and return a QueryHandle for it

Parameters

- **connection** (Connection) – a fornax database connection
- **query_graph** (GraphHandle) – subgraph to find target graph
- **target_graph** (GraphHandle) – Graph to be searched

Returns new QueryHandle

Return type *QueryHandle*

delete()
Delete this query and any associated matches

execute(*n=5*, *hopping_distance=2*, *max_iters=10*)
Execute a fuzzy subgraph matching query finding the top *n* subgraph matches between the query graph and the target graph.

Parameters

- **n** (*int, optional*) – number of subgraph matches to return
- **hopping_distance** (*int, optional*) – lengthscale hyperparameter, defaults to 2
- **max_iters** (*int, optional*) – maximum number of optimisation iterations

Returns query result

Return type dict

static is_between(*target_ids*, *edge*)

query_graph() → fornax.api.GraphHandle
Get a QueryHandle for the query graph

Returns query graph

Return type *GraphHandle*

classmethod read(*connection: fornax.api.Connection*, *query_id: int*)
Create a new QueryHandle to an existing query with unique id *query_id* via *connection*.

Parameters

- **connection** (Connection) – a fornax database connection
- **query_id** (*int*) – unique identifier for a query

Returns new QueryHandle

Return type *QueryHandle*

target_graph() → fornax.api.GraphHandle
Get a QueryHandle for the target graph

Returns target graph

Return type *GraphHandle*

5.1.2 fornax.model module

class fornax.model.**Edge**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Joins Nodes in a Graph

end

end_node

graph_id

meta

start

start_node

class fornax.model.**Graph**(**kwargs)
Bases: sqlalchemy.orm.decl_api.Base

A graph containing nodes and edges

graph_id

class fornax.model.**Match**(**kwargs)
Bases: sqlalchemy.orm.decl_api.Base

Joins Query Nodes to Candidate Target Nodes

end

end_graph_id

end_node

meta

query_id

start

start_graph_id

start_node

weight

class fornax.model.**Node**(**kwargs)
Bases: sqlalchemy.orm.decl_api.Base

Node in a Graph

graph_id

meta

neighbours()

node_id

class fornax.model.**Query**(**kwargs)
Bases: sqlalchemy.orm.decl_api.Base

end_graph_id

query_id

start_graph_id

5.1.3 fornax.opt module

class fornax.opt.**Base**
Bases: numpy.recarray

A Base class for subclassing numpy record arrays

Returns: np.recarray – A subclass of np.recarray

columns = []

types = []

class fornax.opt.**InferenceCost**
Bases: *fornax.opt.Base*

A table representing all valid inference costs between query node u and target node v

columns = ['v', 'u', 'cost']

cost
Get column cost - all valid inference costs for query node v and target node u.

Eq 14 in the paper (U)

Returns: np.ndarray – array of costs as floats

types = [<class 'numpy.int64'>, <class 'numpy.int64'>, <class 'numpy.float32'>]

u
Get column u

Returns: np.ndarray – array of target node ids as integers

v
Get column v

Returns: np.ndarray – array of query node ids as integers

class fornax.opt.**NeighbourHoodMatchingCosts**
Bases: *fornax.opt.Base*

Represents a table of all valid neighbourhood matching costs

columns = ['v', 'u', 'vv', 'uu', 'cost']

cost
Get column cost - all valid neighbourhood matching costs.

Eq 2 in the paper - multiplied by 1 - lambda

Returns: np.ndarray – array of costs and floats

types = [<class 'numpy.int64'>, <class 'numpy.int64'>, <class 'numpy.int64'>, <class '

u
Get column u

Returns: np.ndarray – array of target node ids as integers

uu
Get column uu - written u prime (u') in the paper where u' is a target node within hopping distance h of target node u

Returns: np.ndarray – array of target node ids as integers

v
Get column v

Returns: np.ndarray – array of query node ids as integers

vv
Get column vv - written v prime (v') in the paper where v' is a query node within hopping distance h of query node v

Returns: np.ndarray – array of query node ids as integers

class fornax.opt.**OptimalMatch**
Bases: *fornax.opt.Base*

Table representing the cost of the optimal match for query node v going to u

columns = ['v', 'u', 'cost']

cost

Get column cost - the optimal matching cost for u going to v.

Eq 10 in the paper (O)

Returns: np.ndarray – array of costs as floats

types = [<class 'numpy.int64'>, <class 'numpy.int64'>, <class 'numpy.float32'>]

u

Get column u

Returns: np.ndarray – array of target node ids as integers

v

Get column v

Returns: np.ndarray – array of query node ids as integers

class fornax.opt.**PartialMatchingCosts**

Bases: *fornax.opt.Base*

A table representing all valid partial matching costs

columns = ['v', 'u', 'vv', 'cost']

cost

Get column cost - all valid partial matching costs.

Eq 13 in the paper (W) - but with beta multiplied by a factor of 1 - lambda

Returns: np.ndarray – array of costs as floats

types = [<class 'numpy.int64'>, <class 'numpy.int64'>, <class 'numpy.int64'>, <class '

u

Get column u

Returns: np.ndarray – array of target node ids as integers

v

Get column v

Returns: np.ndarray – array of query node ids as integers

vv

Get column vv - written v prime (v') in the paper where v' is a query node within hopping distance h of query node v

Returns: np.ndarray – array of query node ids as integers

class fornax.opt.**QueryResult**

Bases: *fornax.opt.Base*

Represents a query from the database as a numpy rec array

columns = ['v', 'u', 'vv', 'uu', 'dist_v', 'dist_u', 'weight']

dist_u

The hopping distance between target node u and target node uu (u')

Returns: np.ndarray – array of hopping distances as integers

dist_v
The hopping distance between query node v and query node vv (v')

Returns: np.ndarray – array of hopping distances as integers

types = [<class 'numpy.int64'>, <class 'numpy.int64'>, <class 'numpy.int64'>, <class '

u
Get column u

Returns: np.ndarray – array of target node ids as integers

uu
Get column uu - written u prime (u') in the paper where u' is a target node within hopping distance h of target node u

values less than zero indicate that uu (u') has no corresponding matches to any node v'

Returns: np.ndarray – array of target node ids as integers

v
Get column v

Returns: np.ndarray – array of query node ids as integers

vv
Get column vv - written v prime (v') in the paper where v' is a query node within hopping distance h of query node v

Returns: np.ndarray – array of query node ids as integers

weight
String matching score between uu (u') and vv (v')

Returns: np.ndarray – array of floating point weights

class fornax.opt.**Refiner**(*neighbourhood_matching_costs: fornax.opt.NeighbourHoodMatchingCosts*)
Bases: object

Take each of the matches and recursivly find all of their neighbours via a greedy algorithm

static valid_neighbours(*first: tuple*, *second: tuple*)
Function that governs a valid hop between nodes

Arguments: first {int, int} – source query_node, target_node id pair second {int, int} – target query_node, target_node id pair

Returns: Bool – True is a valid transition

fornax.opt.**group_by**(*columns*, *arr*)
Split an array into n slices where 'columns' are all equal within each slice

Arguments: columns {List[str]} – a list of column names arr {np.array} – a numpy structured array

Returns keys: np.array – the column values uniquly identifying each group groups: List[np.array] – a list of numpy arrays

fornax.opt.**group_by_first**(*columns*, *arr*)
Split an array into n slices where 'columns' all compare equal within each slide Take the first row of each slice Combine each of the rows into a single array through concatination

Arguments: columns {[str]} – a list of column names arr {[type]} – a numpy structured array

Returns: np.array - new concatinated array

fornax.opt.**solve**(*records: List[tuple], max_iters=10, hopping_distance=2*)
Generate a set of subgraph matches and costs from a query result

Arguments: records {List[tuple]}

5.1.4 fornax.select module

fornax.select.**join**(*query_id: int, h: int, offsets: Tuple[int, int] = None*) → sqlalchemy.orm.query.Query

fornax.select.**neighbours**(*h: int, start: bool, query_id: int*) → sqlalchemy.orm.query.Query

5.1.5 Module contents

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

N

O

P

Q

R

S

T

U

V

W