django-directed

Jack Linke

Jan 09, 2024

USER GUIDE

Tools for building, querying, manipulating, and exporting directed graphs with django.

Documentation can be found at https://django-directed.readthedocs.io/en/latest/

Caution: This project is very much a Work In Progress, and is not production-ready. Once it is in a more complete state, it will be moved to the github Watervize organization for long-term development and maintenance.

CHAPTER

ONE

FUNDAMENTALS

Graphs in django-directed are constructed with three models (or potentially more in case of extended features).

- **Graph**: Represents a connected graph of nodes and edges. It makes it easy to associate metadata with a particular graph and to run commands and queries limited to a subset of all the Edges and Nodes in the database.
- **Edge**: Connects Nodes to one another within a particular Graph instance.
- **Node**: A node can belong to more than one Graph. This allows us to represent multi-dimensional or multi-layered graphs.

django-directed includes model factories for building various types of directed graphs. As an example, imagine a project in which you display family trees and also provide a searchable interface for research papers about family trees, where papers can be linked to previous papers that they cite. Both of these concepts can be represented by a Directed Acyclic Graph (DAG), and within your project you could create a set of DAG models for the family tree app and another set of DAG models for the academic papers app.

CHAPTER

TWO

QUICKSTART

Assuming you have already started a django project and an app named `myapp`

2.1 Install django-directed

```
pip install django-directed
```

2.2 Create the concrete models

Using the DAG factory, create a set of concrete Graph, Edge, and Node models for your project. Perform the following steps in your app's models.py

Build a configuration object that will be passed into the factory. Here, we are using the simplest configuration which specifies the graph type (default options include 'CYCLIC', 'DAG', 'POLYTREE', 'ARBORESCENCE') and the model names (with `appname.ModelName`). We fall back to the default values for all other configuration options.

```
from django_directed.config import GraphConfig


my_config = GraphConfig(
    graph_type="DAG",
    graph_model_name="myapp.DAGGraph",
    edge_model_name="myapp.DAGEdge",
    node_model_name="myapp.DAGNode",
)
```

Create the concrete models from a model factory service. In this example, we are adding some fields as an example of what you might do in your own application.

```
from django.db import models
from django_directed.models.model_factory import factory


# Create DAG factory instance
dag = factory.create(config=my_config)

# Create concrete models
class DAGGraph(dag.graph()):
```

(continues on next page)

(continued from previous page)

```
    metadata = models.JSONField(default=str, blank=True)


class DAGEdge(dag.edge()):
    name = models.CharField(max_length=101, blank=True)
    weight = models.SmallIntegerField(default=1)

    def save(self, *args, **kwargs):
        self.name = f"{self.parent.name} -to- {self.child.name}"
        super().save(*args, **kwargs)


class DAGNode(dag.node()):
    name = models.CharField(max_length=50)
    weight = models.SmallIntegerField(default=1)
```

Note: The model names here (DAGGraph, etc) are for example only. You are welcome to use whatever names you like, but the model names should match the names provided in the configuration.

2.3 Migrations

As usual when working with models in django, we need to make migrations and then run them.

```
python manage.py makemigrations
python manage.py migrate
```

2.4 Build a couple graphs using our DAG models

Tip: We are using the `graph_context_manager` here, which is provided in django-directed for convenience. If you decide not to use this context manager, you need to provide the graph instance when creating or querying with Nodes and Edges.

```
from django_directed.context_managers import graph_scope

from myapp.models import DAGGraph, DAGEdge, DAGNode


# Create a graph instance
first_graph = DAGGraph.objects.create()
# Create a second graph instance, which will share nodes with first_graph
another_graph = DAGGraph.objects.create()

with graph_scope(first_graph):

    # Create several nodes (not yet connected)
```

(continues on next page)

(continued from previous page)

```
    root = DAGNode.objects.create(name="root")

    a1 = DAGNode.objects.create(name="a1")
    a2 = DAGNode.objects.create(name="a2")
    a3 = DAGNode.objects.create(name="a3")

    b1 = DAGNode.objects.create(name="b1")
    b2 = DAGNode.objects.create(name="b2")
    b3 = DAGNode.objects.create(name="b3")
    b4 = DAGNode.objects.create(name="b4")

    c1 = DAGNode.objects.create(name="c1")
    c2 = DAGNode.objects.create(name="c2")

    # Connect nodes with edges
    root.add_child(a1)
    root.add_child(a2)

    # You can add from either side of the relationship
    a3.add_parent(root)

    b1.add_parent(a1)
    a1.add_child(b2)
    a2.add_child(b2)
    a3.add_child(b3)
    a3.add_child(b4)

    b3.add_child(c2)
    b3.add_child(c1)
    b4.add_child(c2)

with graph_scope(another_graph):

    # Connect nodes with edges
    c1 = DAGNode.objects.get(name="c1")
    c2 = DAGNode.objects.get(name="c2")

    c1.add_child(c2)
```

2.4.1 django-directed

[][license]

Tools for building, querying, manipulating, and exporting directed graphs with django.

Documentation can be found at https://django-directed.readthedocs.io/en/latest/

Caution: This project is very much a Work In Progress, and is not production-ready. Once it is in a more complete state, it will be moved to the github Watervize organization for long-term development and maintenance.

Fundamentals

Graphs in django-directed are constructed with three models (or potentially more in case of extended features).

- **Graph**: Represents a connected graph of nodes and edges. It makes it easy to associate metadata with a particular graph and to run commands and queries limited to a subset of all the Edges and Nodes in the database.
- **Edge**: Connects Nodes to one another within a particular Graph instance.
- **Node**: A node can belong to more than one Graph. This allows us to represent multi-dimensional or multi-layered graphs.

django-directed includes model factories for building various types of directed graphs. As an example, imagine a project in which you display family trees and also provide a searchable interface for research papers about family trees, where papers can be linked to previous papers that they cite. Both of these concepts can be represented by a Directed Acyclic Graph (DAG), and within your project you could create a set of DAG models for the family tree app and another set of DAG models for the academic papers app.

Quickstart

Assuming you have already started a django project and an app named `myapp`

Install django-directed

```
pip install django-directed
```

Create the concrete models

Using the DAG factory, create a set of concrete Graph, Edge, and Node models for your project. Perform the following steps in your app's models.py

Build a configuration object that will be passed into the factory. Here, we are using the simplest configuration which specifies the graph type (default options include 'CYCLIC', 'DAG', 'POLYTREE', 'ARBORESCENCE') and the model names (with `appname.ModelName`). We fall back to the default values for all other configuration options.

```
from django_directed.config import GraphConfig


my_config = GraphConfig(
    graph_type="DAG",
    graph_model_name="myapp.DAGGraph",
    edge_model_name="myapp.DAGEdge",
    node_model_name="myapp.DAGNode",
)
```

Create the concrete models from a model factory service. In this example, we are adding some fields as an example of what you might do in your own application.

```
from django.db import models
from django_directed.models.model_factory import factory


# Create DAG factory instance
dag = factory.create(config=my_config)

# Create concrete models
class DAGGraph(dag.graph()):
    metadata = models.JSONField(default=str, blank=True)


class DAGEdge(dag.edge()):
    name = models.CharField(max_length=101, blank=True)
    weight = models.SmallIntegerField(default=1)

    def save(self, *args, **kwargs):
        self.name = f"{self.parent.name} -to- {self.child.name}"
        super().save(*args, **kwargs)


class DAGNode(dag.node()):
    name = models.CharField(max_length=50)
    weight = models.SmallIntegerField(default=1)
```

Note: The model names here (DAGGraph, etc) are for example only. You are welcome to use whatever names you like, but the model names should match the names provided in the configuration.

Migrations

As usual when working with models in django, we need to make migrations and then run them.

```
python manage.py makemigrations
python manage.py migrate
```

Build a couple graphs using our DAG models

Tip: We are using the `graph_context_manager` here, which is provided in django-directed for convenience. If you decide not to use this context manager, you need to provide the graph instance when creating or querying with Nodes and Edges.

```
from django_directed.context_managers import graph_scope

from myapp.models import DAGGraph, DAGEdge, DAGNode


# Create a graph instance
```

(continues on next page)

(continued from previous page)

```
first_graph = DAGGraph.objects.create()
# Create a second graph instance, which will share nodes with first_graph
another_graph = DAGGraph.objects.create()

with graph_scope(first_graph):

    # Create several nodes (not yet connected)
    root = DAGNode.objects.create(name="root")

    a1 = DAGNode.objects.create(name="a1")
    a2 = DAGNode.objects.create(name="a2")
    a3 = DAGNode.objects.create(name="a3")

    b1 = DAGNode.objects.create(name="b1")
    b2 = DAGNode.objects.create(name="b2")
    b3 = DAGNode.objects.create(name="b3")
    b4 = DAGNode.objects.create(name="b4")

    c1 = DAGNode.objects.create(name="c1")
    c2 = DAGNode.objects.create(name="c2")

    # Connect nodes with edges
    root.add_child(a1)
    root.add_child(a2)

    # You can add from either side of the relationship
    a3.add_parent(root)

    b1.add_parent(a1)
    a1.add_child(b2)
    a2.add_child(b2)
    a3.add_child(b3)
    a3.add_child(b4)

    b3.add_child(c2)
    b3.add_child(c1)
    b4.add_child(c2)

with graph_scope(another_graph):

    # Connect nodes with edges
    c1 = DAGNode.objects.get(name="c1")
    c2 = DAGNode.objects.get(name="c2")

    c1.add_child(c2)
```

Resulting model data

Here is the resulting data in each model (ignoring the custom fields added in the concrete model definitions).

myapp.DAGGraph

```
 id
----
  1
  2
```

myapp.DAGNode

```
 id  | name | graph
-----+------+------
  1  | root |  1
  2  | a1   |  1
  3  | a2   |  1
  4  | a3   |  1
  5  | b1   |  1
  6  | b2   |  1
  7  | b3   |  1
  8  | b4   |  1
  9  | c1   |  1
  10 | c2   |  1
```

myapp.DAGEdge

```
id  | parent_id | child_id | name    | graph
----+-----------+----------+---------+------
 1  |  1        |  2       | root a1 |  1
 2  |  1        |  3       | root a2 |  1
 3  |  1        |  4       | root a3 |  1
 4  |  2        |  5       | a1 b1   |  1
 5  |  2        |  6       | a1 b2   |  1
 6  |  3        |  6       | a2 b2   |  1
 7  |  4        |  7       | a3 b3   |  1
 8  |  4        |  8       | a3 b4   |  1
 9  |  7        |  10      | b3 c2   |  1
 10 |  7        |  9       | b3 c1   |  1
 11 |  8        |  10      | b4 c2   |  1
 12 |  9        |  10      | c1 c2   |  2
```

Graph visualization

Note: In the visualized graph below, both of the green nodes (c1) refer to the same Node instance, which is associated with two different graph instances. Likewise, both blue nodes (c2) refer to the same Node instance.

Note: The mermaid.js diagrams require different markup for GitHub markdown compared to display within ReadTheDocs. Both versions are included here, but one will likely appear as code depending on where you are viewing this file.

Graph for display on GitHub

```
graph TD;
    root((root));
    a1((a1));
    a2((a2));
    a3((a3));
    b1((b1));
    b2((b2));
    b3((b3));
    b4((b4));
    c1((c1));
    c2((c2));
    c1X((c1));
    c2X((c2));

    root-->a1;
    root-->a2;
    root-->a3;
    a1-->b1;
    a1-->b2;
    a2-->b2;
    a3-->b3;
    a3-->b4;
    b3-->c1;
    b3-->c2;
    b4-->c2;

    c1X-->c2X;

    style c1 fill:#48A127,stroke:#333,stroke-width:4px;
    style c1X fill:#48A127,stroke:#333,stroke-width:4px;
    style c2 fill:#279BA1,stroke:#333,stroke-width:4px;
    style c2X fill:#279BA1,stroke:#333,stroke-width:4px;

    linkStyle default fill:none,stroke:gray
```

Graph for display on ReadTheDocs

Find the shortest path between two nodes

First, let us try to get the shortest path from `c1` and `c2` on `first_graph`, where no path exists:

```
with graph_scope(first_graph):
    c1 = DAGNode.objects.get(name="c1")
    c2 = DAGNode.objects.get(name="c2")

    print(c1.shortest_path(c2))
```

Output: `django_directed.models.NodeNotReachableError`

Next, we will perform the same query on `another_graph`, which *does* have a path from `c1` to `c2` through a single Edge. The value returned is a QuerySet of the Nodes in the path.

```
with graph_scope(another_graph):
    c1 = DAGNode.objects.get(name="c1")
    c2 = DAGNode.objects.get(name="c2")

    print(c1.shortest_path(c2))
```

Output: `<QuerySet [<NetworkNode: c1>, <NetworkNode: c2>]>`

For additional methods of querying, see the API docs for Graph, Edge, and Node.

Example apps

Note: These are in-progress, and not ready for actual use.

A series of example apps demonstrating vaious aspects and techniques of using django-directed.

- **Airports** - An app demonstrating one method of working with multidimensional graphs to model airports with a common set of nodes, and edges for each of the connecting airlines.
- **Electrical Grids** - Demonstrate graphs of neighborhood electrical connections and meters.
- **Family Trees** - Demonstrates building family trees for multiple mythological families.
- **Forums** - Forums and threaded comments.
- **NetworkX Graphs** - Demonstration of using NetworkX alongside django-directed.

See the Example Apps folder.

Why not use a graph database instead?

- **Compatibility** - Graph databases don't play very nicely with Django and the Django ORM. There are 3rd party packages to shoehorn in the required functionality, but django is designed for relational databases.
- **Simplicity** - If most of the work you are doing needs a relational database, mixing an additional entirely different kind of database into the project might not be ideal.
- **Tradeoffs** - Graph databases are not a panacea. They bring their own set of pros and cons. Maybe a graph database is ideal for your project. But maybe you'll do just as well using django-directed. I encourage you to read up on the benefits graph databases bring, the issues they solve, and also the areas where they do not perform as well as a relational database.

2.4.2 Installation

Basic

django-directed can be installed with `pip`

```
pip install django-directed
```

In future iterations of this project, expect to see the option to install 'extras' for access to additional features and capabilities.

External packages and plugins

2.4.3 Terminology and Definitions

Learning to use graphs can be challenging because some concepts have multiple equivalent or similar terms and definitions. For instance, the words 'node' and 'vertex' typically mean the same thing, but some industries or fields may prefer one to the other.

To help clarify what is meant throughout this project, we define the following terms and definitions. We make heavy use of familial terms, which can help with mentally visualizing the concepts.

This document does is not intended as a course in general graph theory. A graph in the context of this project is made up of nodes which are connected by edges. Edges typically link two nodes *asymmetrically* in all of the directed graphs within django-directed.

Node

Here, `A` is a *node*. Another equivalent name for *node* that you may sometimes hear is *vertex*. While they are interchangeable, we will use the term *node* (or *nodes* for plural) exclusively within this project for consistency.

Edge

Here, `e` is an *edge* in the graph between nodes `A` and `B`. Edges connect nodes, and are directed (denoted here with an arrowhead). Edges are also called *lines*, *links*, *arcs*, or *arrows*. For consistency, this project will always use the term *edge* (or *edges* for plural).

Root

Here, Node A is the *root* of the graph. It has an in-degree (number of edges coming 'in') of 0.

Roots

Some types of graphs may have multiple roots. Here, Nodes `A` and `B` are *roots* of the graph. Again, if the in-degree is 0, the node is a root.

Leaf / Leaves

Here, Nodes `D` and `e` are *leaves* in the graph. They both have an out-degree (number of edges 'out' of the node) of 0.

Orphan

In a given Graph, an *orphan* is a node with no parents nor children. Orphans have an in-degree of 0 *and* and out-degree of 0. Here, node `E` is an orphan. There are no edges connecting it to any other node.

(Note, there is no equivalent for edges. Every edge connects two [or in special cases, more] nodes.)

Parent / Parents

The *parents* for a given node *x*, if any exist, are those nodes which have a directed edge 'in' to node *x*. In graph theory, this may be refered to as a direct predecessor.

Here, node `A` is a *parent* of node `B`, and node `B` is a *parent* of node `C`. Depending on the type of graph, nodes may have zero, one, or multiple parents.

We also refer to *parent edges*, which are the directed edges themselves which point to the node. In this example, edge `e1` is a *parent edge* of node `B`, and edge `e2` is a *parent edge* of node `C`.

Child / Children

The *children* for a given node *x*, if any exist, are those nodes which have a directed edge 'out' from node *x*. In graph theory, this may be refered to as a direct successor.

Here, node `B` is a *child* of node `A`, and node `C` is a *child* of node `B`. Depending on the type of graph, nodes may have zero, one, or multiple children.

We also refer to *children edges*, which are the directed edges themselves which point from the node. In this example, edge `e1` is a *child edge* of node `A`, and edge `e2` is a *child edge* of node `B`.

Ancestors

All nodes in connected paths in a rootward direction. In graph theory, this may be refered to as predecessors.

In this example, the *ancestors* for node `I` are nodes `A`, `C`, `E`, and `F`.

Descendants

All nodes in connected paths in a leafward direction. In graph theory, this may be refered to as successors.

In this example, the *descendants* for node `C` are nodes `D`, `F`, `G`, `H`, and `I`.

Clan

The clan of a node includes all ancestor nodes, the node itself, and all descendant nodes. In graph theory, this can be refered to as the maximal paths through a given node.

In this example, the *clan* for node `F` includes nodes `A`, `C`, `E`, `H`, and `I`.

Siblings

All nodes that share a parent with this node, excluding the node itself.

In this example, the *siblings* of node `C` are nodes `B`, and `E`, because they all have node `A` in common as a parent.

Partners

All nodes that share a child with this node, excluding the node itself.

In this example, the *partners* of node `C` are nodes `B`, and `E`, because nodes `B` and `C` share node `D` as a child, and nodes `C` and `E` share node `F` as a child.

Distance

The shortest number of hops from one node to a target node. The *distance* between node `C` and node `H` is 2. This is because the path from `C` to `F` to `H` involves 2 edges.

There is another path from `C` to `H` through nodes `D` and `G`, but that path is longer (3 edges), and when we refer to *distance* in this project, we always mean the smallest number of hops.

Node Depth

The distance of the node from furthest root in the graph. Because this can be a bit challenging to visualize, a few examples are provided below.

Because node `A` is the highest (and only) root in the following graph, its *node depth* is 0.

Using the same graph as before, consider the depth of node `H`. There is only a single root (node `A`) in this graph, and the distance between node `A` and node `H` is 3. So the *node depth* of node `H` is 3.

Finally, we will look at a more complicated example with multiple roots at different levels. Here we want the *node depth* of node `F`.

While both nodes `A` and `D` are roots in this graph (they have in-degree of 0), node `A` has a greater distance from node `F`, so we determine the depth of node `F` from the viewpoint of node `A`. It takes 3 hops to reach node `F` from node `A`, so the *node depth* of node `F` is 3.

2.4.4 Concepts

Internally, django-directed uses a combination of factories and abstract models, which makes possible:

- Composition of graph model sets with limited repetition of code
- Registering base model types for use with other project and in django-directed-admin
- Passing a standardized configuration object to the factory to change model functionality

Within a Django project utilizing django-directed the graph, edges, and nodes are represented as distinct concrete models, and multiple types of graphs can be built within the same project. These three work together to provide a consolidated API for working with graphs.

- a Graph model (extended from BaseGraph and then AbstractGraph)
- an Edge model (extended from BaseEdge and then AbstractEdge)
- a Node model (extended from BaseNode and then AbstractNode)

The connected graph is defined by the Edges associated with a Graph instance. This does mean an additional join on the Gaph table, but for typical use-cases the ratio of Graph instances to those of Nodes and Edges is tiny.

2.4.5 Building Graphs

Building graphs in django-directed starts with configuring the type of graph you want to use, writing the models, and then creating and running migrations.

Configuration

Models

Model Instantiation

Model Migrations

2.4.6 Querying Graphs

Work In Progress

2.4.7 Manipulating Graphs

Work In Progress

2.4.8 Exporting Graphs

Work In Progress

2.4.9 Graph

WORK IN PROGRESS

Manager/QuerySet Methods

For future consideration:

- clone()

Methods used for building/manipulating

For future consideration:

- add_node() add node to graph, optionally providing a list of parent nodes
- remove_nodes(nodes) removes nodes from the graph
- add_edge() adds connections or paths between nodes in graphs
- remove_edges(edges) removes connection or paths between nodes in graphs

Methods returning a QuerySet of Nodes

None

Methods returning a QuerySet of Edges

None

Methods returning a Boolean

None

Methods returning other values

`node_count()`
:return: Number of Nodes in the Graph :rtype: int

`edge_count()`
:return: Number of Edges in the Graph :rtype: int

`graph_hash()`
:return: Hash value for the Graph :rtype: TBD

Model Methods

Methods used for building/manipulating an instance

None

Methods returning a QuerySet of Nodes

None

Methods returning a QuerySet of Edges

None

Methods returning a Boolean

`has_connection`(*node_from*, *node_to*)

Checks if a connection or path exists between two Node instances, within the current Graph.

:param Node node_from: The starting Node :param Node node_to: The ending Node :return: True if path exists from `node_from` to `node_to` :rtype: bool

For future consideration:

- contains_value() check if a graph instance contains a certain value

Methods returning other values

None

2.4.10 Node

Manager/QuerySet Methods

Methods used for building/manipulating

None

Methods returning a QuerySet of Nodes

`roots`(*node=None*)

Returns a QuerySet of all root Nodes (nodes with no parents) in the Node model.

:param Node node: (optional) if specified, returns only the roots for that node :return: Root Nodes :rtype: QuerySet

`leaves`(*node=None*)

Returns a QuerySet of all leaf Nodes (nodes with no children) in the Node model.

:param Node node: (optional) if specified, returns only the leaves for that node :return: Leaf Nodes :rtype: QuerySet

`islands`()

Returns a QuerySet of all Nodes with no parents or children (degree 0).

:return: Island Nodes :rtype: QuerySet

Methods returning a QuerySet of Edges

None

Methods returning a Boolean

None

Methods returning other values

None

Model Methods

Methods used for building/manipulating an instance

`add_child`(*child*)

Provided with a Node instance, attaches that instance as a child to the current Node instance.

:param Node child: The Node to be added as a child :return: The newly created Edge between self and child :rtype: Edge

`add_children`(*children*)

Provided with a QuerySet of Node instances, attaches those instances as children of the current Node instance.

:param QuerySet children: The Nodes to be added as children :return: The newly created Edges between self and children :rtype: list

`add_parent`(*parent*)

Provided with a Node instance, attaches that instance as a parent to the current Node instance.

:param Node parent: The Node to be added as a parent :return: The newly created Edge between self and parent :rtype: Edge

`add_parents`(*parents*)

Provided with a QuerySet of Node instances, attaches those instances as parents of the current Node instance.

:param QuerySet parents: The Nodes to be added as parents :return: The newly created Edges between self and parents :rtype: list

remove_child(*child*, *delete_node=False*)

Removes the edge connecting this node to child if a child Node instance is provided. Optionally deletes the child node as well.

:param Node child: The Node to be removed as a child :return: True if any Nodes were removed, otherwise False :rtype: bool

remove_children(*children*)

Provided with a QuerySet of Node instances, removes those instances as children of the current Node instance.

:param QuerySet children: The Nodes to be removed as children :return: True if any Nodes were removed, otherwise False :rtype: bool

remove_all_children(*delete_node=False*)

Removes all children of the current Node instance, optionally deleting self as well.

:param QuerySet children: The Nodes to be removed as children :return: True if any Nodes were removed, otherwise False :rtype: bool

remove_parent(*parent*, *delete_node=False*)

Removes the edge connecting this node to parent if a parent Node instance is provided. Optionally deletes the parent node as well.

:param Node parent: The Node to be removed as a parent :return: True if any Nodes were removed, otherwise False :rtype: bool

remove_parents(*parents*)

Provided with a QuerySet of Node instances, removes those instances as parents of the current Node instance.

:param QuerySet parents: The Nodes to be removed as parents :return: True if any Nodes were removed, otherwise False :rtype: bool

remove_all_parents(*delete_node=False*)

Removes all parents of the current Node instance, optionally deleting self as well.

:param QuerySet parents: The Nodes to be removed as parents :return: True if any Nodes were removed, otherwise False :rtype: bool

Methods returning a QuerySet of Nodes

ancestors()

Returns all Nodes in connected paths in a rootward direction.

:return: Nodes :rtype: QuerySet

self_and_ancestors()

Returns all Nodes in connected paths in a rootward direction, prepending self.

:return: Nodes :rtype: QuerySet

ancestors_and_self()

Returns all Nodes in connected paths in a rootward direction, appending self.

:return: Nodes :rtype: QuerySet

descendants()

Returns all Nodes in connected paths in a leafward direction.

:return: Nodes :rtype: QuerySet

self_and_descendants()

Returns all Nodes in connected paths in a leafward direction, prepending self.

:return: Nodes :rtype: QuerySet

descendants_and_self()

Returns all Nodes in connected paths in a leafward direction, appending self.

:return: Nodes :rtype: QuerySet

siblings()

Returns all Nodes that share a parent with this Node.

:return: Nodes :rtype: QuerySet

self_and_siblings()

Returns all Nodes that share a parent with this Node, prepending self.

:return: Nodes :rtype: QuerySet

siblings_and_self()

Returns all Nodes that share a parent with this Node, appending self.

:return: Nodes :rtype: QuerySet

partners()

Returns all Nodes that share a child with this Node.

:return: Nodes :rtype: QuerySet

self_and_partners()

Returns all Nodes that share a child with this Node, prepending self.

:return: Nodes :rtype: QuerySet

partners_and_self()

Returns all Nodes that share a child with this Node, appending self.

:return: Nodes :rtype: QuerySet

clan()

Returns a QuerySet with all ancestor Nodes, self, and all descendant Nodes.

:return: Nodes :rtype: QuerySet

connected_graph()

Returns all nodes connected in any way to the current Node instance.

:param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :return: Nodes :rtype: QuerySet

shortest_path(*target_node*)

Returns the shortest path from self to target Node. Resulting Queryset is sorted leafward, regardless of the relative position of starting and ending nodes.

:param Node target_node: The target Node for searching :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :return: Nodes :rtype: QuerySet

all_paths(*target_node*)

Returns all paths from self to target Node. Resulting Queryset is sorted leafward, regardless of the relative position of starting and ending nodes.

:param Node target_node: The target Node for searching :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :return: Nodes :rtype: QuerySet

roots()

Returns a QuerySet of all root Nodes, if any, for the current Node.

:return: Root Nodes :rtype: QuerySet

leaves()

Returns a QuerySet of all leaf Nodes, if any, for the current Node.

:return: Leaf Nodes :rtype: QuerySet

For future consideration:

- immediate_family (parents, self and children)
- piblings (aka: aunts/uncles)
- niblings (aka: nieces/nephews)
- cousins

Methods returning a QuerySet of Edges

ancestor_edges()

Ancestor Edge instances for the current Node.

:return: Ancestor Edges :rtype: QuerySet

descendant_edges()

Descendant Edge instances for the current Node.

:return: Descendant Edges :rtype: QuerySet

clan_edges()

Clan Edge instances for the current Node.

:return: Clan Edges :rtype: QuerySet

Methods returning a Boolean

is_root()

Returns True if the current Node instance has no parents (Node has an in-degree 0 and out-degree >= 0).

:rtype: bool

is_leaf()

Returns True if the current Node instance has no children (Node has an in-degree >=0 and out-degree 0).

:rtype: bool

is_island()

Returns True if the current Node instance has no parents or children (Node has degree 0).

:rtype: bool

path_exists_from(*target_node*, *directional=True*)

Checks whether there is a path from the target Node instance to the current Node instance.

:param Node target_node: The node to compare against :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :rtype: bool

path_exists_to(*target_node*, *directional=True*)

Checks whether there is a path from the current Node instance to the target Node instance.

:param Node target_node: The node to compare against :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :rtype: bool

is_ancestor_of(*target_node*, *directional=True*)

Checks whether the current Node instance is an ancestor of the provided target Node instance.

:param Node target_node: The node to compare against :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :rtype: bool

is_descendant_of(*target_node*, *directional=True*)

Checks whether the current Node instance is a descendant of the provided target Node instance.

:param Node target_node: The node to compare against :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :rtype: bool

is_sibling_of(*target_node*, *directional=True*)

Checks whether the current Node instance is a sibling of the provided target Node instance (see *terminology*).

:param Node target_node: The node to compare against :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :rtype: bool

is_partner_of(*target_node*, *directional=True*)

Checks whether the current Node instance is a partner of the provided target Node instance (see *terminology*).

:param Node target_node: The node to compare against :param Node directional: (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions :rtype: bool

Methods returning other values

ancestor_count()

Returns the total number of ancestor Nodes.

:rtype: int

descendant_count()

Returns the total number of descendant Nodes.

:rtype: int

clan_count()

Returns the total number of clan Nodes.

:rtype: int

sibling_count()

Returns the total number of sibling Nodes.

:rtype: int

partner_count()

Returns the total number of partner Nodes.

:rtype: int

connected_graph_node_count()

Returns the count of all ancestors Nodes, self, and all descendant Nodes.

:rtype: int

node_depth()

Returns the depth of this Node instance from furthest root Node.

:rtype: int

distance(*target_node*)

Returns the shortest hops count to the target Node.

:param Node target_node: The node to compare against :rtype: int

For future consideration:

- descendant_tree()
- ancestor_tree()

graphs()

A Node can be associated with multiple Graphs. This method returns a QuerySet of all Graph instances associated with the current Node.

:return: Graphs to which this Node belongs :rtype: QuerySet

2.4.11 Edge

Manager/QuerySet Methods

None

Methods used for building/manipulating

None

Methods returning a QuerySet of Nodes

None

Methods returning a QuerySet of Edges

ancestor_edges(*target_node*)

All Edge instances which are ancestors of the target Node.

:param Node target_node: The target Node for searching :return: Ancestor Edges :rtype: QuerySet

descendant_edges(*target_node*)

All Edge instances descended from the target Node.

:param Node target_node: The target Node for searching :return: Descendant Edges :rtype: QuerySet

clan_edges(*target_node*)

All Edge instances which are ancestors, self, and descendants of the target Node.

:param Node target_node: The target Node for searching :return: Clan Edges :rtype: QuerySet

shortest_path_edges(*node_from*, *node_to*)

All Edge instances for the shortest path from node_from to node_to.

:param Node node_from: The starting Node :param Node node_to: The ending Node :return: Shortest path Edges :rtype: QuerySet

all_path_edges(*node_from*, *node_to*)

All Edge instances for all paths from node_from to node_to.

:param Node node_from: The starting Node :param Node node_to: The ending Node :return: Edges :rtype: QuerySet

Methods returning a Boolean

path_is_valid()

Verify that the current QuerySet of Edges result in a contiguous path.

:rtype: bool

Methods returning other values

from_node_queryset(*nodes*)

Returns all Edge instances where a parent and child Node are within the provided QuerySet of Nodes.

:param QuerySet nodes: Nodes of interest :return: Edges with both parent and child Nodes in the provided QuerySet of Nodes :rtype: QuerySet

sorted()

Sorts the current Edge QuerySet in a rootward direction

:return: Sorted Edges :rtype: QuerySet

Model Methods

Methods used for building/manipulating an instance

add_edge(*from_node*, *to_node*)

Adds an edge between two Node instances.

:param Node node_from: The starting Node :param Node node_to: The ending Node :return: Newly created Edge :rtype: Edge

insert_node(*node*, *clone_to_rootside=False*, *clone_to_leafside=False*, *pre_save=None*, *post_save=None*)

Insert a Node into an existing Edge instance.

:param Node node: The Node to insert :param bool clone_to_rootside: (optional) Clone properties of the existing Edge to the new rootside Edge :param bool clone_to_leafside: (optional) Clone properties of the existing Edge to the new leafside Edge :param callable pre_save: (optional) Helper function to modify before saving :param callable post_save: (optional) Helper function to modify after saving :return: Newly created rootside Edge (parent to the inserted node) and leafside Edge (child to the inserted Node) :rtype: tuple

Process:

1. Add a new Edge from the parent Node of the current Edge instance to the provided Node instance, optionally cloning properties of the existing Edge.
2. Add a new Edge from the provided Node instance to the child Node of the current Edge instance, optionally cloning properties of the existing Edge.
3. Remove the original Edge instance.

The instance will still exist in memory, though not in database (https://docs.djangoproject.com/en/3.1/ref/models/instances/#refreshing-objects-from-database). Recommend running the following after conducting the deletion:

```
del instancename
```

Cloning will fail if a field has unique=True, so a pre_save function can be passed into this method. Likewise, a post_save function can be passed in to rebuild relationships. For instance, if you have a `name` field that is unique and generated automatically in the model's save() method, you could pass in a the following `pre_save` function to clear the name prior to saving the new Edge instance(s):

```
def pre_save(new_edge):
    new_edge.name = ""
    return new_edge
```

A more complete example, where we have models named DAGEdge & DAGNode, and we want to insert a new Node (n2) into Edge e1, while copying e1's field properties (except `name`) to the newly created rootside Edge instance (n1 to n2) is shown below.

```
Original        Final

n1  o           n1  o
    |                \
    |                 o n2
    |                /
n3  o           n3  o
```

```
from myapp.models import DAGEdge, DAGNode

n1 = DAGNode.objects.create(name="n1")
n2 = DAGNode.objects.create(name="n2")
n3 = DAGNode.objects.create(name="n3")

# Connect n3 to n1
n1.add_child(n3)

e1 = DAGEdge.objects.last()

# function to clear the `name` field, which is autogenerated and must be unique
def pre_save(new_edge):
    new_edge.name = ""
    return new_edge

DAGEdge.objects.insert_node(e1, n2, clone_to_rootside=True, pre_save=pre_save)
```

Methods returning a QuerySet of Nodes

None

Methods returning a QuerySet of Edges

None

Methods returning a Boolean

None

Methods returning other values

None

2.4.12 Reference

django_directed

Initialize module.

admin.py

Admin for the *django_directed* app.

apps.py

App configuration for the django_directed app.

class django_directed.apps.**DjangoDirectedConfig**(*app_name*, *app_module*)

config.py

context_managers.py

Context managers for the django_directed app.

django_directed.context_managers.**get_current_graph_instance**(*graph_fullname*)

Returns the graph if it exists in the local thread.

django_directed.context_managers.**graph_scope**(*graph*)

Context manager for graphs.

Used to set and cleanup Graph instance. If nested, saves outer context and resets it at conclusion of scope.

Parameters
graph (BaseGraph) –

fields.py

Custom model fields for Django Directed.

class django_directed.fields.**CurrentGraphFKField**(**args*, ***kwargs*)

A ForeignKey field that defaults to the current Graph instance.

deconstruct()

Deconstructs the field.

pre_save(*model_instance*, *add*)

Sets the value of the field on save.

forms.py

Forms for django_directed.

manager_methods.py

Manager methods for the django_directed app.

model_methods.py

Model methods for django_directed.

models/abstract_base_graph_models.py

Abstract Base Graph Models for Django Directed.

class django_directed.models.abstract_base_graph_models.**BaseEdge**(**args*, ***kwargs*)
Base Edge Model lets us verify that a given model instance derives from BaseEdge.

class django_directed.models.abstract_base_graph_models.**BaseGraph**(**args*, ***kwargs*)
Base Graph Model lets us verify that a given model instance derives from BaseGraph.

class django_directed.models.abstract_base_graph_models.**BaseNode**(**args*, ***kwargs*)
Base Node Model lets us verify that a given model instance derives from BaseNode.

django_directed.models.abstract_base_graph_models.**base_edge**(*config*)
Creates "Abstract Edge Model".

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_base_graph_models.**base_graph**(*config*)
Creates "Abstract Graph Model".

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_base_graph_models.**base_node**(*config*)
Creates "Abstract Node Model".

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_base_graph_models.**get_graph_aware_manager**(*config*)
Creates a manager that is aware of the current graph instance.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_base_graph_models.**get_graph_aware_queryset**(*config*)
Creates a queryset that is aware of the current graph instance.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_base_graph_models.**get_model_class**(*model_fullname*)
Provided with a model fullname (*app_name.ModelName*), returns the associated model class.

Parameters
model_fullname (*str*) –

Return type
Model

models/abstract_graph_models.py

Abstract models for Django Directed.

django_directed.models.abstract_graph_models.**arborescence_edge_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Edge base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**arborescence_graph_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Graph base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**arborescence_node_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Node base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**cyclic_edge_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Edge base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**cyclic_graph_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Graph base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**cyclic_node_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Node base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**dag_edge_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Edge base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**dag_graph_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Graph base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**dag_node_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Node base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**polytree_edge_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Edge base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**polytree_graph_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Graph base model are implemented.

Parameters
config (*GraphConfig*) –

django_directed.models.abstract_graph_models.**polytree_node_factory**(*config*)

Type: Subclassed Abstract Model. Abstract methods of the Node base model are implemented.

Parameters
config (*GraphConfig*) –

models/model_factory.py

Model factory for directed graph models.

class django_directed.models.model_factory.**ArborescenceService**(*config*)

Returns the actual Graph, Edge, and Node models.

Parameters
config (*GraphConfig*) –

edge()

Returns the actual Edge model.

graph()

Returns the actual Graph model.

node()

Returns the actual Node model.

class django_directed.models.model_factory.**CyclicService**(*config*)

Returns the actual Graph, Edge, and Node models.

Parameters
config (*GraphConfig*) –

edge()

Returns the actual Edge model.

graph()

Returns the actual Graph model.

node()

Returns the actual Node model.

class django_directed.models.model_factory.**DAGService**(*config*)

Returns the actual Graph, Edge, and Node models.

Parameters
config (*GraphConfig*) –

edge()

Returns the actual Edge model.

graph()

Returns the actual Graph model.

node()

Returns the actual Node model.

class django_directed.models.model_factory.**DirectedServiceFactory**

Registers django-directed services.

get(*config*, ***kwargs*)

Creates and returns a new model factory for directed graph models.

Parameters
config (*GraphConfig*) –

register(*key*, *builder*)

Registers model factory services.

services_enum()

Return enum of registered services.

services_list()

Return list of registered services.

class django_directed.models.model_factory.**PolytreeService**(*config*)

Returns the actual Graph, Edge, and Node models.

Parameters
config (*GraphConfig*) –

edge()

Returns the actual Edge model.

graph()

Returns the actual Graph model.

node()

Returns the actual Node model.

django_directed.models.model_factory.**create_arborescence_service**(*config*)

Creates a new ArborescenceService instance.

Parameters
config (*GraphConfig*) –

django_directed.models.model_factory.**create_cyclic_service**(*config*)

Creates a new CyclicService instance.

Parameters
config (*GraphConfig*) –

django_directed.models.model_factory.**create_dag_service**(*config*)

Creates a new DAGService instance.

Parameters
config (*GraphConfig*) –

django_directed.models.model_factory.**create_polytree_service**(*config*)

Creates a new PolytreeService instance.

Parameters
config (*GraphConfig*) –

query_utils.py

Functions for transforming RawQuerySet or other outputs of django-directed to alternate formats.

django_directed.query_utils.**check_field_list**(*obj*)

Verifies that obj is a list of strings.

Used with model_to_dict to ensure that the field_list argument is valid.

django_directed.query_utils.**edges_from_nodes_queryset**(*nodes_queryset*)

Given an Edge Model and a QuerySet or RawQuerySet of nodes, returns a queryset of the associated edges.

django_directed.query_utils.**get_field_value**(*instance*, *field*, *date_strf=None*)

Extracts the value of a field from a model instance.

Used with model_to_dict to extract the value of a field from a model instance.

django_directed.query_utils.**get_instance_characteristics**(*instance*)

Returns a tuple of the node & edge model classes and the instance_type for the provided instance.

django_directed.query_utils.**get_queryset_characteristics**(*queryset*)

Returns a tuple of the node & edge model classes and the queryset type for the provided queryset.

django_directed.query_utils.**model_to_dict**(*instance*, *field_list*, *date_strf=None*)

Returns a dictionary of {field_name: field_value} for a given model instance.

e.g.: model_to_dict(myqueryset.first(), fields=["id",]) For DateTimeFields, a formatting string can be provided Adapted from: https://ziwon.github.io/post/using_custom_model_to_dict_in_django/

django_directed.query_utils.**nodes_from_edges_queryset**(*edges_queryset*)

Given a Node Model and a QuerySet or RawQuerySet of edges, returns a queryset of the associated nodes.

queryset_methods.py

QuerySet methods for the django_directed app.

signals.py

Signals for django_directed.

validators.py

Validators for the django_directed app.

views.py

Views for the django_directed app.

urls.py

Views for the django_directed app.

2.4.13 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

How to request a feature

Request features on the Issue Tracker.

How to set up your development environment

You need Python 3.9+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run django-directed
```

How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the *tests* directory, and are written using the pytest testing framework.

How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

2.4.14 Extending Functionality

Beyond making modifications directly within your project (e.g. inheriting and extending the provided models & managers), there are two ways of extending django-directed for use in additional projects or for community use.

Custom model factories

You can create new django-directed graph types with your own graph factories, which can be used directly within your project or in an installable django package for reuse.

Create a new factory

Start by creating new factory functions for the graph, edge, and node. Like any other graph in django-directed, the GraphConfig object is passed into each function, and is used for customizing functionality of the returned model classes.

```
from django.db import models

from django_directed.components import AbstractGraph, AbstractEdge, AbstractNode


def new_type_graph_factory(config):
    """
    Type: Subclassed Abstract Model
    Abstract methods of the Graph base model are implemented.
    """

    class NewTypeGraph(AbstractGraph):
        some_graph_field = models.IntegerField()

        class Meta:
            abstract = True

    return NewTypeGraph()


def new_type_edge_factory(config):
    """
    Type: Subclassed Abstract Model
    Abstract methods of the Edge base model are implemented.
    """

    class NewTypeEdge(AbstractEdge):
        some_edge_field = models.IntegerField()

        class Meta:
            abstract = True

    return NewTypeEdge()


def new_type_node_factory(config):
    """
```

(continues on next page)

(continued from previous page)

```
    Type: Subclassed Abstract Model
    Abstract methods of the Node base model are implemented.
    """

    class NewTypeNode(AbstractNode):
        some_node_field = models.IntegerField()

        class Meta:
            abstract = True

    return NewTypeNode()
```

Create the service

The service makes it possible to register the new factory within django-directed.

```
class NewTypeService:
    """Returns the actual Graph, Edge, and Node models"""

    def __init__(self, config):
        self._instance = None
        self._config = config

    def graph(self):
        return new_type_graph_factory(config=self._config)

    def edge(self):
        return new_type_edge_factory(config=self._config)

    def node(self):
        return new_type_node_factory(config=self._config)


def create_new_type_service(config):
    return NewTypeService(config)
```

Register your new graph service

Now that the factory and service for our new graph type has been built, we can register the service in our django project and make use of the resulting models.

```
factory.register("NEW_TYPE", create_new_type_service)
```

As usual, within your app's `models.py`, instantiate the actual model instances.

```
# Create NewType factory instance
new_type = factory.create("NEW_TYPE", config=my_custom_config)

# Create model instances
MyNewTypeGraph = new_type.graph()
```

(continues on next page)

(continued from previous page)

```
MyNewTypeEdge = new_type.edge()
MyNewTypeNode = new_type.node()
```

Pluggy Plugins

Throughout django-directed, pluggy hooks have been added to

Combined approach

2.4.15 Plugin Hooks

django-directed plugins use pluggy plugin hooks to customize behavior.

Each plugin can implement one or more hooks using the @hookimpl decorator against a function matching one of the hooks documented on this page.

When you implement a plugin hook, your implementation can accept any or all of the parameters that are documented below as parameters for that hook.

Work In Progress

2.4.16 Signals

2.4.17 About django-directed

This page is not a necessary read for working with the graphs in django-directed, but gives context about the goals and direction of the project, resources for further reading, etc.

Background

This project is the successor of another django package of mine, django-postgresql-dag, which itself was forked and heavily modified from django-dag and django-dag-postgresql.

When I started building django-postgresql-dag, I was rather new to a lot of concepts in both graph theory and database queries. As a result, I felt that I backed myself into corners in some ways with that earlier package. I developed django-postgresql-dag to serve as the underlying structure of an application that modeled real-world infrastructure as a directed acyclic graph, but I soon found that there were other graph-related things I wanted to be able to do that were not DAG-specific. Additionally, using CTE's in django has been somewhat democratized with the django-cte package and other changes over the years, and it might be feasible to port at least a portion of the graph functionality to database backends other than Postgres (though this is not a focus of the initial iteration of the project).

Some design decisions

- **A reasonable amount of flexibility** - The predecessor for this package was limited (in name and in some implementation aspects) solely to working with Directed Acyclic Graphs in Postgresql. I often find, though, that I need other types of directed graphs. This package should still do one thing well - working with directed graphs - but I've opened the scope a bit.
- **DRY** - There are a lot of commonalities between all types of directed graphs, so we should be able to model graphs of different types with a common API, extending when necessary to perform specialized tasks that do not apply to all graphs.
- **Prioritize querying over writing** - For my typical purposes, quickly adding large graphs to the database is an uncommon task. Instead, in most graph applications I am either slowly adding a node here and there (comments, categories, etc), or I am adding large graphs in an asynchronous manner (uploading and building the graph of an entire physical infrastructure model from a CSV file). In either case, the speed at which the graph is written is of much less consequence than the ability to query the resulting graph quickly.
- **Include tools for modifying and reconfiguring graphs** - move or copy subgraphs, insert and delete nodes, and pre-processing (calculating graph hashes or copying a subgraph with a function applied), etc.
- **Optimize for sparse graphs** - Most of the graph structures I find myself building and working with are sparse. There are generally few connections from each node to another. Said another way, the typical degree of the nodes is small (often no more than 5 or so). This seems pretty common for many real-world models such as physical infrastructure, as well as many common web & software related graph uses such as threaded comments, automation processes, and version control systems. If you are trying to model large, highly-connected graphs, this might not be the right package for you.

Scope & Goals

Directed graphs in general can solve or model an incredible number of real-world or web-related problems and concepts. This package should be complete enough to perform a majority of tasks needed for working with an assortment of directed graphs in django applications, but it should also be flexible and extensible enough to allow for customization and novel approaches to problems in practical graph application.

Types of Directed Graphs

The scope of this package includes working with a variety of directed graphs. This includes eventually supporting functionality for each of these types of directed graphs:

- Directed graphs aka DiGraphs
  - Directed cyclic graph
  - Directed acyclic graph (DAG)
    - Polytree (aka directed tree, oriented tree, or singly connected network) - DAGs whose underlying undirected graph is a tree
      - Arborescence (or out-tree or rooted tree) (single-rooted polytree)

Other types of graphs to consider supporting (in expected order of complexity):

- **Subclasses of Arborescence**
  - Directed binary tree
  - Directed quadtree
  - Directed octree

- **Binary Search Trees (BST)**
- **Multigraph** - Graphs where the same pair of nodes may be connected by more than one edge.
  - This might be further constrained in a cyclic graph to limit edges between two nodes to no more than two, with one edge in each direction.
- **Hypergraph** - Graphs where edges can join more than just two nodes.

For further details on *building, querying, manipulating, and exporting* graphs, please Read the Docs

Example Use-Cases of django-directed

Graphs can be used to model an incredibly large range of ideas, physical systems, concepts, web-components, etc. Here is a very incomplete list of some of the ways you might use django-directed, along with the underlying structure that might be best to represent them.

Use-Cases	Potential Data Structure
Threaded discussion comments	Arborescence
Social follows" (which users are following which)"	Directed cyclic graph
Model of resource flow in gas/electrical/water/sewer distribution systems	Arborescence
The underlying structure to business process automation (e.g. tools like Airflow)	Directed cyclic graph or DAG
Hierarchical bill of materials for a product	Polytree or Arborescence
Network mapping (Internet device map, map of linked pages in a website, modeling roadways, modeling airline/train paths, etc)	Directed cyclic graph
Modeling dependencies in software applications	DAG
Scheduling tasks for project management	Directed cyclic graph or DAG
Fault-tree analysis in industrial systems	Polytree
Version control systems	DAG
Which academic papers are cited by later papers	DAG
Dependencies in educational plans (which pieces of knowledge or classes must preceed others as a student progresses toward a goal?)	Arborescence
Modeling supply chains from initial resource (mining, forestry, etc) to manufacturer to retailer to consumer market	DAG or Polytree
Family trees and other genealogical models	DAG
Hierarchical file/folder structures	Arborescence
Mind maps	DAG
TRIE structures	Arborescence
Customer journey maps	DAG
Storing information about phone calls, emails, or other interactions between people	Directed cyclic graph or DAG

Essentially, just about anything involving causal relationships, hierarchies, or dependencies can be modeled with a directed graph. This package may be useful if you need to persist that information for use with django applications.

Further reading and resources

These resources are fantastic for learning about working with graphs in databases and related topics. They are listed in no particular order, and I do not have any affiliation with the authors, publishers, or bookstores.

Books

- Joe Celko's trees and hierarchies in SQL for smarties [B&N, Amazon]
- Effective SQL: 61 Specific Ways to Write Better SQL (Chapter 10) [B&N, Amazon]
- Algorithms for Decision Making (not yet released for print, but available to read at the book's website) [MIT Press]

Blog posts, slide shows, and articles

- A Model to Represent Directed Acyclic Graphs (DAG) on SQL Databases
- Graph Algorithms in a Database: Recursive CTEs and Topological Sort with Postgres
- Postgres: A Graph Database (by Greg Spiegelberg at Pivotal)

2.4.18 Project Roadmap

Work In Progress

Long-term Features

- [ ] Admin functionality
  - [ ] Visualize graphs
  - [ ] Edit graphs
    - [ ] Delete edge
    - [ ] Delete node
    - [ ] Add node
    - [ ] Add edge
    - [ ] Copy graph/subgraph
    - [ ] Move subgraph

2.4.19 Credits

Development Lead

- Jack Linke jack@watervize.com

Contributors

None yet. Why not be the first?

2.4.20 Contributor Covenant Code of Conduct

Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at jacklinke@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

2.4.21 License

```
MIT License

Copyright © 2023 Jack Linke

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

View this project on Github.

PYTHON MODULE INDEX

d

INDEX

C

D

E

F

G

H

I

L

M

N

P

R

S