django-directed Documentation

Release 0.1.3

Jack Linke

Mar 22, 2022

USER GUIDE

1 django-directed **1**
- 1.1 Fundamentals . . . 1
- 1.2 Quickstart . . . 1
- 1.3 Example apps . . . 7
- 1.4 Why not use a graph database instead? . . . 7

2 Terminology and Definitions **9**
- 2.1 Node . . . 9
- 2.2 Edge . . . 9
- 2.3 Root . . . 9
- 2.4 Roots . . . 9
- 2.5 Leaf / Leaves . . . 10
- 2.6 Orphan . . . 10
- 2.7 Parent / Parents . . . 10
- 2.8 Child / Children . . . 10
- 2.9 Ancestors . . . 10
- 2.10 Descendants . . . 10
- 2.11 Clan . . . 11
- 2.12 Siblings . . . 11
- 2.13 Partners . . . 11
- 2.14 Distance . . . 11
- 2.15 Node Depth . . . 11

3 Concepts **13**

4 Installation **15**
- 4.1 Basic . . . 15
- 4.2 External packages and plugins . . . 15

5 Building Graphs **17**
- 5.1 Configuration . . . 17
- 5.2 Models . . . 17

6 Querying Graphs **19**

7 Manipulating Graphs **21**

8 Exporting Graphs **23**

9 Graph **25**
- 9.1 Manager/QuerySet Methods . . . 25

9.2 Model Methods . . . 26

10 Node **29**
10.1 Manager/QuerySet Methods . . . 29
10.2 Model Methods . . . 30

11 Edge **37**
11.1 Manager/QuerySet Methods . . . 37
11.2 Model Methods . . . 38

12 Extending Functionality **41**
12.1 Custom model factories . . . 41
12.2 Pluggy Plugins . . . 43
12.3 Combined approach . . . 43

13 Plugin Hooks **45**

14 Contributing **47**
14.1 Getting up-and-running . . . 47
14.2 Package Structure . . . 49
14.3 Build the docs . . . 50

15 About django-directed **51**
15.1 Background . . . 51
15.2 Some design decisions . . . 51
15.3 Scope & Goals . . . 52
15.4 Types of Directed Graphs . . . 52
15.5 Example Use-Cases of django-directed . . . 52
15.6 Further reading and resources . . . 53

16 Project Roadmap **55**
16.1 Long-term Features . . . 55

17 Credits **57**
17.1 Development Lead . . . 57
17.2 Contributors . . . 57

18 History **59**
18.1 0.1.0 (2022-02-08) . . . 59
18.2 0.1.1 (2022-02-17) . . . 59
18.3 0.1.2 (2022-02-21) . . . 59

Index **61**

CHAPTER ONE

DJANGO-DIRECTED

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.

1.1 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.

1.2 Quickstart

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

1.2.1 Install django-directed

```
pip install django-directed
```

1.2.2 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()):
    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.

1.2.3 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
```

1.2.4 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)
    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)
```

(continues on next page)

(continued from previous page)

```
    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)
```

1.2.5 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
```

1.2.6 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 ReadThe-Docs. 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;
```

(continues on next page)

(continued from previous page)

```
    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

1.2.7 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:

```python
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.NodeNotReachableException`

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.

```python
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.

1.3 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.

1.4 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.

CHAPTER

TWO

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.

2.1 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.

2.2 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).

2.3 Root

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

2.4 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.

2.5 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.

2.6 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.)

2.7 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`.

2.8 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`.

2.9 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`.

2.10 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`.

2.11 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`.

2.12 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.

2.13 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.

2.14 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.

2.15 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.

CHAPTER

THREE

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.

CHAPTER FOUR

INSTALLATION

4.1 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.

4.2 External packages and plugins

CHAPTER

FIVE

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.

5.1 Configuration

5.2 Models

5.2.1 Model Instantiation

5.2.2 Model Migrations

CHAPTER

SIX

QUERYING GRAPHS

Work In Progress

CHAPTER

SEVEN

MANIPULATING GRAPHS

Work In Progress

CHAPTER EIGHT

EXPORTING GRAPHS

Work In Progress

CHAPTER

NINE

GRAPH

WORK IN PROGRESS

9.1 Manager/QuerySet Methods

For future consideration:

- clone()

9.1.1 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

9.1.2 Methods returning a QuerySet of Nodes

None

9.1.3 Methods returning a QuerySet of Edges

None

9.1.4 Methods returning a Boolean

None

9.1.5 Methods returning other values

node_count()

Returns Number of Nodes in the Graph

Return type int

edge_count()

Returns Number of Edges in the Graph

Return type int

graph_hash()

Returns Hash value for the Graph

Return type TBD

9.2 Model Methods

9.2.1 Methods used for building/manipulating an instance

None

9.2.2 Methods returning a QuerySet of Nodes

None

9.2.3 Methods returning a QuerySet of Edges

None

9.2.4 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.

Parameters

- **node_from** (*Node*) – The starting Node
- **node_to** (*Node*) – The ending Node

Returns True if path exists from `node_from` to `node_to`

Return type bool

For future consideration:

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

9.2.5 Methods returning other values

None

CHAPTER

TEN

NODE

10.1 Manager/QuerySet Methods

10.1.1 Methods used for building/manipulating

None

10.1.2 Methods returning a QuerySet of Nodes

`roots`(*node=None*)

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

Parameters **`node`** (*`Node`*) – (optional) if specified, returns only the roots for that node

Returns Root Nodes

Return type QuerySet

`leaves`(*node=None*)

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

Parameters **`node`** (*`Node`*) – (optional) if specified, returns only the leaves for that node

Returns Leaf Nodes

Return type QuerySet

`islands`()

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

Returns Island Nodes

Return type QuerySet

10.1.3 Methods returning a QuerySet of Edges

None

10.1.4 Methods returning a Boolean

None

10.1.5 Methods returning other values

None

10.2 Model Methods

10.2.1 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.

Parameters **child** (*Node*) – The Node to be added as a child

Returns The newly created Edge between self and child

Return type Edge

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

Parameters **children** (*QuerySet*) – The Nodes to be added as children

Returns The newly created Edges between self and children

Return type list

add_parent(*parent*)
Provided with a Node instance, attaches that instance as a parent to the current Node instance.

Parameters **parent** (*Node*) – The Node to be added as a parent

Returns The newly created Edge between self and parent

Return type Edge

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

Parameters **parents** (*QuerySet*) – The Nodes to be added as parents

Returns The newly created Edges between self and parents

Return type 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.

Parameters **child** (*Node*) – The Node to be removed as a child

Returns True if any Nodes were removed, otherwise False

Return type bool

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

Parameters **children** (*QuerySet*) – The Nodes to be removed as children

Returns True if any Nodes were removed, otherwise False

Return type bool

remove_all_children(*delete_node=False*)

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

Parameters **children** (*QuerySet*) – The Nodes to be removed as children

Returns True if any Nodes were removed, otherwise False

Return type 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.

Parameters **parent** (*Node*) – The Node to be removed as a parent

Returns True if any Nodes were removed, otherwise False

Return type bool

remove_parents(*parents*)

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

Parameters **parents** (*QuerySet*) – The Nodes to be removed as parents

Returns True if any Nodes were removed, otherwise False

Return type bool

remove_all_parents(*delete_node=False*)

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

Parameters **parents** (*QuerySet*) – The Nodes to be removed as parents

Returns True if any Nodes were removed, otherwise False

Return type bool

10.2.2 Methods returning a QuerySet of Nodes

ancestors()

Returns all Nodes in connected paths in a rootward direction.

Returns Nodes

Return type QuerySet

self_and_ancestors()

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

Returns Nodes

Return type QuerySet

ancestors_and_self()

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

Returns Nodes

Return type QuerySet

`descendants()`
Returns all Nodes in connected paths in a leafward direction.

Returns Nodes

Return type QuerySet

`self_and_descendants()`
Returns all Nodes in connected paths in a leafward direction, prepending self.

Returns Nodes

Return type QuerySet

`descendants_and_self()`
Returns all Nodes in connected paths in a leafward direction, appending self.

Returns Nodes

Return type QuerySet

`siblings()`
Returns all Nodes that share a parent with this Node.

Returns Nodes

Return type QuerySet

`self_and_siblings()`
Returns all Nodes that share a parent with this Node, prepending self.

Returns Nodes

Return type QuerySet

`siblings_and_self()`
Returns all Nodes that share a parent with this Node, appending self.

Returns Nodes

Return type QuerySet

`partners()`
Returns all Nodes that share a child with this Node.

Returns Nodes

Return type QuerySet

`self_and_partners()`
Returns all Nodes that share a child with this Node, prepending self.

Returns Nodes

Return type QuerySet

`partners_and_self()`
Returns all Nodes that share a child with this Node, appending self.

Returns Nodes

Return type QuerySet

`clan()`
Returns a QuerySet with all ancestor Nodes, self, and all descendant Nodes.

Returns Nodes

Return type QuerySet

`connected_graph()`

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

Parameters **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Returns Nodes

Return type 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.

Parameters

- **target_node** (*Node*) – The target Node for searching
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Returns Nodes

Return type 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.

Parameters

- **target_node** (*Node*) – The target Node for searching
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Returns Nodes

Return type QuerySet

`roots()`

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

Returns Root Nodes

Return type QuerySet

`leaves()`

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

Returns Leaf Nodes

Return type QuerySet

For future consideration:

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

10.2.3 Methods returning a QuerySet of Edges

`ancestor_edges()`
Ancestor Edge instances for the current Node.

Returns Ancestor Edges

Return type QuerySet

`descendant_edges()`
Descendant Edge instances for the current Node.

Returns Descendant Edges

Return type QuerySet

`clan_edges()`
Clan Edge instances for the current Node.

Returns Clan Edges

Return type QuerySet

10.2.4 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).

Return type bool

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

Return type bool

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

Return type bool

`path_exists_from`(*target_node*, *directional=True*)
Checks whether there is a path from the target Node instance to the current Node instance.

Parameters

- **target_node** (*Node*) – The node to compare against
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Return type bool

`path_exists_to`(*target_node*, *directional=True*)
Checks whether there is a path from the current Node instance to the target Node instance.

Parameters

- **target_node** (*Node*) – The node to compare against
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Return type bool

`is_ancestor_of`(*target_node*, *directional=True*)
Checks whether the current Node instance is an ancestor of the provided target Node instance.

Parameters

- **target_node** (*Node*) – The node to compare against
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Return type bool

`is_descendant_of`(*target_node*, *directional=True*)
Checks whether the current Node instance is a descendant of the provided target Node instance.

Parameters

- **target_node** (*Node*) – The node to compare against
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Return type 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*).

Parameters

- **target_node** (*Node*) – The node to compare against
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Return type 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*).

Parameters

- **target_node** (*Node*) – The node to compare against
- **directional** (*Node*) – (optional) if True, path searching operates normally (in leafward direction), if False search operates in both directions

Return type bool

10.2.5 Methods returning other values

`ancestor_count`()
Returns the total number of ancestor Nodes.

Return type int

`descendant_count`()
Returns the total number of descendant Nodes.

Return type int

`clan_count`()
Returns the total number of clan Nodes.

Return type int

`sibling_count()`
Returns the total number of sibling Nodes.

Return type int

`partner_count()`
Returns the total number of partner Nodes.

Return type int

`connected_graph_node_count()`
Returns the count of all ancestors Nodes, self, and all descendant Nodes.

Return type int

`node_depth()`
Returns the depth of this Node instance from furthest root Node.

Return type int

`distance`(*target_node*)
Returns the shortest hops count to the target Node.

Parameters `target_node` (*Node*) – The node to compare against

Return type 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.

Returns Graphs to which this Node belongs

Return type QuerySet

CHAPTER ELEVEN

EDGE

11.1 Manager/QuerySet Methods

None

11.1.1 Methods used for building/manipulating

None

11.1.2 Methods returning a QuerySet of Nodes

None

11.1.3 Methods returning a QuerySet of Edges

`ancestor_edges`(*target_node*)

All Edge instances which are ancestors of the target Node.

Parameters **`target_node`** (`Node`) – The target Node for searching

Returns Ancestor Edges

Return type QuerySet

`descendant_edges`(*target_node*)

All Edge instances descended from the target Node.

Parameters **`target_node`** (`Node`) – The target Node for searching

Returns Descendant Edges

Return type QuerySet

`clan_edges`(*target_node*)

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

Parameters **`target_node`** (`Node`) – The target Node for searching

Returns Clan Edges

Return type QuerySet

`shortest_path_edges`(*node_from*, *node_to*)

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

Parameters

- **node_from** (*Node*) – The starting Node
- **node_to** (*Node*) – The ending Node

Returns Shortest path Edges

Return type QuerySet

all_path_edges(*node_from*, *node_to*)

All Edge instances for all paths from node_from to node_to.

Parameters

- **node_from** (*Node*) – The starting Node
- **node_to** (*Node*) – The ending Node

Returns Edges

Return type QuerySet

11.1.4 Methods returning a Boolean

path_is_valid()

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

Return type bool

11.1.5 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.

Parameters **nodes** (*QuerySet*) – Nodes of interest

Returns Edges with both parent and child Nodes in the provided QuerySet of Nodes

Return type QuerySet

sorted()

Sorts the current Edge QuerySet in a rootward direction

Returns Sorted Edges

Return type QuerySet

11.2 Model Methods

11.2.1 Methods used for building/manipulating an instance

add_edge(*from_node*, *to_node*)

Adds an edge between two Node instances.

Parameters

- **node_from** (*Node*) – The starting Node
- **node_to** (*Node*) – The ending Node

Returns Newly created Edge

Return type 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.

Parameters

- **node** (*Node*) – The Node to insert
- **clone_to_rootside** (*bool*) – (optional) Clone properties of the existing Edge to the new rootside Edge
- **clone_to_leafside** (*bool*) – (optional) Clone properties of the existing Edge to the new leafside Edge
- **pre_save** (*callable*) – (optional) Helper function to modify before saving
- **post_save** (*callable*) – (optional) Helper function to modify after saving

Returns Newly created rootside Edge (parent to the inserted node) and leafside Edge (child to the inserted Node)

Return type 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
```

```python
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)
```

11.2.2 Methods returning a QuerySet of Nodes

None

11.2.3 Methods returning a QuerySet of Edges

None

11.2.4 Methods returning a Boolean

None

11.2.5 Methods returning other values

None

CHAPTER TWELVE

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.

12.1 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.

12.1.1 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.
    """
```

(continues on next page)

(continued from previous page)

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

        class Meta:
            abstract = True

    return NewTypeEdge()


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

12.1.2 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)
```

12.1.3 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()
MyNewTypeEdge = new_type.edge()
MyNewTypeNode = new_type.node()
```

12.2 Pluggy Plugins

Throughout django-directed, pluggy hooks have been added to

12.3 Combined approach

CHAPTER

THIRTEEN

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

CHAPTER

FOURTEEN

CONTRIBUTING

We welcome contributions that meet the goals and standards of this project. Contributions may include bug fixes, feature development, corrections or additional context for the documentation, submission of Issues on GitHub, etc.

For development and testing, you can run your own instance of Postgres (either locally or using a DBaaS), or you can use the provided Docker Compose yaml file to provision a containerized instance and data volume locally.

14.1 Getting up-and-running

14.1.1 Using Your Own postgres Instance

To develop using your own Postgres instance, you may set the following environmental variables on your machine:

- DB_NAME (defaults to "postgres")
- DB_NAME_WITH_EXTENSIONS (defaults to "postgres_with_extensions")
- DB_USER (defaults to "docker")
- DB_PASSWORD (defaults to "docker")
- DB_HOST (defaults to "localhost")
- DB_PORT (defaults to "9932")

The process of setting environmental variables varies between different operating systems. Generally, on macOS and Linux, you can use the following convention in the console:

```
export KEY=value
```

14.1.2 Using the Provided Docker Compose Postgres Instance

This guide assumes you already have Docker and Docker Compose installed.

Build & Bring up the Docker Compose container for Postgres services:

Run the following command to build and bring up the postgres service.

```
docker-compose -f dev.yml up -d --no-deps --force-recreate --build postgres
```

These are the database connection details:

```
DB = postgres *and* postgres_with_extensions
USER = docker
PASSWORD = docker
HOST = postgres
PORT = 9932
```

Note that the docker-compose postgres service creates two databases:

- The default `postgres` database with default functionality. In Django tests, this is the `default` database.
- A database named `postgres_with_extensions`, which has the plpython3u extension installed. In Django tests, this is the `with_extensions` database.

(The PL/Python procedural language is used in tests of advanced backend functionality which can make use of python-based graph libraries)

To check the status of the database container:

```
docker ps
```

Once running, you should be able to connect using the test app, psql, or other Postgres management tools if desired.

To completely remove the container and associated data:

```
docker-compose -f dev.yml down --rmi all --remove-orphans -v
```

14.1.3 Once you have a Running Postgres Instance

Create a Python virtual environment:

```
python3 -m venv venv
```

Activate the virtual environment for local development:

```
source venv/bin/activate
```

Install packages for development and testing:

This installs all packages needed for development and testing, and installs django-directed in editable mode from the local repo.

```
pip install -r ./tests/requirements.txt
```

Install pre-commits:

These ensure code is formatted correctly upon commit. See the pre-commit docs for more information.

```
pre-commit install
```

Run the tests:

```
pytest
```

Run code coverage report:

```
coverage run -m pytest
```

Create html coverage report:

```
coverage html
```

Check the django test project:

```
python manage.py check
```

14.2 Package Structure

Because django-directed supports multiple types of directed graphs, the underlying functionality can be a bit complex to understand at first. This shouldn't matter much if you are only building, querying, or manipulating graphs.

If you want to contribute to development of this package or extend the functionality of django-directed by creating installable apps and plugins for new graph types or customized graph behavior, it is critical to understand what functionality exists where.

Folder/File	
django_directed	
models	
__init__.py	
abstract_base_models.py	Lowest-level models (used for validating model type inheritance)
abstract_base_graph_models.py	Provides functionality common to all graphs
abstract_graph_models.py	Breaks out functionality specific to each graph type
model_factory.py	Provides the means of creating multiple graph moseld for each graph type
static	Package Static files
templates	Package Default Templates
templatetags	Package Templatetags
__init__.py	
admin.py	Tools for working with graphs in Django admin
apps.py	
context_managers.py	Utility context manager to simplify reference to a particular graph instance
exceptions.py	Exceptions specific to this package
fields.py	CurrentGraphField which simplifies working with Edge and Node models
manager_methods.py	WIP
model_methods.py	WIP
query_utils.py	Uilities for building and manipulating queries
queryset_methods.py	WIP
urls.py	
validators.py	WIP
views.py	

14.3 Build the docs

Within the docs directory, run this from the console:

```
make html
```

CHAPTER

FIFTEEN

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.

15.1 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).

15.2 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.

15.3 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.

15.4 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

15.5 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.

15.6 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.

15.6.1 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]

15.6.2 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)

CHAPTER

SIXTEEN

PROJECT ROADMAP

Work In Progress

16.1 Long-term Features

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

CHAPTER

SEVENTEEN

CREDITS

17.1 Development Lead

- Jack Linke jack@watervize.com

17.2 Contributors

None yet. Why not be the first?

CHAPTER

EIGHTEEN

HISTORY

18.1 0.1.0 (2022-02-08)

- Built initial readme entry to start documenting project goals.

18.2 0.1.1 (2022-02-17)

- Continue work on boilerplate to flesh out the foundations of the project.

18.3 0.1.2 (2022-02-21)

- Add terminology documentation.

View this project on Github.

INDEX

A

add_child()
    built-in function, 30
add_children()
    built-in function, 30
add_edge()
    built-in function, 38
add_parent()
    built-in function, 30
add_parents()
    built-in function, 30
all_path_edges()
    built-in function, 38
all_paths()
    built-in function, 33
ancestor_count()
    built-in function, 35
ancestor_edges()
    built-in function, 34, 37
ancestors()
    built-in function, 31
ancestors_and_self()
    built-in function, 31

B

built-in function
    add_child(), 30
    add_children(), 30
    add_edge(), 38
    add_parent(), 30
    add_parents(), 30
    all_path_edges(), 38
    all_paths(), 33
    ancestor_count(), 35
    ancestor_edges(), 34, 37
    ancestors(), 31
    ancestors_and_self(), 31
    clan(), 32
    clan_count(), 35
    clan_edges(), 34, 37
    connected_graph(), 33
    connected_graph_node_count(), 36
    descendant_count(), 35
    descendant_edges(), 34, 37
    descendants(), 31
    descendants_and_self(), 32
    distance(), 36
    edge_count(), 26
    from_node_queryset(), 38
    graph_hash(), 26
    graphs(), 36
    has_connection(), 26
    insert_node(), 39
    is_ancestor_of(), 34
    is_descendant_of(), 35
    is_island(), 34
    is_leaf(), 34
    is_partner_of(), 35
    is_root(), 34
    is_sibling_of(), 35
    islands(), 29
    leaves(), 29, 33
    node_count(), 26
    node_depth(), 36
    partner_count(), 36
    partners(), 32
    partners_and_self(), 32
    path_exists_from(), 34
    path_exists_to(), 34
    path_is_valid(), 38
    remove_all_children(), 31
    remove_all_parents(), 31
    remove_child(), 30
    remove_children(), 30
    remove_parent(), 31
    remove_parents(), 31
    roots(), 29, 33
    self_and_ancestors(), 31
    self_and_descendants(), 32
    self_and_partners(), 32
    self_and_siblings(), 32
    shortest_path(), 33
    shortest_path_edges(), 37
    sibling_count(), 35

siblings(), 32
siblings_and_self(), 32
sorted(), 38

C

clan()
built-in function, 32
clan_count()
built-in function, 35
clan_edges()
built-in function, 34, 37
connected_graph()
built-in function, 33
connected_graph_node_count()
built-in function, 36

D

descendant_count()
built-in function, 35
descendant_edges()
built-in function, 34, 37
descendants()
built-in function, 31
descendants_and_self()
built-in function, 32
distance()
built-in function, 36

E

edge_count()
built-in function, 26

F

from_node_queryset()
built-in function, 38

G

graph_hash()
built-in function, 26
graphs()
built-in function, 36

H

has_connection()
built-in function, 26

I

insert_node()
built-in function, 39
is_ancestor_of()
built-in function, 34
is_descendant_of()
built-in function, 35
is_island()
built-in function, 34
is_leaf()
built-in function, 34
is_partner_of()
built-in function, 35
is_root()
built-in function, 34
is_sibling_of()
built-in function, 35
islands()
built-in function, 29

L

leaves()
built-in function, 29, 33

N

node_count()
built-in function, 26
node_depth()
built-in function, 36

P

partner_count()
built-in function, 36
partners()
built-in function, 32
partners_and_self()
built-in function, 32
path_exists_from()
built-in function, 34
path_exists_to()
built-in function, 34
path_is_valid()
built-in function, 38

R

remove_all_children()
built-in function, 31
remove_all_parents()
built-in function, 31
remove_child()
built-in function, 30
remove_children()
built-in function, 30
remove_parent()
built-in function, 31
remove_parents()
built-in function, 31
roots()
built-in function, 29, 33

S

self_and_ancestors()
    built-in function, 31
self_and_descendants()
    built-in function, 32
self_and_partners()
    built-in function, 32
self_and_siblings()
    built-in function, 32
shortest_path()
    built-in function, 33
shortest_path_edges()
    built-in function, 37
sibling_count()
    built-in function, 35
siblings()
    built-in function, 32
siblings_and_self()
    built-in function, 32
sorted()
    built-in function, 38