Repair Compression Documentation

Release 1.0-alpha

Research Team

Aug 01, 2018

Contents:

Repair is a compression algorithm for graphs. This repository holds the source code for a complete implementation of the compression algorithm as well as the Graphs needed. It also contains reproducable source code for experiments that validate the effectivenes of the algorithm.

CHAPTER 1

Introduction

This documentation is organized to support the development process. It contains both high level discussions of why things are implemented the way they are as well as module level API documentation extracted from the code. You should treat this as the 'source of truth' as far the project is concerned.

If you are joing the team and want to contribute to the source code, you should look through *Getting Started*. If you are trying to refacto parts of the code or just want to understand how things work, take a look at *Design*. If you need information on how experiments are organized and performed, look at *Experiments*. Miscellaneous notes are in *Other Notes*, you might find some useful information there too. The *References* page has both external resources, like our Google Drive folder, and also internal resources like the API documentation. All of this is searchable, on the top left corner.

CHAPTER 2

Getting Started

Getting started, in terms of development environment setup, should be very easy since the project is written in pure Python with zero required dependencies. However, please follow the steps carefully because it will make your life easier later on.

2.1 Getting the source code

All of the source code for this project is version controlled using git and shared on GitHub. You will need to have git installed on your computer, then follow these steps:

1. Create a folder on your computer and inside that folder run:

```
git init
```

2. Add the repo as your remote:

```
git remote add origin git@github.com:yonathanF/Repair-Graph-Compression.git
```

- If that didn't work, you might have to use https version instead:

```
git remote add origin https://github.com/yonathanF/Repair-Graph-Compression.git
```

You now have all of the source code. But there is one more thing to setup. We are using virtual environments to avoid different results between different team members, so we will need to setup Virtual Env and download our dependencies.[1] Follow these steps:

1. Get Python 3, if you don't already have it.[3]
2. Download Pip. Their site has a quick setup[2] but if you are on linux, you might want to consider downloading it using your package manager.

[1] The main program doesn't need any of these packages. They are used for generating this documentation, and linting tools.
[3] https://www.python.org/downloads/
[2] https://pip.pypa.io/en/stable/installing/

3. Create a virtual environment. Go to the folder you created above, and run:

```
python3 -m venv env/
```

4. Activate the virtual env. You want to do this step very time you work on this project. On Posix systems:

```
source env/bin/activate
```

- If you are on Windows:

```
env\bin\activate.bat
```

5. Install the dependencies:

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

That's it! You are good to go. You might want to run the last command to stay upto date when we add new packages, but that's it for now!

2.2 Tools

There are some tools in place to make life easier. These are either the easiest to use or the standards in the community.

2.2.1 Unit Tests

Although this isn't a software development project, we still need some level of confidence in the quality of the code. At minimum we would like to know the code does its most basic task correctly, and so you must provide unit tests that test the happy path. Usually two or three tests are enough but you should try to cover as many edge cases as possible. We use Python's own *Unittest* please don't use any other frameworks since it will just be more work to keep track of them. Here are some commands to get you started:

```
python -m unittest discover -s tests # run all tests
python -m unittest tests.test_file_name # run a module of tests
python -m unittest tests.test_file_name.TestCaseName # to run a single test case
python -m unittest tests.test_file_name.TestCaseName.test_method_name # to run a␣
↪single test in a given test case
```

You should run these commands in the highest level of the directory (same level as *tests/*).

2.2.2 Coverage

We use line coverage to measure out unit tests' coverage. While this isn't the best metric for it, we just need the bare minimum and this does it for us. The package *Coverage.py* should already be on your system if you went through the setup steps above. You can generate a report using this:

```
coverage run --source='.' -m unittest discover -s tests/
coverage html # recommended
coverage report # to just see the numbers
```

Try to have 80% coverage or more at all times.

2.2.3 Linting

Python is very opinionated, perhaps a little too much, but this makes it easy to work in teams. Linting is a way to check your code for known bad practice, possible bugs, and style problems. The CI server will reject your code if the linter returns none 0 values, so please check it locally before pushing. Pylint should be part of your env already. Here is how to use it:

```
pylint folder/python_file_name.py # to lint a single file
pylint folder/ # to lint all the python files within
pylint folder1 folder2 folder3   # to lint multiple python folders
```

There are some rules that are useless and we ignore them (e.g. whitespace). If you come across a rule that you think is useless, please discuss with the team to add it to the ignore list.

2.3 Contribution Guidelines

As we add more and more code to the project, it must agree upon some basic rules to avoid stepping on each other. There are tools in place to make this easier (e.g. Git, linters,TravisCI, etc) but at the end of the day, it's all about communication. If in doubt, use one of the communication channels to talk to someone.

2.3.1 Working with Git

Git is both powerful and annoying. Fortunately, we only need some features to operate: pull, push, commit, merge, branch. Google everything else.

Commit often

Think of it like hitting save. Your last commit for the feature/fix/etc should be VERY detailed. Don't use *-m* for it. Write a *subject line* (a single sentence summary of what you did), followed by an empty line, followed by a paragraph describing *what* you solved/fixed, followed by an empty line, followed by a paragraph describing *how* you solved it. Follow this format strictly for the last commit. This makes the code changes amazingly traceable using git (e.g. someone can see who changed a given line, when, why, and how).

Branch often

Think of it like copy-pasting the source code into a new folder. Branches are very lightweight and an amazing way to isolate different works. Don't be afraid to have many branches; you can quickly delete them if you need to. By using branches to isolate your work, you can avoid merge conflicts for the most part. Here are a few rules related to branches:

1. Name your branches using the following format: *theme_area_firstName*. The theme should be something like *bugfix* or *refactoring*. The area should be more specific to what you are fixing, refactoring, etc; e.g. *graphs* or *generator*. And, ofcourse, the firstname should be your first name.
2. You can't push to the branches *master* and *development*. You need to make a pull request and at least 1 person needs to review it and TravisCI must green light it, before it gets merged into either of those branches. This is because *master* should always be stable and *development* should be stable enough for us to branch off of when we do work.
3. Related to (2) above, use the Git Workflow.[4] Branch off of development, do your work there, write unit tests, etc, push it to Travis, get the green light, and make a pull request. If you are working with someone else on the same problem/feature, you should create a different branch to combine your works. Handle your merge conflicts there and make a pull request from the combined branch. Pull request into development only.

[4] https://www.atlassian.com/git/tutorials/comparing-workflows

Use issues

Issues are the team's todo list. They make it really easy to discuss code, progress, and larger goals. For example, if you have a line like *Issue #12* in your commit, GitHub will link that commit in the issue comments. GitHub also provides a board for Agile development. All new issues go into the Backlog. When we decide to work on a given issue, we move it into Todo. It's then moved into In Progress once someone starts working on it. When the issue is closed, Automation will put the issue into the Done column. This makes it easy to see who's doing what and how much time we need to finish whatever feature/fix.

2.3.2 File Organization

The file organization is self explanatory, but for completeness sake:

- Graph directory –> Contains graphs, and cluster
- Nodes directory –> Contains nodes
- Repair directory –> Contains the Repair algorithm (both compression and decompression)
- Algorithms directory –> Contains algorithms that run on graphs
- Utils directory –> General tools like the graph visualizer and GraphML generator
- Tests directory –> All the tests
- Docs directory –> Contains this documentation
- Requirements directory –> Contains the production and development requirements

CHAPTER 3

Design

3.1 Overview

The source code is made up of three main components:

1. Nodes
2. Graphs and Clusters
3. Repair Compression

3.1.1 1. Nodes

Nodes are the basic building blocks of everything. Different kinds of graphs are formed by connecting these nodes in different ways, repair compresses graphs and clusters by replacing nodes, and decompression works similarly. At the most basic level, a node is simply a value and a list. The list has a references to every other node this node points (for every edge beween this node and any other node). This makes it easier to represent graphs in the standard adjacency form by simply having a list of nodes.[1]

In addition to the value and edges list, nodes also provide some other functionalities. These functionalities are placed in the node class, instead of in graphs, because they might enable parallelism when we need to optimize in the future (e.g. *replace*). In other cases, they simply make more sense inside nodes than anywhere else (e.g. *add_edge*).

Each node has a unique id, *uid*, because every node must be uniquely identifiable.[2] Nodes also have a *graph_id* attribute. This is necessary because when we create clusters, we would like to have nodes from the same graph appear closer to each other.

[1] Older versions of the code actually had a list of list implementation. This quickly got out of control and we ended up refactoring the codebase.

[2] Previous implementations made use of the value to identify nodes. This turned out to be an unnecessary limitation (e.g. the data can't have two 5s).

3.1.2 2. Graphs and Clusters

Graphs are composed of nodes, both in representation and behavior. We transfer most actions to the nodes, e.g. *add_edge*. The graphs' responsibility is to first ensure a given action is acceptable, e.g. when adding an edge at least the first node should be in the graph. Second, graphs act as message relays between nodes by acting as 'owners' of nodes. Agraph owns a node if the graph's ID matches the *graph_id* of the node. This graph is responsible for a few things. If it deletes the node, it must send a notification to other graphs to remove all edges coming to the node. If it recieves an event that the node should be replaced by something else, it must remove the node from its adjacency list and add the replacement in its position. Only the owner graph can delete a node.

There are different graphs that are created by inheriting from the main *Graph* class. These graphs enforce specific properties in addition to the parent's requirements. For example, *CompleteGraph* ensures that every node is always connected to all other nodes. This is useful when we create clusters.

A cluster is a group of graphs that have edges connecting nodes across graph boundaries.This lets us artificially simulate data with various properties. For example, a social network can be simulate by using a cluster of hub and spoke graphs (under the assumption that there are some people who have a lot of edges (hubs) and most people have few edges). Clusters are represented exactly like graphs but they *are* not graphs. They don't provide most of the functionalities graphs provide and they don't have ids. A cluster is basically formed by creating a group of graphs that are interconnected, and then extracting all the nodes from these graphs into the cluster. The function *weakly_connected_graphs* provides a way to create clusters out of graphs. The function is paramterized so that you can control edge density between graphs, within graphs, and number of graphs.

3.1.3 3. Repair Compression

Repair is a simple compression algorithm. It trades more nodes in the graph for fewer edges, hence compressing the graph. Another way to think of this is, it converts a dense but small graph into a sparse but larger graph (size measured in nodes). The basic idea is to look through the graph, or the cluster, and find the most common pair of nodes that show up adjacent to one another. Consider the following example:

```
N1: B, C, D
N2: B, C, A, E
N3: B, C, D
```

The nodes *B* and *C* show up next to each other three times, so they are the most common pair. Then replace the most common pair with a single node which points to the replaced nodes. In the example above, we could replace the pair *BC* by *N4* like so:

```
N1: N4, D
N2: N4, A, E
N3: N4, D
N4: B, C
```

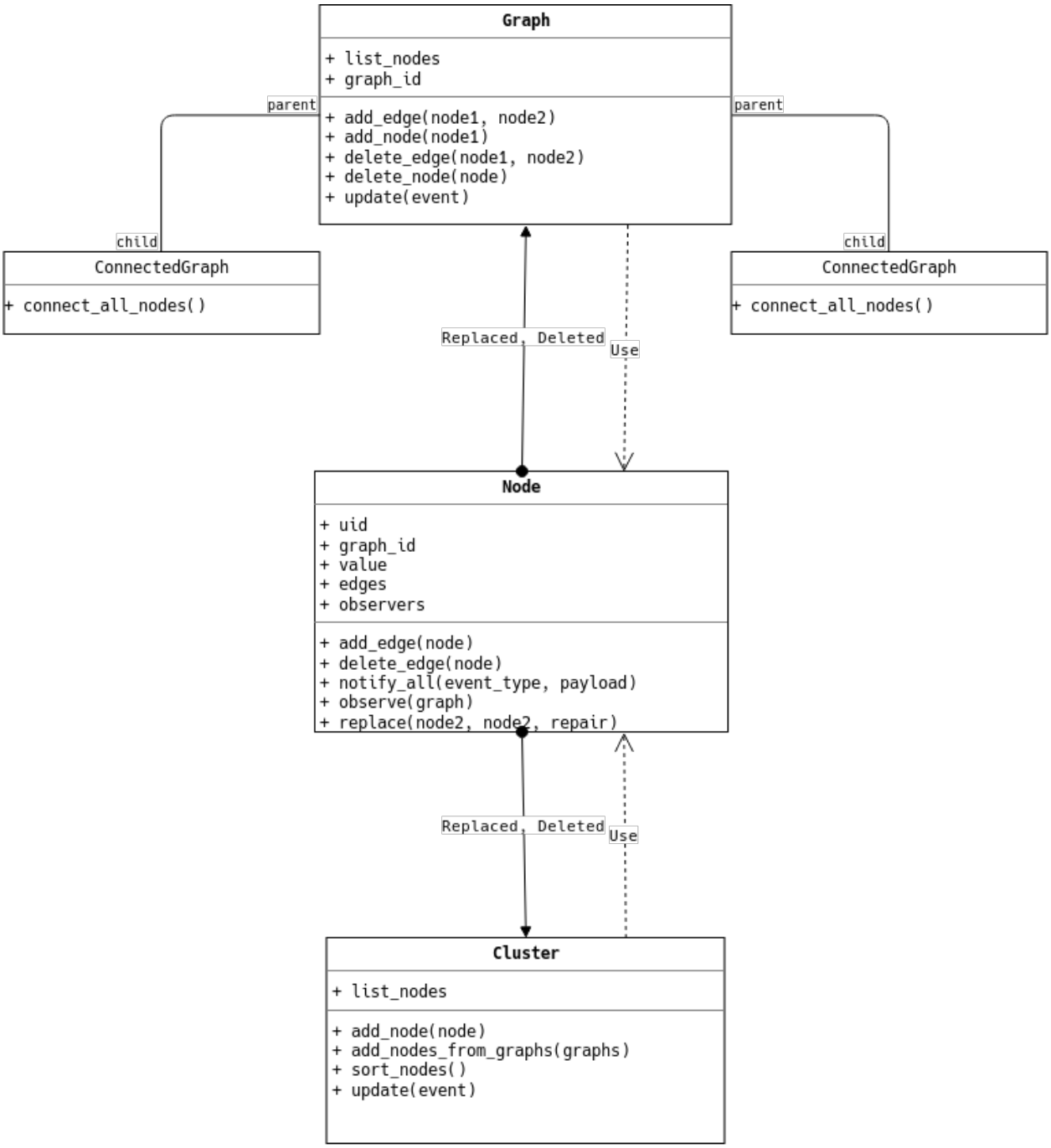
We have increased the total node count by 1 but we have decreased the total number of edges by 1. For some graphs, the decrease in edges can quickly outpace the increase in nodes. See the dissertation in *References* for more on this.

3.2 Architecture

The architecture can be divided into two main components: Cluster/Graphs/Nodes and Repair. Repair is the implementation of the compression algorithm that depends on the other three, but it's design and implementation is fairly independent of them.

3.2.1 Clusters, Graphs, and Nodes

A good portion of the codebase is dedicate to the implementation of an extendable Graph system. This implementation would be trivial if we didn't intend on creating clusters, i.e. edges are formed between two nodes within the same graph only. But if edges could be formed between two different graphs, the problem becomes a little more involved. For example, what if one of the graphs deletes one of the nodes of this edge? Because of the way Python handles references, the node wouldn't really be deleted. Python deletes objects when there are no references to the object and in this case the edge will keep a reference to to the node.



While it is still possible to solve this in a bruteforce way (e.g. check every single node for references), such a solution would make it impossible to extend the implementation in the future and harder to test, not to mention the performance

degradation.

CHAPTER 4

Experiments

CHAPTER 5

Other Notes

CHAPTER 6

References

6.1 repair-compression

6.1.1 graphs package

Submodules

graphs.clusters module

A group of graphs need a light wrapper around them because if we simply put them into a list, algorithms operating on the cluster will only see one adjacency list. This is a problem since, for example, repair will end up compressing only one of the the graphs instead of the whole cluster.

This class provides this wrapper.

class graphs.clusters.**Cluster**(*cluster_graphs*)

Bases: object

A simple wrapper for a list of graphs.

Parameters **graphs** – A list of graphs which it will convert into a cluster.

Return_type graph

add_node(*node*)

Overrides the parent to avoid graph_id check

add_nodes_from_graphs(*cluster_graphs*)

Adds all nodes from a graph to the clusters adjacency list. :return_type: graph

sort_nodes()

Sorts the nodes by graph_id then by node_id. This is important for the repair algorithm to work correctly because we want nodes of the same graph to be clustered around together. So we do a stable sort by graph_id, then by node_id.

update(*event*)
Event handler for nodes

graphs.complete_graph module

These graphs will consist exclusively of nodes which are connected to every other node in the graph

class graphs.complete_graph.**CompleteGraph**(*n_list=None*)
Bases: *graphs.graph.Graph*

A representation of a complete graph based on the more general *Graph* class. It ensures all nodes are connected all the time.

Note: It will reject changes to the graph that might cause it to lose this property.

---

add_edge(*node1*, *node2*)

Parameters **node1** – MUST be a node in the CompleteGraph

Paam node2 MUST be a node outside the graph (in another graph)

for connecting a CompleteGraph to other graphs, NOT for connecting a Complete graph internally.

add_node(*node*)
Adds a node and connects it to all other nodes in the complete graph Duplicate nodes fail silently.

connect_all_nodes(*n_list*)

Parameters **n_list** – A list of new nodes.

Adds a group of new nodes to the internal list, and connects them to each other. This should be used from __init__ only for the most part.

delete_edge(*node1*, *node2*)

Parameters

- **node1** – One of the two nodes that form the edge.
- **node2** – The second node that forms the edge.

node2 is removed from node1's adj list by calling Node.delete_edge() One node must be in graph, the other doesn't have to be so can remove edges between graphs.

graphs.graph module

Implementation of a graph.

This module provides the base graph used across everything. Both the compression and decompression algorithms as well as the cluster creating functions depend on it.

class graphs.graph.**Graph**(*nodes=None*)
Bases: object

The graph class implementation

add_edge(*node1*, *node2*)
node2 is added to node1's adj list by calling Node.add_edge() node1 must be/will be a node in the graph (1st param), if it isn't, it's added to the graph node2 can be in graph, doesn't have to be (add_node not called), must be not equal to node1

add_node(*node*)
Adds a node to the Graph data structures, but it won't be connected by any edge. Duplicate nodes fail silently. New implementations should redefine this function.

delete_edge(*node1*, *node2*)

Parameters

- **node1** – One of the two nodes that form the edge.
- **node2** – The second node that forms the edge.

node2 is removed from node1's adj list by calling Node.delete_edge() One node must be in graph, the other doesn't have to be so can remove edges between graphs.

delete_node(*node*)
removes the node from the graph, clears the adj list therein, resets the node.graph_id = None, and removes external references to the node

update(*event*)

Parameters **event** – an event namedtuple

Recieves an update when nodes change and reacts to the event. For node deletion, it will look through it's list and remove all references to that node. For node replacement, it will search for the replaced node and update it with the payload.

Note: If it is a replacement update, the payload should include the other node being replaced and the replacement node

---

graphs.graph_factory module

A factory class to help with the creation of objects of the different types of graph types. I added this to make it easier for future development. It is not really need for the three graph implementations we have right now.

class graphs.graph_factory.**GraphFactory**(*graph_type*, *num_of_nodes=0*)
Bases: object

This the base class for the factories.

get_graph()
sub classes must implement this method

class graphs.graph_factory.**GraphFactoryAlphaNumeric**(*graph_type*, *num_of_nodes*, *random_seed=-1*)
Bases: *graphs.graph_factory.GraphFactory*

Factory class that populates the graph with alpahnumeric values.

Pass a random seed other than -1 to set the seed.

Note: it doesn't create any edges b/n the nodes. Call rand edge from graph if you would like to randomly create the edges.

---

get_graph()
implementation of the factory

get_random_alpha_numeric(*upper_num=1000*, *lower_num=0*)
returns a string with a randomized, not necessarily unique, value containing numbers and characters

class graphs.graph_factory.**GraphFactoryNoData**(*graph_type*, *num_of_nodes=0*)
Bases: *graphs.graph_factory.GraphFactory*

A graph factory that doesn't add any data.

It wil generate graphs with empty nodes or empty node lists. .. note:: You probably want to use the other implementations since they can also add randomized data and take # of nodes as an argument

get_graph()
Creates graphs of different type based on the parameter

class graphs.graph_factory.**GraphTypes**
Bases: enum.Enum

Defines an enum to avoid using strings directly and add more type checking

complete = 1

generic = 3

hub_and_spoke = 2

graphs.graph_generator module

Creates a cluster of graphs.

The graphs will be connected as specified by the parameters of the function.

graphs.graph_generator.**weakly_connected_graphs**(*connection_num*, *graph_num*, *edge_num*, *graph_factory*)
generates a group of connected graphs as specified by the parameters:

Parameters

- **connection_num** – the number of connections we expect to see on ave
- **graph_num** – the number of graph in the cluster
- **edge_num** – the number of edges we expect to see within each graph on ave
- **graph_factory** – an instance of a graph factory to specify the kind of graphs

Returns A group of 1 or more graphs connected into one large graph

Return_type Graph

graphs.hub_and_spoke_graph module

These graphs will consist of many nodes all connected only to one central hub node. This graph should implement from the top down a guaranteed connected graph rather than the Graph class' possible bottom-up of creation of a connected graph

class graphs.hub_and_spoke_graph.**HubAndSpoke**(*hub*, *nodes=None*)
Bases: *graphs.graph.Graph*

A hub and spoke representation. :param hub: The hub node. :param nodes: All other nodes.

Note: The hub points to the nodes. Not the other way around.

Warning: The hub is not included in the list for now.

add_edge(*n1*, *n2*)
node2 is added to node1's adj list by calling Node.add_edge() node1 must be/will be a node in the graph (1st param), if it isn't, it's added to the graph node2 can be in graph, doesn't have to be (add_node not called), must be not equal to node1

add_node(*n*)
Adds a node and connects it to the hub node Duplicate nodes will raise an exception

delete_edge(*n1*, *n2*)

Parameters

- **node1** – One of the two nodes that form the edge.
- **node2** – The second node that forms the edge.

node2 is removed from node1's adj list by calling Node.delete_edge() One node must be in graph, the other doesn't have to be so can remove edges between graphs.

delete_node(*n*)
removes the node from the graph, clear the adj list therein, resets the node.graph_id to None, and removes all references other nodes hold to the deleted node

Module contents

6.1.2 nodes package

Submodules

nodes.nodes module

A representation of a node/vertex in a graph.

A simple implementation based on adjacency list representation of graphs. Implements the observer pattern so that nodes that have edges connecting to them from outside of their graph can be properly deleted.

class nodes.nodes.**Event**(*observable*, *event_type*, *payload*)
Bases: tuple

Create new instance of Event(observable, event_type, payload)

event_type
Alias for field number 1

observable
Alias for field number 0

payload
Alias for field number 2

class nodes.nodes.**EventType**
Bases: enum.Enum

Defines event types that graphs will recieve when they observe nodes

node_deleted = 1

node_replaced = 2

class nodes.nodes.**Node**(*value*, *edges=None*)
Bases: object

An implementation of a regular node.

Has it's own uid and also a graph id to show which graph owns it.

add_edge(*node*)
Adds an edge from itself to the param node.

Note: the direction is this -> param node.

---

delete_edge(*node*)
Deletes the edge between itself and param node if it exists.

notify_all(*event_type*, *payload=None*)

Parameters **event_type** – the type of the event that will be broadcasted

This notifies all observers of the event that just happened. The notification is passed by wrapping it in an event tuple where we pass the event type as well as a reference to this node

observe(*graph*)

Parameters **graph** – a reference to the graph that wants to observe the node

A graph can observe a node so that it can be notified when certain changes are applied to the node. For example, if the node is deleted by the graph that owns it, all other graphs should remove their bindings to the node as well. Another example, if the node is replace by a repair node, all graphs that have edges going to this node should update their references.

replace(*node1*, *node2*, *repair_node*)
Replaces every occurrence of node1 and node2 with repair_node if they appear in that exact order.

class nodes.nodes.**RepairNode**(*value*, *node1*, *node2*, *isDictNode=True*)
Bases: *nodes.nodes.Node*

A mostly useless implementation of a repair node. It simplifies some code in other place and also helps avoid use of list of lists etc.

Consider refactoring at some point.

Module contents

6.1.3 repair package

Submodules

repair.repair module

Implementation of the Repair algorithm Uses the Graph and Node classes to compress a graph.

class repair.repair.**Repair**(*graph*, *dictionary=None*)
Bases: object

The main class that holds everything together

compress()
Compresses the graph passed into the class

It updates the iternal dictionary, creates a dictionary node for the most common pair, unless all are unique (freq == 1), recurse.

decompress()
Decompression for Repair compressed graphs

It takes the graph passed into the class and decompress it. A graph that gets compressed with the compress method above and gets decompressed with this method should be back to the original graph

remove_compression_nodes()
clean up the decompression nodes

update_dictionary()
Updates the internal dictionary

Takes in a graph object, scans it, and updates the priority queue with new pairs and their frequency. Should only be called by compress_graph

class repair.repair.**RepairPriorityQueue**(*pairs=None*)
Bases: queue.PriorityQueue

A priority queue implementation Overrides a few methods to make python's queue implementation to work the way we need it to

---

Note: Both put and get default block to False instead of true. This makes the implementation closer to put_nowait and get_nowait. It makes a difference in a multithreaded setup

---

get(*block=False*, *timeout=None*)
Uses the priority queue to get the most freq. Then updates the dictionary as well.

put(*item*, *block=False*, *timeout=None*)
Uses a dictionary and the queue to handle duplicates

Get the pair from the dictionary which contains the same number of items. If it finds the time, update the freq before inserting. Otherwise put into the queue

Module contents

6.2 External References

- Shared Google Drive Folder
- Nathan's Dissertation

Python Module Index

Index

O

P

R

S

U

W