Accelerated Graph Features

A cache-accelerated method for storing and accessing graphs in C++

Table of Contents

1 Introduction

1.1 Overview and motivation

Calculating network features is an expensive business, consisting of calculations that often require traversing the entire network in order to compute the feature.

These days, the problem is magnified because of sheer size of the networks. The networks today can reach sizes of millions of nodes and billions of edges.

The networkx library, which is used to deal with large networks, is simply not fast enough to deal with those kinds of networks in an efficient manner.

To deal with those kinds of networks, a lower level solution must be used.

The solution described below is two-fold:

First, the code is written in C++, with an interface to python. The C++ language gives us the speed and flexibility we need to manage the large networks and expensive operations.

Second, the code is based around an extremely efficient data structure for storing large graphs, first devised by Dr. Lev Muchnik. The data structure used arrays to store an adjacency list in such a way that the computer's cache is used extensively, accelerating the access time to the memory.

The manual below is split into two sections:

User manual: a description of the features that can be used in python code.

Developer manual: details of the C++ implementation for future developers using the same code structure.

1.2 Installation

The installation of the package requires several components:

- The Boost.Python library[1]
- The shared library of C++ code (which needs to be built from source)
- The python wrapper code

The Boost.Python library can be built by two different methods:

1. Installation from source
2. Installation using the conda[2] package manager

While the second method is considerably easier, it is also less flexible than the manual installation from source.

1.2.1 Installation from source

Disclaimer: these installation instructions were tested on linux only, use on other operating systems at your own risk.

An example of the makefile that can be used with this method is *makefile.dsi.*

Instructions:

1) Download a compressed version of the Boost libraries. The version used for testing was 1.67, but more advanced versions should also work. The compressed tar.bz2 archive can be downloaded from https://www.boost.org/users/history/version_1_67_0.html.
2) Unpack the archive into a local folder using the command (make sure to change the version number to the one you downloaded):

   ```
   tar xvjf boost_1_67_0.tar.bz2
   ```

   The archive will unpack into the current folder and usually create it's own subdirectory.

3) Install a version of python3. For testing, the python3 version that is downloaded with Conda was used, but any version should work.
4) Run *./bootstrap.sh* from the main Boost folder that was just created with the following options:

   --with-python=[main python path]/bin/python3

   --with-python-version=3.7 (or whatever your version is)

   --with-python-root=[main python path]/lib/python3.7 (make sure the version matches the one above).

[1] https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html
[2] https://conda.io/en/master/

Generally speaking, the main python path will be either the /usr directory or the main Conda path (e.g. ~/anaconda3/).

5) Run the actual installation:
   a. Add the python you are using to the correct environment variable:

      export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:[main python path]/include/python3.7m *(*or whatever directory exists and matched the version you are using.)

   b. Run *./b2*

6) Update the following values in the makefile (it is recommended to create a copy of *makefile.dsi* and edit it).
   → Python version (long and short)
   → Python include and lib directories
   → Boost include and lib directories

   All of these variables can be found in the top of the makefile.

7) Add the same boost lib path the LD_LIBRARY_PATH is ~/.bashrc (which runs whenever you open a shell). Add the following line in ~/.bashrc:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:[main boost path]/stage/lib
```

8) Build the library by running

```
make –f your_makefile_name
```

From the *features_algorithms/accelerated_graph_features/src directory.*

1.2.2 Installation using Conda

Disclaimer: Again, these instructions were only tested on linux.

The makefile for these installation instructions is *Makefile-conda.*

Assuming you already have a version of Conda3 installed:

1) Create an environment from the configuration file *env.yml* that can be found at

   features_algorithms/accelerated_graph_features/src

   Move into the directory and run:

```
conda env create -f env.yml
```

   You should now have an environment called "boost" with all the necessary packages installed.

2) Build the library by running

```
conda activate boost
make –f Makefile-conda
```

If the making fails because of not recognizing existing files, one should consider following instruction 4) and possibly 5) from the source installation instructions.

1.3 Building the library for with GPU-accelerated features

Building the library with support for GPU features is done on top of a Conda installation.

On a machine where CUDA is installed, *Makefile-gpu* can be used to build the library with GPU features. The modifications needed on a per-basis system are the CUDA links in the makefile. The current configurations is a general case where the cuda libraries are in the /usr/local directory. Modifying the makefile requires changing the lines with the NV links on the top of the makefile.

In order to build the library, run the following:

```
conda activate boost
make –f Makefile-gpu
```

2 User manual

The code is exposed to the user in Python via a python package.

The C++ code is hidden behind pure python functions that handle the pre- and post-processing for the C++ code.

The exact processing pipeline is described in the next section – Developer manual.

In this section we'll describe the way the user will interact with the code, with examples.

There are two methods of interacting with the package. You can decide to interact with the base functions directly, which allows some flexibility, or you can use the FeatureCalculators which encapsulates the functions, which allows you to easily use the accelerated features in any existing code that already uses the graph_measures package.

2.1 Package structure

As we said, the code is encapsulated in a pure python package.

The package can be imported like every other python package by using the following line:

import features_algorithms.accelerated_graph_features.src *as af*

The package allows immediate access into the feature functions.

All the functions are of the format:

*af.function_name(nx_graph,**kwargs)*

Where:

- *nx_graph* is a Graph object (or DiGraph) from the networkx[3] package
- ***kwargs* are further **keyword** arguments needed for the feature calculation.
- ***kwargs* will always be able to receive the following arguments:
  - → *with_weights:* a boolean specifying whether to add weights to the C++ code.
  - → *cast_to_directed:* a boolean specifying whether to force to graph into a directed form.
  - → *converter_function:* if given, replaces the internal converter function, the specifications for such a function are given in the developer section.

The function return either a list (of there is a value returned for each node) or a float (if there is only one number returned from the feature calculation.

[3] https://networkx.github.io/

2.2 Code assumptions

Since the code to convert the nx Graph into a format compatible with the C++ code is internally contained inside the python wrapper package, there are several assumptions made on the structure of the graph passed into the function.

If you wish to specify a different converter function, please look at the relevant section in the developer manual.

The assumptions are as followes:

a) The nodes are labeled with numbers
b) Those numbers are the sequence [0,...,num_of_nodes-1]
c) Currently, the library does not support weighted graphs.

Note that there are no assumptions on whether the graph is directed/undirected, but there is an effect on the result of the code (for details, see the developer section).

3 Developer manual

This section dives into the internal structure of all sections of the code, and is meant both for debugging and extending the code.

3.1 Conversion pipeline

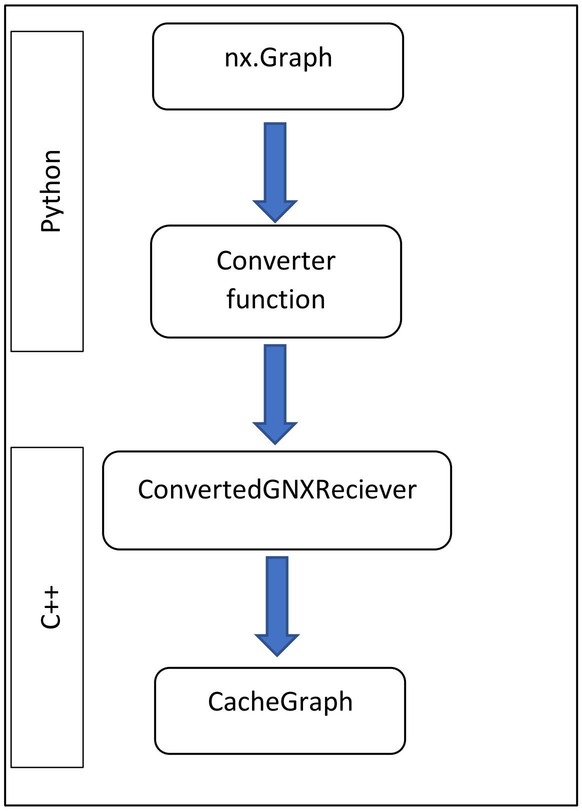
As we are dealing with code from both Python and C++, there is a conversion routine set in place that enables the use of the same graph in both Python and C++.

The conversion takes in (in Python) a Graph or DiGraph object (from the networkx library) and outputs (in C++) a CacheGraph object, which is the format used for cache-aware storage of the graph and access to it in C++.

3.1.1 Pipeline description

The pipeline is described in the diagram to the right.

The input of the conversion is a graph from the networkx package, and the output is a CacheGraph (the graph structure used in the C++ code).



3.1.2 Converter function[4]

The converter function is a pure python function that receives a nx.Graph as input and returns two or three lists: indices, neighbors [,weights].

A weights list is returned only if weights were requested (by setting *with_weights*).

The default converter function is located in the *graph_converter.py* file.

For those who wish to write their own converter function, here are a few guidelines to follow:

- ✓ The function should be able to determine whether the graph is directed (or if a directed version is requested) and act accordingly.
- ✓ The function should handle the case for both weighted and unweighted graphs
- ✓ The function should not break for:
  - A node without neighbors
  - An unweighted graph where the user asked for weights
- ✓ The function should notify the user when working on a directed graph.

[4] Can be found in *graph_converter.py*

3.1.3 ConvertedGNXReciever[5]

This is a C++ object that has dual responsibilities:

- Convert a dictionary of lists (that was created by the Python converter) into a CacheGraph. A pointer to the created CacheGraph is saved in the Reciever.
- Memory management: the graph will be deleted alongside to converter at the end of the converter's scope. This will usually occur at the end of the wrapper function (that is exposed to Python) and so the graph will be cleaned automatically at the end of the calculation.

3.1.4 CacheGraph[6]

The CacheGraph object contains 3 private members:

1) *NumberOfNodes:* an **unsigned int** that holds the number of nodes in the graph
2) *Graph:* a vector of **unsigned int**s which holds a sorted neighbor list for each node.
3) *Offsets:* a vector of type **int64** (which is our shorthand for **unsigned long long** in linux) that holds at position *i* the offset of the first neighbor of node *i* in the *Graph* array.

Access to the graph is conducted by iterating both of the arrays (see the CacheGraph usage section below).

[5] Can be found in the utils directory of the C++ code

[6] Can be found in the *arch* directory of the C++ code

3.1.5 Example[7]

There follows an example of the conversion routine for a simple graph.

The given graph is the one composed of these edges: (0->1, 0->2,0->3,2->0,3->1,3->2).

The behavior of the conversion now depends on whether the graph is directed (either by passing in a nx.DiGraph or by specifying *cast_to_directed* as true) or undirected.

If the graph is directed, the result is as follows:

> Indices: [0, 3, 3, 4, 6]
> Neighbors: [1, 2, 3, 0, 1, 2]

Else, the results for the undirected graph are:

> Indices: [0, 3, 5, 7, 10]
> Neighbors: [1, 2, 3, 0, 3, 0, 3, 0, 1, 2]

It is worth noting a few things:

- Notice that the last element in the indices list actually contains the number of edges in the graph (i.e. the length of *neighbors*). This fact will be used in the C++ code.
- The length of the *neighbor* list in the undirected version isn't double the length of the undirected one, and that is because a bidirectional edge (0<->2) already exists in the directed graph.
- Note the *indices* list of the directed version. Since node 1 has no neighbors (it doesn't point to any other node), it's corresponding neighbor list section is empty and so the index in the *indices* list doesn't increment).

3.2 CacheGraph usage[8]

The CacheGraph object is designed around the principle of cache-awareness. The most important thing to remember when accessing the graph in the C++ code is that we are aiming to accelerate the computations by loading sections of the graph into the cache ahead of time for quick access. When using the CacheGraph, this comes into effect when we iterate over the offset vector first and then access the blocks of neighbor nodes in the graph vector.

By doing this, we are pulling the entire list of a certain node's neighbors into the cache, allowing us to iterate over them extremely quickly.

A concrete example of a good use and bad use case of the CacheGraph are the two popular search strategies BFS and DFS.

[7] This example can be found in the *test_python_converter.py* file.
[8] Example code based on the code in the clustering coefficient feature calculator

When using BFS we are utilizing the full power of the cache-aware storage mechanism by pulling in the entire list of a node's neighbors to be processed at once, which is fast due to the fact that the neighbors are in the cache.

In contrast, when using DFS, we are not going over all of a node's neighbors at once but jumping from one node to the other, which completely nullifies the speed advantage that the cache can give us, as the contents of the cache need to be constantly swapped out by the proccessor.

There is a general code structure used when dealing when a CacheGraph that is designed as part of our cache-aware strategy:

```
// This snippet is taken out of the clustering coefficient feature calculator

    //There is usually no need to declare the CacheGraph manualy, as it is a
    member variable of the FeatureCalculator class.
    CacheGraph* mGraph;
    //Get the neighbors and offsets from the graph
    const unsigned int* neighborList = mGraph->GetNeighborList();
    const int64* offsetList = mGraph->GetOffsetList();

    // For readability, we save the begining and the end of the current
    neighbor list at each step.
    int64 begin_offset, end_offset;
    // We start by iterating over all the nodes in the graph
    for (unsigned int i = 0; i < numOfNodes; i++) {
              // Below are the edges of the neighbor list for the current
      node (including begin_offset, excluding end_offset)
        begin_offset = offsetList[i];
        end_offset = offsetList[i + 1];
        // Now we can iterate over all of the neighbors.
        // It is important to note that behind the scenes a large section of
        the neighbor list is already
        // put in the cache by the os, which means our access time is reduced
        dramatically.
        for (auto p = neighborList + begin_offset; p < neighborList +
        end_offset; ++p)
        {
            //p is a pointer to the current node in the adjacency list.
            //Example usage: the offset of the current node is offsetList[*p]

            //Another example: check if there are neighbors inside the
            neighbor list of the current node.
            for (auto q = neighborList + begin_offset; q < p; ++q) {
            //We need to go over all the pairs, but we don't want to count
            every edge twice, so we only iterate over q<p.
                if (mGraph->areNeighbors(*p, *q))
                    connectedNeighbors++;
            }
        }

    }
```

3.3 Writing a Feature Calculator[9]

Writing a feature calculator consists of two parts:

- Creating a class that inherits from the FeatureCalculator[10] class (or one of its sub-classes, such as NodeFeatureCalculator, GPUFeatureCalculator, etc.).
- Creating a wrapper function that will be exposed to the user in Python
- Creating a pure python function that will perform the preprocessing necessary from the Python side.

3.3.1 Calculator object

The main body of code for the feature calculation will be held in this class.

As a class that inherits from FeatureCalculator, the skeleton of the code is already build, enabling the developer to focus on the calculation itself.

The FeatureCalculator class is a template class that gives you the following:

1. Access to a **const CacheGraph*** that is the CacheGraph given to the Calculator.
2. The need to only implement the *Calculate* function, which will be called by the wrapper function.

Notice that when you extend the base FeatureCalculator class, you must specify the template argument for it.

In the next page there is an example of a skeleton version of a FeatureCalculator class, which includes the basic requirements needed in each FeatureCalculator.

[9] Here we will be building the python side package as described in the Boost.Python manual:

https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/tutorial/tutorial/techniques.html#tutorial.techniques.reducing_compiling_time

[10] Can be found in the *arch* directory of the C++ code

```
/////////////////////////////////////////////////////
////////////// h file ///////////////////////////////


#ifndef EXAMPLEFEATURECALCULATOR_H_
#define EXAMPLEFEATURECALCULATOR_H_

#include "FeatureCalculator.h"

// Notice that we're returning a float from this feature calculation,
// it could just as easily have been a vector<float>

class ExampleFeatureCalculator: public FeatureCalculator<float> {

public:
    ExampleFeatureCalculator();
    // This is the only function you actually need to implement!
    virtual float Calculate();
    virtual ~ExampleFeatureCalculator();
private:
    // You can add whatever private variables\methods you want!

};

#endif /* EXAMPLEFEATURECALCULATOR_H_ */

////////////////////////////////////////////////////////////////////
//////////////////// cpp file //////////////////////////////////////


#include "../includes/ExampleFeatureCalculator.h"

ExampleFeatureCalculator::ExampleFeatureCalculator() {
    // If there is any additional preprocessing you would like to perform,
    // this is the place for it.

}

/**
 * The actual calculation goes here
 */
float ExampleFeatureCalculator::Calculate() {
    return (float) mGraph->GetNumberOfNodes();
}

ExampleFeatureCalculator::~ExampleFeatureCalculator() {
    // Don't forget to release any memory you may have allocated!
}
```

3.3.2 Wrapper function

The code for the wrapper function is usually written in a separate C++ file, under the *wrappers* directory. Truth be told, there are two wrapper functions – one which actually wraps the Feature Calculator, and another that defines it for the Boost.Python module.

An example of the two can be found below:

```
///////////////////////////////////////////////////////////////
///////////////////  h file  /////////////////////////////////

#ifndef CALCULATORWRAPPEREXAMPLE_H_
#define CALCULATORWRAPPEREXAMPLE_H_

#include <boost/python.hpp>
#include <boost/python/dict.hpp>
#include "../includes/ConvertedGNXReciever.h"
#include "../includes/ExampleFeatureCalculator.h"

using namespace boost::python;

void BoostDefExampleCalculator();
void ExampleCalculatorWrapper(dict converted_graph);


#endif

/////////////////////////////////////////////////////////////
////////////////  cpp file  ////////////////////////////////

#include "ExampleWrapper.h"
void BoostDefExampleCalculator(){
    def(ExampleCalculatorWrapper,"example_feature");
}
float ExampleCalculatorWrapper(dict converted_graph){
    ConvertedGNXReciever reciever(convertedGraph);
    ExampleFeatureCalculator calc();
    calc.setGraph(reciever.getCachedGraph());
    return calc.Calculate();
    // At this point in time the graph is also deleted
}

/////////////////////////////////////////////////////////////
//////////////  main file  /////////////////////////////////
//This is the file which is compiled into an .so

#include <boost/python.hpp>
#include "../wrappers/ExampleWrapper.h"
// ... other imports ...


BOOST_PYTHON_MODULE(_features)
{
    // ... other boost def wrappers ...
    BoostDefExampleCalculator();
}
```

Note that if the feature returns a vector, it is the responsibility of the wrapper function to convert it into a python list (boost/python/list.hpp).

The separation of the boost definition functions into different files is important. The compilation against Boost.Python is the most time-expensive compilation, and we don't want one change in a single calculator to cause us to recompile all of the wrapper functions.

3.3.3 Python-side function

The python side function is responsible for both pre- and post-processing for the feature calculator.

Pre-processing consists of converting the graph into a dictionary of lists (as done by the functions in the *graph_coverter.py* file) and of any other additional transformation needed to adapt the rest of the parameters to a format compatible with the C++ code.

Post-processing can include any action necessary to convert the results of the C++ code into a Python-friendly format.

Luckily for us, since most of the actions in the python side function are the same for all exposing functions, there is a python decorator[11] designed to save you the trouble of writing the same code over and over[12].

```python
"""
This code is in a pure python file which contains all the wrapper functions.
"""

from feature_wrapper_decorator import FeatureWrapper
from test_python_converter import create_graph


@FeatureWrapper
def example_feature(graph, **kargs):
    # example_feature is a C++ function exposed to python
    from _features import example_feature

    print(graph['indices'])
    print(graph['neighbors'])

    # Here 0 is the default value for the argument
    example_arg = kargs.get('example_arg', 0)
    res = example_feature(graph, example_arg)
    # Any post-processing goes here
    return res

# Usage is as follows:
result = example_feature(create_graph(), example_arg=42)
```

[11] The *FeatureWrapper* decorator can be found in the *feature_wrapper_decorator.py*
[12] For more on python decorators, see:
https://www.thecodeship.com/patterns/guide-to-python-function-decorators/
https://www.artima.com/weblogs/viewpost.jsp?thread=240808

Notice that the python function must have **two and only two** arguments:

- → *Graph:* the user passes in the nx.Graph, but because of decorator magic, inside the function it is a converted graph dictionary
- → ***kwargs:* any other arguments must be keyword arguments. Among them may be *with_weights, cast_to_directed* and *converter_function*.

In this code example you can also see the name of the compile C++ shared library is _features, which in the context of the python package means that the code is invisible to any code outside the function.

All function which are exposed to the outside python code (i.e. the python wrapper functions) are visible only because they are imported in the *__init__.py* file of the package.

4 List of features

There follows a list of all the features available in the library.

Notes about the table:

- **Name** is the name of the python wrapper function (which is the one the user calls).
- **Arguments** are all the arguments given to **kwargs (in python) - i.e. not including the graph itself.
- **Return type** is the python type returned (e.g. list and not vector)
- **Notes** contains any other assumption and/or conditions for the feature (for example, that the graph must be undirected).

Name	Arguments	Return type	Notes
node_page_rank	dumping: float, iterations: int	List of floats	
k_core		List of ints (unsigned)	
bfs_moments		List of tuples (float,float)	
3- and 4-motifs	level: int gpu: boolean, whether to use the GPU version	A matrix of size num_nodes X num_motifs	Also has a GPU version
flow	threshold:float	List of floats	
Attractor_basin	alpha:int	List of floats	

5 Benchmarks

Since the entire point of creating this package is to enable faster computations of graph features, below we include some benchmark speed tests for the various features.

The purpose of showing the benchmarks is twofold:

1) Prove the necessity of having faster computation for large graphs
2) Allow the user to know when it is better to use the python version of the feature calculation, as using a feature calculator from this package involves some overhead (such as the conversion routine and access to the shared library).

The features will be timed for graphs of increasingly larger size (in both the number of nodes and the number of edges).

The graphs used will be random graphs (specifically, erdös-reyni graphs generated by the networkx package[13]). Since we are using random graphs, we will use several different graphs for each benchmark, taking the average of the time. The graphs will be generated by a predetermined set of seeds, so as to use the same graphs for all benchmarks.

The python features will be calculated either by internal functions in the networkx package or by the feature calculators from *graph-measures*.

[13] https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html#networkx.generators.random_graphs.erdos_renyi_graph

5.1 Initial Benchmarks

As a first test of the viability of the project each feature was tested over a set of random graphs with an increasing number of nodes.

To maintain consistency between all tests, the graphs were erdös-reyni graphs generated by the NetworkX function[14] using a constant random seed set to 123456.

The number of nodes was chosen from the following: [50, 100, 300, 500, 1000, 2000, 3000, 4000, 5000]. The p value for the graphs was a constant 0.8.

The results of the initial benchmarks are extremely promising.

The four features in this benchmark were: Clustering Coefficient, Node Page Rank, K Core and BFS Moment.

The first three were tested against the corresponding functions in the NetworkX package[15], and the latter to the existing code in the *graph-measures* GitHub repository.

The calculation of the feature in C++ is made up of two distinct parts:

1) **Conversion:** converting the Graph object of the networkx package into the format used by the C++ code.
2) **Feature calculation:** the actual calculation of the feature in C++.

Each part is timed separately, as the conversion is only needed once per graph (and afterward the converted graph can be written to a binary file that can be loaded directly into the C++ code), and the feature needs to be calculated on its own.

On the other hand, the python code has no conversion of the graph, and so only has one time measurement.

The actual benchmarking code can be found in the *feature_benchmarks.py* file, and the timer used for timing the benchmarks is in the *graph_timer.py* file.

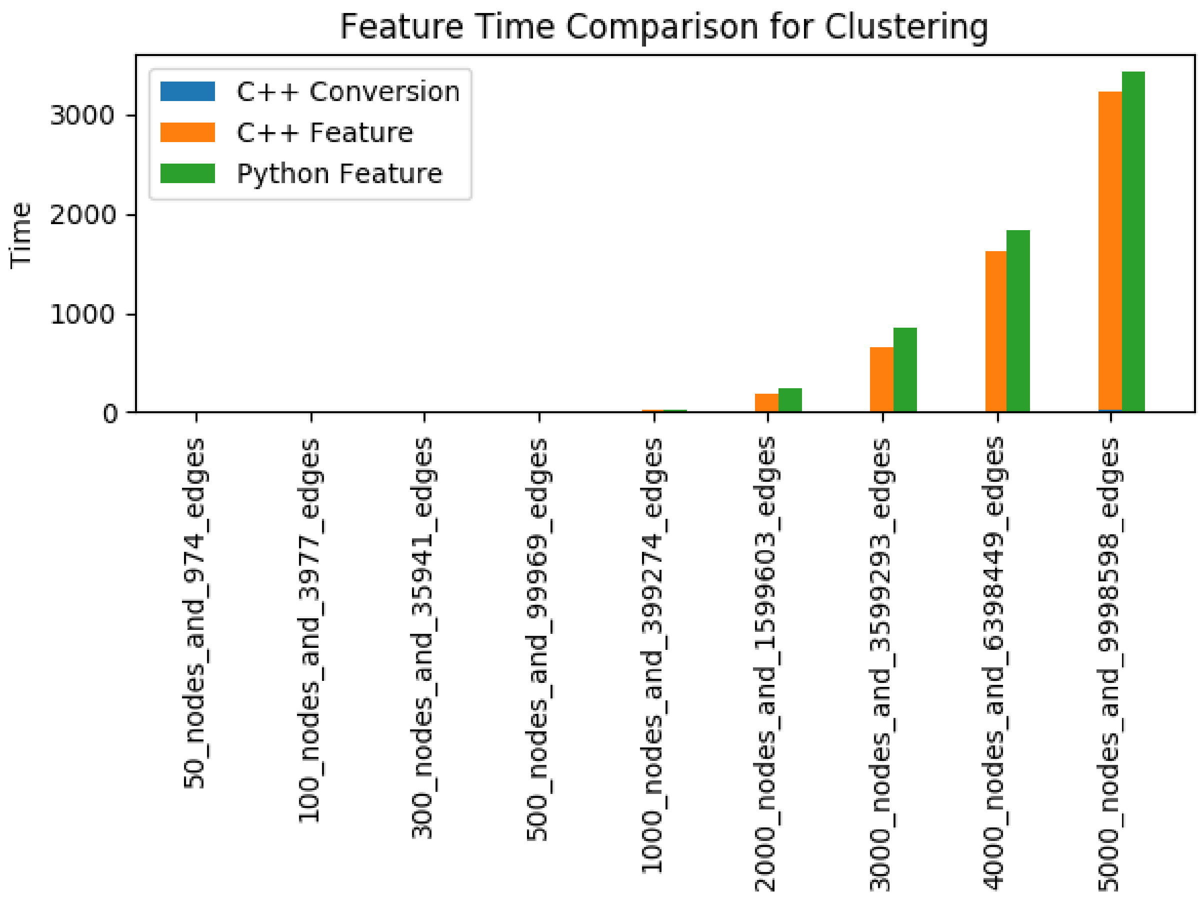
[14] https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.generators.random_graphs.gnp_random_graph.html#networkx.generators.random_graphs.gnp_random_graph

[15] The functions' docs can be found in the following links:
Clustering Coefficient: https://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.cluster.average_clustering.html
Node Page Rank: https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank.html
K Core: https://networkx.github.io/documentation/networkx-1.7/reference/generated/networkx.algorithms.core.k_core.html

5.1.1 Clustering Coefficient

We'll start with the worst results.

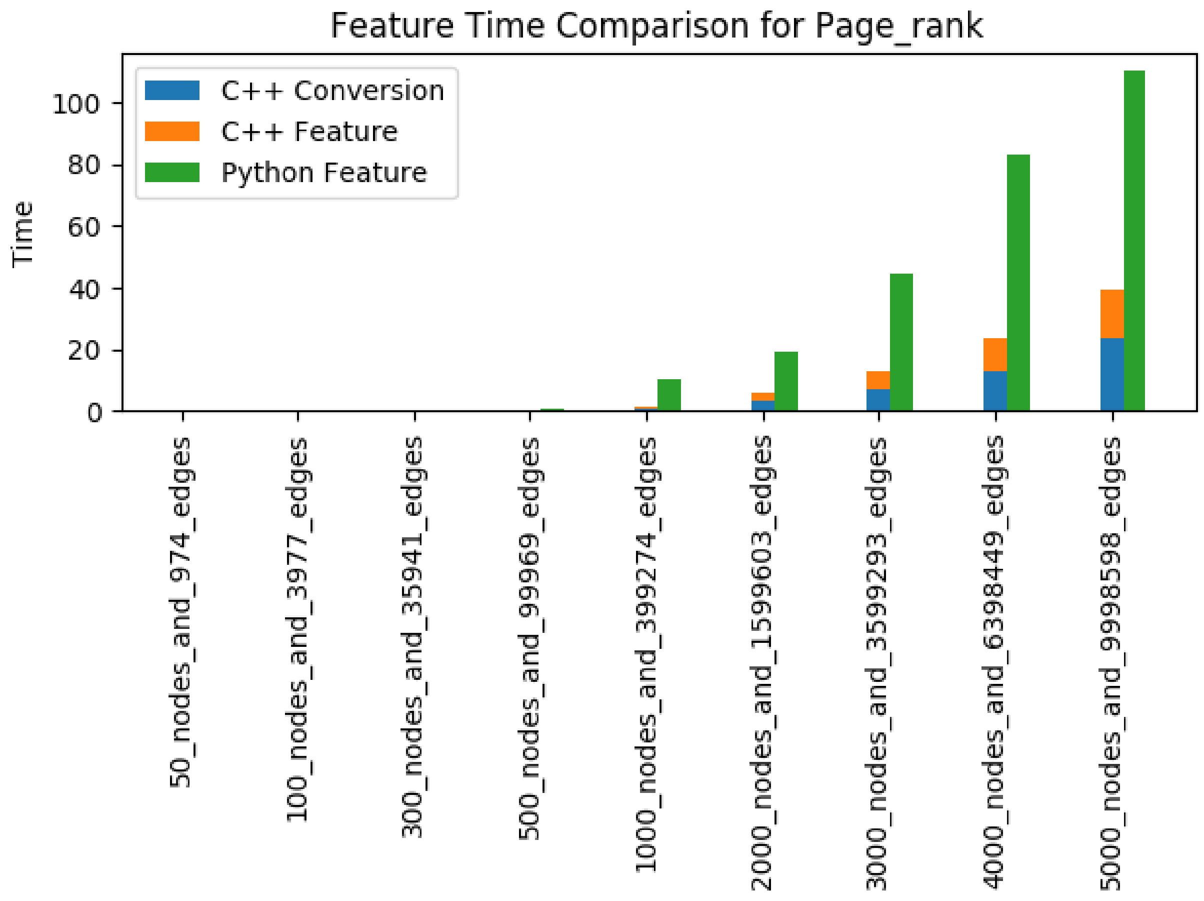
The clustering isn't the ideal feature for the cache-aware format we're using, and as such the time saved, while not negligible, is mainly due to the fact that the C++ language code runs faster than Python.



It can also be seen that in this case, as the feature time is $O(n^2)$, the conversion time is irrelevant.

5.1.2 Node Page Rank

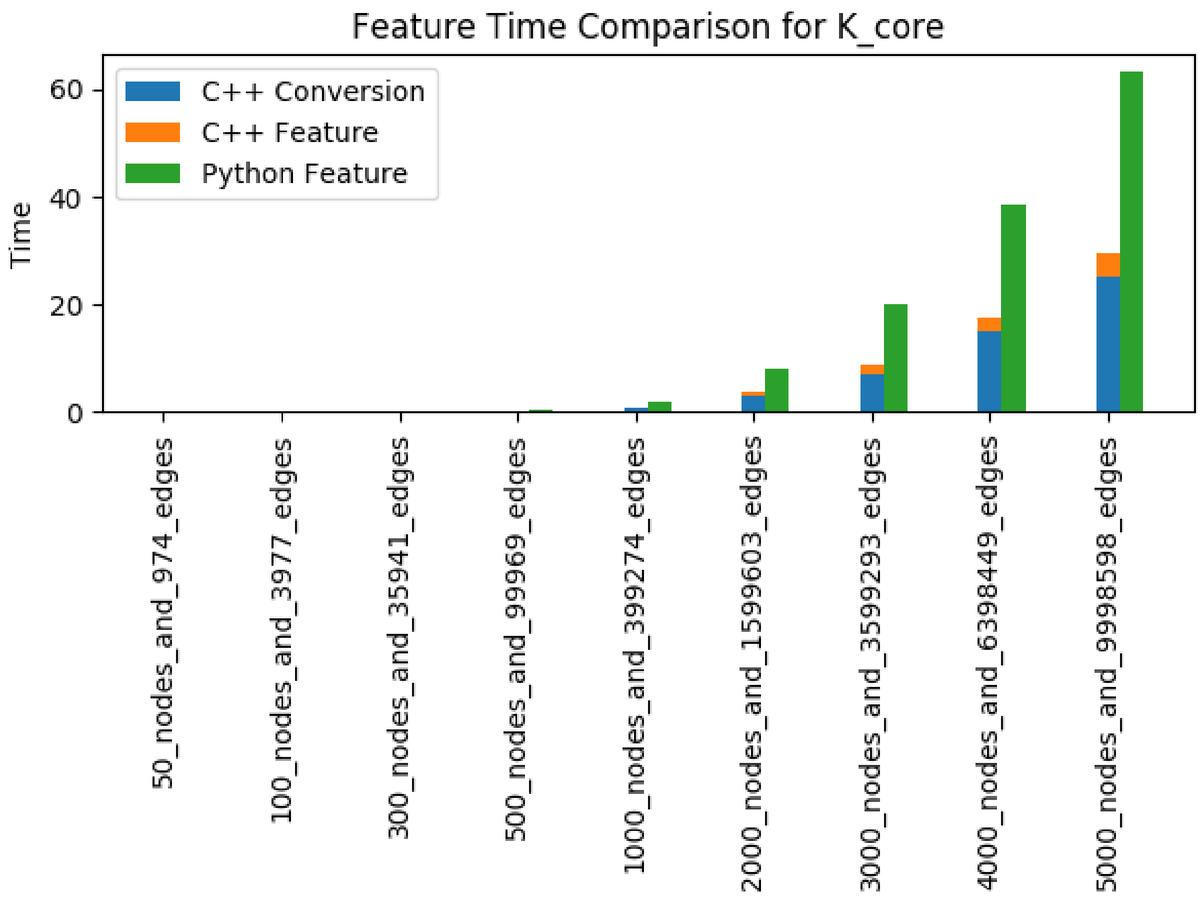
This feature and the ones following are linear by the number of nodes and utilize out cache-aware graph extremely well. As such, the gain of the C++ feature is much greater, especially when considering the face that the graph can be converted ahead of time, saving the conversion time in each feature.



In this case, as the feature itself takes less time, the cost of performing the conversion each time can be seen. This demonstrates a case where converting the graph ahead of time can constitute a large time saver.

5.1.3 K Core

This case is an even better improvement that the Page Rank, again if the graph is pre-converted.

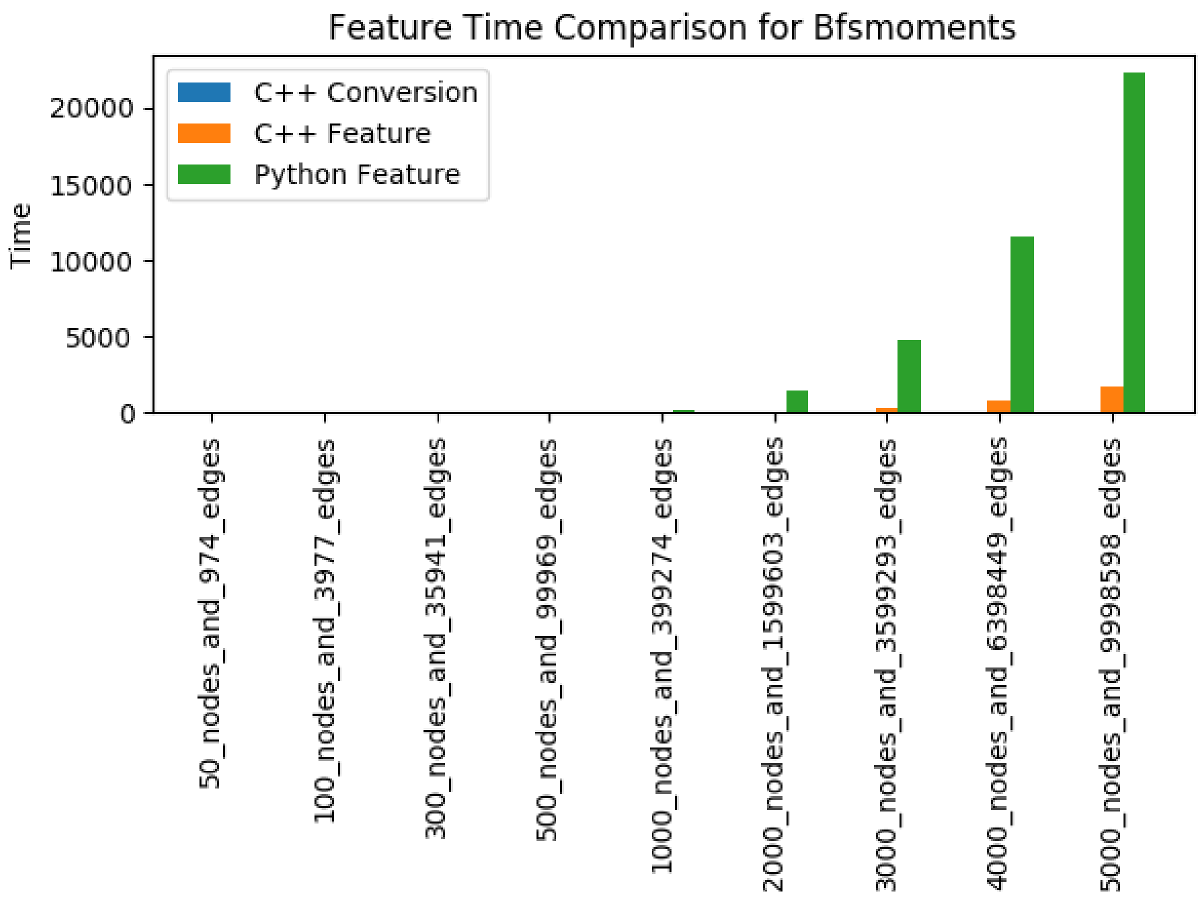


5.1.4 BFS Moments

In comparison to the last two features, this feature takes much longer.

This attribute has two consequences:

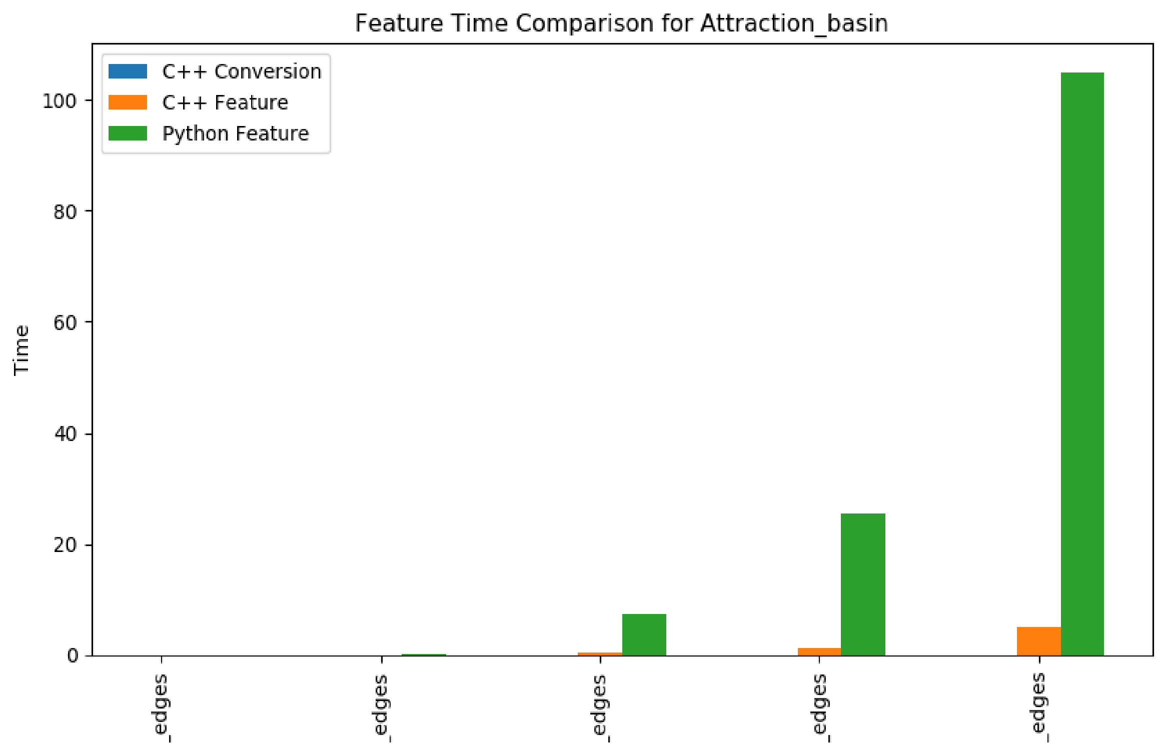
1) The conversion time, in comparison with the time it takes to calculate the feature, is again negligible.
2) The gain of the C++ code is much greater than the previous features, as it has a longer time to take advantage of the cache-aware graph format.



5.1.5 Other BFS based features

As with the previous feature, these two features (attraction basin and flow), give us a 10X boost over the corresponding python code.

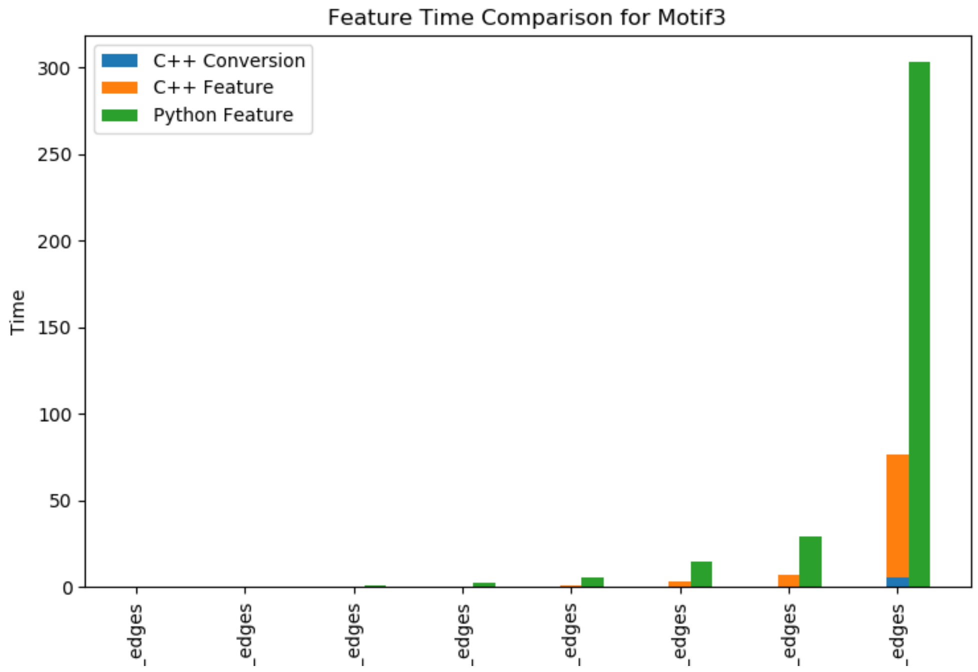
It should be noted that these features were tested on much more powerful servers than the previous features, and so are not in the same time scale as them. The speed improvement given by the C++ code, though, can still be clearly seen.

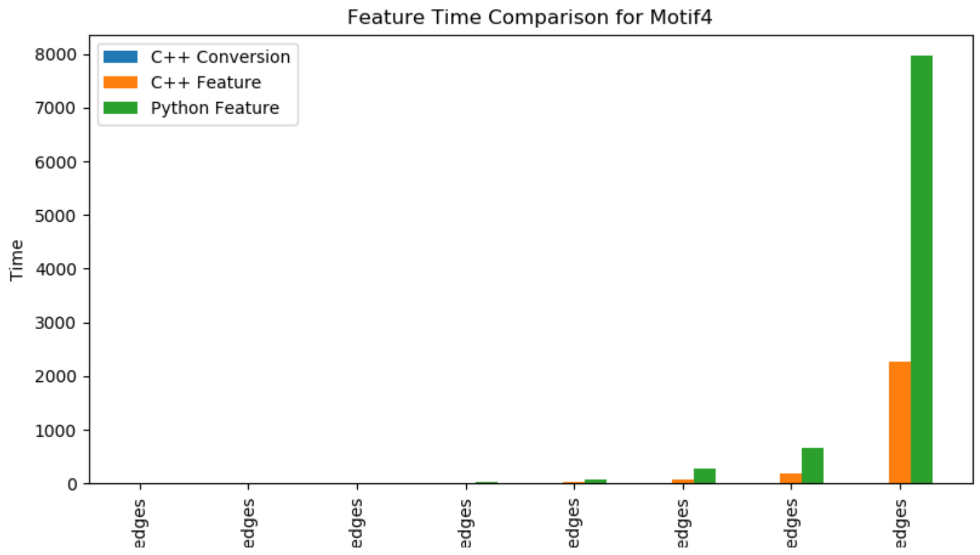


5.1.6 Motifs

The motif algorithm is one of the most commonly used algorithms in the package.

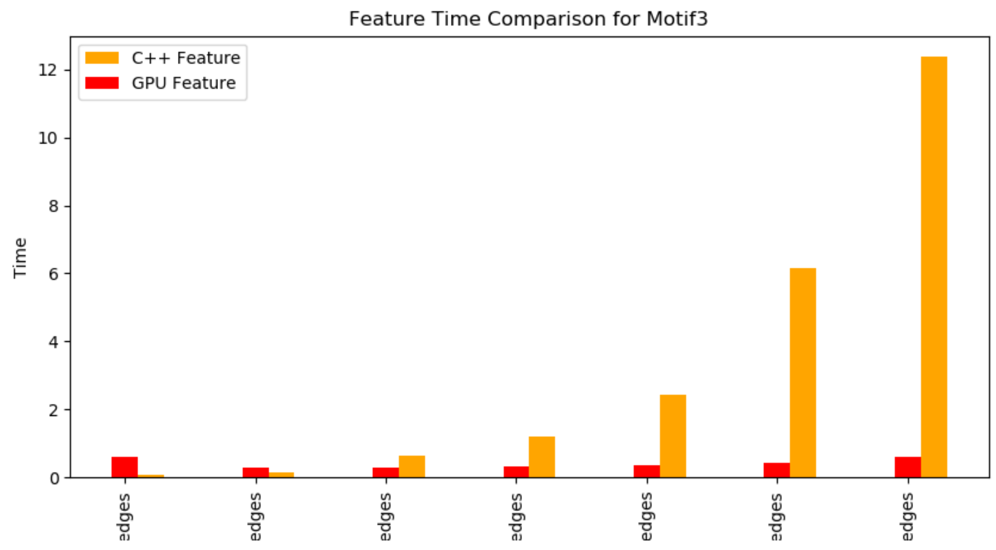
While the C++ code in this case does not give us the 10X boost the BFS features give us, it still gives us a respectable 3X-4X boost.

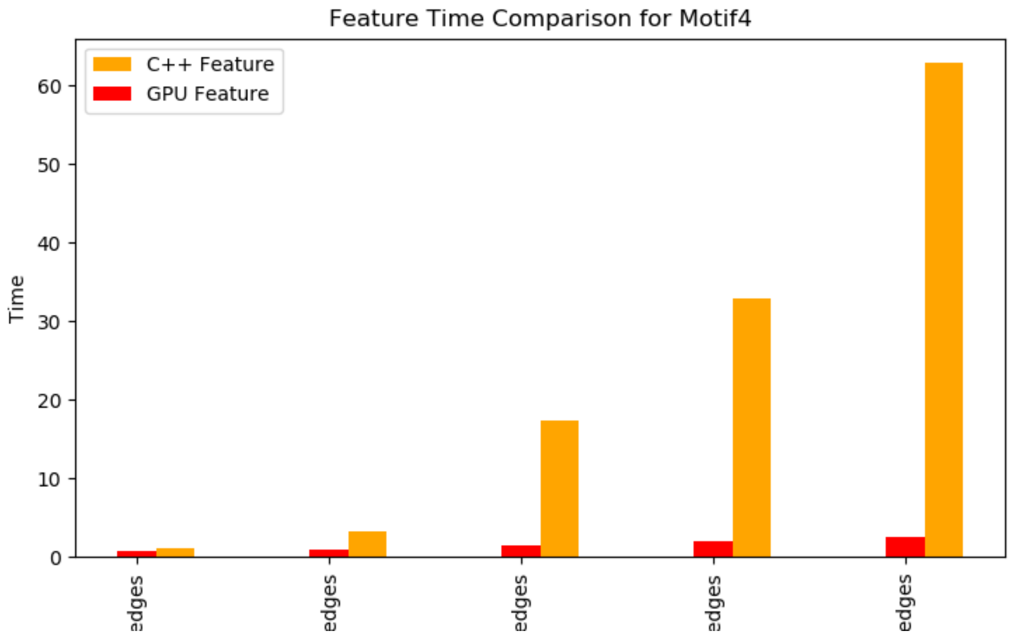




In addition to the usual C++ code, there also exists a GPU version for the motif code.

The GPU gives us an additional 10X boost **over the C++ code**, which results in a total of up to 50X over the python code. It can also be seen that the GPU code requires an overhead, making it unsuited for small graphs.


Feature Time Comparison for Motif3
C++ Feature
GPU Feature
Time
edges
edges
edges
edges
edges
edges
edges


Feature Time Comparison for Motif4
C++ Feature
GPU Feature
Time
edges
edges
edges
edges
edges