Motif Counting Algorithm

Ori Cohen

February 26, 2019

Abstract

This report aims to describe the algorithm used to count all the motifs of a graph in an Optimal manner. The algorithm is optimal in the sense that each motif is only counted once, resulting in an algorithm that is linear in the number of motifs in the graph. The algorithm takes advantage of the combinatorics of the 3- and 4-motifs to explicitly iterate over them. This report further describes the new C++ kernel that has been implemented to facilitate work on large graphs, which can run both on CPU and GPU devices.

This report presents a top-down approach to the algorithm and the theory behind it.

All the code described in this report can be found as an open source library at the GitHub repository[1].

[1] https://github.com/oricc/graph_measures

Contents

List of Algorithms

1 Algorithm Description

1.1 The Short Version

This section aims to describe the essence of the algorithm without going into the details, for those who are only interested in a high level view.

There are three main principles in the algorithm:

1. **Divide and conquer**. As we discussed above, we sort the nodes by their degree, from the node with the highest degree to that with the lowest. This allows us to quickly separate the graph into smaller, unconnected sub-graphs.

2. **Known motif structure.** As we can see in figure 2, we can list all of the options for the different motif structures. The algorithm is build so that it enumerates all the motifs of a certain structure before moving on to the next one, allowing us to make assumptions about the next motifs we should count.

3. **Conditions to stop double-counting.** Following the previous principle, we can make assumptions about the order in which the motifs are counted, and so we can write specific conditions that stop the code from counting certain motifs twice.

All motif building algorithms are a combination of these principles, allowing us to efficiently count all motifs in a graph. The rest of this report describes the algorithms with all details.

1.2 Overview

There are two main ideas implemented in this algorithm.

The first is that of explicit counting - since we know all the possible ways a motif can look, we can iterate over them directly, thus enabling us to enumerate all of them in an efficient fashion, both in memory and in time.

The second is the well known concept of divide and conquer. We use this concept by removing from the graph each node we have gone over. This concept is strengthened by a simple preprocessing of the graph, sorting all the nodes by their degree from highest to lowest. Having the nodes sorted in this way allows us to deal with the "heavier" nodes (the ones with a higher degree) first, and in so doing break up the graph into smaller, independent sub-graphs.

Algorithm 1 Motif Counter Overview

```
function MotifCounter(g,level)
                                         ▷ g is a graph (either directed or undirected)
                              ▷ level is either 3 or 4, the level of motif to count
  global motifCounter ← list of motif counters
            ▷ A motif counter is a dictionary containing a counter for each motif
  sortdNodes ← g.vertices
  sort sortedNodes by node degree
  for node in sortedNodes do
    if level == 3 then
      CreateNodeSubtree3(g,node)
    else                                                          ▷ level is 4
      CreateNodeSubtree4(g,node)
    end if
    g.remove(node)
 ▷ We remove the node after iterating over it so as to not repeat the motifs
                                                   that were already seen.
  end for
end function
```

These two ideas give us the general shape of the algorithm, which can be seen, from above, as an iteration over the sorted nodes.

The *CreateNodeSubtree* function is where most of the hard work occurs, as that is the function where the actual motif counting is performed.

1.3 Motif combinatorics

Counting all motifs containing a given node is done using a **breadth first approach.** For each motif, we can define a *depth.* The depth of a motif is defined as the distance of the furthest vertex in the motif from the root node. This can also be though of as the furthest ring of neighbors that needs to be accessed to build the motif (for example, for a motif a depth 2 we must first select a vertex from the root's first neighbors, and then proceed to selecting a node from that vertex's neighbors, totaling two neighbor rings that we needed to see to build the motif). The second view is the one employed by us in implementing this algorithm. We separate the motifs by their *depth* and count all motifs of the same depth together.

It is obvious that the depth of a motif is bound by the number of nodes in the motif:

- A 3-motif can only be of depth 1 or 2
- A 4-motif can be of depth 1,2 (with two variations) or 3

In order to count all of the motifs of a certain depth, we must understand how each motif looks.

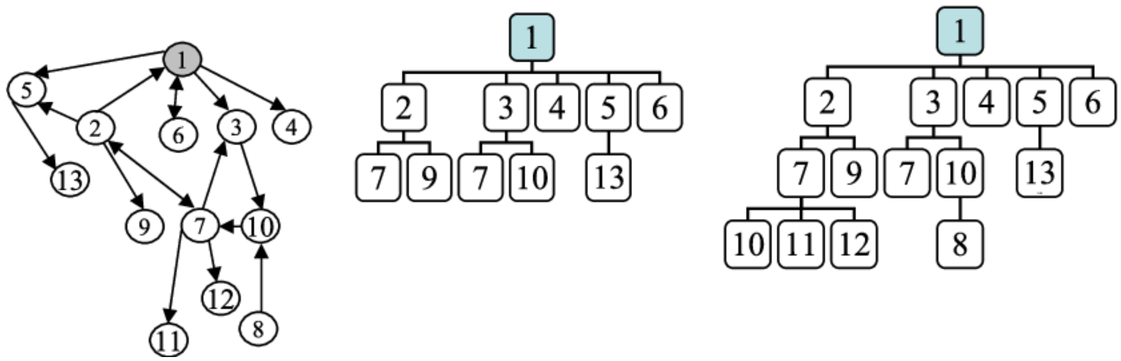


Figure 1: Node order
This figure describes the order of counting all motifs with node 1, which is also the node with the highest rank and so is the first node to be the root for motif counting.
From left to right: the graph itself, the order of counting 3-motifs and the order of counting 4-motifs.

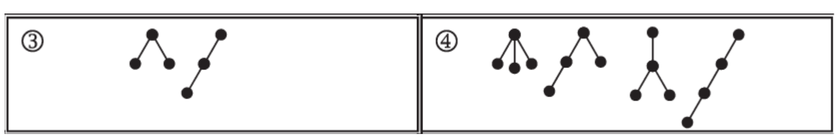


Figure 2: 3- and 4-motif variations
From left to right: 3-motif of depths 1and 2, 4-motifs of depth 1, 2 (option 1), 2 (option 2), and 3.

It can be seen that we can easily iterate over the necessary neighbors (whether they are first degree or second degree neighbors) to construct all the possible motifs containing the root vertex.

1.4 Subtree Construction

A *subtree* is considered to be the vicinity of the node where motifs containing is could appear. The $CreateNodeSubtree$ is responsible for going over all the possible depths and counting all the motifs in each. For readability, we have separated the depths into separate functions.

Algorithm 2 Create Node Subtree for 3-motifs

```
function CreateNodeSubtree3(g,root)
                              ▷ g is a graph (either directed or undirected)
                              ▷ root is the vertex that must be in all motifs
    for n_1 in root's neighbors do
        Mark that we saw n_1
    end for
    Count3MotifsDepth2(g,root)
    Count3MotifsDepth1(g,root)
end function
```

Algorithm 3 Create Node Subtree for 4-motifs

```
function CreateNodeSubtree4(g,root)
                              ▷ g is a graph (either directed or undirected)
                              ▷ root is the vertex that must be in all motifs
    for n_1 in root's first neighbors do
        Mark n_1 as seen at level 1
    end for
    Count4MotifsDepth1(g,root)
    Count4MotifsDepth2(g,root)
    Count4MotifsDepth3(g,root)
end function
```

It is important to note that when we say we iterate over all of a node's neighbors, we iterate over the neighbors in the *full graph*, i.e. we consider both the vertices connected to the node and the ones the node is connected to (which, in a directed graph, are not equivalent).

1.5 Motif Counting

1.5.1 Motif Counter updating

While each is slightly different, all motif counting function are essentially the same.

All of these functions, broadly speaking, have similar behavior:

- Iterate over all the motifs of the specified depth that you can build
  - For each of those motif update the relevant motif counters

The way of iterating over the motifs is the main difference between each of the functions.

Updating the motif counters is done by calculating the motif index (as discussed in the next section), and is relatively straightforward:

Algorithm 4 Updating Motif Counters

```
function UpdateMotifCount(g,motif)
                                  ▷ g is a graph (either directed or undirected)
                                    ▷ motif is the motif that is being updated
    motifIndex ← GetMotifIndex(g,motif)
    for n doode in motif
        motifCounters[node][motifIndex] + +
    end for
end function
```

1.5.2 3-Motif building algorithms

As we discussed in the previous section, 3-motifs can only be of depth 1 or 2.

For both depths, we know what their form looks like:

- A 3-motif of depth 1 must be the root node connected to two of it's first neighbors.
- A 3-motif of depth 2 can only be the root, a first neighbor and a second neighbor.

The in which the motifs are counted is critical, as it directly affects which conditions we need to check to avoid double counting. For 3-motifs, we count the motifs in the following order:

1. Motifs of depth 2; then
2. Motifs of depth 1

Algorithm 5 Motif 3 builders

```
function Count3MotifsDepth2(g,root)
                                  ▷ g is a graph (either directed or undirected)
                                  ▷ root is the vertex that must be in all motifs
    for n1 in root's neighbors do
        for n2 in n1's neighbors do
            if n2 is a neighbor of root's then
                if we saw n1 before n2 then          ▷ Count the pair only once
                    UpdateMotifCount(g,[root, n1, n2])
                end if
            else
                Mark that we saw n2
                UpdateMotifCount(g,[root, n1, n2])
            end if
        end for
    end for
end function


function Count3MotifsDepth1(g,root)
                                  ▷ g is a graph (either directed or undirected)
                                  ▷ root is the vertex that must be in all motifs
    neighborCombinations ← all 2-combinations of root's neighbors
    for tuple in neighborCombinations do
        n1 ← tuple.first
        n2 ← tuple.second
        if we saw n1 before n2 and they aren't neighbors then (*)
 ▷ We only want to count (n1, n2) once, and if they are neighbors, we would
              have discovered the motif as a motif of depth 2 (see figure 3)
            UpdateMotifCount(g,[root, n1, n2])
        end if
    end for
end function
```

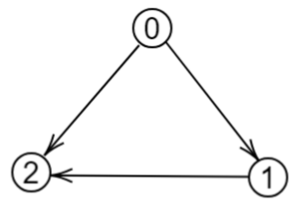


Figure 3: Motif 3 special cases

This is an example of a case where the condition marked **(*)** in $Count3MotifsDepth1$ stop us from double counting a motif.
The motif above can be counted as both a depth-1 and a depth-2 motif, both starting at node 0. depth 1: $0 \to 1$and $0 \to 2$., or depth 2: $0 \to 1 \to 2$.
As the motif is counted first as a depth 2 motif, the condition in the depth 1 counter stops the motif from being counted twice, since they are neighbors.

1.5.3 4-Motif Building algorithms

Like 3-motifs, we can separate the different 4-motifs into groups by their respective depths

- **Depth 1** are 4- motifs constructed from the root and three of it's first neighbors
- **Depth 2** can be one of two options:
  - The root, two of it's first neighbors and one of it's second neighbors (which is a neighbor of one of the first neighbors)
  - The root, one first neighbors, and two second neighbors who are connected to the first neighbor.
- **Depth 3** can only be a chain of the root, and a first, second and third neighbors.

The order between the depths is doubly important for 4-motifs. The order is as they were listed above. It is important to note that **all** motifs of a certain depth are counted before moving on to the next depth.

Algorithm 6 Motif 4 depth 1 and 2 builders

```
function COUNT4MOTIFSDEPTH1(g,root)
                                          ▷ g is a graph (either directed or undirected)
                                          ▷ root is the vertex that must be in all motifs
    neighborCombinations ← all 3-combinations of root's neighbors
    for tuple in neighborCombinations do
        n_11 ← tuple.first
        n_12 ← tuple.second
        n_13 ← tuple.third
        UPDATEMOTIFCOUNT(g,[root, n_11, n_12, n_13])
    end for
end function

function COUNT4MOTIFSDEPTH2(g,root)
                                          ▷ g is a graph (either directed or undirected)
                                          ▷ root is the vertex that must be in all motifs
    for n_1 in root's neighbors do
        for n_2 in n_1's neighbors do
            if n_2 was not seen at level 1 then
                Mark n_2 as seen at level 2
            end if

            ▷ Motifs of the form root, two first neighbors and a second neighbor
            for n_11 in root's neighbors do
                if n_2 was seen at level 2 and n_1 ≠ n_11 then (*)
                    edgeExists ← (n_2, n_11) ∈ E or (n_11, n_2) ∈ E
                    if not edgeExists or (edgeExists and n_1⟨n_11) then (**)
                                          ▷ If there is an edge between n_1 and n_11,
                                          we would have already counted the motif
                                          as a motif of depth 2. If no such edge exists,
                                          we only want to count the motif once.
                        UPDATEMOTIFCOUNT(g,[root, n_1, n_11, n_2])
                    end if
                end if
            end for
        end for
    end for

    ▷ Motifs of the form root, a first neighbors and two second neighbors
    secondNeighborCombinations ← all 2-combinations of n_1's neighbors
    for combination in secondNeighborCombinations do
        n_21 ← secondNeighborCombinations.first
        n_22 ← secondNeighborCombinations.second
        if n_21 and n_22 were seen at level 2 then (***)
    ▷ If both nodes were seen at level 2 then the motif wasn't counted as a
                                          depth 1 motif
            UPDATEMOTIFCOUNT(g,[root, n_1, n_21, n_22])
        end if
    end for
end function
```

Algorithm 7 Motif 4 depth 3 builder

```
function COUNT4MOTIFSDEPTH3(g,root)
                                              ▷ g and root are as before
    for n_1 in root's neighbors do
        for n_2 in n_1's neighbors do
            if n_2 was seen at level 1 then (†)
            ▷ If n_2 was seen at level 1 then we can't complete a depth-3 motif
                continue
            end if
            for n_3 in n_2's neighbors do
                if n_3 was not seen yet then
                    Mark n_3 as seen at level 3
                    if n_2 was seen at level 2 then
                        UPDATEMOTIFCOUNT(g,[root, n_1, n_2, n_3])
                    end if
                else
                    if n_3 was seen at level 1 then (††)
                     ▷ This motif was already counted as a depth-1 or -2 motif
                        continue
                    end if
                    edgeExists ← (n_1, n_3) ∈ E or (n_1, n_3) ∈ E
                    if n_3 was seen at level 2 and not edgeExists then († † †)
                                          ▷ If an edge exists we already counted
                                            this motif as a depth-1 or -2 motif
                        UPDATEMOTIFCOUNT(g,[root, n_1, n_2, n_3])
                    end if
                    if n_3 was seen at level 3 and n_2 was seen at level 2 then
                        UPDATEMOTIFCOUNT(g,[root, n_1, n_2, n_3])
                    end if
                end if
            end for
        end for
    end for
end function
```

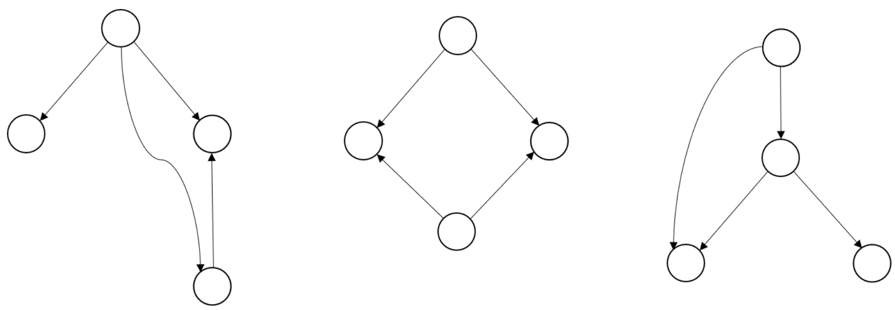


Figure 4: Motif 4 depth 2 special cases
There are three conditions we must check for 4-motifs of depth 2 so that they won't be double counted. The conditions in the code are marked with *****. From left to right, these motifs are examples of conditions **(*)** , **(**)** and **(***)**.

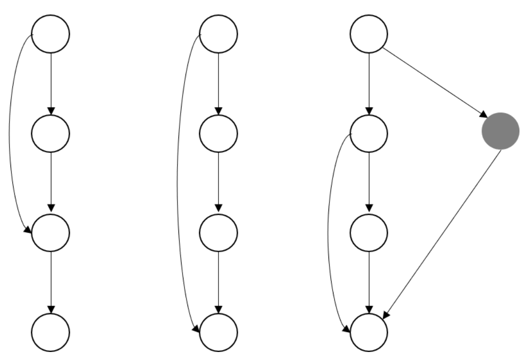


Figure 5: Motif 4 depth 3 special cases
There are three conditions we must check for 4-motifs of depth 3 so that they won't be double counted. The conditions in the code are marked with †. From left to right, these motifs are examples of conditions $(\dagger), (\dagger\dagger)$ and $(\dagger\dagger\dagger)$. The case of $(\dagger\dagger\dagger)$ is delicate. The gray node is one not included in the motif, but it causes n_3 to be marked as a level 2 node. By itself, this is not enough to stop us from counting the motif, for example if the internal link $(n_1 \to n_3)$ didn't exists. Only when the internal link exists can we be sure we have already counted the motif.

1.6 Motif representation

Another important idea contributing to the efficiency of this method is that we can generate a list of all possible motifs and their corresponding isomorphisims beforehand. We can then build a list of all motifs and the motif index they correspond to, and access that list at run-time. The full list can be found in appendix B

In order to build such a list we must be able to represent each motif in a fashion that is both unique and concise. The representation we use is that of a bit array of adjacency. Since each motif contains only 3 or 4 nodes,we can build a miniature adjacency matrix which represents the motif. If we then read the matrix as a single string of bits and convert it to an unsigned int - we have effectively represented the motif using a single, easy to interpret, number.

There are two important points to note about this representation method:

- The connections between a node and itself is not considered when transforming the adjacency matrix of the motif into a number. The reason for that is simply that we assume we are dealing with a graph **without self-loops**. As this is a common assumption, it is considered legitimate in this case.

- The way of building the adjacency matrix for directed and undirected graphs is slightly different. While for a motif in a directed graph a link of $a \rightarrow b$ does not necessarily imply a connection in the other direction ($b \rightarrow a$), In an **undirected** graph we do know this, and therefor can limit ourselves to writing only *half* of the adjacency matrix.

  Algorithmically, this distinction is implemented by using different combinatorical functions.

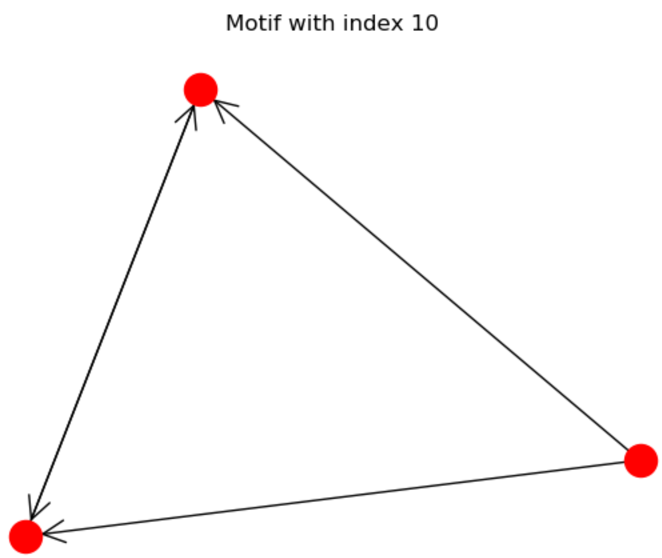
  - For a directed graph we use the *permutations* function
  - For an undirected graph we use the *combinations* function.

  For a further explanation about these functions, refer to Appendix A.

The Transformation itself consists of 4 steps.

1. The motif must be transformed into it's adjacency matrix.
2. The matrix is flattened into a bitstring.
3. The bitstring is read as an unsigned int representation and transformed into such a number.
4. The number is then looked up in the variations table to find the corresponding motif index.

The motif indices and their variations can be found in Appendix B.



$$matrix = \begin{pmatrix} - & 1 & 1 \\ 0 & - & 1 \\ 0 & 1 & - \end{pmatrix} \Rightarrow bitstring = 110101 \Rightarrow number = 53 \Rightarrow index = 10$$

Figure 6: Motif to number transformation example
An example of the entire motif transformation, from the motif to it's corresponding index.

Algorithm 8 Motif to number representation

```
function GetMotifIndex(g, motif)
                                        ▷ g is a graph (either directed or undirected)
                                        ▷ motif is the motif that is being represented
    edges is an empty list of booleans
    if g is directed then
        options ← permutations(motif)
    else
        options ← combinations(motif)
    end if
    for tuple in options do
        n_1 ← tuple.first
        n_2 ← tuple.second
        neighbors ← whether n_1 and n_2 are neighbors in g
        append neighbors to edges
    end for
    motifNumber ← the unsigned int represented by edges
    motifIndex ← the index corresponding to the motif number
                ▷ The index is read from the index file we created ahead of time
    return motifIndex
end function
```

1.7 Motif Visualization

To assist with both understanding and debugging, a motif visualization script was written. The script can be found under the *measure_tests* directory with the name *draw_motifs.py*.

Running the script is simply a matter of changing the values given in the *main* section on the bottom of the script. The scripts outputs an image using networkx and matplotlib of the motif with the given index. You can also change the motif level and toggle whether the graph is directional. An example output is the motif given in figure 6.

2 C++ Kernel

2.1 Overview

As we have reached the optimal algorithmic performance of motif counting (we count each motif only once), in order to facilitate analyzing motifs in larger graphs, we must write faster **code**.

Previously, the code to count the motifs has been written in Python. Python, while easy to use and develop code in, is inherently slow, as the language itself is interpreted. Faster code performance is obtained through the use of code in C++, both running on the CPU and on the GPU. Running the algorithms using a C++ kernel requires two additional changes:

1. The graph must be saved in a LIL (List of Lists) format.
2. The algorithm must be modified so as to that the nodes are not actually removed from the graph, as that is inefficient using this format.

2.2 Algorithm changes

2.2.1 Graph Format

In the C++ kernel, the graph is saved using a graph format which is commonly used for sparse graph, but can be used to our advantage in this case.

As our format is aimed at increasing the efficiency of our code by leveraging the computer's internal cache mechanism, we call this structure a CacheGraph.

The CacheGraph is composed of two arrays:

1. An array which is composed of all the neighborhood lists placed back to back (Neighbors).
2. An array which holds the starting index of each node's neighbor list (Indices).

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 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]

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. 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 processor.

2.2.2 Removal Index

One disadvantage of the CacheGraph is that it can't be modified at runtime, as that is an extremely costly operation. Instead, we will hold an array where each node has a corresponding "removal index", which will be the first iteration where the node no longer exists in the graph. As the algorithm removes each node immediately after it iterates it's subtree, computing the removal indices of all nodes is simply a matter of "flipping" the list which contains the order in which the nodes will be traversed, i.e. the nodes as ordered by their degree. "Flipping" the list entails saving each node's position in the nodes list as the value corresponding to that node in the removal indices array.

For example, if the nodes were traversed in this order : [2,0,1], then the corresponding removal indices array will be [1,2,0] (i.e. node 0 is removed at iteration 1, node 1 at iteration 2, etc.)

We now need to further modify our code to check each node we use, to verify that it's removal index of the node is lower than the removal index of the root of the current subtree.

2.2.3 GPU Changes

To further accelerate the motif counting code, a version of it was written for the GPU using the CUDA library. Writing the code for the GPU means that we will run all of the motif subtree calculations for each node in parallel, which in turn means we must be able to run each of those calculations independently of the others. Luckily, using the removal indices we can already make the subtrees independent of one another, as the removal indices were pre-computed.

As the programming model we can use on the GPU itself is quiet limited (in essence, we are restricted to basic C code i.e. we must use the malloc and free function together with pointers), and as we want to limit the amount of memory used, we must make further modifications to the code.

1. **GPU functions** are written outside the main bodies of code (MotifCalculator,MotifUtils, CacheGraph), as they cannot be member functions. They are instead re-written as stand-alone functions within the file.

2. **Global variables** are used to communicate between the class methods, which provide the CPU-side pre- and post-processing of the results, and the GPU functions.

3. **Atomic add** is used to update the motif counters, so that the GPU threads won't interfere with one another.

4. **All data** which is used in the GPU must be copied into GPU memory. Additionally, the data is prefetched asynchronously to improve the code's performance.

5. **Visited vertices** (which saves which nodes were already visited) is no longer a map, as those can't be used in the GPU. Instead, it is an matrix of size n^2 in which each row corresponds to one node. The type of the matrix changes depending on the motif level. For level 3 a matrix of booleans is used, and for level 4 a matrix of short ints.

6. **No vectors** can be used in the GPU. In order to provide a unified interface, CPU code at the end of the calculation converts the feature from a matrix to the vector of vectors returned by the CPU motif calculator.

Furthermore, the GPU's memory itself is limited, and so we can't run graphs of any size on it. The GPU code, because of it's parallel nature, requires n^2 memory space, and so is unsuited for large graphs.

2.3 GPU requirements

In order to run the GPU code, some requirements must be met, in both software and hardware.

- The GPU must be an NVIDIA GPU of compute capability 3.5 or higher.
- Version 8.0 of the CUDA Toolkit must be installed and in the PATH of the system. Higher versions may work, but were not tested.
- The GPU drivers must be compatible with the CUDA Toolkit.

Specifically, the code was tested on a system with the following specs:

- Centos 7 Linux distribution
- A GeForce GTX 1080 Ti with compute capability 6.1
- CUDA Toolkit version 8.0
- GPU drivers version 396.26

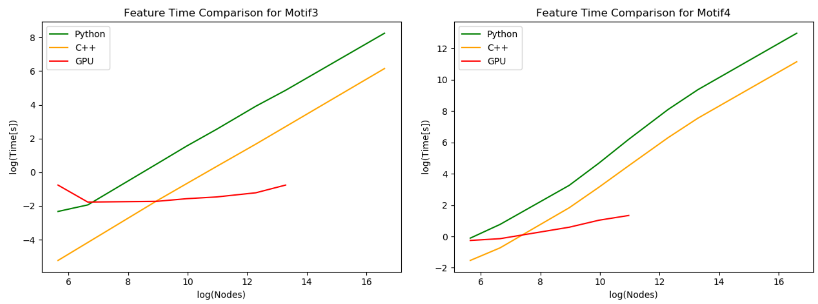


Figure 7: Run time comparison for different kernels
The benchmarks were ran on regular graphs with a regular degree of 20, which is why the graphs are linear in nature.

2.4 Performance comparison

The C++ kernel gives us code that runs several times faster than the original python code, and the GPU version gives us an additional 10X factor on top of the regular C++ code. Below are graph comparing the run-time of the various kernels on Erdos-renyi graphs of varying size. Figure 7 shows the comparisons between the various kernels. The raw benchmarks and code to generate the plots can be found in the repository features_algorithms/accelerated_graph_features/src/ accelerated_graph_features/benchmarks directory.

2.5 Future work

We are yet to implement one variation of the motif code (which exists in python). This variation is that of counting motifs for edges, rather than nodes.

This change requires minimal work, as the bulk of the code is already written for the node-based motif algorithm. Basically, the algorithm must accept the motifs found by the subtree functions and update the counters for each edge in the motif.

Further work could also include an implementation of the algorithm with the nodes used in chunks. Some of the time limitations we are faced with today could be solved if the code could be run on several CPUs/GPUs together. In order to enable this, the algorithm must be able to divide the nodes into smaller groups, perform the algorithm on each group and combine the results.

One more change that could possible increase the number of cases in which the GPU can be used is a sparse-graph implementation of the GPU algorithm. In contrast to the python/C++ algorithms, the GPU can't utilize expandable structures like dictionaries or hashmaps, instead relying on an array with a size of the number of nodes to store the neighbors of each node we iterate over, as we iterate over all nodes concurrently, this results in $O(n^2)$ memory. For sparse graphs, this approach is wasteful, and could be improved.

3 Useful resources

This section is intended for future researchers and developers who have an interest in understanding/modifying the code. The section will contain links to various resources that could help understand the code. Some of the resources are specifically related to the motif code, but most are more general resources, such as guides to understanding the components used in the GPU code.

3.1 General Requirements

Running the code in any version requires some basic python libraries. The recommended way to install them is through the Conda[2] package manager:

- **Networkx** is a graph library used to create, store and manipulate graphs, including those used as input to the motif counting algorithm. (All versions)
- **Bitstring** is used for the motif conversion. (Python only)
- **Numpy,** a general mathematical library, is used various calculations. (Python only)
- **Boost.Python** is a tool used to create the python interface with the C code. (C & GPU versions only).

Instructions for using the FeatureCalculator framework can be found at the repositories README[3].

3.2 C code

- **Installation instructions** as well as further information for developers about the C and GPU code can be found in GitHub repository: C++ Manual.
- **Makefiles** are used for the compilation of the C & GPU code. An introduction to makefiles can be found at the GNU site[4].
- **Boost** is used as a framework for linking the C and python code. For more on boost, see their official website[5]. Notice that we install boost using Conda and that we use a custom build system (via Makefile).

[2]https://conda.io/en/latest/
[3]https://github.com/oricc/graph_measures/blob/master/README.md
[4]https://www.gnu.org/software/make/manual/html_node/Introduction.html
[5]https://www.boost.org/

3.3 GPU code

Diving into GPU programming requires understanding several key concepts, following provide a simple explanation of the most important of them.

- NVIDIA developers
  - **Introduction to CUDA :** https://devblogs.nvidia.com/even-easier-introduction-cuda/.
  - **Unified memory** is a method in which we can allocate memory both the CPU and GPU can access: https://devblogs.nvidia.com/unified-memory-cuda-beginners/https://devblogs.nvidia.com/unified-memory-cuda-beginners/.
- **CUDA Programming Guide** is an extensive overview of every aspect of the CUDA programming model, including explanations, examples and best practices. While long, this is the go-to handbook when writing GPU code, and is highly recommended. The guide can be found here: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

A Combinatorical functions

A.1 Permutations

The *permutations* function is used to create all the possible tuples of a given size which are comprised of elements of the original set **with respect to order**. In our case we only use 2-permutations, which give us all the possible edges that can exist between a given set of nodes.

For example, assume our motif is comprised of nodes 1,2 and 3. The possible 2-permutations in this case would be:

$$perm(\{1,2,3\}) = \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\}$$

And since the order matters, we can see that, for example, both $(1,2)$ and $(2,1)$ appear in the resulting list.

A more intuitive way to view this function, and the one employed by us, is as looking at all of the elements of a matrix, without the main diagonal:

$$\begin{matrix} 0 & X & X \\ X & 0 & X \\ X & X & 0 \end{matrix}$$

Where X is an element accessed by the permutations, 0 is one which was not.

A.2 Combinations

The *combinations* function is very similar to the *permutations* function in that it gives us a list of tuples derived from the original sets, but differs from it in that it **does not respect order**, and so only gives us a single instance of each tuple:

$$comb(\{1,2,3\}) = \{(1,2),(1,3),(2,3)\}$$

In the context of adjacency matrices, the function only iterated over the top half of the matrix:

$$\begin{matrix} 0 & X & X \\ 0 & 0 & X \\ 0 & 0 & 0 \end{matrix}$$

B Motif variations

A core part of this algorithm is it's ability to use a precomputed table of the various motif variations and their corresponding motif index. As was discussed, each motif is represented using a single integer which is read from it's representing bit matrix. The following tables map each such integer to a "motif index", which is shared among all isomorphic motifs.

There are four tables in total, for 3- and 4-motifs in both directed and undirected cases. The code for generating these tables, along with pickle files containing the dictionaries which can be used in the python code can be found in the GitHub repository[6].

The actual implementation of the tables is reverse than the one displayed here. In the code, each motif number is mapped to it's index for quick access. The following tables are meant for quick reference and some examples, and so each index is mapped to all of the motif numbers corresponding to it.

B.1 3-Motif Variations

B.1.1 Undirected

0	3,5,6
1	7
None	0,1,2,4

B.1.2 Directed

0	3,12,48
1	6,9,17,24,34,36
2	7,13,19,44,50,56
3	10,20,33
4	11,14,28,35,49,52
5	15,51,60
6	21,22,26,37,41,42
7	23,45,58
8	25,38
9	27,29,39,46,54,57
10	30,43,53
11	31,47,55,59,61,62
12	63
None	0,1,2,4,5,8,16,18,32,40

[6]https://github.com/oricc/graph_measures/tree/master/features_algorithms/motif_variations

B.2 4-Motif Variations

B.2.1 Undirected

0	7,25,42,52
1	13,14,19,22,26,28,35,37,41,44,49,50
2	15,23,27,29,39,43,46,53,54,57,58,60
3	30,45,51
4	31,47,55,59,61,62
5	63
None	0,1,2,3,4,5,6,8,9,10,11,12,16,17,18,20,21,24,32,33,34,36,38,40,48,56

B.2.2 Directed

0	7,56,448,3584
1	14,49,69,168,336,386,515,1048,1792,2240,2592,3076
2	15,57,71,184,450,464,519,1080,2496,3588,3616,3840
3	21,28,35,42,112,134,196,224 259,280,321,392,560,896,1030,1344 1538,1552,2053,2088,2561,2688,3080,3136
4	22,38,50,52,104,133,261,296 328,352,385,388,536,704,1027,1216 1600,1664,2051,2072,2568,2576,3073,3074
5	23,39,58,60,120,135,263,312 449,452,456,480,568,960,1031,1472 2055,2104,3585,3586,3592,3600,3648,3712
6	29,43,198,240,323,408,1542,1584,2565,2944,3112,3392
7	30,46,51,53,197,232,325,344 368,387,390,424,1539,1560,1856,1920 2563,2600,2608,2752,3077,3078,3096,3264
8	31,47,59,61,199,248,327,440 451,454,472,496,1543,1592,2567,3008 3128,3520,3589,3590,3624,3632,3904,3968
9	54,360,389,1728,2584,3075
10	55,62,376,391,453,488,1984,2616,3079,3587,3608,3776
11	63,455,504,3591,3640,4032
12	76,161,274,522,577,800,1041,1160,1284,2084,2128,2178

13	77,78,169,177,338,402,523,526 579,581,1049,1073,1176,1192,1796,1824 2242,2256,2384,2434,2596,2848,3108,3332
14	79,185,466,527,583,1081,1208,2498,2512,3620,3844,3872
15	84,97,140,162,266,273,529,546 642,672,769,784,1034,1044,1092,1104 1282,1288,2060,2081,2113,2144,2180,2184
16	85,113,142,170,337,394,547,561 771,898,1038,1052,1304,1360,1794,1808 2117,2216,2244,2272,2593,2720,3084,3140
17	86,105,141,178,330,401,537,550 706,773,1035,1076,1232,1320,1604,1696 2115,2200,2400,2436,2572,2832,3105,3330
18	87,121,143,186,458,465,551,569 775,962,1039,1084,1336,1488,2119,2232 2500,2528,3596,3617,3652,3744,3842,3856
19	92,163,204,225,275,282,554,816 833,928,1045,1286,1348,1416,1546,1553 2085,2092,2160,2182,2625,2690,3152,3208
20	93,171,206,241,339,410,555,835 1053,1432,1550,1585,1798,1840,2246,2288 2597,2629,2946,2976,3116,3240,3396,3408
21	94,179,205,233,346,403,558,837 1077,1448,1547,1561,1860,1952,2416,2438 2604,2627,2754,2864,3109,3224,3280,3334
22	95,187,207,249,467,474,559,839 1085,1464,1551,1593,2502,2544,2631,3010 3256,3536,3621,3628,3846,3888,3908,4000
23	99,156,212,226,267,281,533,646 688,912,1066,1094,1136,1346,1556,1570 2061,2089,2369,2440,2692,2817,3168,3336

24	100,164,268,276,289,290,530,592 608,641,648,776,1042,1089,1096,1154 1156,1281,2058,2065,2068,2082,2120,2177
25	101,172,270,305,340,418,531,643 680,848,1050,1093,1112,1410,1793,1800 2062,2097,2241,2248,2594,2656,3092,3204
26	102,180,269,297,354,404,534,645 664,720,1074,1091,1128,1218,1632,1668 2059,2073,2376,2433,2580,2824,3106,3329
27	103,188,271,313,468,482,535,647 696,976,1082,1095,1144,1474,2063,2105 2497,2504,3604,3618,3680,3716,3841,3848
28	106,114,149,150,329,393,540,564 708,900,1059,1062,1248,1376,1602,1680 2307,2309,2328,2344,2569,2704,3081,3138
29	107,157,214,242,331,409,541,710 1067,1264,1574,1588,1606,1712,2371,2456 2573,2821,2948,2960,3113,3368,3394,3424
30	108,165,278,306,332,417,538,705 808,864,1043,1224,1285,1412,1601,1672 2086,2100,2136,2179,2570,2640,3089,3202
31	109,173,334,342,433,434,539,707 1051,1240,1605,1704,1797,1832,2243,2264 2574,2598,2896,2912,3121,3124,3458,3460
32	110,181,333,370,406,425,542,709 1075,1256,1603,1688,1888,1924,2392,2435 2571,2612,2768,2856,3097,3110,3266,3333
33	111,189,335,441,470,498,543,711 1083,1272,1607,1720,2499,2520,2575,3024 3129,3522,3622,3636,3845,3880,3936,3972
34	115,158,213,234,345,395,565,902 1070,1392,1564,1571,1858,1936,2373,2472 2601,2736,2756,2819,3085,3142,3296,3352

35	116,166,277,298,353,396,562,736 792,897,1046,1220,1283,1352,1616,1666 2076,2083,2152,2181,2577,2696,3082,3137
36	117,174,341,369,398,426,563,899 1054,1368,1795,1816,1872,1922,2245,2280 2595,2609,2728,2784,3086,3100,3141,3268
37	118,182,361,362,397,405,566,901 1078,1384,1730,1732,1744,1760,2408,2437 2585,2588,2712,2840,3083,3107,3139,3331
38	119,190,377,399,469,490,567,903 1086,1400,1986,2000,2501,2536,2617,2744 3087,3143,3612,3619,3780,3808,3843,3864
39	122,151,457,572,964,1063,1504,2311,2360,3593,3650,3728
40	123,159,215,250,459,473,573,966 1071,1520,1575,1596,2375,2488,2823,3012 3384,3552,3597,3625,3654,3760,3906,3984
41	124,167,279,314,460,481,570,824 961,992,1047,1287,1476,1480,2087,2108 2168,2183,3594,3601,3649,3664,3714,3720
42	125,175,343,442,462,497,571,963 1055,1496,1799,1848,2247,2296,2599,3040 3132,3524,3598,3633,3653,3752,3920,3970
43	126,183,378,407,461,489,574,965 1079,1512,1988,2016,2424,2439,2620,2872 3111,3335,3595,3609,3651,3736,3778,3792
44	127,191,463,471,505,506,575,967 1087,1528,2503,2552,3599,3623,3641,3644 3655,3768,3847,3896,4034,4036,4048,4064
45	220,227,283,944,1350,1557,1578,2093,2694,2881,3184,3464
46	221,222,235,243,347,411,1565,1579 1582,1589,1862,1968,2605,2758,2883,2885 2950,2992,3117,3312,3398,3440,3480,3496
47	223,251,475,1583,1597,2887,3014,3512,3568,3629,3910,4016
48	228,284,291,624,904,1158,1345,1554,2069,2090,2689,3144

49	229,236,286,307,348,419,880,936 1349,1414,1555,1562,1857,1928,2094,2101 2602,2672,2691,2753,3093,3160,3206,3272
50	230,244,285,299,355,412,752,920 1222,1347,1558,1586,1648,1670,2077,2091 2581,2693,2945,2952,3114,3176,3393,3400
51	231,252,287,315,476,483,952,1008 1351,1478,1559,1594,2095,2109,2695,3009 3192,3528,3605,3626,3696,3718,3905,3976
52	237,350,435,1563,1861,1960,2606,2755,2928,3125,3288,3462
53	238,245,349,371,414,427,1566,1587 1859,1904,1926,1944,2603,2613,2757,2800 2947,2984,3101,3118,3270,3304,3397,3416
54	239,253,351,443,478,499,1567,1595 1863,1976,2607,2759,3011,3056,3133,3320 3526,3544,3630,3637,3909,3952,3974,4008
55	246,363,413,1590,1734,1776,2589,2949,2968,3115,3395,3432
56	247,254,379,415,477,491,1591,1598 1990,2032,2621,2951,3000,3013,3119,3399 3448,3560,3613,3627,3782,3824,3907,3992
57	255,479,507,1599,3015,3576,3631,3645,3911,4024,4038,4080
58	292,584,1153,2066
59	293,294,300,308,356,420,600,616 712,840,1155,1157,1217,1409,1608,1665 2067,2070,2074,2098,2578,2632,3090,3201
60	295,316,484,632,968,1159,1473,2071,2106,3602,3656,3713
61	301,358,436,728,1219,1640,1669,2075,2582,2888,3122,3457
62	302,309,357,372,422,428,744,856 1221,1411,1624,1667,1864,1921,2078,2099 2579,2610,2664,2760,3094,3098,3205,3265
63	303,317,359,444,486,500,760,984 1223,1475,1656,1671,2079,2107,2583,3016 3130,3521,3606,3634,3688,3717,3912,3969
64	310,364,421,872,1413,1729,1736,2102,2586,2648,3091,3203

65	311,318,380,423,485,492,888,1000 1415,1477,1985,1992,2103,2110,2618,2680 3095,3207,3603,3610,3672,3715,3777,3784
66	319,487,508,1016,1479,2111,3607,3642,3704,3719,4033,4040
67	365,366,374,429,437,438,1731,1733 1752,1768,1896,1925,2587,2590,2614,2776 2904,2920,3099,3123,3126,3267,3459,3461
68	367,445,502,1735,1784,2591,3032,3131,3523,3638,3944,3973
69	373,430,1880,1923,2611,2792,3102,3269
70	375,381,431,446,494,501,1912,1927 1987,2008,2615,2619,2808,3048,3103,3134 3271,3525,3614,3635,3781,3816,3928,3971
71	382,439,493,1989,2024,2622,2936,3127,3463,3611,3779,3800
72	383,447,495,503,509,510,1991,2040 2623,3064,3135,3527,3615,3639,3643,3646 3783,3832,3960,3975,4035,4037,4056,4072
73	511,3647,4039,4088
74	585,586,588,804,1161,1169,1185,1316,2130,2194,2322,2340
75	587,589,590,1177,1193,1201,1828,2258,2386,2450,2852,3364
76	591,1209,2514,3876
77	593,609,650,673,778,786,801,802 1100,1105,1162,1164,1292,1297,1298,1300 2124,2129,2132,2148,2186,2209,2210,2212
78	594,612,649,780,1097,1170,1188,1313,2122,2193,2324,2338
79	595,613,651,681,782,850,1101,1113 1178,1196,1329,1426,1804,1825,2126,2225 2250,2257,2388,2466,2660,2850,3236,3348
80	596,610,652,657,658,676,777,788 1098,1106,1121,1124,1172,1186,1289,1314 2121,2146,2185,2196,2314,2316,2321,2337
81	597,611,654,689,779,914,1102,1137 1180,1194,1305,1362,1812,1826,2125,2217 2260,2274,2385,2442,2724,2849,3172,3340

82	598,614,653,665,722,781,1099,1129 1202,1204,1234,1321,1636,1700,2123,2201 2378,2402,2449,2452,2828,2836,3361,3362
83	599,615,655,697,783,978,1103,1145 1210,1212,1337,1490,2127,2233,2506,2513 2516,2530,3684,3748,3852,3860,3873,3874
84	601,617,714,805,806,842,1163,1165 1233,1324,1332,1425,1612,1697,2131,2134 2202,2226,2404,2468,2636,2834,3233,3346
85	602,620,713,812,844,868,1171,1189 1225,1317,1441,1444,1609,1673,2138,2195 2326,2342,2354,2356,2634,2642,3217,3218
86	603,621,715,846,1179,1197,1241,1457 1613,1705,1829,1836,2259,2266,2390,2482 2638,2854,2898,2916,3249,3380,3474,3492
87	604,618,716,820,841,932,1173,1187 1249,1318,1380,1417,1610,1681,2162,2198 2323,2330,2341,2348,2633,2706,3154,3209
88	605,619,718,843,1181,1195,1265,1433 1614,1713,1830,1844,2262,2290,2387,2458 2637,2853,2962,2980,3241,3372,3410,3428
89	606,622,717,845,1203,1205,1257,1449 1611,1689,1892,1956,2394,2418,2451,2454 2635,2770,2860,2868,3225,3282,3365,3366
90	607,623,719,847,1211,1213,1273,1465 1615,1721,2515,2518,2522,2546,2639,3026 3257,3538,3877,3878,3884,3892,3940,4004
91	625,803,906,1166,1308,1361,1810,2133,2218,2276,2721,3148
92	626,628,740,796,905,908,1174,1190 1252,1315,1353,1377,1618,1682,2154,2197 2325,2332,2339,2346,2697,2705,3145,3146

93	627,629,907,910,1182,1198,1369,1393 1820,1827,1874,1938,2261,2282,2389,2474 2729,2737,2788,2851,3149,3150,3300,3356
94	630,909,1206,1385,1746,1764,2410,2453,2713,2844,3147,3363
95	631,911,1214,1401,2002,2517,2538,2745,3151,3812,3868,3875
96	633,807,970,1167,1340,1489,2135,2234,2532,3660,3745,3858
97	634,636,828,969,972,996,1175,1191 1319,1481,1505,1508,2170,2199,2327,2343 2362,2364,3657,3658,3666,3721,3729,3730
98	635,637,971,974,1183,1199,1497,1521 1831,1852,2263,2298,2391,2490,2855,3044 3388,3556,3661,3662,3753,3761,3922,3986
99	638,973,1207,1513,2020,2426,2455,2876,3367,3659,3737,3794
100	639,975,1215,1529,2519,2554,3663,3769,3879,3900,4050,4068
101	659,684,852,1114,1125,1442,1801,2249,2318,2353,2658,3220
102	661,662,668,692,724,916,1123,1126 1130,1138,1250,1378,1634,1684,2315,2317 2329,2345,2377,2441,2708,2825,3170,3337
103	663,700,980,1127,1146,1506,2319,2361,2505,3682,3732,3849
104	666,677,721,790,809,866,1107,1132 1226,1293,1330,1428,1633,1676,2137,2150 2187,2228,2380,2465,2644,2826,3234,3345
105	667,685,723,854,1115,1133,1242,1458 1637,1708,1805,1833,2251,2265,2382,2481 2662,2830,2900,2914,3252,3377,3476,3490
106	669,726,1131,1266,1638,1716,2379,2457,2829,2964,3369,3426
107	670,693,725,918,1134,1139,1258,1394 1635,1692,1890,1940,2381,2393,2443,2473 2740,2772,2827,2857,3174,3298,3341,3353
108	671,701,727,982,1135,1147,1274,1522 1639,1724,2383,2489,2507,2521,2831,3028 3385,3554,3686,3764,3853,3881,3938,3988
109	674,785,1108,1290,2145,2188

110	675,682,787,817,849,930,1109,1116 1294,1306,1364,1418,1802,1809,2149,2161 2190,2220,2252,2273,2657,2722,3156,3212
111	678,690,738,789,793,913,1110,1140 1236,1291,1322,1354,1620,1698,2147,2153 2189,2204,2401,2444,2700,2833,3169,3338
112	679,698,791,825,977,994,1111,1148 1295,1338,1482,1492,2151,2169,2191,2236 2508,2529,3668,3681,3724,3746,3850,3857
113	683,851,1117,1434,1806,1841,2254,2289,2661,2978,3244,3412
114	686,691,853,915,1118,1141,1370,1450 1803,1817,1876,1954,2253,2281,2417,2446 2659,2732,2786,2865,3173,3228,3284,3342
115	687,699,855,979,1119,1149,1466,1498 1807,1849,2255,2297,2510,2545,2663,3042 3260,3540,3685,3756,3854,3889,3924,4002
116	694,917,1142,1386,1748,1762,2409,2445,2716,2841,3171,3339
117	695,702,919,981,1143,1150,1402,1514 2004,2018,2425,2447,2509,2537,2748,2873 3175,3343,3683,3740,3796,3810,3851,3865
118	703,983,1151,1530,2511,2553,3687,3772,3855,3897,4052,4066
119	729,730,813,870,1227,1235,1325,1460 1641,1644,1677,1701,2139,2203,2406,2484 2646,2838,2890,2892,3250,3378,3473,3489
120	731,1243,1645,1709,1837,2267,2894,2902,2918,3505,3506,3508
121	732,948,1251,1382,1642,1685,2331,2349,2710,2889,3186,3465
122	733,734,1259,1267,1643,1646,1693,1717 1894,1972,2395,2459,2774,2861,2891,2893 2966,2996,3314,3373,3430,3442,3481,3497
123	735,1275,1647,1725,2523,2895,3030,3513,3570,3885,3942,4020
124	737,794,810,818,865,929,1228,1299 1301,1302,1356,1420,1617,1674,2140,2156 2164,2211,2213,2214,2641,2698,3153,3210

125	739,795,945,946,1244,1307,1358,1366 1621,1706,1813,1834,2157,2221,2268,2275 2702,2726,2897,2913,3185,3188,3466,3468
126	741,798,882,937,1260,1331,1357,1430 1619,1690,1889,1932,2158,2229,2396,2467 2676,2699,2769,2858,3161,3238,3274,3349
127	742,754,797,921,1238,1268,1323,1355 1622,1652,1702,1714,2155,2205,2403,2460 2701,2837,2956,2961,3177,3370,3402,3425
128	743,799,953,1010,1276,1339,1359,1494 1623,1722,2159,2237,2524,2531,2703,3025 3193,3530,3700,3750,3861,3882,3937,3980
129	745,814,858,869,1229,1333,1427,1452 1625,1675,1868,1953,2142,2227,2420,2470 2643,2668,2762,2866,3226,3237,3281,3350
130	746,821,857,934,1237,1326,1396,1419 1628,1699,1866,1937,2163,2206,2405,2476 2665,2738,2764,2835,3158,3213,3297,3354
131	747,859,1245,1435,1629,1707,1838,1845 1870,1969,2270,2291,2669,2766,2899,2917 2982,2994,3245,3313,3414,3444,3482,3500
132	748,860,884,940,1253,1381,1443,1446 1626,1683,1865,1929,2334,2350,2355,2357 2666,2674,2707,2761,3162,3221,3222,3273
133	749,862,1261,1459,1627,1691,1869,1893 1961,1964,2398,2483,2670,2763,2771,2862 2930,2932,3253,3289,3290,3381,3478,3494
134	750,861,1269,1451,1630,1715,1867,1908 1945,1958,2419,2462,2667,2765,2802,2869 2963,2988,3229,3286,3305,3374,3418,3429
135	751,863,1277,1467,1631,1723,1871,1977 2526,2547,2671,2767,3027,3058,3261,3321 3542,3546,3886,3893,3941,3956,4006,4012

136	753,811,867,922,1230,1309,1363,1436 1649,1678,1814,1842,2141,2219,2278,2292 2645,2725,2954,2977,3180,3242,3404,3409
137	755,923,1246,1371,1653,1710,1821,1835 1878,1970,2269,2283,2733,2790,2901,2915 2958,2993,3181,3316,3406,3441,3484,3498
138	756,924,1254,1379,1650,1686,2333,2347,2709,2953,3178,3401
139	757,926,1262,1395,1651,1694,1891,1906 1942,1948,2397,2475,2741,2773,2804,2859 2955,2985,3182,3302,3306,3357,3405,3417
140	758,925,1270,1387,1654,1718,1750,1766 1778,1780,2411,2461,2717,2845,2957,2965 2969,2972,3179,3371,3403,3427,3433,3434
141	759,927,1278,1403,1655,1726,2006,2034 2525,2539,2749,2959,3001,3029,3183,3407 3449,3562,3814,3828,3869,3883,3939,3996
142	761,815,871,986,1231,1341,1468,1491 1657,1679,2143,2235,2534,2548,2647,3018 3258,3537,3692,3749,3862,3890,3916,4001
143	762,829,985,998,1239,1327,1483,1524 1660,1703,2171,2207,2407,2492,2839,3020 3386,3553,3670,3689,3725,3762,3914,3985
144	763,987,1247,1499,1661,1711,1839,1853 2271,2299,2903,2919,3022,3046,3514,3516 3569,3572,3693,3757,3918,3926,4017,4018
145	764,956,988,1012,1255,1383,1507,1510 1658,1687,2335,2351,2363,2365,2711,3017 3194,3529,3690,3698,3733,3734,3913,3977
146	765,990,1263,1523,1659,1695,1895,1980 2399,2491,2775,2863,3019,3060,3322,3389 3545,3558,3694,3765,3917,3954,3990,4009

147	766,989,1271,1515,1662,1719,2022,2036 2427,2463,2877,2967,3004,3021,3375,3431 3450,3561,3691,3741,3798,3826,3915,3993
148	767,991,1279,1531,1663,1727,2527,2555 3023,3031,3577,3578,3695,3773,3887,3901 3919,3943,4025,4028,4054,4070,4082,4084
149	819,881,931,938,1310,1365,1372,1422 1811,1818,1873,1930,2165,2222,2277,2284 2673,2723,2730,2785,3157,3164,3214,3276
150	822,873,874,933,1334,1388,1421,1429 1738,1740,1745,1761,2166,2230,2412,2469 2649,2652,2714,2842,3155,3211,3235,3347
151	823,889,935,1002,1342,1404,1423,1493 1994,2001,2167,2238,2533,2540,2681,2746 3159,3215,3676,3747,3788,3809,3859,3866
152	826,993,1303,1484,2172,2215,3665,3722
153	827,954,995,1009,1311,1367,1486,1500 1815,1850,2173,2223,2279,2300,2727,3041 3196,3532,3669,3697,3726,3754,3921,3978
154	830,890,997,1001,1335,1431,1485,1516 1996,2017,2174,2231,2428,2471,2684,2874 3239,3351,3667,3673,3723,3738,3786,3793
155	831,999,1017,1018,1343,1487,1495,1532 2175,2239,2535,2556,3671,3705,3708,3727 3751,3770,3863,3898,4042,4044,4049,4065
156	875,1437,1742,1777,1846,2294,2653,2970,2981,3243,3411,3436
157	876,1445,1737,2358,2650,3219
158	877,878,1453,1461,1739,1741,1753,1769 1900,1957,2422,2486,2651,2654,2778,2870 2906,2924,3227,3251,3283,3382,3475,3493
159	879,1469,1743,1785,2550,2655,3034,3259,3539,3894,3948,4005
160	883,939,1373,1438,1822,1843,1875,1905 1934,1946,2286,2293,2677,2731,2789,2801 2979,2986,3165,3246,3278,3308,3413,3420

161	885,942,1397,1454,1881,1882,1884,1931 1939,1955,2421,2478,2675,2739,2793,2794 2796,2867,3166,3230,3277,3285,3301,3358
162	886,941,1389,1462,1747,1754,1765,1772 1897,1933,2414,2485,2678,2715,2777,2846 2908,2922,3163,3254,3275,3379,3477,3491
163	887,943,1405,1470,1913,1935,2003,2010 2542,2549,2679,2747,2809,3050,3167,3262 3279,3541,3813,3820,3870,3891,3932,4003
164	891,1003,1439,1501,1847,1854,1998,2033 2295,2302,2685,2983,3002,3045,3247,3415 3452,3564,3677,3755,3790,3825,3923,3994
165	892,1004,1447,1509,1993,2359,2366,2682,3223,3674,3731,3785
166	893,1006,1455,1525,1916,1959,1995,2009 2423,2494,2683,2810,2871,3052,3231,3287 3390,3557,3678,3763,3789,3817,3930,3987
167	894,1005,1463,1517,1997,2021,2025,2028 2430,2487,2686,2878,2938,2940,3255,3383 3479,3495,3675,3739,3787,3795,3801,3802
168	895,1007,1471,1533,1999,2041,2551,2558 2687,3066,3263,3543,3679,3771,3791,3833 3895,3902,3964,4007,4051,4058,4069,4076
169	947,1374,1819,1877,1962,2285,2734,2787,2929,3189,3292,3470
170	949,950,1390,1398,1749,1756,1763,1770 1898,1941,2413,2477,2718,2742,2780,2843 2905,2921,3187,3190,3299,3355,3467,3469
171	951,1406,2005,2026,2541,2750,2937,3191,3471,3804,3811,3867
172	955,1011,1375,1502,1823,1851,1879,1978 2287,2301,2735,2791,3043,3057,3197,3324 3534,3548,3701,3758,3925,3953,3982,4010
173	957,1014,1391,1526,1751,1767,1786,1788 2415,2493,2719,2847,3033,3036,3195,3387 3531,3555,3702,3766,3945,3946,3981,3989

174	958,1013,1399,1518,1914,1943,2012,2019 2429,2479,2743,2812,2875,3049,3198,3303 3359,3533,3699,3742,3797,3818,3929,3979
175	959,1015,1407,1534,2007,2042,2543,2557 2751,3065,3199,3535,3703,3774,3815,3836 3871,3899,3961,3983,4053,4060,4067,4074
176	1019,1503,1855,2303,3047,3580,3709,3759,3927,4026,4046,4081
177	1020,1511,2367,3706,3735,4041
178	1021,1022,1519,1527,2023,2044,2431,2495 2879,3068,3391,3559,3707,3710,3743,3767 3799,3834,3962,3991,4043,4045,4057,4073
179	1023,1535,2559,3711,3775,3903,4047,4055,4071,4089,4090,4092
180	1755,1773,1901,1965,2779,2910,2926,2934,3291,3507,3509,3510
181	1757,1771,1902,1973,2782,2907,2925,2998,3315,3446,3483,3501
182	1758,1774,1779,1781,1899,1910,1949,1974 2781,2806,2909,2923,2971,2974,2989,2997 3307,3318,3419,3437,3438,3443,3485,3499
183	1759,1775,1787,1789,1903,1981,2783,2911 2927,3035,3038,3062,3323,3515,3517,3547 3571,3574,3949,3950,3958,4013,4021,4022
184	1782,2973,3435
185	1783,1790,2038,2975,3005,3037,3439,3451,3563,3830,3947,3997
186	1791,3039,3579,3951,4029,4086
187	1883,1885,1886,1909,1947,1963,1966,1971 2795,2797,2798,2803,2931,2933,2990,2995 3293,3294,3309,3317,3422,3445,3486,3502
188	1887,1979,2799,3059,3325,3550,3957,4014
189	1907,1950,2805,2987,3310,3421
190	1911,1915,1951,1982,2014,2035,2807,2813 2991,3003,3051,3061,3311,3326,3423,3453 3549,3566,3822,3829,3933,3955,3998,4011
191	1917,1967,2011,2811,2935,3054,3295,3518,3573,3821,3934,4019
192	1918,1975,2013,2027,2030,2037,2814,2939 2941,2999,3006,3053,3319,3447,3454,3487 3503,3565,3805,3806,3819,3827,3931,3995

193	1919,1983,2015,2043,2815,3055,3063,3067 3327,3551,3581,3582,3823,3837,3935,3959 3965,4015,4027,4030,4062,4078,4083,4085
194	2029,2942,3511,3803
195	2031,2045,2943,3070,3519,3575,3807,3835,3966,4023,4059,4077
196	2039,2046,3007,3069,3455,3567,3831,3838,3963,3999,4061,4075
197	2047,3071,3583,3839,3967,4031,4063,4079,4087,4091,4093,4094
198	4095
None	0,1,2,3,4,5,6,8 9,10,11,12,13,16,17,18 19,20,24,25,26,27,32,33 34,36,37,40,41,44,45,48 64,65,66,67,68,70,72,73 74,75,80,81,82,83,88,89 90,91,96,98,128,129,130,131 132,136,137,138,139,144,145,146 147,148,152,153,154,155,160,176 192,193,194,195,200,201,202,203 208,209,210,211,216,217,218,219 256,257,258,260,262,264,265,272 288,304,320,322,324,326,384,400 416,432,512,513,514,516,517,518 520,521,524,525,528,532,544,545 548,549,552,553,556,557,576,578 580,582,640,644,656,660,768,770 772,774,832,834,836,838,1024,1025 1026,1028,1029,1032,1033,1036,1037,1040 1056,1057,1058,1060,1061,1064,1065,1068 1069,1072,1088,1090,1120,1122,1152,1168 1184,1200,1280,1296,1312,1328,1408,1424 1440,1456,1536,1537,1540,1541,1544,1545 1548,1549,1568,1569,1572,1573,1576,1577 1580,1581,2048,2049,2050,2052,2054,2056 2057,2064,2080,2096,2112,2114,2116,2118 2176,2192,2208,2224,2304,2305,2306,2308 2310,2312,2313,2320,2336,2352,2368,2370 2372,2374,2432,2448,2464,2480,2560,2562 2564,2566,2624,2626,2628,2630,2816,2818 2820,2822,2880,2882,2884,2886,3072,3088 3104,3120,3200,3216,3232,3248,3328,3344