# Motif Counting Algorithm

Ori Cohen

February 21, 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.

## Contents

## List of Algorithms

# 1 Algorithm Description

## 1.1 Overview and main concepts

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.

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

**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 $motifCounter \leftarrow$ list of motif counters
          ▷ A motif counter is a dictionary containing a counter for each motif
    $sortdNodes \leftarrow$ 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
```

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

## 1.2 Motif combinatorics

Counting all motifs containing a given node is done using a **depth 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 or 3

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

TODO : add images of motifs

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.

## 1.3 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).**

## 1.4 Motif Counting

### 1.4.1 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
    $motifIndex \leftarrow$ GetMotifIndex(g,motif)
    for n doode in motif
        $motifCounters[node][motifIndex] + +$
    end for
end function
```

### 1.4.2 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.

**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 $n_1$ in root's neighbors do
    for $n_2$ in $n_1$'s neighbors do
      if $n_2$ is a neighbor of root's then
        if we saw $n_1$ before $n_2$ then        ▷ Count the pair only once
          UPDATEMOTIFCOUNT($g$,[$root, n_1, n_2$])
        end if
      else
        Mark that we saw n2
        UPDATEMOTIFCOUNT($g$,[$root, n_1, n_2$])
      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
  $neighborCombinations \leftarrow$ all 2-combinations of root's neighbors
  for $tuple$ in $neighborCombinations$ do
    $n_1 \leftarrow tuple.first$
    $n_2 \leftarrow tuple.second$
    if we saw $n_1$ before $n_2$ and they aren't neighbors then
 ▷ We only want to count $(n_1, n_2)$ once, and if they are neighbors, we would
                                    have discovered the motif as a motif of depth 2
      UPDATEMOTIFCOUNT($g$,[$root, n_1, n_2$])
    end if
  end for
end function
```

### 1.4.3 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.

**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
    $neighborCombinations \leftarrow$ all 3-combinations of root's neighbors
    for $tuple$ in $neighborCombinations$ do
        $n_{11} \leftarrow tuple.first$
        $n_{12} \leftarrow tuple.second$
        $n_{13} \leftarrow tuple.third$
        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 $n_1 \neq n_{11}$ then
                    edgeExists $\leftarrow$ $(n_2, n_{11}) \in E$ or $(n_{11}, n_2) \in E$
                    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
    $secondNeighborCombinations \leftarrow$ all 2-combinations of $n_1$'s neighbors
    for $combination$ in $secondNeighborCombinations$ do
        $n_{21} \leftarrow secondNeighborCombinations.first$
        $n_{22} \leftarrow secondNeighborCombinations.second$
        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 $\leftarrow (n_1, n_3) \in E$ or $(n_1, n_3) \in E$
                    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
```

## 1.5 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.

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:

1. 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.

2. 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.

**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
        $options \leftarrow permutations(motif)$
    else
        $options \leftarrow combinations(motif)$
    end if
    for tuple in options do
        $n_1 \leftarrow tuple.first$
        $n_2 \leftarrow tuple.second$
        $neighbors \leftarrow$ whether $n_1$ and $n_2$ are neighbors in $g$
        append $neighbors$ to $edges$
    end for
    $motifNumber \leftarrow$ the unsigned int represented by $edges$
    $motifIndex \leftarrow$ the index corresponding to the motif number
                  ▷ The index is read from the index file we created ahead of time
    return motifIndex
end function
```

## 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.

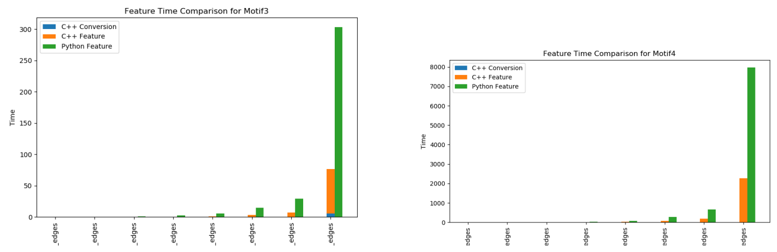

Figure 1: Run time comparison for python/C++ kernels

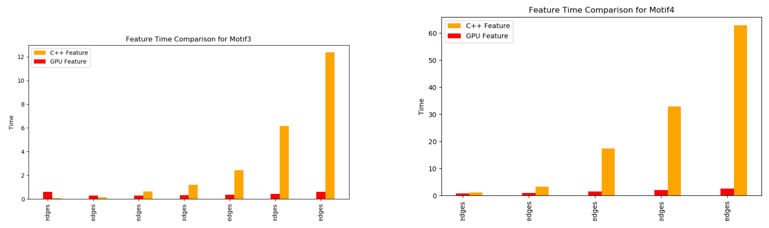

Figure 2: Run time comparison for C++/GPU kernels

"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.

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.

## 2.3 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. Figures 1 & 2 show the comparisons between the various kernels.

## 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:

$$\begin{array}{ccc} 0 & X & X \\ X & 0 & X \\ X & X & 0 \end{array}$$

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:

$$\begin{array}{ccc} 0 & X & X \\ 0 & 0 & X \\ 0 & 0 & 0 \end{array}$$