[Algorithm and their application final report]

Counting triangles in a graph

CSE 2020314303

Jinduk Park

1. Introduction

In this Big-data driven society, analysis of a huge size of graph is getting more importance. Various specific strategy to interpret graph's property is developed. One of the main strategy to capture graphs property is *'Counting triangles in a graph',* which is used for calculating the evaluation metric *clustering coefficient*[1]. By counting the total number of triangular edge connections in a graph, we can capture how much a graph is clustered.

In this report, we will share three strategy to efficiently compute the total number of triangles in a graph. The first one is using *DFS strategy* and *stack* data structure. DFS algorithm is abbreviation of Depth First Search algorithm, which is well known algorithm to efficient searching strategy for appropriate situation. Because we have to know edge existence relation of nodes in a sequential way, Search nodes in a DFS way is quiet reasonable. The second one is using multiplication of adjacency matrix. Adjacency matrix is useful data structure that contains graph's entire edge relation between nodes. Using power of adjacency matrix gives information of total walks between nodes. By using this number of walk information, we can extract total number of triangle patterns in a graph. And at the very last, we will use *Strassen's formula*, which is one of the *Divide & Conquer algorithm* to compute the power of matrix. *Strassen's formula* is useful when the matrix size is huge. So in this report, we will experimentally check from what extend Strassen's formula is useful than usual multiplication technique.

2. Problem definition & settings

What we want to solve is counting triangles in a graph, Denoted Tri_num here.The triangle in a graph has a trace pattern like: [a – b – c - a]. Here, [] means the sequences. a,b,c represent nodes. Hyphen(-) stands for the connection between nodes. G is a graph to calculate total number of triangular patterns there in. We consider G as undirected, unweighted graph. A is a corresponding adjacency matrix for G. A^k is k-th power of adjacency matrix A. Let say a_{ij}^k represent (i,j) element of adjacency matrix A^k.

In first strategy, stack and DFS strategy, let $\boldsymbol{s}$ is a stack that has size of three. $\boldsymbol{s}$ is used for containing

[1] Schank, Thomas, and Dorothea Wagner. "Approximating clustering coefficient and transitivity." *Journal of Graph Algorithms and Applications* 9.2 (2005): 265-275.

nodes that searched by DFS strategy. So, $n-th$ stack element node is connected with $(n-1),(n+1)-th$ element node. We will list up all the traces for the candidate of triangle shape, and will divide it by its permutational number to count triangles, not walks. **T** is triangular trace list, or triangular walk list, which the elements are the candidate for the triangle pattern list.

In second and third strategy, we will compute multiplication of matrix, using normal multiplication and *Strassen's formula*. In *Strassen's formula*, algorithm that using divide and conquer algorithm, we can divide a certain adjacency matrix whose graph size of n into four $\frac{1}{2}n$ graph size adjacency sub matrixes. using the formula, we would reduce the matrix multiplication time complexity and give efficiency to compute our algorithms. The detailed process for the *Strassen's formula* and its notations will be presented in Section 3.3.

Based on the notations above, what we want to solve is following. First, We will count total number of triangles in a graph by using three different strategy. Second, We would compare the three strategy in time complexity and space complexity to know which one is fastest & space-efficient. Also, Because the counting triangle algorithm is highly dependent on the topology of graph, We will only consider graph G as a 'lattice triangle graph' to easily compare the time & space complexity between three different method. 'lattice triangle graph' is a graph that has recursive triangle edge patten there in.

3. Algorithmic details & Its implementation

3.1 DFS with stack

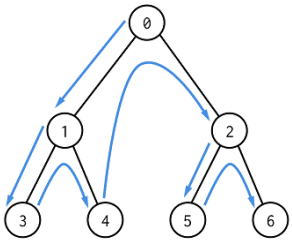


Figure1. Depth First Search tree

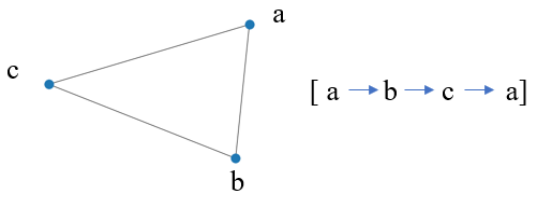


Figure2.triangle trace(walk) pattern

For counting triangles in a graph, we adopted DFS strategy to efficient computing. DFS is searching strategy that search each nodes as giving prior to its connection (**Figure1**). As in **Figure2**, The triangle pattern in a graph is like: [a – b – c - a]. we have to find sequential connected walks which means that it is no doubt to adopt DFS to efficient computing. We also used Stack data structure to efficient memory space management.

Stack data structure contains two method: *push, pop*, which makes algorithm useful and efficient in memory space management. By using *push* method, we can stack a data to the stack class. By using *pop* method, we can delete the very last stack element, which is most fit for the DFS searching algorithm.

detailed procedure is following. At first, we allocate Stack s with size of 3 to temporary contain the walk sequences. Using adjacency matrix A, we first start from random node, and *push* it to the s then as a choosing next element of the previous one, search based on connected nodes: indices that has a value '1' in adjacency matrix. From lower number, we choose node index out of candidates, and *push* it again to the s. we do the same thing once again. then we can get a *full stack* which has size of 3. At last, we consider whether the last element of s in connected with first element of s. If it is connected, than append the sequence to the triangle trace list **T**. else do nothing. After than, *pop* the very last element and do the same thing again. By doing the process above iteratively, we could get whole trace list(= walk list) that could be used for counting total triangular shape in a graph G. However, we are

counting triangles in a graph, not triangles trace or walk. So we have to divide the length of **T** by 3!, which is permutational number of 3 element. In other words, trace that [a – b – c - a], [a – c – b - a], [b – a – c - b], [b – c – a - b], [c – a – b - c], [c – b – a - c] these 6 walks are a same triangle. So we have to compute equation below.

$$Tri\_num = \frac{len(\boldsymbol{T})}{\mathbf{3!}}$$

Algorithm 1 Counting Triangle: DFS and Stack

```
Initialize: stack s, tracelist T, adj.mat A
for i = 1, 2, ..., n do
    s ← push(i)
    for j in A[i][j] == 1 do
        s ← push(j)
        for k in A[j][k] == 1 do
            s ← push(k)
            if A[k][i] == 1 then
                T ← append(s)
            end if
            s ← pop(i)
        end for
        s ← pop(k)
    end for
    s ← pop(j)
```

Pseudo code for counting triangle walks with *DFS & stack*

entire algorithm explained above is described as a diagram in **Figure3.**

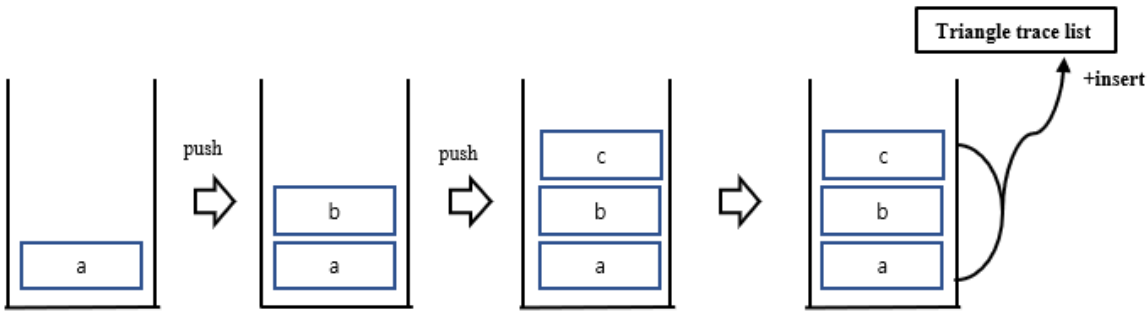
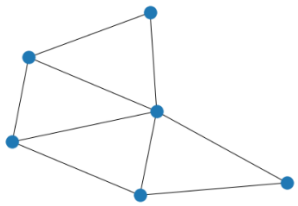


Figure3. Diagram for the DFS with stack algorithm

[Toy example implementation with Algorithm#1]



: Triangle lattice graph G size of 6 which has 4 triangles there in.

```
A
```

```
array([[0., 1., 1., 0., 0., 0.],
       [1., 0., 1., 1., 0., 0.],
       [1., 1., 0., 1., 1., 1.],
       [0., 1., 1., 0., 0., 1.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 1., 0.]])
```

: Corresponding adjacency matrix A

via algorithm suggested above, we can extract the triangle walk list appended from stack **s**.

Dividing length of the list **T**, *tri_trace* in implementation example, with 3! Gives 4, which is the number of triangle in toy example graph **G**

```
tri_trace
```

```
[[0, 1, 2],
 [0, 2, 1],
 [1, 0, 2],
 [1, 2, 0],
 [1, 2, 3],
 [1, 3, 2],
 [2, 0, 1],
 [2, 1, 0],
 [2, 1, 3],
 [2, 3, 1],
 [2, 3, 5],
 [2, 4, 5],
 [2, 5, 3],
 [2, 5, 4],
 [3, 1, 2],
 [3, 2, 1],
 [3, 2, 5],
 [3, 5, 2],
 [4, 2, 5],
 [4, 5, 2],
 [5, 2, 3],
 [5, 2, 4],
 [5, 3, 2],
 [5, 4, 2]]
```

```
len(tri_trace)/6
```

```
4.0
```

3.2 Using Power of adjacency matrix

Using power of adjacency matrix, we can also extract total number of walk list that could be used for calculating the triangle number before we dive into the specified algorithm there is a preliminary to catch the idea of the algorithm.

> Theorem 0.1 The (i,.i)th entry a_{ij}^{k} of A^{k}, where A = A(G), the adjacency matrix of G, counts the number of walks of length k having start and end vertices i and j respectively.[2]

By the theorem 0.1, the diagonal entries of A^3, which denoted by a_{ii}^{3} $(i = 0,1,2,\dots,n)$, represent number of walks of length k having start and end vertices i and i. because the walk pattern for the triangle is like: [a,b,c,a] , we can extract the total number of triangle walks by summing up the diagonal element of A^3, which denoted by sum$(diag(A^3))$.

As we discussed in 3.1, divide total number of walks divided by its permutational number, in this case 6, is the total number of triangles in a graph. In short, the total number of triangles in a graph is below.

$$Tri\_num = \frac{\text{sum}(diag(A^3))}{\mathbf{3}!}$$

3.3 Using Power of adjacency matrix – *Strassen's formula*

Divide & Conquer algorithm is well known algorithmic strategy that can reduce the time computational costs in many tasks. In matrix multiplication, we can also adopt *Divide & Conquer* algorithm to conduct multiplication of matrix. After dividing size n by n matrix to four n/2 size

---

[2] Sciriha and E.tvI. Li Marzi Boolcan Matrices Graph Theory Notes of New York GTN XLVI 20-26 (2004)

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

submatrices we can conduct each matrix multiplication and sum up. Then we can compute whole size of matrix multiplication. Detailed procedure for *Strassen's formula* noted below.
Let there is a relation between matrix A,B,C as C = A*B, Let say $[c_{00}, c_{01}, c_{10}, c_{11}]$, $[a_{00}, a_{01}, a_{10}, a_{11}]$ and $[b_{00}, b_{01}, b_{10}, b_{11}]$ stands for the corresponding matrix element of each matrix A,B,C, then the *Strassen's formula* is defined as below.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

here, $[m_1, m_2, m_3, m_4, m_5, m_6, m_7]$ each are $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$, $m_2 = (a_{10} + a_{11}) * b_{00}$, $m_3 = a_{00} * (b_{01} - b_{11})$, $m_4 = a_{11} * (b_{10} - b_{00})$, $m_5 = (a_{00} + a_{01}) * b_{11}$, $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{11})$, $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$.[3].
when divide the partition, for the case where the number of vertices are not 2^n, we zero-padded with remainder entry element to give generality to the algorithm.

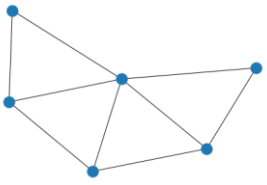
Algorithm 3 Strassen-Recurssive

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4      c11 = a11 · b11
 5  else partition A, B, and C as in equations
 6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
             + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

Pseudo code : Strassen's formula matrix multiplication

By using the Divide & Conquer algorithm above, we can reduce time complexity for the matrix multiplication from $O(n^3)\ to\ O(n^{2.8})$. if the size of the graph is huge enough, the effect of using Strassen's formula would greatly be high. In this big data driven society, It is worth enough using the algorithm.

[Toy example implementation with Algorithm#2,3]



we used same graph G implemented in algorithm#1 whose node number is 6 and 4 triangles there in.

Corresponding adjacency matrix A is:

[3] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao and T. Turnbull, "Implementation of Strassen's Algorithm for Matrix Multiplication," Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, Pittsburgh, PA, USA, 1996, pp. 32-32, doi: 10.1109/SUPERC.1996.183534.

```
In [6]: A
Out[6]:
array([[0., 1., 1., 0., 0., 0.],
       [1., 0., 1., 1., 0., 0.],
       [1., 1., 0., 1., 1., 1.],
       [0., 1., 1., 0., 0., 1.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 1., 0.]])
```

```
A**3 by Strassen formula:=
[[2. 5. 7. 3. 2. 4.]
 [5. 4. 8. 7. 4. 4.]
 [7. 8. 8. 9. 7. 8.]
 [3. 7. 9. 4. 3. 7.]
 [2. 4. 7. 3. 2. 5.]
 [4. 4. 8. 7. 5. 4.]]
```

```
A**3 conventional matrix multiplication:=
[[2. 5. 7. 3. 2. 4.]
 [5. 4. 8. 7. 4. 4.]
 [7. 8. 8. 9. 7. 8.]
 [3. 7. 9. 4. 3. 7.]
 [2. 4. 7. 3. 2. 5.]
 [4. 4. 8. 7. 5. 4.]]
```

We could see the same results of A^3computed by algorithm#2, #3. And corresponding diagonal sum is

Implementation result of $Tri\_num = \frac{\text{sum}(diag(A^3))}{3!}$: `sum of the diagonal elements divided by 3!:  4.0`

Which is fit with our expectation.

4. Experimental Results

we already implemented each of algorithm in algorithmic detail section 3 with Toy example. In this section, we will compare algorithms suggested in section 3 focusing on its computational efficiency – Time complexity.

We used triangle lattice graphs for the evaluation of algorithm. Because some of the suggested algorithm depends on graph's topolgy, we fixed graph to tirangle lattice graph with varying sizes. Triangle lattice graph is a graph that has recursive triangle patterns there in.

We firstly offer time consumption behavior for the algorithm#1, DFS with stack data structure.

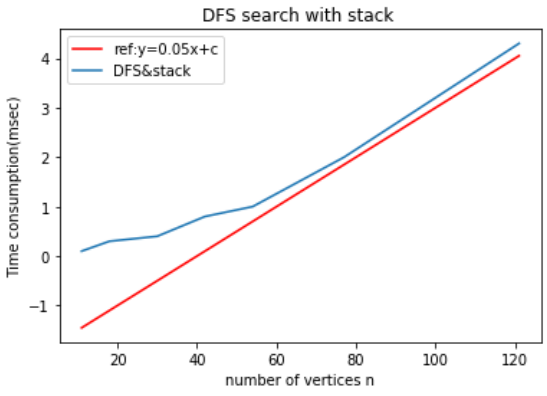


Figure2

As we can verify with the plot above, time complexity is linear with size n. experimentaly, we can verify the time complexity asymtotic notation for algorithm #1 is: $O(n)$.

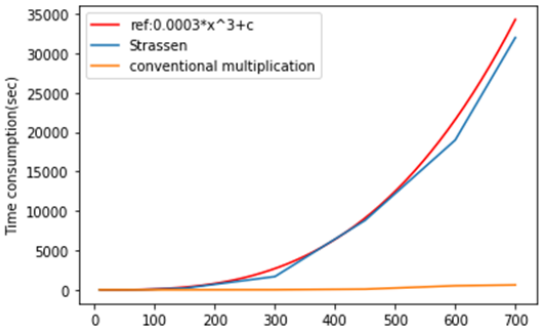


Figure3

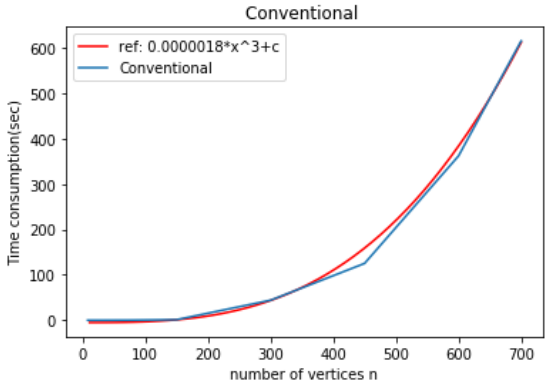


Figure4

next experimental results offer the time consumption behavior for the algorithm#2,#3 at the same time(**Figure3**) and behavior for conventional multiplication only(**Figure4**) .
Here we conducted conventional multiplication not using numpy packages. Because **np.dot()** method is optimized multiplication package method, we did not use the method but we newly implemented conventional multiplication using arrays. Details of the codes are in **Appendix.**
We conductied until the nodes number ~ 700. It tooked about 8 hours to compute with node number 700. The strassen's alogrithm is slightly going under the line $y = an^3 + c\ (a, c : constant)$ and Conventional multiplication is fitted with $y = an^3 + c\ (a, c : constant)$

5. Analysis & Discussion

As we can verify by checking the suggested pseudo code, there are 1 iteration for the total node number n, and 2 iteration for the maximum number of degrees, so the total time complexity for the algorithm #1 is $O(n * maxdegree^2)$, *maxdegree* represents the maximum degree of the graph G. if the maxdegree is constant, we could see the linear monotonic increase in time cost for the algorithm#1.Also the pros of the algorithm#1 is that we can track the walks with stack while other algorithm only gives total number of triangles there in.

For the Algorithm #2 or #3, To compute the equation, we have to derive A^3, by using multiplication of matrix. Conventional multiplication of a matrix A three times gives time complexity $O(n^3)$ because we have to multiply and add every rows and columns of the target matrix. But Strassen's formula can reduce time complexity from $O(n^3)\ to\ O(n^{2.8})$. Because the initial cost for each algorithm is different, time cost for the *Strassen's formula* is known to be higher than conventional one. By the experiment, we can measure from what extend the time consumption would be overtaken by *Strassen's formula.*

As we can see in **Figure3**, initial time cost for Strassen's formula is extremely high compared to conventional one, so It was almost impossible to check from what extend Strassen's formula would be better. (time cost for n=10, Strassen:0.3 sec while Conventional:0.002 sec)
Hence, Instead, a trend line is used to predict when to overtake conventional multiplication.

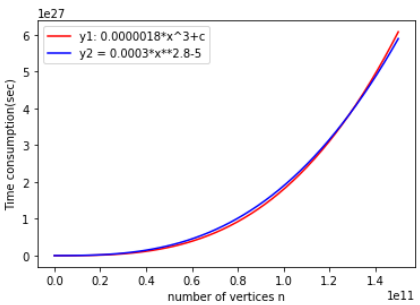


Figure4

The trend line for each is heuristically chosen. By the result described in **Figure4**, If the trendline assumption is well chosen, we could predict that from the node number about $1.2*e^{11}$**,** the Strassen result is efficient than conventional one. As a consequence, It is quite unfeasible to adopt Strassen's algorithm when we counting triangles in a graph. Even if the Strassen's formula has time complexity of $O(n^{2.8})$, the initial cost for the algorithm is too high that It takes too much to overtake conventional one. The initial time cost of *Strassen's formula* would be high. Because In Strassen algorithm, we have to divide matrix into submatrix size of the half, we have to zero padding and prune the padding after multiplication, And finally we recursively calls the procedure described above. The described process somewhat complex to compute, which takes much time than conventional matrix multiplication, which gives more time to take when we adopt the formula.

Asymptotic notation for each algorithm#1,#2,#3 is $O(n * maxdegree^2)$, $O(n^3)$, $O(n^{2.8})$ each. As a consequence, It is no doubt that *DFS with stack* algorithm has most priority in time complexity, not only in tracing the walk sequences.

APPENDIX: source code

Algorithm #1. DFS with stack

```
class Stack(list):
    push = list.append    # Push to Stack
                          # Delete - python internal pop method

    def is_empty(self):   # check whether it is empty or not
        if not self:
            return True
        else:
            return False

    def peek(self):        # very last data checking
        return self[-1]

    def clear(self): # clear stack
        list.clear()

#initiallize graphs
G = nx.triangular_lattice_graph(10,20)
A = nx.to_numpy_array(G)
nodenum = len(A[0])
print(' # of node for G is:', nodenum)
#initiallize stack
s = Stack()


start = time.time()  # Time check start point

#main algorithm using DFS & Stack
tri_trace = []
for i in range(nodenum):
    s.push(i)
    for j in range(nodenum):
        if A[i][j] == 1:
            s.push(j)
            for k in range(nodenum):
                if A[j][k] == 1:
                    s.push(k)
                    if A[k][i] == 1:
                        #print("Stack is full:", s)
                        s_to_append = copy.deepcopy(s)
                        tri_trace.append(s_to_append)
                    s.pop()
            s.pop()
    s.pop()

tri_num = len(tri_trace)/6

print("Completed time :", time.time() - start)  # 현재시각 - 시작시간 = 실행 시간

print('total number of triangles in a graph is:', tri_num)
```

Algorithm#2.Conventional multiplication

```
def matrix_multiplication(M,N):
    # List to store matrix multiplication result
    l = len(M[0])
    R = np.zeros((l,l))
    for i in range(0, l):
        for j in range(0, l):
            for k in range(0, l):
                R[i][j] += M[i][k] * N[k][j]
    return R
```

Algorithm#3. Strassen's formula

```
#Strassen algorithm definition
def Strassen_Algorithm(A,B):
    n1,m1=A.shape; n2,m2=B.shape;
    result=np.zeros((n1,m2));

    # Zero paddings prerocessing for generalization sizes
    temp=0
    #for matrix A
    if n1>=m1:
        temp=0
        while 2**temp<n1:
            temp+=1
    else:
        while 2**temp<m1:
            temp+=1

    AP=np.pad(A,((0,2**temp-n1),(0,2**temp-m1)),'constant', constant_values=0);

    #for matrix B
    if n2>=m2:
        temp=0
        while 2**temp<n2:
            temp+=1
    else:
        while 2**temp<m2:
            temp+=1

    BP=np.pad(B,((0,2**temp-n2),(0,2**temp-m2)),'constant', constant_values=0);


    k=BP.shape[0]; # size of the matrix after padding

    if k==1:
        C=AP.dot(BP);
        result=C[0:n1,0:m2] # removing padding elements & n1*m1 matrix * n2*m2 matrix = n1*m2 matrix
        return result;

    else:
        # D&C Recursive formula for Strassen's multiplication formula
        AP11=AP[0:int(k/2),0:int(k/2)]
        AP12=AP[0:int(k/2),int(k/2):]
        AP21=AP[int(k/2):,0:int(k/2)]
        AP22=AP[int(k/2):,int(k/2):]
        BP11=BP[0:int(k/2),0:int(k/2)]
        BP12=BP[0:int(k/2),int(k/2):]
        BP21=BP[int(k/2):,0:int(k/2)]
        BP22=BP[int(k/2):,int(k/2):]

        # Strassen Recusive : Function calls function itself
        M1=Strassen_Algorithm(AP11+AP22,BP11+BP22);
        M2=Strassen_Algorithm(AP21+AP22,BP11)
        M3=Strassen_Algorithm(AP11,BP12-BP22)
        M4=Strassen_Algorithm(AP22,BP21-BP11)
        M5=Strassen_Algorithm(AP11+AP12,BP22)
        M6=Strassen_Algorithm(AP21-AP11,BP11+BP12)
        M7=Strassen_Algorithm(AP12-AP22,BP21+BP22)


        C11=M1+M4-M5+M7; C12=M3+M5; C21=M2+M4; C22=M1+M3-M2+M6;
        n=C11.shape[0]
        C=np.zeros((2*n,2*n))
        C[:n,:n]=C11; C[:n,n:]=C12; C[n:,:n]=C21; C[n:,n:]=C22
        result=C[:n1,:m2] # removing padding elements & n1*m1 matrix * n2*m2 matrix = n1*m2 matrix
        return result;
```

Experiment on Algorithm #2,#3

```
#initiallize graph G and extract adjacency matrix
G = nx.triangular_lattice_graph(5,5)
A = nx.to_numpy_array(G)
nodenum = len(A[0])
print(' # of node for G is:', nodenum)


start3 = time.time()  # Time check start point for algo#3
#main function for algorithm#3
B=Strassen_Algorithm(A,A)
C=Strassen_Algorithm(A,B)
tri_num = np.sum(np.diag(C))/6

print("Completed time for Strassen's :", time.time() - start3)  # 현재시각 - 시작시간 = 실행 시간
#main function for algorithm#2
start2 = time.time()  # Time check start point for algo#2

B2 = matrix_multiplication(A,A)
C2 = matrix_multiplication(B2,A)
tri_num = np.sum(np.diag(C))/6
print("Completed time for conventional multiplication :", time.time() - start2)  # 현재시각 - 시작시간 = 실행 시간

#print('sum of the diagonal elements divided by 3!: ',tri_num)
```