K-Core Decomposition
Discrete-Math Final Project

ABSTRACT

In this paper, I am going to implement some of the famous algorithm to find a k-core decomposition of a given graph. "*The k-core decomposition is to find the largest subgraph of a network, in which each node has at least k neighbors in the subgraph.*"[1]

The core concept of finding a k-core decomposition of a graph is to remove vertices that has less degree until the graph becomes empty. First, I will try to find k-core decomposition using the Naive Method taught in class. After that, I will implement the **Batagelj and Zaversnik algorithm** for k-core decomposition used in *Webgraph*.[2]

1. INTRODUCTION

With the rapid growth of communication technique, the complexity of social network becomes far more complicated than we had expected, people are having more connection through the help of telecommunication. Besides, it turns out that a variety of topics can be discussed by analyzing the social network, such as the spread of information, friendship-and-acquaintance network, disease transmission, collaboration graphs, business networks, kinship, etc.[3]

Another crucial application of finding k-core decomposition is for network provider to know what part of the network is loosely connected and hence dangerous. We know the utility of a vertex depends on the number of connections of the vertex with the community. If one of the loosely connected users drop out , their friends will become loosely connected too and might also drop out too, such a series of disengagement is quite dangerous. So we rely on finding k-core decomposition to know the minumum *k* required to maintain the utility of the graph.[2]

2. Preliminaries

We represent our network using undirected graphs by ***G = (V,E)***, where **V** is the set of vertices, and **E** is the set of edges. We also use the **|V|** and **|E|** notation.

I denote the adjacent nodes of a vertex **v**, $\{u \mid (u, v) \in E\}$, by $\mathbf{N_G(v)}$**.** The degree of v is denoted by $\mathbf{d_G(v)}$. And set $\mathbf{d_{max}(G)}$ to be the maximum degree in the graph.

Density = (The actual Edges in the graph)/(The maximum edges there can be in the graph)

3. EXPERIMENTAL SETTING

Setup. My implementations are in C++ and the experiments are conducted on a machine with Intel G5400, 3.7Ghz CPU, and 8GB RAM, running on VSCode.

Datasets. I wrote a program to provide random edges with different |V| and |E|

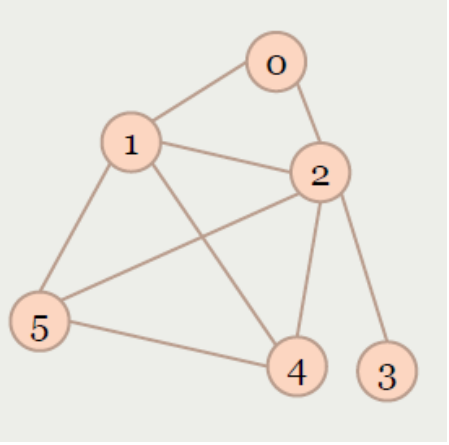
You can check the overall dataset statistics in the **Result&Analysis** part.

4. Algorithm Implementation[4]

Input

Given a set of number pair (a, b), representing the edges of the graph, and the edges will be given in ascending order, and the maximum number in the set is 1000, i.e. (0 <= a, b < 1000). For example :

0 1
0 2
1 2
1 4
1 5
2 3
2 4
2 5
4 5



4.1 Naive Method

Initialization

Variables:

1. *struct Vertex with members*
   int degree //trace the number of $d_G(v)$
   vector<int> **arc** //store $N_G(v)$
2. *vector<Vertex> V;*
   No initialization, just apply resize() when the input number is greater than the current size.
   Also push adjacent nodes in to **arc**
3. *int RemainingNode = V.size():*
   Initialized as n, the number of vertices.
   It will decrease by 1 whenever a vertex is removed from the graph.
4. *vector<Vertex> dup //duplicate*
   Record the previous stage of the set of vertices to print the remaining edges.
   It will duplicate the data of V when k increases.

Termination check

I use the variable RemainingNode to see if the graph has no vertices in it. When there are no vertices left in the graph, the process terminates.

Pseudo Code

Functions used in pseudo code

1. *UpdateNeighbors* :

Simply subtract the vertex's all neighbors's degree by 1

2. *PrintResult* :

For each vertex in **dup**, if the vertex's degree > 0(so it's still in the graph), sort its **arc**(neighbors) and print the desired result.

ALGORITHM 1: NAIVE METHOD

```
//Assume we finish reading data
1: function k-core(V, DegreeMax)
2:   k = 1, RemainingNode = V.size(), dup
3:   while(RemainingNode > 0) do
4:     dup = V
5:     repeat = true
6:     while(repeat) do
7:       repeat = false
8:       for(all v with deg < k) do
9:         v.remove()
10:        RemainingNode- -
11:        v.UpdateNeighbor()
12:        for(u ∈ N_G(v) ) do
13:          if(d_G(v) < k) do
14:            repeat = true
15:          end if
16:        end for
17:      end for
18:    end while
19:  end while
20:  PrintResult(k-1, dup)
21:end function
```

pseudo code for Naive Method

4.2 Batagelj and Zaversnik Method (BZMethod)

Initialization

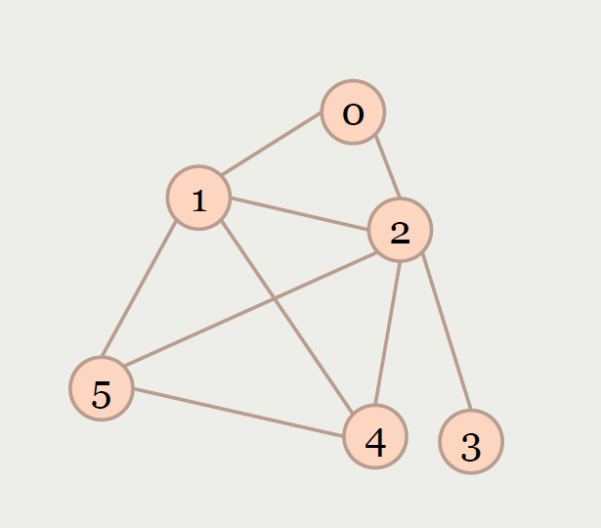
Variables:

1. struct Vertex with members
2. vector<Vertex> V;
3. int DegreeMax
   Just store the Maximum degree in the graph
4. vector<int> CoreValue(V.size(), 0)
   Initialized with 0 to record each vertex's final core value
5. vector<vector<int>>NumofVtxofDeg
   2'D array to store how many vertices and what vertices are there in each degree(from 0 ~ DegreeMax). Example given below.

Termination check

The algorithm is done by removing vertices as mentioned before, and it won't stop the removing operation until the current slot(as we are removing vertices that has less degree than *k*) in *NumofVtxof Deg* is not empty. So it will eventually terminates

once the NumofVtxofDeg is empty.



The 2D array looks like :

degree	Vertices' index
1	3
2	0
3	4,5
4	1
5	2

Example of the NumofVtxofDeg 2D array

Pseudo Code

Functions in the pseudo code :

1. *UpdateNeighbor() :*
   The index of the slot represents the vertex's current degree so we have to move it from the current slot to the lower slot(e.g. move from NVD[k] to NVD[k-1]). And update its degree.
2. *PrintResult() :*
   Do pretty much the same thing as before.

3.3 Optimized BZ Method

The difference between this and the previous method is that this method "flatten" the 2D array in BZ Method and create 2 extra array to store the beginning position of each degree and the position of each vertex in the flattened 2D array. I think it sort of utilize the concept in sparse matrix.

Initialization

Variables:

1. struct Vertex with members

ALGORITHM 2: BZ METHOD

```
//Assume we finish reading data
//NumofVtxofDeg is abbreviated to NVD
1: function k-core(V, DegreeMax)
2:    initialize CoreValue, NVD
3:    int Ans = 0
4:    for(i = 0 to DegreeMax) do
5:       while(not NVD[i].empty()) do
6:          int k = NVD[i].front()
7:          NVD[i].remove_front()
8:          CoreValue[k] = i
9:          Ans = max(Ans, i)
10:         v.UpdateNeighbor()
11:      end while
12:   end for
13:   PrintResult(k-1, dup)
14:end function
```

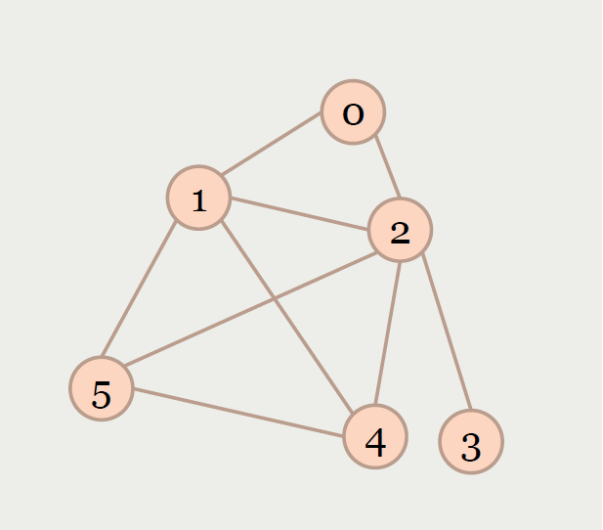
Algorithm for BZ Method

2. vector<Vertex> V;
3. int DegreeMax
4. vector<int>beginOfDeg(DegreeMax+1)(b.)
   Store the starting index of each "degree number" in the NumofVtxofDeg array(1D in this method, so we called it flattend array)
5. vector<int> Position(V.size()) (p.)
   Store the position of each vertex in the NumofVtxofDeg array
6. vector<int> NumofVtxofDeg (NVD.)
   Becomes an 1D array with V.size() integers
   Initialize as empty array since I have to initialize the *beginOfDeg* and *Position* with its size during the insertion process.

Termination check

We will start the iteration at the beginning of NumofVtxofDeg to its end, and at each iteration, we update the degree of each vertex's neighbors by modifying their *Position* in the *NumofVtxofDeg* and *beginOfDeg*. The modification is done by swapping the neighbors position with the position of beginOfDeg[The Neighbor's degree] and

increase beginOfDeg[The Neighbor's degree] by 1.



The arrays looks like :

Vertices	Degree	b.	p.	NVD.
0	0	0	1	3
1	1	0	4	0
2	2	1	5	4
3	3	2	0	5
4	4	4	2	1
5	5	5	3	2

Example of NumofVtxofDeg 1D array

Pseudo Code

Functions in the pseudo code :

1. *swap :*

   Since we want to swap the vertices position in the NVD array so we declare 4 integers, assume "j" is the neighbor :

   "DegJ = V[j].degree
   PosJ = Position[j],
   PosW = beginOfDeg[DegJ],
   SubW = NumofVtxofDeg[PosW] "

   then if SubW is **not** j(it is ok even if SubW is j), we swap their position in NVD by:

   " NumofVtxofDeg[PosJ] = SubW;
   NumofVtxofDeg[PosW] = j; "
   and record the swap in Position array :
   " Position[j] = PosW
   Position[SubW] = PosJ "

2. *maxOfAllVtxDeg() :*

   After the process, the degree of each vertex will be its true core value, so we can get maximum k-core by iterate through the vertex array and get its maximum

3. *PrintResult():* Same as before

ALGORITHM 3: BZ METHOD

```
//Assume we finish reading data
//NumofVtxofDeg is abbreviated to NVD
//beginOfDeg abbreviated to BD
//Position abbreviated to Pos
1: function k-core(V, DegreeMax)
2:    initialize BD, Pos, NVD
3:    for(i = 0 to V.size()) do
4:       int RVtx = NVD[i]
          //Vertex to be removed
5:       for(u ∈ NG(RVtx)) do
6:          if(dG(u) > dG(RVtx)) do
7:             if(NVD[BD[dG(u)]] is not u) do
8:                swap(u,NVD[BD[dG(u)]])
9:             end if
10:            BD[dG(u)]++
11:            dG(u) - -
12:            end if
13:     end for
14:   end for
15:   Ans = maxOfAllVtxDeg()
16:   PrintResult(Ans, V)
17:end function
```

5. Result & Analysis

Execution Time with respect to Density :

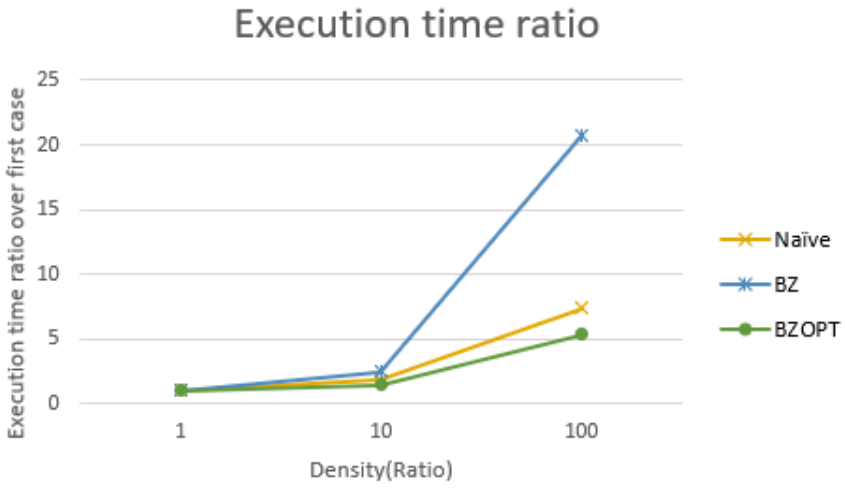
There are two cases when comparing result with respect to density. The first is when the |V| is fixed, how does the execution time of each method increases as the density becomes 10 times larger?

We can see from the result that when |V| increases by 10 times, the execution time of Naive Method does not rise too much when |V| is below 1000, but has a dramatically rise when |V| is over 10000; BZ method has the worse performance since the ratio surges for all cases, although you might argue that the third case where |V| is 32768 has the smallest ratio number, it is still not a good method since the actual exec. time is too long(1mins ~ 2 mins); the best method, as expected, is optimized BZ method. You can see from the line chart that BZOPT has better performance than Naive Method, although in the second case of the 32768 |V| case, it is worse than Naive Method(probabily due to the complexity of its initialization), it eventually beats Naive Method when the density increases.

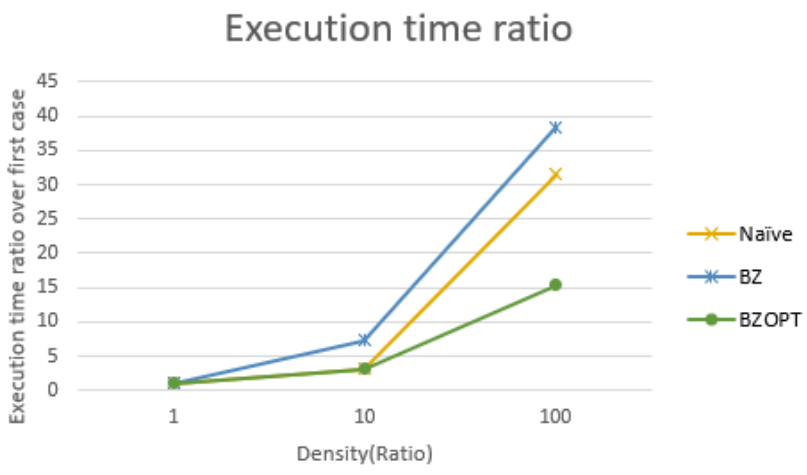
The overall result

|V|	Density	Naïve	BZ	BZOPT
1000	1	1	1	1
1000	10	1.8444444	2.4625	1.4477612
1000	100	7.4	20.7375	5.3432836
10000	1	1	1	1
10000	10	3.1413043	7.2626263	3.1164384
10000	100	31.554348	38.367003	19.061644
32768	1	1	1	1
32768	10	6.8167053	3.1940554	9.0567568
32768	100	90.515081	15.393255	87.627027

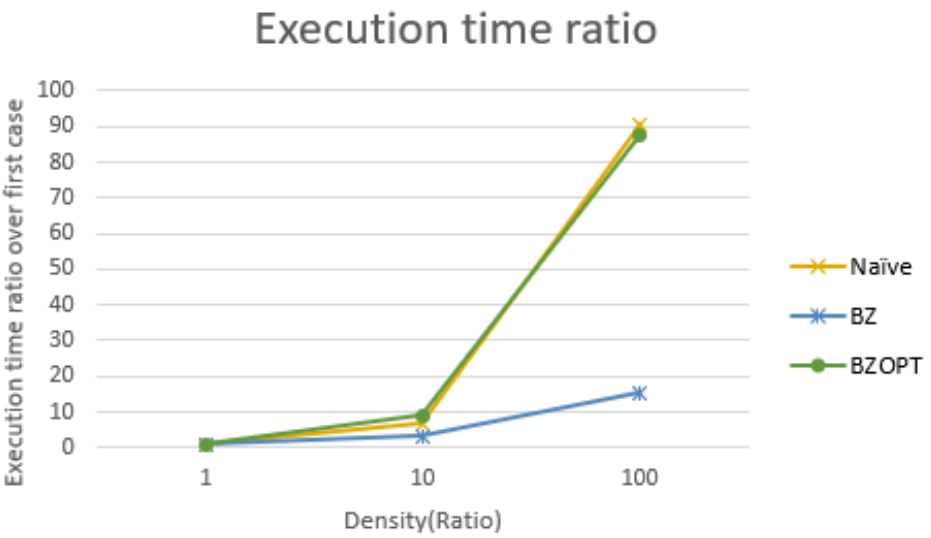
The ratio of increment when the |V| is 1000



The ratio of increment when the |V| is 10000



The ratio of increment when the |V| is 32768

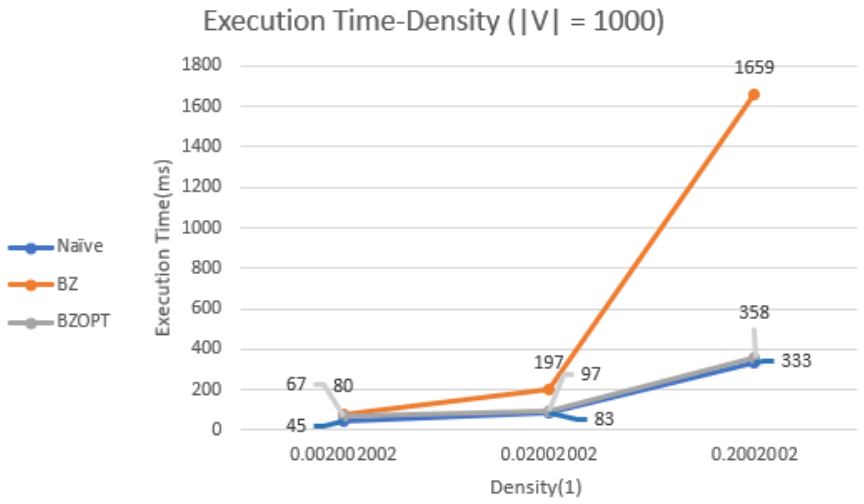


In the second case I will compare three method **under the constraint that |V| and the density is equal.** We will see that Naive Method is better in the cases when the |V| and Density is not too large.(See the red rectangle below). And Optimized BZ Method doesn't beat Naive Method until the last case where density is 0.002 and |V| = 32768(The green rectangle below). I am not sure whether Optimized BZ Method still wins over Naive Method for any cases that is greater than this because my computer made some weird noise when performing the last 2 cases, but it is quite a surprising fact since I finished Naive Method on my own and it works well. We can ignore BZ Method in this stage since its execution time is far more greater than the other 2 methods.

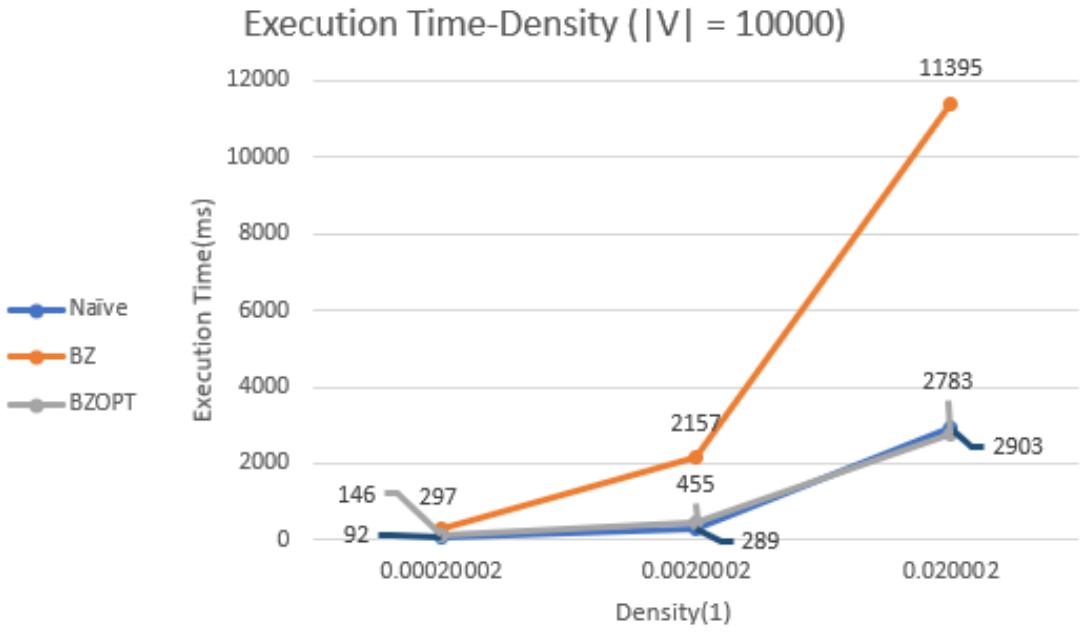
The overall result

|V|	Density	Naïve	BZ	BZOPT
1000	0.002002	45	80	67
1000	0.02002	83	197	97
1000	0.2002	333	1659	358
10000	0.0002	92	297	146
10000	0.002	289	2157	455
10000	0.020002	2903	11395	2783
32768	0.000186	431	6998	370
32768	0.001863	2938	22352	3351
32768	0.018627	39012	107722	32422

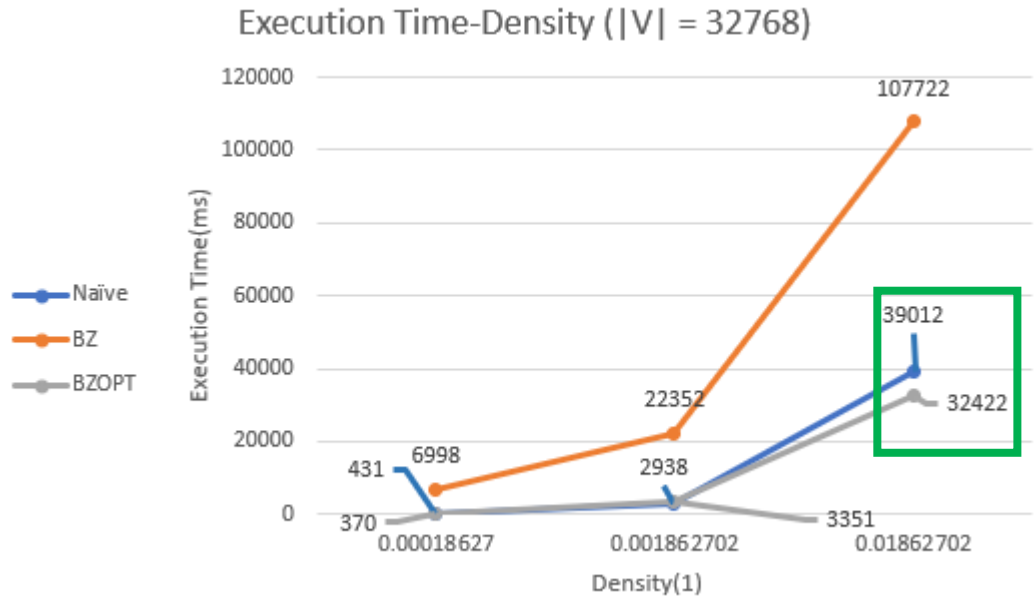
Execution time with respect to density (|V| = 1000)



Execution time with respect to density (|V| = 10000)



Execution time with respect to density (|V| = 32768)



Real Execution Time:

The reason why I want to write this part is because I pass OJ and I wonder how can I speed up my program, and suddenly it occurs to me that when they participate in competitive programming they use a command to speed up the file reading. So I surf the net and add the code and speed up my program for about 0.1s ~ 0.4s faster. The fast i/o command is : ios_base::sync_with_stdio(0), cin.tie(0);

They are different command of different use, "*sync_with_stdio(0)*" will separate the buffer area for scanf/printf from cin/cout, so you should mix them once you make the statement. "*cin.tie(0)*" will "untie" *cin* from *cout*, *cin* is tied to *cout* by default to ensure the message output to terminal before *cin* so that user knows when to insert the value. If they are untied, we have to manually flush the *cout* buffer before *cin* because the buffer will not automatically flush until the buffer is full. Since reading data comes before output, so we can use *cin.tie(0)*.[5]

The Execution time on OJ:

278909	1446. Maximum k-core	109511028	232	7460	AC	C++	Optimized BZ Method (With FastIO)
278908	1446. Maximum k-core	109511028	312	7912	AC	C++	Optimized BZ Method (Without FastIO)
278907	1446. Maximum k-core	109511028	1244	7472	AC	C++	BZ Method (With FastIO)
278906	1446. Maximum k-core	109511028	1148	7756	AC	C++	BZ Method (Without FastIO)
278905	1446. Maximum k-core	109511028	928	11144	AC	C++	Naive Method (With FastIO)
278904	1446. Maximum k-core	109511028	1332	11396	AC	C++	Naive Method (Without FastIO)

And here are the result of my program running the 9 testcases with and without fast i/o, the result is based on optimized BZMethod.

Testcase	|V|	|E|	MaxDegree	AvgDegree	k_max	k_avg	Density	Read file time	without fast i/o	with fast i/o
MyTest1	1000	1000	9	2	2	1.351	0.002002	0.043	0.063	0.104
MyTest2	1000	10000	35	20	14	13.756	0.02002	0.054	0.068	0.091
MyTest3	1000	100000	244	200	170	169.847	0.2002	0.164	0.298	0.264
MyTest4	10000	10000	9	2	2	1.3313	0.0002	0.079	0.098	0.073
MyTest5	10000	100000	43	20	13	12.8341	0.002	0.202	0.346	0.347
MyTest6	10000	1000000	304	200	157	156.871	0.020002	1.402	3.002	2.603
MyTest7	32768	100000	19	6.10352	4	3.72174	0.000186	0.255	0.401	0.35
MyTest8	32768	1000000	97	61.0352	47	46.8792	0.001863	1.741	3.033	2.614
MyTest9	32768	10000000	713	610.352	550	549.925	0.018627	14.927	32.508	25.439

1. The result is quite faster than previous figure is because my computer was kind of overloaded when I performed it, and it works well while doing this part of my project.

2. Read file time is measured by commenting out the call function line (//BZ_Method())

So as we can see from the figure, programs with **fast i/o command speed up the execution time when the input file is quite large.**

Other Optimization Added:

1. Sorting Method

   I use another sorting method based on **counting sort** instead of just using the sort function in <algorithm>

```
if(V[i].Neighbor.size() < 250)
    sort(V[i].Neighbor.begin(), V[i].Neighbor.end());
else{
    vector<int> temp(V[i].Max+1, 0);
    for(auto j:V[i].Neighbor) temp[j]++;
    B = 0;
    for(int j = 0; j < temp.size(); j++){
        if(temp[j]) V[i].Neighbor[B++] = j;
    }
}
```

The counting sort part

   The Time complexity is O(Max+Neighbor+Max) //Max is the maximum index in its neighbors

   And the sort function in algorithm is NlgN, so I check in the calculator that NlgN exceed the previous time complexity at x = 280, under the condition that Max is at most 1000.(xlgx = x+2000). so I add a condition to use counting sort. I select 250 instead because it runs faster on OJ.

2. Getting the core value

   I go over the Vertex array and get the core value by finding the Maximum in the array at first. Then I realized that the last index in the NumofVtxofDeg must has the final answer so I change the code :

from

```
int Ans = 0;
for(auto i:V) Ans = max(Ans, i.degree);
cout << Ans << "-core" << '\n';
```

to

```
A = V[RVtx].degree;
cout << A << "-core\n";
```

   This enhance the performance from O(|V|) to O(1).

Summary :

I implemented 3 kinds of Method and below is the overall result.

								Method Used		
Testcase	|V|	|E|	MaxDegree	AvgDegree	k_max	k_avg	Density	Naïve	BZ	BZOPT
sample1	9	17	6	3.77778	3	2.55556	0.472222	0.057	0.066	0.049
sample2	7	21	6	6	6	6	1	0.059	0.066	0.073
MyTest1	1000	1000	9	2	2	1.351	0.002002	0.045	0.08	0.067
MyTest2	1000	10000	35	20	14	13.756	0.02002	0.083	0.197	0.097
MyTest3	1000	100000	244	200	170	169.847	0.2002002	0.333	1.659	0.358
MyTest4	10000	10000	9	2	2	1.3313	0.0002	0.092	0.297	0.146
MyTest5	10000	100000	43	20	13	12.8341	0.0020002	0.289	2.157	0.455
MyTest6	10000	1000000	304	200	157	156.871	0.020002	2.903	11.395	2.783
MyTest7	32768	100000	19	6.10352	4	3.72174	0.0001863	0.431	6.998	0.37
MyTest8	32768	1000000	97	61.0352	47	46.8792	0.0018627	2.938	22.352	3.351
MyTest9	32768	10000000	713	610.352	550	549.925	0.018627	39.012	107.722	32.422

k_max is the maximum core in the graph, execution Time's unit is in second(sec).

Density is given by (The actual Edges in the graph)/(The maximum edges there can be in the graph),

My source code at :
https://github.com/krz-max/Discrete-Math-Programs

6. References

[1]https://www.sciencedirect.com/science/article/pii/S037015731930328X
[2]https://dl.acm.org/doi/pdf/10.14778/2850469.2850471
[3]https://en.wikipedia.org/wiki/Social_network_analysis
[4]https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-207
[5]https://stackoverflow.com/questions/31162367/significance-of-ios-basesync-with-stdiofalse-cin-tienull