MAXIMUM K-CORE

Fall 2021

The present report is about the Maximum k-core of an undirected graphs, comparison between the different available algorithms, the method to achieve an efficient algorithm, and as well as present my implementation using C++.

INTRODUCTION

A k-core is the maximal subgraph where all vertices have degree at least k. This concept has been applied to such diverse areas as hierarchical structure analysis, graph visualization, and graph clustering.
The present document is organized as follows: part I defines the main purpose of a the maximum k graph, its historicity and the notions behind it. While part II provides an explanation of the different methods available to find the maximum k-core, as well as their respective advantages and cons, and part III provides the template of my implementation of such an algorithm the issues that I faced and how I managed to resolve them.

I. STRUCTURE OF A K-CORE

Historicity:
The term k-degenerate was introduced in 1970 by Lick and White; the concept has been introduced under other names and many evolution has happened to this field please refer to here for more. K-core's applications in graph theory, ranges from large-scale visualization of data,to data mining. It is a crucial topic to graph theory as it is know.

Note
In this assignment I will use extensively the term k-degenerate since it is reasonably standard and helps to simplify formulas, while also using k-core(free) when convenient.

What is the k-core of a graph ?
The k-core of a graph G, as the union of the k-cores of the components of G implies, if G is a k-core, then every component of G is also a k-core.
Hence, we can naturally restrict our attention to connected graphs.
Each component of Ck(G) has order at least k + 1, and any order p ≥ k + 1 can be achieved. For example, G = K(p) is a k-core. The unique k-core of order k + 1 is K(k+1).

What is the relationship between Max K degenerate subgraphs and k-cores?
It is easily shown that the subgraph of a k-core is unique, the cores of a graph are nested, and therefore, it can be found by iteratively deleting vertices with degree less than k.
The maximum k such that G has a k-core is the maximum core number of G, graph has a 1-core those graphs call monocore.
Many common graph classes including trees and regular graphs are monocore.
To find the maximum k degenerate subgraph, a deletion sequence is formed by iteratively deleting a vertex of smallest degree, and a construction sequence reverses a deletion sequence.
Following these basic results, we can find the "k-shell" of a graph that is the subgraph induced by edges in the k-core and not in the
k + 1-core.
The 1-shell is a forest with no trivial components.
The structure of 2-cores and 3-cores is analysed and an operation characterization of 2-monocore graphs is presented.

Our task:
Maximal k-degenerate graphs are the upper extremal graphs. We are asked to find
k-degenerate of a given undirected graph and output such graph in the correct format.

Documentation:
To achieve an algorithm that perform efficiently, we had to turn towards other scholars that experimented more on the topic, therefore thanks to the references provided by the teacher and the various papers, websites, and blogs that I found online I could implement the following project.
For the preliminaries of the k-core computation this paper .
It basically proposes some of the pivotal pseudocodes for the implementation of our code.

Questions:

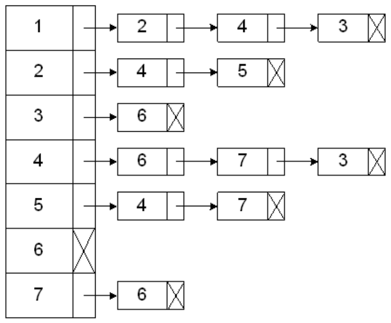
In graph theory and data structure there are two ways to represent a graph, namely adjacency matrix and adjacency list.

However to better grasp what those two are about, and importantly which one would be beneficial to the code later on would be to list out, their respective pros and cons, on this thread, I found out that the biggest properties or difference is that, one is space requirement while the

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

other is access time.

Since the adjacency matrix is a N-by-N array, either filled with true/false (if unweighted), or the weight of the edge. This requires O(N^2) space complexity. This is rather inefficient when the graph is sparse, meaning that there is a lot less edges than N^2 nodes.



However, for adjacency lists, you are only storing the nodes that is connected in a linked list or a vector. This results in much lower memory use as you are only storing the nodes that actually exists in the graph. This works nicely in all kinds of graphs, either dense or sparse, since the maximum space usage is always smaller than N^2.

Access Time

- In a adjacency matrix, it takes O(1) time to check the distance between two connected nodes, since you can just look it up in the array. However, in the adjacency list, you need to list all the nodes that the node is connected to, in order to find the other node of the edge

required.
Adjacency list is much more efficient for the storage of the graph, especially sparse graphs, when there is a lot less edges than nodes.

In terms of the accessing time, adjacency matrix is much more efficient when finding the relationships in a graph.
But adding a graph is O(n^2) complexity and detecting an edge is)(deg(vertex)) time complexity same along with remove a vertex, which in case would be a bit too slow.

- In contrast the required memory storage for the matrix is certainly larger, but if the graph has n vertex, therefore the complexity grows to theta(n^2)

  Adding vertex results in a time complexity of O (1), while detecting the edge between two vertex O(1), however removing a vertex is equivalent to complexity of O(n), same along with the degree of the vertex.

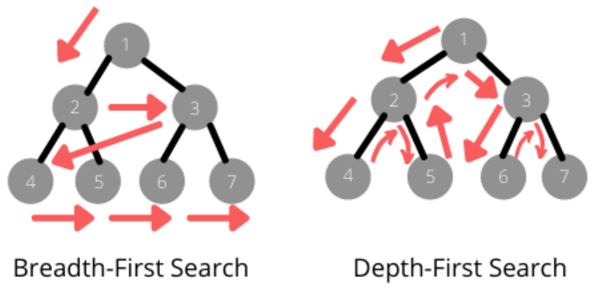
  Thus, using the adjacency matrix for our network of undirected graph is a better choice.

How to optimize the search of existing vertices?

• DFS/BFS from the lowest node degree?

A breadth-first search algorithm works best when counting nodes laterally, and a depth-first search algorithm works best when trying to count the height of a tree.

Typically, the breadth-first search algorithm would work best to find the max-width of the tree, and the depth-first search algorithm would work best to find the max-depth. Both algorithms run linear time in terms of performance, which means that we don’t have to consider other complex algorithms for binary tree traversal.



To find the max-width of the tree, we can use the depth-first search algorithm and for the max-depth, breadth-first traversal.

The time complexity of both DfS and BfS is equivalent to O(V+E) and to better settle our discussion this let’s take a look at the analysis of both of these algorithms.

DFS(analysis):

- Setting/getting a vertex/edge label takes O(1) time
- Each vertex is labeled twice
    - once as UNEXPLORED
    - once as VISITED
- Each edge is labeled twice
    - once as UNEXPLORED
    - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in O(n + m) time provided the graph is represented by the adjacency list structure
- Recall that $\Sigma_v \deg(v) = 2m$

BFS(analysis):

- Setting/getting a vertex/edge label takes O(1) time
- Each vertex is labeled twice
    - once as UNEXPLORED
    - once as VISITED
- Each edge is labeled twice
    - once as UNEXPLORED
    - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence Li
- Method incidentEdges is called once for each vertex
- BFS runs in O(n + m) time provided the graph is represented by the adjacency list structure
- Recall that $\Sigma_v \deg(v) = 2m$

Thus, we can picture that from the lower node degree, the breast fist search would make a better job in our code.

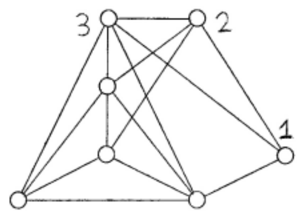
Comment on the reviewed papers

- The paper highlight that the in the notion presented, that implementations of the protocol introduced in on GraphChi and Webgraph are of major use until this day.
- Those two techniques aim to use only one consumer PC and not divide to achieve their goal.
- It also provide a guide towards an accurate implementation of the Batagelj and Zaversnik (BZ) algorithm for k-core decomposition in Webgraph.
- As well as an optimized implementation of the EMcore algorithm.
- We observe that the Webgraph implementation is faster, whereas the GraphChi implementation is slower than EMcore. However, the Webgraph implementation and EMcore have certain requirements on the available memory budget, whereas GraphChi does not.
- The cores of a graph always form an inclusion relationship.

Lastly the paper point to the different theoretical running time of each of the methods according to their implementation, and also prove their pseudocodes.
Therefore in our code implementation, we vow to stay as close as possible to the pseudocode, to achieve better performance. I also referred to some other implementations of this pseudocode that I reference in the bottom of the page.
In short, for the majority of the implementation of the code I focus on the Webgraph implantation, while giving interest to the Batagelj and Zaversnik Algorithm came only at the end.
The pseudocode to find the k-core of the graph is given as such:

```
function K-CORES(Graph $G$)
    $\mathbf{L} \leftarrow \mathbf{0}$, $\mathbf{d} \leftarrow \mathbf{0}$, $\mathbf{D} \leftarrow [\emptyset, \dots, \emptyset]$
    for all $i \leftarrow 0$ to $n$ do
        $\mathbf{d}[i] \leftarrow d_G(i)$
        $\mathbf{D}[\mathbf{d}[i]].insert(i)$
    end for
    for all $k \leftarrow 0$ to $d_{\max}(G)$ do
        while not $\mathbf{D}[k].empty()$ do
            $i \leftarrow \mathbf{D}[k].remove()$
            $L[i] \leftarrow k$
            for all $j \in N_G(i)$ do
                if $\mathbf{d}[j] > k$ then
                    $\mathbf{D}[\mathbf{d}[j]].remove(j)$
                    $\mathbf{D}[\mathbf{d}[j] - 1].insert(j)$
                    $\mathbf{d}[j]--$
                end if
            end for
        end while
    end for
    return $L$
end function
```

Basically, this computation of the K-core uses a set array that I call ‘Network’ in my code.
This sets of arrays in order to achieve the maximum k-degenerate subgraph needs to be “pruned “, that is by each iteration of the if the core is not maximum then we can move on to retrieve the maximum K-degenerate subgraph.
The principle behind this is illustrated below for a given graph G, by deleting the nodes l, 2, and 3, we can find the 3-degenerate graph. Which indeed is the maximum k-core subgraph corresponding to our original graph.
The size of a maximal k-degenerate with order n is k · n – (k+1)/2



Most of the properties given can be generalized with appropriate modification to all k-degenerate graphs. Our first result is the size of a maximal k-degenerate graph.

If G is k-degenerate, then its vertices can be successively deleted so that when deleted they have degree at most k:

Proof.

Since G is maximal, the degrees of the deleted vertices will be exactly k until the number of vertices remaining is at most k. After that, the n − j th vertex deleted will have degree j.
Thus, the size m of G is:

$$m = \sum_{i=0}^{k-1} i + \sum_{i=k}^{n-1} k = \frac{k(k-1)}{2} + k(n-k) = k \cdot n + \frac{k(k-1)}{2} - \frac{2k^2}{2} = k \cdot n - \binom{k+1}{2}$$

In this formula n represent the Order of the graph, that is the number of vertices in the graph.
Size of a graph is the number of edges in the graph.
We tried using the two graphs given by the assignment as well as some other undirected graphs that are larger for more flexibility, found on this website.
One property that retained my attention is that for for k-core-free graphs, maximal and maximum are equivalent.

And as a result, every graph with order n, size of $m \geq (k-1)n - \binom{k}{2} + 1,\ 1 \leq k \leq n-1$
Has a k-core.

GRAPHCHI IMPLEMENTATION

- GraphChi is based on communicationbetween vertices
- a computation on a graph runs an update(v) function operating on a vertex v by accessing and modifying its value along with its attached edges.
- Function update(v) is carried out for each scheduled vertex iteratively.
- Uses Parallel Sliding Windows (PSW), for processing very large graphs from external memory
- Uses selective scheduling

As will be shown later in our implementation part, we use these basic properties of maximal k-core graphs to implement a relatively efficient algorithm using brute force algorithm, to find the maximum k-core and k-degenerate subgraph in the given simple undirected graph.

PART II.

In our main implementation, several steps are taken into consideration:

1. Given the initial network (made of an undirected graph), will the network have a remaining k-core after the pruning process?
2. Where is the corresponding critical point?
3. What is this critical behavior?
4. If k-core exists, what is its network structure?
5. And finally output the answer in the correct format.

The code works looks like this:

Code implementation

1	`void freeNetwork(bool **Network)`	`//this help us save memory by erasing the network from memory after used.`	.
2	`void GetData(bool **Network)`	`//This function gets the input from the automatic handler and pasted it to our two variables x and y, that later on gets added to our network`	
3	`int *getNotches(bool **Network)`	`//gets the values of each of the notches.`	
4	`bool **deleteVertex(bool **Network, int redundant)`	`//deletes the vertices until there's none for the core.`	
5	`bool NullTab(bool **Network)`	`//this function is actual a checkpoint to see if we erased the graph, then we would know that we have passed the maximum`	

6	`void Pruned(bool **Network)`	//this function implements the pruning operation which consists of getting rid of the already traversed part of our network.	

7	`int main ()`
	`{`
	`bool **Network = Adj_Network();`
	`GetData(Network);`
	`bool **Output;`
	`for (int k = 1;; k++)`
	`{`
	`    Output = Prune_core(k, Network);`
	
	`    if (NullTab(Output))`
	`    {`
	`        Pruned(Output);`
	`        Output = Prune_core(k - 1, Network);`
	`        cout<<(k-1)<<"-core"<<endl;`
	`        break;`
	`    }`
	`    else`
	`    {`
	`    Pruned(Output);`
	`    }`
	`}`
	`Print_Max_KCore(Output);`
	`freeNetwork(Network);`
	`Pruned(Output);`
	
	`return 0;`
	`}`
		//this for loop takes into consideration the k value until it reaches the maximum core, then it would break.
		
		//the if statement guaranties that we have a maximum k core (k-1), which is also output in our final results first part.
		
		//means we are done with the iterations to find the maximum core.	

Brute force

Pros:
The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.
It is a generic method and not limited to any specific domain of problems.
The brute force method is ideal for solving small and simpler problems.
It is known for its simplicity and can serve as a comparison benchmark.

Cons:
The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.
This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
Brute force algorithms are slow.
Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.

Brute force is detailed more in this post:

However, the steps we use in our code are the following:

- First for all the vertices in our graph, with found the degree, after that if the degree of this particular vertex is smaller than our current K-core value we proceed to remove it from our working Network.

- And lastly we repeat the above process until all the degree of the vertices in our graph are actually larger than k-core value.

PART III

1. **CHALLENGES/DIFFICULTIES FACED:**

- **Choice of programming language:**

  In the beginning I wanted to use python as the language of choice since it's my favorite, but due to the competition nature of the assignment, and the fact that python most of the testcases perform not as well as the others, I decided to go with C++ instead.

- **For the assignment(coding)**

  I also noticed that the computation of the k-core is relatively straightforward, however for the maximum k degenerate subgraph, major research has been made to understand how to it efficiently, instead of pasting the whole array in a function for example, just paste the address to this one, and therefore save lot of computational memory, see **memory allocation(malloc/new/delete) in cpp**

  Before struggling with this issue, I also faced how the input should be taken and the hints come up later kind of helped a bit on that.

- **Language barrier**

  C++ is not my preferred programming language therefore going back to it, (because python was quite slow) was kind of tough in the beginning, but after a while it got better quickly. At the end I think it was fruitful. Furthermore, I used this testcases generator website to generate testcases for larger undirected graphs.

Conclusion:

Getting the maximum k-core degenerate is a very meaningful and popular algorithm in graph theory used extensively across a wide range of applications.

Through this assignment, though a bit tough at first, I learned a lot of important concepts and boost my abilities even further.

2. References

https://towardsdatascience.com/two-of-the-most-famous-coding-interview-questions-9746a4111011

https://github.com/andy489/Data_Structures_and_Algorithms_CPP

https://www.thedshandbook.com/depth-first-search/#:~:text=The%20worst%20case%20occurs%20when,%7C%20%2B%20%7CV%7C%20).

https://poloclub.github.io/papers/18-bigdata-kcore.pdf

https://algs4.cs.princeton.edu/41graph/

https://scholarworks.wmich.edu/cgi/viewcontent.cgi?article=1507&context=dissertations

Wasserman, S. and Faust, K. (1994) Social Network Analysis: Methods and Applications, Cambridge University Press, Cambridge.

Garey, M. R. and Johnson, D. S. (1979) Computer and intractability, Freeman, San Francisco.

Seidman S. B. (1983) Network structure and minimum degree, Social Networks, 5, 269–287.

Batagelj, V. and Mrvar, A. (1998) Pajek – A Program for Large Network Analysis, Connections, 21 (2), 47–57.

Beebe, N.H.F. (2002) Nelson H.F. Beebe's Bibliographies Page, http://www.math.utah.edu/~beebe/ bibliographies.html.

Jones, B., Computational Geometry Database, February 2002, ftp://ftp.cs.usask.ca/pub/geometry/, http://compgeom.cs.uiuc.edu/~jeffe/compgeom/ biblios.html.

Batagelj, V. and Zaverˇsnik, M. (2002) Generalized cores, submitted.

Batagelj, V., Mrvar, A. and Zaverˇsnik, M. (1999) Partitioning approach to visualization of large graphs, In Kratochv´ıl, J. (ed), Lecture notes in computer science, 1731, Springer, Berlin, 90–97.

Batagelj, V. and Mrvar, A. (2000) Some Analyses of Erd˝os Collaboration Graph, Social Networks, 22, 173–186.