Application of k-core decomposition method in connectivity graphs based on C#

Mihajlo Mulić

Faculty of Sciences
University of Novi Sad

This dissertation is submitted for the degree of
Master of Science (MSc)

October 2024

Table of contents

List of figures **iv**

List of tables **v**

1 Introduction **1**

1.1 Graph theory 2
1.2 Large scale network data 4
1.3 Connectivity graphs 6

2 *K-core* decomposition in large scale graphs **11**

2.1 *K-core* graph decomposition 11
2.2 *K-core* for unweighted graphs 14
2.3 *K-core* for weighted graphs 15
2.4 Python implementation 20

3 Efficient solution for k core decomposition in C# **24**

3.1 C# and .Net overview 24
3.2 Benchmarking system overview 28
3.3 Efficient algorithm for weighted k-core decomposition 30
3.4 C# Algorithm implementation 32

4 Results **37**

4.1 Performance comparisons 37
4.2 Current implementation and potential improvements 39
4.2.1 Data structures 39
4.2.2 Sorting algorithms 41
4.2.3 Potential for concurrent execution 42

5 Conclusion **44**

References 45

Appendix A Ključna dokumentacijska informacija 47

Appendix B Key words documentation 50

List of figures

1.1 Example of a graph. 3
1.2 Graph plot of first 100 edges 7
1.3 Graph plot of few nodes zoomed in 8
1.4 Graph plot after adding mode nodes zoomed in 9
1.5 Regular grid over spatial area of Milan city [12] 10
1.6 Example of the final edge list file, first 10 lines 10

2.1 The example of graph cores [11] 12
2.2 Batagelj and Zaversnik's algorithm 1 [3] in pseudo-code 14
2.3 Graph core decomposition by base algorithm [3] 16
2.4 Batagelj and Zaversnik's algorithm 4 [3] in pseudo-code 17
2.5 Min Heap and its opposite Max Heap 18
2.6 Zhou - Huang - Hua's Algorithm [14] in pseudo-code 19
2.7 Python code Interpreter 21
2.8 Batagelj and Zaversnik's algorithm 4 core implementation in Python 23

3.1 CLR flow 25
3.2 CLR components 26
3.3 JIT compiler 27
3.4 C# data consumption 33
3.5 Total weight of vertex 34
3.6 Edge class structure 34
3.7 Main body of the algorithm 36

4.1 Comparison of common data structures 41
4.2 Comparison of common sorting algorithms 42
4.3 Difference between multiprocessing and multithreading 43

List of tables

1.1 Statistics over graph vertices, edges and weights. 8

4.1 Python implementation statistics 38
4.2 C# implementation statistics 38

Chapter 1

Introduction

In the modern societies people live their day to day lives constantly interacting with technology, producing more and more data with each interaction. To catch a glimpse of the meaning and valuable information in that immense pool of data we need both powerful technology and powerful, well optimized algorithms. From that need to have more efficient ways to analyse the data some new scientific fields emerged such as Big Data, Data Science, Network Science, etc. Particularly interesting is Network Science [1], which is a relatively young academic field which studies complex networks such as telecommunication networks, computer networks, web networks, biological networks, social networks etc. Any complex relational data could be presented in the form of a network, which opens up a new potential to inspect the data and relations in completely different way using methods from Network Science. We live in the era of Big Data and complex systems that grow hopelessly complicated.

Traditionally the study of complex networks has been strongly related to the graph theory. When we talk about networks we often imply to graphs because graphs are mathematical representations of networks. Although a network and a graph is essentially the same thing, there is subtle difference between the two terminologies. The network consists of the nodes connected by edges, and they often refer to real systems such as WWW, or metabolic network or telecommunications, etc. In contrast, when we use the terms graph, vertex, edge, we usually discuss the mathematical representation of these networks. However, in practise and among professionals this distinction rarely made, so the terminologies are considered as synonyms.

To extract meaningful knowledge about complex networks that are represented using graphs we can apply various algorithms. Commonly applied are traversal algorithms that are used to find the shortest paths in the graph, graph cycles or graph directions, algorithms to find minimum spanning tree, connected components, clustering algorithms, etc. In this thesis

we would focus on one special group of algorithms that are used to find the subset of vertices in the graph based on their importance for connectivity in the graph called *K-cores*.

1.1 Graph theory

Graph theory is field in mathematics that study graphs, which are mathematical structures used to model pairwise relations between objects. Its roots go back to 1735 in Königsberg, when Leonard Euler, a Swiss born mathematician was trying to solve the problem of Seven Bridges of Königsberg[1].

The problem consist of the question: Can one walk across all seven bridges and never cross the same one twice? Euler represented the land areas separated by the river with single point for each area. Next he connected with lines each piece of land that had a bridge between them. He thus built a graph, whose nodes were pieces of land and links were the bridges. Then Euler made a simple observation: if there is a path crossing all bridges, but never the same bridge twice, then nodes with odd number of links must be either the starting or the end point of this path. Indeed, if you arrive to a node with an odd number of links, you may find yourself having no unused link for you to leave it. A walking path that goes through all bridges can have only one starting and one end point. Thus such a path cannot exist on a graph that has more than two nodes with an odd number of links. The Königsberg graph had four nodes with an odd number of links, so no path could satisfy the problem. Eulers proof that such path does not exists was the first time someone solved a mathematical problem using a graph.

Before presenting some applications and properties of graphs, we need to introduce some basic terminology. A *graph* G is the tuple (V, E) which consists of a finite set V of *vertices* and a finite set E of *edges*, where each edge is a connection between pair of vertices [13]. The two vertices associated with an edge e are called the *end-vertices* of e. An edge between two vertices u and v are often denoted by (u, v). The set of vertices of a graph G is denoted by V(G) and the set of edges of G by E(G). Let $e = (u, v)$ be an edge of a graph G. Then the two vertices u and v are said to be *adjacent* in G and the edge e is said to be *incident* to the vertices u and v. The vertex u is also called a *neighbor* of v in G and vice versa. The graph in Figure 1.1 has seven vertices a, b, c, d, e, f, g and ten edges. Vertices a and b are end vertices of edge (a, b). So, a and b are adjacent. Vertices b, c and f are the neighbors of the vertex a.

Graphs are data structures that have applications in many science and engineering disciplines, but to efficiently use them for large complex networks analysis we need to develop

[1] The Seven Bridges of Königsberg problem

fast and efficient algorithms. Some of the most common algorithms for graph analysis are [6]:

- Graph Traversal and Search - Depth-First Search (DFS) and Breadth-First Search (BFS) can be used to explore the structure and properties of graphs.
- Graph Isomorphism - Algorithms that determine if two graphs are structurally identical, such as the Weisfeiler-Lehman algorithm.
- Minimum Spanning Tree (MST) - Kruskal's and Prim's Algorithms can be used to find the MST, which can help analyze network structure and connectivity.
- Strongly Connected Components - Tarjan's Algorithm and Kosaraju's Algorithm identify strongly connected components in directed graphs.
- Graph Clustering - Techniques like spectral clustering use eigenvalues of matrices associated with graphs to identify clusters.
- Graph Partitioning - Algorithms like the Kernighan-Lin algorithm can be used to partition a graph into subsets while minimizing the number of edges between them.
- Network Flow Algorithms - Ford-Fulkerson Algorithm: Determines the maximum flow in a flow network.
- Random Walks - Random walk algorithms can help analyze the structure of a graph and are used in PageRank, which ranks web pages.

These algorithms for graph analysis can solve different problems and provide better insight to the data, but one specially interesting, which have many applications in the filed of complex networks is the problem of **graph decomposition**. Graph decomposition aims to find the subset of a given graph for which we would know its importance based on the connectedness property, or simply how well is the subset of the graph connected to the rest of the graph. The problem is similar to the graph clustering problem, where we want to find the subset of nodes that are strongly connected between each other, but with graph decomposition we want to find the subset of "great" importance.

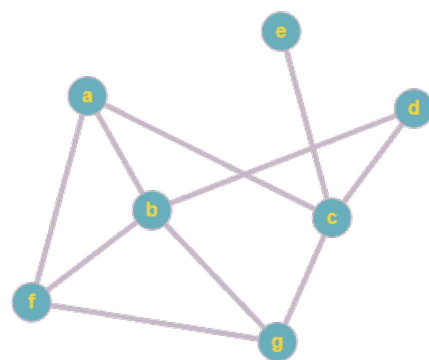

Fig. 1.1 Example of a graph.

Simply stated, the core decomposition of a network (graph) assigns to each graph node v, an integer number $c(v)$ (the core number), capturing how well v is connected with respect to its neighbors. This concept is strongly related to the concept of graph degeneracy, which has a long history in Graph theory [11].

There are several concepts and methods that could be used to detect cohesive group of vertices or subgraphs such as cliques, n-cliques, n-clans, n-clubs, k-plexes, etc., but for most of them it turns out they are algorithmically difficult. However, there is one technique that proved to have higher efficiency, its *k-core* graph decomposition, which would be further described in the next Chapter.

Next, we will describe the data that we used for experimental part of this thesis, data preprocessing and graph preparation.

1.2 Large scale network data

We live in the era of great complexity. Everything around us is hopelessly interconnected and we are using so much technology in our every day lives that generates new data with each interaction that our digital footprints extend our ability to comprehend them. Nevertheless, the run for more and more data and novel ways to analyse them and to extract meaningful knowledge and patterns from it is never been greater. Our ability to reason and comprehend our world requires the coherent activity of billions of neurons in our brain. Our biological existence is rooted in seamless interactions between thousands of genes and metabolites within our cells. These systems are collectively called complex systems, capturing the fact that it is difficult to derive their collective behavior from a knowledge of the system's components. Given the important role complex systems play in our daily life, in science and in economy, their understanding, mathematical description, prediction, and eventually control is one of the major intellectual and scientific challenges of the 21st century [1].

The domains where we can apply the knowledge of complex systems are diverse, ranging from biology to power grid networks. Some examples of large scale complex networks are:

- The *cellular network* contains the interactions between genes, proteins, and metabolites integrates these components into live cells.
- The *neural network* holds the key to our understanding of how the brain functions and to our consciousness, but in computer science it is also the heart of AI.
- In the recent time *social networks* are fluid of our society and determines the spread of knowledge, behavior and resources.

- *Communication networks*, describing which communication devices interact with each other, through wired internet connections or wireless links, are at the heart of the modern communication system.
- The *power grid network*, a network of generators and transmission lines, supplies with energy virtually all modern technology.
- *Trade networks* maintain our ability to exchange goods and services, being responsible for the material prosperity.

The study of large scale complex systems has been revolutionized by the unprecedented amount of digital records that are constantly being produced by human activities such as accessing Internet services, using mobile devices, and consuming energy and knowledge [2].

The overall extensive use of mobile phones and the exponential increase in the use of Internet services is generating an enormous amount of data that can be used to provide new fundamental and quantitative insights on socio-technical systems [2]. One type of mobile phone data is specially interesting in the area of computational social sciences, it is the so called Call Detail Records (CDR). The Call Detail Records (CDRs) of the 6.8 billion mobile phone subscribers worldwide, potentially represent the most invaluable proxy for people's communication and mobility habits at a global scale. The availability of these data is indeed defining a novel area of research that exploits CDRs to extract human mobility patterns and social interactions [9, 5], estimates population densities, models cities structures, predicts socio-economic indicators and outcomes of territories, and models the spread of diseases [10], and many more.

As we previously noted, one of the most valuable real world large scale data are telecom CDR data. Each time a user makes any interaction using mobile phone (sms, call or internet traffic) one record is generated and stored in telecom operator database. These records are used for billing purpose of the operator, but they contain much more valuable information about general people activity. These records are very sensitive from user privacy perspective and telecom operators are obliged to keep the safe and away from any external usage. In some cases scientific community makes an agreement with telecom operator to provide the anonymized data sets for the research purpose. In 2015 Telecom Italia opened some of their data sets for the purpose of Big Data Challenge and those multi source data sets are still available for research [2].

In this Thesis we will use telecom CDR data as an example of large scale network data and for further analysis we will make connectivity graphs from it.

1.3 Connectivity graphs

One way to analyse CDR data is through connectivity graphs. As described earlier, each time a user makes an interaction using mobile phone (sms, call or internet traffic) one record is generated and stored in telecom operator database. That one record represent connection between two radio base stations in telecom operators network and those networks can be represented as graphs. With the data that we are using, those connections are aggregated to time intervals and real locations of radio base stations are approximated with regular grid cells, so connectivity between radio base stations is actually connectivity between grid cells. As these telecom data graphs are large and have very dense structure it is not simple to visualize them.

In Figure 1.2 is presented graph structure plotted using Python NetworkX library, only first 100 edges from the graph. As we can see from the Figure 1.2 these nodes seem clustered and not well connected, but that is only because we plotted just part of the graph for the simplicity. In Figure 1.3 is presented graph structure of only few nodes zoomed in from previous graph plot. If we continue adding nodes to the graph plot we will see how structure becomes more dense and graph more connected, Figure 1.4. By using NetworkX library we can only plot parts of the graph, because our graph contains more than 6 000 000 edges and NetworkX can not plot more than 1000.

In this Thesis we would not explore the diverse context of the data but we will focus on computational and algorithmic challenges to extract knowledge from such complex data sources, particularly from CDR data. We will use CDR data for Milan city and make connectivity graphs from those records. Data is already processed by the operator prior to releasing it for research, with the aim to anonymize the records and to preserve the true location of operators Radio Base Stations hidden. To anonymize the data operator performed the aggregation of telecom traffic between two Radio Base Stations over time interval of 10 minutes and assigned the *weight* value to each edge that reflect the real amount of traffic. Since the data is georeferenced, to keep the true location of Radio Base Stations, the operator distributed the traffic over regular grid with 10 000 cells. In this grid, each cell represent one Radio Base Station in telecom network which refers to vertices in connectivity graphs. The traffic that was registered between two cells refers to edges in connectivity graphs. In Figure 1.5 is presented the regular grid over spatial area of Milan city [12].

Data is available for time period of two months, and in this research we would use only one graph for one day to perform benchmark testing. When the data is obtained, first step is to perform data preprocessing over raw data set. Our preprocessing include few steps:

1. Forming egde list from raw data following the structure (v1 v2 weight)

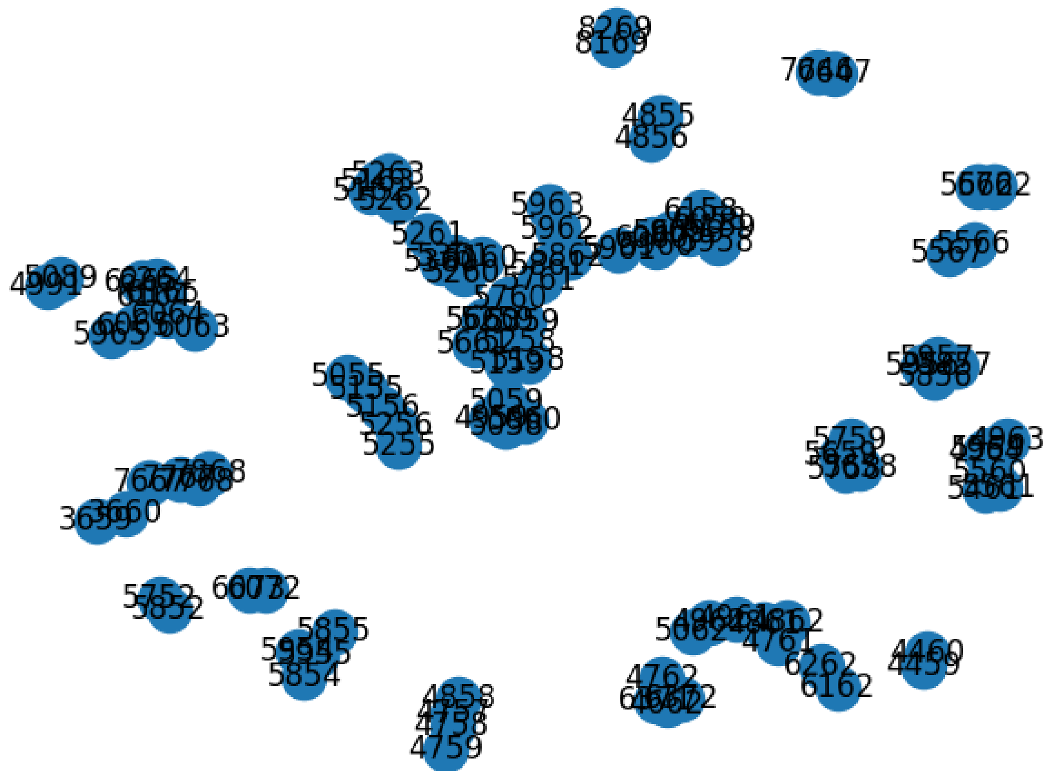

Fig. 1.2 Graph plot of first 100 edges

2. Performing additional aggregation of weight to obtain unique edges per one day
3. Cleaning the edge list from self loops and directions
4. Eliminating not significant edges based on weights
5. Rescaling the weights and converting them from double to integer

In the table 4.2 is presented some basic statistics over graph edges, vertices and edge weights. From table 4.2 we can see that graph has very dense structure with much greater number of edges then number of vertices. Also, we can see that the weights over the edges are not following any regular distribution with highest number of values between 0 and 1. Mean weight is also below 1 meaning that the whole weight scale is shifted to interval below 1. Considering the minimal value for weight, we were motivated to eliminate those edges that have weight less than 0.001, there is 202 695 edges with such small weight value. Finally, we rescaled the weight values by multiplying them with 1000 and converting them from double

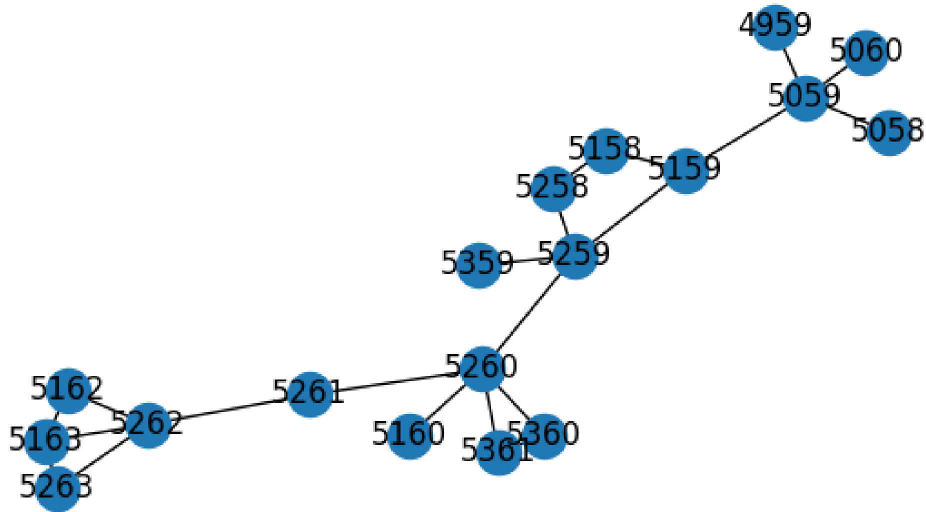

Fig. 1.3 Graph plot of few nodes zoomed in

to integer to preserve the memory space. In Figure 1.6 is presented the final structure of the edge list file. The file follows the structure (v1, v2, weight, rescaled weight).

Table 1.1 Statistics over graph vertices, edges and weights.

Number of edges	6 463 696
Number of vertices	10 000
Min weight	2.647e-05
Max weight	92.062
Mean weight	0.049
Number of weights between 0 and 1	6 434 211
Number of weights between 1 and 10	28 753
Number of weights between 10 and 100	732

From this brief statistical analysis we can conclude that the greatest challenge with analysing real world telecom connectivity graphs is their very dense structure with high

Fig. 1.4 Graph plot after adding mode nodes zoomed in

number of edges that also have the weight attribute which should be taken into account during any knowledge extraction from the data.

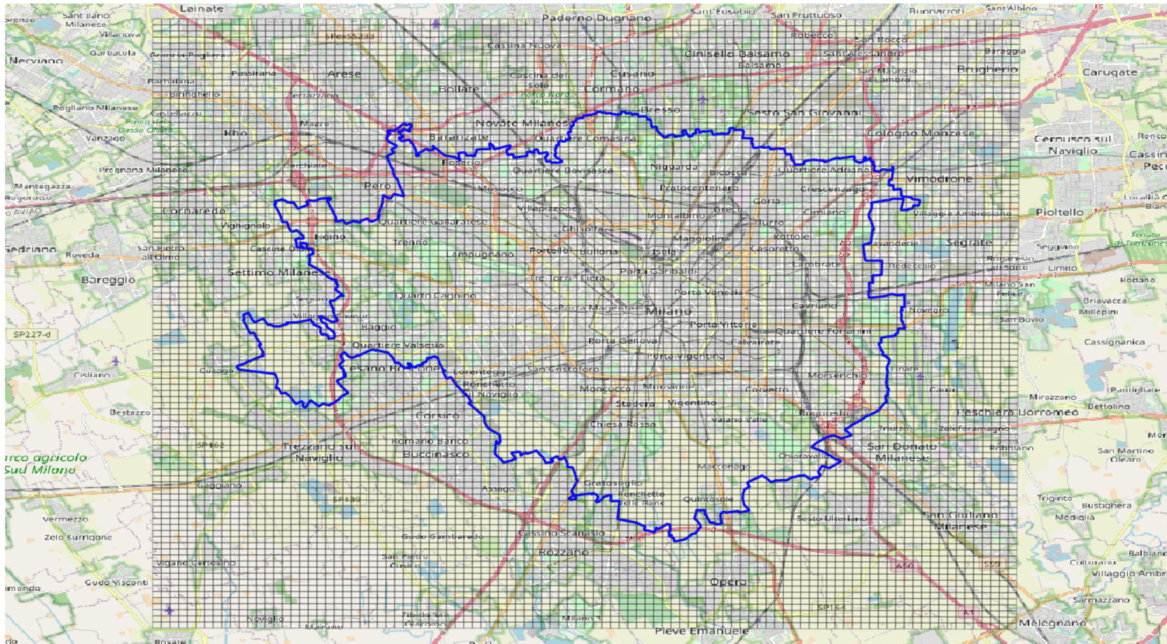

Fig. 1.5 Regular grid over spatial area of Milan city [12]

```
1   v1,v2,weight,re_weight
2   71,948,0.0023488317880019,2
3   71,5157,0.0048216759538691,5
4   71,1058,0.0036383186226935,4
5   71,1160,0.0032786571327228,3
6   71,1459,0.0024332263157195,2
7   71,1853,0.0040916620159256,4
8   71,471,0.0111988160139328,11
9   71,679,0.0021214638888811,2
10  71,79,0.0144733812088955,14
11  71,4456,0.007624381754262,8
```

Fig. 1.6 Example of the final edge list file, first 10 lines

Chapter 2

K-core decomposition in large scale graphs

One of the most common problems in Graph Theory is finding the subset of the graph that has significant importance to overall information flow through the graph. *K-core* decomposition is one approach to the problem, because knowing the *core number* of the node would give us an insight into how important the node is in the graph. Simply speaking, the higher the core number of the specific node, the importance of the node is greater because the nodes in the higher core are more connected and have higher degree. Graph core decomposition would give us the subsets of the graph depending on the importance of the nodes, which is in some use cases very valuable information. When we are working with large scale graphs from real world data such as Web connectivity graphs, telecom data graphs, social media graphs, etc. we are working with millions of nodes and edges which makes any kind of knowledge extraction difficult and computationally intensive, that is why the algorithms that would provide the subset of a graph with it's importance are bringing so much attention in the network science community. In this thesis we would focus specifically to *K-core* decomposition in large scale graphs.

2.1 *K-core* graph decomposition

The core decomposition of networks has attracted significant attention in science and research due to its numerous applications in real-life problems. Simply stated, the core decomposition of a network (graph) assigns to each graph node v, an integer number c(v) (the core number), capturing how well v is connected with respect to its neighbors [11]. Although the core decomposition concept is extremely simple, it is still largely explored field mainly due

to its capability to analyze a network in a simple and concise manner by quantifying the significance of graph nodes. *K-core* is a property that is important both as global graph property because it tells a lot about graphs connectivity, as well as local property of the nodes because it denotes the importance of the node in the network. Next, we will provide formal definition of graph cores.

A graph is denoted by G(V, E), where V is the set of nodes or vertices and E is the set of edges or links. The number of nodes is n = |V| and the number of edges is m = |E|. The number of neighbors of a node $u \in V$ plays a central role in general, and it will be denoted by *deg(u)*.

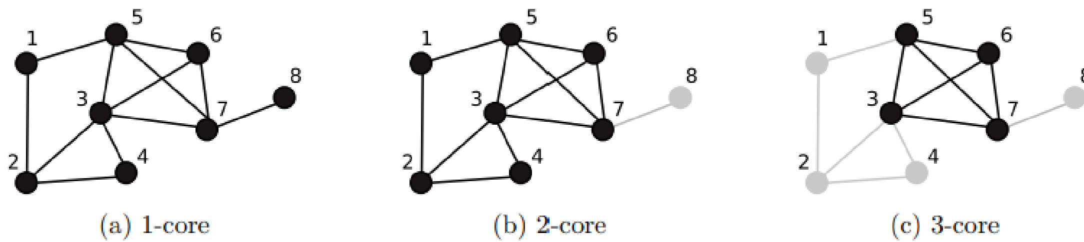

Fig. 2.1 The example of graph cores [11]

Figure 2.1 presents a simple graph G(V, E) with n = 8 nodes and m = 12 edges. Based on the degree definition, *deg(v1)* = 2, *deg(v4)* = 2 whereas node *v3* has the highest degree *deg(v3)* = 5. Therefore, node *v8* has the smallest degree and node *v3* the highest. From Figure 2.1 we can see that 1-core corresponds to the nodes that have the degree at least 1, 2-core corresponds to the nodes that have the degree at least 2, 3-core corresponds to the nodes that have the degree at least 3, etc. Intuitively it is clear what a *core number* represents in a graph.

Exploring and analyzing massive complex networks involves the execution of (usually) computationally intensive tasks, aiming at uncovering the network structure and detecting the presence of useful patterns that could be proven significant. Some important graph mining tasks involve: reachability queries, graph partitioning, graph clustering, classification of graph nodes, predicting network evolution, discovering dense subgraphs, detecting influential spreaders [11].

In many cases we are searching for graph nodes that are considered **"central"** with respect to a specific problem at hand. Therefore, the concept of node importance is crucial in network analysis, since it is expected that among the nodes of a massive network, only a small fraction is of high significance. Evidently, one should first determine a method to quantify this significance (importance), since this concept is highly related to the application and context of the network. One of the popular measures of importance is the total number of shortest paths passing through a specific node (also known as betweenness centrality [7]).

Also, one can quantify node importance using the concept of random walks and applying techniques similar to PageRank [4]. In such a case, the importance of a node is represented by the probability that this node will be visited by a random walker.

The concept of *core decomposition* can be used efficiently and effectively to quantify node importance in many different domains, while avoiding the use of more complex and computationally intensive algorithmic techniques. To be precise, the *core decomposition* of a simple graph G can be computed in linear time with respect to the number of edges of G, if the computation is done in main memory. Simply put, the *k-core* of a graph G is the maximal induced subgraph G_k, where the number of neighbors of every node u in G_k is at least k. The core number of a node u (c(u)) is defined as the maximum value of k such that u is contained in G_k.

The k-core decomposition in general case considers that graphs are unweighted and undirected. However, many real-world networks carry rich semantics, as expressed by more complex graph types. To that end, there exist research efforts towards meaningful extensions of the k-core decomposition to other types of graphs. In most of the cases, these extensions pose additional challenges to the efficient computation of the decomposition as well [11]. Two most common extensions are for the cases of directed or weighted graphs, because many real world networks have those characteristics.

Directed graphs or digraphs are characterized by rich semantics in comparison to simple graphs, simply because edge direction is important. In a directed graph the degree of a node u may refer to the number of incoming links *degin(u)* or to the number of outgoing links *degout(u)*. These are also known as the *in-degree* and the *out-degree* respectively. Giatsidis et al. [8] introduced D-cores, an extension of the *k-core* structure to directed graphs. In this case, the notion of *(k, l)-core* is used to represent subgraphs in which all nodes have in-degree at least k and out-degree at least l respectively [11].

Another spacial case of the graph is weighted graph, more details of that case would be described in this Thesis. A weighted graph is characterized by the existence of weights on the graph edges. Each edge e is associated with a weight *w(e)* that may represent the cost of the edge, or the strength of the link between the participating nodes, or any other type of quantification, depending on the application. Computing the core decomposition in a weighted graph is significantly harder than the computation in a simple graph, mainly because there is no easily derived bound on the core number of a node [11]. In [3, 14], the authors propose efficient algorithms for computing the core decomposition in weighted graphs.

Core decomposition is a powerful tool for analyzing complex networks and it is proved to be more efficient comparing to other techniques for network analysis. By focusing on

densely connected subgraphs, researchers can gain insights into the network's structure and dynamics, and also time evolution of the network. In the following Chapter we will describe in more details the general case of *K-core decomposition* for simple graphs and special case of graph *K - core decomposition* for weighted graphs.

2.2 *K-core* for unweighted graphs

Base case for any core decomposition should be for unweighted graphs. This makes a good starting point, and makes understanding of base and any subsequent Algorithm easier. Here we present the "base" Algorithm for computing K-core decomposition for unweighted graphs in Figure 2.2 [3].

INPUT: graph $\mathcal{G} = (\mathcal{V}, \mathcal{L})$ represented by lists of neighbors $Neighbors(v)$ for each vertex v
OUTPUT: table $core$ with core number $core[v]$ for each vertex v

1.1 compute the degrees of vertices;
1.2 order the set of vertices $\mathcal{V}$ in increasing order of their degrees;
2 for each $v \in \mathcal{V}$ in the order do begin
2.1 $core[v] := degree[v]$;
2.2 for each $u \in Neighbors(v)$ do
2.2.1 if $degree[u] > degree[v]$ then begin
2.2.1.1 $degree[u] := degree[u] - 1$;
2.2.1.2 reorder $\mathcal{V}$ accordingly
end
end;

Fig. 2.2 Batagelj and Zaversnik's algorithm 1 [3] in pseudo-code

Although we believe the pseudo code of the Algorithm is pretty self explanatory we will try to dig a bit deeper. We start from a graph that is represented by lists of neighbors for each vertex and we want to get core values for each vertex in this graph. In order to get these values, we start by computing the degrees of all the vertices in the graph, and storing them. After we have these values we should sort them in increasing order of their degrees.

As will be mentioned later, please note that this here is a base algorithm, and that more optimal data structures than lists can be chosen, as well as a fast sorting algorithm and similar optimizations. All these changes might not affect the theoretical worst case run time, but in real world will result in more than significant performance differences.

After data preparation the main body of algorithm begins, it consists of two nested for loops. We start by going through all the vertices in the sorted list of vertices by weight, Firstly, we assign core value of that vertex to be the degree, and for all it's neighbours we do the following. If the degree of neighbour is higher than the degree of initial vertex from first for loop, we subtract 1 from the value of degree of the current neighbour in iteration. And we finish by reordering the list of vertices (because the values changed). All this mentioned above is done in two loops, firstly we go through all the vertices, and then for each of them we go through all their neighbours and rearrange the list.

This algorithm guarantees we will not skip over any vertex, since at most we subtract 1 from the degree of neighbour vertices. This is because we are only focusing on unweighted graphs in this case, and every edge has the same weight-1. So there can be no leaping during sorting, that would result in improper behaviour of Algorithm described here. In the subsequent Chapter we will describe a more general scenario for the case graphs have weights, and since we have to worry about vertices leaping, since the weights can be greater than 1, we will have to adapt our algorithm to take this into account.

Although the Algorithm itself is not overly complex, we should pay significant attention to the details of the implementation, the subroutines, data structures and general optimally of code that is used, because as we will see later depending on this run times can vary dramatically.

In Figure 2.3 we are presenting a case of K-core graph decomposition in a very simplified graph. We start by assigning 0 for core values to all unconnected vertices. Then all the vertices with degree 1 are removed, after sorting at each step, we proceed with examination of vertices with degree 2. Interestingly, as can be seen on the Figure the majority of vertices that have core of 2, started with higher initial degree, 3 or more. They get assigned 2 as core, because their neighbours had lower degree and were removed, after which action a lot of vertices got their degrees lowered by at least 1. The calculation for 3-core is identical to the iterations before, so will not go into further detail.

2.3 *K-core* for weighted graphs

As we mentioned before, K-core for unweighted graphs since it is significantly easier for calculation, mainly served as a good base ground for further explanations. In real world scenarios, rarely are unweighted graphs encountered, much more often we have a case of weighted and directed graphs. Direction of graph does not change the meaning in our particular case, since we are more concerned with the algorithm itself. It is of no significance, as it will be evident in examples to follow, if we use in-degree, out-degree or a sum of them.

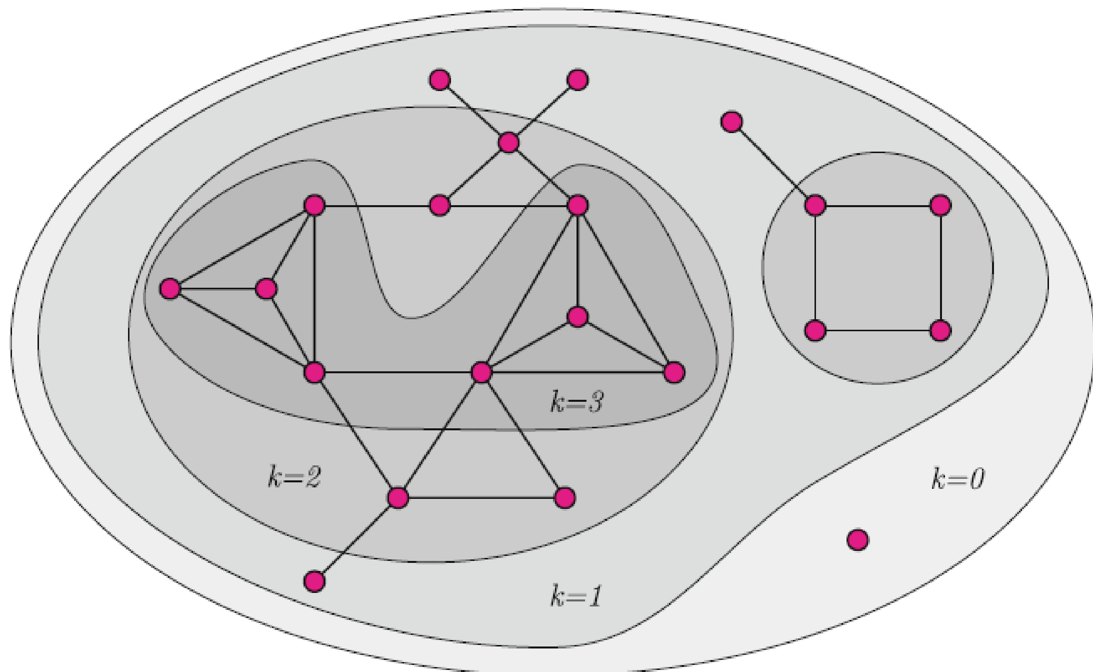

Fig. 2.3 Graph core decomposition by base algorithm [3]

The majority of complexity lies in weights, and potential leaping of vertices during sorting. For that reason we will discuss modifications to the definition and improved Algorithms in this chapter that handle weighted graphs.

We start by defining p-functions. **P-function** are vertex property function on a set of vertices. For example:

- degree of vertex for unweighted graphs
- in-degree for weighted graphs
- out-degree for weighted graphs
- sum of weighted in-degrees and out-degrees
- all similar monotone functions

We define monotone functions and their main properties as follows:

Definition 1 Monotone function is a function which is either entirely non-increasing or non-decreasing. Or in other words, a function is monotone if its first derivative does not change sign.

This term is commonly used to describe set functions which map subsets of the domain to non-decreasing values of the co-domain. In particular, if f:X->Y is a set function from a collection of sets X to an ordered set Y, then f is said to be monotone if for every A which is subset of B, both of which have elements from X, the following holds f(A)<=f(B).

We will continue by listing few Theorems and Corollaries that will enable us to claim that we have an efficient Algorithm for the weighted graph case, proofs will be omitted here, since most are a simple case of counter-example but can be studied in depth in the following paper [3]:

Theorem 1 For each monotone vertex property function p Algorithm determines the p-core at level t.

Corollary 1 For each monotone p-function p the cores are nested

Theorem 2 For a monotone and local vertex property function p Algorithm 4 determines the p-core hierarchy.

We will now present and describe the Algorithm 4 in more detail.

INPUT: graph $\mathcal{G} = (\mathcal{V}, \mathcal{L})$ represented by lists of neighbors,
monotone vertex property function p
OUTPUT: table *core* with core-value for each vertex

1. $\mathcal{C} := \mathcal{V}$;
2. for $v \in \mathcal{V}$ do $p[v] := p(v, N(v, \mathcal{C}))$;
3. $build_min_heap(v, p)$;
4. while $sizeof(heap) > 0$ do begin
4.1. $\mathcal{C} := \mathcal{C} \setminus \{top\}$;
4.2. $core[top] := p[top]$;
4.3. for $v \in N(top, \mathcal{C})$ do begin
4.3.1. $p[v] := \max\{p[top], p(v, N(v, \mathcal{C}))\}$;
4.3.2. $update_heap(v, p)$;
end;
end;

Fig. 2.4 Batagelj and Zaversnik's algorithm 4 [3] in pseudo-code

As before, we start from a graph represented by lists of neighbors, and we want to calculate core values for every vertex. The main difference being, we are not working in specific unweighted case. Now we also have a monotone vertex property function p. It can be any number of different functions, some of which are listed above. We are naturally mostly interested in sum of weights for weighted graphs.

We initialize a helper list that will keep track of visited vertices. Also we should calculate the values of defined p function for every vertex in set, and store the values. Afterwards,

we build a Min Heap data structure based on values p function values for each vertex we previously calculated. Now the main body of the algorithm is executed, it as before consists of two nested loops. The first one is a while loop that spins until there are elements in the min heap. Inside the loop, we start by removing the vertex that is on top of min heap from the temporary list of vertices we defined before. Then we assign core value for the vertex to be the value from the list of p-values we are keeping track. Finally we go over a list of every neighbour from the recently removed vertex and update the p-function values list for the neighbour to be maximum value of the top element that was recently removed or the value of recalculated p-value for the subset without the newly removed vertex. This last part is very important, as it enables us to bypass the problem we mentioned earlier, the leaping of nodes as they are removed. We finish the iteration by updating the min heap, and execute both loops again until the conditions for their termination are met.

Just for clarification, we will briefly go over Min Heap data structure explanation and key points. Min Heap is as a type of a Heap Data Structure in which each node is smaller than or equal to its children. While the heap data structure is a type of binary tree that is commonly used in computer science for various purposes, including sorting, searching, and organizing data.

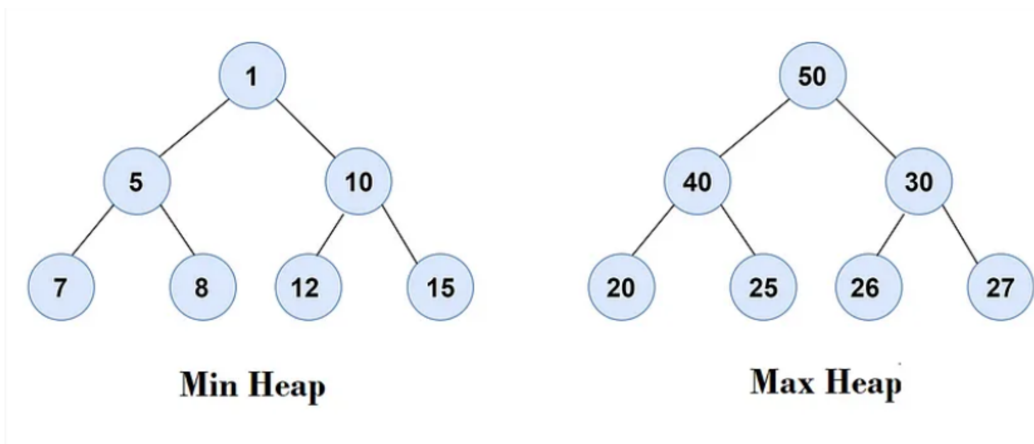

Fig. 2.5 Min Heap and its opposite Max Heap

Without going into formal Big O notation and similar methodology for determining the performance of data structures and algorithms. Min Heap has certain certain advantages that are not easily overlooked:

- Efficient insertion and deletion. Min heap allows fast insertion and deletion of elements with a rather low complexity.
- Efficient retrieval of minimum element. The minimum element in a min heap is always at the root of the heap, which can be retrieved instantly.

- Memory efficiency. Min heap is a compact data structure that can be implemented using basic data structures.
- Sorting speed. Min heap can be used to implement an efficient sorting algorithm.

Finally, we will be presenting an Algorithm by Zhou et al. [14] in Figure 2.6 which describes an alternative way of computing K-cores for weighted graphs. We believe this Algorithm to also be important, because it highlights similarities between different Algorithms, and enables us to better modify Algorithm 1 in next Chapter, and achieve best of both worlds, an real-world efficient algorithm for computing K-cores for weighted graphs.

```
Input  : Graph G; V: a set of vertices in G;
Output: Each vertex's weighted core number
Initialization
    foreach vertex u ∈ V do
        Compute the current degree cd[u]
    k ← 0 ;
    foreach u ∈ V do
        The weight core number k_u of vertex u ∈ V equals to 0
if V ≠ ∅ then
    repeat
        Find the vertex v ∈ V which has the minimum weighted degree
        if cd[v] > k then
            core_w(v) ← cd[v];
            k ← core_w(v);
        else
            core_w(v) ← k;
        foreach u ∈ N(v) do
            cd[u] ← cd[u] − w(v)
        Delete vertex v and its connected edges from V;
```

Fig. 2.6 Zhou - Huang - Hua's Algorithm [14] in pseudo-code

We we will briefly go over main idea of the algorithm. The basic strategy of the algorithm is deleting a vertex with the minimum weighted degree recursively. In each iteration, the vertex with the minimum weighted degree is found, whose weighted core number is then determined. The principle is that if its current weighted degree is larger than the weighted core number determined in the last iteration, this means that current vertex can be in a

weighted core with larger weighted core number, and so current vertex weighted core number is set to its current weighted degree, or weighted core number is set to be the weighted core number determined in the last iteration. At the end of the iteration, currently examined vertex and its connected edges are deleted.

Previously discussed algorithm and Algorithm 4 by Batagelj and Zaversnik, share a common feature and in this they mitigate the main concern with the Algorithm we discussed for unweighted graphs. A leap can happen because a neighbour has an edge with high enough weight so that the vertex we are currently examining is leaped when the set of vertices is sorted again. Instead of allowing this, both algorithms do not reduce the weight or core number of neighbouring edge if it is about to leap the edge that we are currently examining. They do this with different approaches, but the idea and end goal are the same. In Batagelj's algorithm maximum value is assigned in each iteration. While in Zhou's algorithm we will reduce the number of neighbor but we will keep assinging the higher core number as before.

2.4 Python implementation

Although Python is a great language for starting point for any algorithm, for several reasons:

- large community
- plenty of resources
- lower barrier for entry - easier to learn
- plethora of libraries, especially in the field of Machine Learning

The main drawback of implementing compute or memory heavy algorithms in Python will in majority of cases be the underlying execution speed. This is mainly due to the way Python code is run. If there is need to speed things up in Python, more often than not, the best approach would be to write the compute part in plain C programming language, and wrap it, so that it can be used from Python libraries. This step also includes some performance loss when compared to only writing everything in C, due to interoperability and the conversions that are necessary when data is transferred to and from the C part of library. In most cases, it is not a clear cut which program should be used. If we were looking at pure speed, everything would be written in plain C, if we were looking at worldwide adoption, library support or something similar Python would certainly be a serious contender. In our work here we have chosen C# because we believe it offers a nice blend of performance, support and readability as well as some other topics.

To go into a bit more depth, code written in Python is executed via the Interpreter in the following way.

When a Python program is compiled, the python compiler converts the Python source code into another code that is called byte code (named due to the size of each byte code instruction). This code can run on any Operating System and hardware. So mainly, byte code instructions are platform-independent.

In order to run byte code on a machine, we first need to convert it to a machine understandable code or machine code, those are the 0s and 1s that the computer executes in the end. To achieve this Python uses an interpreter called PVM (Python Virtual Machine), which understands the byte code, includes all the used libraries and converts everything into machine code.

Finally, machine code instructions can be executed by the CPU.

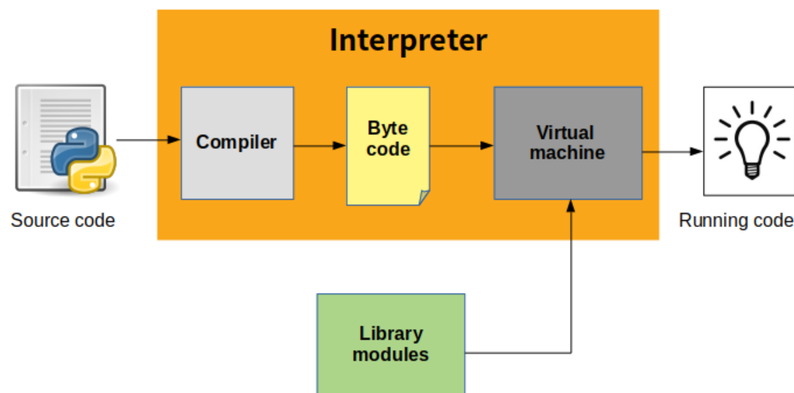

Fig. 2.7 Python code Interpreter

We started from a base algorithm for K-core decomposition for weighted graphs (generalized k-cores) written by Antoine J.-P. Tixier and only slightly improved as a baseline. The algorithm itself is based on Batagelj and Zaversnik's (2010) algorithm number 4 [3].

Algorithm 4 is discussed and explained at length in the previous Sub-chapter, K-core for weighted graphs, please refer to it if further clarification is needed.

Finally, we will explain Python implementation.

It is worth noting that this implementation is using IGraph library instead of some more popular alternatives since IGraph library has support for weighted edges, which is in the core of the problem of the dataset we are exploring.

In this explanation we have left out data consumption out of the more serious discussion to follow. But we do feel it is worth noting that this part also posed quite a serious performance challenge using Python. Although, the data was just consumed by IGraph library. The only

operations were creation of graph, initialization of node weights tuples to zero and computing the vertex strength by iterating over all the vertices.

Now we will describe the implementation of core algorithm in more details. We begin by cloning the graph using deep copy, this is done so that original data is also maintained. This is followed by necessary initialization, both the dictionary that will contain the core numbers and min heap that will contain degrees are created and populated by relevant data at this point. After neap data structure is reformatted, the main while loop begins. It iterates as long as there are elements in the heap. We take the top element from heap, meaning the smallest one. After we get its neighbours and setting the core number to the current value we delete this vertex from the graph. If the vertex had no neighbours, we just restructure the heap so that new minimal element is at the top. But if the removed vertex had neighbours, than the flow is a bit more complex. We go over all of its neighbours, and for each one, we calculate the maximum of the core value of the original vertex whose neighbours we are currently examining and the strength of the vertex in the newly updated graph (without the vertex that was removed previously), We den update this maximum as the new weight of the neighbour we are currently iterating. This step enables the use of this algorithm on weighted graphs, as we previously mentioned. This is because, by choosing maximum instead of just recalculating the weight for the neighbour we essentially mitigate any issues we could have with leaping vertices. Afterwards we just restructure the heap so that new minimal element is at the top. We run this for loop for all the neighbours of the initial vertex in questions. After it finishes, we will take another element from the top - vertex with the smallest weight, and continue the whole procedure. This is done until we go over all the elements from the initial graph. At the end we have a dictionary with keys as names and K-core values as values of the dictionary.

There will be a more comprehensive description of the algorithms performance with Python implementation in later Chapters. This will include comparisons with different implementations, relevant benchmarks, identified bottlenecks and potential areas for improvement. For now it is sufficient to note that performance with Python was sub-par, and was not sufficient to perform any deeper analysis on the data in any meaningful way, due to the size of the graph. In a way, poor Python performance prompted us to explore possibilities of alternative implementations and is partly responsible for the work that follows.

```python
def weighted_core_dec(g):
    # Cloning initial graph to perserve it
    gg = copy.deepcopy(g)
    # Initialization of dictionary that will contain the core numbers
    cores_g = dict(zip(gg.vs["name"], [0] * len(gg.vs["name"])))

    # Initialization of min heap containing degrees
    heap_g = list(zip(gg.vs["weight"], gg.vs["name"]))
    heapq.heapify(heap_g)

    while len(heap_g) > 0:

        top = heap_g[0][1]
        # Finding vertex index of heaps top element
        index_top = gg.vs["name"].index(top)
        # Saving names of its neighbors
        neighbors_top = gg.vs[gg.neighbors(top)]["name"]
        # Excluding self-edges
        neighbors_top = [elt for elt in neighbors_top if elt != top]
        # Setting core number of heaps top element as its weighted degree
        cores_g[top] = gg.vs["weight"][index_top]
        # Deleting top vertex (weighted degrees are automatically updated)
        gg.delete_vertices(index_top)

        if len(neighbors_top) > 0:
            # Iterating over top elements neighbors
            for i, name_n in enumerate(neighbors_top):
                index_n = gg.vs["name"].index(name_n)
                max_n = max(cores_g[top], gg.strength(weights=gg.es["weight"])[index_n])
                gg.vs[index_n]["weight"] = max_n
                # Updating heap
                heap_g = list(zip(gg.vs["weight"], gg.vs["name"]))
                heapq.heapify(heap_g)
        else:
            # Updating heap
            heap_g = list(zip(gg.vs["weight"], gg.vs["name"]))
            heapq.heapify(heap_g)

    return(cores_g)
```

Fig. 2.8 Batagelj and Zaversnik's algorithm 4 core implementation in Python

Chapter 3

Efficient solution for k core decomposition in C#

In this Chapter we cover several relevant topics with the end goal, an efficient algorithm implemented in C# programming language. We start by giving a brief overview of C# and .Net. In addition, we cover the most important part of .Net, CRL (Common Language Runtime) and JIT (Just In Time) compiler in more details. This is followed by an overview of the most important specification of the system on which tests for all the algorithms were run. We list both important hardware and software components to guarantee reputability. Meaning that the results that will be shown later will always be the same, if the machine with close enough specification is used. Afterwards, we optimized the k-core decomposition algorithm for the weighted graph case. We present the algorithm and give a brief commentary on its key points and its importance. Finally, we present our implementation of the previously mentioned algorithm in C# with necessary comments.

3.1 C# and .Net overview

We will briefly go over C# and .Net Framework basics. Then we will cover JIT compiler and its specifics. Finally we will compare C# and Python execution speed.

C# (C Sharp) is a modern, object-oriented programming language developed by Microsoft. It was introduced in the early 2000s as part of the .NET initiative. It's key features are:

- **Object-Oriented** - Supports concepts like encapsulation, inheritance, and polymorphism
- **Type Safety** - C# enforces type checking at compile time, reducing runtime errors

- **Versatility** - Suitable for various applications, including desktop, web, mobile, and game development
- **Rich Standard Library** - Provides a vast set of libraries for common tasks
- **Interoperability** - Can easily interact with other languages and technologies, such as C++, COM, and REST APIs

.NET Framework is a software development framework created by Microsoft that provides a platform for building and running applications. Its key component is CLR or Common Language Runtime. It is execution engine that manages running applications, providing services like garbage collection, exception handling, and security.

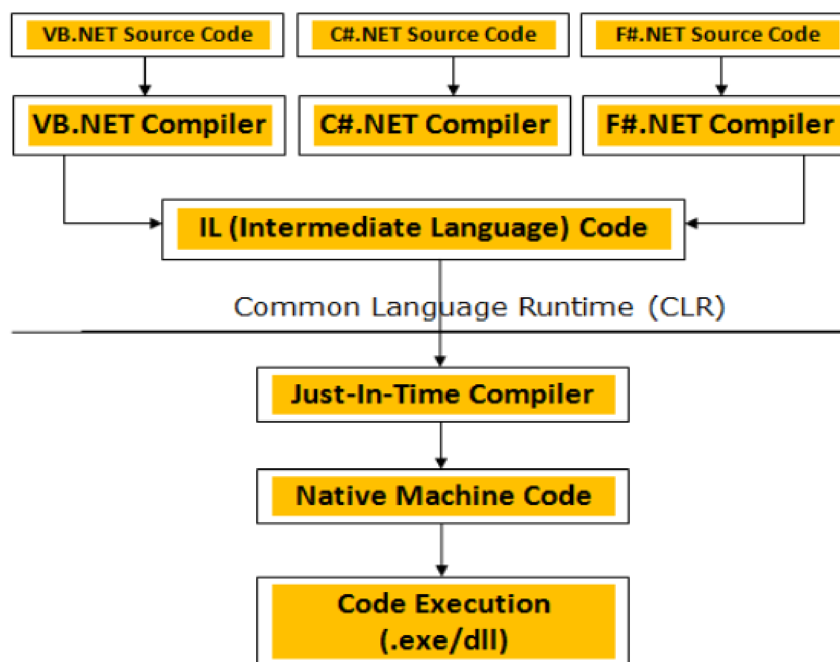

Fig. 3.1 CLR flow

In .NET, CRL (Common Language Runtime) is a heart of .NET Framework. Without CLR you can't imagine .NET Framework at all. When you write any program using any .NET language and run it, it won't run directly on your system. In order to run a program it is require to convert into binary so that operating system can understand and generate desire output.

Here it is important to understand that each system may have different system architecture and operating system. In order to run a program in any system, your program must be

converted into the system specific native code. To do this CLR comes into the picture, which takes program source code as input, compile and convert into the MSIL (Microsoft Intermediate Language) which consists of CPU-independent code and instructions which is platform independent. At run time this MSIL again convert into system specific native code to run the program.

CLR handles the execution of code and provides useful services for the implementation of the program. In addition to executing code, CLR provides services such as memory management, thread management, security management, code verification, compilation, and other system services.

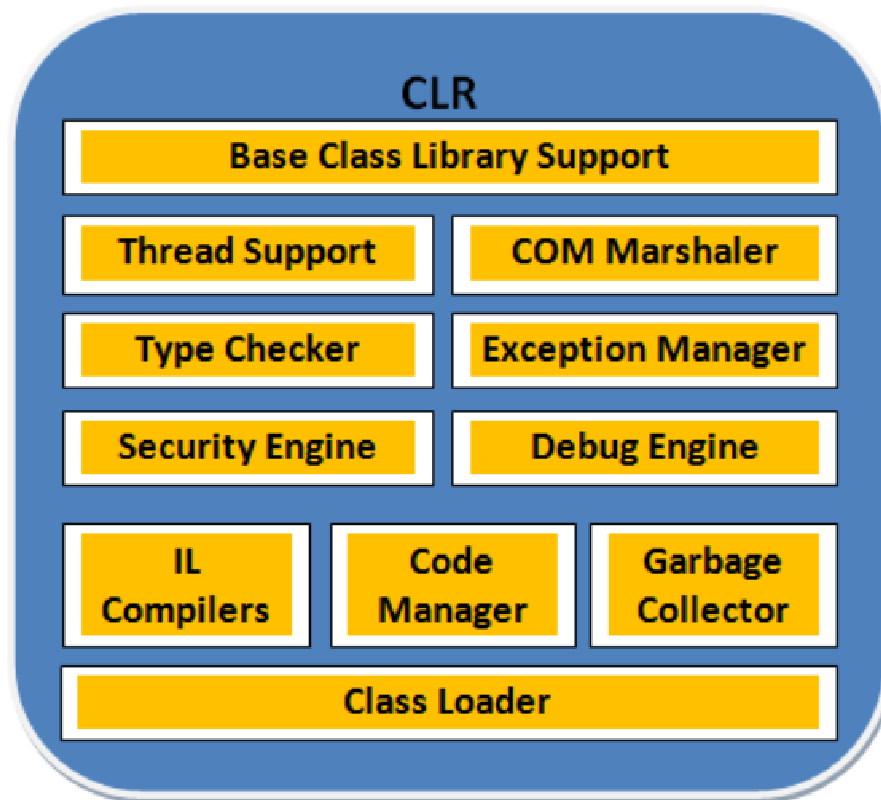

Fig. 3.2 CLR components

In the past, it was often necessary to compile your code into several application and each of which targeted to specific operating system and CPU architecture.

In .NET framework, when you compile your program it didn't immediately create operating system specific native code. Instead, it first compiles your code into Microsoft Intermediate Language (MSIL). Usually all .NET languages such as C#, VB and F# etc are initially compiled into MSIL by CLR compiler.

The .Net language such as C#, VB and F#, which conforms to the Common Language Runtime (CLR), uses its corresponding runtime which is responsible to run the application on different operating system. Only needed managed code (MSIL) code is executed just before the function is called. To do so CLR takes helps of Just In Time (JIT) compiler.

With the help of Just In Time compiler (JIT) the Common Language Runtime (CLR) does these tasks. JIT converts the MSIL code to native code which is CPU-specific code that runs on the same computer architecture as the JIT compiler and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process.

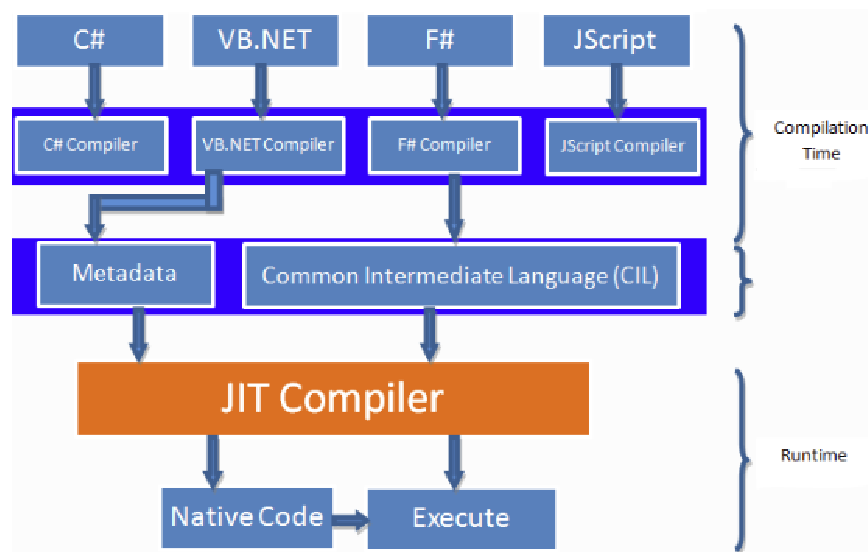

Fig. 3.3 JIT compiler

We will go over performance differences between C# and Python for different operations.

- **Arithmetic Operations**
  C# Typically shows significant speed advantages in executing basic arithmetic operations. Benchmarks often indicate that C# can be 5 to 10 times faster than Python for simple tasks.Due to Pythons ts interpreted nature, Python incurs overhead, making arithmetic operations slower.

- **Loop Performance**
  JIT compilation optimizations allow C# to handle loops very efficiently, often achieving execution speeds that are several times faster than Python for tight loops. Loops in Python tend to be slower due to dynamic typing and interpreted execution.

- **Function Calls**
  Function calls in C# are optimized, especially with inlining and other compiler optimizations. While on the other hand, Python can be significantly slower due to the overhead of argument passing and dynamic type checking.

3.2 Benchmarking system overview

In order to guarantee repeatability of our experiments we have taken every precaution to detail all the specifications of the machine that was used for testing purposes. The same configuration (hardware and OS) was used for all the runs of algorithms for which we were tracking performance,

We will list all the relevant hardware and software components:

- **CPU - AMD Ryzen 5 2600**
  Cores: 6
  Threads: 12
  Socket: AMD Socket AM4
  Cache L1: 96 KB (per core)
  Cache L2: 512 KB (per core)
  Cache L3: 16 MB (shared)
  Frequency: 3.4 GHz
  Turbo Clock: up to 3.9 GHz
- **RAM - 32GB**
  Type: DDR4
  Size: 32768 MBytes
  Number of channels: 2
  DRAM Frequency: 2666 Mhz
  CL Latency: 16-18-18-39-60
- **Motherboard - Gigabyte B450M DS3H**
  Manufacturer: Gigabyte Technology Co., Ltd.
  Model: B450M DS3H-CF (AM4)
  BIOS revision: F67d (Date 9/2/2024)
- **GPU - Radeon RX 570 8GB**
  GPU Name: PowerColor Red Dragon RX 570 OC 8 GB (AXRX-570-8GBD5-3DHD/OC)

Base Clock: 1168 MHz
Boost Clock: 1250 MHz
Memory Clock: 1750 MHz
Bus Interface PCIe 3.0 x16
Memory Size: 8 GB (GDDR5)
Memory Bus: 256 bit
Bandwidth: 224.0 GB/s
TDP: 150 W

- **Storage:**
  - **KINGSTON SA400S37480G**
    Manufacturer: Kingston
    Type: SSD
    Capacity: 447 GB
    SATA type: SATA-III 6.0Gb/s
  - **TOSHIBA HDWD130**
    Manufacturer: Toshiba
    Type: HDD
    Capacity: 2794 GB
    SATA type: SATA-III 6.0Gb/s
- **OS: Windows 10 Pro**
  Version: 22H2
- **Framework version:**
  - **Python 3.12.6**
  - **.NET 8.0 Framework**
- **IDE:**
  - **Visual Studio Code** (Python)
    Version: 1.93.1
  - **Microsoft Visual Studio Community 2022** (C#)
    Version: 4.8.09037

We took special note to document all the relevant hardware and software that was used to guarantee that results are reproducible. Please note, although the performance of any specific

algorithm could be different on other machine, the relative performance of algorithms one to another should always stay the same. Of course, if obvious minimum requirements are met, i.e the system has enough RAM or base CPU power necessary for execution. Finally, it is also worth mentioning that we did everything in our power to eliminate any effect of background processes on the execution algorithm, and we believe they should not present any impact in execution time.

3.3 Efficient algorithm for weighted k-core decomposition

As we discussed previously, the main problem with Algorithm 1 for unweighted graphs from [3] are leap vertices in graph that has weights. Here we will present our solution to the problem and adapt the base algorithm, so that all the cases are handled. We will also proceed and prove that this algorithm is indeed good and that p-core hierarchy will indeed be determined by our new algorithm.

In the algorithm the core number of vertex v, core(v), is represented by the table element core[v], and its degree by the table element degree[v].

```
INPUT: graph G = (V, L) represented by lists of neighbors Neighbors(v) for each vertex
OUTPUT: table core with core number core[v] for each vertex v
01    compute the degrees of vertices
02    order the set of vertices V in increasing order of their degrees
03    for each v ∈ V in the order do begin
04       core[v] := degree[v]
05       for each u ∈ Neighbors(v) do
06          if degree[u] > degree[v] then begin
07             degree[u] := degree[u] − 1
08             reorder V accordingly
09       end for
10    end for
```

Now we will present modification of the base algorithm in such a way that weighted Graphs will also be supported. In addition we will provide relevant explanation. We will apply the following modifications to the base algorithm:

remove line 06

replace line 07 with **degree[u] := max (degree[v], degree[u] − w(u,v))**

After the aforementioned changes the algorithm should look as in the following pseudo-code:

INPUT: graph G = (V, L) represented by lists of neighbors Neighbors(v) for each vertex
OUTPUT: table core with core number core[v] for each vertex v

```
01    compute the degrees of vertices
02    order the set of vertices V in increasing order of their degrees
03    for each v ∈ V in the order do begin
04        core[v] := degree[v]
05        for each u ∈ Neighbors(v) do
06            degree[u] := max ( degree[v], degree[u] − w(u,v) )
07            reorder V accordingly
08        end for
09    end for
```

We take graph as an input, represented by lists of neighbors for each vertex. And want core number for each vertex as output. The same as before, for unweighted scenario. The only difference is that now we have to take into mind the weights also. So the initial lists of neighbours will have to either be double list, or alternative structures, because they have to store weight of the edge also.

After the algorithm is started, the behaviour should be as follows. Degree computation should stay the same, keeping in mind that we will add the weights of edges instead of incrementing the value of total weight always by 1 as before. Afterwards, we order the set of all vertices by their total degree - total weight of all the edges in increasing order.

Now we can start the main loop, where we go over all the vertices from the list. Keep in mind that values, or more specifically the order of vertices can change after each iteration, since we will sort it before the start of next iteration. This is done so that we always take the vertex with lowest total weight from all the vertices that are not yet visited. At the start of the loop we assign core value to the current total weight of the vertex.

Inside we begin another for loop, this time we will be looping through all the neighbours from the original vertex V. The most important par is setting the value of total weight of neighbour u to the maximum value from total weight of V or recalculated degree without the vertex V in graph. This step enables us to take into account weighted graphs also. And is differentiating part from the previously mentioned algorithm.

Please note that a slight optimization that is not mentioned here is sorting the collection from the first for loop in increasing order. If the sorting is done after the inner for loop finished, instead of inside double for loop we believe it will have significant positive impact on performance, so in the actual implementation we have done so. Since here we are only

focusing on general case for weighted graphs, we have omitted this change in the algorithm to simplify things.

We finish when the first loop has gone through all the vertices and assigned core numbers. Evidently we will also go over each of the remaining neighbours in the inner loop. So it is worth noting, that in the real world implementation, the algorithm gets faster and faster after each visited vertex from the initial set, since the edges will be removed inside the inner for loop. This means that as the time goes by, less and less edges remain in the graph, and it should take less time to go over them.

Finally, we would like to point out that this algorithm shares one important feature as the algorithms for weighted graphs mentioned in the previous chapter. It will assignee maximal values to the neighbours in the inner loop, this alone guarantees that there will be no leaping done by vertices. This alone makes us confident that the algorithm is sound and core values will be the same. Essentially, they represent similar ways of doing the same work, with the main idea behind them being the same. We believe that this newly shown algorithm should also have the benefit of better performance. We will go over this in more detail in next chapter.

It is worth noting that in scenario where we have a weighted and directed graph, if p-function is represented by the following formula:

$$p(V) = \sum_{n=1}^{N} indegree + outdegree$$

where V is vertex we are interested in, and some goes through the neighbours for that vertex 1,...,N. Clearly, for the given p-function Theorem 2 will hold, since it is obvious that it is monotone. So our algorithm is clearly determining the p-core hierarchy, which was our objective.

3.4 C# Algorithm implementation

No we will go over the concrete implementation of the algorithm we presented in the previous section in C# programming language with short commentary. We will skip the class structure and similar commentary for simplicity and will focus on methods.

We will briefly go over data consumption here, because it has proven to be much faster than in Python. We create a Dictionary of Dictionaries to store adjacency lists. This is done mainly because of performance, but more on that in later chapter. The rest is pretty straightforward. Using StreamReader we go line by line through data, skipping the line with headings, and add the edges to graph. We take special note, because of the input, and the

fact that we are interested in undirected graphs to add the same edge for both vertices. For example if we had in input that vertices 4 and 7 have and edge with weight 14, we will add vertex 7 to vertex 4 neighbours list with the weight, but also we will do the opposite, edge 7 should also have edge 4 with the appropriate weight. This is done to reduce code complexity later on, and speed up access to the neighbour we want.

```
1 reference
public IDictionary<int, IDictionary<ushort, uint>> ConsumeData(string pathToData)
{
    IDictionary<int, IDictionary<ushort, uint>> adjecencyDictionaries =
        new Dictionary<int, IDictionary<ushort, uint>>();
    using (StreamReader sr = new StreamReader(pathToData))
    {
        // Skip the heading
        string? line = sr.ReadLine();
        line = sr.ReadLine();
        while (line != null)
        {
            string[] split = line.Split(',');

            ushort fromVertex = ushort.Parse(split[0]);
            ushort toVertex = ushort.Parse(split[1]);
            uint weight = uint.Parse(split[3]);

            AddEdgeToGraph(adjecencyDictionaries, fromVertex, toVertex, weight);
            // Since the graph is unordered we have to add the edge for both verticies
            AddEdgeToGraph(adjecencyDictionaries, toVertex, fromVertex, weight);

            line = sr.ReadLine();
        }
    }

    return adjecencyDictionaries;
}

2 references
private void AddEdgeToGraph(IDictionary<int, IDictionary<ushort, uint>> adjecencyDictionaries,
    ushort vertex1, ushort vertex2, uint weight)
{
    if (adjecencyDictionaries.TryGetValue(vertex1, out IDictionary<ushort, uint> vertexDict))
        vertexDict.Add(vertex2, weight);
    else
        adjecencyDictionaries.Add(vertex1, new Dictionary<ushort, uint> { { vertex2, weight } });
}
```

Fig. 3.4 C# data consumption

Before the main body of algorithm, we will present also a helper method that will be used for calculating the total weight of all the edges. This will be used quite extensively, since the calculation of total weight is used inside the double for loop, when determining the weight of the neighbour.

In addition, it is worth noting the structure of class Edge that will be used thought the main body of the algorithm.

```
2 references
private uint GetTotalWeightOfVertex(IDictionary<ushort, uint> edges)
{
    uint totalWeight = 0;
    foreach (uint edgeWeight in edges.Values)
        totalWeight += edgeWeight;

    return totalWeight;
}
```

Fig. 3.5 Total weight of vertex

```
4 references
public class Edge
{
    public ushort Id;
    public uint Weight;

    2 references
    public Edge(ushort id, uint weight)
    {
        Id = id;
        Weight = weight;
    }
}
```

Fig. 3.6 Edge class structure

The main body of algorithm, or its core, is pretty straightforward. Firstly we create a data structure - Dictionary that will hold the values of k-cores for all the vertices. We then compute the weighted degrees of all the vertices, and create a List to store the newly calculated data. Afterwards we sort the newly created list by total weight value for every node. This list will serve as a starting point, and we will go through all the vertices in it.

Once the data preparation is done, we start going over all the vertices in increasing order of total weighted degrees in a for loop. Just to be on the safe side, we set the value of the current vertex to (0,0), since we cannot remove an element from the list whom we are iterating, at least not in a way that would be readable. After we add the value of k-core to the dictionary that is storing the output, we determine the neighbours of the vertex we need to iterate over.

Once the neighbourhood is determined, we can go over each of them, in a foreach loop and do the following. For each of the Neighbours, lets note one of them with U for easier notation. Firstly, we remove the vertex from original loop (V) from the neighbours of vertex U. Then we recalculate the new total weighted degree of U, after V has been removed from its neighbourhood. Secondly, we get the vertex object by reference from the list of vertices we are iterating over. Finally, we set the weight of vertex obtained in such a way to maximum of total weighted degree of V or the previously recalculated total weighted degree of U.

After we finish with all the neighbours, we sort the list again. And continue going through all the vertices. By choosing the maximum as is defined in the previous step for each of the neighbours, we guarantee that all vertices will be visited. Or differently put, there will be no leap vertices, and the total weighted degrees will appropriate. Sorting the list is needed at this step to lower the vertices that have had V in their neighbours, because those edges are no longer existing. Since we removed vertex V.

In the end, after all the iterations, we get k-core values for all the vertices in a Dictionary structure.

```
1 reference
public IDictionary<ushort, uint> ComputeDecomposition(IDictionary<int,
    IDictionary<ushort, uint>> graph)
{
    IDictionary<ushort, uint> kCores = new Dictionary<ushort, uint>();

    // Compute the degrees of vertices
    var vertexDegrees = new List<Edge>();
    foreach (ushort fromVertex in graph.Keys)
    {
        uint totalWeight = GetTotalWeightOfVertex(graph[fromVertex]);
        vertexDegrees.Add(new Edge(fromVertex, totalWeight));
    }

    // Sort the vertex degrees
    vertexDegrees.Sort((pair1, pair2) => pair1.Weight.CompareTo(pair2.Weight));

    var numOfVertices = vertexDegrees.Count;
    for (int i = 0; i < numOfVertices; i++)
    {
        var fromVertex = vertexDegrees[i];
        vertexDegrees[i] = new Edge(0, 0);
        kCores.Add(fromVertex.Id, fromVertex.Weight);

        var neighboursToGoThrough = graph[fromVertex.Id];
        // Remove the edges from Dictionary of Neigbours to fromVertex
        foreach (ushort neighbourId in neighboursToGoThrough.Keys)
        {
            var neighbour = graph[neighbourId];
            uint weightForRemoval = neighbour[fromVertex.Id];
            neighbour.Remove(fromVertex.Id);

            uint newVertexTotalWeight = GetTotalWeightOfVertex(graph[neighbourId]);

            var vertexForEdit = vertexDegrees.First(x => x.Id == neighbourId);
            vertexForEdit.Weight = Math.Max(newVertexTotalWeight, fromVertex.Weight);
        }

        vertexDegrees.Sort((pair1, pair2) => pair1.Weight.CompareTo(pair2.Weight));
    }

    return kCores;
}
```

Fig. 3.7 Main body of the algorithm

Chapter 4

Results

In this chapter we will mainly cover comparison of performance between Python and our new C# implementation of the previously discussed adapted algorithm for weighted graphs. In addition, we will discuss some of the choices we have taken during the implementation that have led to performance difference between the two algorithms. Finally, we will touch on potential for improvements in some key areas including Data structures, Sorting algorithms and multithreading.

4.1 Performance comparisons

We measured performance for both the implementations of k-core decomposition algorithm that were described in previous chapters. The performance was measured for Python and for C# in fashion that would guarantee their comparability. We measured performance only for the main body of the algorithm, when all the data is fully loaded.

Before we start, it is worth noting that C# implementation also showed to be superior when it came to Memory consumption and than **C# was continuously consuming less memory that Python implementation**. Also the process **loading the data into relevant structures was also significantly faster for C#**, initial loading of data into Python and creating Graph took several minutes while for C# it was a matter of seconds.

Because of the very bad performances of algorithm in Python we measured it in following way. After we set everything up, we let it run for 12 hours straight and determined how many vertices were processed in that time. That gives us a great insight into how much time would it take for the whole algorithm to finish, because we know that there are 10 000 vertices to go over. Without the need to wait for the whole process, we were able to run it several times and average the resuls from runs. Also we can determine the average time it takes the algorithm

to process a single vertex. These parameters will be very useful, since they will enable us to compare the results for both implementations.

After running for 12 hours Python implementation was only able to determine the k-core for 294 vertices on average out of 10 000 vertices that the graph contains. We can easily convert that to estimated run time of algorithm. So if there are 10 000 vertices in total, and for 12h only 294 were process, that means that in order to process every vertex algorithm would take approximately 408.16 hours, or more than 17 days. Or if we use the same initial values to try and get the time necessary for computation of k-core for a single vertex, the value amounts to .

To put the values into perspective:

Table 4.1 Python implementation statistics

Single k-core computation	**2.44 minutes (147 seconds)**
Determining k-core for the whole graph	**17 days (408.16 hours)**

Let us now go over the performance while using the C# implementation. Since the algorithm had a reasonable runtime, we didn't have to result to similar approximations that we used to determine the runtime of Python implementation. C# implementation took on average 10 minutes and 15 seconds to complete for the whole graph of 10 000 vertices. That is true in Debug mode, when we switched to Release build the performance was even better. We note that programs are usually run in Release mode after the implementation is finished and that Debug is only used when the application is still under development. So, for Release version all the work was done in just **06 minutes and 11 seconds**!

When we divide the total runtime by the number of vertices, we can easily obtain the time it took C# implementation to process a single vertex. On average C# algorithm processed a node every 0.0371 seconds, or in milliseconds it amounts to just **37 milliseconds**! So when results for C# are put into table, it should look like following:

Table 4.2 C# implementation statistics

Single k-core computation	**37 milliseconds**
Determining k-core for the whole graph	**6.17 minutes**

The results are even beyond what we have hoped to achieve! We started from an algorithm that was pretty much unusable, since for only a single day of data to process it would need more than 17 days. If we wanted to run analysis in real time, it would not be possible with Python, since for a single day of data it would take 17 days to process them. The final

solution does the same work for around 6 minutes! The implementation in C# is orders of magnitude faster, and enables serious data analysis to be performed on a series of similar datasets, with each representing a day.

When we put the time from both runs in the same format, in this case minutes make most sense as the common denominator we get the following comparison. Python took 24 489.6 minutes to determine the k-cores for the whole dataset, while C# took around 6.17 minutes. **The C# algorithm implementation is about 4 000 times faster than its Python counterpart!** Or in other words, in C# all the processing is done for the whole 10 000 vertices, while in the same Python would only process 2.5 vertices.

It is also worth noting, that these huge performance gains were obtained without any sacrifices. Memory footprint of C# algorithm during the execution was significantly lower, as well as data preparation which took significantly less time.

4.2 Current implementation and potential improvements

In this section we will go over some aspects of the algorithm we were paying special attention while we were implementing it for performance reasons. We will present these ideas briefly, go over their usage and potential benefits as well as drawbacks. When measuring performance of algorithm the most important things we can measure is time and space. Or in other words CPU utilization and Memory consumption. We would also like to add here that potential for parallelization also plays a big role in overall ability to increase performance of an algorithm even further. So in the following subsections, we have taken special note on data structures, sorting and multithreading that were used in our implementation as well as potential for additional improvements.

4.2.1 Data structures

Lets first review the simple structures that were used in algorithm. Thought the algorithm we used *ushort* for identifying vertices instead of more commonly used *int* structure. We believe this structure would suffice our needs since the maximal value it can receive is 65 535, and the memory consumption would only be 2 bytes (16 bits). And since we only had 10 000 vertices the structure had enough space. While on the other hand maximal value for *int* is 2 147 483 647 while it takes twice the memory - 4 bytes (32 bits).

Although, it might seem by a minuscule saving, since we are only saving 2 bytes per vertex, we have to take account of the number of edges we have in the graph. Each of these edges has to have a starting and ending vertex. So for 6 463 696 edges, each of whom have 2

vertices, we would save 25 854 784 bytes, or around 25 MB (megabytes). And this is only for one structure, if we were to store vertices in several places, or data structures the memory saving would stack.

The same goes for using *uint* instead of *double* for weights for edges. Since the values of weights can be stored in *uint*, at no cost whatsoever we could reduce memory consumption by using it instead of more traditional double. We already explained that each *uint* consumes 4 bytes (32 bits) while *double* has twice the memory footprint of 8 bytes (64 bits). So the memory usage difference, if we only used each edge once for every two vertices it connects, would be 50 MB (megabytes).

Now that we have covered basic structures, we will go over our decision to try and use Dictionary data structure over List as much as possible, both for storing vertices and edges. We even used a *IDictionary<int, IDictionary<ushort, uint»* data structure for storing neighbours for all the vertices together with their respective total weighted degrees. Using a *Dictionary<TKey, TValue>* in C# has several key advantages over using a *List<T>* data structure:

- **Significantly faster lookup**
  Dictionary provides O(1) average time complexity for lookups based on keys, whereas searching for an item in a List typically requires O(n) time, as it may involve scanning through the entire list. This is the main reason all the algorithms try and avoid using lists as much as possible, excluding some very specific scenarios.

- **Key-Value pair storage**
  Dictionary stores data in key-value pairs, allowing us to associate a unique key with each value. This is useful when there is need to retrieve values based on specific identifiers. This case was present for as, as we used ids of vertices as keys in all instances.

- **Uniqueness of Keys**
  Each key in a Dictionary must be unique. This ensures that duplicate will not accidentally be added to Dictionary. As we mentioned before this was very useful, since we used ids of vertices as keys in all instances of Dictionaries. This is contrary to List, which can accommodate duplicates, thus enabling some mistakes to go uncaught for longer period of time.

The only instance we used *List<T>* for is for storing the total weighted degrees of all the vertices in the ascending order. We chose a plain List here, instead of *SortedList<TKey, TValue>* or *SortedDictionary<TKey, TValue>* for example because we didn't want resorting

of the structure to happen after every iteration of the inner for loop. It is our opinion that sorting only once at the end of the inner for loop would yield better performance overall.

	add to end	remove from end	insert at middle	remove from middle	Random Access	In-order Access	Search for specific element
Array	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
List<T>	best case O(1); worst case O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Collection<T>	best case O(1); worst case O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
LinkedList<T>	O(1)	O(1)	O(1)	O(1)	O(n)	O(1)	O(n)
Stack<T>	best case O(1); worst case O(n)	O(1)	N/A	N/A	N/A	N/A	N/A
Queue<T>	best case O(1); worst case O(n)	O(1)	N/A	N/A	N/A	N/A	N/A
Dictionary<K,T>	best case O(1); worst case O(n)	O(1)	best case O(1); worst case O(n)	O(1)	O(1)*	O(1)*	O(1)

Fig. 4.1 Comparison of common data structures

4.2.2 Sorting algorithms

Because list of vertices is sorted in best case scenario 10 000 times and in worst case scenario 100 000 000 times. We believe this is a very important topic, when discussing performances. Even in the case when there are not a lot of vertices, because of the loops we end up going over them quite a lot, and the sorting algorithm as well as number of calls will have a significant impact.

For start, we have to decide if we want everything to sort everything when every neighbour is affected in the double for loop or sort only at every iteration of single loop. If we wanted auto sorting property, we could use some of the always sorted data structures that have that functionality out of box. But that would incur a penalty, because w would sort the collection very often. Exactly for that reason we have opted to use a regular List for storing the vertices we go over. By choosing this data structure we can sort only at the end of the outer for loop, thus reducing performance overhead by constantly shuffling things around.

Regarding sorting in C#, the underlying implementation is as follows. Until a certain number of elements in a list(100), Selection sort will be used. This is because selection sort has excellent constant parameters, and although very slow for large sets, for smaller one it can hardly be beaten. For larger sets, sorting will revert to using Quick sort algorithm, this is because on average it has n * log(n) run time. Although there are theoretically faster

algorithms, that have the same speed as Quick sort even in worst case scenario, they usually incur a penalty. This penalty is either higher constant factors or higher memory consumption. By choosing Quick sort pivot element wisely, we can get excellent performance. So when we compare it to, Merge sort for example, there is no need to take a serious hit in memory for negligible performance difference.

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)

Fig. 4.2 Comparison of common sorting algorithms

4.2.3 Potential for concurrent execution

Although it is very important to note the differences between multithreading and multiprocessing. In this subsection we will primarily take note about multithreading potential, but the majority of the discussion can be applied to the multiprocessing as well. We will not go over in much details over the differences here, but offer an illustration instead.

Let us first explain few very important concepts, namely thread starvation and thread contention. Thread starvation occurs when a thread is unable to gain access to shared resources or CPU time due to other threads monopolizing those resources. Thread contention is a condition where one thread is waiting for a lock/object that is currently being held by another thread. Therefore, this waiting thread cannot use that object until the other thread has unlocked that particular object.

Due to the nature of the dataset and algorithm we believe that applying any serious mutithreading would not be beneficial in a single dataset scenario. In order to accommodate

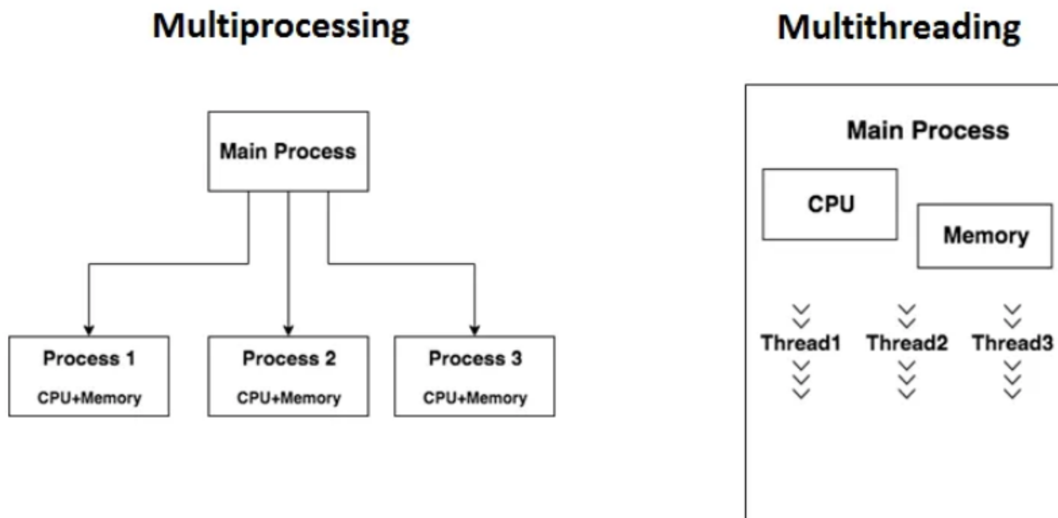

Fig. 4.3 Difference between multiprocessing and multithreading

multithreading in a single dataset scenario we would have to use multithreading safe data structures and/or locks. But due to the nature of the algorithm there wouldn't be much room to do work in parallel. The main part of work that could be divided is going through neighbors for each vertex and recalculating the total weighted degree. But since the number of vertices is in general not very large, it follows that neighbourhood would be even smaller. Thus any performance benefits that could be made by using multiple threads would be negated by the overhead thread creation, synchronization and taking care about data access would imply.

So for this reason we have opted not to introduce any multithreading in our implementation. Since we were focused on a single dataset, and there are no gains to be had by spawning additional threads. We believe that would only reduce the performance of a single threaded case. Where we do see significant possibility for improvement, is using multithreading to process multiple datasets at the same time.

The increase in performance in this case would be linear compared to the single threaded scenarios. Since the memory footprint is rather low (<500MB) for our C# implementation. We see very large potential in parallel execution for different days. That way we would avoid any thread starvation and thread contention. Also there would be no need for synchronization, since each thread would be working on its dataset, it would process a whole day on its own. We believe there is no upper limit as to when this approach would start to yield diminishing returns, provided we have enough CPU resources and Memory.

Chapter 5

Conclusion

To summarize, we have given a brief overview on Graph Theory and have covered large scale connectivity graphs in detail. In addition, certain data preparation and analytics was performed over the dataset that was in scope. This was followed by a discussion about k-core graph decomposition, where we presented and explained several algorithms both for weighted and unweighted graphs. Afterwards, we presented a Python implementation as a base scenario.

After giving an overview of C# and .Net we detailed the benchmarking system. All in preparation for our optimization of k-core decomposition algorithm. For which we provided optimized pseudo-code, relevant theory as well as a very fine-tuned implementation in C# programming language. Finally, we compared performance results and touched on current implementation specifics and potential for improvement.

We have achieved unexpected **performance speedup of around 4 000 times** between the initial Python implementation and our optimized solution in C#. This enables us to continue our work on similar and larger datasets, since with the new performance baseline we were able to achieve, we can process and analyze the data in a more meaningful way. Even near real-time processing or heavy parallelized execution are now possible.

With our optimized solution for K-core decomposition of weighted graphs in C#, we can calculate cores for graphs representing large scale networks. Insight into core numbers would give us more knowledge about complex networks that we are exploring.

References

[1] Barabási, A.-L. et al. (2016). *Network science*. Cambridge university press.

[2] Barlacchi, G., De Nadai, M., Larcher, R., Casella, A., Chitic, C., Torrisi, G., Antonelli, F., Vespignani, A., Pentland, A., and Lepri, B. (2015). A multi-source dataset of urban life in the city of milan and the province of trentino. *Scientific data*, 2(1):1–15.

[3] Batagelj, V. and Zaveršnik, M. (2011). Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145.

[4] Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117.

[5] Csáji, B. C., Browet, A., Traag, V. A., Delvenne, J.-C., Huens, E., Van Dooren, P., Smoreda, Z., and Blondel, V. D. (2013). Exploring the mobility of mobile phone users. *Physica A: statistical mechanics and its applications*, 392(6):1459–1473.

[6] Even, S. (2011). *Graph algorithms*. Cambridge University Press.

[7] Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41.

[8] Giatsidis, C., Thilikos, D. M., and Vazirgiannis, M. (2013). D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and information systems*, 35(2):311–343.

[9] Gonzalez, M. C., Hidalgo, C. A., and Barabasi, A.-L. (2008). Understanding individual human mobility patterns. *nature*, 453(7196):779–782.

[10] Isaacman, S., Becker, R., Cáceres, R., Kobourov, S., Martonosi, M., Rowland, J., and Varshavsky, A. (2011). Identifying important places in people's lives from cellular network data. In *Pervasive Computing: 9th International Conference, Pervasive 2011, San Francisco, USA, June 12-15, 2011. Proceedings 9*, pages 133–151. Springer.

[11] Malliaros, F. D., Giatsidis, C., Papadopoulos, A. N., and Vazirgiannis, M. (2020). The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92.

[12] Mulić, O. (2023). *Big Data analysis applied in space - time human dynamics research*. Phd thesis, University of Novi Sad.

[13] Rahman, M. S. et al. (2017). *Basic graph theory*, volume 9. Springer.

[14] Zhou, W., Huang, H., Hua, Q.-S., Yu, D., Jin, H., and Fu, X. (2021). Core decomposition and maintenance in weighted graph. *World Wide Web*, 24:541–561.

Appendix A

Ključna dokumentacijska informacija

Redni broj:
RBR
Identifikacioni broj:
IBR
Tip dokumentacije: Monografska dokumentacija
TD
Tip zapisa: Tekstualni štampani materijal
TZ
Vrsta rada: Master rad
VR
Autor: Mihajlo Mulić
AU
Mentor: dr Dušan Jakovetić
MN
Naslov rada: Primena metode K-core dekompozicije za grafove konektivnosti u programskom jeziku C#
NR
Jezik publikacije: engleski
JP
Jezik izvoda: engleski
JI
Zemlja publikovanja: Srbija
ZP
Uže geografsko područje: Vojvodina
UGP

Godina: 2024.
GO
Izdavač: Autorski reprint
IZ
Mesto i adresa: Departman za matematiku i informatiku, Prirodno-matematički fakultet, Univerzitet u Novom Sadu, Trg Dositeja Obradovića 4, Novi Sad
MA
Fizički opis rada: 5/57/3/24/3/0/2
(broj poglavlja/strana/lit.citata/tabela/slika/grafika/priloga)
FO
Naučna oblast: Matematika
NO
Naučna disciplina: Primenjena matematika
ND
Predmetna odrednica/Ključne reči: teorija grafova, algoritmi, k-core dekompozicija, programski jezik C#, Python, nauka o mrežama
PO
UDK:
Čuva se: Biblioteka Departmana za matematiku i informatiku,
Prirodno-matematički fakultet, Univerzitet u Novom Sadu
ČU
Važna napomena:
VN
Izvod: Tema ovog master rada je analiza efikasnih metoda za K-core dekompoziciju težinskih grafova i njihova implementacija u programskom jeziku C#. Kao polaznu osnovu koristili smo već postojeće algoritme za k-core dekompoziciju u slučaju težinskih grafova, potom smo konkretan algoritam unapredili radi povećanja performansi i implementirali smo unapređenu verziju algoritma u programskom jeziku C#. Takođe, radili smo testiranje performansi koristiće kompleksne CDR podatke od kojih smo pravili grafove konektivnosti. Rezultati testova ukazuju na veliko unapređenje performansi u odnosu na početnu verziju algoritma koja je bila implementirana u programskom jeziku Python.
IZ
Datum prihvatanja teme od strane NN veća: 26.06.2023.
DP
Datum odbrane:
DO

Članovi komisije:

KO

Predsednik: dr Nikola Obrenović, naučni saradnik,
Institut Biosens, Univerzitet u Novom Sadu
Mentor: dr Dušan Jakovetić, vanredni profesor,
Prirodno-matematički fakultet, Univerzitet u Novom Sadu
Član: dr Oskar Marko, naučni saradnik,
Institut Biosens, Univerzitet u Novom Sadu

Appendix B

Key words documentation

Accession number:
ANO
Identification number:
INO
Document type: Monograph type
DT
Type of record: Printed text
TR
Contents Code: Master's thesis
CC
Author: Mihajlo Mulić
AU
Mentor: Dušan Jakovetić, Ph.D.
MN
Title: Analysis of the consumer basket, consumer price index and inflation – the case of Serbia
TI
Language of text: Serbian
LT
Language of abstract: Serbian and English
LA
Country of publication: Serbia
CP
Locality of publication: Vojvodina
LP

Publication year: 2024.
PY
Publisher: Author’s reprint
PU
Publ. place: Novi Sad, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4
PP
Physical description: 5/57/3/24/3/0/2
(chapters/pages/literature/tables/pictures/graphics/appendices)
PD
56
Scientific field: Mathematics
SF
Scientific discipline: Applied Mathematics
SD
Subject/Key words: graph theory, k-core decomposition, large scale graphs, Network Science, C#, Python
SKW
UC:
Holding data: The Library of the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad
HD
Note:
N
Abstract: The topic of this master’s thesis is applied k-core decomposition for large scale connectivity graphs. The aim of the thesis is to explore the efficient algorithms for graph core decomposition in weighted graphs and to implement the algorithm in programming language C#. We also performed benchmark testing to evaluate the efficiency of our implementation which proved to be realistic comparing to implementation in Python. For testing we used large scale connectivity graphs made from telecom CDRs, which is very complex data considering the size and density of the graphs.
Accepted by the Scientific Board on: 26.06.2023.
ASB
Defended:
DE
Thesis defend board:

DB

President: Nikola Obrenović, Ph.D., research associate,
Institute BioSense, University of Novi Sad
Mentor: Dušan Jakovetić, Ph.D., associate professor,
Faculty of Sciences, University of Novi Sad
Member: Oskar Marko, Ph.D., research associate,
Institute BioSense, University of Novi Sad