Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Models for cascade failures in complex networks and systems

MASTER DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)

AUTHOR: Shima Aflatounian

ADVISOR: Francesc Comellas Padró

DATE: November, 18th 2020

Overview

Given the complexity, size, cost, and importance of network infrastructures like the internet, telecommunication, transportation, and social networks, it is not surprising the attention that research on network analysis and protection is receiving.

Today's world of technological advances creates high expectations in all those areas that humans and computers can manage. Thus, in the context of network science, we expect networks to keep their performance and be adaptative and resilient to diverse circumstances.

As all networks are subject to attacks and collapse, safety protocols should be designed to provide tools to mitigate possible network failures. Many networks suffer attacks periodically: Twitter, Amazon, eBay have experienced denial-of-service for hours and even days, power grids have undergone massive blackouts, and similar incidents have happened for many other real networks. The failure usually propagates from one component to another in a cascade manner, known as cascade failure. Cascading failure studies investigate potential solutions to make networks more robust. There are some vital nodes within each network that have a profound influence on cascading phenomena.

In this TFM, giving focus to cascade failure analysis, we explore several network structural properties that might impact the network performance when it is under attack, in particular *k-core* measure. Moreover, a recently proposed deterministic hierarchical graph family, which appears to be a good model for real networks, has been examined under attacks along with some other graph models. By comparing their relative performance, we can assess if this network can be a model to improve future network designs.

Our findings show that networks with a high minimum *k-core* and negative degree-degree correlation (disassortative) perform better under attacks than assortative networks with low minimum *k-core*. To enhance the robustness of existing networks, we can modify them in a way to have less assortativity and such that their minimum *k-core* value is increased by about 1-core to 2-core. Regarding the deterministic hierarchical graph family, by considering a simple control strategy along with an 18% extra load capacity for each component, the network would be a proper choice for future network design since it can handle all attacks, according to the standard established model considered in this TFM.

INDEX

0: INTRODUCTION

With a history of more than two centuries, network science and graph theory is an essential tool to analyze networks, design, and modify them. Each network owns a structure and follows a specific generation model. Usually, the nodes within a network do not play the same role, some having more power than others with respect to specific functions.

Cascading is a network phenomenon in which a minor local act propagates through the network components and result in a major incident. Cascading phenomenon is everywhere around us: a small local movement that results in national strikes, a local virus that turns out to be a widespread virus and panic, the spread of rumors, content, crime and, etc. Determining those individual nodes, whose triggering can result in huge cascades is of particular concern.

One of the cascading formats is cascade failure, referring to the series of failures, which initiates with the failure of a single component. Some cascade failures have the potential to engage the whole network in collapse. Cascade failure is an important research topic in network safety.

While focusing on cascade failure in this TFM, we investigate the *k-core* measure and a specific graph family of deterministic hierarchical.
Studying the graph family of deterministic hierarchical is of special importance since this recently proposed graph, despite many other graph families, captures the main features of real networks simultaneously, including modular and hierarchical properties. Moreover, investigating the concept of *k-core* or core decomposition is interesting since it provides an effective method to find influential nodes within a network.

The chapters of this report are organized as follows:

- The fundamentals of networks and graph theory are described in the first chapter, including all analysis parameters required for this study.

- The second chapter is about those previous studies on determinant factors in network robustness that inspired this work's test scenarios. This chapter also explains the cascading failure model of Wang & Rong [3], which is considered in this TFM.

- Finally, in the third chapter, to address questions raised from previous research works, we propose three different scenarios:

    - In the 1st scenario, we explore if attacking nodes with the highest *k-core* is worse than attacking nodes with the lowest *k-core*. Furthermore, considering some attack strategies other than *k-core*,

would *k-core* be the worst attack scenario, or all might have the same impact on the network?

- In the 2nd scenario, we explore whether networks with many similar properties but different values for their minimum and maximum *k-core* respond the same way to attacks.
- In the 3rd study scenario, we compare the performance of the deterministic hierarchical network with other networks that have been explored in scenario 1, under the same attacks.

To generate graphs and calculate the properties of networks such as *k-core*, we used the Networkx package in python.

The result presented in this thesis shows the relationship between the level of robustness of a network and its minimum *k-core* and degree-degree correlation values. For a given network, increasing the value of minimum *k-core* and decreasing the degree-degree correlation seems to help the network mitigate attacks easier. Moreover, the deterministic hierarchical network's considerable robustness introduces this network as a great option for future networks' design and implementation.

CHAPTER 1: GRAPH THEORY

A network is composed of a set of entities and their interactions. There are many networks around us; natural ones and human-made ones: Food webs, neural networks, supply chains, and airline networks, to name a few. To be able to study networks, researchers represent them mathematically using graphs. Graphs represent network entities by vertices (nodes) and the network interactions between vertices by edges. These edges can express different formats of connections, tangible and intangible ones. For instance, there are cables as tangible physical connections for infrastructure, such as the power grid. Examples of intangible connections are social relationships and packet transmission in telecommunications.

1.1. Graph parameters

The graph *G* with set *V* of vertices and set *E* of edges, is represented by $G = (V, E)$. The number of vertices and edges are known, respectively, as the graph's order and size.

1.1.1. Geodesic path

A geodesic path is the shortest sequence of edges connecting a pair of vertices. The length of this path is the number of edges it includes, also referred to as distance. There might be several geodesic paths between a particular node pair. There are two geodesic paths in the network beneath between node 3 and 6 with a length of two. One goes through node 1 and the other through node 3.

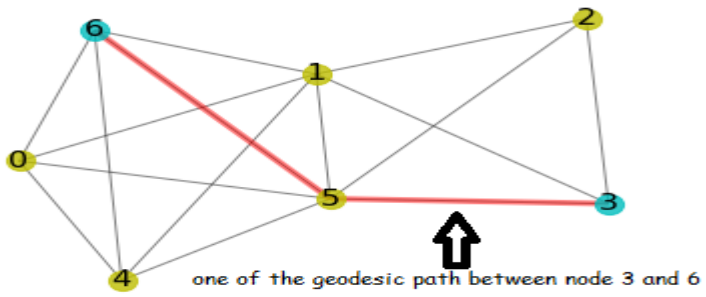


Fig 1-1- representation of a graph with seven vertices and fifteen edges

1.1.2. Mean distance

The mean distance is obtained by averaging over the distance (geodesic path length) between node pairs, formally specified as:

$$\overline{d} = \frac{\sum\sum d(u,v)}{n(n-1)} \quad \textbf{(1.1)}$$

1.1.3. Eccentricity

The eccentricity of each vertex, e_v, refers to the longest distance to other nodes in a graph. The eccentricity formally specified as:

$$e_v = max\{d(v,x)\} \quad \textbf{(1.2)}$$

1.1.4. Diameter

The graph's diameter can be determined by the largest distance or by the highest eccentricity value the network holds, considering all possible node pairs.

1.1.5. Wiener index

The Wiener index proposed by Harry Wiener computes the sum of the distance between all node pairs. This topological index is practical in chemical networks, and it is specified as.

$$W(g) = \frac{1}{2}\sum_{u}\sum_{v} d(u,v) \quad \textbf{(1.3)}$$

1.1.6. Degree

Edges link vertices of a network. Directed networks have directional edges, and undirected networks have bidirectional edges. Directed networks have two distinctive degree values for each vertex, in-degree and out-degree. In undirected networks, nodes have a degree value. The number of edges attached to a node is known as its degree. For an undirected network, the degree formula is specified as :

$$k_i = \sum_{j \in V} A_{ij} \quad \textbf{(1.4)}$$

In formula 1.4, *A* refers to the graph adjacency matrix and the value of A_{ij} would be 1, if there is an edge between node *i* and *j*.

Hubs, authority holders are identified as nodes that have the highest degrees. In figure 1-1, hubs are nodes 5 and 1, which have the highest number of connections.

1.1.6.1. Degree distribution

By counting the number of nodes having degree *k*, we can determine the graph degree distribution. The probability of each degree is specified as:

$$P(k) = \frac{n_k}{N}. \qquad \textbf{(1.5)}$$

In formula 1.5, n_k refers to the number of nodes with degree *k,* and *N* is the total number of nodes in the network

The degree distribution of a network reveals facts about its wiring rules; the type of tendency its nodes have to link to each other. Some of the well-known degree distributions are Poisson, binomial, and power-law. In a random graph with the probability *p* for edge creation, the degree distribution is binomial, specified as:

$$\Pr(\deg(v) = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k} \qquad \textbf{(1.6)}$$

Another common distribution, power-law, is described as:

$$Pr(deg(v) = k) \propto k^{-\gamma} \qquad \textbf{(1.7)}$$

Noting the exponent, γ, has a positive value. As the formula says, the probability of finding nodes with degree *k* decreases as *k* increases. An approximate distribution to this has been observed in many real networks. Networks with power-law distribution are known as scale-free. The following figure shows the distribution on a log-log scale.

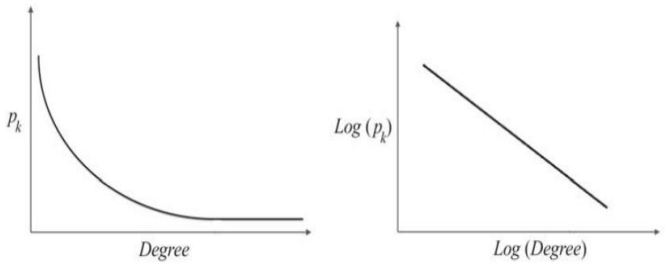


Figure 1-2- Power-law degree distribution [19]

Only a few populations have a high degree due to the preferential attachments rule, which is based on degree. Based on this attachment rule, the new incoming

nodes tend to create links to previously existing nodes that have a high degree; those with many connections.

It should be indicated that many directed real networks have different in-degree and out-degree distribution. For example, transcription networks have power-law degree distribution for in-degree and exponential degree distribution for out-degree.

1.1.6.2. Degree-degree correlation

By checking the connections of a given network, we can see whether nodes are linked to those with similar degrees or not. The tendency to connect to those with a similar degree is known as assortativity. In some networks such as social media, nodes in hubs connect with others in hubs. Therefore there is one central hub, and low degree nodes are also connected to each other. In contrast, in some other networks, such as biological, nodes with a high degree tend to connect with low degree ones, which results in hub separation and disassortativity. Some real networks, such as power grids, are considered neutral.

The quantification of degree-degree correlation is done by computing the Pearson correlation coefficient, *r,* between two distinctive degrees k_i , k_j. If the average Pearson correlation coefficient is negative, we call the network disassortative, and if it is positive, the network will be assortative. Neutral networks have $r = 0$

1.1.7. Clustering coefficient

The clustering coefficient determines the transitivity of the network. Perfect transitivity means neighbors of a node are connected directly with each other, or in other words, my friends are friends among themselves. For a given vertex v, dividing the number of triangles by the number of triads, to which the node v is contributing result in the clustering coefficient value of vertex v.
The formula for the clustering coefficient of node v is specified as:

$$C_v = \frac{\lambda_G(v)}{\tau_G(v)} \qquad \textbf{(1.8)}$$

$\lambda_G(v)$ refers to the number of subnetworks, including node v, in which there are three vertices and three edges between them. $\tau_G(v)$ refers to the number of subnetworks, including node v , which have three vertices and two edges between them. Then, the graph's clustering coefficient is obtained by averaging over the clustering coefficient values of all nodes.

1.2. Specific subsets within a graph

1.2.1. Clique

A clique is a maximal subset of a network in which each node connects to others. The concept arises from dense networks and has main applications in bioinformatics and social network analysis. Nodes within a clique have more connection with each other rather than to other nodes of the network.

1.2.2. k-clique

k-clique is a maximal subset of a network including n nodes within the distance of *k* from each other. The path between these nodes is considered with respect to the whole network, not only edges within the *k-clique* subnetwork.

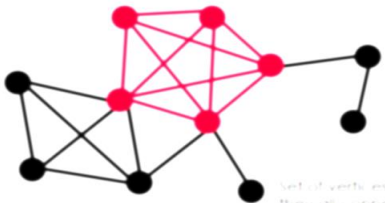


Figure 1-3- Sample of a *5-clique* [20]

1.2.3. k-core

k-core is a maximal subset of n nodes, each connected to at least k other nodes within the subset. The *k-core* value of nodes can be computed in linear time. [10]

The process for core decomposition initiates by recursively removing all nodes having a degree less than *k*, where *k* is an incremental variable starting from 1. Then in the second step, each node's unique k-core value is defined by a rule that says “the node should belong to the *k-core* subnetwork but not to the *(k+1)-core* subnetwork”.

In the following figure, the process of core decomposition is demonstrated for a sample network. The *k-core* of nodes 1, 2, and 4 is 2-core since they all belong to the 2-core subnetwork but not to the 3-core subnetwork.

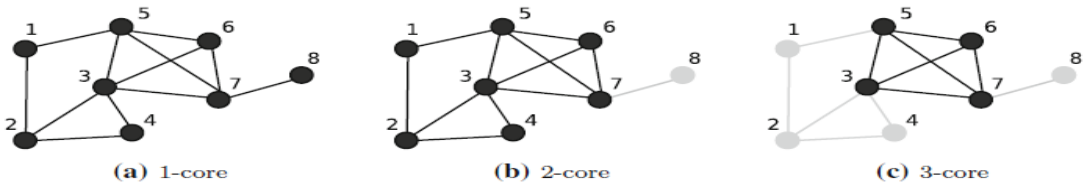


Fig 1-4- Core decomposition of a sample network [17]

In the figure below, the nodes of a sample graph are labeled by their *k-core* value.

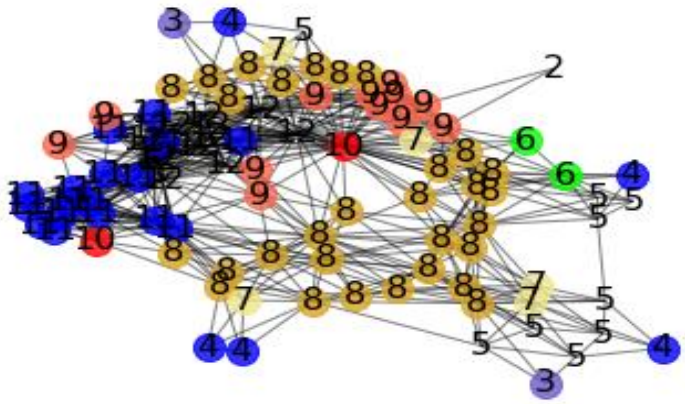


Fig 1-5- A sample graph nodes with *k-core* labels

It should be noted that *k-core* computation is for unweighted and undirected networks. The extensions of the *k-core* structure are used for other networks, such as D-core, S-core.

1.3. Networks properties

This section lists the most common properties of graphs:

1.3.1. Connected

A network is considered as connected if there is a path between any node pairs or, in other words, all nodes are reachable from any node.

1.3.2. Small-world

Networks with the small-world property have a small diameter and high local clustering coefficients and can spread the data from a node to any other node in the network in a few steps. Many real networks exhibit this property, especially social networks. As the number of nodes increases in networks with this property, the distance between any chosen node pair would be proportional to $log(n)$, where n refers to the number of nodes.

Stanley Milgram inspired the research on the small-world phenomenon by proving that social networks could connect strangers with only a small number of hubs. In 1998, Duncan Watts and Steven Strogatz developed the first random graph model with the small-world property. [1]

1.3.3. Scale-free

Networks with power-law degree distribution are known as scale-free networks. This property was introduced by studying citation networks. In a citation network, the number of citations a paper receives follows a rule that says, “the more citations a paper has, the more new citation it will receive.” Price called it a cumulative advantage since it seems some nodes within the network have an advantage over others for attracting new incoming nodes, meaning that only the number of connections of some nodes actually increases. Within the decades after Price, analysts generalized the cumulative advantage to preferential attachment and fitness models and called the described degree distribution power-law. Further information regarding power-law can be read in section 1.1.6.1.

1.3.4. Hierarchical

Many complex networks are fundamentally hierarchical as nodes are classified to multiple levels like an organization. The hierarchy level varies from a network to another, and there are several types of hierarchies in network science. The common framework that can conduct hierarchical computation on all types of graphs is the global reaching centrality or GRC. GRC is a function of LRC, local reaching centrality, a variant of classic closeness centrality. The LRC and GRC are specified as follows: [13]

$$C_R(i) = \frac{1}{N-1} \sum_{j:0<d(i,j)<\infty} \frac{1}{d_{ij}} \qquad \textbf{(1.9)}$$

$$GRC = \frac{\sum_{i \in V} [C_R^{MAX} - C_R(i)]}{N-1} \qquad \textbf{(1.10)}$$

$C_R(i)$ calculate the LRC value of node i . In formula 1.9, d_{ij} refers to the length of the shortest path between the node i and j and N is the number of nodes in the network.

There have been too many investigations to generate graph models capable of capturing scale-free and small-world properties of real networks. However, the models do not have the hierarchical levels that exist in real networks.

Enys Mones and his colleague in [4] compared the GRC of some real networks with the GRC of their computer models. A part of the comparison can be seen in the following table.

Table 1-1- Comparing GRC of real networks with GRC of computer models [4]

Type	Meaning of A→B	Network	N	⟨k⟩	GRC	GRC^{rand}
Food web	A eats B	Ythan [48]	135	4.452	0.814	0.507
		Seagrass [49]	49	4.612	0.723	0.253
		LittleRock [50]	183	13.628	0.811	0.045
		GrassLand [48]	88	1.557	0.961	0.695
Electric	B depends on the value at A	s1488 [51]	667	2.085	0.482	0.298
		s1494 [51]	661	2.116	0.482	0.289
		s5378 [51]	2993	1.467	0.231	0.062
		s9234 [51]	5844	1.4	0.424	0.050
		s35932 [51]	17828	1.683	0.459	0.015
Metabolic	B is an end product of A	*C. elegans* [52]	1173	2.442	0.048	0.052
		E. coli [52]	2275	2.533	0.043	0.058
		S. cerevisiae [52]	1511	2.537	0.037	0.042
Neuronal	A synapse goes from A to B	*C. elegans* [53,54]	297	7.943	0.133	0.023
		Macaque brain [55]	45	10.289	0.000	0.000
Internet	A communicates with B	p2p-1 [56,57]	10876	3.677	0.598	0.597
		p2p-2 [56,57]	8846	3.599	0.600	0.599

Activate Windo

As the table shows, the computer-generated models cannot capture the hierarchal properties of real networks. Therefore in recent works, it has been important to generate computer models that, in addition to capturing small-world and scale-free properties of real networks, can capture their hierarchical level.

1.3.5. Community structure

The subnetworks within a graph that are densely connected are called communities. All vertices in each community are reachable from one another. The internal degree and external degree should be calculated for each node to determine to which communities they belong. Internal degree refers to the number of connections a node has within the same community, and external degree refers to the number of connections it has outside of the community.

If the internal degree is zero, it means that the node does not belong to that community. In case all nodes within a community have an internal degree that is higher than the external one, then the community is classified as a strong community.

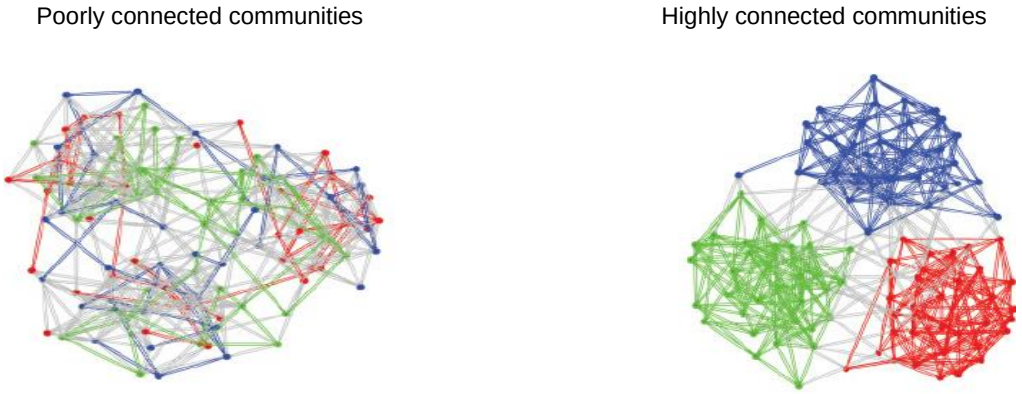


Fig 1-6- Sample of communities within a graph [18]

1.3.5.1. Modularity

Modularity quantifies a community structure. For a given community, modularity is computed by comparing the link density of a community with the same order random graph's link density. For a given graph, modularity values determine the extent to which connections in a community differs from randomly emerged connections. The modularity measure also determines which communities within a graph are better than others with respect to a given property.

1.3.6. Robustness

Being robust enough for a network means that it would be hard to break the network into isolated components. After encountering problems, a robust network can still maintain an adequate performance for delivering its functions.
Networks have a critical threshold for failure. Once the failure level exceeds that certain threshold, it will experience fragmentation, and then integration and communication within the network fail.

Barabasi and Posfai studied random and scale-free networks under random and deliberate attacks and showed a profound relationship between the degree distribution and robustness. [2]

As they concluded, in random networks with Poisson degree distribution such as Erdos-Renyi, the consequences of attacking nodes randomly will be the same as attacking nodes intentionally since, in this network, nodes play the same role. Thus there is no difference in targeting nodes intentionally or randomly.

Real networks have inspired the concept of complex networks, being made long ago, are still functioning. The reason underlies the natural robustness they have against random attacks. These real networks are modeled mathematically by synthesized scale-free networks and have been tested under random attacks.

The study results showed that to fragment these networks by a random attack strategy, almost all nodes should be removed. This is because, in a random attack, there is a low probability of targeting one of the important nodes among thousands of ordinary nodes in these networks.

There are a considerable number of studies on scale-free networks under intentional attacks. In contrast to random attacks, these networks would be fragmented in a few steps by planning an intentional attack. Some network analysts have found relationships between the robustness of a network and its topological characteristics such as assortativity measure, scale-free exponent.[5]

It should be said that scale-free is only one out of many properties a network has. Although the Barabasi-Albert networks follow the power-law degree distribution, they differ noticeably from the power-law cluster networks and most real scale-free networks. In addition to the scale-free exponent, they are different in terms of their degree values, degree-degree correlation, minimum degree, minimum *k-core,* and many other topological indices. Each of these properties affects network performance under attack, up to a certain extent.

Still, there are many arguments among analysts, and thus the research on this topic has a long way to reach a conclusion on the impact of topological indices on network robustness. This is because each research work can only test a limited number of scale-free networks. Some such as [5] and [6] test their own generated synthesized networks, and some others investigate only specific real networks such as the Internet.

1.4. Graph families

1.4.1. Erdös-Rényi

Erdös and Rényi proposed the Erdos-Renyi network in 1959. In this network, edges are wired between every two nodes with a probability, *p*. Generation parameters for this network model are *n*, a certain number of isolated nodes, *p*, a certain probability, and r, a randomly chosen value for each pair of nodes from the range of $[0,1]$. In case p is larger than r, an edge will be wired between that pair. For $p = 0$, the network will remain disconnected. As the p increases, the network becomes gradually connected, and in $p = 1$, the network will be fully connected.

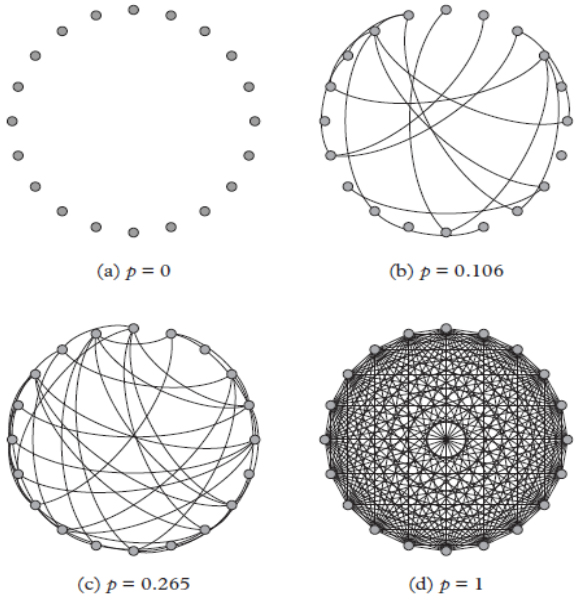


Fig 1-7- Erdos-Renyi network [1]

1.4.2. Watts-Strogatz

As explained in section 1.3.2, this graph model is the first random graph with the small-world property. To build such a network, a *k-circulant* graph should be modified by rewiring its edges at random.

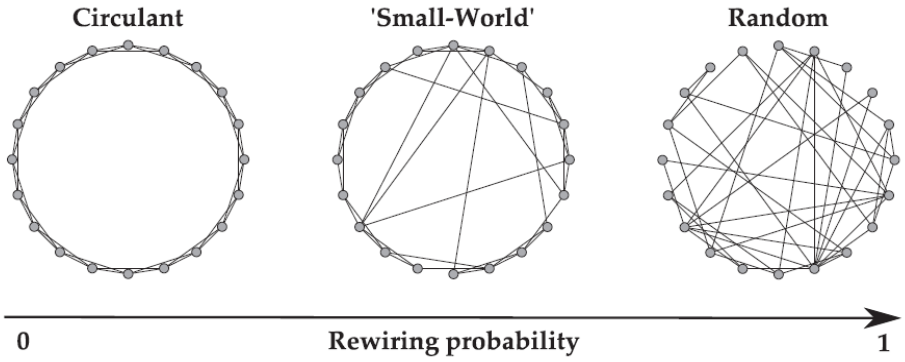


Fig 1-8- Rewiring process for generating a Watts-Strogatz network. If the probability of rewiring an edge is large, it results in a random Erdös-Rényi graph.[1]

1.4.3. Barabasi-Albert

As described in the degree distribution section 1.1.6.1, many real networks have a power-law degree distribution. To create such a network, Barabasi Albert suggested a set of generation rules in 1999. The following figure shows the evolution of the Barabasi-Albert model in 6 subsequent steps. The incoming nodes, which are represented with empty circles in figure 1.10, will connect to existing nodes based on the preferential attachment.

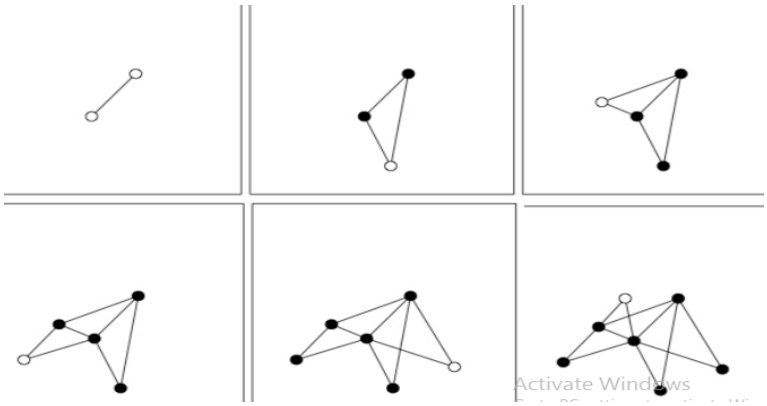

Activate Windows

Fig 1-9- Evolution of the Barabasi-Albert network in 6 subsequent steps [2]

1.4.4. Power-law cluster

The Watts-Strogatz network captures only the small-world property of some real networks and does not capture the power-law degree distribution of many real networks. Moreover, the Barabasi-Albert network with power-law degree distribution lacks a short diameter and high local clustering coefficient and therefore is not small-world. This implies the need for another computer network model with both scale-free and small-world properties. Holme and Kim [16] built the power-law cluster upon the Barabasi-Albert network methodology, with some modifications; after adding a new node Y in each time step and connecting it to high degree nodes, by a probability, there will be extra edges placement from node Y to the neighbors of recently connected nodes.

In this way, the network diameter, which depends on each node pair's geodesic path, will be reduced, and the average clustering will be increased. So it will be a kind of Barabasi Albert network with the small-word property.

1.4.5. Random d-regular

In the random d-regular graph, there are n vertices, each connected randomly to d nodes. So, all nodes have the degree value of d.

1.4.6. Deterministic hierarchical

The graph family of deterministic hierarchical, proposed in [7], is a generic family of hierarchical networks that capture the high modularity and self-similarity of many real-life networks in addition to having small-world and scale-free properties. This graph family generalizes the previous research works on hierarchical networks.

Based on [7], the process to generate the deterministic hierarchical graph $H_{n,k}$, where *n* is the number of initial nodes and *k* is the number of replicas, starts from a complete graph of order n; K_n. Then *k* replicas of K_n are created, and some edges between these replicas are wired.

Then *k* copies of the generated structure are made and connected following some rules (see figure 1.12). This way of copying the initial structure multiple times and creating links among them leads to self-similarity and high modularity properties for the deterministic hierarchical graph. The following figures clarify the way to construct a network of $H_{4,4}$.

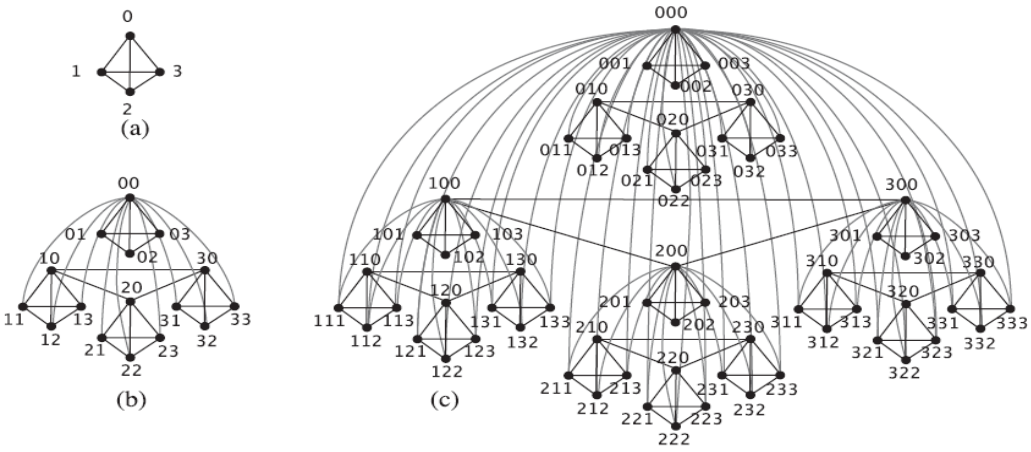


Fig 1-10- Generation of the $H_{4,4}$ deterministic hierarchical network [7]

Initial Vertex 000..0, by having the highest number of connections, has a vital role in network integration.
As described in [7], the diameter of this network is:

$$D_k = 2k - 2 \qquad \textbf{(1.11)}$$

As many real networks, this graph family has a high local clustering coefficient for any network size.

1.4.7. Connected Caveman

Connected Caveman is built upon isolated *k-cliques*; by rewiring one of the edges within each clique to link to the adjacent *k-cliques.* This network model comes from social network theory.

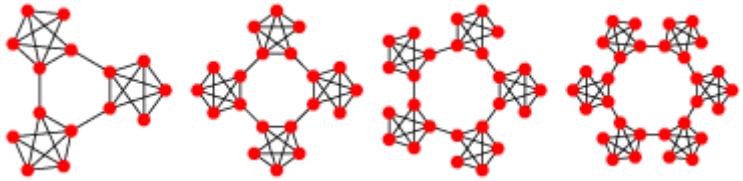


Fig 1-11 - Sample of 4-clique connected caveman graphs [21]

1.4.8. Random geometric

The simplest undirected spatial network is random geometric. The network is generated by distributing *n* nodes into spaces with respect to the uniform distribution. Then edges are placed between pairs of nodes, whose euclidean distance is within a certain range, specified by *r*, known as the connectivity parameter.

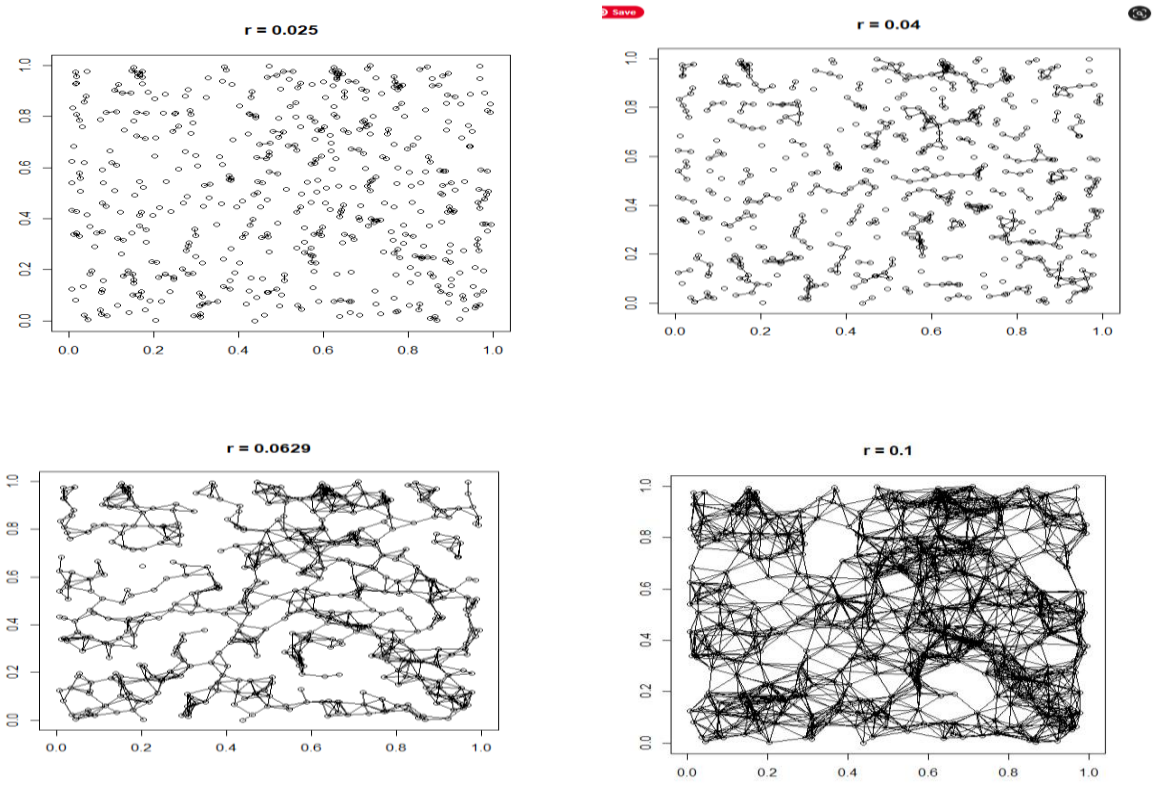


Fig 1-12- Same order random geometric networks with different *'r'* parameters [1]

[1] https://www.wikiwand.com/en/Random_geometric_graph

1.4.9. Geographical threshold 2D & 3D

The geographical threshold graph family (GTG) is a generalized model of random geometric graphs. In this model, the establishment of edges is a function of nodes' weights and distance threshold. A non-negative random weight has been assigned to each node of GTG.

The creation of such a geographically constrained network model was necessary since many real networks are not growing. In the GTG family, these constraints are modeled by assigning weights to nodes.

The condition for the establishment of an edge between a node pair *i* and *j* is specified as:

$$G(w_i, w_j)h(r) \geq \theta_n \quad \textbf{(1.12)}$$

In formula 1.12, θ_n is a threshold parameter, determined by the network size.

$$h(r) = r^{-\beta} \quad \textbf{(1.13)}$$

In formula 1.13, *h(r)* is a function of Euclidean distance between nodes. The formula of *h(r)* is specified as follows, where β is a positive value. [8]

$G(w_i, w_j)$ is the interaction strength between the node pair *i* and *j*. $G(w_i, w_j)$ is separable to multiplication or addition format as:

$$G(w_i, w_j) = g(w_i) * g(w_j) \; or \; G(w_i, w_j) = g(w_i) + g(w_j) \quad \textbf{(1.14)}$$

1.4.10. Random partition

This graph model generates graphs that categorize the nodes in the same size groups. This graph aims to minimize the edges between graph partitions or communities and focus on edge establishment within each community rather than outside the communities. For this purpose, there are two different probabilities for edge creation specified as *p* and *r*, in which *r<p*.

1.4.11. Random degree sequence

This model has been proposed to generate a random graph with a prescribed degree sequence. Despite Erdos-Renyi and random d-regular, this random network shows an approximate degree distribution to the power-law when the network order is large.

1.5. Centralities

The centrality concept was first introduced for social network analysis to identify the central nodes within a social network. Later it became a critical measure for analyzing any type of network.

1.5.1. Degree centrality

The prominent centrality of some networks, such as citation networks, is the degree. In such networks, nodes are ranked by their degree values. The more the degree value of a node, the more central role it has. The centrality of each node is calculated by dividing its degree value by the highest possible degree value. [1]

1.5.2. Closeness centrality

This centrality measures the average extent to which a node is close to other nodes. The measure is of interest for networks whose connections follow the shortest paths, such as communication networks and social media. The shorter the shortest paths of a node, the better candidate it is for transmitting information within the network. Closeness centrality formula of node *v* in the graph *G* is specified as :

$$c_C(v) = \frac{\mathrm{n} - 1}{\sum_{t \neq v} d_G(v,t)} \qquad \textbf{(1.15)}$$

In formula 1.15, $d_G(v,t)$ refers to the length of the shortest path between the node *v* and *t*, where *t* is any reachable node from node *v* and *n* is the total number of nodes in the network.

1.5.3. Current flow closeness centrality (CFCC)

This centrality inspired by electric circuits is a variant of closeness centrality and identical to information centrality. [9]
Some believe that information spreads more efficiently following a flow path similar to an electrical current. This centrality is specified as follows, where the term $p_{st}(s) - p_{st}(t)$ refers to the effective resistance between two points *s* and *t* or, in other words, corresponds to the difference in potential: the voltage between them.

$$c_{CC}(s) = \frac{n_C}{\sum_{t \neq s} p_{st}(s) - p_{st}(t)} \qquad for\ all\ s \in V \qquad \textbf{(1.16)}$$

Information centrality is a measure with a given focus on the amount of information that can be transmitted between nodes. The information centrality of vertex *i* is the average information that spread from all the shortest paths that pass through *i*. The amount of information a path contains is calculated by inverting the length of that path.

For vertex *i,* the information centrality is specified as follows, where *n* is the number of nodes in the network and I_{ij} is the information between two nodes of *i* and *j*.

$$\overline{I_\iota} = \frac{n}{\sum_{j=1}^{n} \frac{1}{I_{ij}}} \qquad \textbf{(1.17)}$$

1.5.4. Betweenness centrality

As its name states, betweenness centrality captures how well a node is positioned between two nodes. It measures the average level of contribution a node makes to the shortest paths of a network. Brokers are identified then as nodes with high betweenness centralities. The following formula calculates the betweenness centrality of node *v*:

$$c_B(v) = \frac{1}{n_B} \sum_{s,t \in V} \frac{\sigma_{st}(v)}{\sigma_{s,t}} \qquad \textbf{(1.18)}$$

The term $\sigma_{st}(v)$ refers to the number of the shortest paths that pass through node *v* as an inner vertex and $\sigma_{s,t}$ is the total number of shortest paths between each node pair *s* and *t*. n_B is a normalization factor, which is determined by the following formula: [9]

$$n_B = n(n-1) \qquad \text{for the initial and last node of the graph}$$

$$n_B = (n-1)(n-2) \text{ for the rest of the nodes} \qquad \textbf{(1.19)}$$

1.5.5. k-core

k-core of each node can be calculated in linear time, as explained in section 1.2.3. This criterion can be used to sort nodes by their *k-core* in an ascending or descending manner.

CHAPTER 2: CASCADE FAILURES

2.1. Introduction to cascading in networks

Cascading is a network phenomenon in which a minor local act propagates through the network components and result in a major incident. Cascading phenomenon is everywhere around us: a small local movement that results in national strikes, a local virus that turns out to be a widespread virus and panic, the spread of rumors, content, crime. Determining individual nodes, whose triggering can result in huge cascades, is of interest to marketing, politics, socioeconomic, epidemiology, computer science, network safety applications, and many other fields.

Depending on the network's model and its objectives, there are different tools for identifying crucial individuals within the network. For instance, marketing and advertising campaigns' objective is usually to maximize cascade, whereas the concern of network safety is about minimizing cascades.

For finding single influencers within the network, nodes are usually ranked by some criteria, such as centrality measures. Each centrality measures a distinctive aspect of influence. Degree centrality, betweenness centrality, *k-core,* and PageRank are among commonly employed centralities.

Degree centrality only captures each node's direct influence, whereas betweenness and eigenvector capture a longer range of influence as they consider the spreading capability of the neighbors of each node.

In some networks, the shortest path has a significant impact on spreading. For this kind of spreading, mostly closeness and betweenness centrality are taken into account. Due to these two measures' high computational requirements, calculating them is only feasible for medium-sized networks.

k-core measure, on the other hand, has a low computational requirement, which makes it a cost-effective ranking option for even large-scale networks and big-data analysis.

In [10], authors have proven that nodes with high *k-core* serve as efficient spreaders in SIR (susceptible, infectious, recovered) and SIS cascade models. However, the same conclusion may not be valid for other network propagation models as if *k-core* values do not influence the spreading model. That is why the diverse centralities have been introduced, and each has its own use case.

Apart from finding single influencers, finding a set of nodes within the network, whose collaboration demonstrates a remarkable spreading capability, is of great concern in some areas such as marketing and politics. It is desired for a

campaign that is based on multiple influencers to reduce the triggering costs while driving the most outcome. For this purpose, the minimal set of essential nodes should be found, which is an NP-hard problem to solve.

The Optimal Percolation theory can approximately solve this. One of the practical algorithms for Optimal Percolation is Collective Influence (CI). In this algorithm, through maximizing a function, nodes are ranked by their CI centrality. A set of nodes with the highest CI centrality, for instance, can drive the best out of a marketing budget. It should be noted that deploying CI algorithm is too computational demanding for large networks. [10]

2.2. The unexpected power of low degree nodes

From early studies on the centrality topic, nodes with the highest centrality ranks were expected to significantly impact network performance and be the essential ones for network integration. In contrast, numerous recent papers have found the extraordinary power of nodes with low centrality ranks.

For decades, studies on the integration of brain network and memory performance have been sources of inspiration for creating robust networks. In 2018 Gino Del Ferraro, Flaviano Morone, and their colleagues found evidence that the nodes of minimal set, whose inactivation collapses the giant component of the brain to the largest extent, are located in the NAc area of the brain (nucleus accumbens). The NAc nodes do not have high *k-core* values and high degree. Their case study observed that the removal of 20% of hubs reduces only the size of the brain giant component equal to the loss of hubs, meaning the giant component size becomes 80% of its original size. Thus, they concluded the inactivation of hubs does not spread failure to the rest of the network. Whereas, the inactivation of only a small fraction of nodes with high CI ranks, around 7% of total nodes, is enough to reduce the giant component size to 5% of its original value. [11]

The authors of [11] categorize centralities into two groups, hub-centralities and integrative centralities. Hub centralities such as degree, *k-core,* and eigenvector identify only hubs, but integrative centralities such as CI (collective influence) and BC (betweenness centrality) can find weak nodes; those with low degree values yet highly influential for brain integration.

Wang and Rong's article on the failure of the western USA power grid also captured the importance of nodes with low centrality values. [3]
They consider a parameter α to control the strength of initial loads in the network. When $\alpha \leq 0.5$, attacking nodes with the lowest centrality-loads engage a larger proportion of the network in cascade, to the extent that an extra load capacity of

70% is required for the network to be able to resist cascade failures. In comparison, the network needs an extra load capacity of 30% to mitigate the attack on nodes with the highest centrality-loads. Authors have used a combination of degree and betweenness centrality measures to define each node's load.

2.3. Relationship between k-core structure and network robustness

Authors in [6] constructed a new network and introduced it as KCBA, a Barabasi-Albert network with *k-core* layers from 1-core to k_m-core. They identify the maximum *k-core* value of a network as a depth of *k-core*. Based on an idea that the value of depth of *k-core*, k_m, might have an impact on the robustness of networks, authors performed tests to verify this hypothesis. In particular, they wanted to know whether a graph with a higher k_m would respond better to attacks than a graph with lower k_m values.

They examined only the KCBA network under a random attack strategy. The result reveals some interesting relationships between the *k-core* structure of a network and its robustness against cascade failures. One of these interesting results is that they observe that the less heterogeneous the *k-core* structure, the better the robustness. Their conclusions might not be valid for all networks since they conducted some tests on the KCBA network under random attack. By the way, their work has inspired other researchers to study such a relationship.

Many real and computer-generated networks do not follow a *k-core* structure in order from 1-core to k_m-core. The minimum *k-core* value is not necessarily 1, and some values might be absent between minimum *k-core* and maximum *k-core* subject to the network model considered. Besides, as mentioned in section 2.2, recently, many authors devote their research to study the influence of low *k-core*, low degree nodes, whereas authors in [6] focused the study on the highest k-core values. These are other reasons that many further studies are required to clarify the impact of a network's k-core structure on its robustness.

2.4. Relationship between disassortativity and robustness

The influence of degree-degree correlation or assortativity on system dynamics have been explored to a considerable extent. The computation of this measure has been described in section 1.1.6.2. The well-known applications of assortativity study are in epidemiology, cascading, and network safety. Depending on the assortativity type of a network, it might provide a solution for stopping or controlling the epidemic phenomenon by removing, vaccinating, or supporting a certain fraction of nodes. Disassortativity is known to protect

networks to some extent against the spread of viruses. However, more studies are required to draw such a conclusion.

Kazuhiro Takemoto and Tatsuya Akutsu studied the effect of degree-degree correlation on the size of minimum dominating sets both analytically and numerically. Previous works have not done such a direct study on MDS's size, since finding it is an NP-hard problem. MDS or minimum dominating set is the smallest set of driver nodes to control the whole network. Takemoto and Akutsu's research reveals that disassortativity reduces the size of MDS. [12] Several studies demonstrated that nodes within MDS could have critical biological rules and contain specific genes that are responsible for spread within the network.

There are arguments that network features other than assortativity might affect MDS and even MDS is not necessarily an appropriate set for network controllability under the condition that energy and cost are limited. That is why further studies on this topic are required.

2.5. Cascade failure

Networks under attack can go to a phase transition from being connected to a fragmented network. In the cascade failure process, the attack starts with a single node's failure, which is the way most intentional attacks are planned. On the other hand, in epidemic failure studies, the attacks start with a specific set of nodes. [14]

2.5.1. Reason for cascade failure

Not triggering every node of a network could result in a cascading failure phenomenon.

Every node within a network has some assigned responsibilities that are quantified as loads which they should process.

After the failure of a node, the responsibility for its loads will lie with its neighbors. Thus, the neighbors must have an extra load capacity to be able to process loads coming from the failed nodes. Depending on the load value of the failed node, the neighbor can manage the extra assigned load if and only if:

New load (=The initial load of node + excess incoming load) $\leq$ *node load capacity*

So If the condition is not satisfied, and the new total load exceeds the node load capacity, then this node will also fail, and the distribution of loads to functional nodes will continue. This process may stop soon if there are nodes capable of handling these extra loads. If not, more and more nodes will become non-functional, engaging the whole network in a global collapse.

2.5.2. Load definition in cascade failure modeling

Cascading failures in complex networks have been widely explored, and several mathematical models have been proposed to model the load redistribution, statically, or dynamically. These models assign load capacity to each node proportional to its initial load, and the initial load is a function of the node centrality value. As there are many centralities, the appropriate centrality for load definition would be decided by network application.

One of these popular models is the Motter-Lai, which was introduced in 2002 for data packet transportation. [14][3]
In the Mother-Lai, the number of shortest paths passing through each node matters, so the initial load is a function of betweenness centrality. In other networks and other cascade failure models, the load might be a function of degree centrality, etc. The term loads used regularly in this work differs notably from the loads that nodes have in real networks. In cascade failure models, the load term is used only for modeling purposes. These loads, which are based on centrality measures, capture a virtual influence of nodes within a network.

Wang and Rong introduce a new load model in [3] because, as they explained, all previous load definitions have their own drawback. In this new model, they tried to reduce the complexity of the betweenness centrality and improve the practicability of the degree centrality. In the new model, loads are calculated using the node degree and summation of degrees of neighboring nodes:

$$L_j = (k_j \sum_{m \in \Gamma_j} k_m)^{\alpha} \qquad \textbf{(2.1)}$$

In 2.1 formula, Γ_j is the set of nodes neighboring *j*, and α is a tunable parameter, which controls the strength of the initial loads. There is a belief that says, “the more responsibility an individual has, the more capacity it demonstrates for taking new responsibility.” That is why the capacity formula is proportional to the initial load. The linear formula of capacity is specified as :

$$C_j = T * L_j \qquad \textbf{(2.2)}$$

The term *T* refers to network tolerance, which should be higher than 1 to allow nodes to take on extra loads in addition to their initial loads.
In the model, the neighbors share their failed neighbor's load, proportionally to the initial load each one has. Considering node *i* as the failed node, the share of neighbor *j*, for example, is specified as:

$$\Delta L_{ij} = L_i \frac{L_J}{\sum_{n \in \Gamma_i} L_n} \qquad \textbf{(2.3)}$$

2.5.3. Quantifying the network damage after failures

There are several methods for quantifying the level of damage. The most common one is based on the largest connected component, also known as the giant component. In this measure, after the cascade process is over, the number of nodes in the giant component will be counted and divided by the number of nodes in the giant component before the breakdown.

The mentioned measure does not correspond to the total number of the overloaded (disabled) nodes during the cascading failure process but only to those located in the giant component. As a small number of failed nodes can cause the severe fragmentation of the network, the giant component measure is questioned by some researchers such as Wang and Rong and Goran Muric. [3][14]
Wang and Rong prefer to quantify the cascading damage with the total number of nodes that become disabled by an attack. For an attack that targets nodes from an initial set *A,* the damage level is calculated as follows:

$$CF_{attack} = \frac{\sum_{i \in A} CF_i}{N_A(N-1)} \qquad \textbf{(2.4)}$$

Where *N* is the number of vertices of the network before the breakdown, CF_i refers to the number of failed nodes after attacking a node from the initial attack set *A*. The CF_i value for each attacked node lies between *0* and *N-1*. N_A is the number of nodes in set *A*. Finally, the CF_{attack} formula is presented as a normalized measure, evaluating the damage result between 0 and 1.

The value of CF_{attack}, demonstrates the network power to mitigate the cascade propagation. In the presented attack quantification, $CF_{attack} = 1$ refers to the result that all nodes of the network have failed due to the attack on the nodes of set *A* and $CF_{attack} = 0$ means nodes were able to take on the excess loads coming from the attacked nodes.

2.5.4. Simulation method

2.5.4.1. Statistical validation for simulations' results

Since part of our tests are conducted on non-deterministic graphs, we ran the same graph generator code of each graph family 20 times to offer validity for their result. For each graph family, 20 sample versions are created, which share a certain order number and other generation parameters such as the probability of edge creation.

Of course, these sample versions resulting from the same graph generator code have many common features, but they also have different values for some

topological measures. These versions might be different in terms of their number of edges, *k-core* range, diameter, and many other topological indices. For instance, a sample of these differences can be seen in annex A. It includes properties of 3 sample versions of the 2D geographical threshold graph family with an order of 216.

Testing each sample version under attack separately and then calculating an average over 20 graphs provided us with the required statistical validation for the result. The attack strategies studied in this work are based on single node failure. To reduce random errors, we considered 5 nodes instead of 1, from the list of nodes sorted for attack purposes. Therefore, for each sample version of the graph, we ran the specific attack code 5 times, each time targeting and failing separately only one of those 5 nodes chosen as attack target potentials. In the end, we calculated an average over the 5 results for each sample version. After repeating the same process for the other 19 sample graphs, we calculated the average result over these 20 sample versions.

2.5.4.2. Networkx package

To generate graphs and calculate network properties such as degree, *k-core*, assortativity, we used the Networkx package in python. In annex B part 1, the graph generator code for graph families deployed in this study has been listed. More on Networkx functions can be read in [22].

CHAPTER 3: STUDY SCENARIOS

3.1. The 1st study scenario

k-core computation, which is described in section 1.2.3, can be done in linear time, and it is one of the methods to classify nodes into different subnetworks.

k-core, as other centrality measures, is used for ranking purposes to sort nodes in an ascending or descending manner. The application of ranking for cascading studies is discussed in section 2.1. We use the low or high ranked nodes as targets in the context of attacks and cascade failure.

Would triggering nodes with the highest *k-core* value bring larger collapse to the network or nodes with the lowest *k-core* value? How would the network performance be under the *k-core* attack compared to other centrality-based attacks that Llorenç Sánchez Marín studied in [15]? These are the main questions we are going to address in the 1st scenario.

3.1.1. Test

For this and other test scenarios, Wang and Rong's cascade model, described in section 2.5, is deployed. As can be seen on page 27, to calculate the loads, a centrality measure should be considered and we chose CFCC and degree centralities for this work. To choose the nodes for the attacks, we sort and rank nodes according to the attack strategy we are studying.

We are going to compare network performance under four attack strategies as specified as follows:

- *k-core* attack with CFCC-load
- CFCC-load based attack
- Degree-load based attack
- *k-core* attack with degree-load

The following graph families with 100 nodes are tested under the mentioned attacks:

- 2D geographical threshold
- 3D geographical threshold
- Watts-Strogatz/ high clustering
- Watts-Strogatz/ low clustering
- Barabasi-Albert
- Erdos-Renyi
- Power-law-cluster

- Random geometric
- Random partition
- Random d-regular
- North American airport network

In this scenario, each graph has been tested under high load and low load conditions. To demonstrate the effect of the initial load on attack result, each high centrality-load attack is performed under two different loads; $\alpha = 0.8$ and $\alpha = 0.5$ and each low centrality-load attack is performed under $\alpha = 0.1$ and $\alpha = 0.3$. The mentioned graph families' attack results can be found out in Annex C. The tested graphs are precisely the same as those studied in [15].

3.1.2. Discussing the result

By assessing the result, we can see for some networks such as Barabasi-Albert and the power-law cluster, the *k-core* attack is not as successful as centrality-load attacks. It is because graphs such as Barabasi-Albert have the same *k-core* value for all nodes, and selecting them this way is the same as choosing them randomly for a random attack.

It should be indicated that for graph families such as the geographical threshold, random geometric, Watts-Strogatz, and Erdos-Renyi, there is not a remarkable difference among results of centrality-load attack strategies; degree-load attack, CFCC-load attack, and *k-core* attack.

Regarding the load type, as observed for some graphs, such as the North American airport network, the network performance differs notably when the load is CFCC-based rather than degree-based.

With respect to the load level of a network, Wang and Rong's research on the USA power grid demonstrates when the network is under low loads, the low degree-load attack hits the network much worse than the high degree-load attack. When the network has high loads, the contrary is true; meaning the high degree-load attack requires a notably higher tolerance level for the network.[3]

By assessing the result of scenario 1, it can be seen that this is not true for all graph families but most of them. For example, for the random d-regular network, all attacks resulted in the same performance regardless of the initial load.

In the end, for a network subject to its graph family, properties, degree, and *k-core* distribution and initial load, some attack strategies may target the nodes more smartly, requiring higher tolerance of the network.

3.2. The 2nd study scenario

To study the impact of *k-core* range, graphs of the same order from the same graph family are required that have comparative *k-core* ranges. When graphs are from the same graph family and have the same number of vertices, they have many common properties that allow us to explore the k-core range's impact precisely. Comparative *k-core* ranges mean that only one side of the ranges should be different among them: the minimum k-core value or the maximum k-core. In the KCBA graph family, which was described in section 2.3, graphs have *k-core* structures from 1-core to k_m-core, in which k_m is an adjustable parameter. So the impact of k_m, the maximum *k-core* value on network performance under attacks can be studied.

As discussed in section 2.3, the result of Dong and Fang's work on the *k-core* structure does not apply to any real or computer-generated networks except their generated graph family of KCBA. This inspired our second study scenario to explore the impact of *k-core* range on robustness of graph families other than KCBA.

3.2.1. Test

Many graph families such as Erdos-Renyi, Watts-Strogatz, and random partition do not generate graphs with comparative *k-core* ranges required for the study. Even there are graph families such as the Barabasi-Albert that for a given number of vertices, all nodes have the same *k-core* value.

Switching our attention to some feasible tests, the 2D geographical threshold graph family and random geometric graph family, whose *k-core* range satisfies the requirement for this analysis, are chosen for the test. As concluded from the attack result of these two graph families in scenario 1 (annex C, Sc1), for the high centrality-load attacks, the highest level of damage occurred when the initial load was the highest, and for the low centrality-load attacks, it occurred when the initial load was the lowest. Therefore the tests of this section are made for $\alpha = 0.1$ and $\alpha = 0.8$.

3.2.1.1. 2D geographical threshold

2D geographical threshold graphs with 100 nodes with the properties given in the next tables are tested for studying the impact of the lowest value of the *k-core* range: the minimum *k-core*. These graphs have different minimum *k-core* values. Indeed, the values of maximum *k-core* are not the same and are a bit different but not to the extent that the minimum *k-core* varies from each other.

Table 3-1- Graph properties

Number of edges	*k-core* range	Avg *k-core*	*k-core* counter	Avg min *k-core* over 5 nodes	Avg max *k-core* over 5 nodes
774	**1** to 13	9.4	1:1, 3:3, 4:2, 5:1, 6:9, 7:8, 8:22, 9:8, 10:3, 11:5, 12:20, 13:18	**2.8**	**13**
847	**2** to 14	10.34	2:1, 4:1, 5:1, 6:5, 7:1, 8:22, 9:15, 10:15, 11:1, 12:3, 13:12, 14:23	**4.6**	**14**
929	**5** to 13	11.07	5: 1, 6 : 3, 7 : 6, 8: 6, 9: 5, 10: 3, 11: 24, 12: 21, 13: 31	**6**	**13**
878	**7** to 12	11.02	7:1, 8:6, 9:8, 10: 3, 11:39, 12:43	**7.8**	**12**

To study the impact of the maximum k-core on the high centrality-load attacks, we will repeat the tests on other graphs that have almost the same minimum *k-core* but different maximum *k-core* values. The tested graphs for this purpose are listed in table 3-2.

Table 3-2- Graph properties

Number of edges	*k-core* range	Avg *k-core*	Avg min *k-core* over 5 nodes	Avg max *k-core* over 5 nodes
746	5 to **11**	9.04	5.2	11
951	4 to **15**	11.49	5	15
1163	4 to **18**	14.06	5.2	18

3.2.1.2. Random geometric

The table beneath gives information about random geometric graphs that we are going to test under low centrality-load attacks. Same as the 2D geographical threshold test, the number of vertices is 100.

Table 3-3- Graph properties

Number of edges	*k-core* range	Avg *k-core*	Avg min *k-core* over 5 nodes	Avg max *k-core* over 5 nodes
1022	3 to 14	12.23	3.6	14
946	5 to 14	12.35	6.8	14
993	10 to 14	12.37	10	14

Next, we will repeat the tests for other random geometric graphs that have almost the same minimum *k-core* but diverse maximum *k-core* values. The chosen graphs are :

Table 3-4- Graph properties

Number of edges	*k-core* range	Avg *k-core*	Avg min *k-core* over 5 nodes	Avg max *k-core* over 5 nodes
797	5 to **12**	10.11	5.4	12
858	5 to **13**	10.58	5.2	13
950	5 to **17**	12.54	5.6	17

The result of attacks on the graphs of tables 3-4, 3-3, 3-2, and 3-1 can be found in Annex C, Sc2 under the table's name.

3.2.2. Discussing the result

From the study of the results of high centrality-load attacks on graphs of tables 3-2 and 3-4, we infer that the value of maximum *k-core* does not influence the attack result. However, assessing the results of low centrality-load attacks on graphs of tables 3-1 and 3-3 reveals the relationship that exists between the minimum *k-core* value and the network performance under attacks.

As the minimum value of the *k-core* range increases, the tolerance level required from the networks drops significantly. Therefore, we can conclude that network performance under the low centrality-load attacks would improve by increasing its minimum *k-core*. It is clear that as the need for the extra load capacity of nodes decreases, the required cost for network safety decreases

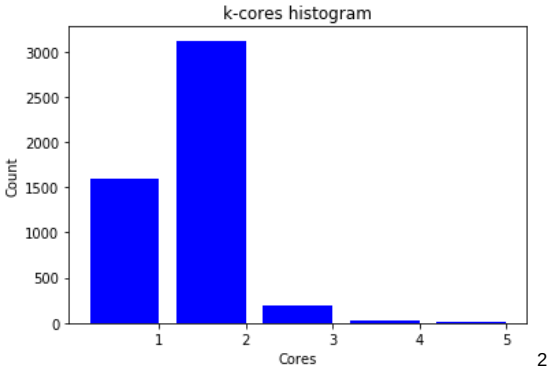
It should be indicated that as the proposed idea comes from experiments on some non-deterministic graphs, real networks should also be tested to prove whether the idea can work on them or not.

Regarding cost concerns, how much would the cost be for such a modification in comparison with the required cost for providing large load capacities for nodes of the unmodified network? All of these need to be assessed, which itself is enough workload for a new thesis.

In the following section, we are going to watch the *k-core* distribution of some graph families. Do a few nodes or a large proportion of nodes have the minimum *k-core* value? We expect that the higher the number of nodes with the lowest *k-core*, the more challenging and expensive it becomes to apply the *k-core* modification.

3.2.2.1. k-core distribution of some networks

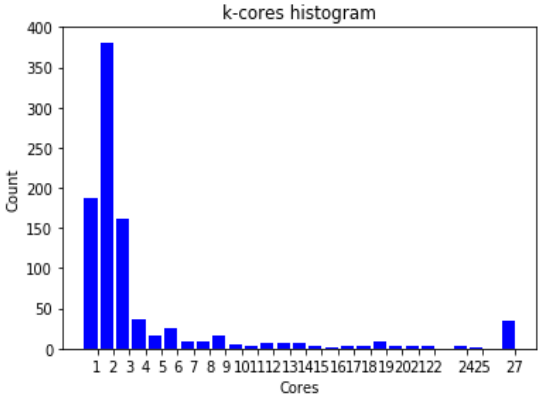
3.2.2.1.1. USA power grid



[2]

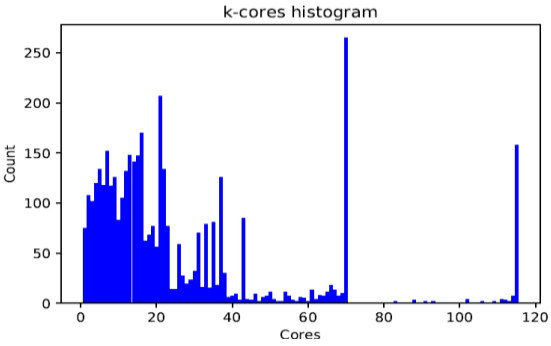
Number of nodes: 4941
Number of edges: 6594
Average degree: 2.6691
---
the proportion of nodes with 1-core:32.1%

3.2.2.1.2. North American airport network



Number of nodes: 940
Number of edges: 3446
Average degree: 7.33191
---
the proportion of nodes with 1-core:19.8%

3.2.2.1.3. Facebook

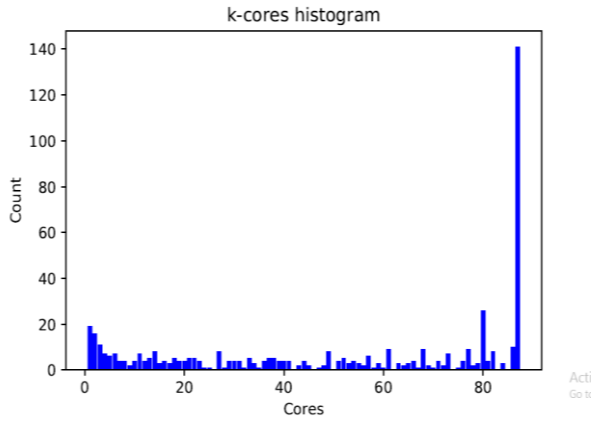


Number of nodes: 4039
Number of edges: 88234
Average degree: 43.6910
k-core range:[1,..., 115]
---
the proportion of nodes with 1-core:1.8%

---

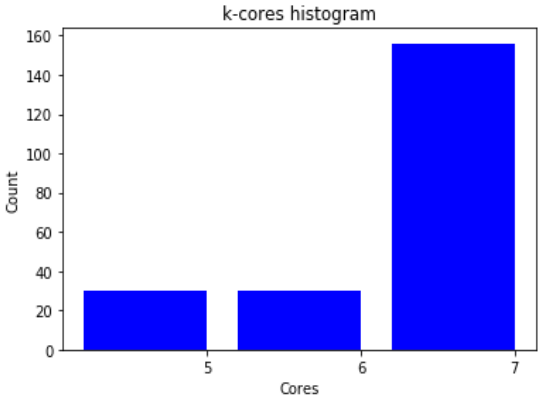
[2] Regarding the *k-core* value of each bar column, the value written at the end of each bar in 'cores' axis, should be considered and not the range [0 to 1], for example. Noting that *k-core* values are integer values.

3.2.2.1.4. European airport route connections 2018



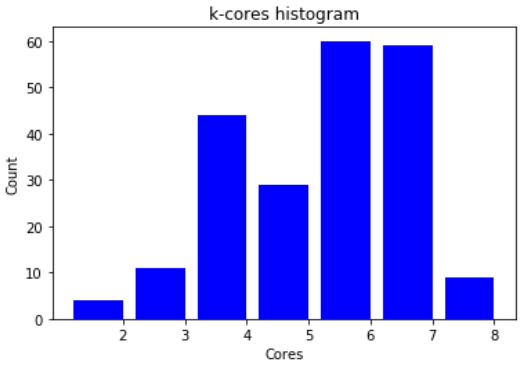
Number of nodes: 509
Number of edges: 22789
Average degree: 89.5442
k-core range: 1 to 87
---
the proportion of nodes with 1-core:3.7%

3.2.2.1.5. Deterministic hierarchical with 216 nodes



Number of nodes: 216
Number of edges: 885
Average degree: 8.1944
---
the proportion of nodes with 5-core:13.8%

3.2.2.1.6. Sample of random geometric with 216 nodes (non -deterministic)



Number of nodes: 216
Number of edges: 915
Average degree: 8.4722

The following table shows the proportion of nodes that belong to the minimum *k-core* subnetwork for graph families of 216-order. It should be noted that for non-deterministic graphs, the average has been obtained over the 20 sample versions.

Table 3-5- Proportion of nodes with min *k-core* value

graph family	min *k-core* value	Nr of nodes with min *k-core* value (%)
connected caveman	9	100%
random d-regular	8	100%
random degree sequence	5	100%
Watts-Strogatz	4	1.9%
Barabasi -Albert	3.8	80%
Erdos-Renyi	1.95	1.4%
random partition	1.8	1%
random geometric	1.45	1.5%
2D geographic threshold	1.05	1.7%

k-core modification might be more viable for networks such as the 2D geographical threshold, random geometric, random partition, Erdos-Renyi, Watts-Strogatz, and real networks such as Facebook and European airport route connections, in which less than 4% of nodes belong to the minimum *k-core* subnetwork.

3.3. The 3rd study scenario

The generic family of deterministic hierarchical graphs presented in [7] and described in section 1.4.6 is an appropriate network model for real networks. This graph family has a specific *k-core* structure. Therefore, we became interested in comparing its performance with other graph families that were tested in the first scenario and also the connected caveman and random degree sequence graphs. The random degree sequence graph has a degree distribution approximate to the degree distribution of the deterministic hierarchical.

Kashif Bilala and his colleagues worked on the relationship between the hierarchy level of a network and its robustness against failures. [13]

]However, they studied graph families and attack strategies that differ completely from those considered in this work. Therefore, we decided to calculate the GRC value for the networks considered in this scenario and explore if the performances of graphs under attacks can be ranked according to their GRC value. The computation of the GRC measure has been explained in section 1.3.4.

The properties of three deterministic hierarchical graphs are specified in the following table:

Table 3-6- Properties of 3 deterministic hierarchical graphs

Nr node	Nr edges	*k-core* range	GRC	Deg-Deg correlation
216	885	[5, 6, 7]	0.71	-0.138
343	1617	[6, 7, 8]	0.74	-0.120
625	2412	[4, 5, 6, 7]	0.56	-0.124

Network order	Avg degree	Avg diameter	Avg distance	Avg clustering	Avg qidx	Avg Wiener
216	8.1944	5	2.424	0.808	-3.6653	56280
343	9.4286	5	2.389	0.842	-5.4389	140133
625	7.7184	7	3.098	0.723	-6.2425	604100

3.3.1. Test

In this scenario, we test the deterministic hierarchical network with 216 nodes along with the same order networks of other graph families such as connected caveman, random degree sequence. The attack result of the mentioned networks can be found in Annex C, Scenario 3.

3.3.2. Discussing the result

Assessing the result of low centrality-load attacks on graphs with 216 nodes, we see a similar performance under all low *k-core*, low CFCC-load, and low degree-load attacks. By exploring the data of those nodes that were chosen as attack targets, we found that nodes with the lowest CFCC-load or lowest degree-load also have the lowest *k-core* value. Thus, the attacked nodes are approximately the same, leading to a similar performance under low centrality-load attacks for each graph family.

The graphs that were considered in the test scenario all have the same order and approximately the same number of edges but different *k-core* structures. This shows the potential for studying the impact of minimum *k-core* and exploring more of the results that were discussed in scenario 2; in section 3.2.2.

In the following table, graph families are sorted according to their maximum tolerance requirement under low centrality-load attack. The worst attack scenario

has been considered for each graph family. For example, if the maximum required tolerance for the deterministic hierarchical graph is 1.01 under low *k-core* attack and is 1.06 under low CFCC-load attack, 1.06 has been considered. It should be indicated that the result for non-deterministic graph families has been averaged over 20 sample versions of them.

Table 3-7- Properties of some networks of order 216

nr of nodes= 216	Max Tol	Avg att *k-core*	Avg *k-core*	Avg *k-core* range	GRC	Clustering coefficient	Deg-Deg Correlation
deterministic hierarchical	1.06	5	6.583	[5,7]	0.7180	0.808	-0.1387
connected caveman	1.12	9	9	[9,9]	0.1000	0.947	-0.102
random regular	1.14	8	8	[8,8]	0.3665	0.030	0
random degree Sequence	1.19	5	5	[5,5]	0.7181	0.137	-0.1179
Watts-Strogatz Lclst	1.23	4.4	5.39	[4,5.5]	0.4011	0.032	-0.0596
Barabasi-Albert	1.23	4	3.99	[3.8, 4]	0.5434	0.102	***-0.1012***
Erdos-Renyi	1.3	2.4	5.03	[1.95, 5.3]	0.4143	0.037	-0.0227
random partition	1.36	2.2	4.83	[1.8, 5.05]	0.4020	0.056	-0.0081
random geometric	1.8	3	5.53	[1.45,8.3]	0.2058	0.665	0.0232
2D geographic thereshold	1.8	2.2	5.046	[1.05, 8.25]	0.2658	0.625	0.5491

The highlighted column are the ones that correlate with the column of the maximum required tolerance.

The graphs with higher tolerance requirements also have positive degree-degree correlation and lower minimum k-core compared to the graphs at the top of the table. No relationship can be observed between the GRC values and values of the maximum tolerance requirement. Thus, the level of hierarchy which is computed by GRC does not have an impact on the robustness of the networks.

In table 3-7, the average attack *k-core* also corresponds to the average minimum *k-core* of the graphs since the attacks are based on the low centrality-load strategy. This is because, as said previously, nodes with the lowest CFCC-load or degree-load also have the lowest *k-core* value.

Looking at the average attack *k-core* cell of the connected caveman network, we see that it is 9-core, which is much higher than the average attack *k-core* of the deterministic hierarchical. From the result of the second study scenario, we

expected the higher the minimum *k-core,* the better would be the network's performance. So why the deterministic hierarchical outperform those with higher minimum *k-core* values? This shows that there should be parameters other than minimum *k-core* determining the level of robustness of networks.

Looking at table 3-7 again, one of the reasons for the better performance of the deterministic hierarchical network might lie with its degree-degree correlation: the most negative one in comparison with others. The impact of negative degree-degree correlation (disassortativity) on robustness and size of the minimum dominating set has been discussed in section 2.4. Disassortativity is known to protect networks to some extent against the spread of viruses. Other success factors might be for the deterministic hierarchical network, which needs further explorations to find out.

3.3.2.1. Description of column names in table 3-7

The 'Avg att *k-core*' column refers to the average *k-core* of nodes that are targeted in the attack. The average *k-core* is calculated considering the *k-core* of all nodes. The *k-core* range column for the deterministic graphs, hierarchical and connected caveman, shows the exact *k-core* range they have. However, for other graph families, an average over the minimum *k-core* of each sample version of the graph family has been calculated and considered as the beginning of the *k-core* range. The same approach is taken for calculating the other side of the *k-core* range, averaging over the maximum k-core values. The GRC measure quantifies the hierarchical level of a graph and has a normalized value between 0 and 1.

3.3.2.2. Discussion on deterministic hierarchical graphs

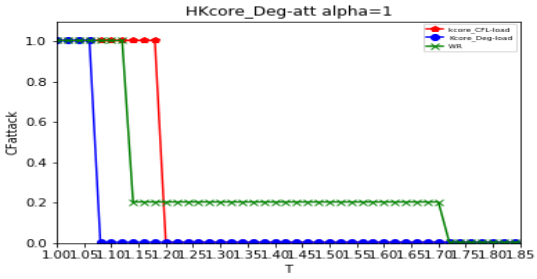
The deterministic hierarchical network exhibits good performance under all attack types; it is not demanding in terms of tolerance requirement except for the high degree attack when the network is under high load ($alpha > 0.7$). In this specific load situation, the mentioned attack requires about 70% extra load capacity to prevent the network from a global collapse. The reason for this underlies the structure of this graph family, having a root node with the largest degree that the rest of the network having been constructed on it. The high degree-load attack exactly targets the root vertex and therefore leads to such a performance.

To show the impact of the failure of the root vertex on network performance, we are going to compare the result of the high degree attack with the result of another high degree attack, in which the root vertex has been excluded from the target list.

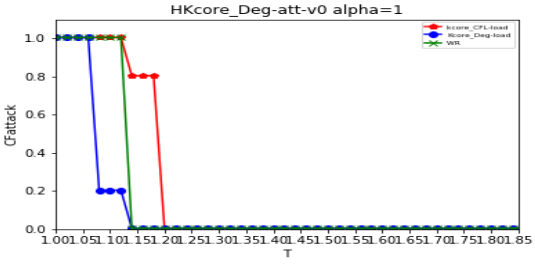
The following figures show the attack results of the deterministic hierarchical graphs with 216, 343, and 625 nodes. The green line is dedicated to the result of high degree-load attack, the red line refers to the result of high *k-core* attack when the load is CFCC-based, and the blue line refers to result of high *k-core* attack when the load is degree-based.

Deterministic hierarchical with 216 nodes

when root vertex is included in the attack list

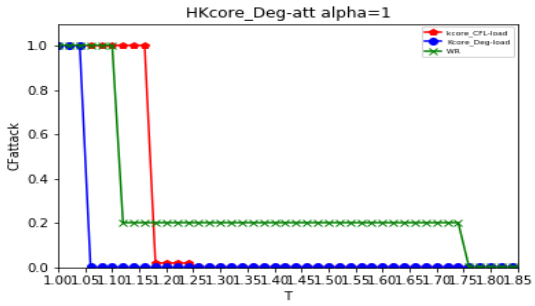


when root vertex is excluded from the attack list

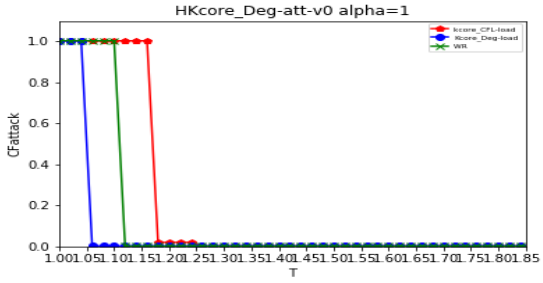


Deterministic hierarchical with 343 nodes

when root vertex is included in the attack list

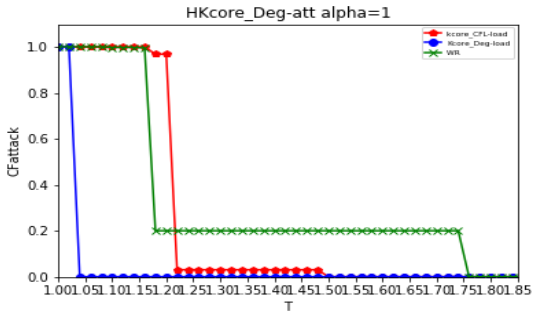


when root vertex is excluded from the attack list



Deterministic Hierarchical with 625 nodes

when root vertex is included in the attack list



when root vertex is excluded from the attack list

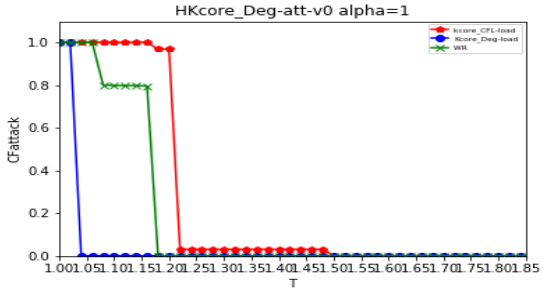


Fig 3-1- High centrality-load attacks

To find the load threshold over which the deterministic hierarchical tolerance requirement increases significantly, we will conduct high centrality-load attacks

under different load conditions. The considered deterministic hierarchical network in this test has the order of 216.

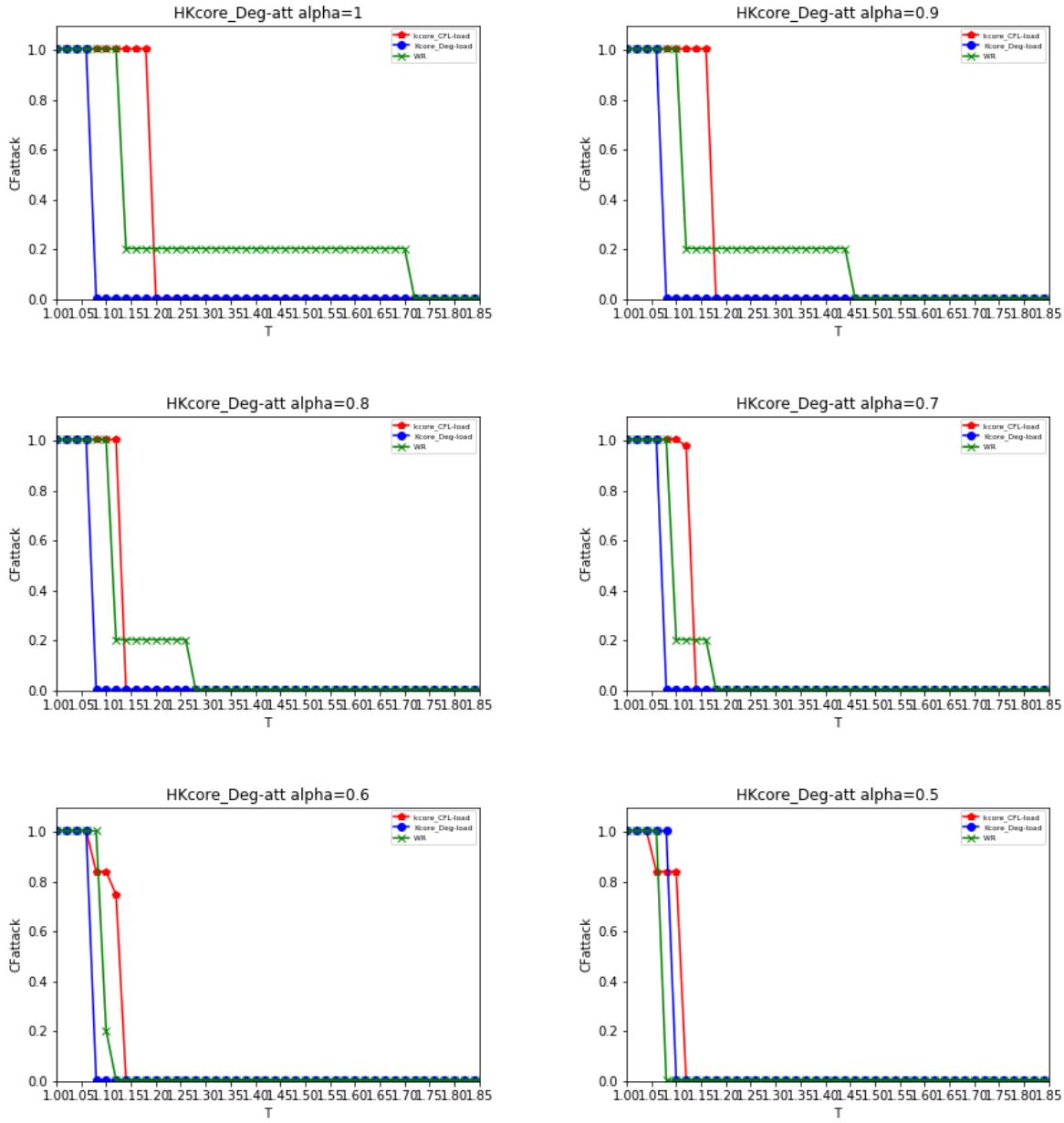


Fig 3-2- High centrality-load attacks on the deterministic hierarchical graph under different loads

It might be possible to provide adequate protection for the root vertex to become kind of bullet-proof under high degree-load attack when the network is under full or high load ($\alpha = 1\ or\ 0.9$). However, regardless of the cost needed to make the root vertex bullet-proof, there is always a way to attack a node and destroy it.

By analyzing more the above figures, the better proposal would be, to consider a control system that sets a limit for the network to avoid working at its full potential under high loads.

The critical load threshold would be the load amount that higher than that, the network tolerance requirement would significantly grow. For example, the critical load threshold is around $\alpha = 0.7$ for the deterministic hierarchical with 216 nodes. The load threshold might vary based on the order of the network. For the deterministic hierarchical networks of orders 216, 343, 625, or 1296, under the circumstance that the mentioned control system is deployed, could mitigate attacks easily if nodes are equipped with 18% extra load capacity according to the standard established model considered in this TFM.

So by controlling the load, the deterministic hierarchical networks could have the potential to be among the best design options for future cascading-resistant networks.

CONCLUSIONS

Each network might be in danger of collapse under certain attack strategies, which these strategies depend on the network structure, properties, type, and amount of load the network has. Recognizing the attacks that severely hit a network is of particular importance for network safety. So that a network can be equipped with tools to protect itself against the worst attack scenario.

k-core targeting nodes based on their *k-core* rank is one of the attack strategies that has been studied in this TFM. Subject to the *k-core* distribution of the target network, this attack sometimes serves as an efficient strategy.

For discovering the maximum tolerance level that a network requires under attacks, we can simulate attacks on the network computer model. The network should be tested under both low and high load conditions.

The result of attacks conducted on the 2D geographical threshold and random geometric networks revealed a strong relationship between the minimum *k-core* value and the tolerance requirement for the network; the higher the minimum *k-core*, the less the tolerance requirement from the network. This introduces a potential solution for increasing the robustness of networks. We can modify a network *k-core* range in a way that increases its minimum *k-core* value. With this modification, networks can mitigate attacks easier and also with less cost, since the capacity requirement determines how much the network safety cost. However, further research is needed to validate the viability of the proposal. The feasibility might depend on the *k-core* distribution of the network. For example, *k-core* modification might be more feasible for networks such as Facebook and European airport route connections, in which less than 4% of nodes have the minimum *k-core* value.

After comparing the performances of the 216-order networks under the same attacks, the following relationships were observed. First, there is no direct relationship between GRC values of networks (their hierarchical levels) and the required values for tolerance under attacks. It sometimes happens that networks with low GRC values perform noticeably better than networks with much higher GRC values. Second, the is a relationship between the values of minimum *k-core* and the required values for the networks' tolerance levels. A network with a higher minimum *k-core* can manage attacks with less tolerance requirement than a network with a lower value for its minimum k-core. Third, the negative degree-degree correlation (disassortativity) profoundly impacts the tolerance requirement, which dominates the effect that the minimum *k-core* value has. It happens that networks with higher disassortativity and lower minimum *k-core* value outperform networks with lower disassortativity and higher minimum *k-core*

value. So it can be inferred that disassortativity is the determining factor in network robustness.

Regarding the deterministic hierarchical networks, they manage and mitigate all attacks easily except for the high degree attack strategy when the initial load is high. This underlies the network structure, having a vertex at the core which the rest of the network has been built upon that. Having the highest number of connections; highest degree centrality makes this node the first target of high degree attack.

However, if a simple control strategy is deployed that sets a limit for loads of the network, the network can easily manage the high degree attack. For instance, for the deterministic hierarchical network of 216-order, limiting the load by $a < 0.7$, the required tolerance to resist all attacks is only 18% extra load capacity according to the standard established model considered in this TFM. So the deterministic hierarchical networks equipped with a control section can be used as design models for future cascading-resistant networks.

ACRONYMS

CFCC	Current Flow Closeness Centrality
GRC	Global Reaching Centrality

BIBLIOGRAPHY

1. Estrada, E., and P. Knight. *A First Course in Network Theory*. New York: Oxford University Press Inc, 2015

2. Barabasi, A., and M. Posfai. *Networks Science*.United Kingdom: Cambridge University Press, 2014

3. Wang, J., and L. Rong. “Cascade-based attack vulnerability on the US power grid” *Safety Science 47*. 5 February 2009: 1332-1336.

4. Mones, E. et al. “Hierarchy measure for complex networks,” *PLoS ONE* 7(3). 28 March 2012.

5. Kasthurirathna, D. et al. “Network robustness and topological characteristics in scale-free networks”, *2013 IEEE Conference on EAIS*: 122-129

6. Dong, Z. et al. “The influence of the depth of *k-core* layers on the robustness of interdependent networks against cascading failures”, *INT J MOD PHYS C 28(02)*, Feb 2017: 1750020

7. Barrière, L. et al. “Deterministic hierarchical networks”, *J. Phys. A: Math. Theor 49(2),* 3 May 2016: 225202

8. Bradonji´c, M. et al. “The structure of geographical threshold graphs” *Internet Mathematics 5(1-2),* 14 Oct 2009: 111–137

9. Brandes, U., and D. Fleischer.” Centrality measures based on current flow”, *STACS 2005*: 533-544

10. Pei, S. et al.” Theories for influencer identification in complex networks”, *Complex Spreading Phenomena in Social Systems*, 2018: 125-148

11. Del Ferraro, G. et al. “Finding influential nodes for integration in brain networks using optimal percolation theory”, *Nature Communications 9*, Article number: 2274, 11 June 2018

12. Takemoto, K., and T. Akutsu. “Analysis of the effect of degree correlation the size of minimum dominating sets in complex networks”. *PLoS ONE 11(6)*: e0157868, 21 Jun 2016

13. Bilal, K. et al. “Robustness quantification of hierarchical complex networks under targeted failures”. *Computer and Electrical Engineering 72*, Nov 2018: 112-124

14. Muric, G. *Resilience of the Critical Communication Networks Against Spreading Failures - Case of the European National Research and Education Networks*. Technischen Universitat Dresden: Ph.D. Thesis, 2017

15. Sánchez Marín, LIorenc. *Cascade Failures in Complex Networks*. Universitat Politècnica de Catalunya: Master Thesis, 2018.

16. Holme, P., and B. Kim. “Growing Scale-free Networks with Tunable Clustering”. Phys. Rev. E 65, 026107. 11 Jan 2002

17. D. Malliaros, F. et al. “The core decomposition of networks: theory, algorithms and applications”. The VLDB Journal 29. Nov 2019

18. Rahimian, F. et al. “A distributed algorithm for large-scale graph partitioning”. ACM Transactions on Autonomous and Adaptive Systems 10(2). Jun 2015: 1-24

19. Zafarani, R. et al. *Social Media Mining An Introduction*. Cambridge University Press, 2014

20. Burchill, J.” Algorithm Design and Analysis” From Slideplayer – A Web Resource. https://slideplayer.com/slide/3353240/

21. Weisstein, Eric W. "Caveman Graph." From MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/CavemanGraph.html

22. Hagberg, A. et al. *NetworkX Reference Release 2.5*. 22 Aug 2020

ANNEXES

TITLE: Models for cascade failures in complex networks and systems

MASTER DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)

AUTHOR: Shima Aflatounian

ADVISOR: Francesc Comellas Padró

DATE: November, 7th 2020

ANNEXES

Annex A - Table

Properties of some 2D geographical threshold graphs with 216 nodes generated by Networkx

Sample Nr	Nr node	Nr edges	Avg *k-core*	*k-core* range	Avg degree	Degree range	GRC
19	216	861	5.18	2 to 9	7.97	2 to 25	0.2661
18	216	937	5.49	1 to 8	8.67	1 to 34	0.2764
14	216	880	5.23	1 to 9	8.14	1 to 29	0.2906

Sample Nr	clustering coefficient	diameter	avg dist	Wiener index	Q index	Correlation
19	0.658	12	5.4081	125575	-2.3838	0.084
18	0.6977	14	0.6977	124348	-2.8008	0.0485
14	0.6815	13	5.2434	121751	-2.4667	-0.0759

Annex B - Codes

B-P1. Graph generator

It should be noted that the generated graph should be checked to be connected

```
#gr=nx.random_regular_graph(8, 216, seed=None)
#gr=nx.geographical_threshold_graph(216,150, dim=2)
#gr=nx.erdos_renyi_graph(216, 0.038, seed=None, directed=False) #216 node & 880 E
#gr=nx.watts_strogatz_graph(216, 9, 0.9, seed=None) #216 node & 864
#gr=nx.random_geometric_graph(216, 0.118) #216 node & 864
#gr=nx.random_partition_graph([55,55,55,51], 0.1, 0.015, seed=None, directed=False)
#gr=nx.barabasi_albert_graph(216,4,seed=None)#216 node & 864
# deg seq
idx=0
while(idx<=19):
    ul=[5 for i in range(30)]  #30 deg 5
    vl=[6 for i in range(75)]  #25 deg 6
    wl=[7 for i in range(75)]  # 125 deg 7
    xl=[9 for i in range(30)]  #30 deg 9
    tl=[34 for i in range(5)]  #5 deg 34
    yl=[55 for i in range(1)]
    zl=ul+vl+wl+xl+tl+yl
```

```
    gr=nx.random_degree_sequence_graph(zl,seed=idx,tries=10)
                                    for nod in range(25,123):
        if gr.has_edge(nod,215) or gr.has_edge(215,nod):
            print('.'),
        else:
            gr.add_edge(nod,215)
```

B-P2. Graph properties

The following code is written for not deterministic graphs, and attributes are computed by calculating each sample graph's feature and averaging over all of them. The ‘reps’ will determine how many sample version for each graph family is intended to be created. For computing features of deterministic hierarchical graphs, a similar version of this code has been used with some modification.

```
#importing libraries

from __future__ import print_function
import time
import itertools
import statistics
from random import seed
from random import randint
import networkx as nx
from pylab import *

from  math import log
import matplotlib.pyplot as plt
import numpy as np
from collections import Counter
from collections import defaultdict
import collections
from itertools import chain
import copy
#####################################################################
print('works with Python 3.7.6 i NetworkX 2.4\n')
import sys
#####################################################################
# defining functions

def getList(dict):
    return dict.keys()
def FCcommunicability_exp(G):
    import scipy.linalg
    nodlst = list(G) # ordering of nodes in matrix
    A = nx.to_numpy_matrix(G,nodlst)
    # convert to 0-1 matrix
    # convert to 0-1 matrix
```

```
    A[A!=0.0] = 1
    # communicability matrix
    expA = scipy.linalg.expm(A.A)
    mapping = dict(zip(nodlst,range(len(nodlst))))
    c = {}
    for u in G:
        c[u]={}
        for v in G:
            c[u][v] = float(expA[mapping[u],mapping[v]])
    return c
 ##############
# Communicability distance index (big Gamma)

def comm_dist_idx(G):
    Gamma_idx=0.0
    cdst=FCcommunicability_exp(G)
    for u in G:
        for v in G:
            val=cdst[u][u]+cdst[v][v]-2*cdst[u][v]
            val=abs(val)
            Gamma_idx+=sqrt(val)
    return Gamma_idx/2

def wiener_idx(G, weight=None):
    # compute sum of distances between all node pairs
    # (with optional weights)
    wiener=0.0
    if weight is None:
        for n in G:
            path_length=list(nx.single_source_shortest_path_length(G,n).values())
            wiener+=sum(path_length)
    else:
        for n in G:
            path_length=list(nx.single_source_dijkstra_path_length(G,n,weight=weight).values())
            wiener+=sum(path_length)
    return int(wiener/2.0)
 ##############
# Q index
def Q_idx(G,wieneridx):
    return log(0.857763885*wieneridx/comm_dist_idx(G))
#################################################################
#choosing the graph we tend to check properties

filename="Graphs_edges-file/216g/random degree sequence/216_rnd_deg_seq_"
grafname="deg_seq-"

#filename="Graphs_edges-file/216g/random regular/216_rnd_regular_"
#grafname="ran_reg-"
```

```
#filename="Graphs_edges-file/216g/random geometric/216_random_geometric_"
#grafname="ran_geo-"

#filename="Graphs_edges-file/216g/random partition/216_random_partition_"
#grafname="ran_part-"

#filename="Graphs_edges-file/216g/erdos-renyi/216_erdos_reyni_"
#grafname="er_renyi-"

#filename="Graphs_edges-file/216g/watts strogatz/216_watts_strogatz_"
#grafname="watts_str-"

#filename="Graphs_edges-file/216g/2d geo threshold/216_dim2_geo_thr_"
#grafname="2D-geo_thr-"

#filename="Graphs_edges-file/216g/barabasi albert/216_barabasi_albert_"
#grafname="BA-"

 ##############
# setting initial variables

reps=20

KcrRange=[]
DegRange=[]

GRC=[]

B=[i for i in range(reps)]
C=[i for i in range(reps)]

CoreValues=[[] for i in range(reps)]
DegVal=[[] for i in range(reps)]

Avg_K-core_Total=[]

corr=[i for i in range(reps)]
diameter=[i for i in range(reps)]
avgdist=[i for i in range(reps)]
clust=[i for i in range(reps)]
wiener=[i for i in range(reps)]
qidx=[i for i in range(reps)]
avgdeg=[i for i in range(reps)]
histgrm=[]

LkcrRange=[]
HkcrRange=[]
```

```
LdegRange=[]
HdegRange=[]

Avg_K-core=[i for i in range(reps)]

GRC1=[i for i in range(reps)]
g=[i for i in range(reps)]
#####################################################################
# Main Execution

for idx in range(reps):

    print('\n\n ',grafname,'\n')
    print('\n\n ',idx,'\n')
    gr=nx.read_edgelist(filename+str(idx)+'.edges',nodetype=int) #idx r
    g[idx]=gr

    sum2=0
    nrNode=gr.number_of_nodes()
 ##############
# Degree Distribution + averaging on degree range

    Gdegree = sorted([d for n, d in gr.degree()], reverse=True)  # degree sequence
    c = collections.Counter(Gdegree)
    c=dict(sorted(c.items()))
    C[idx]=c
    for key, val in sorted(c.items()):
          sum2=(key*val)+sum2
    avgdeg[idx]= sum2/nrNode
    Degg=getList(c)
    DegsSort=sorted(Degg)
    DegVal[idx]=DegsSort
    LdegRange.append(DegsSort[0])
    HdegRange.append(DegsSort[-1])
 ##############
# k-core Distribution

    Gcore=nx.core_number(gr)
    sum1=0
    A=nx.core_number(gr).values()
    b=collections.Counter(A)
    b=dict(sorted(b.items()))
    B[idx]=b
    res = defaultdict(list)
    for key, val in sorted(Gcore.items()):
         res[val].append(key) # here val is node number have that k-core
    GroupCore=dict(res)
```

```
    for key, val in sorted(res.items()):
        sum1=(key*(len(val)))+sum1 # len val return number of nodes
    Avg_K-core[idx]= sum1/nrNode
    cores=getList(GroupCore)
    coresSort=sorted(cores)
    CoreValues[idx]=coresSort
    LkcrRange.append(coresSort[0])
    HkcrRange.append(coresSort[-1])
     ##############
# General properties

    diameter[idx]=nx.diameter(gr)
    avgdist[idx]=round(nx.average_shortest_path_length(gr),4)
    clust[idx]= round(nx.average_clustering(gr),4)
    wiener[idx]=round(wiener_idx(gr),4)
    qidx[idx]=round(Q_idx(gr,wiener[idx]),4)
    corr[idx]=round(nx.degree_assortativity_coefficient(gr),4)
 ##############

# GRC computation_hiearchical level measure

    nrNode=gr.number_of_nodes()
    print(nx.info(gr))
    closeness=nx.closeness_centrality(gr, u=None, distance=None, wf_improved=False)
    AllVal=closeness.values()
    maxCent=max(AllVal)
    sumGrc=0
    for j in gr:
        CentJ=closeness[j]
        CentJJ=CentJ/(nrNode-2)
        sumGrc=sumGrc+(maxCent-CentJJ)
    grrc=sumGrc/(nrNode-1)
    GRC1[idx]=round((grrc),4)

 ##############
# Averaging over reps of each graph family
### Deg Distribution AVG
AVGdeg=round(statistics.mean(avgdeg),4)
LdegRange1=statistics.mean(LdegRange)
HdegRange1=statistics.mean(HdegRange)
DegRange.append(LdegRange1)
DegRange.append(HdegRange1)

### General graph parameters AVG
AVGdiameter=statistics.mean(diameter)
AVGavgdist=round(statistics.mean(avgdist),4)
AVGclust=round(statistics.mean(clust),4)
```

```
AVGwiener=statistics.mean(wiener)
AVGqidx=round(statistics.mean(qidx),4)
AVGcorr=round(statistics.mean(corr),4) # random regular AVGcorr=1

### K-core Distribution AVG
Avg_K-core_Total= round(statistics.mean(Avg_K-core),4)
LkcrRange1=statistics.mean(LkcrRange)
HkcrRange1=statistics.mean(HkcrRange)
KcrRange.append(LkcrRange1)
KcrRange.append(HkcrRange1)
### GRC AVG
GRC=round(statistics.mean(GRC1),4)

 ##############
### Saving graph Info

f=open('GRC_Gcore_Ginf_r_'+grafname+'.txt','w')
f.write(grafname+'\n')
f.write('\n Average of this graph family ! \n Avg degree: '+str(AVGdeg)+'\n')
f.write(' \n diameter: '+str(AVGdiameter)+'\n avg distance:'+str(AVGavgdist)+'\n
clustering:'+str(AVGclust)+'\n Wiener index:'+str(AVGwiener)+'\n Q index:'+str(AVGqidx)+'\n
Correlation:'+str(AVGcorr)+'\n')
f.write('the avg hierarchy = '+str( GRC)+'\n')
f.write('Avg k-core = '+str(Avg_K-core_Total)+'\n')
f.write('the avg k-core range =:[min core,max core] '+str( KcrRange)+'\n')
f.write('################')

f.write('\n each random version info \n')
for idx in range(reps):

      f.write(nx.info(g[idx])+'\n')
      f.write('Degree range '+str(DegVal[idx])+'\n \n')
      f.write('Degree distribution  1st: deg value, 2nd: counter'+str(C[idx])+'\n \n')
      f.write('\n diameter: '+str(diameter[idx])+'\n avg distance:'+str(avgdist[idx])+'\n
clustering:'+str(clust[idx])+'\n   Wiener index:'+str(wiener[idx])+'\n Q index:'+str(qidx[idx])+'\n
Correlation:'+str(corr[idx])+'\n')
      f.write('GRC value:   '+str(GRC1[idx])+'\n')
      f.write('k-core range '+str(CoreValues[idx])+'\n')
      f.write('collection k-core  1st:value, 2nd: counter'+str(B[idx])+'\n')
      f.write('Avg k-core of this graph '+str(Avg_K-core[idx])+'\n')
      f.write('####################'+'\n')
f.close()
################################################################
```

B-P3. Comparing the performance of different graphs under the same attack

The following code is written according to the high *k-core* attack strategy, which targets nodes with the highest *k-core* values. The codes for other attack strategies is the same with some minor modification :
The only difference between high and low centrality-load attacks of the same attack strategy is reversing the attack target list in ascending or descending order.

The *cask-coredownup()* function is used to sort nodes as needed for *k-core* attack strategy and *Listdownup()* for centrality-load based attack strategies, CFCC-load, and degree-load.

```
#importing libraries

from __future__ import print_function
import time
import itertools
import statistics
from random import seed
from random import randint
import networkx as nx
from pylab import *
from  math import log
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from collections import Counter
from collections import defaultdict
import collections
from itertools import chain
import os
######################################################################
print('works with Python 3.7.6 i NetworkX 2.4\n')
import sys
print('PYTHON version',sys.version)
print('NX version',nx.__version__)
starttime = time.time()
######################################################################
# defining functions

### calculating initial load based on the centrality and certain alpha, in the end, refer to load capacity function with certain Tol

def ini_carr(alpha,tol):
    for j in g:
```

```
        kj=gload[j]
        v=g.neighbors(j)
        sumkm=0
        for m in v:
            sumkm=sumkm+gload[m]
        Lj=(kj*sumkm)**alpha
        g.nodes[j]['Lj']=round(Lj,4)
    carr_max(tol)
##############
### calculating maximum load capacity of each node

def carr_max(T):
    for j in g:
        Cj=(T*g.nodes[j]['Lj'])
        g.nodes[j]['Cj']=round(Cj,4)
##############
### distributing the load of the failed node between its ok neighbors

def distrib_carr(i):
    sum_carr_veins=0
    for j in g.neighbors(i):
        if g.nodes[j]['OK']==1:
            sum_carr_veins=sum_carr_veins+g.nodes[j]['Lj']
    for j in g.neighbors(i):
        if g.nodes[j]['OK']==1 and sum_carr_veins!=0:
            newLj=g.nodes[j]['Lj']+g.nodes[j]['Lj']*g.nodes[i]['Lj']/sum_carr_veins
            g.nodes[j]['Lj']=round(newLj,4)
##############
def getList(dict):
    return dict.keys()
################
# making the attack target list by sorting the nodes by their ranking  in terms of k-core  or  their
load capacity
# the load capacity is based on initial load, In this study is CFCC-load or degree-load
# list.reverse() for attacking the list in Ascending order

def listdownup_cfl(): # same valid for degree-load attack
    gsort=sorted( list(g.nodes(data=True)), key=lambda labels: labels[1]['Cj'])
    ordlist=[]
    for i in range(len(gsort)):
        nod=gsort[i][0]
        ordlist.append(nod)
    return ordlist
##
def  cask-coredownup(): # for k-core attack strategy
     Gcore=nx.core_number(g)
     for j in g:
            kcj=Gcore[j]
```

```
            g.nodes[j]['kcj']=kcj
    gsort=sorted(list(g.nodes(data=True)), key= lambda labels : labels[1]['kcj'])
            ordlist=[]
    for i in range(len(gsort)):
            nod=gsort[i][0]
            ordlist.append(nod)
    return ordlist
################
# cascade phenomena function, turn the OK status of the node to zero, for each neighbor of the failed node, if the new assigned load plus its initial load, exceeds the load capacity it has, it will also fail and load distribution will continue till we check on the last reachable node
# the output of this function is a list containing all failed nodes through these cascade phenomena

def cascada(v):
    eccg=nx.eccentricity(g)
    ecc=eccg[v]
    for i in g:
        g.nodes[i]['OK']=1
    g.nodes[v]['OK']=0
    dist=-1
    finit=0
    oldlevel=[v]
    hanfallat=[oldlevel]
    while (finit==0):
        dist=dist+1
        newlevel=[]
        for pos in oldlevel:
            distrib_carr(pos) #new neighbour's node load
            for j in g.neighbors(pos): #which nodes are failing?
            if g.nodes[j]['OK']==1:
                        if ((g.nodes[j]['Lj']) >= (g.nodes[j]['Cj'])):
                        g.nodes[j]['OK']=0
                            newlevel.append(j)
        hanfallat.append(newlevel)
        oldlevel=newlevel
        if (dist==ecc):
            finit=1
    return hanfallat
#####################################################################
# setting initial variables

centrality= 'CFCC '
seed(1)
reps=20
print( "\\\ ATENTION: centrality:"+centrality+"   repeticions: ",reps,"\n")

alfa=0.9
```

```
alpha='0_9'
dt = 0.02
ngraus=5
tolvals = arange(1.00, 1.85, dt)
granslist=[]
gransnodelist=[]
filename=[i for i in range(10)]
grafname=[i for i in range(10)]
granslist=[i for i in range(10)]
CFlist20L=[[] for i in range(10)]
g=[i for i in range(10)]
filename[0]="Graphs_edges-file/DetHier/DetHier_216n_885e"
filename[1]="Graphs_edges-file/216g/209_1045ConCaveK19x11"
filename[2]="Graphs_edges-file/216g/random degree sequence/216_rnd_deg_seq_"
filename[3]="Graphs_edges-file/216g/random regular/216_rnd_regular_"
filename[4]="Graphs_edges-file/216g/random geometric/216_random_geometric_"
filename[5]="Graphs_edges-file/216g/random partition/216_random_partition_"
filename[6]="Graphs_edges-file/216g/erdos-renyi/216_erdos_reyni_"
filename[7]="Graphs_edges-file/216g/watts strogatz/216_watts_strogatz_"
filename[8]="Graphs_edges-file/216g/2d geo threshold/216_dim2_geo_thr_"
filename[9]="Graphs_edges-file/216g/barabasi albert/216_barabasi_albert_"
grafname[0]="DetHier_216n-"
grafname[1]="Concaveman-"
grafname[2]="deg_seq-"
grafname[3]="ran_reg-"
grafname[4]="ran_geo-"
grafname[5]="ran_part-"
grafname[6]="er_renyi-"
grafname[7]="watts_str-"
grafname[8]="2D-geo_thr-"
grafname[9]="BA-"
#####################################################################
# Main execution part

### deterministic hierarchical Graph Cascading test

idx1=0
if idx1==0:
    idx=idx1
    att_k-core_save=0
    print('\n\n ',idx,'\n')
    print( 'alpha:',alfa)
    g=nx.read_edgelist(filename[idx]+".edges",nodetype=int) #idx refer to rep 1...to 20
    g.name=grafname[idx]
    print('\n\n ',grafname[idx],'\n')
    gload=nx.current_flow_closeness_centrality(g)
   Gcore=nx.core_number(g)
    print(time.asctime( time.localtime(time.time()) ))
```

```
    CFlist=[]
    tol=1.00
    ini_carr(alfa,tol) #Asigna una carga a cada nodo (Todos activos)
    while (tol<1.85):
        if len(CFlist)==0:
            load= cask-coredownup()          ## sort the attack list
            load.reverse()
            grans=[load[i] for i in range(ngraus)]
            gransload=[]
            for i in grans:
            gransload.append(Gcore[i])
            att_k-core_save+=Gcore[i]
            AVGattK-core[0]=att_k-core_save/5

        start = time.time()
        faillist=[]
        for i in grans:
            ini_carr(alfa,tol)
            fail=cascada(i) #nodes that failed
            flat=list(itertools.chain.from_iterable(fail)) #give all nodes in 'fail'
            CFi= len(flat)
            faillist.append(CFi)
            elapsed = (time.time() - start)
        CFattck=round(sum(faillist)/(ngraus*(g.order()-1)*1.0),4)
        CFlist.append(CFattck)
        tol=tol+dt
    CFlist20L[idx].append(CFlist)
    gransnodelist.append(grans)

    Lj0=[]
    Ljn=0
    for i in grans:
        Ljn=g.nodes[i]['Lj']
        Lj0.append(Ljn)
        inf1=list(zip(grans, gransload,Lj0))
    granslist[idx]=inf1
os.system( "say finished hierarchical" )
################
### Connected caveman Graph Cascading test

idx1=1
if idx1==1:
    idx=idx1
    att_k-core_save=0
    print('\n\n ',idx,'\n')
    print('\n\n ',grafname[idx],'\n')
    print( 'alpha:',alfa)
    g=nx.read_edgelist(filename[idx]+".edges",nodetype=int) #idx refer to rep 1...to 20
```

```
    gload=nx.current_flow_closeness_centrality(g)
    Gcore=nx.core_number(g)
    print(time.asctime( time.localtime(time.time()) ))
    CFlist=[]
    tol=1.00
    ini_carr(alfa,tol) #Asigna una carga a cada nodo (Todos activos)

    while (tol<1.85):
        if len(CFlist)==0:
            load= cask-coredownup()

            load.reverse()
            grans=[load[i] for i in range(ngraus)]
           #print '\nGRANS '+str(grans)
            gransload=[]

            for i in grans:
            gransload.append(Gcore[i])
            att_k-core_save+=Gcore[i]
            AVGattK-core[1]=att_k-core_save/5
        start = time.time()
        faillist=[]
        for i in grans:
            ini_carr(alfa,tol)
            fail=cascada(i) #nodes that failed
           flat=list(itertools.chain.from_iterable(fail)) #give all nodes in 'fail'
            CFi= len(flat)
            faillist.append(CFi)
            elapsed = (time.time() - start)
        CFattck=round(sum(faillist)/(ngraus*(g.order()-1)*1.0),4)
        CFlist.append(CFattck)
        tol=tol+dt
    CFlist20L[idx].append(CFlist)
    gransnodelist.append(grans)

    Lj0=[]
    Ljn=0
    for i in grans:
        Ljn=g.nodes[i]['Lj']
        Lj0.append(Ljn)
        inf1=list(zip(grans, gransload,Lj0))
    granslist[idx]=inf1
os.system( "say finished caveman" )
###############
### non-determenistic Graphs Cascading test

for idx2 in range(2,10):
    idx=idx2
```

```
os.system( "say starting " +str(idx)+grafname[idx])
print('\n\n ',idx,'\n')
print('\n\n ',grafname[idx],'\n')
att_kk-core=[]
for idxSt in range(reps):
    att_k-core_save=0
    g=nx.read_edgelist(filename[idx]+str(idxSt)+".edges",nodetype=int) #idx refer to rep 1...to
20
    gload=nx.current_flow_closeness_centrality(g)
    Gcore=nx.core_number(g)
    CFlist=[]
    tol=1.00
    ini_carr(alfa,tol) #Asigna una carga a cada nodo (Todos activos)

    while (tol<1.85):
        if len(CFlist)==0:
        load= cask-coredownup()
        load.reverse()
        grans=[load[i] for i in range(ngraus)]
        gransload=[]
        att_kk-core=[]

        for i in grans:
                gransload.append(Gcore[i])
                    att_k-core_save+=Gcore[i]
          att_kk-core.append(att_k-core_save/5)
        start = time.time()
       faillist=[]
        for i in grans:
                    ini_carr(alfa,tol)
                    fail=cascada(i) #nodes that failed
           flat=list(itertools.chain.from_iterable(fail)) #give all nodes in 'fail'
                    CFi= len(flat)
                    faillist.append(CFi)
                    elapsed = (time.time() - start)
       CFattck=round(sum(faillist)/(ngraus*(g.order()-1)*1.0),4)
        CFlist.append(CFattck)
        tol=tol+dt
    CFlist20L[idx].append(CFlist)
    gransnodelist.append(grans)

    Lj0=[]
    Ljn=0
    for i in grans:
        Ljn=g.nodes[i]['Lj']
        Lj0.append(Ljn)
    inf1=list(zip(grans, gransload,Lj0))
    granslist[idx]=inf1
```

```
    os.system( "say finishing " +str(idx)+grafname[idx])
    AVGattK-core[idx]=statistics.mean(att_kk-core) # for each non-deterministic graph family

 ##############
### Saving graph Info

print
t = arange(1.00, 1.85, dt)
processtime = time.time()-starttime
f=open('results/216/info/216-st_r'+str(reps)+'_Hkcr_att_ a'+alpha+'_info.txt','w')
f.write('\nreps: '+str(reps)+'\nalpha:'+str(alfa)+'\ngraus:'+str(ngraus)+'\ncentrality:
'+centrality+'\n')
f.write('\nprocess time [s]: '+str(processtime)+'\n')
f.write('\ntolvals: '+str(tolvals)+'\n')
for idx in range(10):
      f.write('\ngraf type: '+grafname[idx])
      f.write('the avg k-core of targeted nodes are '+str( AVGattK-core[idx])+'\n')
#       f.write('\n\nLowCore\n[node  k-core load
]\n'+str(granslist[idx])+'\n\n'+str(CFlist20L[idx])+'\n')
      f.write('\n ##############################################\n')
f.close()

### Plotting cascading quantification in each tolerance

print(tolvals)
ll=[i for i in range(10)]
for idx in range(10):
   ll[idx] =  [round(float(sum(col))/len(col),4) for col in zip(*(CFlist20L[idx]))]
   print('------------------')
   print (ll[idx])

plot(tolvals, ll[0] , 'r-',marker='p',  label=grafname[0]+'_attKcr:'+str(AVGattK-core[0]))
plot(tolvals, ll[1] , 'b-',marker='o',  label=grafname[1]+'_attKcr:'+str(AVGattK-core[1]))
plot(tolvals, ll[2],  'g-',marker='x', label=grafname[2]+'_attKcr:'+str(AVGattK-core[2]))
plot(tolvals, ll[3],  'y-', marker='p', label=grafname[3]+'_attKcr:'+str(AVGattK-core[3]))
plot(tolvals, ll[4],  'k-', marker='o', label=grafname[4]+'_attKcr:'+str(AVGattK-core[4]))
plot(tolvals, ll[5],  'c-',marker='x',  label=grafname[5]+'_attKcr:'+str(AVGattK-core[5]))
plot(tolvals, ll[6],  'm-',marker='o', label=grafname[6]+'_attKcr:'+str(AVGattK-core[6]))
plot(tolvals, ll[7],  'b-',marker='v', label=grafname[7]+'_attKcr:'+str(AVGattK-core[7]))
plot(tolvals, ll[8], 'k-',marker='*', label=grafname[8]+'_attKcr:'+str(AVGattK-core[8]))
plot(tolvals, ll[9], 'm-',marker='s', label=grafname[9]+'_attKcr:'+str(AVGattK-core[9]))

axis([1,1.3,0,1.1])
tcks = arange(1.00, 1.85, 0.05)
xticks(tcks)
xlabel('T')
ylabel('CFattack')
```

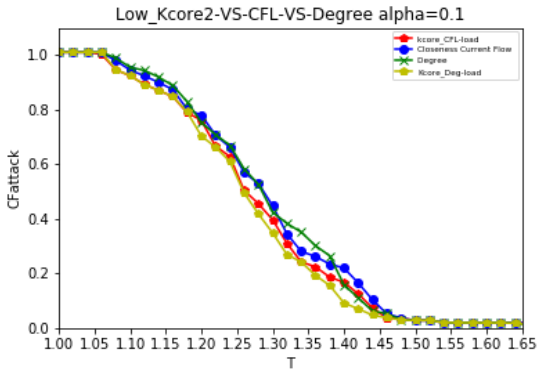
```
title('r'+str(reps)+'_High_K-core_ATT'+'-'+centrality+' a='+str(alfa)+' rep: '+str(reps)+' infail:
'+str(ngraus),fontsize='xx-small' )
legend(loc='best',fontsize='xx-small')
savefig('results/216/216-st_r'+str(reps)+'_Hkcr_ATT'+'-a'+alpha+'.png')
show()
```

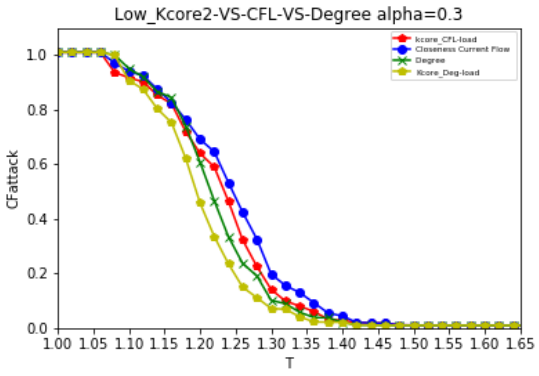
Annex C - Results

C-Scenario 1

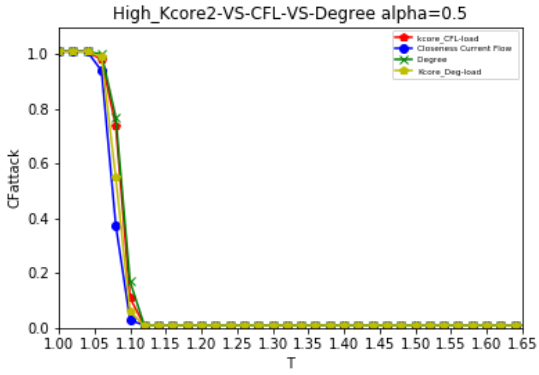
C-Sc1-2D geographical threshold

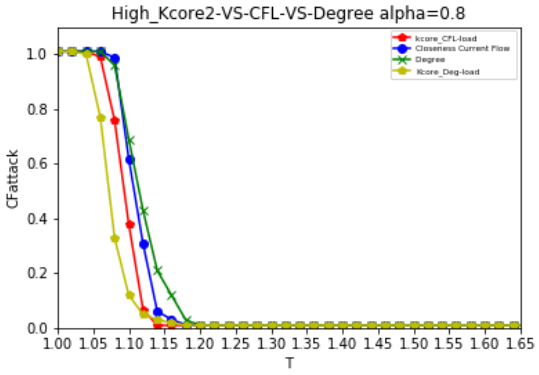
low centrality-load attack





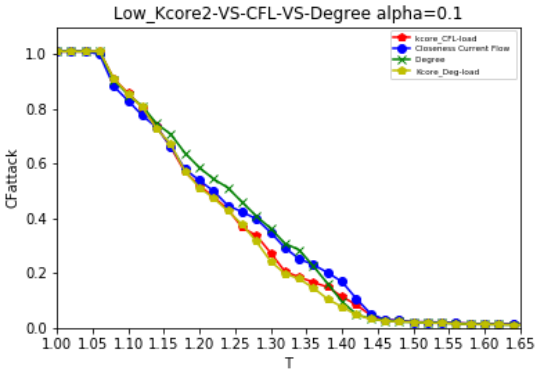
high centrality-load attack

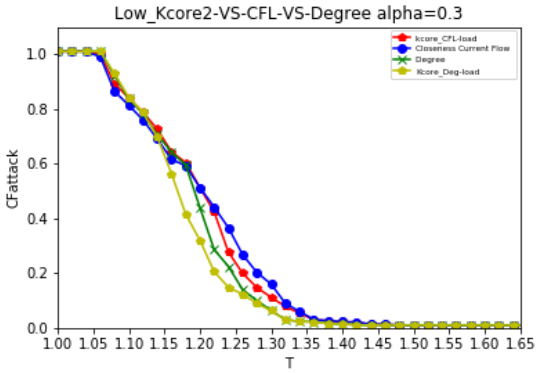




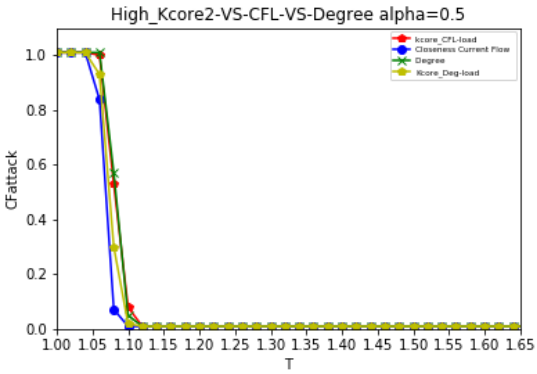
C-Sc1-3D geographical threshold

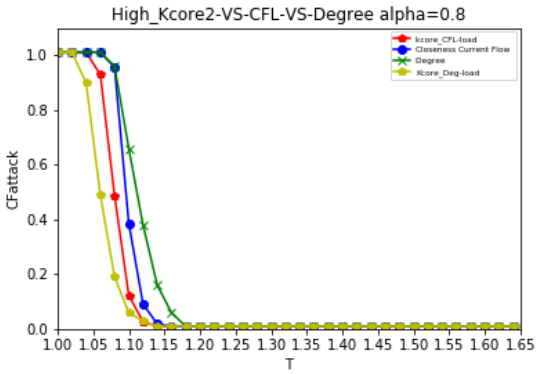
low centrality-load attack





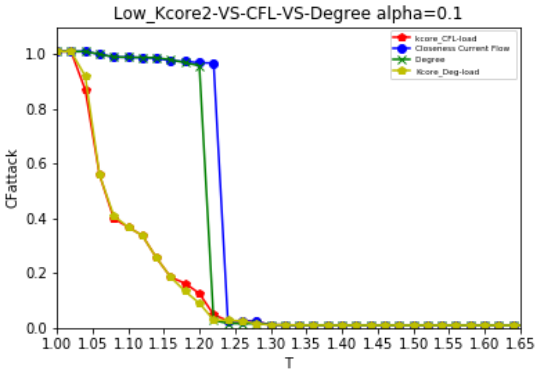
high centrality-load attack

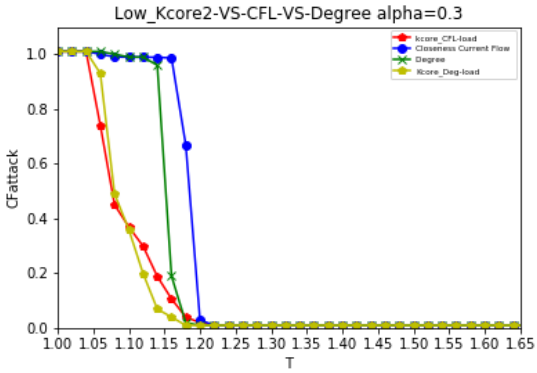




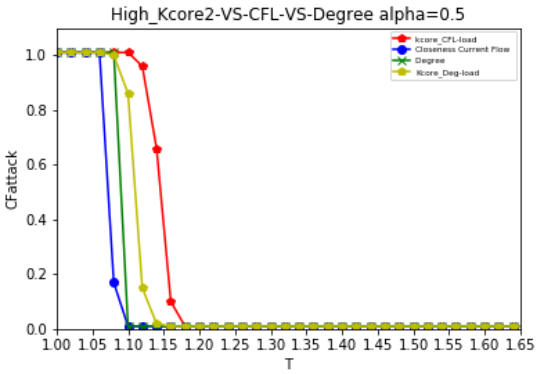
C-Sc1-Barabasi-Albert

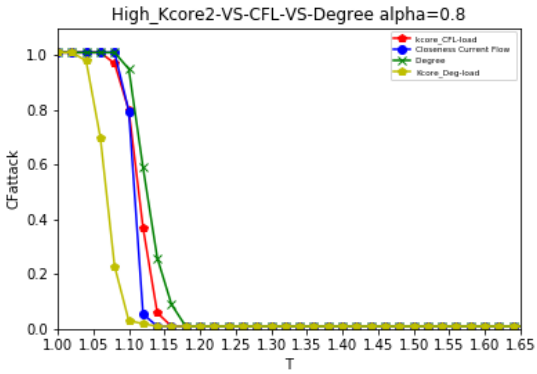
low centrality-load attack





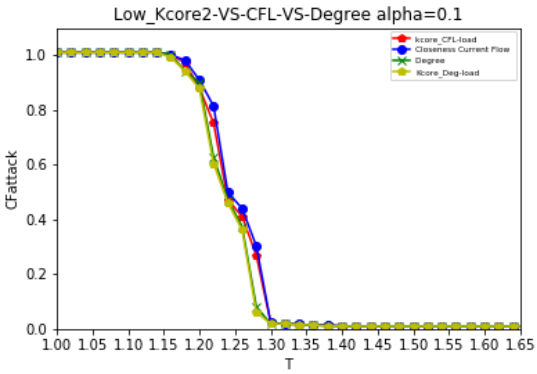
high centrality-load attack

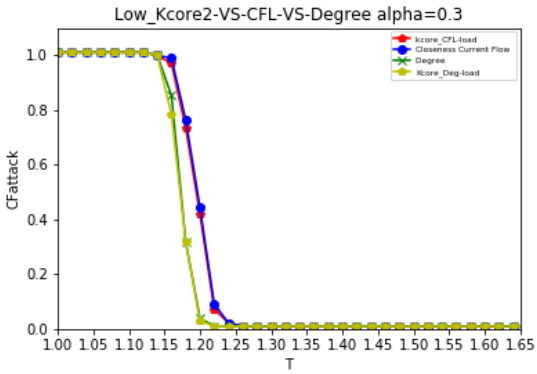




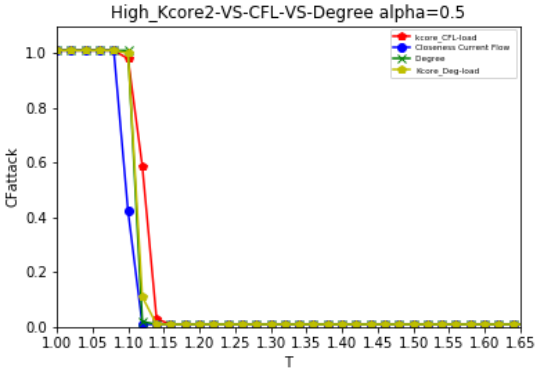
C-Sc1-Erdos-Renyi

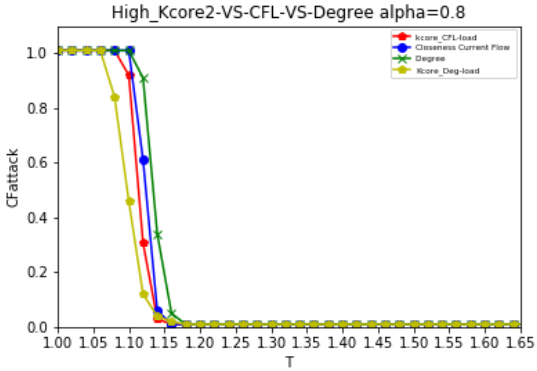
low centrality-load attack





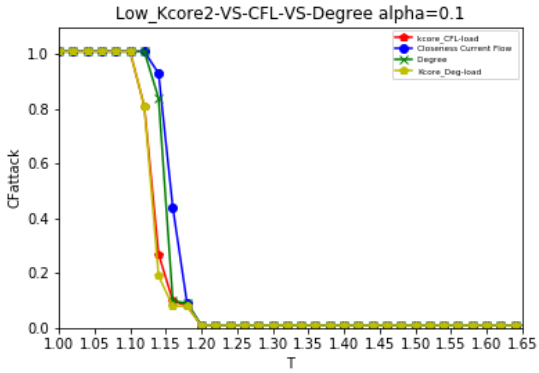
high centrality-load attack

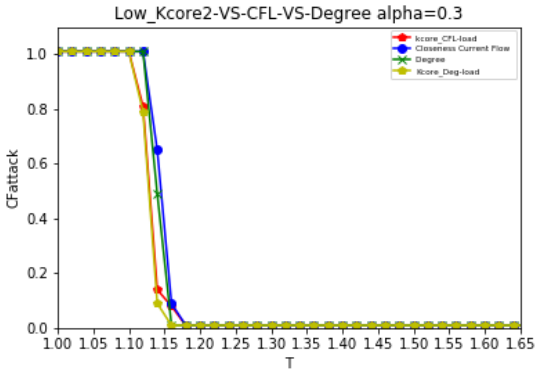




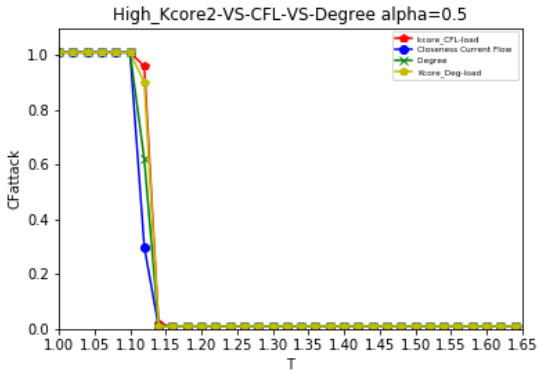
C-Sc1-Watts-Strogatz high clustering

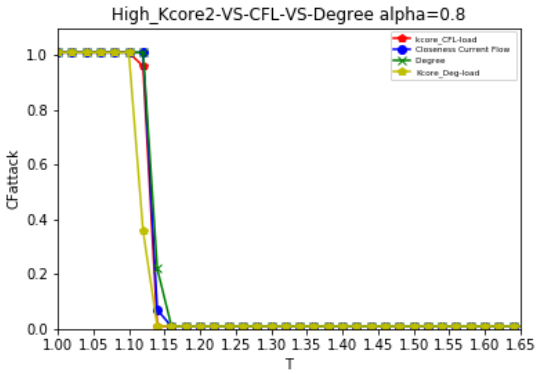
low centrality-load attack





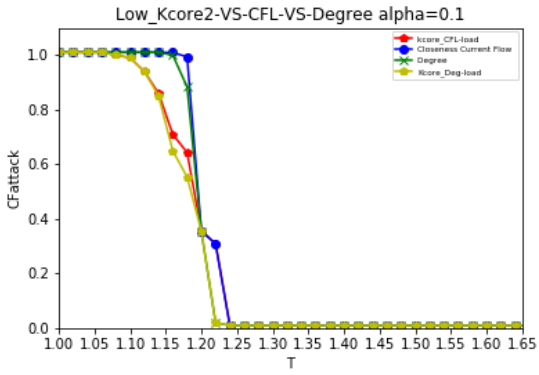
high centrality-load attack

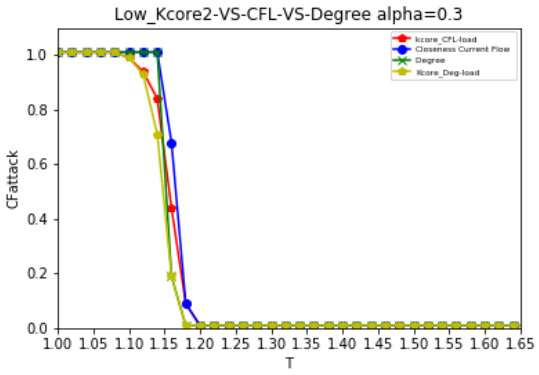




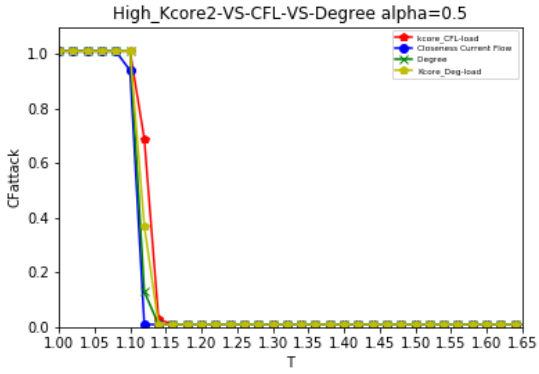
C-Sc1-Watts-Strogatz low clustering

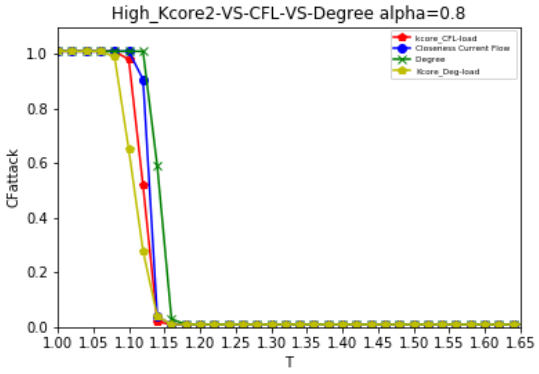
low centrality-load attack





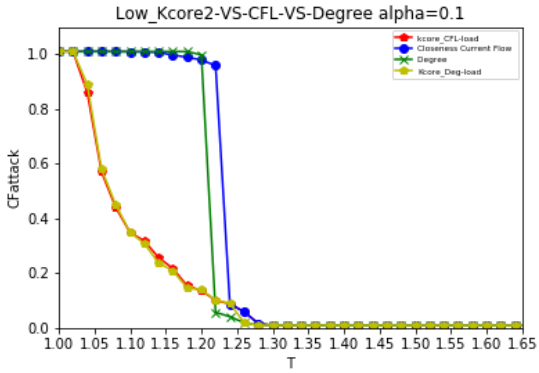
high centrality-load attack

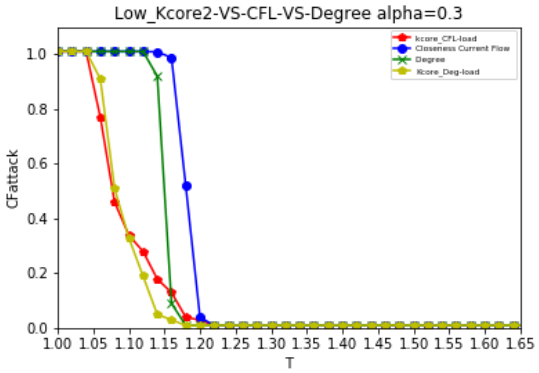




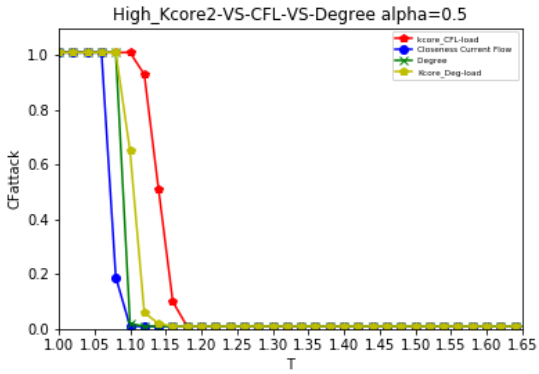
C-Sc1-Power-law cluster

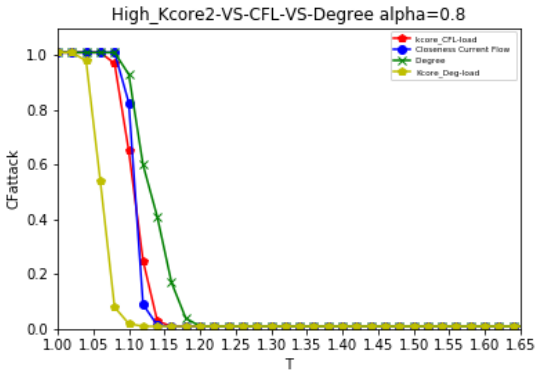
low centrality-load attack





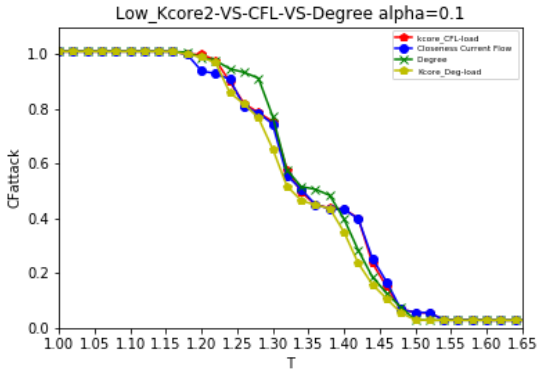
high centrality-load attack

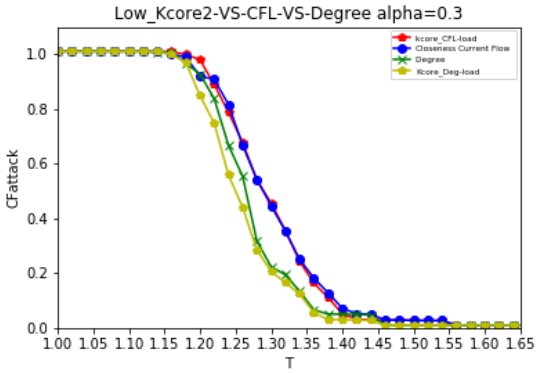




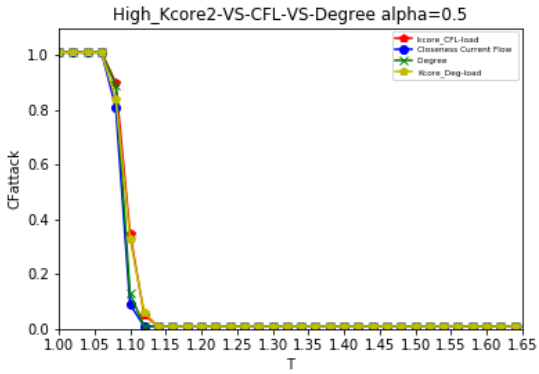
C-Sc1-Random geometric

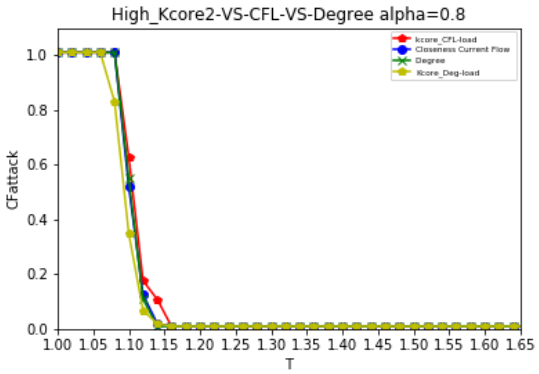
low centrality-load attack





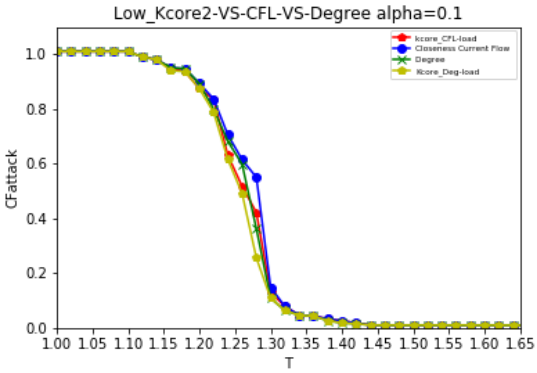
high centrality-load attack

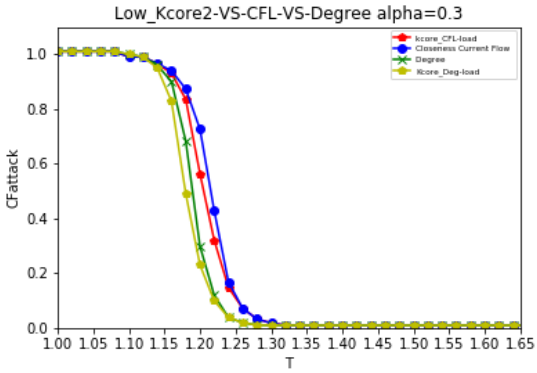




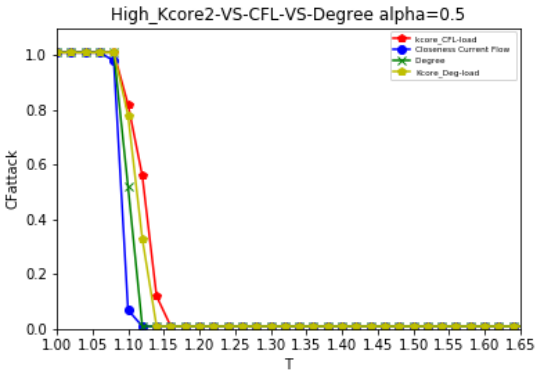
C-Sc1-Random partition

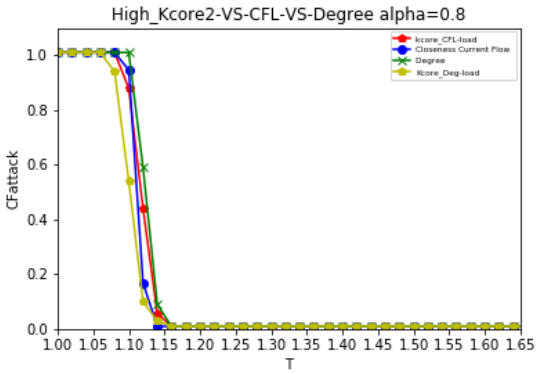
low centrality-load attack





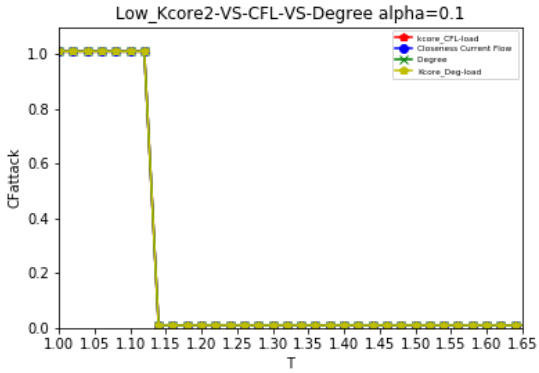
high centrality-load attack

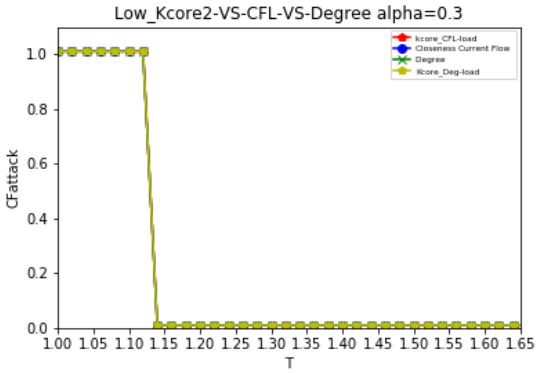




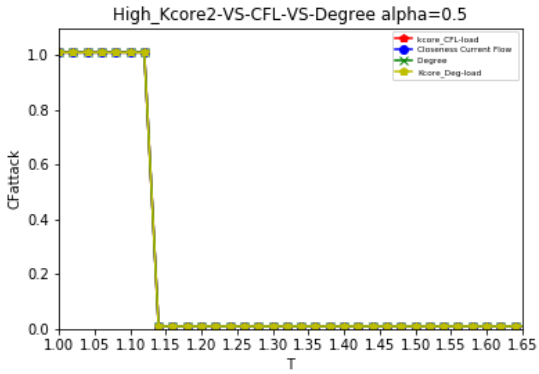
C-Sc1-Random d-regular

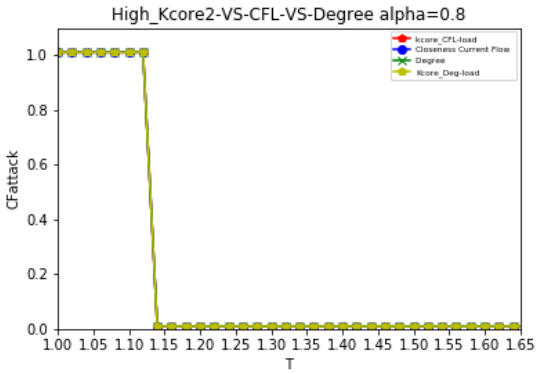
low centrality-load attack





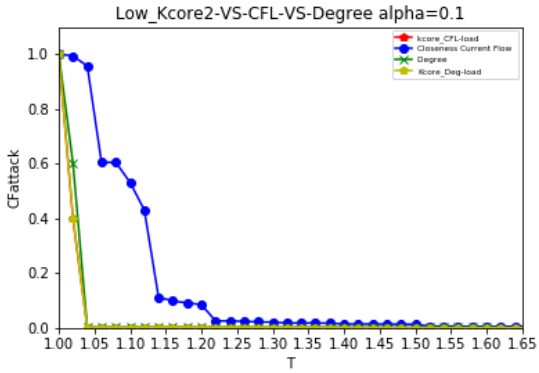
high centrality-load attack

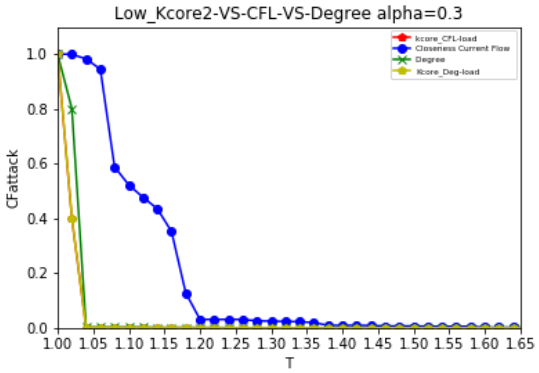




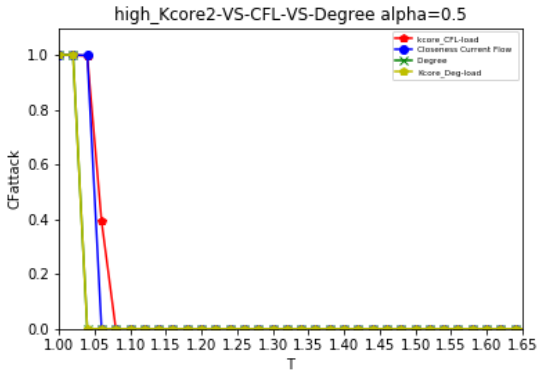
C-Sc1-North America airport network

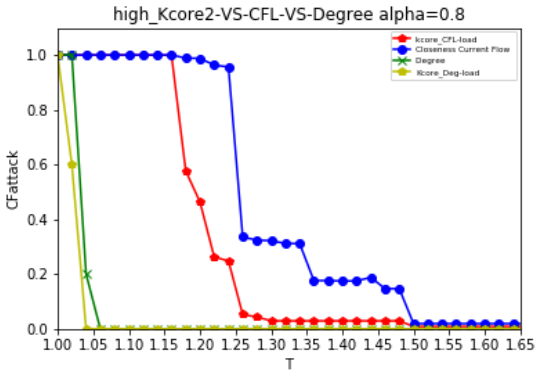
low centrality-load attack





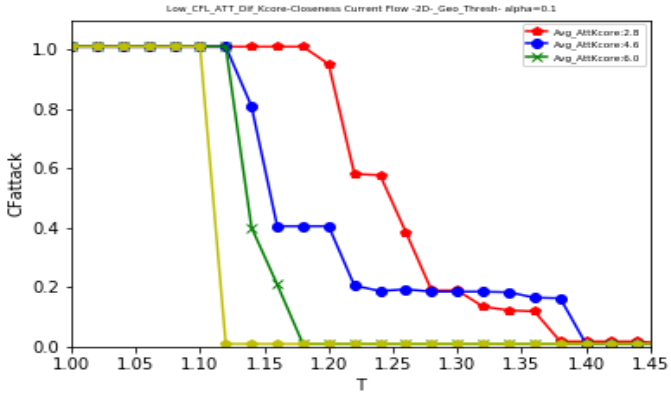
high centrality-load attack

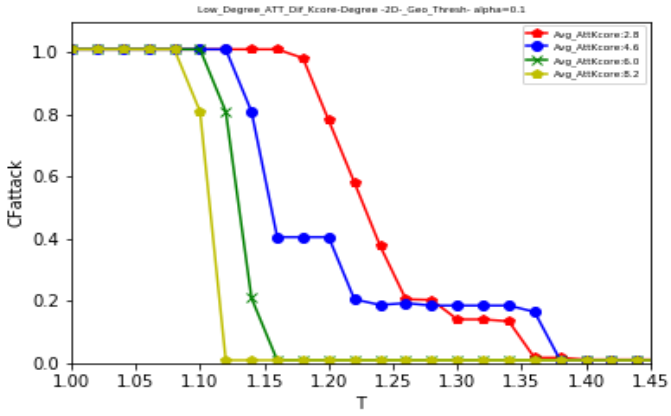


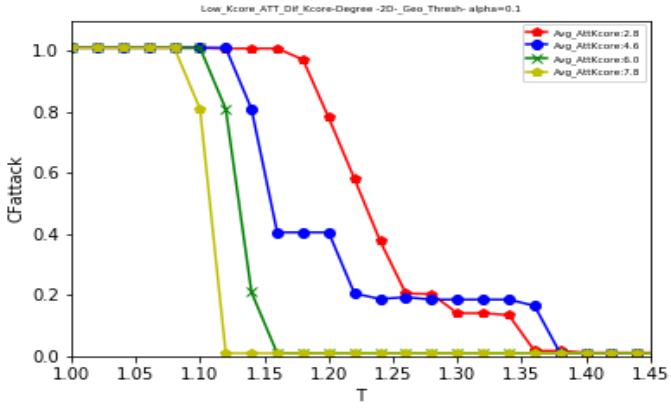


C-Scenario 2

C-Sc2-2D geographical threshold- table 3-1- results of low centrality-load attacks & low k-core attack

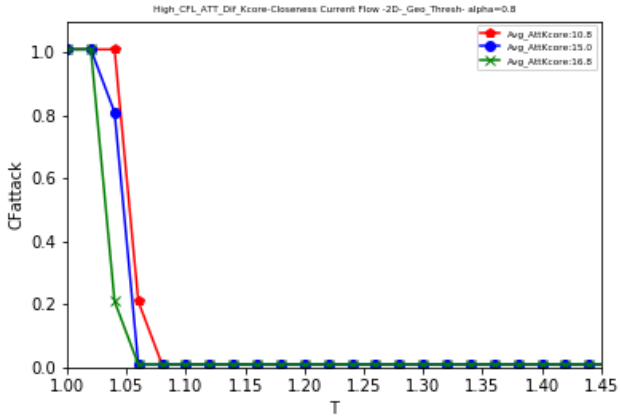


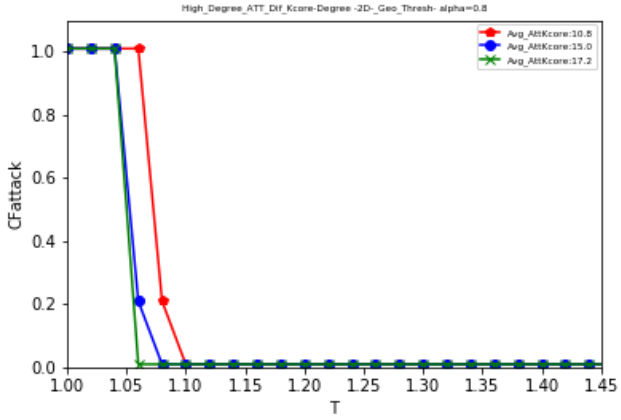


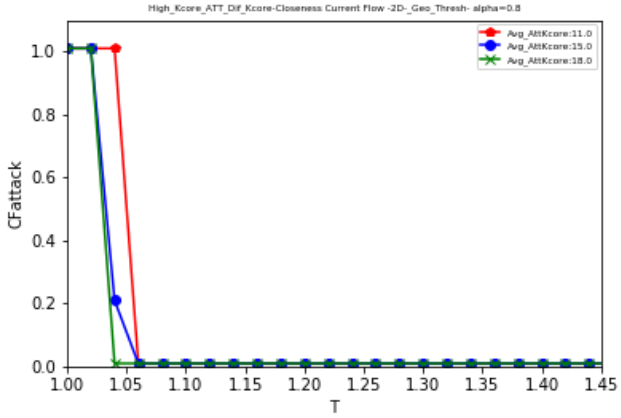


[3] The guideline beside each figure, illustrate the average *k-core* value of 5 nodes that have been chosen as attack targets.

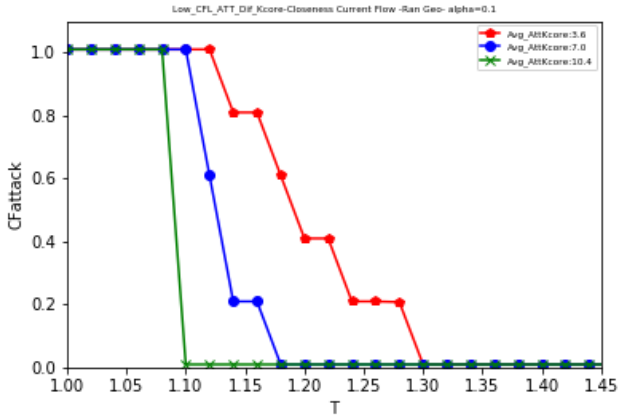
C-Sc2-2D geographical threshold- table 3-2 results of high centrality-load attacks & high k-core attack

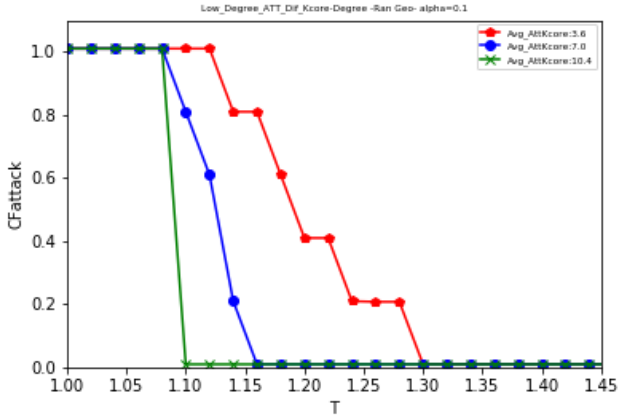


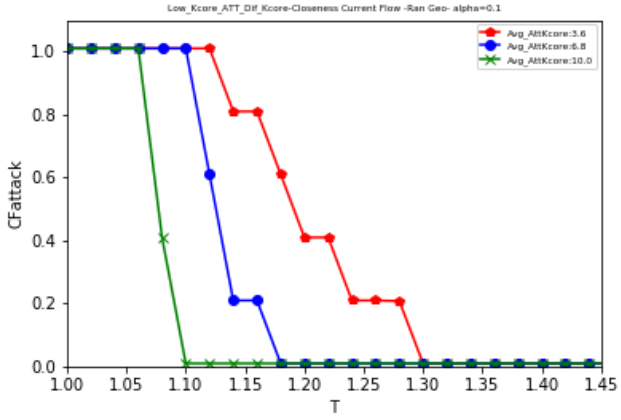




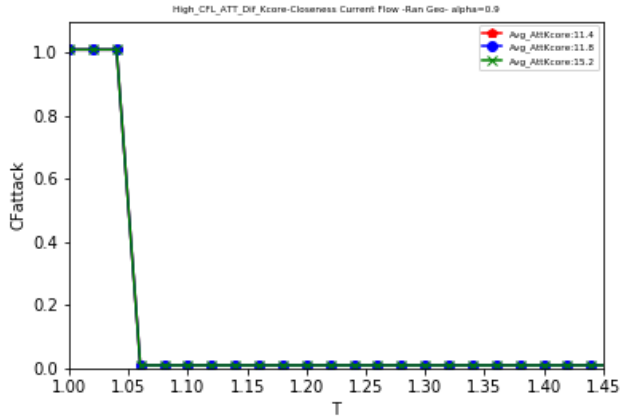
C-Sc2-Random geometric- table 3-3- results of low centrality-load attacks & low k-core attack

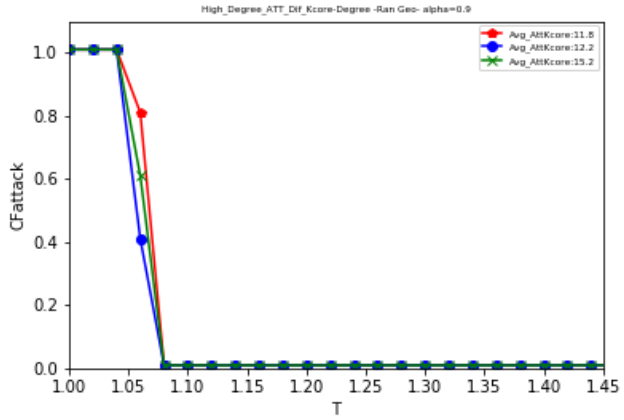


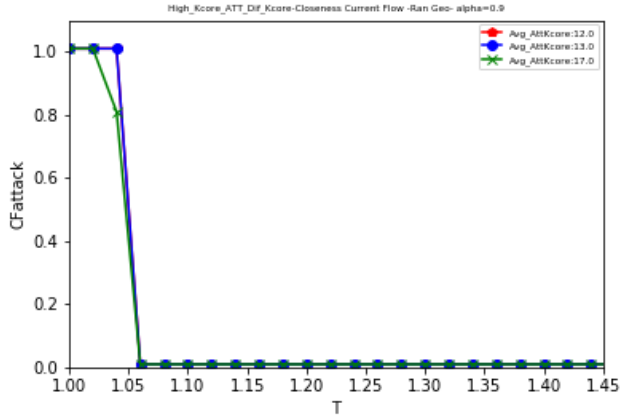




C-Sc2-Random geometric- table 3-4- results of high centrality-load attacks & high k-core attack



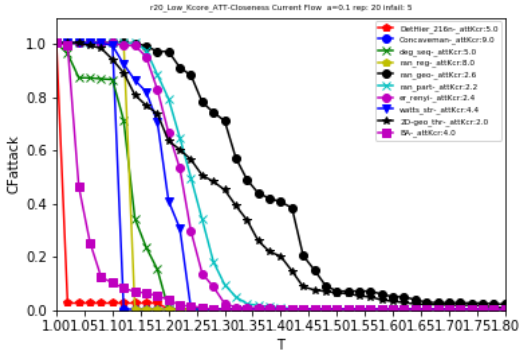




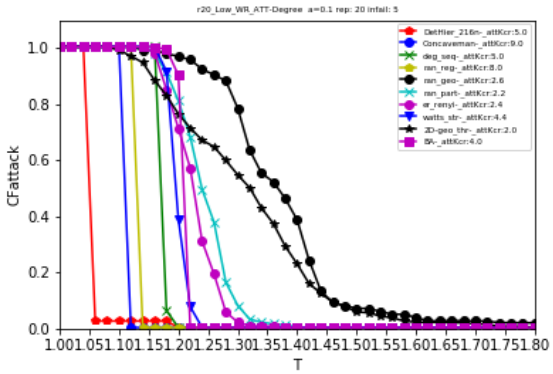
C-Scenario 3

Results of low centrality-load or k-core when alpha=0.1

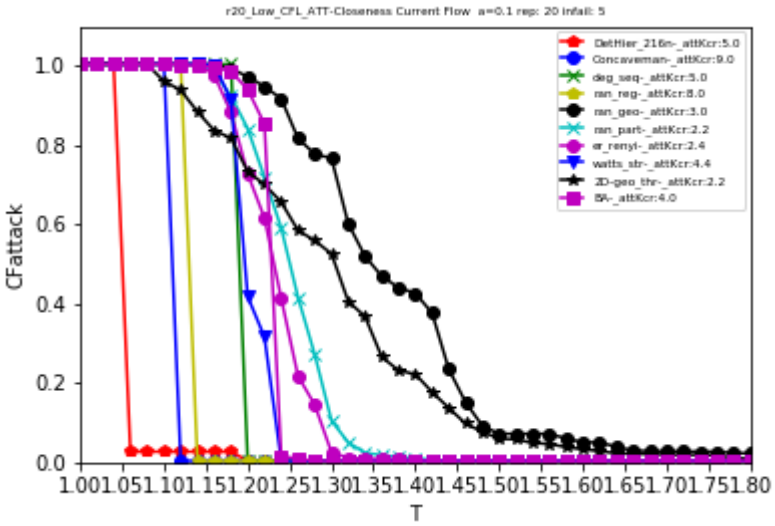
Low *k-core* attack



Low degree-load attack

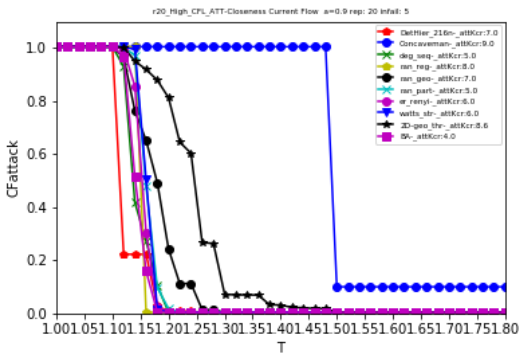


Low CFCC-load attack



Results of high centrality-load & k-core when alpha=0.9

High CFCC attack



High *k-core* attack