

Network robustness

Network Analysis - Assignment 3

Enrico Pezzano

June 2025

1 Introduction

In this assignment, I will analyze how the structure of a network affects its resilience to node failures or targeted attacks. I first implemented and compared four attack strategies (random failures, highest-degree removal, PageRank-based and betweenness-based removals) on small toy graphs (§3) and then on the same political-blogs network (§4) of the previous two assignments. After that, I tried to improve the robustness of each of the mentioned networks by adding edges in “rings” among the neighbours of the top- k highest-degree nodes (§5).

Additionally, since the `polblogs` network did not show interesting results, I searched for the community (subgraph) with the lowest theoretical percolation threshold f_c and repeated the experiments on that worst component (§6), trying to improve its robustness as well, but the results were not as expected.

2 Methodology

First of all, I implemented the attack strategies as 4 different functions, each of which takes a graph, a fraction of nodes to remove, and a number of steps that simply corresponds to the number of iterations to perform, removing the specified fraction of nodes at each step. Note that a higher number of steps means that the attack simulation will be more detailed, but the execution time will also increase. The four attack strategies are, as showed in the AulaWeb example, random failures, highest-degree removal, PageRank-based and betweenness-based removals.

After the attacks are executed, I tried to improve the robustness of the network by adding edges in “rings” among the neighbours of the top- k highest-degree nodes and also including a *randomized* version of the original graph with the same degree sequence, generated via double-edge swaps, in order to use it as a baseline for comparison. The following listing shows the python code for what I have just described.

```
1 def attack_random(G, rm_fraction=0.01, steps=100):  
2     nodes = list(G.nodes())
```

```

3      random.shuffle(nodes)
4      batch_size = max(1, int(rm_fraction * len(nodes)))
5      sizes, Gc = [], G.copy()
6      for i in range(steps):
7          rem = nodes[i*batch_size:(i+1)*batch_size]
8          if not rem:
9              break
10         Gc.remove_nodes_from(rem)
11         sizes.append(giant_component_size(Gc))
12     return sizes
13
14 def attack_highest_degree(G, rm_fraction=0.01, steps=100):
15     Gc, sizes = G.copy(), []
16     batch_size = max(1, int(rm_fraction * len(G)))
17     for _ in range(steps):
18         to_rm = [n for n, _ in sorted(Gc.degree(),
19                                     key=lambda x: x[1],
20                                     reverse=True)
21                 [:batch_size]]
22         if not to_rm: break
23         Gc.remove_nodes_from(to_rm)
24         sizes.append(giant_component_size(Gc))
25     return sizes
26
27 def attack_pagerank(G, rm_fraction=0.01, steps=100):
28     Gc, sizes = G.copy(), []
29     batch_size = max(1, int(rm_fraction * len(G)))
30     for _ in range(steps):
31         to_rm = [n for n, _ in sorted(nx.pagerank(Gc).items(),
32                                     key=lambda x: x[1],
33                                     reverse=True)[:batch_size]]
34         if not to_rm: break
35         Gc.remove_nodes_from(to_rm)
36         sizes.append(giant_component_size(Gc))
37     return sizes
38
39 def attack_betweenness(G, rm_fraction=0.01, steps=100):
40     Gc, sizes = G.copy(), []
41     batch_size = max(1, int(rm_fraction * len(G)))
42     for _ in range(steps):
43         to_rm = [n for n, _ in sorted(
44             nx.betweenness_centrality(Gc).items(),
45             key=lambda x: x[1],
46             reverse=True)[:batch_size]]
47         if not to_rm: break
48         Gc.remove_nodes_from(to_rm)
49         sizes.append(giant_component_size(Gc))
50     return sizes
51
52

```

```

53 results_real = {
54     'random':
55         attack_random(G, rm_fraction=0.01, steps=100),
56     'degree':
57         attack_highest_degree(G, rm_fraction=0.01, steps=100),
58     'pagerank':
59         attack_pagerank(G, rm_fraction=0.01, steps=100),
60     'betweenness':
61         attack_betweenness(G, rm_fraction=0.01, steps=100),
62 }

```

Listing 1: Attack strategies.

```

1 def add_ring_among_high_degree(G, top_k=3):
2     Gc = G.copy()
3     top_nodes = sorted(Gc.degree(),
4                         key=lambda x: x[1],
5                         reverse=True
6                        )[:top_k]
7     for n, _ in top_nodes:
8         neigh = list(Gc.neighbors(n))
9         if len(neigh) > 2:
10             for i in range(len(neigh)):
11                 u, v = neigh[i], neigh[(i+1) % len(neigh)]
12                 if not Gc.has_edge(u, v):
13                     Gc.add_edge(u, v)
14     return Gc
15
16 def randomize_graph(G, n_swaps=None):
17     Gc = G.copy()
18     if n_swaps is None:
19         n_swaps = Gc.number_of_edges() * 10
20     nx.double_edge_swap(Gc, n_swaps, max_tries=n_swaps * 10)
21     return Gc
22
23
24 G_rand = randomize_graph(G)
25 G_ring1 = add_ring_among_high_degree(G, top_k=3)
26 G_ring2 = add_ring_among_high_degree(G_ring1, top_k=3)
27 G_ring3 = add_ring_among_high_degree(G_ring2, top_k=3)

```

Listing 2: Edge augmentation.

Finally, I computed the theoretical critical threshold f_c for each graph, using the formula

$$f_c = 1 - \frac{1}{\frac{\langle k^2 \rangle}{\langle k \rangle} - 1}.$$

and the first and second moments of the degree distribution using the following python code.

```

1 degs = np.array([d for _, d in Gs.degree()])
2 k1, k2 = degs.mean(), (degs**2).mean()

```

Listing 3: First and second moments of the degree distribution.

I then printed the results in a table and plotted the giant component size during the attacks, as seen in the example in AulaWeb.

Additionally, I used the `greedy_modularity_communities` function from `NetworkX`, i.e. Clauset-Newman-Moore algorithm, to detect communities in the graph and identify the one with the lowest valid f_c (and at least 10 nodes), which I then used to repeat the robustness experiments on that worst component.

3 Toy Graph

I first tested my code on a small undirected graph of 100 nodes, using the `NetworkX` functions for synthetic networks, in particular, I choose the Barabási-Albert and the Erdős-Rényi models. The following figures 1 show the results of the various attacks.

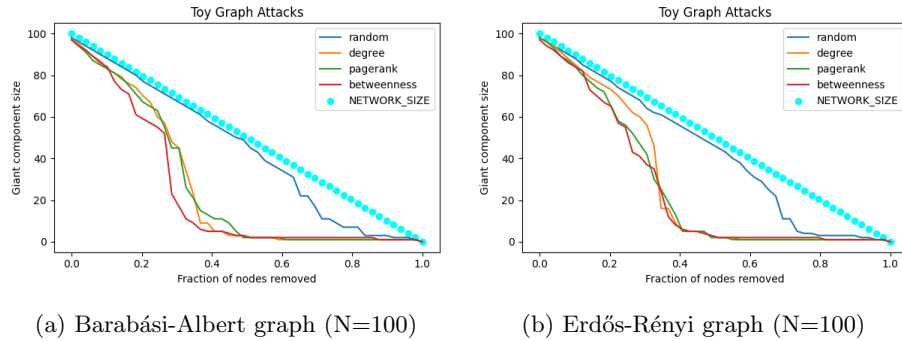


Figure 1: Attack strategies on toy graphs: giant component size vs. fraction of nodes removed.

The Barabási-Albert graph is slightly more robust than the Erdős-Rényi graph to degree and PageRank attacks, but both graphs are very fragile to betweenness attacks, which break the giant component very quickly. Random failures are the least destructive, as expected.

4 PolBlogs Graph

The successive part of the assignment was to apply the same attack strategies to the chosen network: in my case, I used the usual `polblogs` network, but this time I removed immediately the two isolated nodes, in order to convert it

into an undirected graph with $N = 1222$ nodes and $L = 16714$ edges, which of course is the largest connected component of the original directed graph.

Successively, I applied the four attack strategies to this network, as described in the previous section, and plotted the giant component size as a function of the fraction of nodes removed. Figure 2 shows the results of the attacks on the polblogs network, with its original robustness.

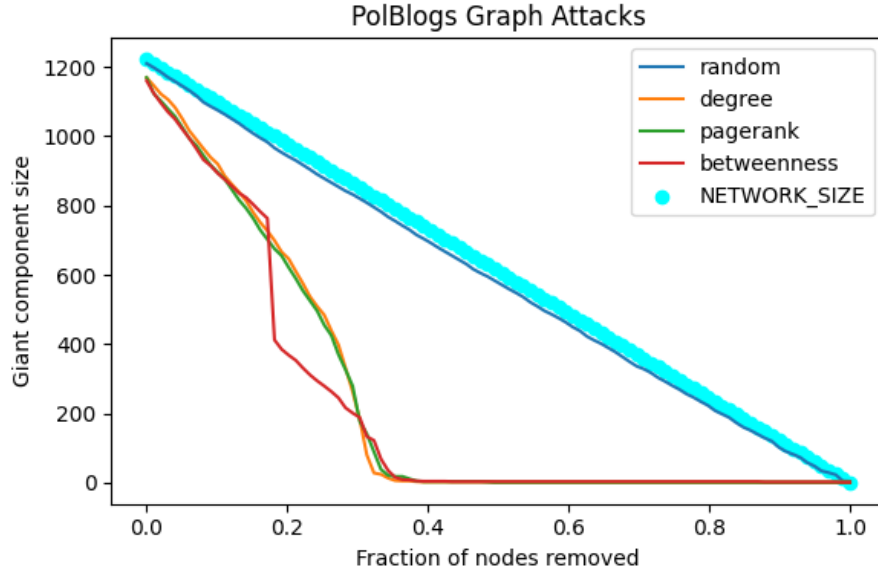


Figure 2: PolBlogs: giant-component size under different attack strategies.

The results are very similar with the toy experiments: random failures cause progressive degradation in the network performance, but with little to no effect on the giant component size, which remains almost constant until the end of the attack. On the other hand, attacks on the highest degree nodes, PageRank, and betweenness severely damage the graph structure, with the betweenness attack being the most harmful, but not by a big margin.

5 Building robustness

As requested by the assignment, I tried to improve the robustness of the chosen network by adding edges in “rings” among the neighbours of the top- k highest-degree nodes, and I hardcoded $k = 40, 80, 120$, since they were only 3 iterations. I wanted to replicate the example on AulaWeb, where there was 3 iterations of adding one ring at a time on a 35 nodes network; so, for my 1222 nodes network, I decided to go for the 10% of the nodes, which is 122 nodes, and I added 40 rings at a time, for a total of 3 iterations.

I used the `python` code mentioned in the methodology section 2 to add the said rings, and then computed the first and second moments of the degree distribution, as well as the critical threshold f_c before and after the edge additions.

The following table summarizes the results of the robustness improvement on the chosen network.

Table 1: PolBlogs robustness improvement.

Graph	Size (N,L)	$\langle k \rangle$	$\langle k^2 \rangle$	f_c
Original	N=1222, L=16714	27.36	2222.98	0.9875
Randomized	N=1222, L=16714	27.36	2222.98	0.9875
After 40 rings	N=1222, L=20174	33.02	2755.23	0.9879
After 80 rings	N=1222, L=21282	34.83	2959.87	0.9881
After 120 rings	N=1222, L=23713	38.81	3441.95	0.9886

We can see that both the mean degree $\langle k \rangle$ and the second moment $\langle k^2 \rangle$ increase after each ring addition, which is expected since we are adding edges. The critical threshold f_c also increases, but only slightly, which is not surprising since the `polblogs` network is already quite robust, with a high initial f_c value of 0.9875, due of its large variance in the degree distribution, which is a consequence of its hub-and-spoke structure, reflecting the underlying political blogosphere. Nonetheless, the edge additions do improve the robustness of the network, as we can see from the increasing f_c values after each iteration.

Resultant Plots The following figure 3 shows the results of the same 4 types of attacks mentioned before, but on the randomized version of the `polblogs` network, which has the same degree sequence as the original graph, but with edges randomly shuffled.

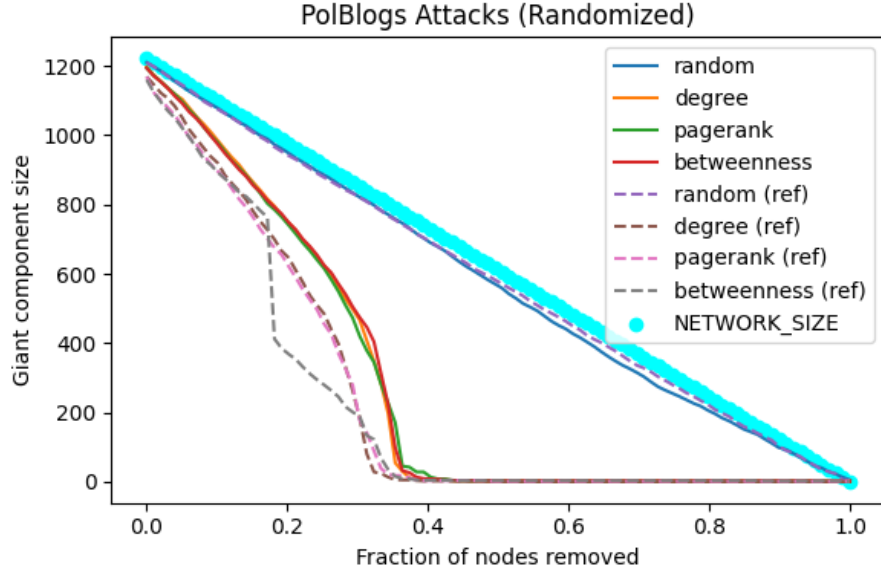


Figure 3: PolBlogs: giant-component size under different attack strategies on the randomized graph.

The randomized graph shows a similar behaviour to the original one, but with smoother curves, as expected. The giant component size decreases linearly with the fraction of nodes removed under the random failures attack, while the targeted attacks cause a more abrupt drop in the giant component size, after around 38% of nodes removed. This graph proved to be slightly more resilient to the betweenness attack, compared to the original `polblogs` graph.

The following figures, instead 4, 5 and 6, show the results of the same attacks on the graph after each ring addition, i.e. after adding 40, 80 and 120 rings, respectively.

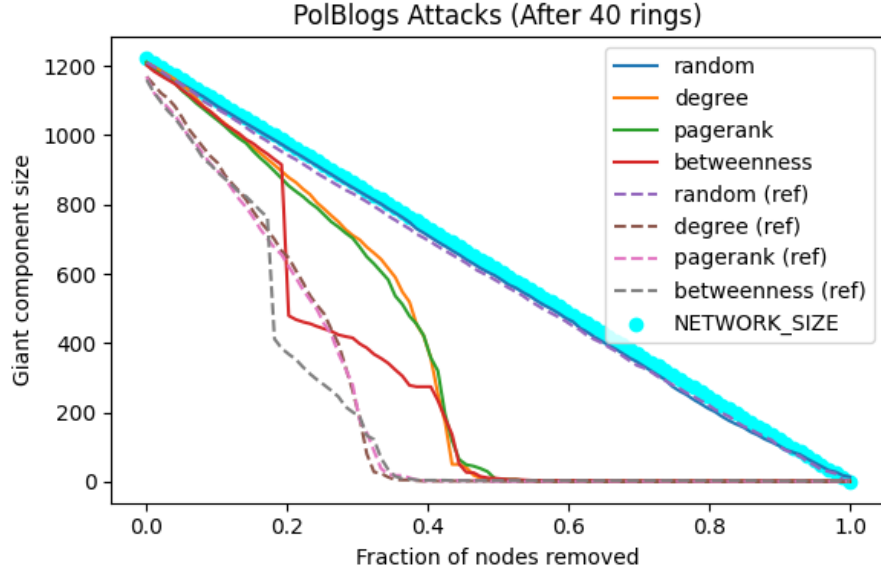


Figure 4: PolBlogs: giant-component size under different attack strategies after adding 40 rings.

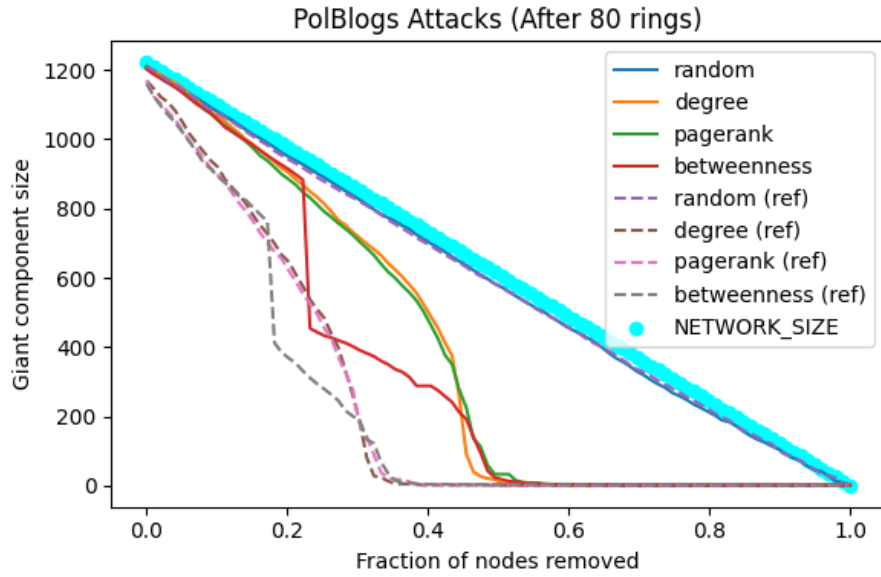


Figure 5: PolBlogs: giant-component size under different attack strategies after adding 80 rings.

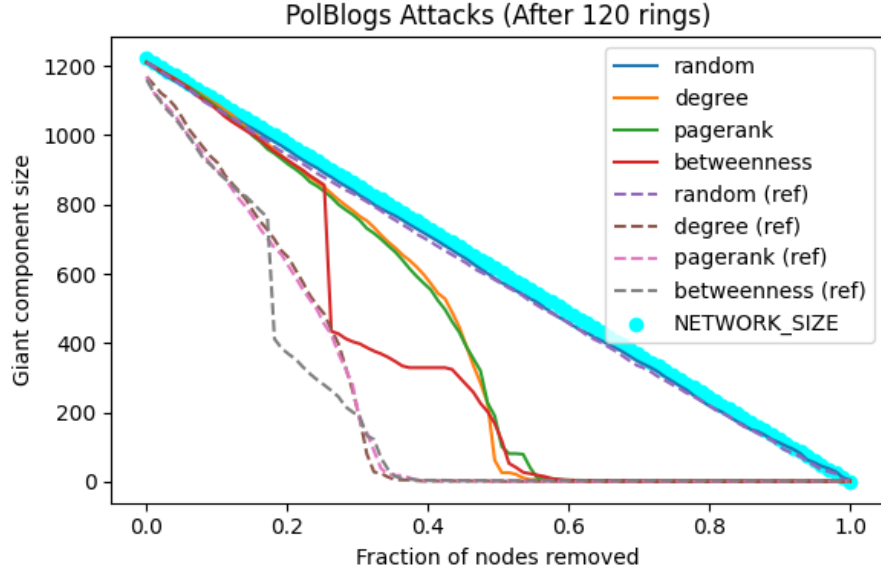


Figure 6: PolBlogs: giant-component size under different attack strategies after adding 120 rings.

Sadly, due to the critical threshold f_c increasing only slightly, the robustness improvement between the modified graphs is difficult to observe by the naked eye. Nonetheless, we can compute f_c for each of the graphs after the ring additions, and have mathematical proof of said improvement, as shown in Table 1.

Instead, we can observe the differences between the original plot of the (polblogs) network under attack, versus the last improved graph with 120 rings added and $f_c = 0.9886$ versus $f_c = 0.9875$. In particular, the random failures attack is slightly less destructive, the degree attack and the PageRank attack make the giant component size drop more gradually and around 50% of nodes removed instead of the original 38%, and the betweenness attack is still the most destructive, but this time with a less abrupt drop in the giant component size, falling around 25% of nodes removed instead of the original 18%, completely falling at around 57% of nodes removed instead of the original 38%.

6 Community with the worst critical threshold

Since the polblogs network did not show interesting results, only a f_c improvement of 0.0011 was observed after adding 120 rings on 1222 nodes, I thought that it would be interesting to search for the community with the lowest critical threshold f_c and repeat the robustness experiments on that worst component, in order to see if the edge additions would have a more significant effect on

the robustness of the network, even though it does not make any sense from a theoretical point of view, since the critical threshold f_c is defined for networks with $N \rightarrow \infty$, but still, I wanted to try it.

I used the NetworkX function `greedy_modularity_communities`, i.e. the Clauset Newman Moore algorithm, to detect communities with at least 10 nodes. The following listing 4 shows what already described, and the Table 2 lists the detected communities and their relative f_c values.

```

1  comms = community.greedy_modularity_communities(G_real)
2  print(f"\nDetected_{len(comms)}_communities")
3  for i, nodes in enumerate(comms):
4      sub = G_real.subgraph(nodes)
5      degs = np.array([d for _, d in sub.degree()])
6      k1, k2 = degs.mean(), (degs**2).mean()
7      denom = k2/k1 - 1 if k1>0 else -1
8      fc = (1 - 1/denom) if denom>0 else float('nan')
9      print(f"\tCommunity_{i}:_N={sub.number_of_nodes()},
10  _L={sub.number_of_edges()},
11  _fc={fc:.4f}")
12
13  MIN_SIZE = 10
14  cands = []
15  for i, nodes in enumerate(comms):
16      if len(nodes) < MIN_SIZE: continue
17      sub = G_real.subgraph(nodes).copy()
18      degs = np.array([d for _, d in sub.degree()])
19      k1, k2 = degs.mean(), (degs**2).mean()
20      denom = k2/k1 - 1 if k1>0 else -1
21      if denom<=0: continue
22      fc = 1 - 1/denom
23      cands.append((f"Community_{i}", sub, fc))
24
25  if not cands:
26      print("No_valid_community_>=MIN_SIZE_with_fc>0.")
27      exit()
28
29  worst_name, worst_subG, worst_fc = min(
30      cands,
31      key=lambda x: x[2]
32  )
33  Nw, Lw = worst_subG.number_of_nodes(),
34      worst_subG.number_of_edges()

```

Listing 4: Finding the worst community.

Table 2: Detected communities and their f_c .

Community	Size (N,L)	f_c
0	634, 8163	0.9858
1	544, 7226	0.9868
2	23, 22	0.8736
3	5, 5	0.2857
4	4, 3	0.0000
5	3, 2	-1.0000
6	3, 2	-1.0000
7	2, 1	nan
8	2, 1	nan
9	2, 1	nan

The community with the lowest f_c is Community 2, which has only 23 nodes and 22 edges, with a critical threshold of 0.8736, lower than the other communities, and lower than the original `polblogs` network. Additionally, the chosen community is one out of three with at least 10 nodes, which is the minimum size that I considered valid for a community, since smaller communities are not very interesting for this kind of analysis. Then, I computed the first and second moments of the degree distribution for this community, and I obtained $\langle k \rangle = 1.91$ and $\langle k^2 \rangle = 17.04$, which resulted in the critical threshold $f_c = 0.8736$.

Successively, I repeated the four attack simulations on this worst subgraph (Community 2), and the results are summarized in the following Figure 7.

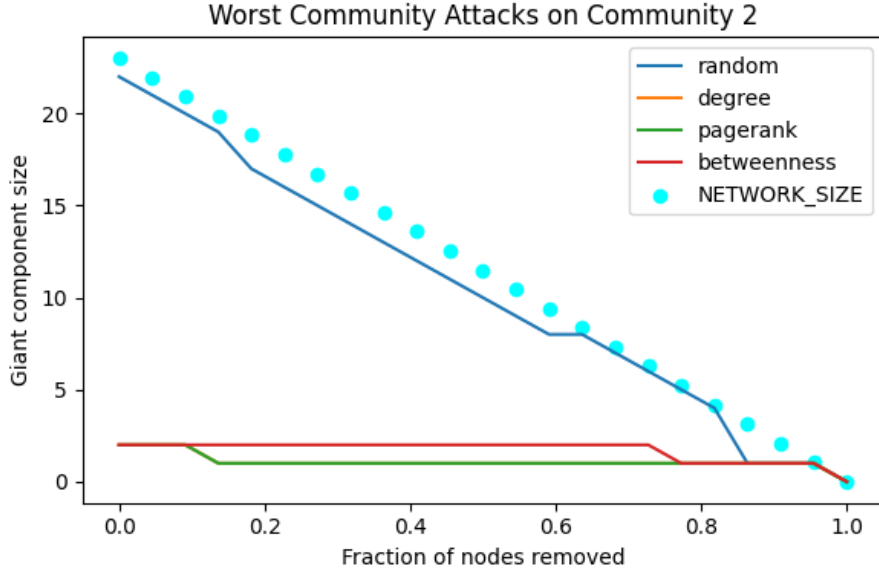


Figure 7: Community 2: giant-component size under different attack strategies.

The resultant graph show, this time, shows a very different behaviour compared to the `polblogs` network: the random failures attack causes a very quick degradation of the giant component size, which drops to zero after removing only 10% of the nodes, while the targeted attacks are still much more destructive. This shows that even if the critical threshold f_c is not very low, the community is very fragile to failures, which is a common characteristic of small communities.

As done for the `polblogs` network, I tried to improve the robustness of this community by adding edges in “rings” among the neighbours of the top- k highest-degree nodes, and I hardcoded $k = 3$, since this time the community is very small, with only $N = 23$ nodes, so I decided to add one ring at a time, for a total of 3 iterations, for the top 3 highest-degree nodes. The following Table 3 summarizes the results.

Table 3: Robustness improvement on the worst community.

Graph	Size (N,L)	$\langle k \rangle$	$\langle k^2 \rangle$	f_c
Original	N=23, L=22	1.91	17.04	0.8736
Randomized	N=23, L=22	1.91	17.04	0.8736
After 1 ring	N=23, L=41	3.57	24.17	0.827
After 2 rings	N=23, L=44	3.83	27.39	0.8376
After 3 rings	N=23, L=49	4.26	32.35	0.8483

The results show that the mean degree $\langle k \rangle$ and the second moment $\langle k^2 \rangle$ in-

crease after each ring addition, as expected, but the critical threshold f_c actually decreases after the first ring addition, which is not what I expected.

This behavior can be explained by the fact that adding edges in a small community like this one tends to make the degree distribution more homogeneous, thus reducing the ratio $\langle k^2 \rangle / \langle k \rangle$ and consequently lowering f_c . This is in line with the findings from the percolation theory discussed in the Appendix 7. The following Figure 8, 9 and 10 show the results of the attacks on the community after each ring addition, i.e. after adding 1, 2 and 3 rings, respectively.

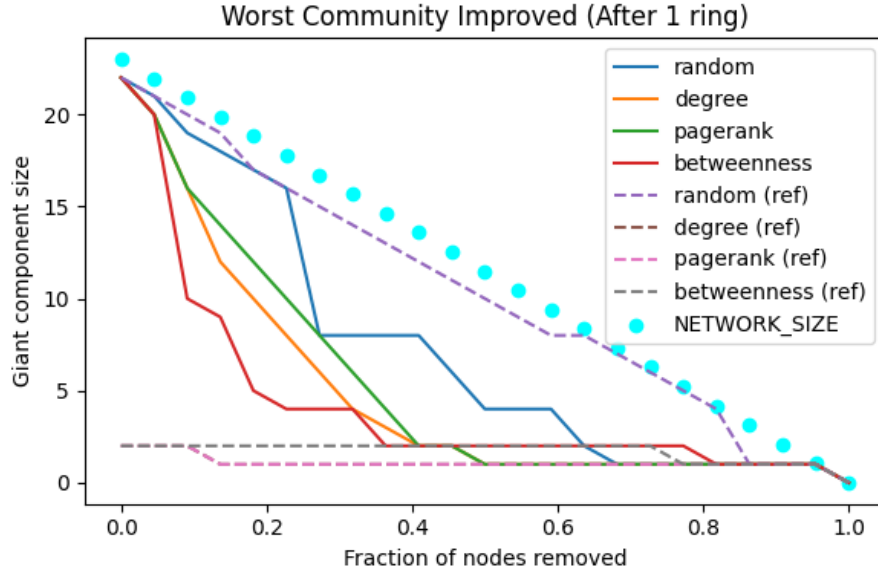


Figure 8: Community 2: giant-component size under different attack strategies after adding 1 ring.

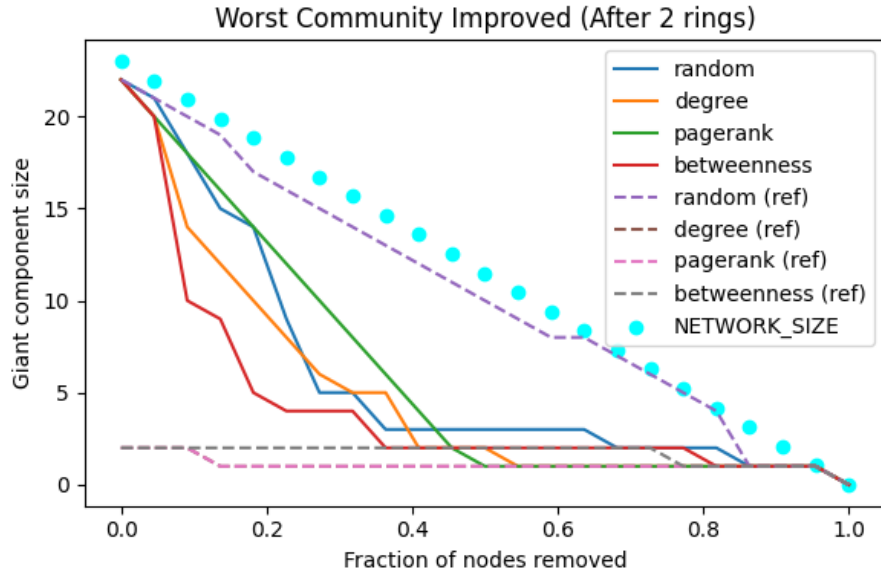


Figure 9: Community 2: giant-component size under different attack strategies after adding 2 rings.

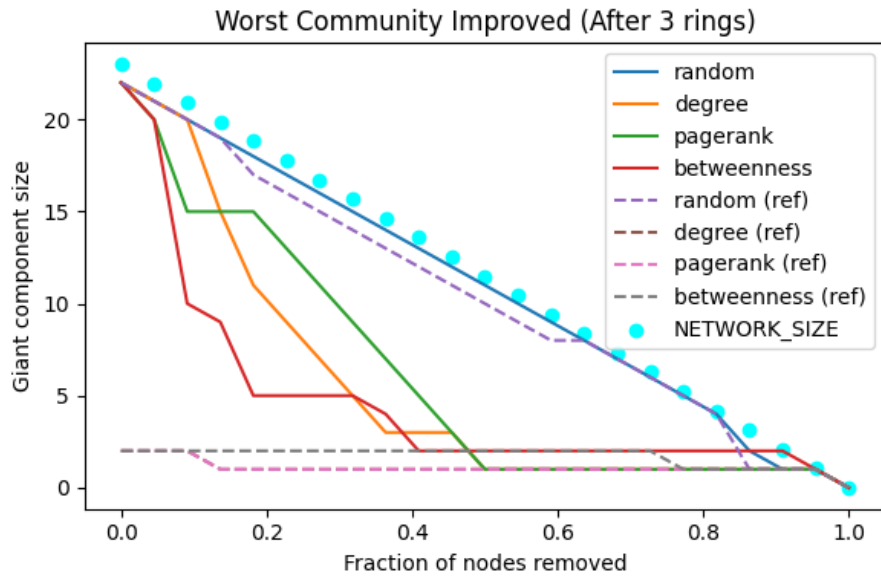


Figure 10: Community 2: giant-component size under different attack strategies after adding 3 rings.

The final resultant plots show a very different community compared to the original **Community 2** graph: a more robust network, each time with more rings. In particular, the random failures attack is still the least destructive, followed by the remaining three targeted attacks.

It is the opposite of what I expected from the Table 3, where the critical threshold f_c lowers after the first ring addition.

The contrast between the various expectations and the actual results can be explained by the fact that the **Molloy-Reed** formula behind the critical threshold f_c is based on the assumption that N is in a very large limit, thus ignoring small or finite networks side-effects, and loops or clustering effects, which are more pronounced in small networks like **Community 2**.

7 Conclusions

In this assignment, I have explored the robustness of networks against node failures and targeted attacks, focusing on the effects of different attack strategies and edge additions to improve resilience. I implemented four attack strategies (random failures, highest-degree removal, PageRank-based and betweenness-based removals) on both toy and real networks, specifically the **polblogs** network, and analyzed the giant component size as a function of the fraction of nodes removed. I also attempted to improve the robustness of the networks by adding edges in “rings” among the neighbours of the top- k highest-degree nodes, and I observed that this strategy can improve the robustness of the network, but only slightly, especially for the **polblogs** network.

In the end, I observed that targeted removals are far more destructive than random failures, and that the betweenness attack is the most harmful, followed by the highest-degree and PageRank attacks.

Additionally, I tried to isolate the weakest subgraph (**Community 2**) of the **polblogs** network, which had the lowest theoretical critical threshold f_c , and repeated the robustness experiments on that worst component, finding that the edge additions did not improve the robustness as expected, but rather decreased the critical threshold f_c after the first ring addition, due to the small size of the community and the resulting homogeneity in the degree distribution.

A Why f_c can decrease under ring addition

The critical threshold f_c is defined as the fraction of nodes that can be removed before the giant component disappears, and it is given by the formula

$$f_c = 1 - \frac{1}{\frac{\langle k^2 \rangle}{\langle k \rangle} - 1}.$$


Although ring additions raise both $\langle k \rangle$ and $\langle k^2 \rangle$, they can *lower* the ratio $\langle k^2 \rangle / \langle k \rangle$ if they preferentially boost mid-degree nodes, thus decreasing f_c . In-

deed, an alternative “spoke” strategy linking high-degree to low-degree nodes can increase variance more, pushing f_c upward.

This happens because the ring addition strategy tends to make the degree distribution more homogeneous, i.e. ring-adding among neighbors of top-degree nodes tends to boost the degrees of mid or low-degree nodes more than it boosts the variance of the distribution, thus reducing the ratio $\langle k^2 \rangle / \langle k \rangle$ and consequently, the degree distribution becomes more “flat”, lowering f_c .

B Standard Output

The following is the standard output of the code used in this assignment, which includes the results of the attack simulations, the robustness improvement after edge additions, and the critical threshold f_c for each graph.

 Standard Output