Complex Networks

Szymon Kowalczyk

February 23, 2023

1 Short info

The theoretical results for the parameters based on an average degree $< k >$ for the graphs are equal to:

- Erdős–Rényi-Gilbert: $p = \frac{<k>}{(N-1)}$,
- Barabasi-Albert: $m = \frac{<k>}{2}$,
- Watts and Strogatz: $k =< k >$.

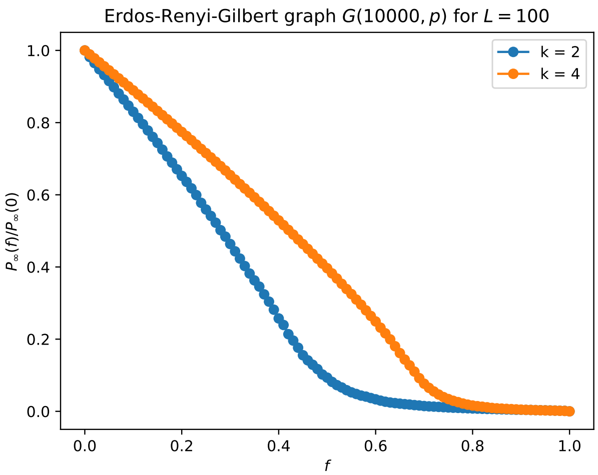
To improve performance for computing closeness centrality leveraging the adjacency matrix and calculation of the shortest paths using the Floyd-Warshall method was implemented.

Algorithm 1 Floyd-Warshall algorithm

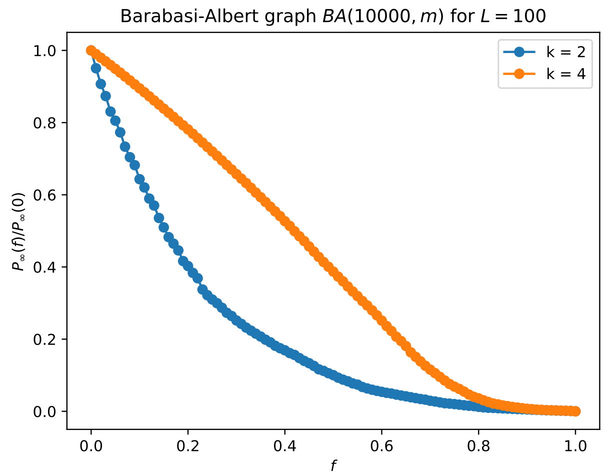
Input: Initial graph G with with n number of vertices
Output: Shortest distance matrix A

$A \leftarrow [0]_{n \times n}$
for $k \leftarrow 1$ to n **do**
  $A \leftarrow \left(a_{ij}^{(k)}\right)$
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      $a_{ij}^{(k)} \leftarrow \min\left(a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\right)$
    end for
  end for
end for
return A

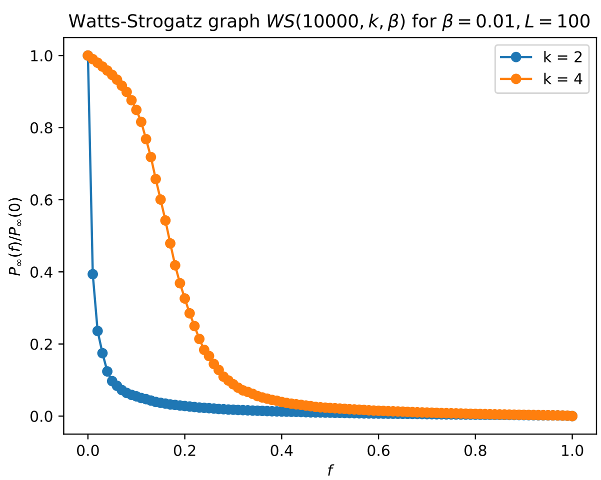
2 Presentation of results



(a) Erdős–Rényi-Gilbert graph $G(N, p)$

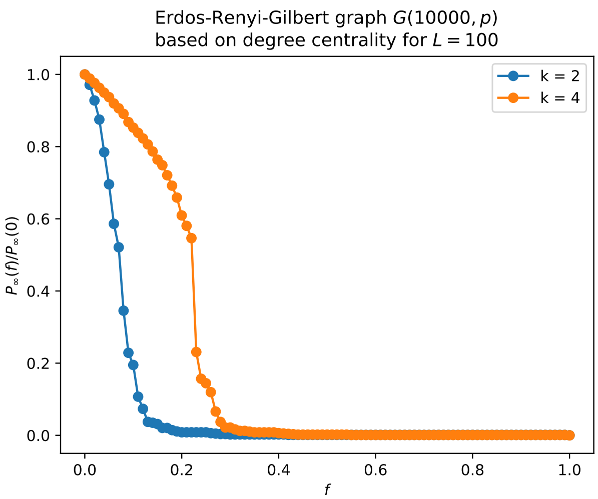


(b) Barabasi-Albert graph $BA(N, m)$

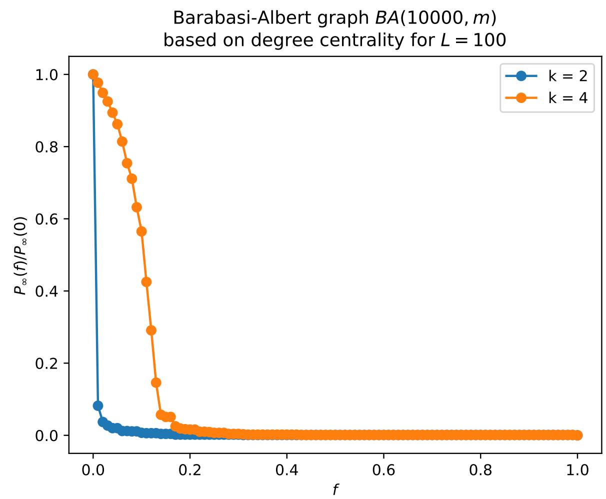


(c) Watts and Strogatz graph $WS(N, k, \beta)$

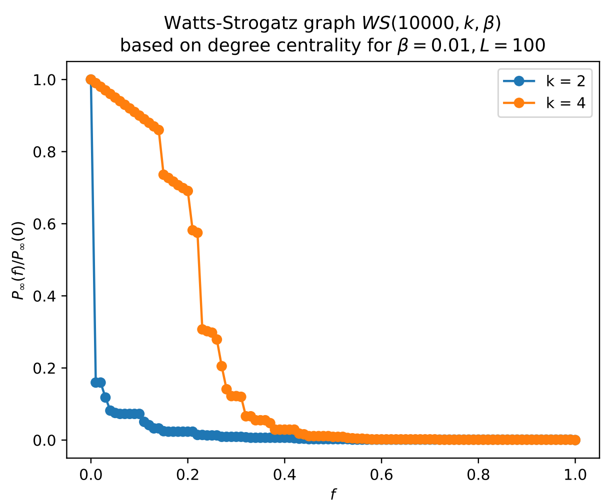
Figure 1: Presentation of the robustness analysis based on random attacks for the giant component $P_\infty(f)/P_\infty(0)$ as a function of f for several networks of size $N = 10^4$ using an ensemble average over $L = 100$ samples.



(a) Erdős–Rényi-Gilbert graph $G(N, p)$

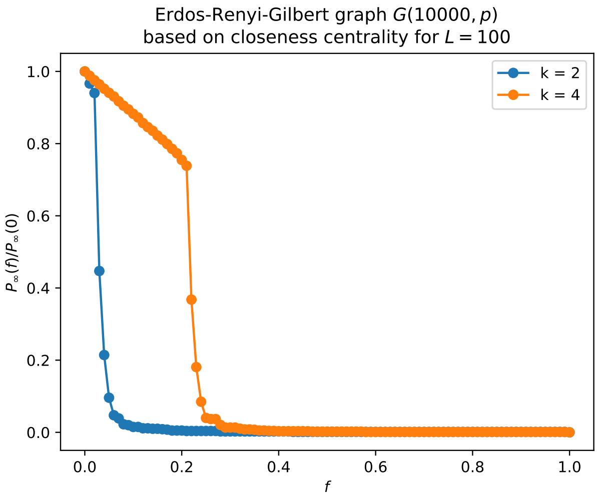


(b) Barabasi-Albert graph $BA(N, m)$

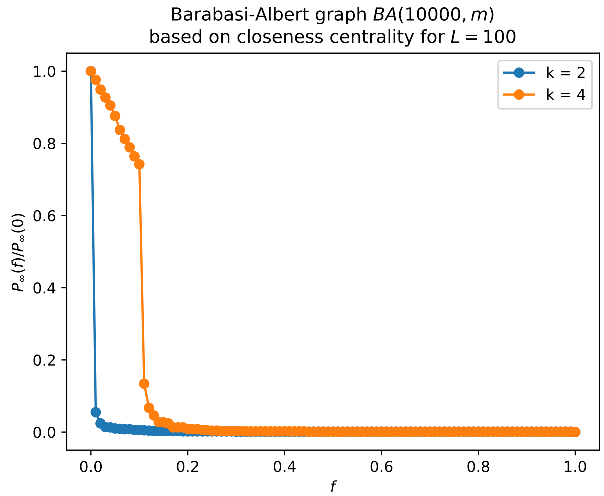


(c) Watts and Strogatz graph $WS(N, k, \beta)$

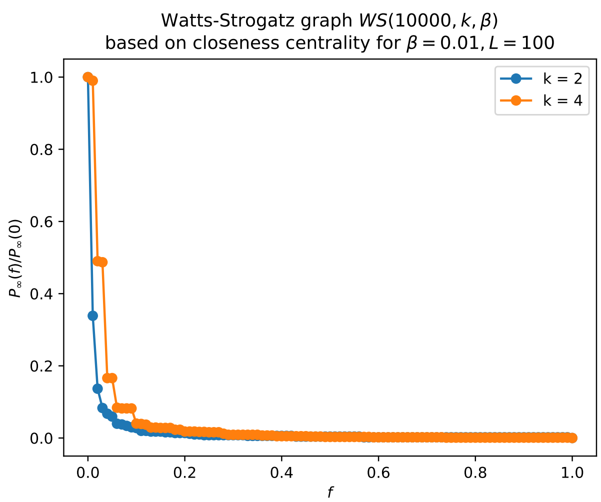
Figure 2: Presentation of the robustness analysis based on degree centrality for the giant component $P_\infty(f)/P_\infty(0)$ as a function of f for several networks of size $N = 10^4$ using an ensemble average over $L = 100$ samples.



(a) Erdős–Rényi-Gilbert graph $G(N, p)$

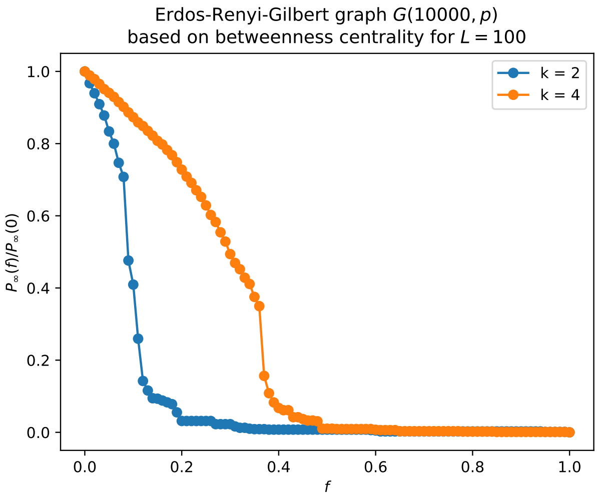


(b) Barabasi-Albert graph $BA(N, m)$

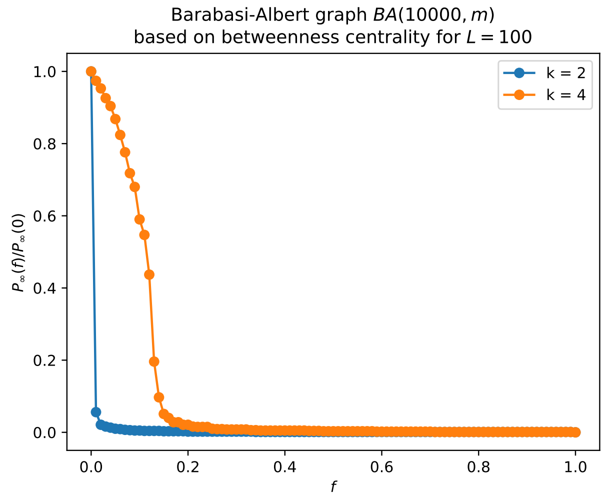


(c) Watts and Strogatz graph $WS(N, k, \beta)$

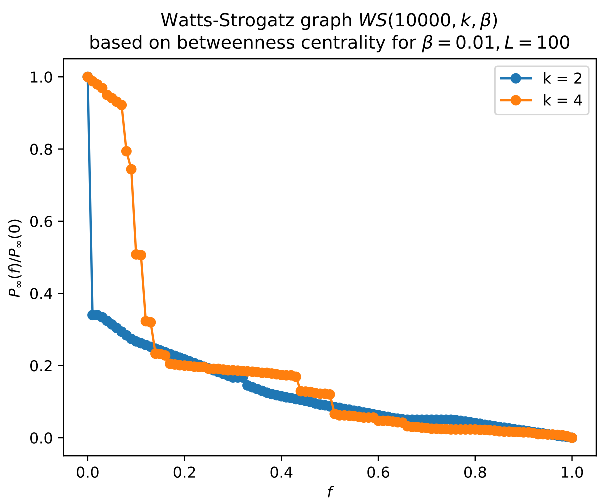
Figure 3: Presentation of the robustness analysis based on closeness centrality for the giant component $P_\infty(f)/P_\infty(0)$ as a function of f for several networks of size $N = 10^4$ using an ensemble average over $L = 100$ samples.



(a) Erdős–Rényi-Gilbert graph $G(N, p)$

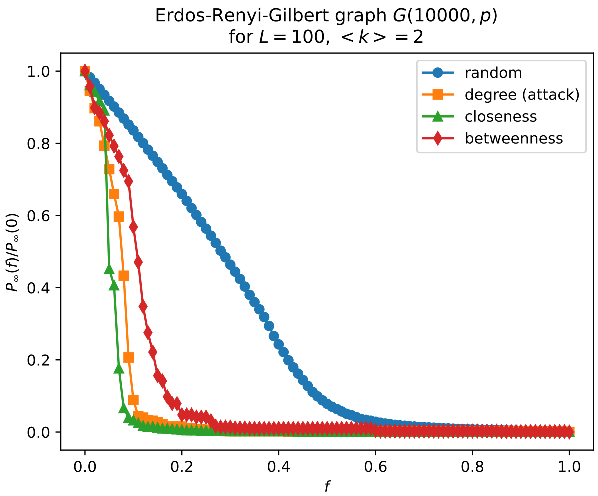


(b) Barabasi-Albert graph $BA(N, m)$

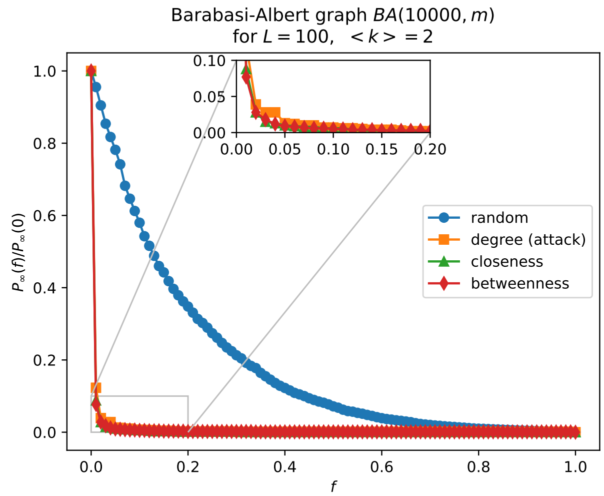


(c) Watts and Strogatz graph $WS(N, k, \beta)$

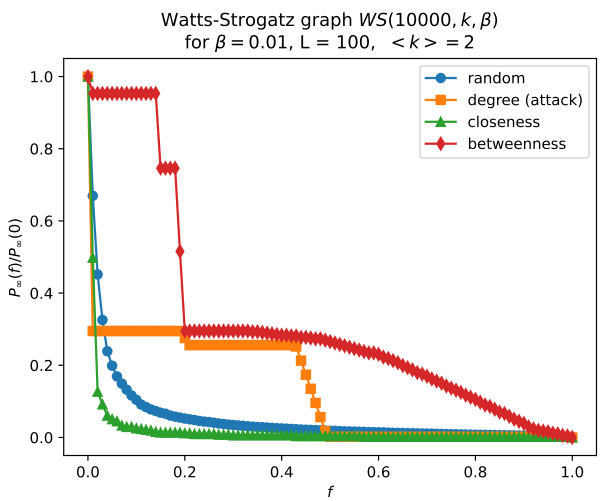
Figure 4: Presentation of the robustness analysis based on betweenness centrality for the giant component $P_\infty(f)/P_\infty(0)$ as a function of f for several networks of size $N = 10^4$ using an ensemble average over $L = 100$ samples.



(a) Erdős–Rényi-Gilbert graph $G(N, p)$



(b) Barabasi-Albert graph $BA(N, m)$



(c) Watts and Strogatz graph $WS(N, k, \beta)$

Figure 5: Robustness comparison for various models with an average degree $<k>= 2$.

3 Conclusions

In these three studied graph models (random graphs (ER), small-world (WS), scale-free (BA)) it's remarkable how results show the same pattern. There are very minor differences in the success of sequential targeted attacks based on various centrality measures. In networks with clustering, attacks based on betweenness and closeness, as well as those based on degree centrality, are essentially identical in efficacy. Attacks that focus on eliminating random nodes are unlikely to have a significant impact on the network. Nevertheless, an attack strategy that relies on random node removal will likely involve the removal of a huge number of nodes, greatly reducing the attack's effectiveness. Therefore, for example, the power grid network is considered to be resilient to random attacks. For scale-free networks the degree-based strategy performs better than the betweenness-based strategy.

Appendices

```python
def random_graph(N, p):
    '''Returns a random graph with N nodes which are connected with
     probability p
       Parameters:
        N (int):   Number of nodes
        p (float): Probability of connecting nodes'''

    if p < 0 or  p  > 1:
        raise ValueError("p is probability between 0 and 1")
    if isinstance(N, int) == False or N < 0:
        raise ValueError("N is non-negative integer number")
    G = nx.Graph()
    list_of_nodes = list(range(0, N, 1))
    G.add_nodes_from(list_of_nodes)
    for pair in combinations(list_of_nodes, 2):
        rand_number = np.random.random()
        if rand_number < p:
            G.add_edge(pair[0], pair[1])
        else:
            pass
    return G
```

Listing 1: Implementation of Erdős–Rényi-Gilbert model

```python
def Barabasi_Albert_graph(m0, m, t):
    ''' Returns Barabasi-Albert graph with m0+t nodes, m0 number of initial nodes
    and m number of new node's links. We start with complete graph with m0 nodes.
    Then each node is adding and connecting to m existing nodes with probability p,
    which depends on the degree
    Parameters:
        m0 (int): Number of initial nodes
        m (int):  Number of new nodes links
        t (int):  Number of new nodes (future steps)'''

    if m < 1 or m >= m0:
        raise ValueError("Barabasi Albert network must have m >= 1")

    if m0 < 0 or t < 0:
        raise ValueError("m0 and t are non-negative number")

    G = nx.complete_graph(m0)
    new_node = m0

    for _ in range(t):
        G.add_node(new_node)
        for e in range(0, m):
            counter = 0
            while counter != 1:
                if len(G.edges()) == 0:
                    chosen_node = 0
                else:
                    probabilities_of_nodes = []
                    for node in G.nodes():
                        node_degree = G.degree(node)
                        node_probability = node_degree / sum(dict(G.degree()).values()) # (2 * len(G.edges()))
                        probabilities_of_nodes.append(node_probability)
                    chosen_node = np.random.choice(G.nodes(), p = probabilities_of_nodes)
                new_edge = (chosen_node, new_node)
                if new_edge in G.edges():
                    chosen_node = np.random.choice(G.nodes(), p = probabilities_of_nodes)
                elif (new_edge[1], new_edge[0]) in G.edges():
                    chosen_node = np.random.choice(G.nodes(), p = probabilities_of_nodes)
                else:
                    G.add_edge(new_edge[0], new_edge[1])
                    counter = 1
        new_node += 1
    return G
```

Listing 2: Implementation of Barabasi-Albert model

```python
def Watts_Strogatz_graph(N, K, p):
    ''' Returns Watts-Strogratz graph with N nodes, K number of connections between nodes
    at the beggining and probability of rewiring p. At the begining there is a regular,
    circular graph. Then each node is rewired to a randomly chosen node with
    probability p
    Parameters:
        N (int):  Number of nodes
        K(int):   Number of connections between nodes at the beggining
        p(float): Probability of rewiring'''

    if p < 0 or  p > 1:
        raise ValueError("p is probability between 0 and 1")

    if isinstance(N, int) == False or isinstance(K, int) == False:
        raise ValueError("N and K are integer numbers")

    G = nx.Graph()
    list_of_edges = set()
    list_of_nodes = list(range(N))
    list_of_nodes_x2 = list(range(N)) * 2
    if int(K) % 2 == 0:
        k = int(K/2)
    else:
        k = int((K-1)/2)

    for i in range(N):
        from_vert = [i] * k
        to_vert = list_of_nodes_x2[i + 1 : i + k+1]
        list_of_edges.update(set(zip(from_vert, to_vert)))
    list_of_edges=list(list_of_edges)

    for link in list_of_edges:
        rand_number = np.random.random()
        if rand_number >= p:
            counter = 0
            while counter != 1:
                if link in G.edges(): # check if link is in graph
                    random_index = np.random.choice(list(range(len(list_of_edges))))
                    link = list_of_edges[random_index]
                elif (link[1], link[0]) in G.edges():
                    random_index = np.random.choice(list(range(len(list_of_edges))))
                    link = list_of_edges[random_index]
                else:
                    G.add_edge(link[0], link[1])
                    counter = 1
        elif rand_number < p:
            list_of_other_edges = []
            for i in range(len(list_of_nodes)): # all possible edges with one node
                list_of_other_edges.append((link[0], i))
            list_of_other_edges.remove((link[0], link[0]))
            new_links = []
            new_links = [i for i in list_of_other_edges if i not in G.edges()] # all possible edges which are not existing
            random_index = np.random.choice(list(range(len(new_links))))
            new_edge = new_links[random_index]
            G.add_edge(new_edge[0], new_edge[1])
    return G
```

Listing 3: Implementation of Watts-Strogratz model