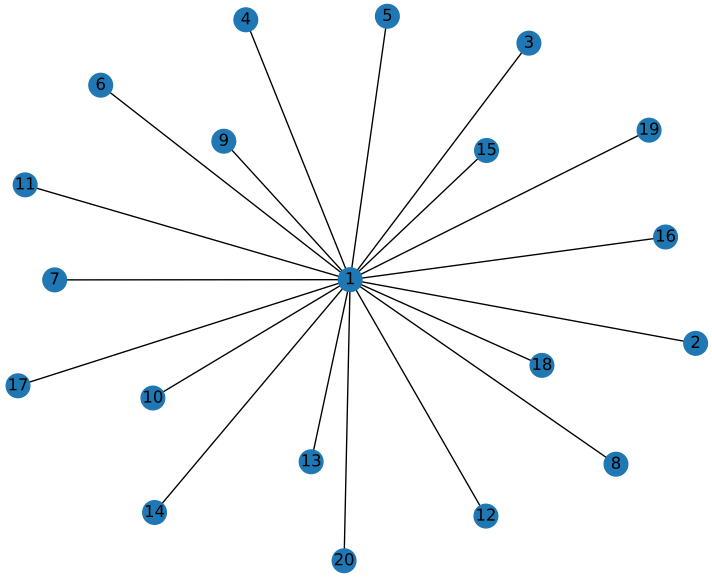
Jakub Omieljaniuk (250090) – sprawozdanie z Listy 2 (Technologie Sieciowe)

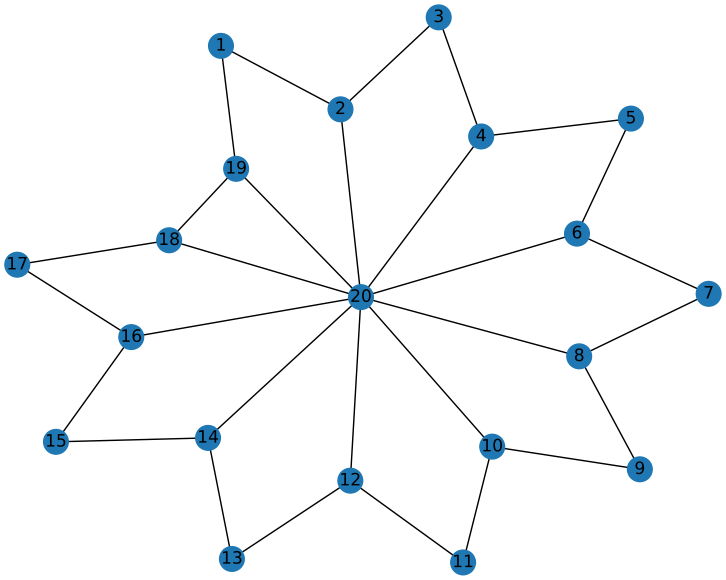
Topologię sieci dzielimy na **fizyczną**, która określa, w jaki sposób urządzenia są ze sobą połączone oraz **logiczną** opisującą, w jaki sposób przesyłane są dane pomiędzy urządzeniami.

W mojej symulacji sieci będę testował 4 rodzaje fizycznych topologii sieci:

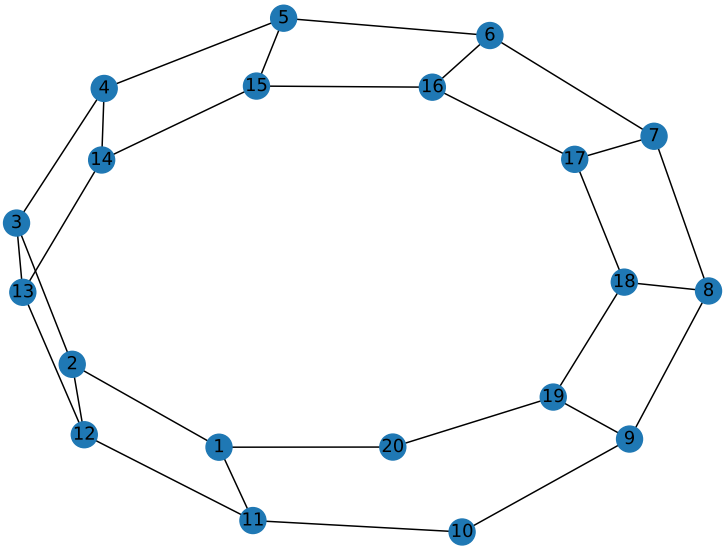
- ***topologię gwiazdy***, gdzie wyróżnia się jeden wierzchołek, z którym połączone są wszystkie pozostałe.
- ***topologię koła,*** która jest połączeniem topologii gwiazdy (*rys. 1*) i pierścienia (gdzie każdy wierzchołek jest połączony z dwoma sąsiednimi – w tym końcowy z początkowym, tworząc okrąg).
  Ze względu na ograniczenie liczby krawędzi $|E| < 30$ muszę zastosować częściową implementację topologii koła, gdzie centralny wierzchołek jest połączony tylko z połową pozostałych *(rys. 2)*.
- ***topologię siatki***, a dokładniej ponownie jej częściową implementację. Topologia siatki zakłada, że każdy wierzchołek ma poprowadzoną krawędź do wszystkich pozostałych. Moja topologia będzie łączyła każdy wierzchołek (poza dwoma) z 3 innymi – wtedy liczba krawędzi będzie wynosiła 29.
- ***topologię losową***, z zachowaniem spójności grafu i ograniczeniem liczby krawędzi.



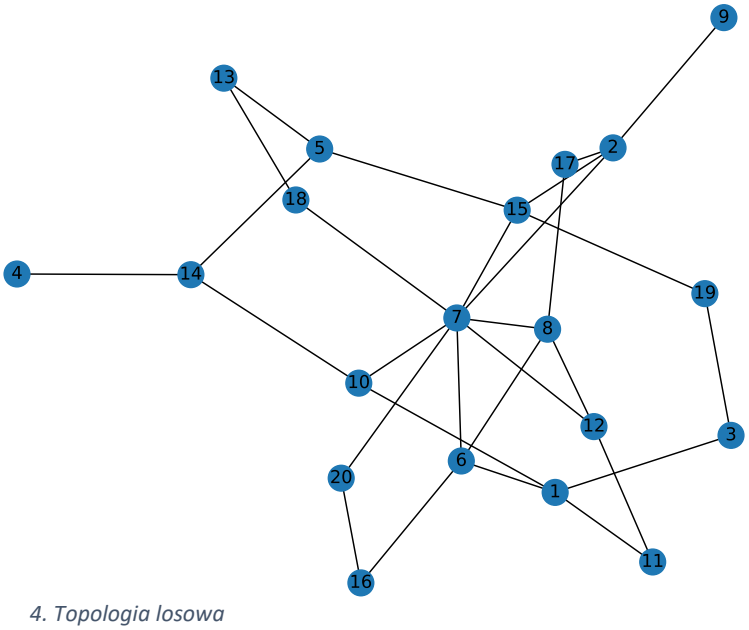
1. Topologia gwiazdy



2. Topologia częściowego koła



3. Topologia częściowej siatki



4. Topologia losowa

Topologia logiczna mojej sieci będzie zakładać, iż **pakiety są wysyłane po najkrótszej możliwej ścieżce**. Wybrałem ten sposób przede wszystkim na łatwość implementacji wyznaczania takiej ścieżki w języku Python (używając funkcji *shortest_path* z biblioteki *networkx*), a także brak jednoznacznie lepszej alternatywy prostej w implementacji. Ilość wysyłanych pakietów pomiędzy poszczególnymi wierzchołkami będzie determinowana przez **macierz natężeń**.

wierzchołek odbierający \ wierzchołek wysyłający	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	**0**	24	13	17	8	18	15	25	19	1	17	11	25	9	19	15	6	23	22	20
2	5	**0**	25	27	21	18	13	20	18	27	21	4	10	7	14	12	12	26	16	4
3	7	17	**0**	7	18	8	9	21	10	27	14	1	11	26	24	21	5	17	7	10
4	19	14	27	**0**	17	1	12	8	25	7	15	24	3	3	23	17	16	0	21	14
5	17	8	12	2	**0**	4	10	2	7	6	27	0	3	25	10	11	11	17	14	18
6	6	23	26	10	17	**0**	5	18	27	2	15	2	0	8	13	18	14	15	2	5
7	19	3	8	1	5	1	**0**	11	20	3	7	9	9	18	5	19	20	4	8	20
8	13	17	5	9	16	20	19	**0**	6	20	15	13	18	27	15	12	3	18	24	20
9	5	23	22	19	23	19	27	17	**0**	9	7	11	18	26	26	5	14	5	9	12
10	16	24	16	25	25	14	10	14	17	**0**	15	11	2	9	15	26	3	17	17	18
11	20	6	13	12	22	26	10	12	6	0	**0**	25	6	4	8	5	17	13	12	1
12	1	13	13	4	8	27	19	22	8	22	19	**0**	15	0	12	10	2	14	18	11
13	15	23	16	8	8	15	14	27	26	8	11	6	**0**	14	16	23	20	19	9	8
14	8	27	20	19	22	15	5	22	3	7	4	12	0	**0**	10	16	22	8	1	27
15	17	3	19	7	24	19	9	3	7	22	22	12	22	23	**0**	5	10	3	13	21
16	8	19	14	5	10	10	11	21	7	13	2	4	0	6	14	**0**	26	9	17	15
17	1	2	6	25	5	2	21	13	24	18	1	7	16	5	13	6	**0**	13	26	23
18	15	21	25	12	24	13	9	4	12	8	10	3	18	10	5	15	11	**0**	11	13
19	4	15	3	4	26	3	27	6	13	25	20	12	7	2	23	27	11	11	**0**	23
20	3	18	5	15	0	19	9	11	7	22	7	17	8	24	7	15	7	7	18	**0**

5. Macierz natężeń (liczba wysyłanych pakietów)

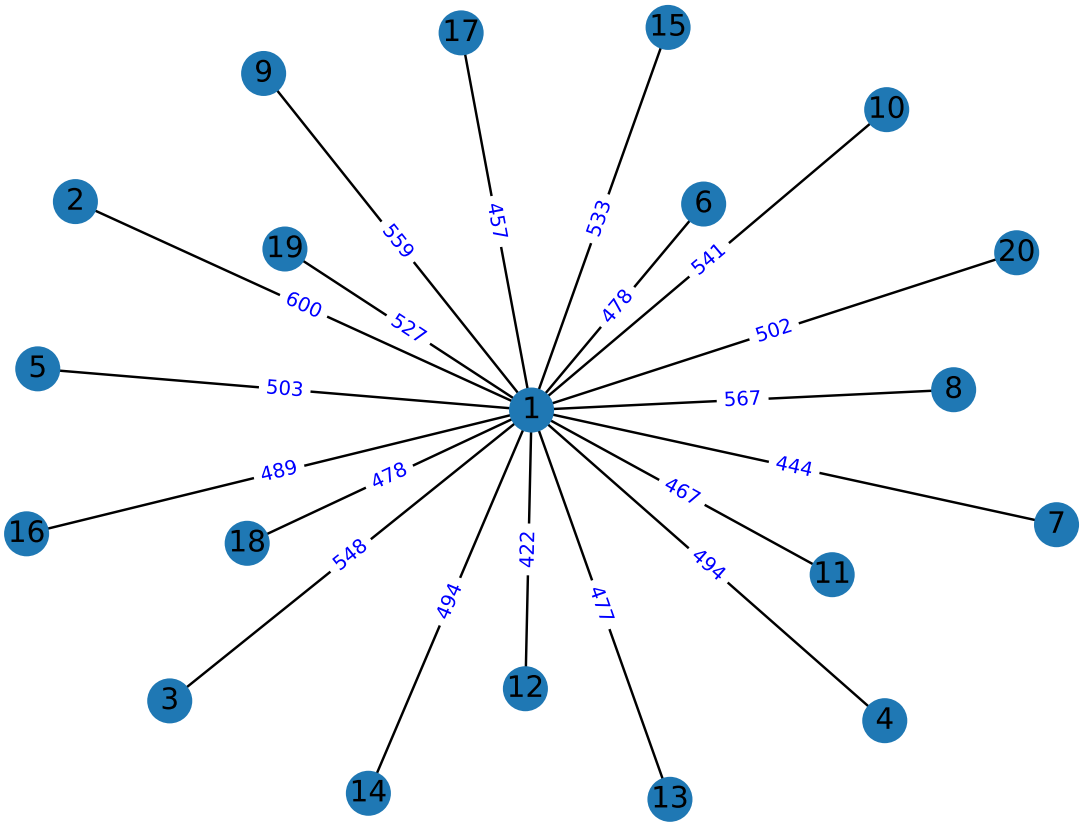
Przy generowaniu losowej macierzy posłużyłem się ograniczeniem maksymalnej liczby pakietów, które można wysłać do pojedynczego wierzchołka (mając na uwadze, iż później będziemy je zwiększać). Dlatego też początkowo żaden z wierzchołków nie wysyła jednorazowo więcej niż **27 pakietów** do innego wierzchołka. Poniżej kod generujący losową macierz natężeń o podanej liczbie wierzchołków i maksymalnej możliwej liczbie pakietów wysyłanych pomiędzy dwoma wierzchołkami:

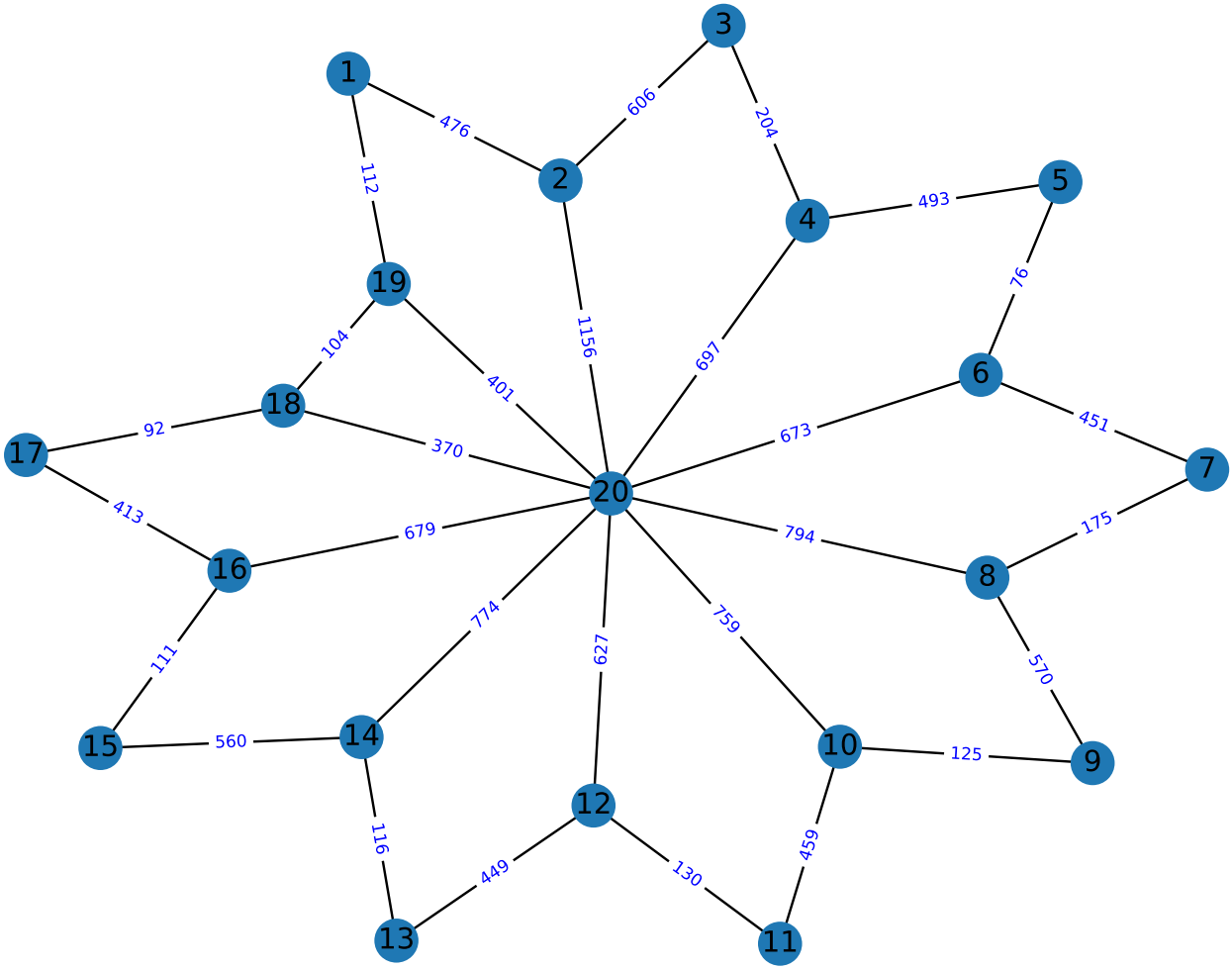
```python
def make_random_intensity_matrix(vertices, max_packets):
    matrix = []
    for x in range(vertices):
        row = []
        for y in range(vertices):
            if x == y:
                row.append(0)
            else:
                row.append(random.randint(0, max_packets))
        matrix.append(row)
    return matrix
```

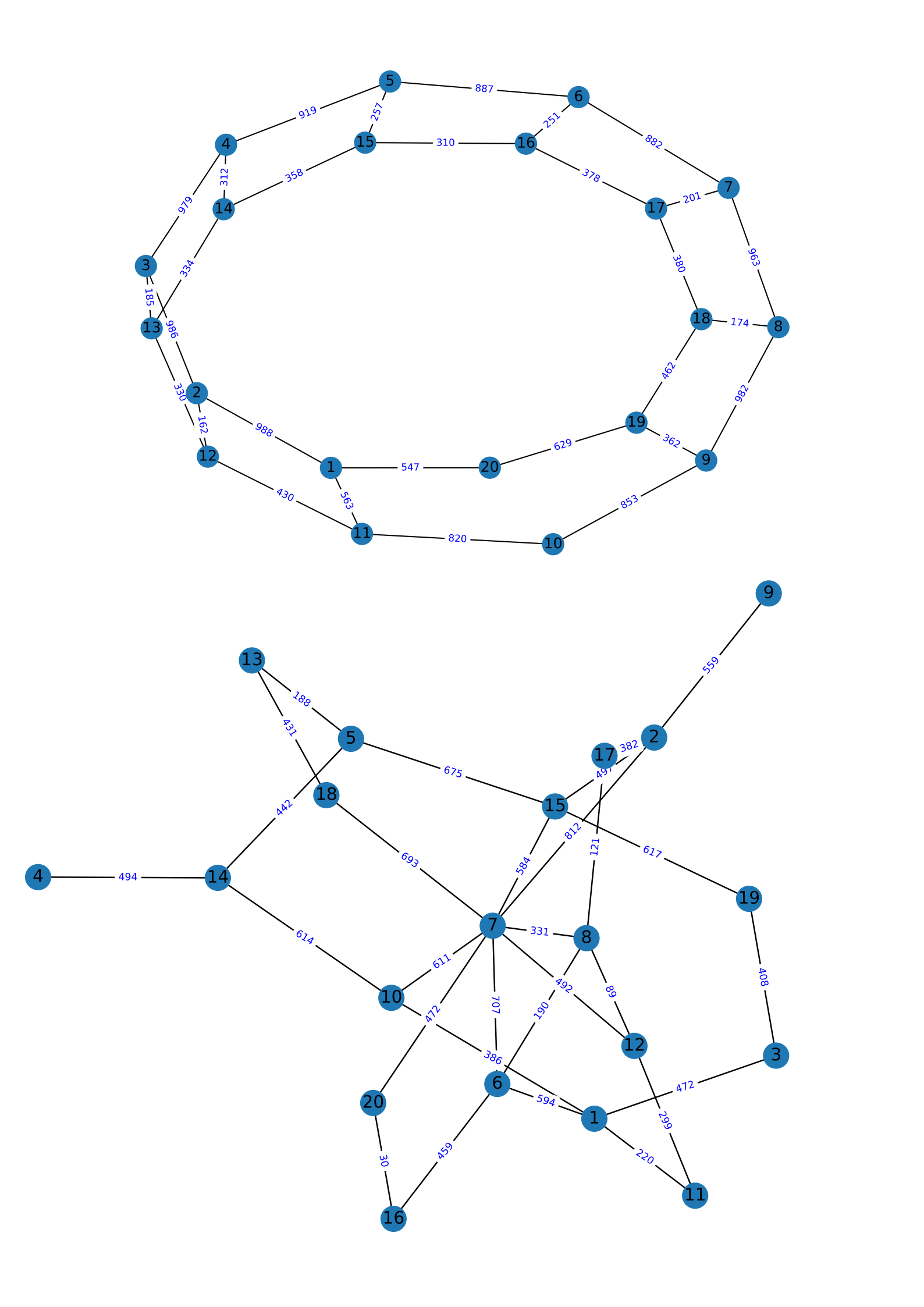
6. Kod Python: generowanie macierzy natężeń

Program podczas symulacji sieci będzie wyznaczał najkrótsze ścieżki między każdymi wierzchołkami i obciążał krawędzie występujące na tej ścieżce odpowiednią liczbą pakietów. **Funkcją przepływu** będzie zatem suma pakietów, które zostały

wprowadzone do danego wierzchołka. Poniżej przedstawienie graficzne ile pakietów przepłynęło na poszczególnych krawędziach.






5
887
6
919
257
251
4
15
310
16
882
312
358
378
7
979
14
17
201
334
380
963
3
185
986
18
174
8
13
462
330
2
982
162
988
19
362
12
1
547
20
629
9
430
563
853
11
820
10

Przepustowość krawędzi wyznaczałem dla każdej z osobna proponując następujący wzór dla funkcji *c(e)*:

$$c(e) = 3 * a(e) * m$$

Gdzie *a(e)* oznacza aktualną liczbę pakietów, które przechodzą przez daną krawędź *e*, natomiast *m* jest rozmiarem pakietów wyrażoną w bitach.

W programie do przechowywania informacji o przepływie i przepustowości krawędzi stworzyłem dwa odpowiadające im słowniki. Dla obu słowników **kluczem** jest para krawędzi *(i, j)* między którymi występuje krawędź, a **wartością** odpowiednio liczba pakietów przechodząca przez tą krawędź i ustalona na jej podstawie przepustowość krawędzi. Poniżej kod funkcji obliczających przepływ i przepustowość dla krawędzi:

```python
class NetworkGraph:
    def __init__(self):
        self.edges = []
        self.edge_flow_Dict = {}
        self.edge_max_capacity_Dict = {}
        self.nx_format = nx.Graph()

        def calculate_edge_overload(self, intensity_matrix):
            for x in range(len(intensity_matrix)):
                for y in range(len(intensity_matrix)):
                    if x != y:
                        edges_path = nx.shortest_path(self.nx_format, x + 1, y + 1)
                        for index in range(len(edges_path) - 1):
                            if (edges_path[index], edges_path[index + 1]) in self.edge_flow_Dict.keys():
                                self.edge_flow_Dict[(edges_path[index], edges_path[index + 1])] += intensity_matrix[x][y]
                            else:
                                self.edge_flow_Dict[(edges_path[index + 1], edges_path[index])] += intensity_matrix[x][y]

        def calculate_edge_max_capacity(self, package_size):
            for load_edge in self.edges:
                self.edge_max_capacity_Dict[load_edge] = 3 * self.edge_flow_Dict[load_edge] * package_size
```

7. Kod Python: obliczanie funkcji przepustowości i przepływu

Przy obliczaniu **miary niezawodności sieci** przekazuję graf do specjalnej funkcji, która symuluje uszkodzenia krawędzi (bazując na parametrze *p* - prawdopodobieństwa nie uszkodzenia oraz funkcji *random.random()* zwracającej liczbę z zakresu (0,1)), „filtrując„ z listy bazowej krawędzi te, które zostały uszkodzone (*p < random.random()*). Następnie na tej „uboższej" kopii grafu dokonuję ponownego wytyczenia ścieżek dla pakietów i obliczenia funkcji przepływu *a(e)*. Na koniec obliczana jest wartość opóźnienia *T* zgodnie z podanym wzorem:

$$T = \frac{1}{G}\sum_{e \in Edges} \frac{a(e)}{\frac{c(e)}{m} - a(e)}.$$

Czynność tą powtarzam zadaną przez parametr *attempts* razy. Niezawodnością sieci będzie liczba prób, które spełniły warunek $T < T_{max}$ i nie rozspójniła grafu, w stosunku do wykonanych prób (*attempts).* Kod funkcji badającej niezawodność sieci:

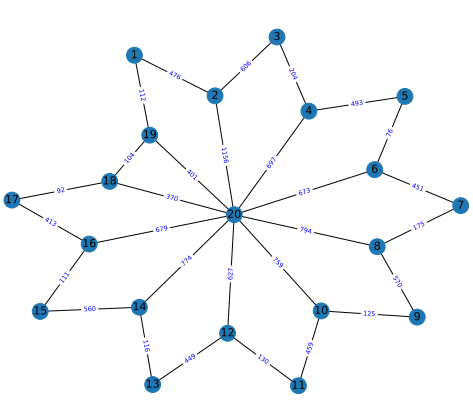
```python
def simulate_damage(edges, p):
    new_edges = []
    for edge in edges:
        if p > random.random():
            new_edges.append(edge)

    return new_edges
```

8. Kod Python: symulowanie uszkodzenia krawędzi

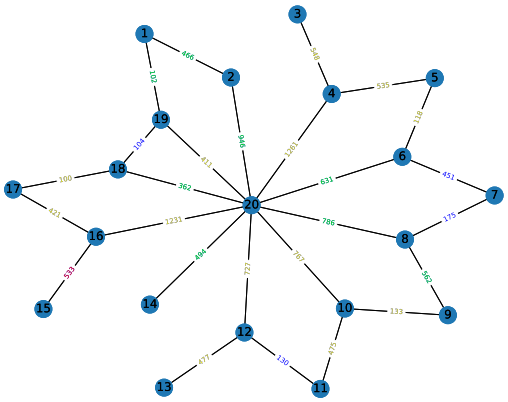
```python
import random
import networkx as nx
import GraphGenerator as ggen
import NetworkIntensity


def calculate_network_reliability(network_graph, intensity_matrix, working_probability, max_delay, package_size, attempts=100):
    successful_attempts = 0
    for i in range(attempts):
        try:
            copy_graph = ggen.NetworkGraph()
            copy_graph.edges = simulate_damage(network_graph.edges, working_probability)
            copy_graph.nx_format.add_nodes_from(range(1, 21))
            copy_graph.nx_format.add_edges_from(copy_graph.edges)
            for load_edge in copy_graph.edges:
                copy_graph.edge_flow_Dict[load_edge] = 0
            copy_graph.calculate_edge_overload(intensity_matrix)

            g = NetworkIntensity.get_sum(intensity_matrix)
            sum_e = 0.0
            for edge in copy_graph.edges:
                a_e = copy_graph.edge_flow_Dict[edge]
                c_e = network_graph.edge_max_capacity_Dict[edge]
                sum_e += a_e / ((c_e / package_size) - a_e)
            delay = (1 / g) * sum_e

            if max_delay > delay > 0:
                successful_attempts += 1
        except nx.exception.NetworkXNoPath:
            pass
    return successful_attempts / attempts
```

9. Kod Python: obliczanie niezawodności sieci

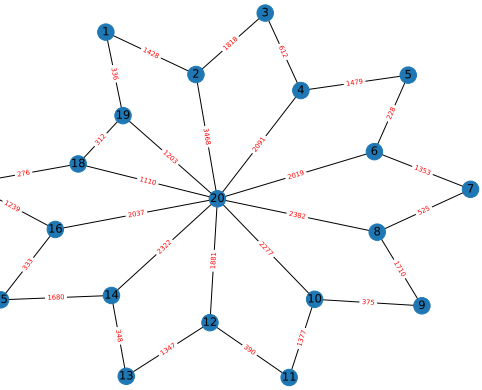
Poniżej grafy przedstawiające przykładową zmianę przepływu i uszkodzeń krawędzi dla pojedynczej iteracji symulacji oraz graf wyjściowy i graf przedstawiający maksymalną liczbę pakietów wyznaczoną w poprzednim punkcie (jest to *c(e)/m*):



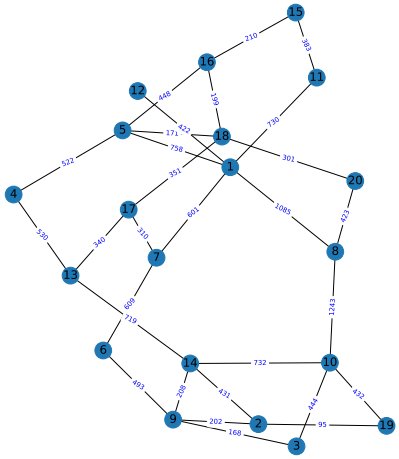
Wyjściowy przepływ pakietów

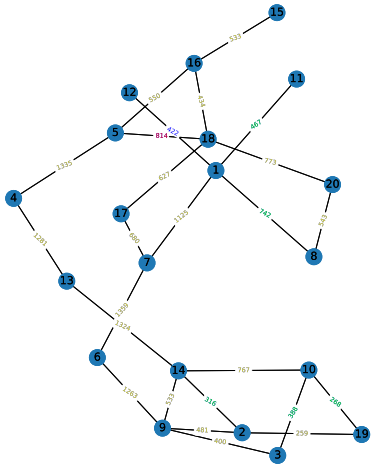


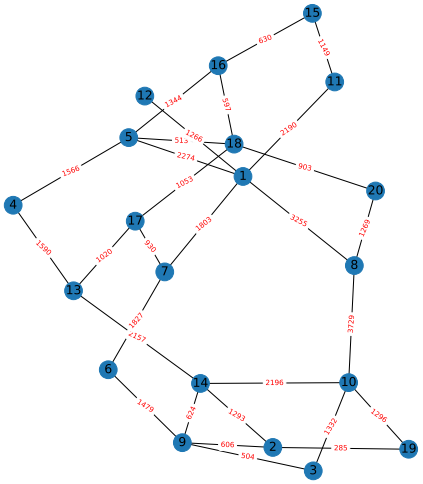
Przepływ pakietów po usunięciu (uszkodzeniu) części krawędzi dla p = 0.9

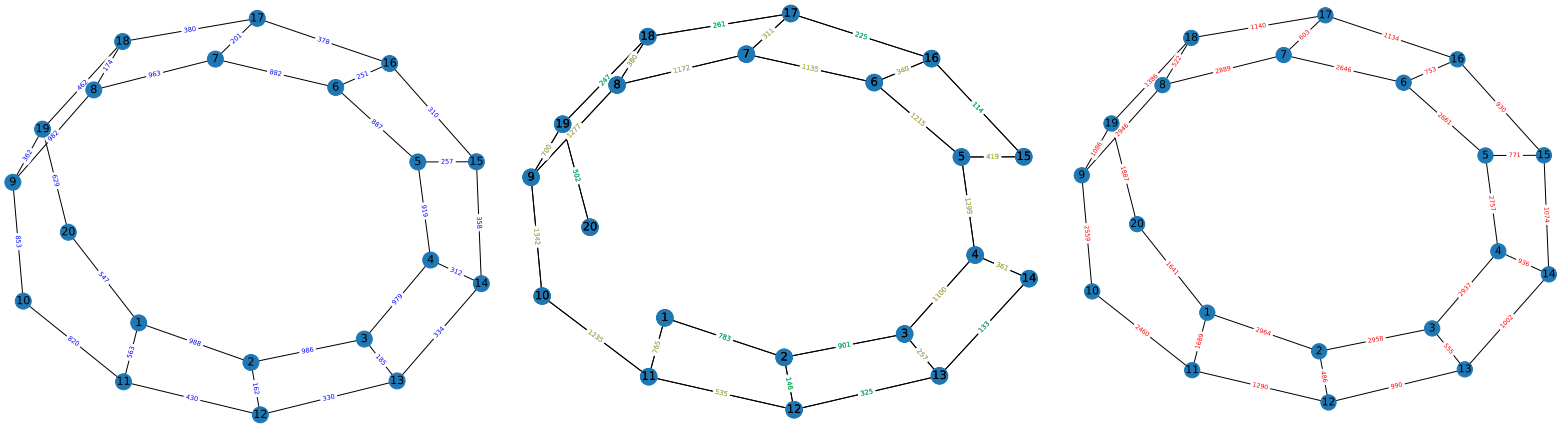


Maksymalna liczba pakietów



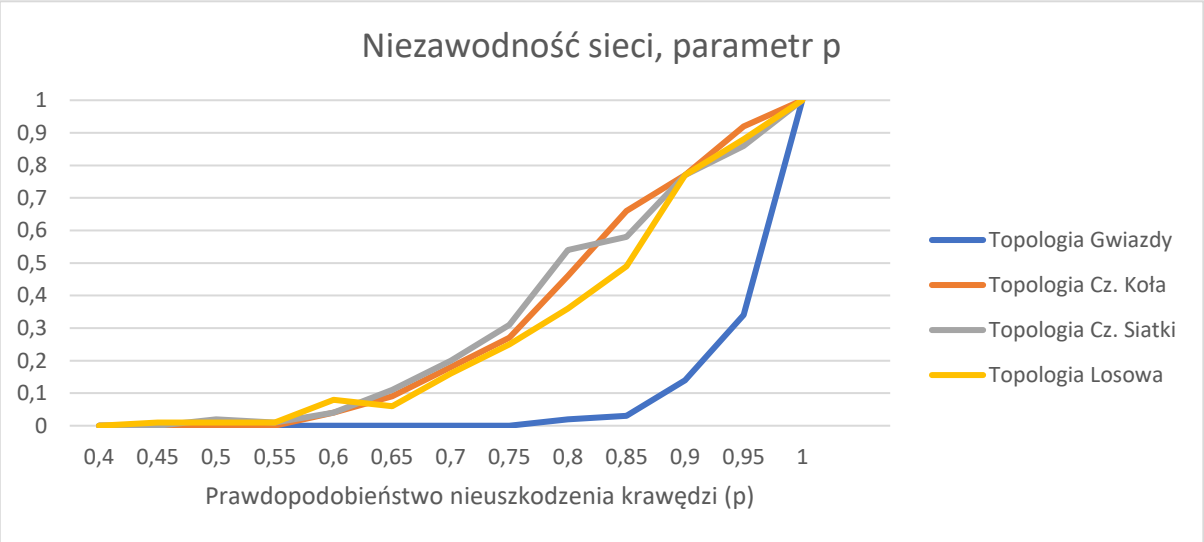




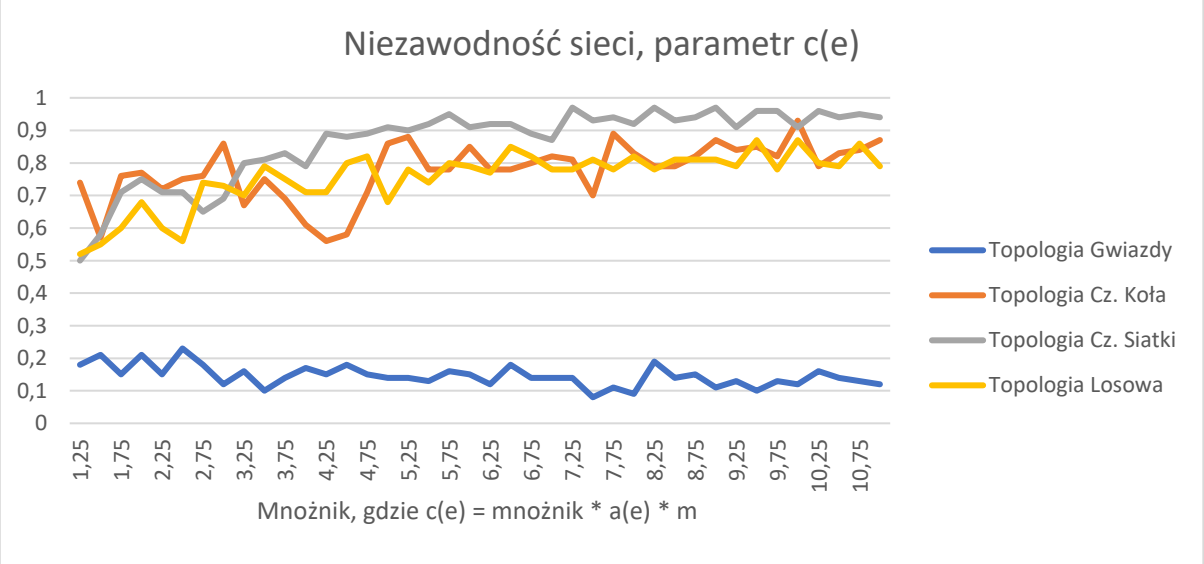


Topologia gwiazdy nie sprawdza się w tym przypadku – wystarczy, że jedna krawędź zostanie uszkodzona, aby graf stał się niespójny. Niezawodność sieci byłaby zatem przybliżonym prawdopodobieństwem, że żadna z krawędzi nie zostanie uszkodzona.

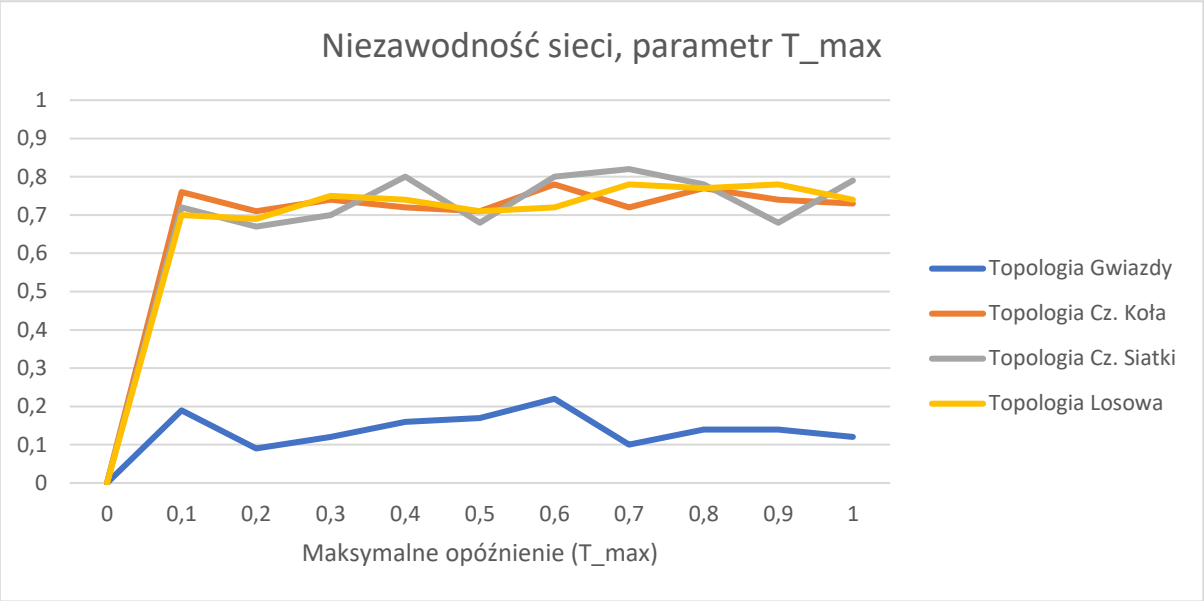
Poniżej przedstawiam **wykresy dla testów** niezawodności sieci ze względu na **parametr**: *p, T_max*, *N* (macierz natężenia), *c(e)*. Dla uproszczenia przyjąłem, że wielkość pakietu *m = 1*, ponieważ i tak wyznaczone wcześniej wartości największego dopuszczalnego przepływu *c(e)* są bezpośrednio zależne od *m*. W późniejszych badaniach sieci będę zwiększał mnożnik we wzorze na c(e) co jest tożsame ze zwiększanie wielkości pakietu *m* przy stałym mnożniku (aktualnie wynoszącym 3). Wartości funkcji *c(e)* są takie same dla wszystkich pomiarów, a ich wartości przedstawiłem na poprzednich grafach.



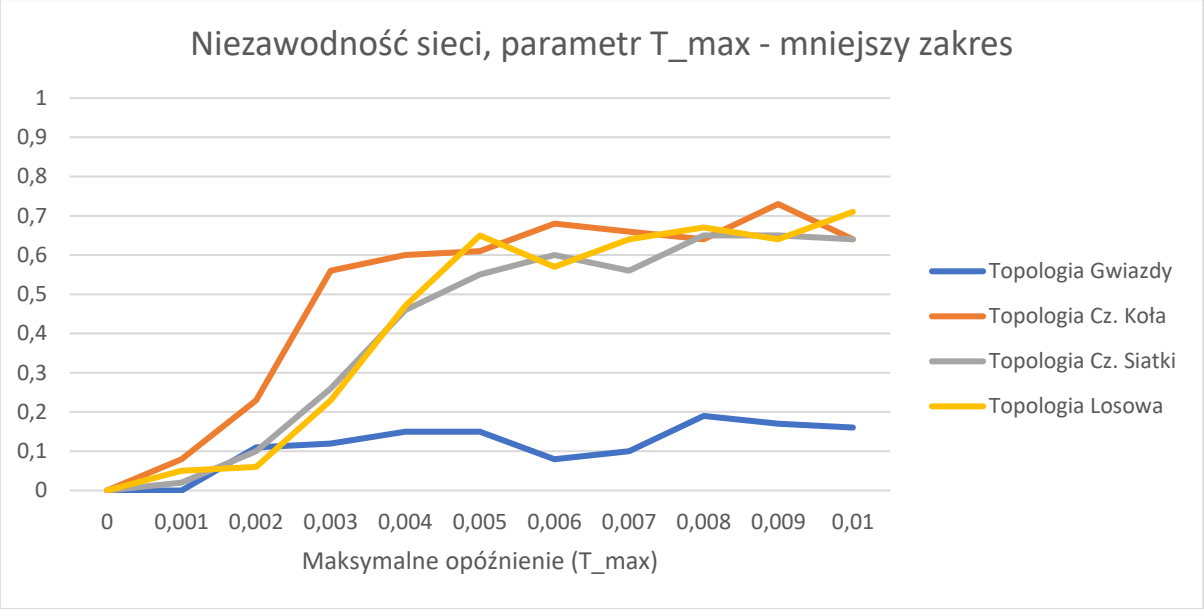
*10. Niezawodność sieci dla ustalonej macierzy, T_max = 0.1, c(e) = 3*a(e)*



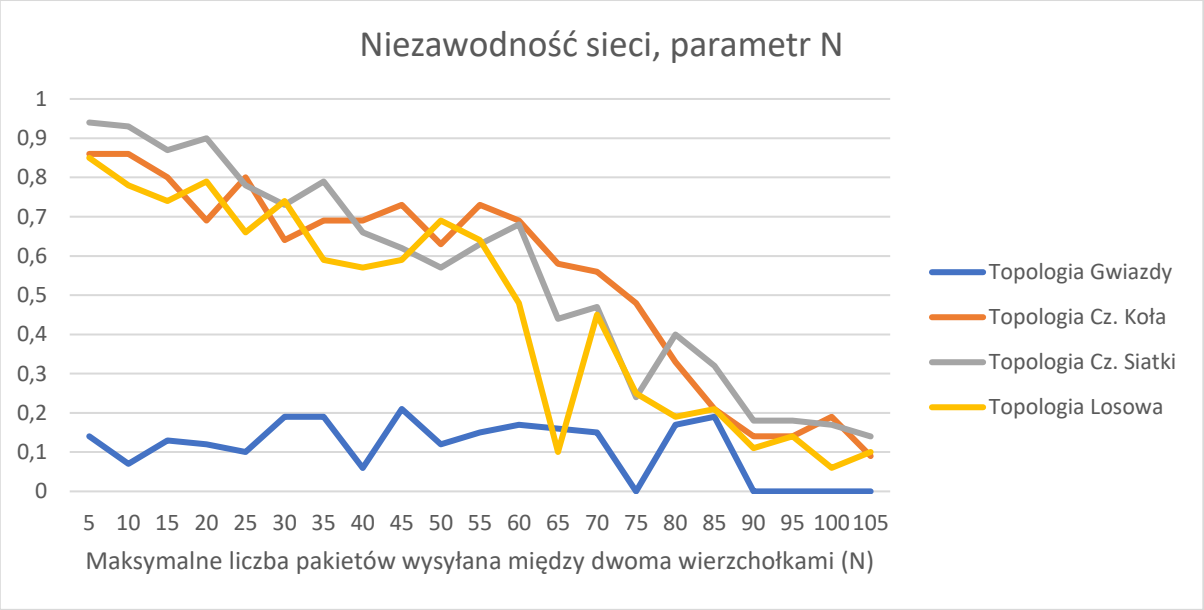
11. Niezawodność sieci dla ustalonej macierzy, T_max = 0.1, p=0.9



*12. Niezawodność sieci dla ustalonej macierzy, p=0.9, c(e) = 3*a(e)*

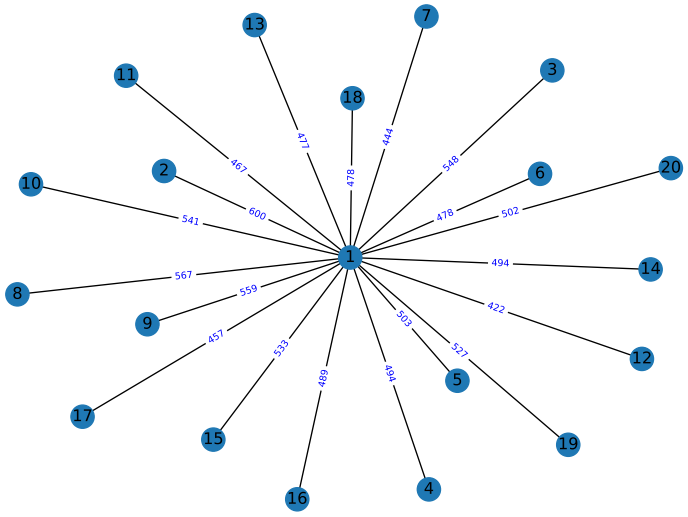


*13. Niezawodność sieci dla ustalonej macierzy, p=0.9, c(e) = 3*a(e) - mniejszy zakres*

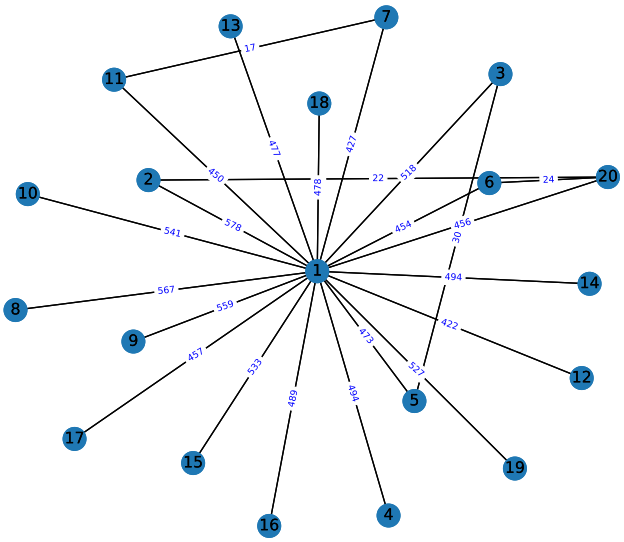


*14. Niezawodność sieci dla T_max = 0.1, p=0.9, c(e) = 3*a(e)*

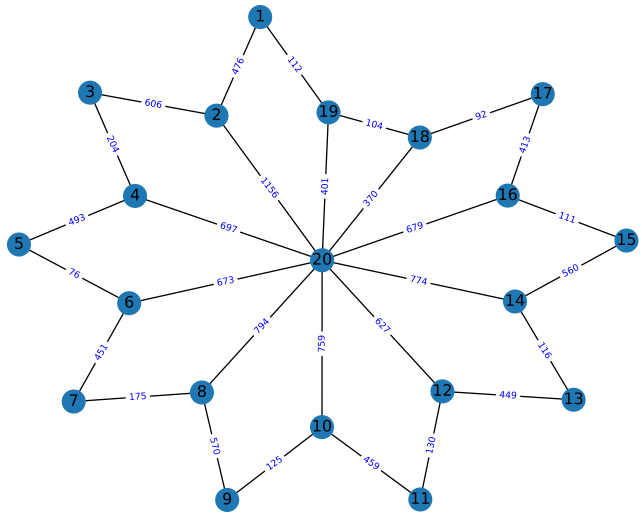
Ostatnim doświadczeniem jest **dodawanie krawędzi do poszczególnych topologii**. Poniżej przedstawiam topologie za pomocą grafów, gdzie widać jak rozkłada się przesyłanie pakietów po dodaniu krawędzi:



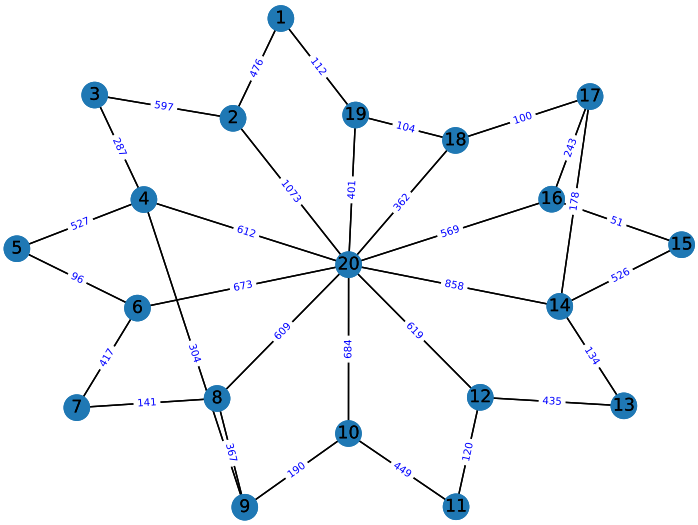
15. Topologia Gwiazdy - początkowy przepływ pakietów



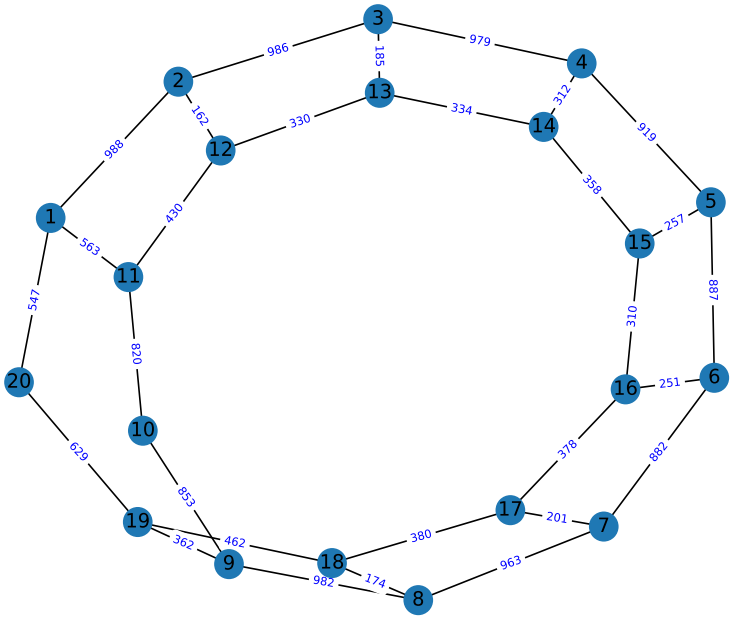
16. Topologia Gwiazdy - przepływ pakietów po dodaniu 4 krawędzi



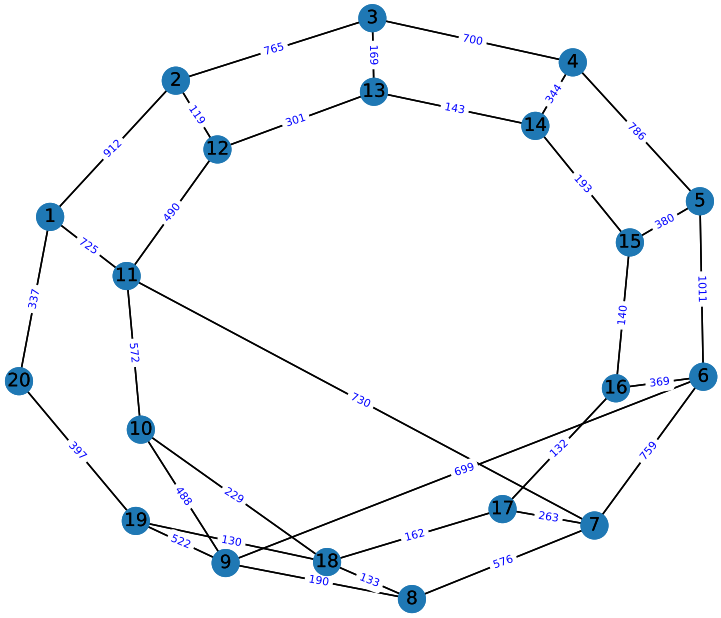
17. Topologia Cz. Koła - początkowy przepływ pakietów



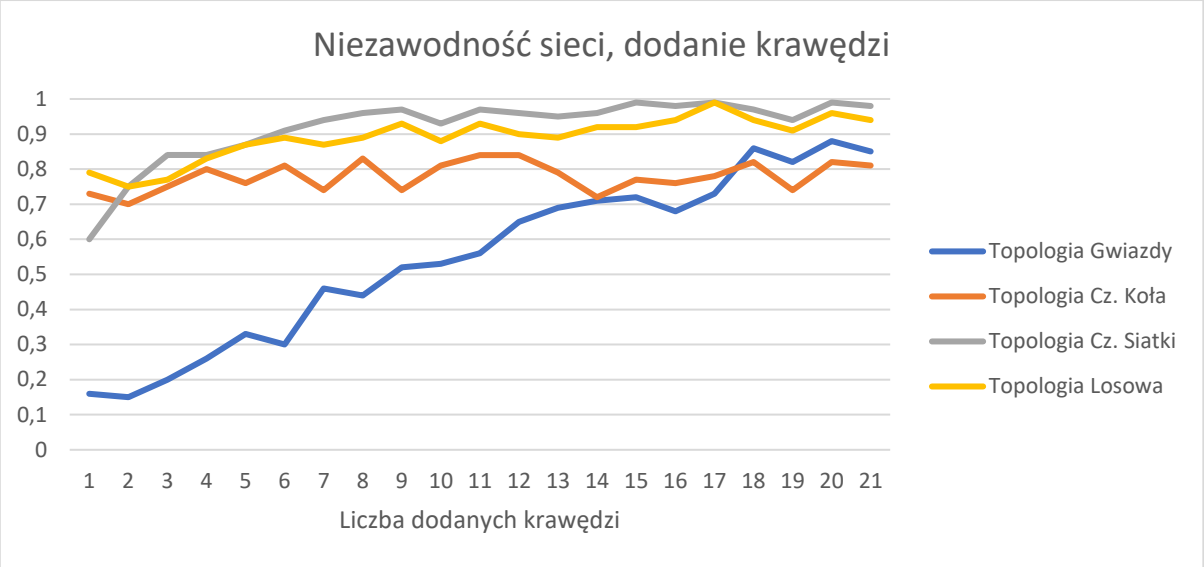
18. Topologia Cz. Koła - przepływ pakietów po dodaniu 2 krawędzi



19. Topologia Cz. Siatki - początkowy przepływ pakietów



20. Topologia Cz. Siatki - przepływ pakietów po dodaniu 3 krawędzi



*21. Niezawodność sieci dla ustalonej macierzy, T_max = 0.1, p=0.9, c(e) = 3*a(e) i avg(c(e)) dla nowych krawędzi*

Podsumowanie:

- Krawędzie zazwyczaj zapychały się tam, gdzie maksymalny dopuszczalny przepływ (*c(e)*) miał najniższą wartość w danym grafie. Aby poprawić ten stan rzeczy należałoby zmodyfikować funkcję *c(e)* w taki sposób, aby wartości funkcji były bardziej uśrednione w skali całego grafu. W niektórych jednak przypadkach metoda ta wcale nie byłaby lepsza – np. przy generowaniu grafu dla topologii losowej, gdzie wytworzyłby się most (*krawędź, która uspójnia nasz graf łącząc dwa mniejsze podgrafy spójne*). Taka krawędź potrzebowałaby większej przepustowości od pozostałych.
- Topologia gwiazdy najlepiej rozkładała przepływ pakietów na poszczególne krawędzie, pomimo ich najmniejszej ilości. Wykazała ona również największą odporność na wzrost wartości macierzy intensywności. Jej wadą jest jednak skupienie całej logistyki przepływów na jednym wierzchołku, co przy uwzględnieniu prawdopodobieństwa awarii wierzchołków klasyfikowało tę topologię na ostatnim miejscu.
- Topologia gwiazdy wraz z dodawanymi krawędziami upodabniała się do topologii koła i sieci. Na wykresie (*rys. 21*) widać wyraźnie, że wraz z tym procesem wzrastała także niezawodność tej sieci. Pokazuje to, że topologie rozproszone są bardziej niezawodne.
- Topologia losowa wykazywała niekiedy chaotyczne zachowania negatywnie wpływające na zachowanie tej sieci (widać to szczególnie na wykresie *rys. 14*, gdzie zwiększane były wartości macierzy natężeń). Topologie cz. koła i siatki wykazywały lepszą stabilność i niezawodność od topologii losowej.
- Topologia cz. koła i losowa słabo reagowały na zwiększanie krawędzi w sieci (wzrost niezawodności jedynie o ok. 0,1 przy dodaniu 21 węzłów). Topologia gwiazdy odnotowywała największe wzrosty niezawodności startując z najniższego pułapu i zatracając swoją charakterystykę wraz z dodawanymi krawędziami. Topologia cz. sieci szybko osiągnęła niezawodność bliską 1, co oznacza, że od pewnego momentu (dodaniu ok. 8 węzłów) była niemal niezawodna podczas testów.