Technologie sieciowe 2

Gabriel Wechta

10.04.2020

1 Wstęp

Pierwsza część listy drugiej skupia się na zamodelowaniu sieci komputerowej. Do modelu będę używał obiektów matematycznych jak grafy i macierze, oraz języka Python, który ów obiekty ma w przyjazny sposób zaimplementowane. Druga część listy skupia się na przetestowaniu i przedstawieniu wyników dla różnych zaproponowanych modeli, i porównaniu ich.
Do fragemntów programu, które wymagają wyjaśnienia dołączę kawałki kodu albo pesudokodu. Pozostałe fragmenty opiszę i opiszę ich działanie. Cały program dołączę do sprawozdania.

2 Model

Do reprezentacji grafu używam biblioteki netwrokx. Funckja main najpierw generuje grafy $G = (V, E)$, następnie macierz N i słownik przepustowości krawędzi gdzie kluczem jest para $(a, b) \in E, a < b$, a wartością - przepustowość krawędzi. Testy przeprowadzam modyfikując ciało funckji main, a więc poniższy listing to raczej pseudokod.

```
def main():
    size = 20
    # Generate graph G with size nodes

    N = createMatrix(size)
    N = evaluate(N, par)
    # Matrix N is randomly populated with integers in range (1, par)

    capacity_edge_dic = create_edge_dic(G)
    c(capacity_edge_dic, capacity)

    print(delay_reliability(G, capacity_edge_dic, N, T_max, p))
```

Funkcja przepływu a również wykorzystuje słownik krawędzi i planowane trasy dla pakietów ustalane za pomocą algortymu Dijkstry do znajdowania najktórtszych ścieżek w grafach. Niestety nie mam przyjemności być bogaty w wiedzę

z teorii grafów, ale obiecuję rychłą poprawę w tej kwestii. Ponadto najkrótsza ścieżka do grafów, które przetestowałem, wydaje się sprawdzać nienajgorzej. W kilkunastu próbach przejrzałem wartości słownika krawędzi po usunięciu paru krawędzi i przepływ był akceptowalnie równomiernie rozłożony na pozostałe krawędzie.

```
def all_paths_to_travel(graph, matrix):
    to_travel = []
    for x in range(len(matrix[0])):
        for y in range(len(matrix[0])):
            path = []
            if matrix[x][y] != 0:
                path.append((x, y, matrix[x][y]))
                path.append(nx.shortest_path(graph, x, y))
                to_travel.append(path)
    # Element in list to_travel has format [(src, dst, weight), path]
    # This format later is very handy
    return to_travel

def a(to_travel, edges):
    for guidance in to_travel:
        cost = guidance[0][2]
        for x in range(len(guidance[1]) - 1):
            first = guidance[1][x]
            second = guidance[1][x + 1]
            if first < second:
                edges[(first, second)] += cost
            else:
                edges[(second, first)] += cost
```

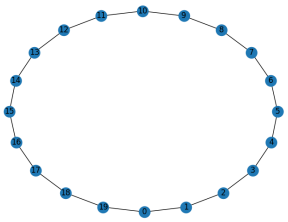
3 Miara niezawodności

Miarę niezawodności sieci mierzę poniższymi funkcjami. Idea jest następujaca, kopiujemy stworzony graf, przeprowadzamy *repetitions* prób z *inetrvals* interwałami. W każdym interwale jest prawdopodobieństwo $1-p$, że krawędź zostanie uszkodzona. Jeżeli graf przestanie być spójny odrzucamy interwał, w przeciwnym razie liczymy na nowo ścieżki dla pakietów i obliczamy a na E. Następnie według wzoru $T = \frac{1}{G} \cdot \sum_{e \in E} \frac{a(e)}{\frac{c(e)}{m} - a(e)}$ obliczamy T (funckja pomocnicza delay) i jeżeli jest dodatnie oraz akceptowalne na tle dopuszczalengo opóznienia T_{max} liczymy interwał jako zaliczony. Funkcja delayReliability zwraca stosunek interwałów udanych do wszystkich interwałów. Po zakończeniu próby, graf zostaje przywrócony do pierwotnego kształtu.

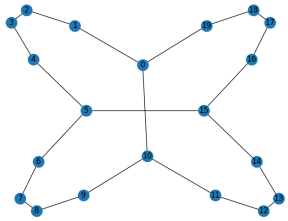
```
def delay(graph, matrix, m, edge_dic, capacity_edge_dic):
    G = sum(reduce(lambda x, y: x + y, matrix))
    sumus_maximus = 0
    for edge in graph.edges():
        try:
            sumus_maximus += edge_dic[edge] /
            (capacity_edge_dic[edge] / m - edge_dic[edge])
        except ZeroDivisionError:
            return -1
    return 1 / G * sumus_maximus

def delayReliability(start_graph, capacity_edge_dic, matrix, t_max,
                     p, intervals=10, repetitions=1000):
    positive = 0
    for _ in range(repetitions):
        graph = start_graph.copy()

        for _ in range(intervals):
            for edge in list(graph.edges()):
                if random.random() > p:
                    graph.remove_edge(*edge)
            if not nx.is_connected(graph):
                break
            all_paths = all_paths_to_travel(graph, matrix)
            edge_dic = create_edge_dic(graph)
            a(all_paths, edge_dic)
            print(edge_dic)
            d = delay(graph, matrix, 8, edge_dic, capacity_edge_dic)
            if 0 < d < t_max:
                positive += 1

    return positive / (repetitions * intervals)
```

4 Testy

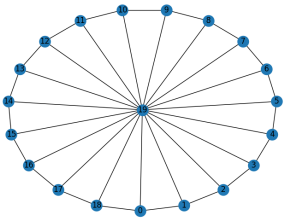
Wynik testów znacząco zależy od wyboru topolgii sieci. Biorąc pod uwagę, że nasze badanie polega na wyłączaniu pewnych krawędzi, rozsądne wydaje się badanie topologi pierścienia i pochodnych. Graf liniowy, przedstawiający topologie magistrali, przestaje być spójny w momencie utraty pierwszej krawędzi. Pozostają jeszcze grafy k-spójne, o wysokim k oraz grafy regularne ale testowanie ich na Technologiach Sieciowych wydaje się być nienajrozsądniejsze z powodu tego jak w rzeczywistości projektowane są sieci komputerowe, to jest z uwzględnieniem ceny za metr kabla.
Skupię się więc na testowaniu topologii pierścienia i podobnych.
W zadaniu ostatnim (podrozdział 4.3) pomęczę trochę graf cykliczny, ponieważ widzę w nim największy potencjał do zmian na lepsze.



Rysunek 1: Graf cykliczny



Rysunek 2: Graf shuriken



Rysunek 3: Graf kołowy

4.1 Zwiększanie wartości macierzy N

Ustalona topologia i dobrana przepustowość(= 10000)			
Zakres $n(i,j)$	Graf cykliczny	Graf shuriken	Graf kołowy
0-20	0.063	0.137	0.654
1-20	0.064	0.172	0.618
1-40	0.0	0.066	0.457
10-20	0.0	0.123	0.641
20-20	0.0	0.085	0.398

Przykładowe wywołanie programowe:

```
print(delay_reliability(shuriken, shuriken_cap, N, 0.1, 0.95)
```

Uwagi:

- W żadkim przypadku gdy graf cykliczny nie utracił spójności, wykazuje bardzo małą odporność na wzrost natężenia.
- Graf kołowy jest najbardziej odporny na wzrost natężenia.
- $T_{max} = 0.1$ i $p = 0.95$ wydają się być na ten moment rozsądne.

4.2 Zwiększanie przepustowości

Ustalona topologia i macierz natężeń (zakres(0,20))			
Przepustowość	Graf cykliczny	Graf shuriken	Graf kołowy
1000	0.0	0.0	0.103 (!)
2500	0.007	0.020	0.334
5000	0.0	0.069	0.431
10000	0.063	0.152	0.647
20000	0.157	0.260	0.658
1000000	0.149	0.268	0.651

Przykładowe wywołanie programowe:

```
print(delay_reliability(shuriken, shuriken_cap, N, 0.1, 0.95)
```

Uwagi:

- Wyniki pokazują, że w okolicach 20000 graf cykliczny i shurkien osiągają swoje maksima sprawności i wzrost przepustowość nie ma już wpływu. Graf kołowy osiąga swoje maksimum już w okolicach 10000.
- Grał kołowy niemrawo (ale za to z jaką klasą!) daje rade przy bardzo małej przepustowości. Zastosowanie lepszego algorytmu znajdowania ścieżki z całą pewnoscią jeszcze zwiększyłoby jego możliwości.

4.3 Dodawnaie krawędzi

Graf cykliczny, ustalona macierz natężenia	
Liczba dodanych krawędzi	Niezawodność
0	0.063
1	0.116
2	0.141
3	0.153
5	0.199
10	0.384
20	0.634
30	0.786
60	0.889

Funkcja dodająca krawędzie:

```
def improve(graph, how_many):
    added = 0
    while added < how_many:
        first = random.randint(0, graph.number_of_nodes())
        second = random.randint(0, graph.number_of_nodes())
        if first != second and not graph.has_edge(first, second):
            graph.add_edge(first, second)
            added += 1
```

Uwagi:

- Fantastyczne jak dodanie jednej krawędzi polepszyło topologie.
- Oczywiście wszystko zależy od tego jak zostaną wylosowane węzły, które połączymy. Może się zdarzyć, że będą losowane symetrycznie budując topologie siatki, lub jeden węzeł będzie najbardziej połączony przypominając graf kołowy. Oba przypadki mają swoje zalety i minusy. Mogą oczywiście też zdarzyć się topologie patologiczne (np. duża liczba krawędzi przy paru węzłach), natomiast topologie właśnie tego typu są przecież najczęściej spotykanymi.
  Przy uzupełnianiu tabelki starałem się wybierać tylko te grafy, które przypominały poprzednika.
- Wynik dla 10 dodanych krawędzi odpowiada liczbą krawędzi grafowi kołowemu. Symetryczne umiejscowienie krawędzi jak widać ma ogromny wpływ (0.384 vs. 0.618).
- Przy 30 i 60 przez długi czas czekałem na lepszy graf. Było to spowodowane tendencją grafów do posiadania wyizolowanych węzłów.

5 Wnioski, uwagi i usprawiedliwienia

- Najistotniejsze w trwałości sieci komputerowej jest jej topologia, przepustowość jednego połączenia i jego trwałość. Parametry współgrają ze sobą przez co ciężko wybrać najważniejszy z nich.
- Najlepsze okazały się grafy wysoce spójne i regularne.
- Przeprowadziłem również parę testów dla grafów kompletnych, które (bez zaskoczenia) okazały się osiągać najlepsze wyniki rzędu 0.990, dla tych samych danych dla których przeprowadzony był test z dodawaniem krawędzi.
- Wbrew FAQ na stronie, w swoim sprawozdaniu, nie uzmienniłem m i capacity, oraz zastosowałem prosty algorytm szukanie najkrótszej ścieżki, natomiast za każdym razem patrzyłem jak zmiany tych parametrów wpływają na wyniki testów, co jak rozumiem jest celem zadania tego zadania, ponadto zostawałem przy tych parametrach, które dawały najciekawsze wyniki.