Problem wykluczania tras komunikacyjnych w mieście

Projekt z przedmiotu Optymalizacja Kombinatoryczna

Katarzyna Róg

grudzień 2022

Spis treści

1 Cel projektu

Celem projektu było stworzenie narzędzia do badania liczby tras komunikacyjnych w mieście, które można by wykluczyć jednocześnie z użytku. Rozwiązanie problemu ma być optymalne, tj. wynik ma reprezentować jak największą liczbę wykluczonych jednocześnie tras.

2 Streszczenie problemu

Bardzo często władze miasta wykorzystują dostępne w bieżącej chwili zasoby na wyremontowanie linii komunikacyjnych w mieście. Tymi liniami mogą być ulice, trakcje tramwajowe, ścieżki rowerowe, chodniki itd. Nieraz jednak przedsięwzięcia te nie są zbytnio przemyślane. Wykluczając dane trasy uniemożliwiane jest mieszkańcom bezproblemowe i szybkie przemieszczanie się po mieście. W projekcie postaram się rozwiązać ten problem.

Program konkretnie mówi o tym, ile tras komunikacyjnych można jednocześnie remontować, tak aby nie komplikować mieszkańcom nadmiernie dojazdów. Innymi słowy - ile linii komunikacyjnych można wykluczyć w tym samym czasie, tak aby czas przejazdu z każdego węzła komunikacyjnego do innego węzła nie przekraczał pewnego limitu. W projekcie tym limitem jest 150% poprzedniego czasu potrzebnego na dojazd z pewnego węzła do innego. Oznacza to, że jeśli najkrótsza trasa z puntku A do punktu B przed remontem zabierała x czasu, tak po odcięciu dróg ma ona zabierać nie więcej niż $1,5x$ czasu.

3 Opis rozwiązania

Rozwiązanie postawionego problemu opiera się głównie na **teorii grafów**. Grafy będą zasadniczym narzędziem, którym należy się tutaj posłużyć. Implementacja opiera się na pracy na grafach ważonych. Graf ważony jest reprezentacją mapy miasta, na którą składają się węzły komunikacyjne, połączone są liniami komunikacyjnymi. Trasy komunikacyjne (krawędzie grafu) posiadają wagi, które reprezentują czasy potrzebne na dotarcie z danego węzła do innego (z wierzchołka A do wierzchołka B grafu). Węzłem komunikacyjnym jest wierzchołek grafu. Po przerobieniu wersji fabularnej problemu na bardziej matematyczną, celem projektu jest znalezienie największej liczby krawędzi, które można usunąć z grafu, pod określonymi warunkami.

Tymi warunkami są:

- najkrótsze ścieżki pomiędzy każdą parą wierzchołków w powstałym grafie, nie mogą być dłuższe niż $1,5$ długości odpowiadających im ścieżek w grafie wyjściowym
- nowo powstały graf musi być spójny

Do sprawdzenia spójności grafu wykorzystano algorytm przeszukiwania grafu w głąb (DFS). Zastosowano odpowiednią funkcję, która wykorzystując ten algorytm może bezpośrednio odpowiedzieć na pytanie, czy graf jest spójny. Do wyznaczenia najkrótszych ścieżek przydatny był algorytm Dijkstry. W głównej części programu, która wprost badała liczbę możliwych do usunięcia krawędzi, zaimplementowane są 3 algorytmy:

1. Brute force
2. Algorytm zachłanny
3. Heurystyka

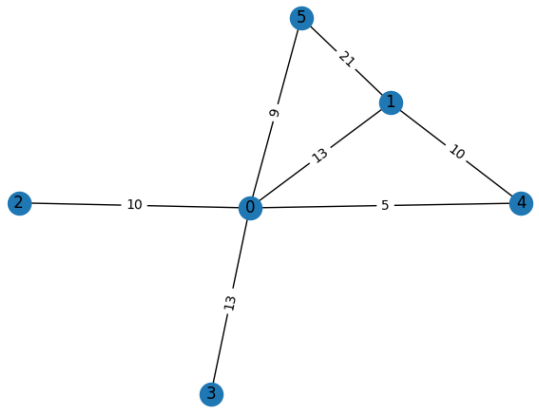
Każdy z tych algorytmów daje rozwiązanie dopuszczalne, takie które jest zadowalające, jednak nie każde zawsze daje rozwiązanie optymalne.

4 Implementacja

- Językiem programowania użytym w projekcie był Python.
- Maszynową reprezentacją grafu jaką użyto w projekcie jest macierz sąsiedztwa, rozszerzona dodatkowo o wagi krawędzi łączących dane wierzchołki. Macierz, w miejscach w których dwa wierzchołki są incydentne ma wpisane wagi krawędzi łączącej te dwa wierzchołki. Komórki reprezentujące pętle własne uzupełniono zerami, natomiast wszystkie pozostałe komórki macierzy zawierają nieskończoność (krawędź łącząca dane dwa wierzchołki nie istnieje). W tej implementacji wagi krawędzi mieszczą się w przedziale $< 1; 30 >$. Współczynnik nasycenia grafu ustawiono na 50%, co oznacza, że liczba krawędzi grafu o n wierzchołkach będzie równa połowie liczby krawędzi grafu pełnego o n wierzchołkach. Program generuje grafy spójne.
  - Przykładowa macierz:

$$\begin{bmatrix} 0 & 13 & 10 & 13 & 5 & 9 \\ 13 & 0 & \infty & \infty & 10 & 21 \\ 10 & \infty & 0 & \infty & \infty & \infty \\ 13 & \infty & \infty & 0 & \infty & \infty \\ 5 & 10 & \infty & \infty & 0 & \infty \\ 9 & 21 & \infty & \infty & \infty & 0 \end{bmatrix}$$

– Jej reprezentacja graficzna:



– Generator losowych grafów opratych o macierz:

```
def generate_matrix(n):

rate = int(0.5 * (n * (n - 1) / 2))
matrix = []
for i in range(n):
        matrix.append([])
        for j in range(n):
                matrix[i].append(math.inf)
                if i == j:
                        matrix[i][j] = 0

for _ in range(rate):
        v1 = random.randint(0, n - 1)
        v2 = random.randint(0, n - 1)
        while matrix[v1][v2] != math.inf or
              matrix[v2][v1] != math.inf or v1 == v2:
                v1 = random.randint(0, n - 1)
                v2 = random.randint(0, n - 1)
        value = random.randint(1, 30)
        matrix[v1][v2] = value
        matrix[v2][v1] = value

return matrix
```

- W projekcie zawarte są również pomocnicze funkcje, które są potrzebne w implementacji. Są nimi:
  - zliczanie krawędzi w grafie na podstawie macierzy - `def edges_(mtarix)`
  - znajdowanie najkrótszej ścieżki z jednego wierzchołka do wszystkich pozostałych (algorytm Dijkstry) - `def find_shortest_path_dijkstry(matrix, vert)`
  - modyfikacja algorytmu Dijkstry tak, aby ścieżki wyznaczane były dla każdej pary wierzchołków - `def find_shortest_paths(matrix)`
  - algorytm DFS - sprawdzenie spójności grafu - `def DFS_check(matrix)`
  - przedstawienie grafu w postaci graficznej - `def draw_graphs(matrices)`
  - sprawdzenie podstawowego warunku (ograniczenia problemu) - `def check_the_limit:`

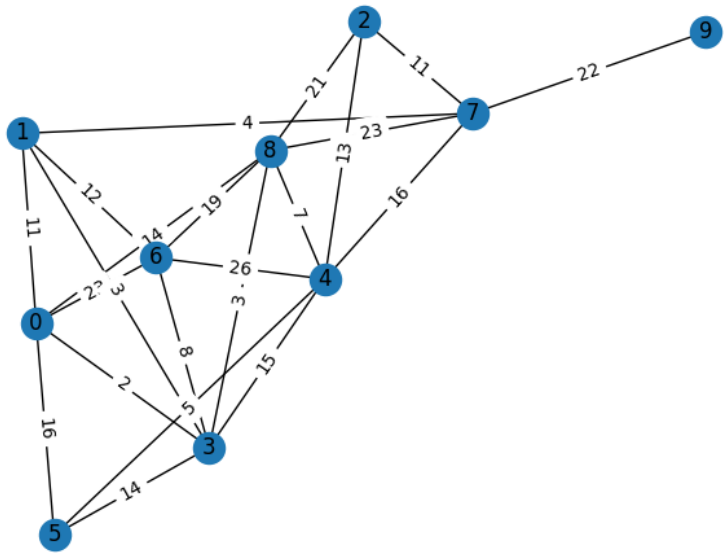
```
def check_the_limit(paths, new_paths):
        n = len(paths)
        condition = True

        for i in range(n):
                for j in range(n):
                        if new_paths[i][j] > 1.5*paths[i][j]:
                                condition = False
                                break
                        if not condition:
                                break

        return condition
```

5 Opis właściwego programu

Aby porównać wyniki, wszystkie trzy algorytmy zastosowano do tego samego grafu. Macierz tego grafu jest następująca:

$$\begin{bmatrix} 0 & 11 & \infty & 2 & \infty & 16 & 23 & \infty & 14 & \infty \\ 11 & 0 & \infty & 3 & \infty & \infty & 12 & 4 & \infty & \infty \\ \infty & \infty & 0 & \infty & 13 & \infty & \infty & 11 & 21 & \infty \\ 2 & 3 & \infty & 0 & 15 & 14 & 8 & \infty & 3 & \infty \\ \infty & \infty & 13 & 15 & 0 & 5 & 26 & 16 & 7 & \infty \\ 16 & \infty & \infty & 14 & 5 & 0 & \infty & \infty & \infty & \infty \\ 23 & 12 & \infty & 8 & 26 & \infty & 0 & \infty & 19 & \infty \\ \infty & 4 & 11 & \infty & 16 & \infty & \infty & 0 & 23 & 22 \\ 14 & \infty & 21 & 3 & 7 & \infty & 19 & 23 & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 22 & \infty & 0 \end{bmatrix}$$

Wizualizacja:



Rozwiązaniem, jakiego należy oczekiwać jest maksymalna ilość wierzchołków, które można usunąć z grafu tak, aby spełnione były warunki.

5.1 Brute force

Algorytm siłowy zawsze znajduje **rozwiązanie optymalne**, jednak niestety działa on tylko dla małych instancji problemu. Czas, w jakim wykonuje obliczenia jest wykładniczy, dlatego ciężko jest otrzymać rozwiązanie w „skończonym czasie". Program generuje wszystkie możliwe podgrafy grafu wyjściowego, co daje nam 2^E przypadków, gdzie E jest liczbą krawędzi w grafie. W danej implementacji generowane są grafy o współczynniku nasycenia równym 50%, stąd liczbę krawędzi można wyliczyć ze wzoru:

$$E = \lfloor \frac{0,5 \cdot [V(V1)]}{2} \rfloor \tag{1}$$

Gdzie V jest liczbą wierzchołków.

Liczba uzyskanych możliwości jest zatem dużo większa od liczby wierzchołków w grafie. W poniższej tabeli przedstawiono kilka wartości V, E i odpowiadającej im liczbie permutacji.

V	E	*permutacje*
5	5	2^5
6	7	2^7
7	10	2^{10}
8	14	2^{14}
9	18	2^{18}
10	22	2^{22}

Z tego powodu czas w jakim przeszukiwane są wszystkie rozwiązania rośnie wykładniczo. Dlatego działanie algorytmu sprawdzono dla grafów o maksymalnie 10 wierzchołkach, co daje rozwiązanie we względnie krótkim czasie.

W poniższej implementacji generowane są wszystkie **permutacje**, reprezentowane przez liczby binarne na E bitach. Każdy z bitów stanowi o jednej z krawędzi grafu wyjściowego. W zależności od tego czy w liczbie binarnej występuje 1 lub 0 na miejscu odpowiadającym danej krawędzi, to krawędź ta istnieje w nowym podgrafie, lub nie. W ten sposób możliwe jest wyznaczenie wszystkich podgrafów. Aby spełnione były założenia problemu usuwania krawędzi, należy sprawdzić czy wygenerowane podgrafy są spójne oraz czy spełniają warunek wyjściowy w problemie. W tym celu należy użyć funkcji `def check_the_limit` oraz `def DFS_check(matrix)`.

Do implementacji algorytmu siłowego użyto również pomocniczych funkcji:

- `def generate_binary_permutations(bits)` - generuje liczby binarne o liczbie bitów bits

- `def make_matrix_from_permutation(matrix, permutation, n)` - modyfikuje macierz wejściową na podstawie danej permutacji (gdy w liczbie binarnej występuje 0 to usuwamy krawędź) i zwraca liczbę usuniętych krawędzi

Funkcje te będą użyte również w pozostałych algorytmach.

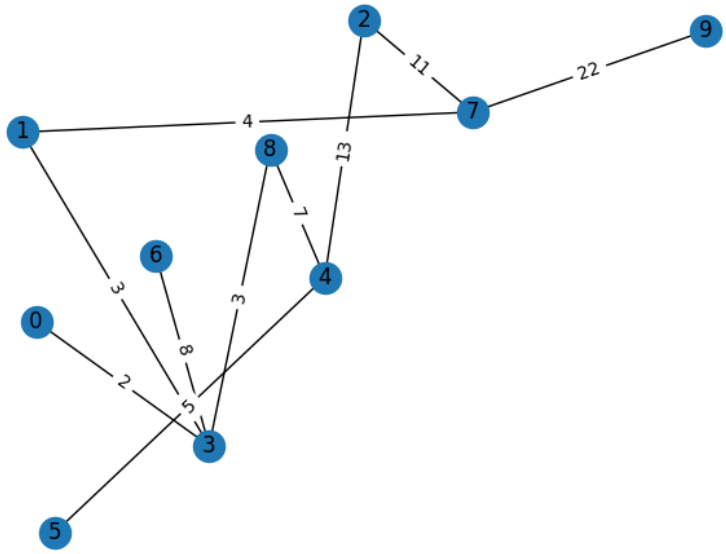
Algorytm Brute Force:

```
def delete_edges_brute_force(matrix):
        n = len(matrix)
        edges_count = edges_counter(matrix)
        paths = find_shortest_paths(matrix)
        solutions = []
        new_matrices = []

        edges_permutations = generate_binary_permutations(edges_count)

        for permutation in edges_permutations:
                new_matrix = copy.deepcopy(matrix)
                counter = make_matrix_from_permutation(new_matrix,
                                permutation, n)

                if DFS_check(new_matrix):
                        new_paths = find_shortest_paths(new_matrix)
                        if check_the_limit(paths, new_paths):
                                solutions.append(counter)
                                new_matrices.append(new_matrix)

        solution = max(solutions)
        return [solution, new_matrices[solutions.index(solution)]]
```

W wyniku działania tego algorytmu uzyskano rozwiązanie optymalne:

12

Graf można wtedy przedstawić w następującej postaci:



Czas potrzebny do wykonywania tego programu wynosił *676,6s* (czyli około 11min).

5.2 Algorytm zachłanny

Algorytm zachłanny nie generuje zawsze rozwiązania optymalnego. Najczęściej właśnie rozwiązanie jakie otrzymujemy jest **nieoptymalne**. Dlaczego zatem warto zastosować ten algorytm? Głównym atutem tego rozwiązania jest czas obliczeń. Jest on bardzo krótki w porównaniu do algorytmu siłowego - nie musi przeszukiwać całego spektrum rozwiązań, tylko stopniowo generuje własną drogę. Jeśli zgodzimy się na pewną niedokładność rozwiązania, a nie posiadamy odpowiedniej mocy obliczeniowej bądź zależy nam na rozpatrywaniu dużych instancji prolemu, algorytm zachłanny może okazać się dobrym rozwiązaniem.

W ogólności algorytm zachłanny działa na zasadzie wybierania rozwiązania najlepszego na dany moment. Nie świadczy to jednak o tym, że na koniec rozwiązanie będzie najlepszym ze wszystkich możliwych.

W bieżącym problemie, kryterium, którym kieruje się algorytm jest waga krawędzi. Rozważając krawędzie incydentne do danego wierzchołka, algorytm zdecyduje o usunięciu krawędzi o **największej wadze**. Po usunięciu danej krawędzi sprawdza, czy nowy graf spełnia warunki, a następnie powtarza czynność dla nowego wierzchołka, do które-

go dochodziła usunięta krawędź. W ten sposób uzyskane rozwiązanie będzie najlepsze spośród tych, wynikających z podanego wzorca działań (nie jest to rozwiązanie optymalne). Aby zwiększyć efektywność tego algorytmu, powyższe czynności wykonywane są dla *każdego wierzchołka grafu*. Wtedy można uzyskać względnie dobre rozwiązanie.

W implementacji nie użyto żadnych funkcji pomocniczych.

Algorytm zachłanny:

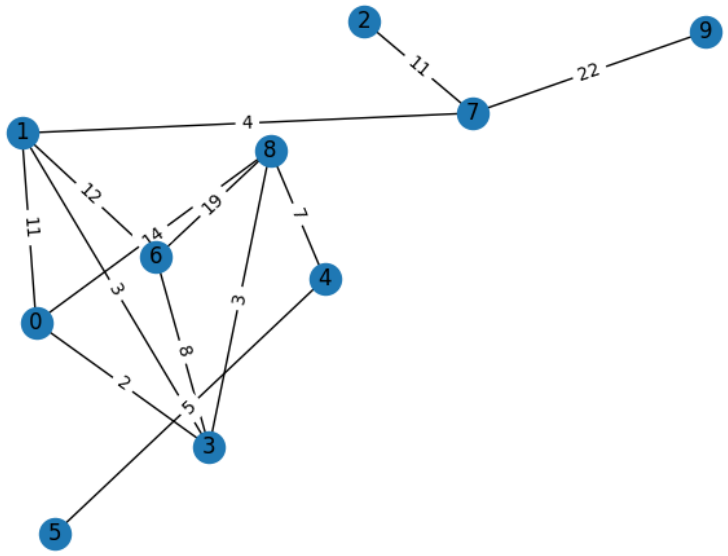
```
def delete_edges_greedy(matrix):
        n = len(matrix)
        edges = edges_counter(matrix)
        paths = find_shortest_paths(matrix)
        solutions = []
        new_matrices = []

        for v in range(n):
                new_matrix = copy.deepcopy(matrix)
                current_edge = 0
                while current_edge <= edges:
                        max_value = 0
                        ind = 0
                        for vert in range(n):
                                if new_matrix[v][vert] != 0 and
                                new_matrix[v][vert] != math.inf and
                                new_matrix[v][vert] > max_value:
                                        max_value = new_matrix[v][vert]
                                        ind = vert
                        new_matrix[v][ind] = math.inf
                        new_matrix[ind][v] = math.inf
                        current_edge += 1
                        v = ind

                        if DFS_check(new_matrix):
                                new_paths = find_shortest_paths(new_matrix)
                                if check_the_limit(paths, new_paths):
                                        solutions.append(current_edge)
                                        new_matrices.append(new_matrix)
                                else:
                                        break
                        else:
                                break
```

```
    solution = max(solutions)
    return [solution, new_matrices[solutions.index(solution)]]
```

Dla wcześniej ustalonego grafu uzyskano rozwiązanie:

$$8$$

Graf można wtedy przedstawić w następującej postaci:



Czas potrzebny do wykonywania tego programu wynosił *0,019s*.

- **Ciekawostka**:
  Dla większej instancji (200 wierzchołków - około 615 krawędzi) obliczenia zajęły *627,9s*. Jest to czas podobny do czasu wykonywania Brute force'a dla dużo mniejszej instancji (10 wierzchołków - 22 krawędzie).
  Otrzymane rozwiązanie: 451

5.3 Heurystyka

Opracowana heurystyka **łączy** w sobie idee działania *algorytmu siłowego* oraz *algorytmu zachłannego*. Łączy pozytywne cechy każdego z tych dwóch rozwiązań. Dzięki **szybkości** algorytmu aproksymacyjnego czas działania programu jest dużo krótszy, a korzystając z wnikliwości Brute force'a znajdowane rozwiązanie jest **dokładne**.

Heurystyka rozpoczyna działanie od wykonania algorytmu zachłannego dla jednego z wierzchołków. Gdy program wygeneruje graf, który nie spełnia warunków (spójność i limit), pobierane jest ostatnie dobre rozwiązanie. Dla tego rozwiązania tworzona jest permuatacja w postaci liczby binarnej (na miejscach odpowiadających danej krawędzi występuje 1 lub 0, w zależności od tego czy w danym rozwiązaniu krawędź istnieje lub nie). To rozwiązanie **będzie punktem** wyjściowym dla algorytmu Brute force. Ta część programu na podstawie wyjściowego grafu tworzy wszystkie permutacje, które mają ten sam element wspólny z rozwiązaniem podanym przez algorytm zachłanny - usunięte krawędzie. Graf z usuniętymi danymi krawędziami daje rozwiązanie względnie dobre, zatem na tej podstawie można generować pozostałe możliwości.

- Przykład:

  Druga część programu otrzymała permutację (1001101001). Punktem wyjściowym do tworzenia kolejnych permutacji (nie wszystkich możliwych) jest permutacja $(-00-0-00-)$, gdzie w miejsca myślników wstawić można dowolną wartość (krawędź będzie, albo jej nie będzie), natomiast tam gdzie są zera, pewne jest że danych krawędzi nie będzie we wszystkich następnych rozwiązaniach.

Z wygenerowanych możliwości, tworzone są macierze dla grafów. Dalej, na tej strukturze graf badany jest pod kątem spójności i warunku. Spośród wszystkich grafów, które spełniają warunki wybierany jest ten, który ma największą liczbę usuniętych krawędzi.

Dodatkowym ograniczeniem w algorytmie, pozwalającym zmniejszyć czas wykonywania programu, jest limit ilości krawędzi, na których wykonywany będzie brute force. Tutaj ustawiony on został na maksymalnie 70% wszystkich krawędzi. Implikacją tego jest to, że nie dla wszystkich grafów będzie można uzyskać rozwiązanie. Można to zmienić poprzez zmianę tego ograniczenia w programie, co spowoduje jednak wydłużenie czasu wykonywania programu.

Do implementacji heurystyki użyto pomocniczych funkcji:

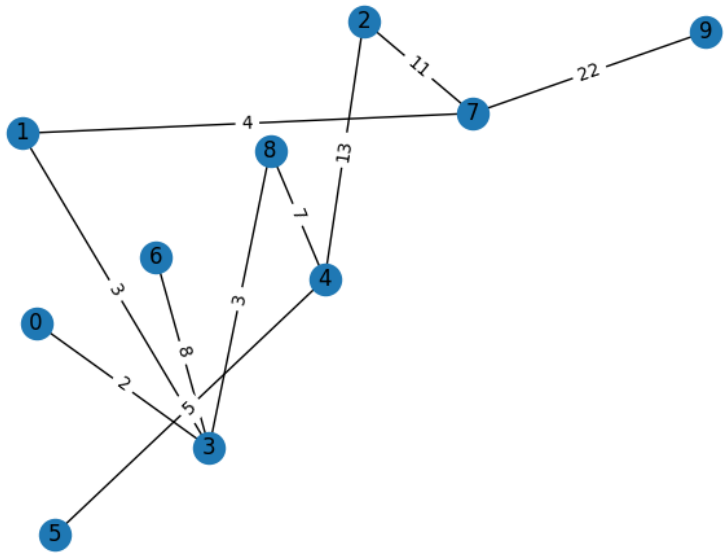
- `def generate_predefined_permutations(perm)` - generuje permutacje, zaczynając od jednej permutacji wyjściowej
- `def compare_matrices(matrix1, matrix2, limit)` - porównuje 2 macierze i na tej podstawie tworzy permutację (które krawędzie zostały usunięte), sprawdza też czy nie przekroczony został limit usuniętych krawędzi

Heurystyka:

```
def delete_edges_heuristics(matrix):
  n = len(matrix)
  edges_count = edges_counter(matrix)
  limit_for_bruteforce = 0.7
  paths = find_shortest_paths(matrix)
  solutions = []
  new_matrices = []

  for v in range(n):
        new_matrix = copy.deepcopy(matrix)
        previous_matrix = copy.deepcopy(new_matrix)
        while 1:
        previous_matrix = copy.deepcopy(new_matrix)
            max_value, ind = 0, 0
        for vert in range(n):
              if new_matrix[v][vert] != 0
                 and new_matrix[v][vert] != math.inf:
                 if new_matrix[v][vert] > max_value:
                                max_value = new_matrix[v][vert]
                                ind = vert
               new_matrix[v][ind] = math.inf
               new_matrix[ind][v] = math.inf

               if DFS_check(new_matrix):
                 new_paths = find_shortest_paths(new_matrix)
                   if check_the_limit(paths, new_paths):
                         previous_matrix = copy.deepcopy(new_matrix)
                         v = ind
                   else:
                     break
               else:
                 break

        permutation = compare_matrices(matrix,
                      previous_matrix,limit_for_bruteforce)
        if not permutation:
          continue
        else:
          edges_permutations = generate_predefined_permutations(permutation)

          for perm in edges_permutations:
```

```
                new_matrix = copy.deepcopy(matrix)
                counter = make_matrix_from_permutation(new_matrix,
                                                       perm,n)

                if DFS_check(new_matrix):
                  new_paths = find_shortest_paths(new_matrix)
                  if check_the_limit(paths, new_paths):
                    solutions.append(counter)
                        new_matrices.append(new_matrix)

    if len(solutions) == 0:
      return "W tym grafie nie mozna zastosowac tego algorytmu"
    else:
      solution = max(solutions)
      return [solution, new_matrices[solutions.index(solution)]]
```

Uzyskany graf wtedy przedstawić w następującej postaci:



Graf jest taki sam, jak wcześniejszy, wytworzony przez algorytm siłowy.

W wyniku działania tego algorytmu uzyskuje się rozwiązanie optymalne (pod warunkiem, że po przejściu zachłannym nie trzeba będzie przekroczyć limitu dla brute force'a w tej metodzie):

12

Czas potrzebny do wykonywania tego programu wynosił *1,93s*. Pokazuje to, jak bardzo zoptymalizowany został czas wykonywania programu.

6 Wnioski

Algorytmem, który daje nam stuprocentową pewność odnośnie optymalności rozwiązania jest **algorytm siłowy**. Niezależnie od żadnych czynników, dla każdego grafu daje optymalne rozwiązanie zadanego problemu. Problemem jaki się pojawia przy tej metodzie jest złożoność obliczeniowa. Czas w jakim wykonuje się program rośnie wykładniczo, przez co jednostka obliczeniowa nie zwróci nam rozwiązania w „skończonym" czasie. Przez to Brute Force jest wykorzystywany jedynie do małych instancji problemu.

Z tych powodów warto jest zastosować **heurystykę**, która również daje optymalne rozwiązanie, w dużo krótszym czasie. Jednakże przez ograniczenie dotyczące wykonywania algorytmu siłowego (jako podprogramu heurystyki) niemożliwe jest zastosowanie tego algorytmu do wszystkich instancji problemu. Manipulując tym ograniczeniem można zwiększyć ilość instancji, na których można zastosować algorytm, jednak nastąpi to kosztem czasu wykonywania programu.

Algorytm zachłanny jest zatem kompromisem pomiędzy optymalizacją czasu wykonywania obliczeń, a optymalizacją głównego problemu. Nie zapewnia on otrzymania najlepszego rozwiązania, ale jeśli jesteśmy w stanie zgodzić się na pewną niedokładność, to nadrabia to swoją szybkością.

Literatura

[1] Algorytm Dijkstry https://pl.wikipedia.org/wiki/Algorytm_Dijkstry

[2] Wyszukiwanie wyczerpujące https://pl.wikipedia.org/wiki/Wyszukiwanie_wyczerpujace