Projekt - nr 3

Mateusz Wardawa [1], Mateusz Siwy [1], Maciej Leśniak [1], and Igor Sala [1]

[1]**Faculty of Physics and Applied Computer Science, AGH University of Science and Technology**

ABSTRACT

Treść Zadań
1.
Korzystajac z programów z poprzednich zestawów wygenerować spójny graf losowy. Przypisać każdej krawedzi tego grafu losowa wage bedaca liczba naturalna z zakresu 1 do 10
2.
Zaimplementować algorytm Dijkstry do znajdowania najkrótszych ścieżek od zadanego wierzchołka do pozostałych wierzchołków i zastosować go do grafu z zadania pierwszego, w którym wagi krawedzi interpretowane sa jako odległości wierzchołków. Wypisać wszystkie najkrótsze ścieżki od danego wierzchołka i ich długości
3.
Wyznaczyć macierz odległości miedzy wszystkimi parami wierzchołków na tym grafie
4.
Wyznaczyć centrum grafu, to znaczy wierzchołek, którego suma odległości do pozostałych wierzchołków jest minimalna. Wyznaczyć centrum minimax, to znaczy wierzchołek, którego odległość do najdalszego wierzchołka jest minimalna.
5.
Wyznaczyć minimalne drzewo rozpinajace (algorytm Prima lub Kru skala).

Keywords: Graf spójny, Dijkstra, macierz odległości, centrum grafu, centrum minimax, Kruskal

ZADANIE 1

Została stworzona klasa ConnectedGraph reprezentujaca spójny graf losowy.

- wybierana jest losowa wartość k (stopień wierzchołków) i tworzony jest graf regularny o n wierzchołkach i stopniu k.
- Graf tworzony jest za pomoca funkcji $regular\_graph(n,k)$ a spójność jest sprawdzana funkcja $is\_connected(G)$. Obie funkcje pochodza z Projekt 2.
- Prób tworzenia grafu może być maksymalnie 10 000 jeśli sie nie uda zgłaszany jest TimeoutError.
- Dla każdej krawedzi w grafie przypisywana jest losowa waga przedziału $[1,10]$
- Na końcu wizualizacja jest zapisywana do pliku

```python
class ConnectedGraph:
    """
    Klasa reprezentujaca graf spójny, wazony

    Attributes:
        G (nx.Graph): Obiekt grafu z biblioteki NetworkX
        matrix_weights (np.array): Macierz wag grafu
    """
```

```
    def __init__(self,n):
        """
        Inicjalizacja garfu

        Args:
            n (int): Liczba wierzchołków grafu.
        """
        k = max(2, np.random.randint(2, n - 2))
        #print("k = ", k)
        self.G = self.__generate_connected_graph(n,k)
        self.matrix_weights = nx.to_numpy_array(self.G, weight="weight")

    def __generate_connected_graph(self,n,k):
        """
        Prywatna metoda generujaca spójny graf regularny.

        Args:
            n (int): Liczba wierzchołków
            k (int): Stopień każdego wierzchołka

        Returns:
            nx.Graph: Wygenerowany graf spójny z losowymi wagami

        Raises:
            TimeoutError: Jeśli przekroczono maksymalna liczbe prób
        """
        it=0
        max_it=10_000
        while it<max_it:
            it+=1
            G=regular_graph(n,k)
            if G is not None and is_connected(G):
                break

        if it >= max_it:
            raise TimeoutError("nie można utowrzyć grafu spójnego.
                Przekroczono limit prób")

        self.__add_random_weights(G)

        return G

    def __add_random_weights(self,G,min_weight=1,max_weight=10):
        """
        Dodaje losowe wagi do krawedzi grafu.

        Args:
            G (nx.Graph): Graf do którego dodajemy wagi
            min_weight (int): Minimalna waga krawedzi (domyślnie 1)
            max_weight (int): Maksymalna waga krawedzi (domyślnie 10)
```

```
        """
        for u,v in G.edges():
            G.edges[u, v]['weight'] = np.random.randint(min_weight,
                max_weight)

    def draw(self):
        """
        Wizualizacja grafu z wagami krawedzi.

        Zapisuje wykres do pliku 'connected_graph.png'.
        """
        path="examples"
        if not os.path.exists(path):
            os.makedirs(path)

        pos = nx.circular_layout(self.G)
        plt.figure(figsize=(6,6))
        nx.draw(self.G, pos, with_labels=True, node_color="lightblue",
            edge_color="gray", node_size=1500, font_size=12,
            font_weight="bold")
        edge_labels = {(u,v): self.G[u][v]['weight'] for u, v in
            self.G.edges}
        nx.draw_networkx_edge_labels(self.G, pos, edge_labels=edge_labels,
            font_color="red", font_size=12)
        plt.savefig(path+'/connected_graph.png')
```

WYNIK

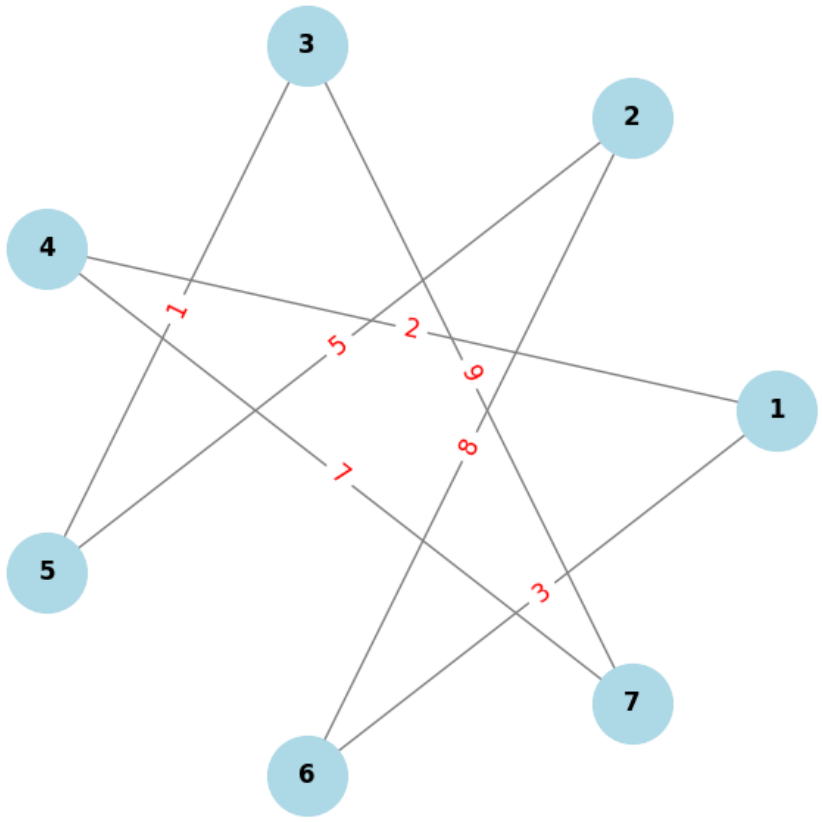

Figure 1. Graf spójny

ZADANIE 2

Została zaimplemenowany algorytm Dijkstry do znajdowania najkrótszych ścieżek od zadanego wierzchołka do pozostałych wierzchołków

- Funkcja init $->$ Inicjalizuje słowniki odległości i poprzedników dla algorytmu Dijkstry.
- Funkcja relax $->$ służy do relaksacji krawedzi.
- Funkcja dijkstra $->$ Oblicza najkrótsze ścieżki z wierzchołka źródłowego

```
def init(G,s):
    """
    Inicjalizacja s␣lowników odleg␣lości i poprzedników dla algorytmu
        Dijkstry.

    Args:
        G (nx.Graph): Graf wejściowy
        s (int): Wierzcho␣lek startowy

    Returns:
```

```
        tuple: (ds, ps) gdzie:
            ds (dict): S␣lownik najkrótszych odleg␣lości
            ps (dict): S␣lownik poprzedników
    """
    ds = {v: float('inf') for v in G.nodes()}
    ps = {v: None for v in G.nodes()}
    ds[s]=0
    return ds,ps

def relax(u,v,ds,ps,w):
    """
    Relaksacja krawedzi (u, v) o wadze w.

    Args:
        u (int): Wierzcho␣lek źród␣lowy
        v (int): Wierzcho␣lek docelowy
        ds (dict): Aktualne oszacowania odleg␣lości
        ps (dict): Aktualni poprzednicy
        w (float): Waga krawedzi miedzy u i v
    """
    if ds[v] > ds[u] +w:
        ds[v] = ds[u]+ w
        ps[v]=u

def dijkstra(G,s):
    """
    Oblicza najkrótsze ścieżki z wierzcho␣lka źród␣lowego s

    Args:
        G (nx.Graph): Graf wejściowy (wagi krawedzi musza być nieujemne)
        s (int): Wierzcho␣lek źród␣lowy

    Returns:
        tuple: (ds, ps) gdzie:
            ds (dict): Najkrótsze odleg␣lości od s do wszystkich
                wierzcho␣lków
            ps (dict): Poprzedniki w najkrótszych ścieżkach

    Note:
        Wszystkie wagi krawedzi musza być nieujemne
    """
    ds,ps=init(G,s)
    S=set()
    while len(S) != len(G.nodes()):
        u =min((v for v in G.nodes() if v not in S),key=lambda v:ds[v]) #
            powoduje czas dzia␣lania O(n^2) ,kolejka priorytetowa
            poprawi␣la by wydajność
        S.add(u)
        for v in G.neighbors(u):
            if v not in S:
```

```
                edge_data= G.get_edge_data(u, v)
                w = edge_data['weight'] if edge_data is not None else 0 #
                    jesli krawedź miedzy wierzcho␣lkami nie istnieje to w =
                    0
                relax(u,v,ds,ps,w)

    return ds,ps
```

WYNIK

d(1) = 0
d(2) = 11
d(3) = 17
d(4) = 2
d(5) = 16
d(6) = 3
d(7) = 9

ZADANIE 3

Stworzona została funkcja *create_distance_matrix*(G) która tworzy macierz odległości miedzy wszystkimi parami wierzchołków poprzez wywołanie algorytmu dijkstry dla każdego wierzchołka.

```
def create_distance_matrix(G):
    """
    Tworzy macierz odleg␣lości miedzy wszystkimi parami wierzcho␣lków.

    Args:
    G (nx.Graph): Graf wejściowy

    Returns:
        list: Dwuwymiarowa lista, gdzie matrix[i][j] = odleg␣lość miedzy
            wierzcho␣lkiem i+1 a j+1
    """
    distance_matrix=[]

    for v in G.nodes():
        ds,ps=dijkstra(G,v)
        distance_matrix.append([ds[key] for key in ds])

    return distance_matrix
```

WYNIK

$$\begin{bmatrix} 0 & 11 & 17 & 2 & 16 & 3 & 9 \\ 11 & 0 & 6 & 13 & 5 & 8 & 15 \\ 17 & 6 & 0 & 16 & 1 & 14 & 9 \\ 2 & 13 & 16 & 0 & 17 & 5 & 7 \\ 16 & 5 & 1 & 17 & 0 & 13 & 10 \\ 3 & 8 & 14 & 5 & 13 & 0 & 12 \\ 9 & 15 & 9 & 7 & 10 & 12 & 0 \end{bmatrix}$$

ZADANIE 4

napisano dwie funkcje *find_the_centre_of_the_graph*(*distance_matrix*), która wyznacza wierzchołek, którego suma odległości do pozostałych wierzchołków jest minimalna.Oraz *find_minmax*(*distance_matrix*) która wyznacza wierzchołek, którego odległość do najdalszego wierzchołka jest minimalna.

```python
def find_the_centre_of_the_graph(distance_matrix):
    """
    Znajduje centrum grafu

    Args:
        distance_matrix (list): Macierz odległości

    Returns:
    tuple: (centrum, min_distance) gdzie:
            centrum (int): Numer wierzchołka o minimalnej sumie
                odległości
            min_distance (int): Minimalna suma odległości
    """
    sum_of_distances = [sum(row) for row in distance_matrix]
    min_distance =min(sum_of_distances)

    return sum_of_distances.index(min_distance) + 1 , min_distance

def find_minmax(distance_matrix):
    """
    Znajduje centrum minimax grafu

    Args:
        distance_matrix (list): Macierz odległości

    Returns:
        tuple: (centrum, min_distance) gdzie:
                centrum (int): Numer wierzchołka
                min_distance (int):  odległość
    """
    max_distances = [max(row) for row in distance_matrix]
    min_distance = min(max_distances)

    return max_distances.index(min_distance)+1,min_distance
```

WYNIK

centrum grafu = 6 (suma odleglosci: 55)
Centrum minimax = 6 (odleglosc od najdalszego: 14)

ZADANIE 5

Stworzono algorytm kruskala do znajdowania minimalnego drzewa rozpinajacego. Wykorzystano strukure DSU, która pozwala szybko sprawdzać, czy dwa wierzchołki należa do tego samego drzewa (czy sa już połaczone), i łaczyć zbiory.

- __init__ $->$ Inicjalizacja struktury DSU.
- find $->$ Zwraca reprezentanta zbioru, do którego należy element i.
- union $->$ Łaczy dwa różne zbiory.
- kruskal $->$ Znajduje minimalne drzewo rozpinajace

```python
class DSU:
    """
    Struktura danych Disjoint Set Union.
    Uzywana do efektywnego zarzadzania zbiorami roz␣lacznymi i znajdowania
        ich reprezentantów.
    """
    def __init__(self,n):
        """
        Inicjalizacja struktury DSU.

        Args:
            n (int): Liczba elementów (wierzcho␣lków grafu)
        """
        self.parent = list(range(n))
        self.rank= [1] * n

    def find(self,i):
        """
        Znajduje reprezentanta zbioru, do którego nalezy element i.

        Args:
            i (int): Indeks elementu

        Returns:
            int: Reprezentant zbioru zawierajacego i
        """
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])

        return self.parent[i]

    def union(self,x,y):
        """
```

```
        ␣Laczy zbiory zawierajace elementy x i y.

        Args:
            x (int): Pierwszy element
            y (int): Drugi element

        Returns:
            bool: True jeśli zbiory zosta␣ly po␣laczone, False jeśli juz
                by␣ly w tym samym zbiorze
        """
        s1 =self.find(x)
        s2 =self.find(y)

        if s1 != s2:
            if self.rank[s1] < self.rank[s2]:
                self.parent[s1]=s2
            elif self.rank[s1] > self.rank[s2]:
                self.parent[s2] = s1
            else:
                self.parent[s2] = s1
                self.rank[s1] +=1
            return True
        return False


def kruskal(matrix_weights):
    """
    Algorytm Kruskala znajdowania minimalnego drzewa rozpinajacego (MST) w
        grafie.

    Args:
        matrix_weights (list): Macierz wag. Wartość 0 oznacza brak
            krawedzi.

    Returns:
        list: Lista krawedzi MST w formacie (u, v, w), gdzie:
              - u, v to numery wierzcho␣lków
              - w to waga krawedzi

    Note:
        Wagi krawedzi musza być nieujemne
    """

    edges=[]
    n =len(matrix_weights)
    for row in range(n):
        for column in range(row+1,n):
            if matrix_weights[row][column] != 0:
                edges.append((matrix_weights[row][column], row, column))

```

```
    edges.sort()
    dsu = DSU(n)
    minimum_spannig_tree = []

    for weight,u,v in edges:
        if dsu.union(u,v):
            minimum_spannig_tree.append((u+1,v+1,weight))
        if len(minimum_spannig_tree) == n-1:
            break

    return minimum_spannig_tree
```

WYNIK

minimalne drzewo rozpinajace [(3, 5, 1.0), (1, 4, 2.0), (1, 6, 3.0), (2, 5, 5.0), (4, 7, 7.0), (2, 6, 8.0)]