Projekt - nr 6

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.
Zaimplementować algorytm PageRank dla digrafu. Zastosować dwie poniższe metody i porównać wyniki. (a) Metoda polegajaca na przechodzeniu od wierzchołka do sasiedniego wierzchołka za pomoca błądzenia przypadkowego z prawdopodobieństwem 1d i teleportacji z prawdopodobieństwem d. Przyjąć d = 0.15. PageRank wyliczyć jako czestość odwiedzin danego wierzchołka. (b) Metoda iteracji wektora obsadzeń pt . Dla t = 0 przyjąć p0 = (1/n, . . . , 1/n), a nastepnie powtarzać iteracyjnie obliczenie pt+1 = ptP, dla t = 1, 2, . . ., gdzie P jest macierza stochastyczna postaci Pij = (1 d)Aij/di + d/n, a dj jest stopniem wyjściowym wierzchołka j, a Aij macierza sasiedztwa. PageRank wylicza sie jako wartości elementów wektora obsadzeń po wielu interacjach. Jeżeli te wartości sie zmieniaja w czasie, to PageRank wylicza sie jako średnie tych elementów.
2.
Zaimplementować algorytm do wyszukiwania możliwie najkrótszej zamknietej drogi przechodzacej przez wszystkie zadane wierzchołki rozrzucone na planszy kwadratowej. Zastosować metode symulowanego wyżarzania oparta o łańcuch Markowa, którego pojedyncze kroki sa wykonywane jako operacje 2-opt zgodnie z algorytmem Metropolisa-Hastingsa.

Keywords: Symulowane wyszarzane, łańcuch markowa, Metropolis-Hastings

ZADANIE 1

```python
import numpy as np

graph = {
    'A': ['E', 'F', 'I'],
    'B': ['A', 'C', 'F'],
    'C': ['B', 'D', 'E', 'L'],
    'D': ['C', 'E', 'H', 'I', 'K'],
    'E': ['C', 'G', 'H', 'I'],
    'F': ['B', 'G'],
    'G': ['E', 'F', 'H'],
    'H': ['D', 'G', 'I', 'L'],
    'I': ['D', 'E', 'H', 'J'],
    'J': ['I'],
    'K': ['D', 'I'],
    'L': ['A', 'H']
}

nodes = sorted(list(graph.keys()))
n = len(nodes)
node_to_index = {node: i for i, node in enumerate(nodes)}
index_to_node = {i: node for i, node in enumerate(nodes)}
```

```python

d = 0.15

def pagerank_random_walk(graph, nodes, d, num_steps=1000000):
    visits = {node: 0 for node in nodes}
    current_node = np.random.choice(nodes)

    for _ in range(num_steps):
        visits[current_node] += 1

        if np.random.rand() < d:
            current_node = np.random.choice(nodes)
        else:
            if graph[current_node]:
                current_node = np.random.choice(graph[current_node])
            else:
                current_node = np.random.choice(nodes)

    total_visits = sum(visits.values())
    pagerank = {node: count / total_visits for node, count in
        visits.items()}
    return pagerank

print("--- PageRank z teleportacja ---")
pr_random_walk = pagerank_random_walk(graph, nodes, d)
sorted_pr_rw = sorted(pr_random_walk.items(), key=lambda item: item[1],
    reverse=True)
for node, pr_value in sorted_pr_rw:
    print(f"{node} ==> PageRank = {pr_value:.6f}")

def pagerank_power_iteration(graph, nodes, d, max_iterations=100,
    tolerance=1e-7):
    n = len(nodes)
    node_to_index = {node: i for i, node in enumerate(nodes)}

    pr_vector = np.full(n, 1/n)

    P = np.zeros((n, n))
    for i, u in enumerate(nodes):
        out_links = graph[u]
        out_degree = len(out_links)

        for j, v in enumerate(nodes):
            if v in out_links:
                P[j, i] = (1 - d) / out_degree
            P[j, i] += d / n

        if out_degree == 0:
            for j in range(n):
                P[j, i] = 1/n
```

```
    for iteration in range(max_iterations):
        new_pr_vector = np.dot(P, pr_vector) # p_t+1 = P * p_t
        if np.linalg.norm(new_pr_vector - pr_vector, ord=1) < tolerance:
            break
        pr_vector = new_pr_vector
    else:
        print("nie udalo sie")

    pagerank = {index_to_node[i]: pr_vector[i] for i in range(n)}
    return pagerank

print("\n--- PageRank z wektorem obsadzen ---")
pr_power_iteration = pagerank_power_iteration(graph, nodes, d)
sorted_pr_pi = sorted(pr_power_iteration.items(), key=lambda item: item[1],
    reverse=True)
for node, pr_value in sorted_pr_pi:
    print(f"{node} ==> PageRank = {pr_value:.6f}")
```

```
--- PageRank z teleportacja ---
I ==> PageRank = 0.150664
H ==> PageRank = 0.140303
E ==> PageRank = 0.120486
D ==> PageRank = 0.102948
G ==> PageRank = 0.098363
C ==> PageRank = 0.072191
F ==> PageRank = 0.071644
B ==> PageRank = 0.057916
L ==> PageRank = 0.057680
A ==> PageRank = 0.053409
J ==> PageRank = 0.044596
K ==> PageRank = 0.029800

--- PageRank z wektorem obsadzen ---
I ==> PageRank = 0.150941
H ==> PageRank = 0.139879
E ==> PageRank = 0.120326
D ==> PageRank = 0.102308
G ==> PageRank = 0.098443
F ==> PageRank = 0.072116
C ==> PageRank = 0.072024
B ==> PageRank = 0.058454
L ==> PageRank = 0.057529
A ==> PageRank = 0.053512
J ==> PageRank = 0.044575
```

```
K ==> PageRank = 0.029892
```

ZADANIE 2

Zaimplementowany został algorytm symulowanego wyszarzania oparta o łańcuch Markowa gdzie pojedyncze kroki to operacje 2-opt zgodnie z algorytmem Metropolisa-Hastingsa.

- MAX_IT = 200 000
- ITERATIONS = 10 (ponowne wywołania algorytmu)
- T = $0.001 * i^2$ (funkcja chłodzenia)

Algorithm 1 Algorytm symulowanego wyżarzania

Wyznacz dowolny cykl startowy P
for $i \leftarrow 100$ downto 1 **do** ▷ Petla chłodzenia.
  $T \leftarrow 0.001i^2$ ▷ Aktualna temperatura.
  for $it \leftarrow 0$ to MAX_IT **do** ▷ Petla po iteracjach dla ustalonej T.
    Wylosuj (a,b) oraz (c,d) należace do cyklu P ▷ Patrz: Uwagi poniżej.
    Utwórz cykl $P_{\text{new}} \leftarrow P$
    W cyklu P_{new} zamień (a,b) i (c,d) na (a,c) i (b,d)
    if $d(P_{\text{new}}) < d(P)$ **then**
      $P \leftarrow P_{\text{new}}$ ▷ Jeśli nowy cykl jest krótszy: akceptujemy go.
    else
      $r \leftarrow \text{rand}(0,1)$ ▷ Losowanie liczby rzeczywistej z przedziału [0, 1].
      if $r < \exp\left(-\frac{d(P_{\text{new}})-d(P)}{T}\right)$ **then**
        $P \leftarrow P_{\text{new}}$ ▷ Akceptacja dłuższego cyklu z prawd. zależnym od T.
      end if
    end if
  end for
end for
return P

```python
MAX_IT = 200_000
ITERATIONS = 10

def read_file(filename):
    """
    Czyta plik z zapisanym grafem

    Args:
    -----
    filename : str
        nazwa pliku

    Returns
    -------
    numpy array 2D
```

```
        zwraca liste 2D z współrzednymi wierzchołków
    """
    graph=[]
    with open(filename,"r") as file:
        for line in file:
            graph.append([int(x) for x in line.split()])
    graph=np.array(graph)
    return graph

def draw(graph,avgs,stds,bests,length):
    """
    Rysuje graf

    Args:
    -----
    graph : numpy array 2D
        lista 2D z współrzednymi wierzchołków
    avgs: numpy array
        przechowuje średnia długości cyklu z każdej iteracji
    stds :  numpy array
        przechowuje odchylenia standardowe z każdej iteracji
    bests : numpy array
        przechowuje najkrótsze długości cyklu każdej iteracji
    length : float
        długość cyklu

    """
    fig, axs = plt.subplots(2, 2, figsize=(12, 7))
    graph = np.vstack([graph, graph[0]])
    axs[0,0].plot(graph[:,0],graph[:,1],'-o',c='purple')
    axs[0,0].text(90,-1,f"{length:.3f}",fontsize = 12,weight='bold')

    axs[0,1].plot(avgs, label='Średnia wartość', linewidth=2,c='orange')
    axs[0,1].set_title("Średnia wartości ")
    axs[0,1].set_xlabel("Iteracja")
    axs[0,1].set_ylabel("Średnia wartość")
    axs[0,1].grid(True)
    axs[0,1].legend()

    axs[1,0].plot(stds, label='Odchylenie standardowe',
        linewidth=2,c='green')
    axs[1,0].set_title("Odchylenia standarowe ")
    axs[1,0].set_xlabel("Iteracja")
    axs[1,0].set_ylabel("Odchylenie standardowe")
    axs[1,0].grid(True)
    axs[1,0].legend()

    axs[1,1].plot(bests, label='Najkrótsze sciezki w każdej iteracji',
        linewidth=2,c='r')
    y = min(bests)
```

```
    x = np.where(bests == y)[0]
    axs[1,1].scatter(x,y, c= 'black')
    axs[1,1].set_title("Najkrótsze sciezki")
    axs[1,1].set_xlabel("Iteracja")
    axs[1,1].set_ylabel("Najkrótsze sciezki")
    axs[1,1].grid(True)
    axs[1,1].legend()

    plt.tight_layout()
    plt.show()

@njit
def vector_length(v1, v2):
    """
    oblicza d␣lugość krawedzi

    Args:
    -----
    v1, v2 : list
        wierzcho␣lki krawedzi. gdzie v to [x,y]

    Returns:
    --------
    float
        d␣lugość odcinka

    """
    return ((v2[0] - v1[0])**2 + (v2[1] - v1[1])**2)**0.5

@njit
def randomize_edges(graph,length):
    """
    Wykonuje operacje optymalizacyjna 2-opt

    Args:
    -----
    graph : numpy array 2D
        lista 2D z współrzednymi wierzcho␣lków
    length : float
        d␣lugość cyklu

    Returns:
    --------
    new_graph : numpy array 2D
        lista 2D z współrzednymi wierzcho␣lków po optymalizacji
    new_length : float
        d␣lugość cyklu po optymalizacji
    """

    new_graph = graph.copy()
```

```
    while True:
        b, c = np.sort(np.random.choice(len(graph), size =2
            ,replace=False))
        a = b-1 if b != 0 else len(graph) - 1
        d = c+1 if c != len(graph) - 1 else 0
        if a != c and a != d and b != c and b != d:
            break

    new_graph[b:c+1] = new_graph[b:c+1][::-1]
    old_edges = vector_length(graph[a], graph[b]) + vector_length(graph[c],
        graph[d])
    new_edges = vector_length(new_graph[a], new_graph[b]) +
        vector_length(new_graph[c], new_graph[d])
    new_length = length - old_edges + new_edges

    return new_graph, new_length


@njit
def calcluate_length(graph):
    """
    Oblicza d␣lugość grafu

    Args:
    -----
    graph : numpy array 2D
        lista 2D z współrzednymi wierzcho␣lków

    Returns:
    --------
    s : float
        d␣lugość grafu
    """
    s=0
    for idx in range(len(graph) - 1):
        s+=vector_length(graph[idx],graph[idx+1])
    s+=vector_length(graph[0],graph[-1])
    return s

@njit
def simulated_annealing(graph,d,lengths,best_graph):
    """
    implementacja  symulowanego wyzarzania

    Args:
    -----
    graph : numpy array 2D
        lista 2D z współrzednymi wierzcho␣lków
    d : float
        d␣lugość grafu
```

```
    lengths : numpy array
        przechowuje wszystkie znalezione odleg␣lość w grafie
    best_graph : numpy array 2D
        graf o najkrótszej d␣lugości

    Returns:
    --------
    graph : numpy array 2D
        lista 2D reprezentujaca znaleziony cykl Hamiltona o
            zminimalizowanej d␣lugości.
    d : float
        d␣lugość grafu
    best_graph : numpy array 2D
        graf o najkrótszej d␣lugości
    """
    lengths[0]=d
    T = 100
    for i in range(100,0,-1):
        T = 0.001 * i**2
        for it in range(MAX_IT):
            new_graph, d_new = randomize_edges(graph,d)
            lengths[(100-i)*MAX_IT +it] = d_new
            if  d_new < d:
                graph=new_graph
                d = d_new
                if calcluate_length(best_graph) > d:
                    best_graph=graph.copy()
            else:
                r = np.random.rand()
                if r < np.exp(-(d_new -d)/T):
                    graph = new_graph
                    d = d_new

    return graph,d,best_graph

@njit(nogil=True)
def main_loop(graph,progress,avgs,stds,bests,best_graph):
    """
    g␣lówna petla programu

    Args:
    -----
    graph : numpy array 2D
        lista 2D reprezentujaca
    progress
        obiekt s␣luzacy do wyświetlania postepu wykonywania petli.
    avgs : numpy array
        przechowuje średnia d␣lugości cyklu z kazdej iteracji
    stds :  numpy array
        przechowuje odchylenia standardowe z kazdej iteracji
```

```
    bests : numpy array
        przechowuje najkrótsze długości cyklu każdej iteracji
    best_graph : numpy array 2D
        graf o najkrótszej długości

    Returns:
    --------
    graph : numpy array 2D
        lista 2D reprezentujaca znaleziony cykl Hamiltona o
            zminimalizowanej długości.
    d : float
        długość grafu
    best_graph : numpy array 2D
        graf o najkrótszej długości
    """
    d=calcluate_length(graph)
    for i in range(ITERATIONS):
        lengths = np.empty(100*MAX_IT)
        graph,d,best_graph=simulated_annealing(graph,d,lengths,best_graph)
        avgs[i] = np.mean(lengths)
        stds[i] = np.std(lengths)
        bests[i] = d
        progress.update(1)

    return graph,d,best_graph

if __name__ == "__main__":
    graph=read_file('data.csv')
    avgs = np.empty(ITERATIONS)
    stds = np.empty(ITERATIONS)
    bests = np.empty(ITERATIONS)
    best_graph = graph.copy()
    with ProgressBar(total=ITERATIONS) as progress:
        graph,d,best_graph =
            main_loop(graph,progress,avgs,stds,bests,best_graph)
    draw(best_graph,avgs,stds,bests,calcluate_length(best_graph))
```

WYNIK

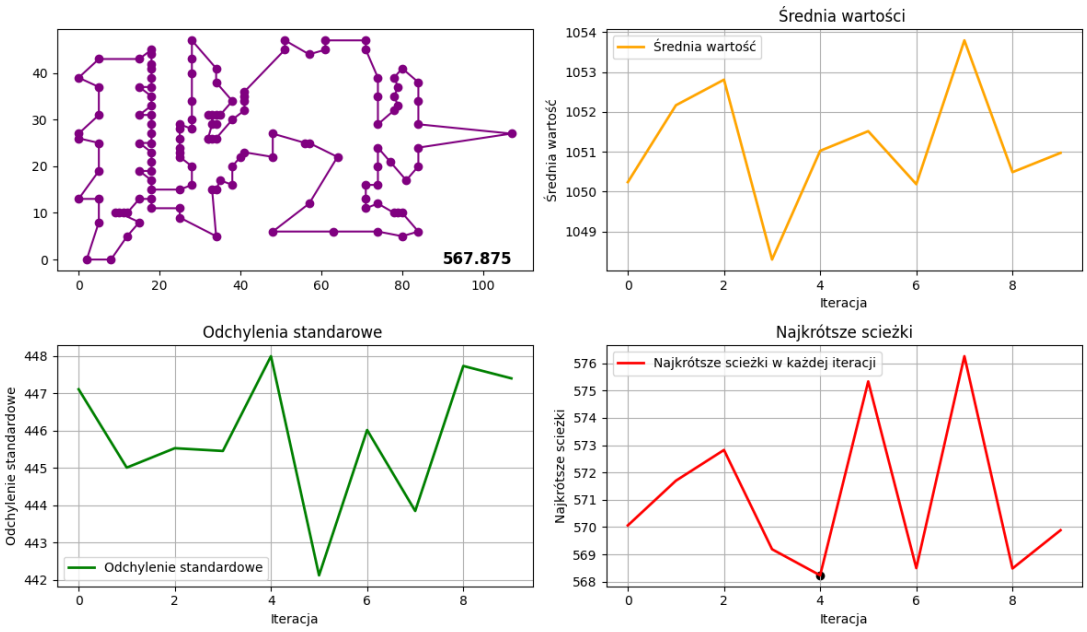

Figure 1. Wynik