

Jakub Goleń nr. 169781
Anna Turek nr. 169856
Przemysław Litwa nr 169809

Sprawozdanie z projektu

Projektowanie Systemów i Sieci Komputerowych

Algorytm LEACH

1. Wstęp teoretyczny

CH - cluster head

WSN - Wireless Sensor Network

Aby lepiej wyjaśnić specyfikę algorytmu LEACH, najpierw omówimy WSN (Wireless Sensor Network - Bezprzewodową sieć czujników). Bezprzewodowa sieć czujników (WSN) jest platformą komunikacyjną, na którą może wpływać wszystko, co ma potencjał kilku funkcji komunikacji informacyjnej w przyszłości. Do tej pory WSN traktowano poważnie przez badaczy ze względu na liczne zastosowania w wielu dziedzinach ludzkiej działalności.

WSN zależy od kilku małych jednorazowych niezależnych urządzeniach zwanych węzłami czujnikowymi, tworzących sieć. Specyficzne węzły w WSN mogą wykrywać środowisko, przetwarzać wykryte dane lub wysyłać je do jednostki centralnej przetwarzanej przez łącze bezprzewodowe.

Dziennie zapotrzebowanie na WSN stale rośnie, począwszy od zastosowań wojskowych do krajowych, naziemnych i kosmicznych. WSN powstał w wyniku zmian w technologii mikro schematów elektromechanicznych (MEMS) oraz w komunikacji bezprzewodowej. WSN stały się ciekawym obszarem badań. Składają się z kilku węzłów czujników (bezprzewodowych), które łączą się, tworząc pole czujnika i zlew. Głównymi problemami w WSN są: duże liczby używanych węzłów, ich niska moc znamionowa oraz ich ograniczenie do komunikacji na krótkie odległości. Te węzły współpracują ze sobą, aby zapewnić wykrywanie, śledzenie i transmisję informacji, tworząc czujniki bezprzewodowe odpowiednie do monitorowania zjawisk naturalnych i zmian środowiskowych, szacowania ruchu ulicznego, kontrolowania bezpieczeństwa i monitorowania wezwań wojskowych. Te aplikacje wymagają wysokiej niezawodności sieci sensorowych oraz niezawodności sensora sieci, ostatnie badania koncentrowały się na heterogenicznych sieciach WSN.

Naukowcy mają szeroko pogrupowane węzły czujników w klastry w kierunku osiągnięcia celu, jakim jest skalowalność sieci; każda grupa ma CH, który jest wybierany przez inne klastry lub jest wstępnie wyznaczony przez sieć developer. Podobnie każdy czujnik, który jest bogatszy w zasoby, również ma możliwość bycia CH. Istnieje kilka korzyści tworzenia klastrów, a największym z nich jest wdrożenie wzmocnionej strategii organizacji, która dodatkowo wydłuża żywotność baterii czujników i dodatkowo poprawia żywotność sieci.

Algorytm LEACH (algorytm zachłanny) został opracowany jako metoda grupowania Wireless Sensor Network (bezprzewodowych sieci czujników), aby zapewnić równomierne rozproszenie energii w całej sieci. LEACH został opracowany w celu zapewnienia zrównoważonego wykorzystania energii poprzez losową rotację CH. Procedura jest zorganizowana w taki sposób, że szybkość propagacji danych może zostać zmniejszona przez dane połączenie/fuzję. Wybór CH w algorytmie LEACH odbywa się dynamicznie we wszystkich interwałach.

Ta grupa CH opiera się wyłącznie na niezdolności innych niezależnych węzłów do posiadania większej liczby zasobów niż wybrana grupa CH, ponieważ opiera się na proporcjach optymalnych CH w sieci; to również zależy od tego, jak często oczekiwany węzeł stał się wcześniej CH.

W LEACH funkcja $T(n)$ jest oznaczona jako $\frac{P}{1-P(r \bmod P^{-1})}$ tylko wtedy, gdy $n \in G$, albo, $= 0$, gdzie dany węzeł jest reprezentowany jako n , a z góry określone szanse na wybranie węzła jako CH są reprezentowane jako P . Aktualna liczba rund jest reprezentowana jako r , a G oznacza węzły, które nigdy nie zostały wybrane jako CH w ostatniej rundzie $\frac{1}{P}$. Każdy z węzłów wygeneruje dowolną sumę w zakresie od $[0 \ 1]$ do CH. Węzeł zostanie wybrany jako CH, jeśli suma będzie mniejsza od progu $T(n)$. Po wybraniu jako CH, nowy CH będzie rozgłaszał swoją pozycję do sąsiednich węzłów, które następnie określą optymalną CH (w oparciu o minimalną wymaganą energię transmisji) i przekażą swoją chęć bycia w tej grupie.

Aby zminimalizować kolizję, wiadomość nadawana przez CH jest transmitowana przy użyciu Carrier Sense Multiple Access (CSMA). Po nadaniu swojej pozycji CH utworzy harmonogram transmisji, który będzie rozgłaszany do wszystkich węzłów w odpowiednich klastrach. Ten harmonogram transmisji zawiera gniazda dla każdego z sąsiednich węzłów, co pozwala na niskie zużycie energii, ponieważ węzły mogą się przełączać i np. wyłączać radio w chwilach bezczynności. Głównym celem struktur routingu opartych na klastrach jest zachęcanie do niskiego zużycia energii przez węzły sieci w celu zapewnienia dłuższej żywotności usług sieciowych. Organizacja sieci jest zwykle wprowadzana w celu zapewnienia dodatkowej efektywności energetycznej całości sieć. W tej części dokonano przeglądu niektórych protokołów uwzględniających zużycie energii w Wireless Sensor Network.

LEACH został opracowany jako solidne ramy klastrowania dla Wireless Sensor Network, które zależą od oczekiwanego sygnału przez klastry i wykorzystuje CH jako routery, aby dotrzeć do stacji bazowej. Obsługa informacji (w tym łączenie informacji) jest realizowane lokalnie przez klastry. W algorytmie klastry są tworzone przy użyciu procedury rozproszonej, której węzły są tworzone z niezależnych wyborów (brak ingerencji zewnętrznej z jednostki centralnej). Węzeł jest pierwotnie wyznaczony jako CH na podstawie jego wartości prawdopodobieństwa p ; w przypadku węzłów innych niż CH określają swoją grupę, wybierając CH, z którymi można nawiązać połączenie przy użyciu minimalnej energii. Aby zrównoważyć obciążenie w każdym klastrze, rola CH jest okresowo obracana między węzłami w każdym klastrze. Ta odmiana jest wykonywana aby zmusić każdy węzeł do losowego wyboru dowolnego „T” spośród 0 i 1.

2. Działanie algorytmu LEACH w Pythonie

```
1 import pandas as pd
2 import networkx as nx
3 import random
4 import matplotlib.pyplot as plt
5 import math
6 import numpy as np
7
```

Na początku importujemy potrzebne nam biblioteki. Pandas odpowiada za ramki danych, networkx odpowiada za grafy sieciowe, random za liczby pseudolosowe, matplotlib odpowiada za wykresy. Math służy nam do działań matematycznych, a numpy do generowania sekwencji potrzebnej do wykresu.

```

8
9 class Environment:
10     def __init__(self, nodes_amount: int, rounds_amount: int) -> None:
11         self.nodes_amount = nodes_amount
12         self.rounds_amount = rounds_amount
13         self.graph = nx.Graph()
14         self.graph_df = self.generate_df()
15         self.dead = []
16         self.energy = []
17

```

Inicjalizacja klasy Environment.

Przypisanie atrybutów klasy do wewnętrznych atrybutów.

Self.graph jest to na razie pusty graf. Self.graph_df służy nam do przechowywania ramki danych. Self.dead do przechowywania ilości martwych węzłów, self.energy przechowuje sumę energii węzłów w sieci.

```

def generate_df(self):
    """
    method that generate DataFrames which contains network data
    """
    data = []
    for i in range(self.nodes_amount):
        node = f"nodes_{i}"
        pos_x = random.randint(0, self.nodes_amount)
        pos_y = random.randint(0, self.nodes_amount - 10)
        pos = (pos_x, pos_y)
        energy = float(random.randint(300, 1000))
        is_ch = False
        color = "Green"
        is_dead = False
        is_taken = False
        ch = ""
        data.append(
            [node, pos, energy, is_ch, color, is_dead, is_taken, ch]
        )
    data.append(
        ["antenna", (self.nodes_amount/2, self.nodes_amount), float(300000), None, "Black", None, None, ""]
    )
    df = pd.DataFrame(
        data=data, columns=["node", "pos", "energy", "is_ch", "color", 'is_dead', 'is_taken', 'ch'],
        index=[i for i in range(len(data))]
    )
    print("generated dataframe")
    return df

```

Metoda generate_df służy do generowania ramki danych. W ramce danych znajdują się takie wartości jak nazwa węzła, pozycja w grafie, losowa ilość energii od 300 do 1000. Wartość typu boolean sprawdzająca czy węzeł jest CH, kolor, wartość boolean sprawdzająca czy jest martwy, wartość boolean sprawdzająca czy jest już zklastrowany i nazwa CH do którego jest zklastrowany.

```
def add_nodes(self):
    for i in range(self.nodes_amount+1):
        self.graph.add_node(self.graph_df.node[i], pos=self.graph_df.pos[i])

    print('added_nodes')
```

Funkcja add_nodes służy nam do dodawania węzłów do grafu

```
def cluster_head_selection(self):
    def randomOddNumber(a,b):
        a = a // 2
        b = b // 2 - 1
        number = random.randint(a,b)
        number = (number * 2) + 1
        return number

    alive_nodes = self.graph_df.loc[(self.graph_df['is_dead'] == False)]
    random_amount = randomOddNumber(1, 9)
    while random_amount > len(alive_nodes.index.to_list()) and len(alive_nodes.index.to_list()) != 0:
        random_amount = randomOddNumber(1, 9)
    sample = random.sample(alive_nodes.index.to_list(), random_amount)

    for el in sample:
        alive_nodes.loc[el, ['is_ch']] = True
        alive_nodes.loc[el, ['color']] = "Red"

    self.graph_df.loc[(self.graph_df['is_dead'] == False)] = alive_nodes
    print('selected cluster heads')
```

Funkcja cluster-head-selection pozwala nam na wybór losowych CH z węzłów, które nie są martwe.

```
def antenna_edges(self):
    chs = self.graph_df.loc[self.graph_df['is_ch'] == True] #cluster heads
    antenna_pos = self.graph_df.loc[self.graph_df['node'] == "antenna"].pos.values[0] #antenna x, y cords
    antenna_name = "antenna"
    distances = []
    nodes = chs.node.values
    pos = chs.pos.values

    for i in range(len(nodes)):
        distance = math.sqrt(
            (pos[i][0]-antenna_pos[0])**2 + (pos[i][1]-antenna_pos[1])**2
        )
        distances.append([nodes[i], distance])
    distances = sorted(distances, key= lambda x:x[1])

    for i, distance in enumerate(distances):
        self.graph.add_edge(distance[0], antenna_name)
        antenna_name = distance[0]
    print('created edges to ant')
```

Funkcja `antenna_edges` dodaje krawędzie pomiędzy anteną, a cluster headami. Wybór pierwszych w kolejności CH jest zależna od odległości od anteny (oznacza to kolejność wysyłania).

```
def clustering(self):
    chs = self.graph_df.loc[self.graph_df['is_ch'] == True]
    for ch_node, ch_pos in zip(chs.node.values, chs.pos.values):
        random_cluster_child = random.randint(1,6)
        not_chs = self.graph_df.loc[(self.graph_df['is_ch'] == False)
            & (self.graph_df['is_taken'] == False) & (self.graph_df['is_dead'] == False)]
        distances = []
        for node, pos in zip(not_chs.node.values, not_chs.pos.values):
            distance = math.sqrt(
                (pos[0]-ch_pos[0])**2 + (pos[1] - ch_pos[1])**2
            )
            distances.append([node, distance])
        distances = sorted(distances, key= lambda x:x[1])

        for top in distances[0:random_cluster_child]:
            self.graph.add_edge(top[0], ch_node)
            index = not_chs.index[not_chs['node'] == top[0]].tolist()[0]
            not_chs.loc[index, 'is_taken'] = True
            not_chs.loc[index, 'ch'] = ch_node
        self.graph_df.loc[(self.graph_df['is_ch'] == False) & (self.graph_df['is_taken'] == False) & (self.graph_df['is_dead'] == False)] = not_chs
    print('clustered')
```

Funkcja `clustering` służy do klastrowania losowej ilości węzłów z CH. Wybierane są najbliższe dostępne węzły.

```

def energy_consumption(self):
    ch = self.graph_df.loc[self.graph_df['is_ch'] == True]
    antennna_pos = self.graph_df.loc[self.graph_df['node'] == 'antenna'].pos.values[0]
    for ch_node, ch_pos in zip(ch.node.values, ch.pos.values):
        cluster_children = self.graph_df.loc[self.graph_df['ch'] == ch_node]
        #Cluster head power consumption
        distance_ant = math.sqrt(
            (ch_pos[0]-antennna_pos[0])**2 + (ch_pos[1] - antennna_pos[1])**2
        )
        ch_power_consum = len(cluster_children)*distance_ant + distance_ant
        index = self.graph_df.index[self.graph_df['node'] == ch_node].tolist()[0]
        self.graph_df.loc[index, ['energy']] = self.graph_df.loc[index, ['energy']] - ch_power_consum

    #Cluster children power consumption

    nodes_power_consum = []
    for node, pos in zip(cluster_children.node.values, cluster_children.pos.values):
        distance = math.sqrt(
            (pos[0]-ch_pos[0])**2 + (pos[1] - ch_pos[1])**2
        )
        nodes_power_consum.append([node, round(distance)])

    for node, power_consum in nodes_power_consum:
        index = self.graph_df.index[self.graph_df['node'] == node].tolist()[0]
        self.graph_df.loc[index, ['energy']] = self.graph_df.loc[index, ['energy']] - power_consum
    print('energy consume calculated')

```

Funkcja `energy_consumption` służy do obliczenia kosztu energii wysłania danych dla sklastrowanych węzłów. Koszt energii CH wynosi ilość węzłów sklastrowanych (uwzględniając CH) pomnożone przez odległość do anteny. Funkcja także aktualizuje wartości energii w ramce danych.

```

def dead_update(self):
    indexes = self.graph_df.index[
        (self.graph_df['energy'] <= 0.0) & (self.graph_df['is_dead']==False)
    ].tolist()

    for index in indexes:
        self.graph_df.loc[index, ['is_dead']] = True
        print("dead: update info")
        self.graph_df.loc[index, ['color']] = "Blue"
        print("dead: update color")
        self.graph_df.loc[index, ['is_ch']] = False
        print("dead: update isch")
        self.graph_df.loc[index, ['ch']] = ''
        self.graph_df.loc[index, ['energy']] = 0.0
        print("dead: update ch")
        self.graph_df.loc[index, ['is_taken']] = False
    print('dead updated')

```

Funkcja ta aktualizuje informacje o śmierci węzłów w ramce danych.


```
def new_state(self):
    indexes = self.graph_df.index[
        ((self.graph_df['is_ch'] == True) | (self.graph_df['ch'] != ""))
        & (self.graph_df['is_dead'] == False)
    ].tolist()

    for index in indexes:
        self.graph_df.loc[index, 'color'] = "Green"
        self.graph_df.loc[index, 'is_ch'] = False
        self.graph_df.loc[index, 'is_taken'] = False
        self.graph_df.loc[index, 'ch'] = ''
    print('clean_state')
```

Funkcja new_state pozwala nam usunąć informacje o CH i aktualizuje dane w ramce.

```
def not_all_dead(self):
    not_dead = self.graph_df.loc[self.graph_df['is_dead'] == False]
    if len(not_dead) != 0:
        return True
    else:
        return False
```

Funkcja ta zwraca prawdę jeżeli wszystkie węzły są żywe i fałsz kiedy są martwe.

```
def get_sum_of_dead_nodes(self):
    dead = len(self.graph_df.loc[self.graph_df['is_dead'] == True])
    self.dead.append(dead)

def get_sum_of_energy(self):
    energy = self.graph_df['energy'].sum()
    self.energy.append(energy-300000.0)
```

Obie funkcje służą sumowaniu. Pierwsza zwraca ilość martwych węzłów, a druga zwraca sumę energii.

```

def draw_tests_plot(self):
    fig, axs = plt.subplots(2)
    x = np.linspace(0, self.rounds_amount, self.rounds_amount)
    print(x)
    y_dead = self.dead
    print(y_dead)
    y_energy = self.energy
    print(y_energy)
    axs[0].set_title('amount of dead nodes per round')
    axs[0].set_xlabel("amount of rounds")
    axs[0].set_ylabel("amount of dead nodes")
    axs[0].plot(x, y_dead)

    axs[1].set_title('amount of total energy per round')
    axs[1].set_xlabel("amount of rounds")
    axs[1].set_ylabel("amount of total energy")
    axs[1].plot(x, y_energy)

    fig.tight_layout()
    return fig

```

Funkcja ta rysuje wykresy na temat ilości energii oraz ilości martwych węzłów w czasie rund.

```

def draw_graph(self):
    self.graph = nx.Graph()
    self.dead_update()
    self.get_sum_of_dead_nodes()
    self.get_sum_of_energy()
    self.new_state()
    self.add_nodes()
    if self.not_all_dead():
        self.cluster_head_selection()
        self.antenna_edges()
        self.clustering()
        self.energy_consumption()

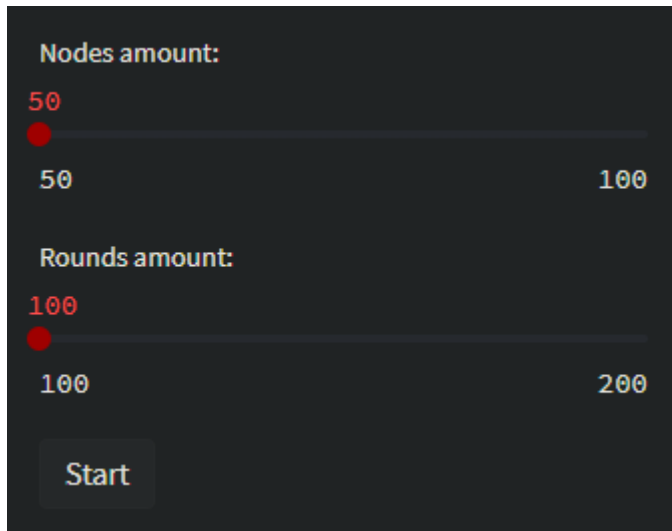
    pos = nx.get_node_attributes(self.graph, 'pos')

    color = self.graph_df.color.tolist()
    return nx.draw(self.graph, pos=pos, node_color=color, node_size=50)

```

Funkcja draw_graph wykonuje wszystkie poprzednie funkcje i rysuje graf ukazujący aktualny stan sieci.

3. Wizualizacja



Nodes amount:

50

50 100

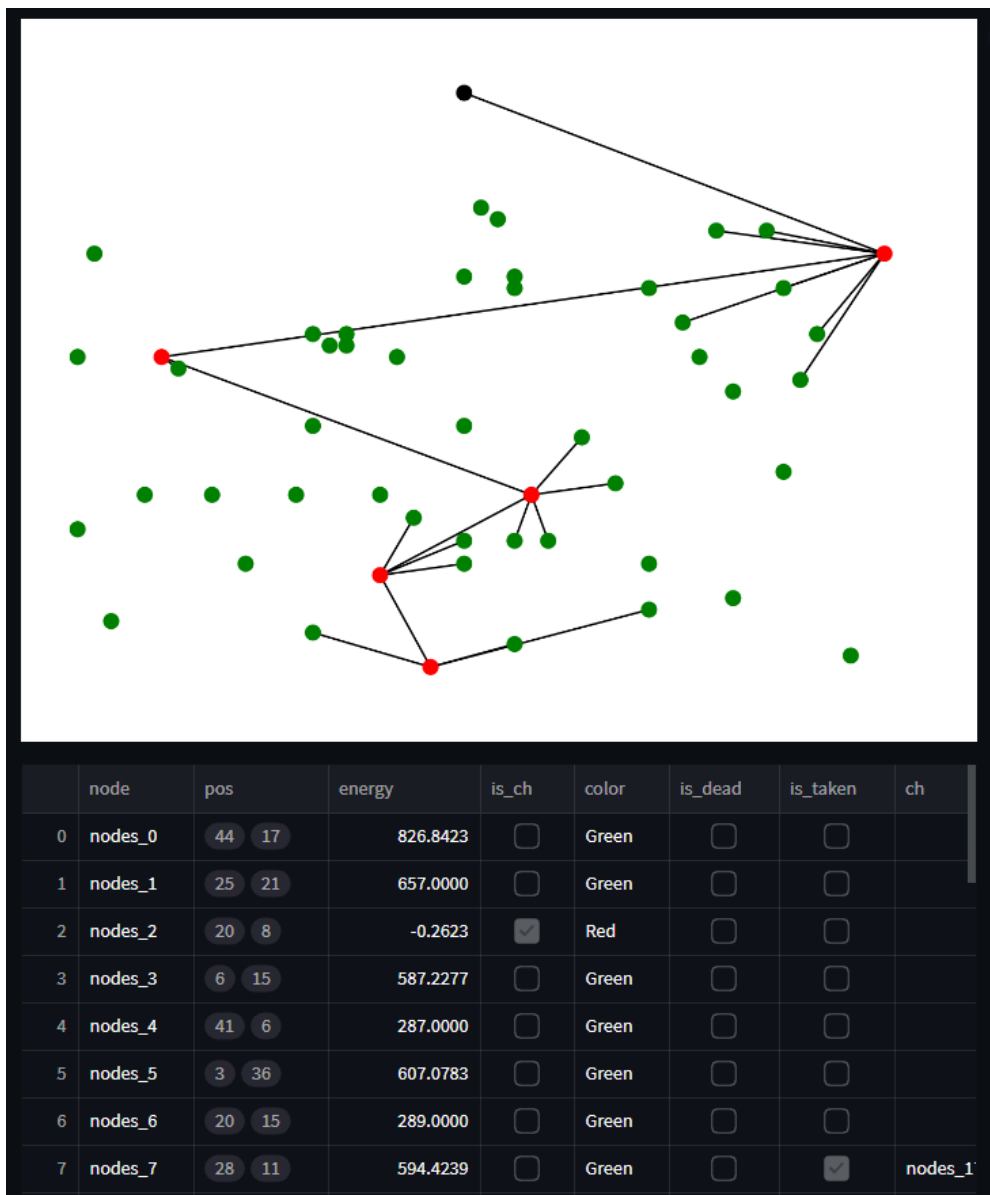
Rounds amount:

100

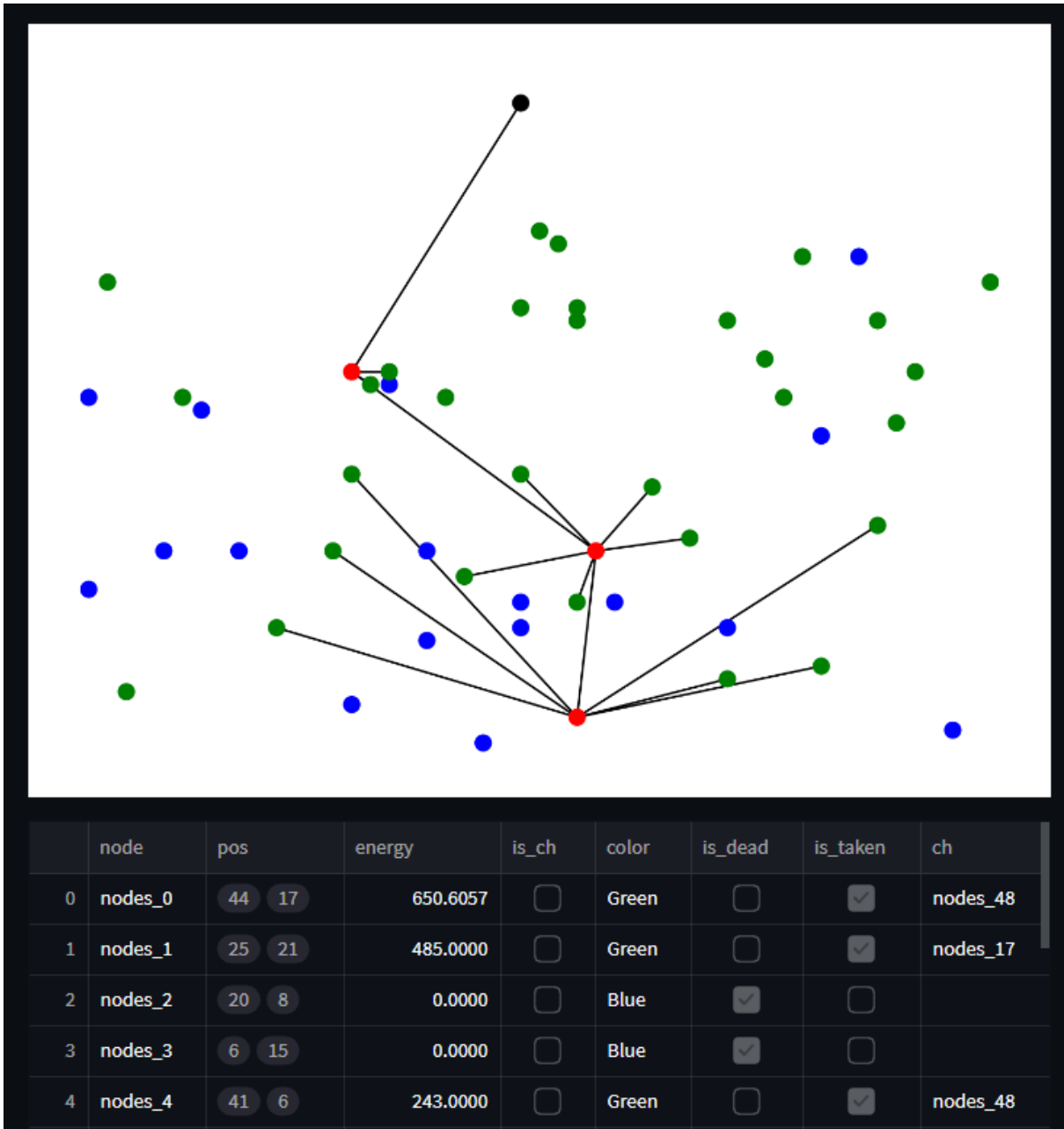
100 200

Start

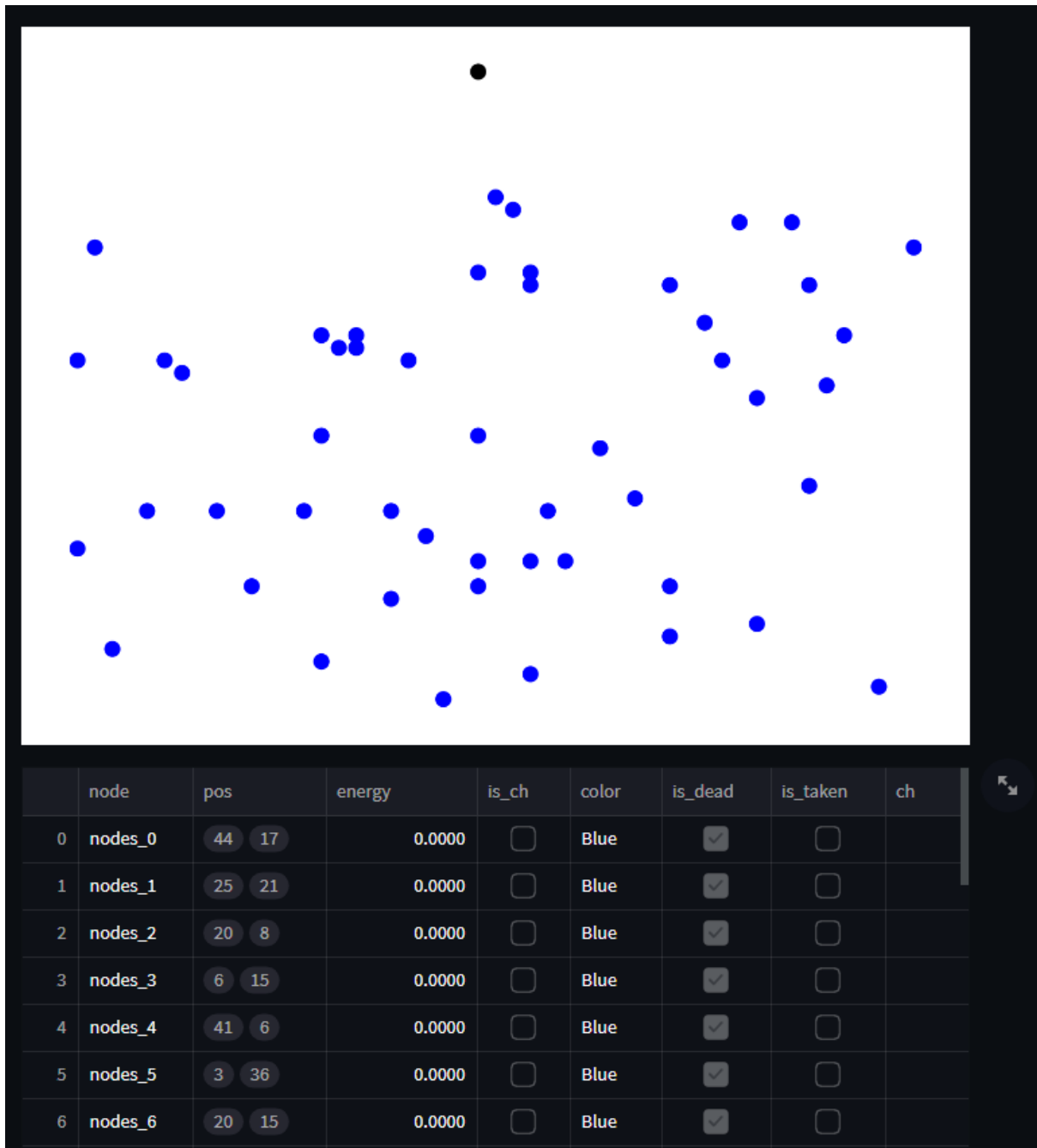
Użytkownik wstępnie może określić ilość węzłów i ilość rund. Po wciśnięciu przycisku start program rozpoczyna swoje działanie.



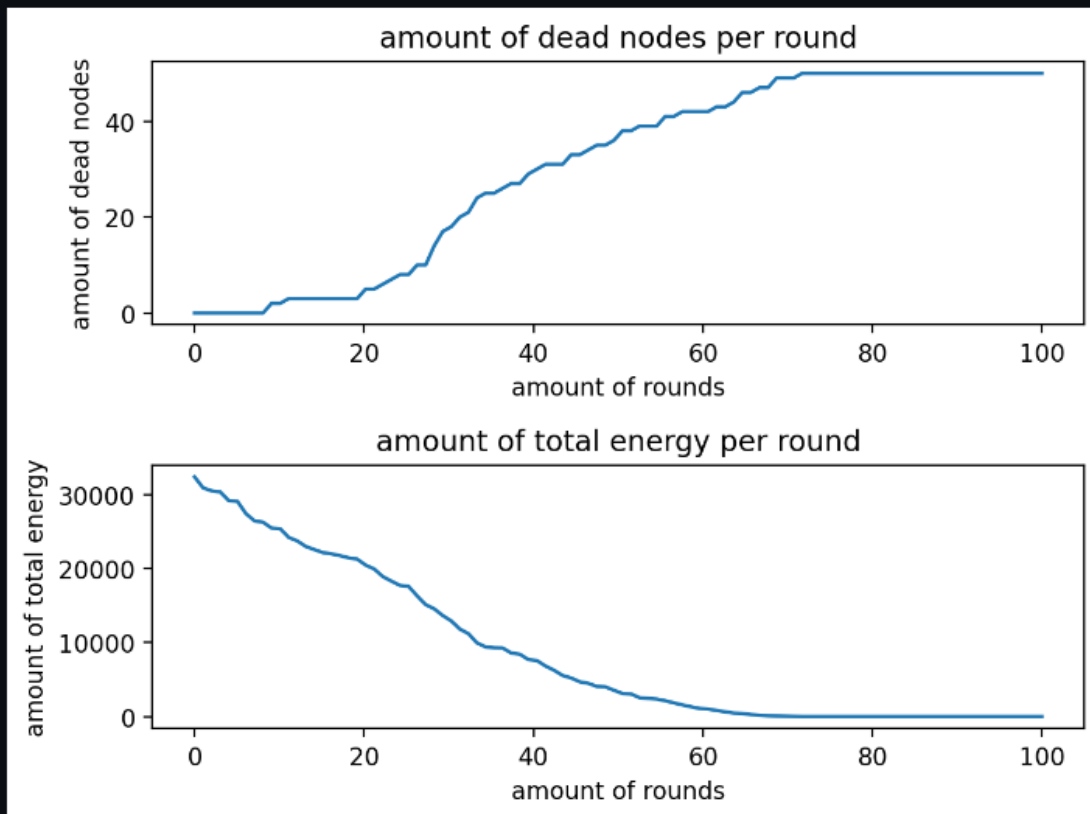
W przykładowej rundzie pokazuje nam się graf oraz zawartość ramki danych.



Powyższy przykład pokazuje rundę z martwymi węzłami, które nie mają już w sobie energii.



W pewnym momencie wszystkie węzły są martwe ale program będzie się wykonywał do momentu, który wskazał użytkownik wyznaczając ilość rund.



Na końcu pokazane są wykresy dotyczące ilości energii i ilości martwych węzłów w każdej rundzie.

3. Podsumowanie

Dzięki temu projektowi poznaliśmy jeden z wielu rodzajów algorytmów klastrujących w sieciach WSN. Pokazaliśmy jego zastosowanie poprzez powyższe wizualizacje i poszczególne opisy.