Opracowanie Algorytmu Oszczędnego Połączenia Zabudowań w Sieć Elektryczną

Jakub Gąsiorowski, Dominik Banach

21 maja 2025
Repozytorium GitHub:
`https://github.com/DominikBanach/Minimal-Electric-Grid-MST/tree/release-1.0`

1 Wprowadzenie

Projekt ma na celu opracowanie i implementację metody planowania struktury sieci elektrycznej we wsi w sposób minimalizujący całkowitą długość (lub koszt) potrzebnych połączeń kablowych. Problem ten został zamodelowany jako zagadnienie znalezienia Minimalnego Drzewa Rozpinającego (MST) w grafie. W grafie tym, zabudowania reprezentowane są przez wierzchołki, a potencjalne połączenia kablowe przez krawędzie o wagach odpowiadających długości lub kosztowi tych połączeń. Do rozwiązania tego problemu wykorzystano algorytm Kruskala, zaimplementowany w języku Python wraz z modułami do parsowania danych wejściowych oraz wizualizacji otrzymanych wyników.

2 Podstawy Teoretyczne

2.1 Problem Minimalnego Drzewa Rozpinającego (MST)

Graf nieskierowany ważony to zbiór wierzchołków połączonych krawędziami, gdzie każdej krawędzi przypisana jest waga (liczba rzeczywista, często dodatnia, reprezentująca np. odległość, koszt, czas). Drzewo rozpinające grafu spójnego to podgraf, który jest drzewem (acykliczny i spójny) i zawiera wszystkie wierzchołki oryginalnego grafu. Minimalne Drzewo Rozpinające (MST) to drzewo rozpinające o minimalnej możliwej sumie wag krawędzi. Znalezienie MST jest kluczowe w wielu problemach optymalizacyjnych, takich jak projektowanie sieci (telekomunikacyjnych, elektrycznych, drogowych), klasteryzacja danych czy analizy ścieżek.

2.2 Algorytm Kruskala

Algorytm Kruskala jest zachłannym algorytmem służącym do znajdowania MST w grafie ważonym nieskierowanym. Działa on na zasadzie stopniowego budowa-

nia MST poprzez dodawanie do niego kolejnych krawędzi o najmniejszej wadze, o ile dodana krawędź nie tworzy cyklu z krawędziami już należącymi do drzewa. Kroki algorytmu Kruskala:

1. Posortuj wszystkie krawędzie grafu w kolejności niemalejącej według ich wag.
2. Zainicjalizuj strukturę danych Union-Find (DSU), w której każdy wierzchołek jest w początkowo osobnym zbiorze.
3. Przeglądaj krawędzie grafu w posortowanej kolejności.
4. Dla każdej rozpatrywanej krawędzi (u, v) o wadze w:
5. Sprawdź, czy wierzchołki u i v należą do tego samego zbioru używając operacji `find`.
6. Jeśli u i v należą do różnych zbiorów (czyli krawędź (u, v) nie tworzy cyklu z dotychczas dodanymi krawędziami), dodaj krawędź (u, v) do zbioru krawędzi MST i połącz zbiory zawierające u i v używając operacji `union`.
7. Jeśli u i v należą do tego samego zbioru, pomiń krawędź (u, v).
8. Algorytm kończy działanie, gdy do MST dodano $V - 1$ krawędzi (gdzie V to liczba wierzchołków) lub gdy wszystkie krawędzie zostały rozpatrzone.

2.3 Struktura DSU (Union-Find)

Struktura DSU (Disjoint Set Union), zwana również Union-Find, jest używana w algorytmie Kruskala do efektywnego śledzenia komponentów spójności grafu i szybkiego sprawdzania, czy dodanie krawędzi utworzy cykl. DSU przechowuje zbiór rozłącznych zbiorów elementów. Obsługuje dwie kluczowe operacje:

- `find(element)`: Zwraca reprezentanta (korzeń) zbioru, do którego należy dany element.
- `union(element1, element2)`: Łączy zbiory, do których należą `element1` i `element2`, jeśli są one w różnych zbiorach.

Efektywność DSU jest kluczowa dla wydajności algorytmu Kruskala. Standardowe implementacje DSU wykorzystują tablicę rodziców (`parent`), gdzie `parent[i]` przechowuje indeks rodzica elementu `i`. Reprezentant zbioru jest elementem, którego rodzicem jest on sam (`parent[i] == i`). Dwie główne heurystyki optymalizacyjne dla DSU to:

- Kompresja ścieżki (Path Compression): Podczas operacji `find`, każdy element na ścieżce od elementu zapytania do korzenia jest bezpośrednio przypinany do korzenia.

- Łączenie według rangi (Union by Rank) lub rozmiaru (Union by Size): Podczas operacji `union`, drzewo o mniejszej randze (lub rozmiarze) jest przyczepiane jako potomek korzenia drzewa o większej randze (rozmiarze). Zapobiega to tworzeniu się wysokich, niezbalansowanych drzew, co przyspiesza operacje `find`.

Nasza implementacja DSU w klasie `KruskalMSTBuilder` wykorzystuje tablicę `parent` oraz heurystykę łączenia według rangi (`rank`). Implementacja `find` nie używa pełnej kompresji ścieżki (zamiast tego po prostu iteruje do korzenia), ale używa heurystyki rangi w `union`, co znacząco poprawia wydajność w porównaniu do prostej wersji.

2.4 Analiza Złożoności Algorytmu Kruskala

Złożoność obliczeniowa algorytmu Kruskala jest zdominowana przez dwa główne etapy: sortowanie krawędzi grafu oraz wykonywanie operacji na strukturze danych Union-Find (DSU).

- Sortowanie krawędzi: Algorytm Kruskala wymaga przetworzenia krawędzi w kolejności niemalejącej według ich wag. Jeśli graf wejściowy ma E krawędzi, posortowanie ich zajmuje $O(E \log E)$ czasu. Nasz moduł `InputParser` realizuje sortowanie przy użyciu implementacji algorytmu sortowania przez scalanie (Merge Sort), który ma gwarantowaną złożoność czasową $O(E \log E)$.
- Operacje DSU: W głównej pętli algorytmu Kruskala, dla każdej z E krawędzi wykonywane są operacje `find` na końcach krawędzi oraz, jeśli wierzchołki należą do różnych zbiorów, operacja `union`. W sumie wykonanych zostanie E operacji `union` (niektóre mogą nic nie zrobić, jeśli wierzchołki są w tym samym zbiorze) oraz co najwyżej $2E$ operacji `find`. Nasza implementacja struktury DSU w klasie `KruskalMSTBuilder` wykorzystuje tablicę rodziców (`parent`) oraz heurystykę łączenia według rangi (`rank`). Pojedyncza operacja `find` może w najgorszym przypadku przejść ścieżkę o długości logarytmicznej względem liczby wierzchołków, tj. $O(\log V)$, gdzie V to liczba wierzchołków w grafie. Zatem łączny koszt wszystkich $O(E)$ operacji `find` i `union` w pętli algorytmu Kruskala wynosi w najgorszym przypadku $O(E \log V)$.

Całkowita złożoność czasowa algorytmu Kruskala w naszej implementacji jest sumą złożoności sortowania i operacji DSU. Wynosi ona $O(E \log E + E \log V)$.

Ponieważ w typowych grafach, gdzie $E \geq V - 1$ (dla grafu spójnego), $\log E$ jest zazwyczaj tego samego rzędu wielkości co $\log V$ (lub większe, jeśli graf jest gęsty), złożoność $O(E \log V)$ jest często dominowana przez $O(E \log E)$. W przypadku grafów rzadkich, gdzie $E \approx V$, złożoność wynosi $O(V \log V + V \log V) = O(V \log V)$. W przypadku grafów gęstych, gdzie $E \approx V^2$, wynosi $O(V^2 \log V^2 + V^2 \log V) = O(V^2 \log V)$.

3 Implementacja

Rozwiązanie zostało zaimplementowane w języku Python i podzielone na cztery główne moduły, z których każdy odpowiada za inną część procesu.

3.1 Struktura Kodu

Projekt składa się z następujących plików:

- `Main.py`: Główny skrypt, który spaja działanie programu, wczytując dane z plików wejściowych, inicjując budowę MST i wywołując wizualizację dla każdego przykładu.
- `InputParser.py`: Moduł odpowiedzialny za wczytywanie danych grafu z plików tekstowych oraz wstępne przetwarzanie danych, w tym sortowanie krawędzi.
- `KruskalMSTBuilder.py`: Zawiera implementację algorytmu Kruskala wraz ze strukturą Union-Find.
- `Visualizer.py`: Moduł do graficznej prezentacji grafu wejściowego i wyznaczonego MST przy użyciu bibliotek `networkx` i `matplotlib`.

3.2 Moduł Wczytywania Danych (`InputParser`)

Klasa `InputParser` odpowiada za przetworzenie surowych danych grafu z pliku tekstowego. Metoda `parseInputFileIntoNumberOfVerticesAndEdgesList` odczytuje plik linia po linii. Każda linia powinna zawierać trzy liczby całkowite rozdzielone spacjami: identyfikator pierwszego wierzchołka, identyfikator drugiego wierzchołka oraz wagę krawędzi między nimi. Moduł dynamicznie określa liczbę wierzchołków (`n`) na podstawie zbioru unikalnych identyfikatorów wierzchołków występujących w pliku. Wszystkie odczytane krawędzie są przechowywane w liście. Kluczową funkcjonalnością tego modułu jest metoda `sortEdges`, która wykorzystuje implementację algorytmu sortowania przez scalanie (Merge Sort)] do posortowania listy krawędzi według ich wag rosnąco. Tak posortowana lista krawędzi jest zwracana do głównego programu i jest niezbędnym wejściem dla algorytmu Kruskala.

3.3 Moduł Algorytmu Kruskala (`KruskalMSTBuilder`)

Klasa `KruskalMSTBuilder` zawiera implementację algorytmu Kruskala. W konstruktorze inicjalizowana jest struktura DSU w postaci dwóch list: `parent` (tablica rodziców) i `rank` (tablica rang dla heurystyki union by rank), gdzie każdy wierzchołek jest początkowo swoim własnym rodzicem i ma rangę 0. Metoda `find(x)` służy do znajdowania reprezentanta zbioru zawierającego wierzchołek `x` poprzez nawigację w górę drzewa do korzenia. Metoda `union(x, y)` próbuje połączyć zbiory zawierające `x` i `y`. Wykorzystuje heurystykę łączenia według

rangi, aby zachować zbalansowaną strukturę drzew (chociaż implementacja aktualizacji rangi przy równych rangach jest specyficzna). Jeśli wierzchołki `x` i `y` są w różnych zbiorach (operacja zakończy się połączeniem), metoda zwraca `True`. Główna logika algorytmu znajduje się w metodzie `kruskalMST`, która iteruje przez listę posortowanych krawędzi dostarczoną w konstruktorze. Dla każdej krawędzi wywoływana jest metoda `union` dla jej końców. Jeśli `union` zwróci `True` (oznaczając, że krawędź łączy dwa różne komponenty i nie tworzy cyklu), krawędź jest dodawana do listy `self.MST`, która po przetworzeniu wszystkich krawędzi zawiera krawędzie Minimalnego Drzewa Rozpinającego.

3.4 Moduł Wizualizacji Grafu (`Visualizer`)

Moduł `Visualizer` odpowiada za graficzną prezentację grafu wejściowego oraz wyniku działania algorytmu Kruskala. Wykorzystuje biblioteki `networkx` do manipulacji strukturą grafu i `matplotlib.pyplot` do jego rysowania. Metoda `present` tworzy obiekt grafu `nx.Graph`, dodaje do niego wszystkie wierzchołki oraz wszystkie krawędzie z danymi wagami. Pozycje wierzchołków na wykresie są wyznaczane przy użyciu algorytmu rozkładu sił o nazwie `spring layout`. Krawędzie należące do wyznaczonego MST są odróżniane wizualnie od pozostałych: krawędzie MST są rysowane jako grube, żółte linie, podczas gdy pozostałe krawędzie są cienkie, szare i przerywane. Wierzchołki są reprezentowane przez emoji domków umieszczone na odpowiednich pozycjach, co nawiązuje do modelowanego problemu połączenia zabudowań. Wagi wszystkich krawędzi są wyświetlane jako etykiety na krawędziach. Tytuł wykresu informuje o prezentowanej strukturze sieci. Wykres jest następnie wyświetlany za pomocą `matplotlib`.

4 Zastosowanie i Przykład Działania

4.1 Modelowanie Połączeń jako Grafu

Problem oszczędnego połączenia zabudowań we wsi w sieć elektryczną jest naturalnie modelowany jako problem grafowy. Każde zabudowanie stanowi wierzchołek grafu. Potencjalne połączenia kablowe między zabudowaniami są krawędziami. Wagi krawędzi odpowiadają kosztowi lub długości ułożenia kabla między dwoma zabudowaniami. Celem jest połączenie wszystkich zabudowań w jedną spójną sieć elektryczną przy minimalnym łącznym koszcie, co dokładnie odpowiada definicji Minimalnego Drzewa Rozpinającego w tym grafie. Algorytm Kruskala znajduje optymalny zestaw połączeń (krawędzi MST), które spełniają ten warunek.

4.2 Format Plików Wejściowych

Program wczytuje dane opisujące graf z plików tekstowych. Każdy plik powinien zawierać listę krawędzi, gdzie każda krawędź jest opisana w osobnej linii w formacie: `wierzchołek_źródłowy wierzchołek_docelowy waga_krawędzi`

Wierzchołki są identyfikowane liczbami całkowitymi. Wagi krawędzi są również liczbami całkowitymi. Przykład kilku pierwszych linii takiego pliku:

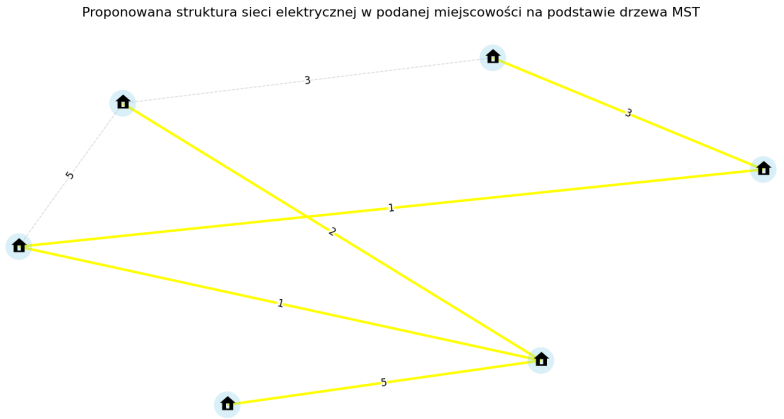
Listing 1: Przykład zawartości pliku wejściowego

```
0 1 14
1 2 2
2 3 16
3 0 5
...
```

Moduł `InputParser` automatycznie wykrywa liczbę wierzchołków na podstawie unikalnych identyfikatorów wierzchołków pojawiających się w krawędziach.

4.3 Przykładowe Dane i Wynik Algorytmu

Jako ilustrację działania algorytmu, rozpatrzmy dane z pierwszego przykładowego pliku wejściowego (np. `inputExample0.txt`). Plik ten zawiera opis małego grafu. Po wczytaniu danych, moduł `InputParser` sortuje krawędzie, a następnie `KruskalMSTBuilder` przetwarza je, budując Minimalne Drzewo Rozpinające. Wizualizacja przedstawiająca graf wejściowy z wyróżnionym MST dla tych danych jest pokazana na Rysunku 1. Wyraźnie widać, które krawędzie zostały wybrane do MST (grube, żółte linie), a które pozostały poza nim (szare, przerywane linie). Etykiety na krawędziach pokazują ich wagi. Suma wag krawędzi należących do MST stanowi minimalny całkowity koszt połączenia wszystkich wierzchołków w tym grafie. Pozostałe wizualizacje są załączone na końcu dokumentu.



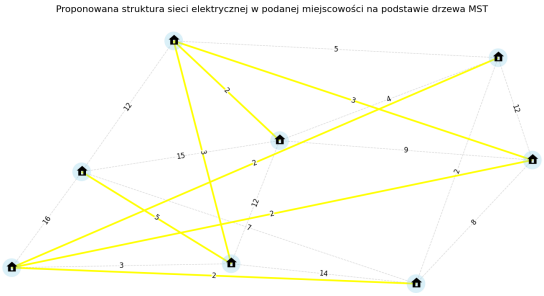
Rysunek 1: Wizualizacja grafu wejściowego z wyróżnionym Minimalnym Drzewem Rozpinającym dla danych z pliku inputExample0.txt

5 Podsumowanie

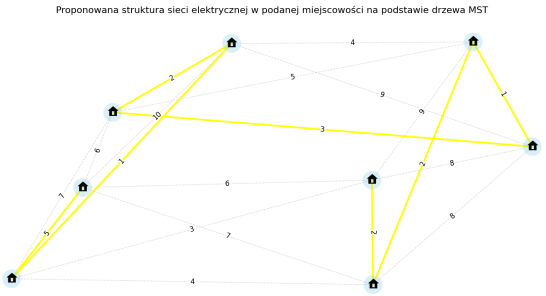
W ramach projektu z powodzeniem zaimplementowano algorytm Kruskala do wyznaczania Minimalnego Drzewa Rozpinającego, co pozwoliło na rozwiązanie problemu optymalnego połączenia zabudowań w sieć elektryczną o minimalnym całkowitym koszcie. Stworzona aplikacja umożliwia wczytywanie danych grafu z plików tekstowych o prostym, ustandaryzowanym formacie, przetwarza je wykorzystując zaimplementowany algorytm Kruskala (bazujący na strukturze Union-Find i sortowaniu przez scalanie), a następnie prezentuje wyniki w formie czytelnych wizualizacji, na których wyraźnie zaznaczone jest wyznaczone Minimalne Drzewo Rozpinające.

6 Wyniki dla Wszystkich Przykładów

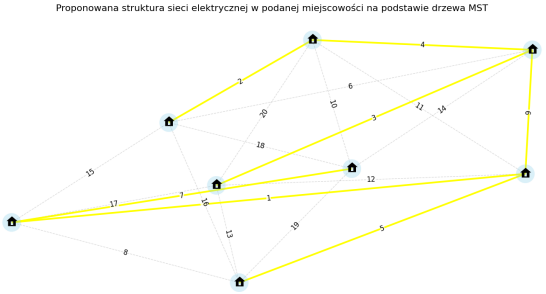
Poniżej zaprezentowano wizualizacje Minimalnych Drzew Rozpinających wyznaczonych dla wszystkich 8 przygotowanych zestawów danych wejściowych (pliki `inputExample0.txt` do `inputExample7.txt`). Każdy rysunek przedstawia graf wejściowy z wyróżnionymi krawędziami należącymi do wyznaczonego MST.



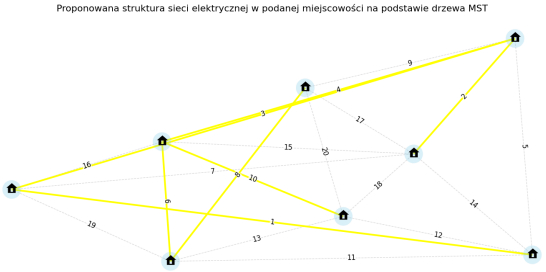
Rysunek 2: Wizualizacja MST dla danych z pliku inputExample1.txt. Łączna waga MST: 19.



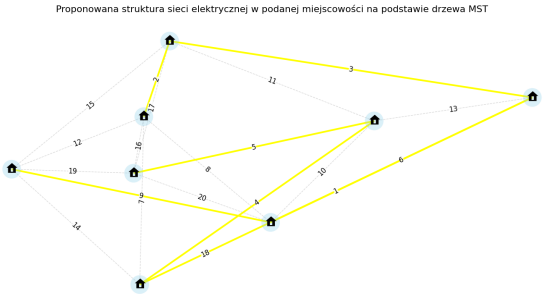
Rysunek 3: Wizualizacja MST dla danych z pliku inputExample2.txt. Łączna waga MST: 15.



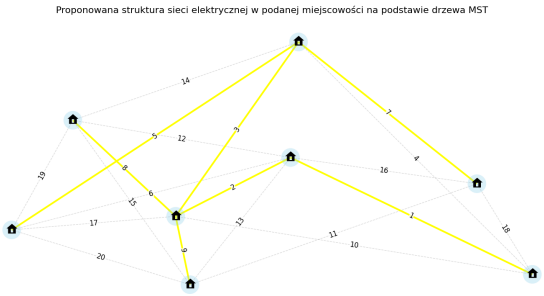
Rysunek 4: Wizualizacja MST dla danych z pliku inputExample3.txt. Łączna waga MST: 28.



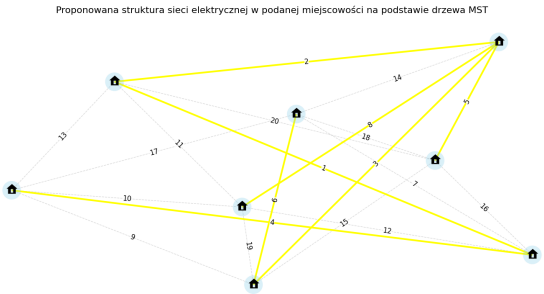
Rysunek 5: Wizualizacja MST dla danych z pliku inputExample4.txt. Łączna waga MST: 34.



Rysunek 6: Wizualizacja MST dla danych z pliku inputExample5.txt. Łączna waga MST: 30.



Rysunek 7: Wizualizacja MST dla danych z pliku inputExample6.txt. Łączna waga MST: 35.



Rysunek 8: Wizualizacja MST dla danych z pliku inputExample7.txt. Łączna waga MST: 32.