

Spis treści

1	Wstęp	2
1.1	Treść zadania	2
1.2	Zastosowany algorytm	2
2	Algorytm Kruskala	2
2.1	Opis algorytmu	2
2.2	Pseudokod	3
2.3	Dowód działania algorytmu Kruskala	3
2.4	Prosty przykład działania algorytmu Kruskala	4
3	Przykłady	8
3.1	Przykład 1	8
3.2	Przykład 2	11
3.3	Przykład 3	14
3.4	Przykład 4	15
4	Strona techniczna	17
4.1	Program	17
4.1.1	Sortowanie przez kopcowanie (HeapSort)	17
4.1.2	Losowanie liczby miast i dróg	18
4.2	Język Programowania	19
4.3	Pakiety	19
5	Złożoność algorytmu	19
5.1	Złożoność obliczeniowa	19
5.2	Złożoność pamięciowa	21

1 Wstęp

1.1 Treść zadania

W Bajtocji jest **n** głównych miast, $n \in [1, 100]$. W związku z ich gwałtownym rozwojem zachodzi potrzeba utworzenia szybkich połączeń między tymi miastami.

Wiadomo, że istnieje system **m** połączeń drogowych, $m \in [1, 300]$, jest on jednak przestarzały co spowalnia i istotnie ogranicza rozwój ekonomiczny głównych miast bajtockich. Każda z m dróg ma swój własny koszt modernizacji **k**, $k \in \mathbb{R}^+$.

Celem zadania jest opracowanie planu, który umożliwi renowację istniejących dróg tak, by powstała sieć nowoczesnych autostrad. Koszt budowy autostrad ma być możliwie najniższy oraz z każdego głównego miasta musi istnieć możliwość dotarcia autostradami do wszystkich pozostałych głównych miast.

1.2 Zastosowany algorytm

Nasz plan modernizacji dróg bajtockich oparłyśmy na **Algorytmie Kruskala**. Algorytm Kruskala jest idealnym rozwiązaniem dla tego zadania, ponieważ szukamy minimalnego kosztu budowy autostrad pomiędzy wszystkimi głównymi miastami Bajtocji. Algorytm Kruskala znajduje minimalne drzewo rozpinające (tzw. MST) dla danego grafu, co oznacza, że istnieje ścieżka łącząca wszystkie wierzchołki grafu oraz suma wag jej krawędzi jest minimalna wśród wszystkich możliwych sposobów połączenia wszystkich wierzchołków.

Najczęściej używanymi algorytmami do znajdywania minimalnego drzewa rozpiętego są algorytm Kruskala i algorytm Prima. Wybrany został algorytm Kruskala, ponieważ dzięki niemu można uzyskać lepszą złożoność czasową. Ponadto ze względu na fakt, że algorytm Kruskala opiera się na rozważaniu krawędzi, to działa on najlepiej przy rzadkich grafach, zaś algorytm Prima, którego podstawą są wierzchołki, przy gęstych. W zadaniu projektowym były rozważane głównie grafy rzadkie, ponieważ graf gęsty może powodować, że pomiędzy dwoma miastami będzie niepotrzebnie rozważane kilka bezpośrednich dróg. Dodatkowo, został przyjęty za bardziej realistyczny przypadek, gdy dwa miasta są połączone maksymalnie jedną bezpośrednią trasą.

W przypadku naszego zadania, miasta stwarzyszamy z wierzchołkami grafu, a istniejące drogi z jego krawędziami. Taki graf opisuje połączenia między głównymi miastami Bajtocji, gdzie waga krawędzi to koszt modernizacji danej drogi do standardu autostady. Minimalne drzewo rozpinające będzie reprezentować optymalny plan budowy autostrad o minimalnym koszcie, łączący wszystkie miasta.

2 Algorytm Kruskala

2.1 Opis algorytmu

Do znajdywania minimalnego drzewa rozpinającego w zbiorze najczęściej używa się algorytmu Kruskala lub algorytmu Prima. Do zadania projektowego został zaimplementowany algorytm Kruskala.

Algorytm Kruskala znajduje MST w grafie skierowanym lub nieskierowanym. W przypadku grafu niespójnego można za pomocą algorytmu znaleźć minimalnie rozpinający las. Algorytm działa w następujący sposób:

1. Sortowanie wszystkich krawędzi grafu według ich wag.
2. Dla każdej krawędzi zostaje sprawdzone, czy dodanie jej do drzewa nie spowoduje utworzenia cyklu. Jeśli tak, to ta krawędź jest odrzucana. W przeciwnym razie krawędź zostaje dodana do drzewa.
3. Krok 2. zostaje powtarzany, aż do momentu, gdy wszystkie wierzchołki są połączone w tworzonym drzewie.

Jedną z trudności związanych z implementacją algorytmu Kruskala jest utrzymanie struktury danych, która pozwala na szybkie sprawdzanie, czy dodanie krawędzi do drzewa spowoduje utworzenie cyklu. Można to zrobić za pomocą zbiorów rozłącznych.

2.2 Pseudokod

Poniżej przedstawiony jest algorytm Kruskala w postaci pseudokodu, w którym:

- G oznacza graf;
- GV oznacza zbiór wierzchołków grafu G ;
- GE oznacza zbiór par wierzchołków tworzących krawędzie w grafie G ;
- $MAKE-SET(v)$ to funkcja tworząca jednoelementowy zbiór z wierzchołkiem v ;
- $FIND-SET(v)$ to funkcja, która służy do odnajdywania reprezentanta danego zbioru;
- $UNION(u,v)$ to funkcja, która przypisuje rodzica v jako rodzica wierzchołka u .

```

KRUSKAL(G):
    MST = {}
    for each v in GV:
        MAKE-SET(v)
    for each (u,v) in GE ordered by weight(u,v) increasing:
        if FIND-SET(u) != FIND-SET(v):
            add (u,v) to set MST
            UNION(u,v)
    
```

2.3 Dowód działania algorytmu Kruskala

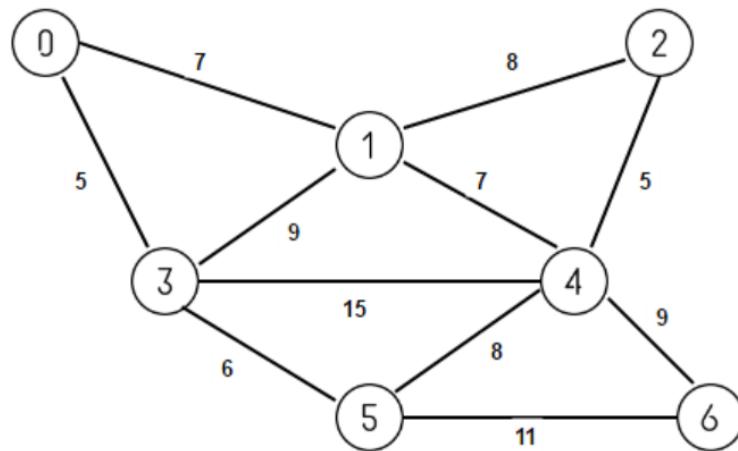
Aby udowodnić, że algorytm Kruskala zwraca minimalne drzewo rozpinające należy pokazać, że:

1. Ten algorytm zwraca drzewo rozpinające.
2. Zwracane drzewo jest drzewem o możliwie najmniejszej wadze.

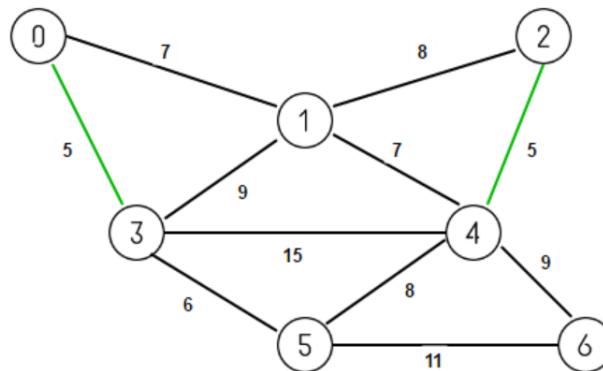
Pierwszy warunek zachodzi, ponieważ algorytm zawsze zwraca drzewo, które zawiera wszystkie wierzchołki grafu i nie zawiera cykli. Drugi warunek również zachodzi, ponieważ algorytm wybiera krawędź o najmniejszej możliwej wadze pomiędzy danymi wierzchołkami, która może być dodana do drzewa tylko wtedy, gdy nie tworzy cyklu. MST musi zawierać wszystkie krawędzie o najmniejszych możliwych wagach, zatem algorytm zawsze zwraca drzewo o najmniejszej możliwej wadze.

2.4 Prosty przykład działania algorytmu Kruskala

Szukamy MST dla poniżej przedstawionego grafu.

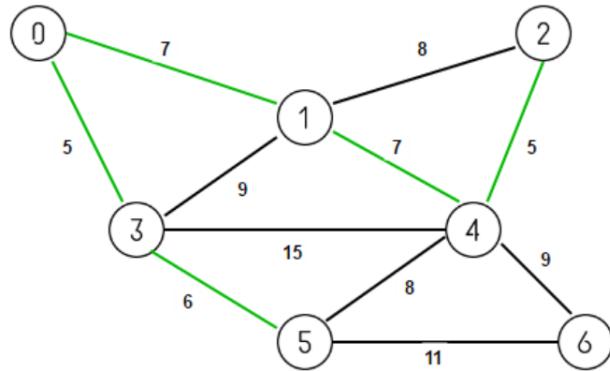


Po posortowaniu krawędzi wybieramy kolejno krawędzie z listy. Weźmy krawędź o najmniejszej wadze, czyli przykładowo $(0,3)$. Wierzchołków tworzących tą krawędź nie ma w drzewie MST, zatem dodajemy ją do drzewa. Następnie rozważmy krawędź $(2,4)$, która nie tworzy cyklu w MST, zatem może zostać dodana do drzewa. MST powstałe po tych krokach zaznaczone jest zielonym kolorem na Rysunku 1.



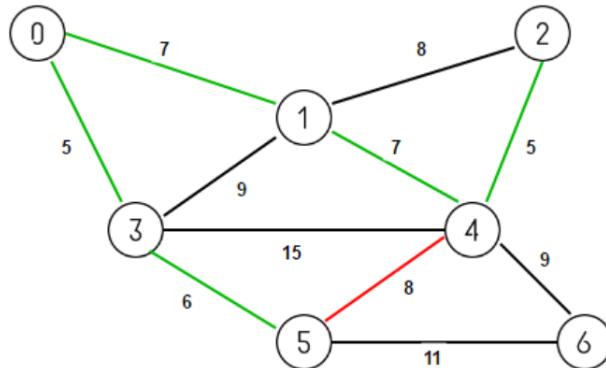
Rysunek 1: MST zaznaczone zielonym kolorem po dołączeniu krawędzi $(0,3)$ i $(2,4)$.

Następnie w ten sam sposób zostają dołączone krawędzie $(3,5)$, $(0,1)$ oraz $(1,4)$ do MST, co jest pokazane na Rysunku 2.



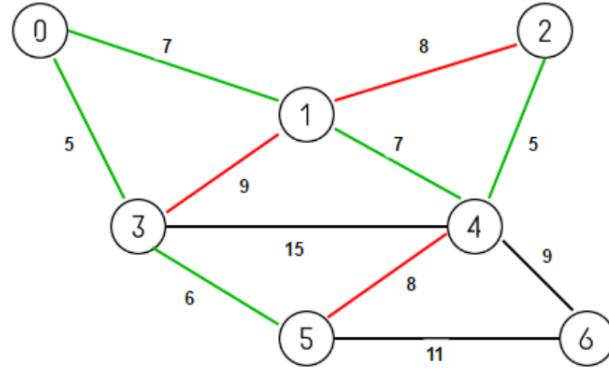
Rysunek 2: MST zaznaczone zielonym kolorem po dołączeniu krawędzi $(3,5)$, $(0,1)$ oraz $(1,4)$.

W kolejnym kroku rozważmy krawędź $(5,4)$ o wadze 8. Gdyby dołączyć tą krawędź do drzewa, powstałby cykl $(0,3,5,4,1)$. Zatem $(5,4)$ nie zostaje dodana do MST, co oznaczono czerwonym kolorem na Rysunku 3.



Rysunek 3: MST zaznaczone zielonym kolorem, $(5,4)$ zaznaczono czerwonym kolorem po odrzuceniu tej krawędzi.

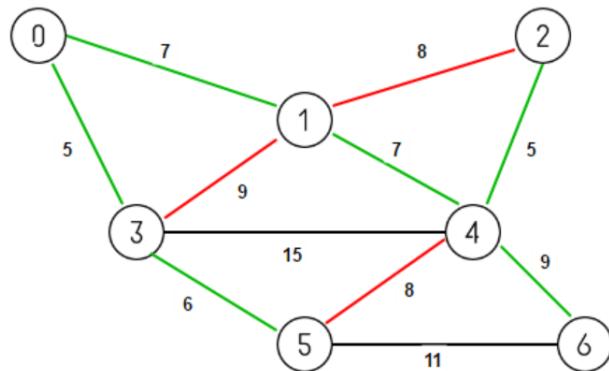
Analogicznie krawędź $(1,2)$ dodana do MST tworzy cykl $(1,2,4)$, zatem nie zostaje dodana do drzewa i oznaczona kolorem czerwonym. Następnie weźmy krawędź $(3,1)$ o wadze 9. Zauważmy, że ta krawędź również tworzy cykl w MST $(0,1,3)$, zatem nie zostaje dodana do drzewa, co przedstawiono na Rysunku 4.



Rysunek 4: MST zaznaczone zielonym kolorem, rozważane i odrzucone krawędzie zaznaczone kolorem czerwonym.

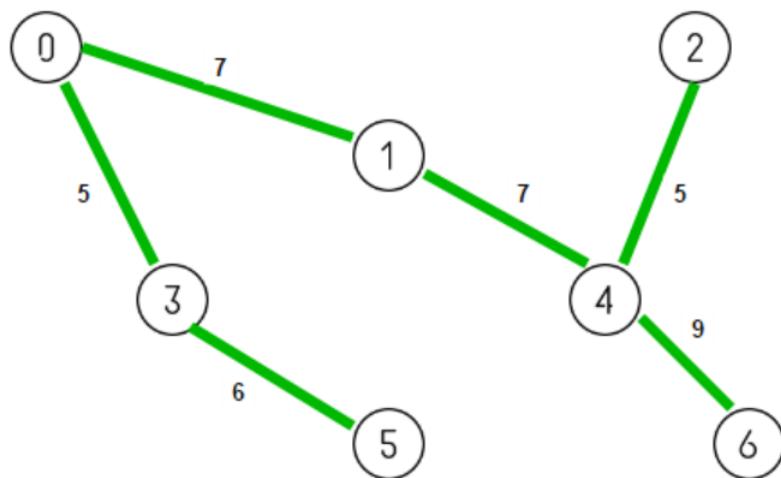
Następnie rozważmy krawędź $(4,6)$. Krawędź ta nie tworzy cyklu w MST, więc zostaje dodana do drzewa.

Zauważmy, że zielone krawędzie oznaczające MST łączą wszystkie wierzchołki grafu, co pokazuje Rysunek 5.



Rysunek 5: MST zaznaczone zielonym kolorem, rozważane i odrzucone krawędzie zaznaczone kolorem czerwonym.

Otrzymane minimalne drzewo rozpinające dla rozważonego grafu przedstawiono poniżej.



3 Przykłady

Wyniki zadania stworzenia sieci autostrad dla n - liczby miast i m - liczby dróg:

- $n < m$;
- $n \ll m$;
- $n \lesssim m$;
- $n > m$

3.1 Przykład 1

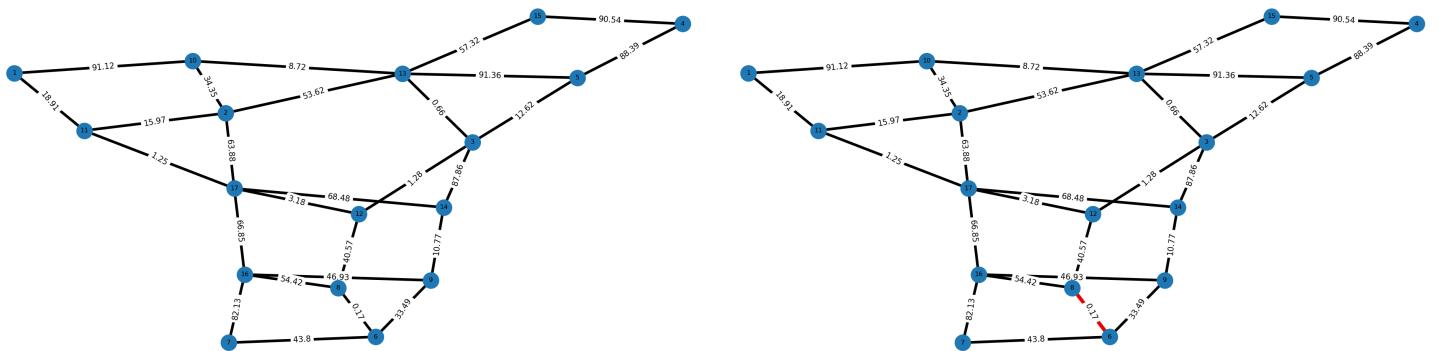
Gdy $n < m$.

Niech $n = 16$, $m = 27$.

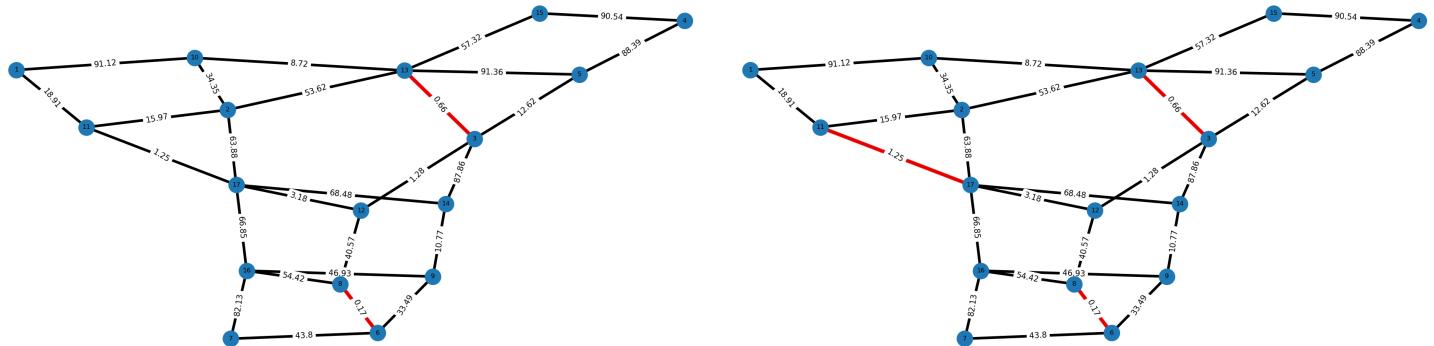
Dla podanych liczb n i m weźmy następujący ciąg trójek:

```
roads=[(10, 13, 8.72),(7,6,43.8), (14, 9, 10.77), (2, 13, 53.62), (11, 2, 15.97), (8, 6, 0.17),
      ↪ (14, 3, 87.86), (13, 3, 0.66), (15, 4, 90.54), (3, 5, 12.62), (1, 10, 91.12), (16, 9, 46.93)
      ↪ , (2, 10, 34.35), (16, 7, 82.13), (17, 11, 1.25), (17, 14, 68.48), (16, 17, 66.85), (12, 17,
      ↪ 3.18), (5, 13, 91.36), (9, 6, 33.49), (5, 4, 88.39), (15, 13, 57.32), (11, 1, 18.91), (3,
      ↪ 12, 1.28), (12, 8, 40.57), (17, 2, 63.88), (16, 8, 54.42)]
```

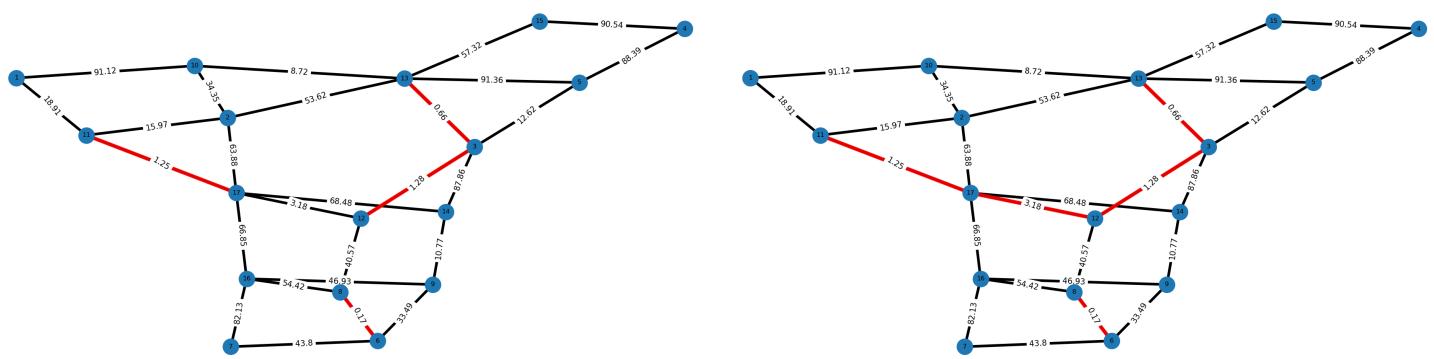
Zgodnie z algorytmem tworzymy drzewo rozpinające, które krok po kroku przyjmuje postać:



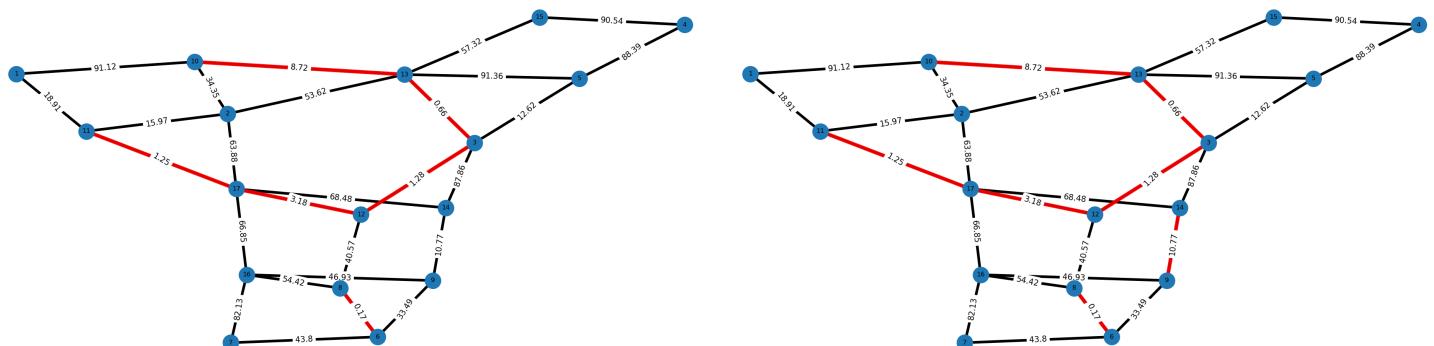
Rysunek 6: Początkowa sieć dróg (lewy graf), krok 1 - czerwona krawędź to pierwszy fragment autostrady, ma najmniejszy koszt (prawy graf)



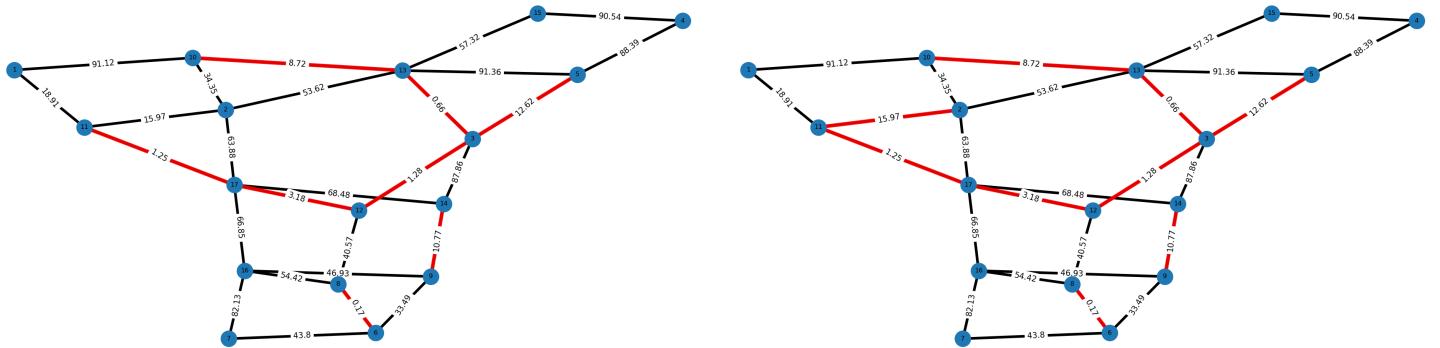
Rysunek 7: Krok 2 (lewy graf), krok 3 (prawy graf)



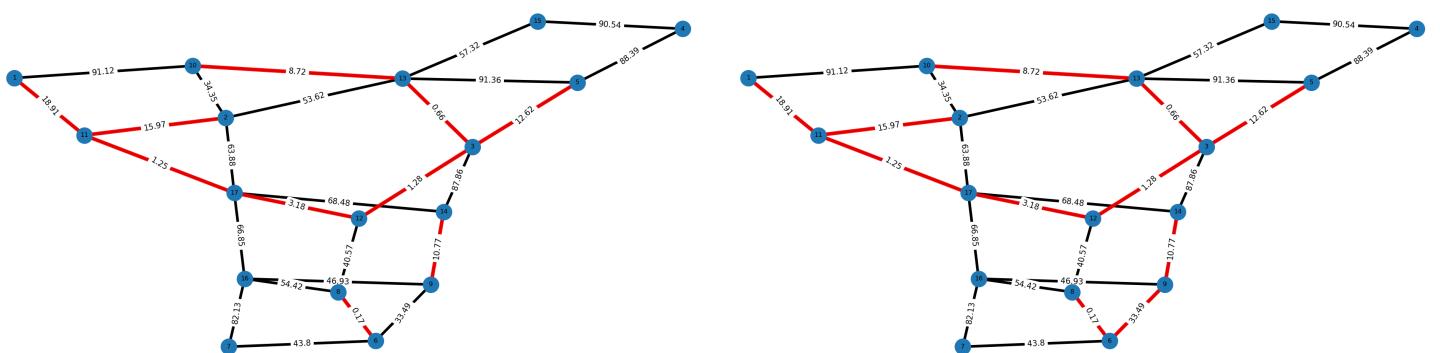
Rysunek 8: Krok 4 (lewy graf), krok 5 (prawy graf)



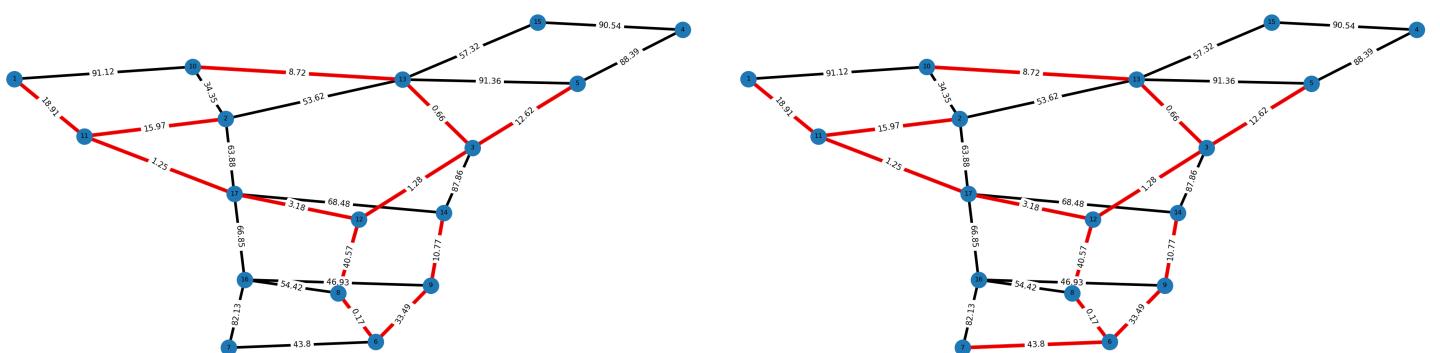
Rysunek 9: Krok 6 (lewy graf), krok 7 (prawy graf)



Rysunek 10: Krok 8 (lewy graf), krok 9 (prawy graf)



Rysunek 11: Krok 10 (lewy graf), krok 11 (prawy graf)



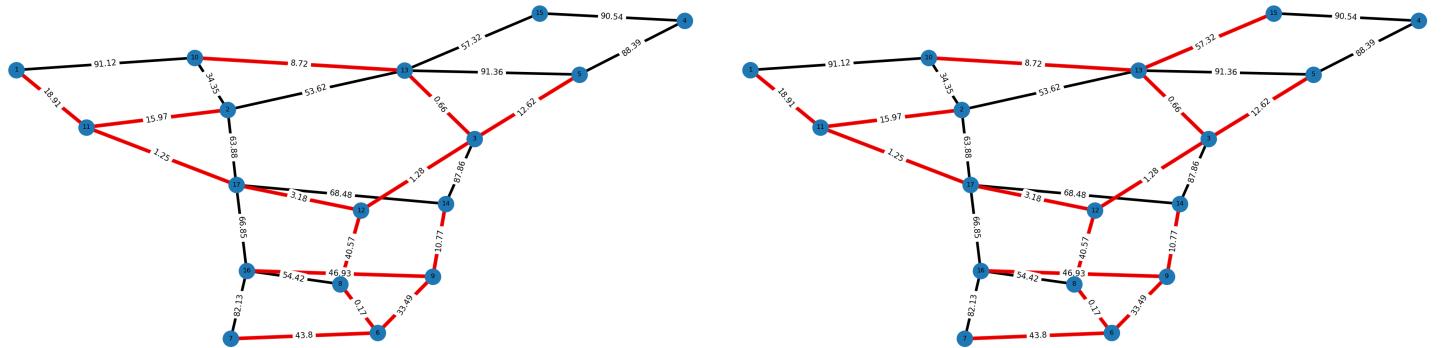
Rysunek 12: Krok 12 (lewy graf), krok 13 (prawy graf)

Na Rysunku 12 pokazana została znaleziona autostrada łącząca wszystkie miasta, składająca się z krawędzi o minimalnych wagach.

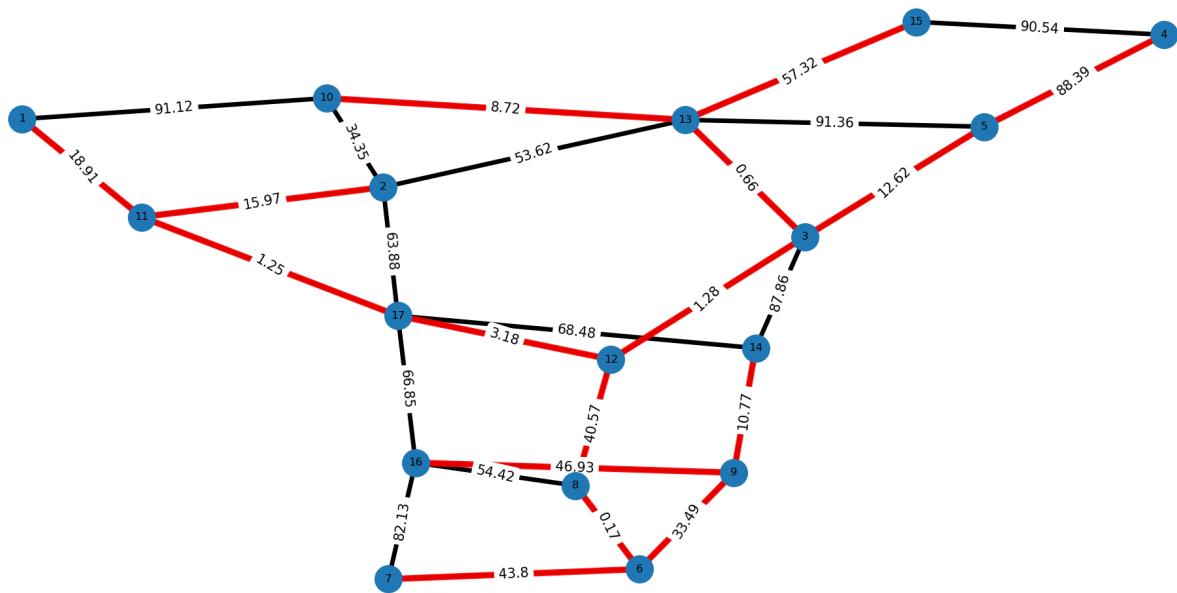
Łączny koszt budowy tej autostrady wynosi 384.03.

Krawędzie tworzące MST to :

$$\begin{aligned} \text{MST} = & [(8, 6), (13, 3), (17, 11), (3, 12), (12, 17), (10, 13), (14, 9), (3, 5), (11, 2), (11, 1), \\ & \hookrightarrow (9, 6), (12, 8), (7, 6), (16, 9), (15, 13), (5, 4)] \end{aligned}$$



Rysunek 13: Krok 14 (lewy graf), krok 15 (prawy graf)



Rysunek 14: Czerwona ścieżka to MST grafu, czyli nasza szukana autostrada o minimalnym koszcie.

3.2 Przykład 2

Gdy $n \ll m$.

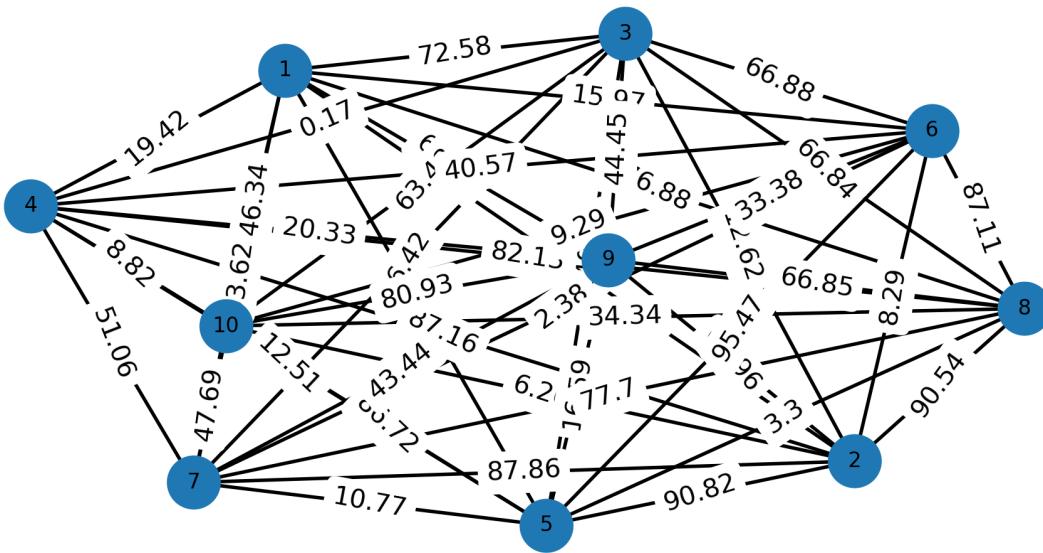
Niech $n = 10$, $m = 100$.

W tym przypadku graf ma krawędzie wielokrotne, co oznacza, że pomiędzy dwoma miastami jest wiele bezpośrednich dróg. Ponieważ do budowy autostrady potrzebne jest do rozważenia jedynie jedno połączenie bezpośrednie pomiędzy miastami, to przyjmujemy, że $m = \frac{n \cdot (n-1)}{2}$. Wtedy graf nie ma krawędzi wielokrotnych oraz każdy wierzchołek jest połączony krawędzią z wszystkimi innymi wierzchołkami.

Dla podanych liczb n i m weźmy następujący ciąg trójkę:

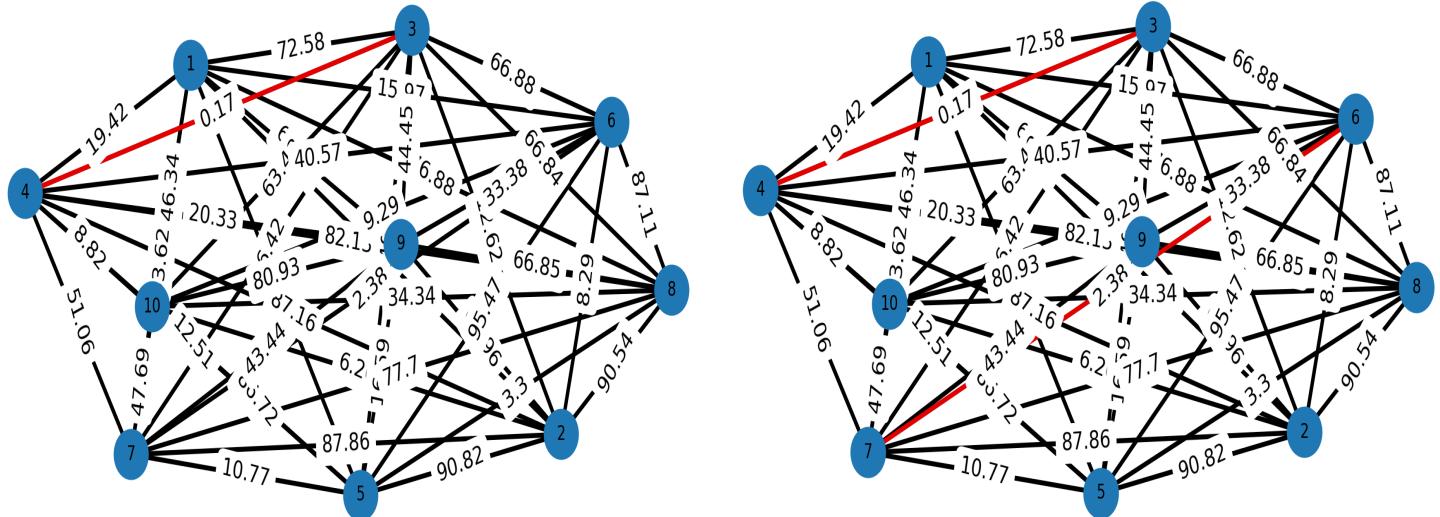
```
roads=[(1, 5, 8.72), (7, 5, 10.77), (1, 7, 53.62), (6, 1, 15.97), (6, 9, 33.38), (4, 3, 0.17), (7, 2, 87.86), (8, 2, 90.54), (2, 3, 12.62), (7, 10, 47.69), (8, 1, 76.88), (8, 4, 82.13), (9, 10, 80.93), (7, 9, 43.44), (8, 9, 66.85), (1, 3, 72.58), (5, 3, 33.49), (4, 10, 8.82), (6, 4, 40.57), (9, 1, 63.88), (3, 9, 44.45), (6, 5, 95.47), (2, 10, 6.26), (6, 2, 8.29), (5, 8, 3.3), (7, 6, 2.38), (3, 7, 6.42), (3, 6, 66.88), (2, 1, 75.81), (9, 2, 62.96), (3, 8, 66.84), (6, 8, 87.11), (10, 6, 9.29), (9, 4, 20.33), (4, 1, 19.42), (8, 7, 77.7), (5, 10, 86.72), (3, 10, 63.48), (10, 8, 34.34), (5, 2, 90.82), (10, 1, 46.34), (4, 2, 87.16), (5, 9, 16.59), (7, 4, 51.06), (5, 4, 12.51)]
```

Wtedy sieć połączeń drogowych w Bajtociji ma postać:

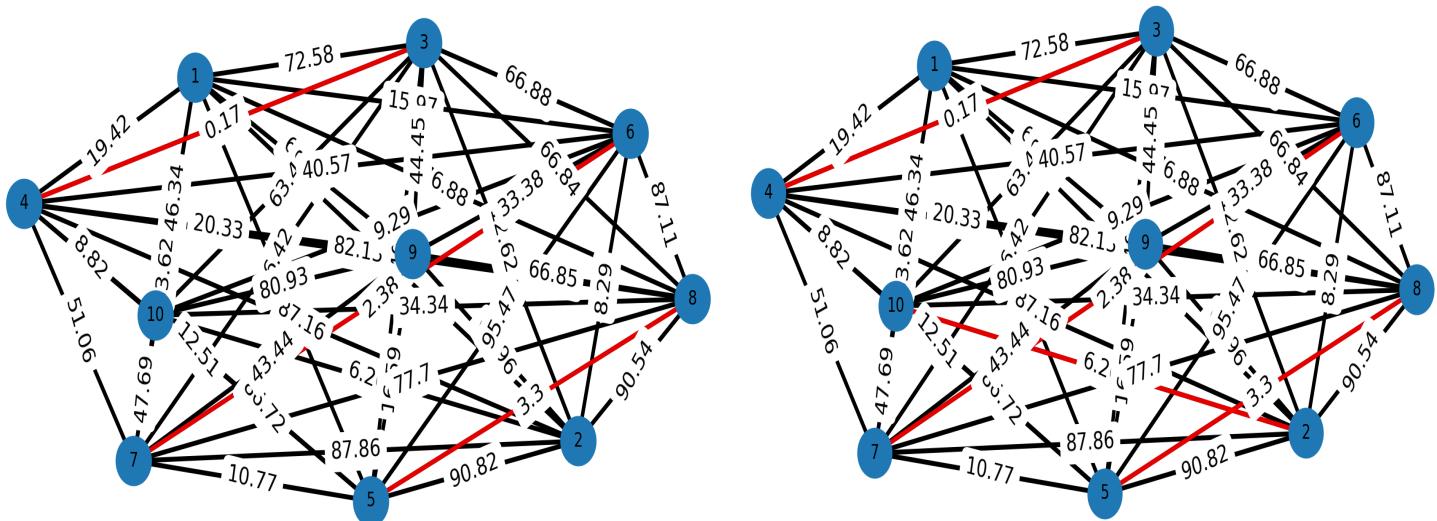


Rysunek 15: Postać sieci połączeń drogowych dla $n = 10$ i $m = 45$.

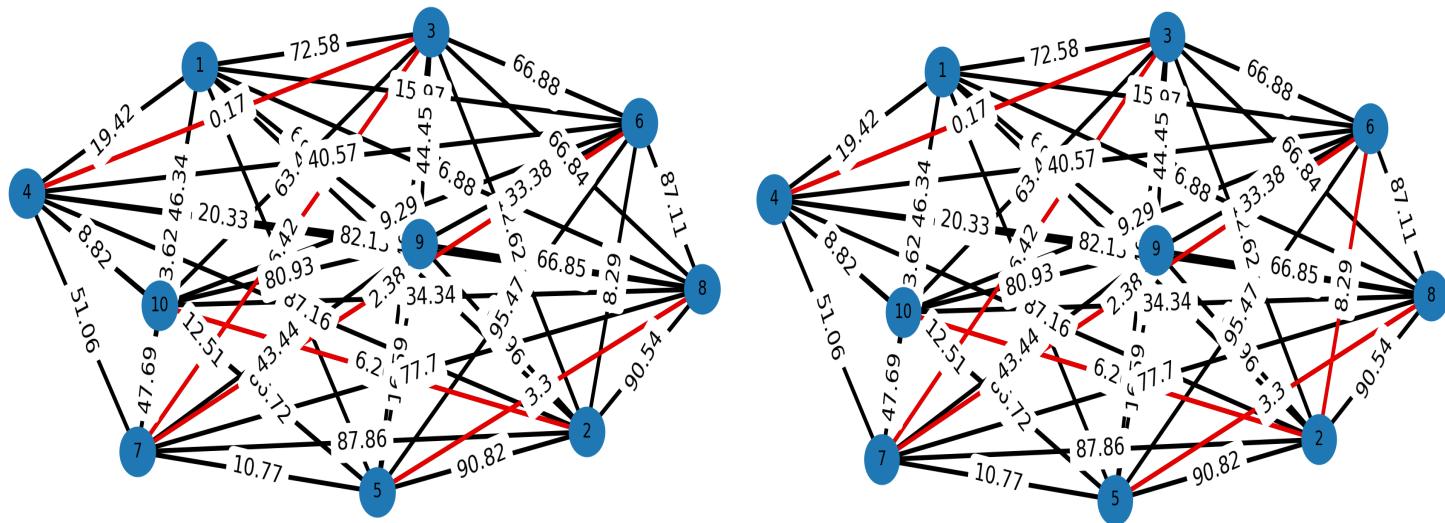
W następnych krokach algorytmu tworzy się MST:



Rysunek 16: Krok 1 (lewy graf), krok 2 (prawy graf)



Rysunek 17: Krok 3 (lewy graf), krok 4 (prawy graf)



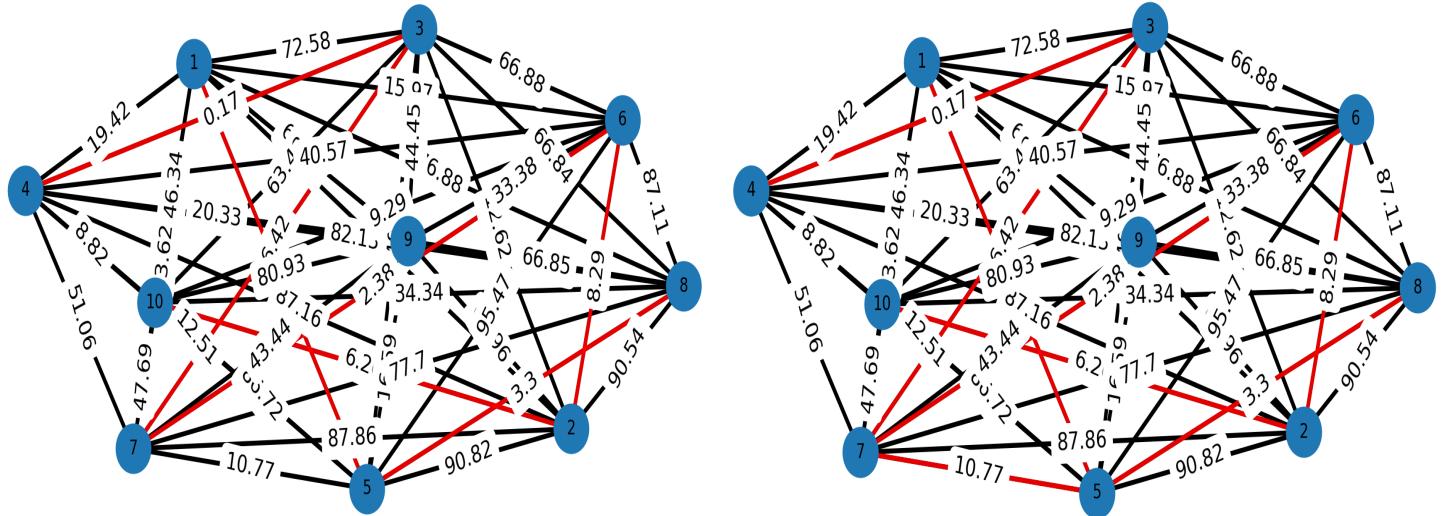
Rysunek 18: Krok 5 (lewy graf), krok 6 (prawy graf)

Na Rysunku 20 pokazana została znaleziona autostrada łącząca wszystkie miasta, składająca się z krawędzi o minimalnych wagach.

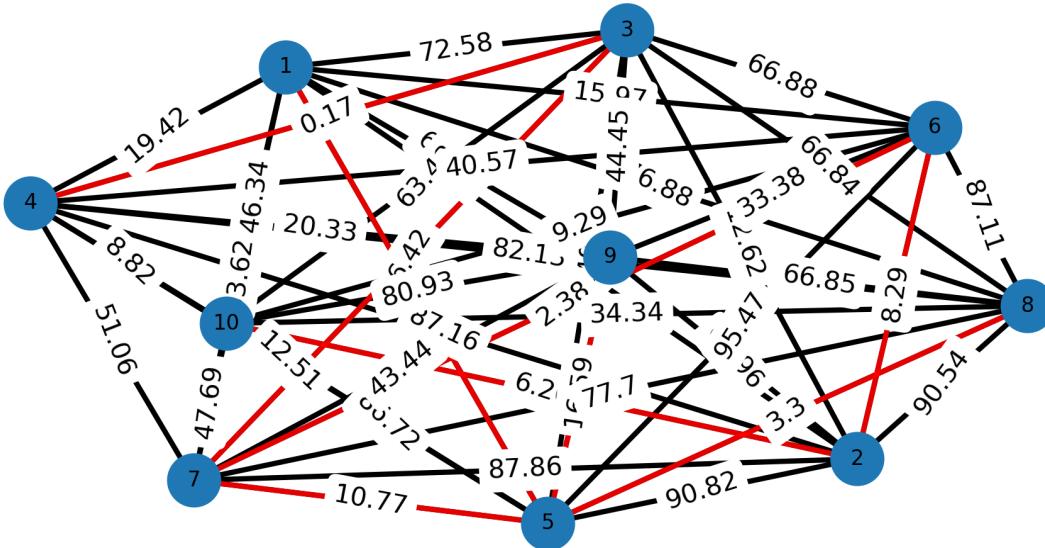
Łączny koszt budowy tej autostrady wynosi 62.9.

Krawędzie tworzące MST to:

$$\text{MST} = [(4, 3, 0.17), (7, 6, 2.38), (5, 8, 3.3), (2, 10, 6.26), (3, 7, 6.42), (6, 2, 8.29), (1, 5, \rightarrow 8.72), (7, 5, 10.77), (5, 9, 16.59)]$$



Rysunek 19: Krok 7 (lewy graf), krok 8 (prawy graf)

Rysunek 20: MST zaznaczone na czerwono dla $n = 10$ i $m = 45$.

3.3 Przykład 3

Gdy $n \lesssim m$.

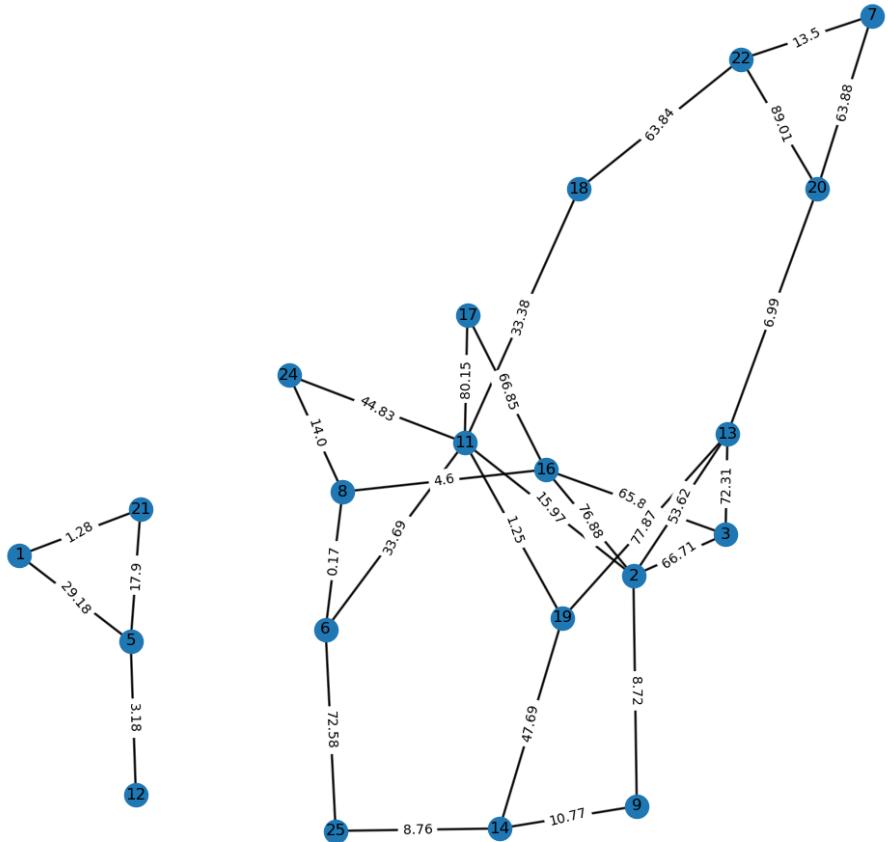
Niech $n = 25$, $m=30$.

Rozważmy sytuację gdy **nie da** utworzyć się sieci autostrad pomimo tego, że dróg jest więcej niż miast.

Dla podanych liczb n i m weźmy następujący ciąg trójkę:

```
roads= [(2, 9, 8.72), (14, 9, 10.77), (2, 13, 53.62), (8, 24, 14), (11, 2, 15.97), (11, 18, 33.38),
       ↪ (8, 6, 0.17), (14, 25, 8.76), (8, 16, 4.6), (20, 13, 6.99), (11, 24, 44.83), (2, 3, 66.71), (5,
       ↪ 1, 29.18), (14, 19, 47.69), (5, 21, 17.9), (16, 2, 76.88), (11, 17, 80.15), (7, 20, 63.88),
       ↪ (19, 11, 1.25), (1, 21, 1.28), (22, 18, 63.84), (22, 20, 89.01), (16, 17, 66.85), (12, 5,
       ↪ 3, 1.18), (25, 6, 72.58), (3, 16, 65.81), (7, 22, 13.5), (6, 11, 33.69), (13, 3, 72.31), (13, 19,
       ↪ 77.87)]
```

Wtedy sieć połączeń drogowych w Bajtocji ma postać:



Rysunek 21: Postać sieci połączeń drogowych dla $n=25$ i $m=30$ dla której nie da utworzyć się autostrady łączącej wszystkie miasta.

Jak widać na rysunku miasta 1, 5, 12 i 21 są odseparowane od pozostałych miast. Pomimo tego, że liczba dróg jest większa od liczby miast, występuje brakujące połączenie co oznacza, że nasz graf nie jest spójny. W takiej sytuacji nie da utworzyć się autostrady, która połączy wszystkie miasta w Bajtocji. Zwracany jest komunikat:
"Nie ma połączenia pomiędzy dwoma miastami"

3.4 Przykład 4

Gdy $n > m$.

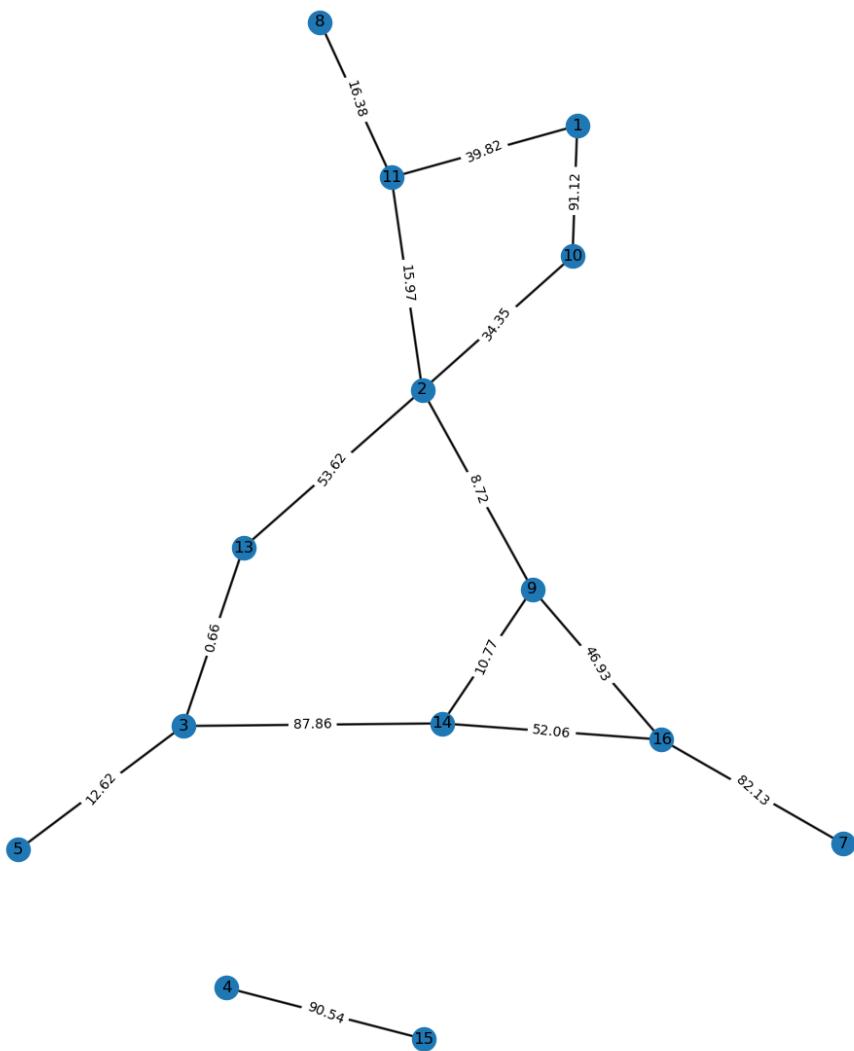
Niech $n = 16$, $m=15$.

Rozważmy sytuację gdy ponownie **nie da** utworzyć się sieci autostrad, tym razem miast jest więcej od dróg.

Dla podanych liczb n i m weźmy następujący ciąg trójkę:

```
roads= [(2, 9, 8.72), (14, 9, 10.77), (2, 13, 53.62), (11, 2, 15.97), (11, 8, 16.38), (14, 3,
      ↪ 87.86), (13, 3, 0.66), (15, 4, 90.54), (3, 5, 12.62), (1, 10, 91.12), (16, 9, 46.93), (2,
      ↪ 10, 34.35), (16, 7, 82.13), (11, 1, 39.82), (14, 16, 52.06)]
```

Wtedy sieć połączeń drogowych w Bajtocji ma postać:



Rysunek 22: Postać sieci połączeń drogowych dla $n=16$ i $m=15$ dla której nie da utworzyć się autostrady łączącej wszystkie miasta.

Jak widać na rysunku nie istnieje połączenie łączące miasta 4 i 15 z pozostałymi miastami. Nie można utworzyć autostrady łączącej ze sobą wszystkie miasta. Zwracany jest komunikat: **"Mniej dróg niż miast nie da się utworzyć autostrady graf nie jest spójny"**

4 Strona techniczna

4.1 Program

Aby znaleźć MST, zastosowany przez nas algorytm Kruskala sortuje krawędzie według ich wag. W celu posortowania krawędzi rozważyłyśmy dwa algorytmy sortowania: algorytm sortowania szybkiego (QuickSort) i algorytm sortowania przez kopcowanie (HeapSort).

Algorytm sortowania szybkiego i algorytm sortowania przez kopcowanie są dobrymi opcjami do posortowania wag krawędzi w algorytmie Kruskala dla grafów o liczbie krawędzi w zakresie od 1 do 300.

- **Algorytm sortowania szybkiego** jest szybki i efektywny, jego złożoność obliczeniowa dla n - elementowej tablicy wynosi $\mathcal{O}(n \log n)$ w średnim i najlepszym przypadku, ale może wzrosnąć do $\mathcal{O}(n^2)$ w najgorszym przypadku (np. gdy tablica jest już posortowana), co może być nieoptymalne dla dużych grafów.
- **Algorytm sortowania przez kopcowanie** jest również szybki i efektywny, a jego złożoność obliczeniowa dla n - elementowej tablicy wynosi $\mathcal{O}(n \log n)$ w każdym przypadku, co czyni go stabilniejszym niż algorytm sortowania szybkiego.

Dlatego też patrząc na pesymistyczną złożoność tych algorytmów zdecydowałyśmy się na algorytm HeapSort.

4.1.1 Sortowanie przez kopcowanie (HeapSort)

Pseudokod naszej implementacji sortowania przez kopcowanie:

```

procedure heap_sort(arr)
    n = length(arr)
    build_heap(arr, n)
    for i = n-1 to 1
        swap(arr[i], arr[0])
        heapify(arr, i, 0)

procedure build_heap(arr, n)
    for i = n//2-1 to 0
        heapify(arr, n, i)

procedure Heapify(arr, n, i)
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]
        largest = l
    if r < n and arr[r] > arr[largest]
        largest = r
    if largest != i
        swap(arr[i], arr[largest])
        heapify(arr, n, largest)

```

Algorytm heap sort działa następująco:

1. Tworzy kopiec z elementów tablicy za pomocą funkcji Heapify.
2. Iteruje przez cały kopiec (od końca tablicy do indeksu 1).
3. Zamienia największy element w kopcu (czyli pierwszy element tablicy) z ostatnim elementem tablicy.
4. Przeprowadza funkcję Heapify na reszcie kopca (od indeksu 0 do indeksu i).
5. Powtarza kroki 2-4, aż do uzyskania posortowanej tablicy.

Funkcja Heapify działa następująco:

1. Określa największy element na podstawie obecnie sprawdzanego elementu, lewego dziecka i prawego dziecka.
2. Jeśli największy element nie jest obecnym elementem, to zamienia je miejscami.
3. Rekurencyjnie wywołuje funkcję Heapify na podkopcu utworzonym po zamianie elementów.

4.1.2 Losowanie liczby miast i dróg

Aby uzyskać rozwiązanie należy wprowadzić następujące warunki dotyczące liczby krawędzi i wierzchołków:

- liczby n i m muszą być liczbami naturalnymi;
- jeżeli $m < n$ to autostrada nie będzie łączyć wszystkich miast, co przedstawione jest za pomocą warunku:

```
if m < n:  
    print("Jest mniej dróg niż miast, więc nie da się utworzyć  
          → autostrady, graf nie jest spójny!")
```

- graf nie może mieć wielokrotnych krawędzi, zatem w takim przypadku przyjmujemy, że m równa się maksymalnej możliwej liczbie krawędzi w grafie prostym, co pokazuje poniższy fragment kodu:

```
if m > n * (n - 1) // 2:  
    m = n * (n - 1) // 2
```

- z każdego miasta musi być chociażby jedna droga do innego. Ten warunek jest sprawdzany przy tworzeniu macierzy dróg (oznaczona jako B), której wartości oznaczają koszt drogi pomiędzy miastem o numerze danego wiersza a miastem o numerze odpowiedniej kolumny, co obrazuje poniższy kod:

```
for row in B:
    if all(element == 0 for element in row) == True:
        print("Nie ma połączenia pomiędzy dwoma miastami!")
```

4.2 Język Programowania

Zadanie projektowe zostało zrealizowane w środowisku PyCharm. Prezentacja projektu zostanie zrealizowana wykorzystując funkcje plików o rozszerzeniu *exe*.

4.3 Pakiety

W trakcie tworzenia tego projektu zostały wykorzystane następujące pakiety:

- random - generator liczb pseudolosowych został użyty do wyznaczenia wartości m i n ;
- numpy - biblioteka służąca do przeprowadzania operacji na macierzach i ciągach, która została użyta do wygenerowania macierzy zerowej o danych wymiarach;
- matplotlib - biblioteka umożliwiająca rysowanie wykresów, została wykorzystana do rysowania grafów
- networkx - pakiet służący do złożonej analizy grafów. Do tworzenia grafów zostały głównie użyte następujące funkcje z tego pakietu:
 - Graph() tworząca graf;
 - draw_networkx_edges() rysująca krawędzie grafu;
 - draw_networkx_edge_labels() dopisująca wagę krawędzi;
 - draw() rysująca wierzchołki.

5 Złożoność algorytmu

W naszej implementacji korzystamy z oznaczeń:

- n - liczba wierzchołków (miasta)
- m - liczba krawędzi z wagami (drogi)

5.1 Złożoność obliczeniowa

1. Operacje na zbiorach rozłącznych.

- MAKE-SET;

Złożoność obliczeniowa metody MAKE-SET wynosi $\mathcal{O}(n)$, ponieważ jest ona wywoływana dla każdego wierzchołka w grafie i polega na ustaleniu rodzica wierzchołka na samym sobie.

Złożoność ta nie ma bezpośredniego wpływu na ogólną złożoność algorytmu Kruskala, ponieważ jest wywoływana tylko raz na początku, tworząc jednoelementowe zbiory rozłączne. Wpływa ona jednak na ogólną złożoność operacji na zbiorach rozłącznych, ponieważ tworzy ona zbiory rozłączne dla wszystkich wierzchołków w grafie.

- **FIND-SET;**

Metoda FIND-SET ma złożoność $\mathcal{O}(\log n)$ w najgorszym przypadku, ponieważ może być wywoływana rekurencyjnie do głębokości drzewa reprezentującego zbiory rozłączne.

- **UNION;**

Metoda UNION ma złożoność $\mathcal{O}(1)$, ponieważ polega tylko na ustaleniu rodzica jednego wierzchołka jako rodzica drugiego wierzchołka.

2. Sortowanie przez kopcowanie (HeapSort).

Złożoność obliczeniowa algorytmu sortowania przez kopcowanie (HeapSort) wynosi $\mathcal{O}(m \log m)$ dla wszystkich przypadków (najlepszy, średni, najgorszy).

Składowe HeapSort:

- **Heapify;**

Złożoność obliczeniowa funkcji heapify w algorytmie sortowania przez kopcowanie wynosi $\mathcal{O}(\log m)$.

Krok po kroku:

m - liczba elementów w tablicy (wag krawędzi), na której przeprowadzamy Heapify.

Każdy węzeł w kopcu (szczególnym przypadku drzewa binarnego) ma co najwyżej dwoje dzieci, więc poziom drzewa jest logarytmiczny względem liczby elementów w tablicy – wysokość kopca $h = \lceil \log(m + 1) \rceil$. W rezultacie, złożoność obliczeniowa rekurencji wynosi $\mathcal{O}(\log m)$.

Pozostałe operacje w funkcji heapify to operacje porównania i zamiany elementów, które wykonujemy co najwyżej dla każdego węzła drzewa. Złożoność obliczeniowa tych operacji wynosi $\mathcal{O}(1)$.

Łączna złożoność obliczeniowa funkcji Heapify wynosi $\mathcal{O}(m \log m)$, ponieważ złożoność obliczeniowa rekurencji jest dominująca w stosunku do złożoności obliczeniowej pozostałych operacji.

- **Budowa kopca (build heap);**

Aby utworzyć kopiec binarny, należy przejść przez każdy z m wierzchołków w drzewie i sprawdzić, czy spełnia on warunki kopca. Jeśli nie, należy go przemieścić na odpowiednie miejsce w drzewie, co może wymagać przesuwania innych

wierzchołków.

Złożoność obliczeniowa stworzenia drzewa binarnego wynosi $\mathcal{O}(m)$. W najgorszym przypadku, sprawdzając czy warunek kopca jest spełniony będzie trzeba przenieść element z korzenia do liścia co wiąże się z $\log(m)$ porównaniami i zamianami miejsc węzłów.

Budując kopiec wykonujemy tą operację dla $\lfloor \frac{m}{2} \rfloor$ elementów (kopiec budujemy “od dołu”, tj. od węzłów o najwyższych indeksach, a ponieważ wszystkie liście są kopcami, więc rozpoczynamy procedurę od najniższego węzła “nie-liścia” – węzła $\lfloor \frac{m}{2} \rfloor$). Procedurą Heapify poprawiamy kolejno podkopce o korzeniach $i = \lfloor \frac{m}{2} \rfloor, \lfloor \frac{m}{2} \rfloor - 1, \dots, 1$, więc w najgorszym przypadku budowa kopca ma złożoność $\mathcal{O}(\frac{m}{2} \log m) \approx \mathcal{O}(m \log m)$.

- **Kolejne kroki;**

Podczas każdego z następnych kroków zamieniamy korzeń z ostatnim elementem (skrajnie prawym na ostatnim poziomie), zmniejszamy rozmiar kopca o 1 i wykonujemy Heapify na korzeniu pomniejszonego kopca. Dla każdego elementu, ponownie w najgorszym przypadku mamy $\log m$ porównań i zamian miejsc węzłów. Operację tę powtarzamy dla każdego węzła stąd ostateczna złożoność $\mathcal{O}(m \log m)$.

3. Algorytm Kruskala;

Koszt implementacji algorytmu Kruskala z wykorzystaniem zbiorów rozłącznych i sortowania przez kopcowanie to $\mathcal{O}(m \log m)$, gdzie m to liczba wszystkich krawędzi w grafie. Złożoność ta jest determinowana przez sortowanie krawędzi grafu za pomocą funkcji HeapSort, która ma złożoność $\mathcal{O}(m \log m)$. Algorytm Kruskala działa poprzez sortowanie krawędzi i dodawanie ich do drzewa rozpinającego, dlatego sortowanie jest kluczowym elementem jego złożoności.

Ponadto złożoność obliczeniowa operacji na zbiorach rozłącznych ma wpływ na ogólną złożoność algorytmu. W tej implementacji złożoność obliczeniowa operacji na zbiorach rozłącznych wynosi $\mathcal{O}(log n)$ w najgorszym przypadku, ponieważ metoda find jest wywoływana rekurencyjnie.

5.2 Złożoność pamięciowa

1. Sortowanie przez kopcowanie

Złożoność pamięciowa naszej implementacji Heap Sort jest stała i wynosi $\mathcal{O}(1)$, ponieważ nie wymaga ona dodatkowej pamięci poza tablicą, która jest sortowana. Algorytm nie tworzy żadnych dodatkowych zmiennych ani nie wymaga użycia dodatkowych struktur danych.

Złożoność pamięciowa jest zwykle mierzona względem rozmiaru danych wejściowych. Jeśli tablica, która jest sortowana, jest bardzo duża, to może wymagać dużo pamięci, aby ją przechować. W takim przypadku złożoność pamięciowa algorytmu Heap Sort będzie wynosić $\mathcal{O}(m)$, gdzie m to rozmiar tablicy (wag krawędzi grafu).

2. Algorytm Kruskala ze strukturą zbiorów rozłącznych

W przypadku podanej implementacji algorytmu Kruskala, gdzie używamy klasy DisjointSet do przechowywania zbiorów rozłącznych złożoność pamięciowa implementacji wynosi $\mathcal{O}(n + m)$, gdzie n to liczba wierzchołków w grafie, a m to liczba krawędzi.

Jest tak ponieważ:

Na przechowanie struktury danych reprezentujących zbiory rozłączne potrzebna złożoność pamięciowa wynosi $\mathcal{O}(n)$, gdzie n to liczba elementów w zbiorach rozłącznych. Każdy element jest reprezentowany przez pojedynczy węzeł w strukturze drzewa, a więc potrzebna jest pamięć na przechowywanie tego węzła.

Oprócz tego algorytm Kruskala wymaga pamięci na przechowywanie tablicy z krawędziami grafu oraz tablicy zawierającej wierzchołki MST. Złożoność pamięciowa tych tablic wynosi $\mathcal{O}(m)$, i $\mathcal{O}(n)$ odpowiednio.

Literatura

- [1] Dr Anna Maria Radzikowska, prezentacje z wykładów Algorytmy i Struktury Danych.
- [2] <https://pl.wikipedia.org/wiki/AlgorytmKruskala>