POLITECHNIKA WARSZAWSKA

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

ANALIZA ALGORYTMÓW MST II

Autorzy:
Joanna Maciak
Kacper Dominik

Prowadzący: dr inż. Sebastian Kozłowski

Warszawa 2018

Spis treści

1. Założenia projektowe

Tematem projektu jest **przeprowadzenie eksperymentalnej analizy czasu wykonywania dwóch algorytmów wyznaczania najlżejszego drzewa rozpinającego dla różnych klas grafów**.

Wybranymi algorytmami, będącymi przedmiotem tematu niniejszego projektu, są:
– Algorytm Kruskala
– Algorytm Prima

Językiem programowania, wybranym do implementacji powyższych algorytmów jest język Java.

2. Opis i implementacja algorytmów

Problem (zdefiniowany na podstawie [2]):
Znalezienie takiego podzbioru krawędzi spójnego grafu nieskierowanego ważonego, który zapewnia połączenie każdego wierzchołka grafu z dowolnym innym i ponadto posiada najmniejszą możliwą sumę wag krawędzi.
Taki podzbiór nie może zawierać żadnego cyklu, a zatem jest drzewem i musi zawierać dokładnie *n-1* krawędzi dla grafu o *n* wierzchołkach. Drzewo takie ze względu na minimalną sumę wag nazywane jest minimalnym drzewem rozpinającym.

2.1. Algorytm Prima

Dane wejściowe:
Spójny ważony graf nieskierowany, zawierający n wierzchołków, gdzie $n \geqslant 2$.
Graf jest reprezentowany przez macierz sąsiedztwa *graph*, gdzie element *graph[i][j]* reprezentuje wagę krawędzi łączącej wierzchołki *i* oraz *j*.

2.1.1. Pseudokod zaimplementowanego algorytmu

1. Inicjalizacja
    - Utwórz pusty zbiór zawierający krawędzie minimalnego drzewa rozpinającego: *minSpanningTree*.
    - Zainicjalizuj pustą listę odwiedzonych wierzchołków: *visitedVertices*.
    - Utwórz pustą listę krawędzi incydentnych do odwiedzonych wierzchołków: *adjacencyEdges*.
2. Losuj wierzchołek początkowy: *startV*.

3. Oznacz wierzchołek początkowy jako odwiedzony poprzez dodanie go do listy wierzchołków odwiedzonych *visitedVertices*.

4. Pobierz do zmiennej *adjacencyEdges* posortowaną względem wag listę wszystkich krawędzi incydentnych do wierzchołka *startV*.

5. Dopóki lista *visitedVertices* nie zawiera wszystkich wierzchołków grafu:

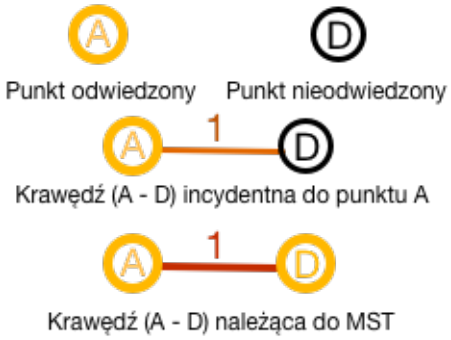
a. Zainicjalizuj zmienną aktualnie rozpatrywanej krawędzi – *minEdge* – poprzez pobranie najmniejszego elementu listy *adjacencyEdges*.

b.1. Jeśli lista odwiedzonych wierzchołków *visitedVertices* nie zawiera jeszcze wierzchołka końcowego aktualnie rozpatrywanej krawędzi *minEdge*:
– dodaj krawędź *minEdge* do zbioru krawędzi minimalnego drzewa rozpinającego *minSpanningTree*
– przypisz zmiennej *startV* wartość wierzchołka końcowego krawędzi *minEdge*
– oznacz wierzchołek końcowy krawędzi *minEdge* jako odwiedzony
– usuń krawędź uwzględnioną w *minSpanningTree* – *minEdge* – z listy krawędzi incydentnych do odwiedzonych wierzchołków – ustal aktualną listę nietworzących cyklu krawędzi incydentnych do wierzchołków odwiedzonych

b.2. Jeśli lista odwiedzonych wierzchołków *visitedVertices* zawiera już wierzchołek końcowy aktualnie rozpatrywanej krawędzi *minEdge*:
– usuń krawędź tworzącą cykl z listy krawędzi incydentnych do odwiedzonych wierzchołków

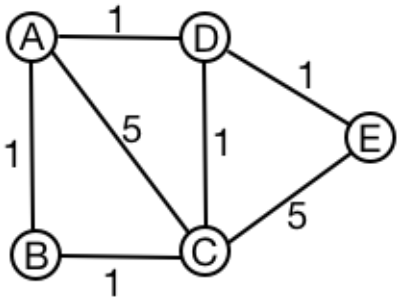
2.1.2. Zasada działania zaimplementowanego algorytmu na przykładzie

Poniżej znajduje się legenda oznaczeń wykorzystanych w opisie działania algorytmu na przykładzie.



rys. 2.1. Oznaczenia wykorzystane w opisie

Dany jest graf widoczny na rys.2.2. Wierzchołki oznaczono wielkimi literami alfabetu, natomiast krawędziom nadano wagi $w \subseteq \{1,5\}$.



rys. 2.2. Graf wejściowy.

Inicjalizacja

Zbiór: *minSpanningTree* ={ };

Lista wierzchołków odwiedzonych: *visitedVertices* = [ ];

Lista krawędzi incydentnych: *adjacencyEdges* = [ ];

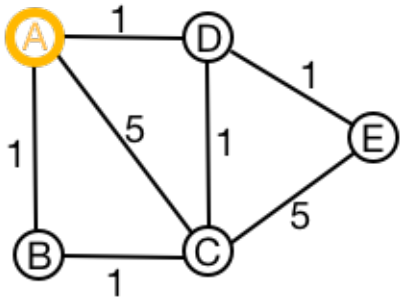
Krok 1.

Losowanie wierzchołka początkowego. W tym przypadku niech będzie to wierzchołek **A**.

$$startV = \text{A}$$

Krok 2.

Oznaczenie wierzchołka *startV* jako odwiedzony.



rys. 2.3. Prim – Krok 2.

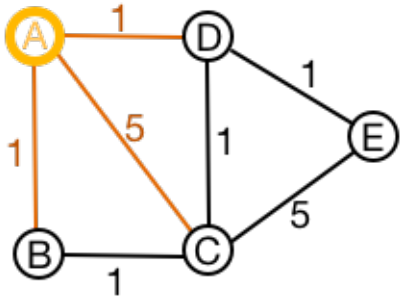
Zbiór: *minSpanningTree* ={ };

Lista wierzchołków odwiedzonych: *visitedVertices* = [A];

Lista krawędzi incydentnych: *adjacencyEdges* = [ ];

Krok 3.

Pobranie listy krawędzi incydentnych do wierzchołka *startV*, które nie tworzą cyklu.



rys. 2.4. Prim – Krok 3.

Zbiór: *minSpanningTree* ={ };

Lista wierzchołków odwiedzonych: *visitedVertices* = [A];
Lista krawędzi incydentnych: *adjacencyEdges* = [(A – D), (A – B), (A – C)];

Krok 4.

Inicjalizacja *minEdge* poprzez wybór pierwszego elementu posortowanej rosnąco listy krawędzi incydentnych *adjacencyEdges*.

$$minEdge = (A - D)$$

Krok 5.

Lista *visitedVertices* nie zawiera jeszcze wierzchołka końcowego krawędzi *minEdge* (D), dlatego:
– Brak cyklu z krawędzią *minEdge*
– Dodanie krawędzi *minEdge* do zbioru krawędzi minimalnego drzewa rozpinającego *minSpanningTree*

Zbiór: *minSpanningTree* ={ (A – D) };

– przypisanie zmiennej *startV* wartości wierzchołka końcowego krawędzi *minEdge*

$$startV = D$$

– oznaczenie wierzchołka końcowego krawędzi *minEdge* jako odwiedzony
– usunięcie krawędzi uwzględnionej w *minSpanningTree* – *minEdge* – z listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

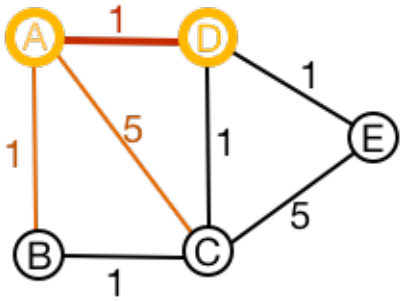
Lista wierzchołków odwiedzonych: *visitedVertices* = [A , D];
Lista krawędzi incydentnych: *adjacencyEdges* = [(A – B), (A – C)];

– aktualizacja listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

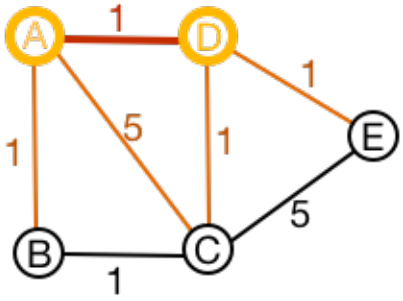
Lista krawędzi incydentnych: *adjacencyEdges* = [(A – B), (D – C), (D – E), (A – C)];

Krok 6.

Aktualizacja *minEdge* poprzez wybór pierwszego elementu posortowanej rosnąco listy krawędzi incydentnych *adjacencyEdges*.



rys. 2.5. Prim – Krok 5.



rys. 2.6. Prim – Krok 5 - aktualizacja listy krawędzi.

$$minEdge = (A - B)$$

Krok 7.

Lista *visitedVertices* nie zawiera jeszcze wierzchołka końcowego krawędzi *minEdge* (B), dlatego:

– Brak cyklu z krawędzią *minEdge*

– Dodanie krawędzi *minEdge* do zbioru krawędzi minimalnego drzewa rozpinającego *minSpanningTree*

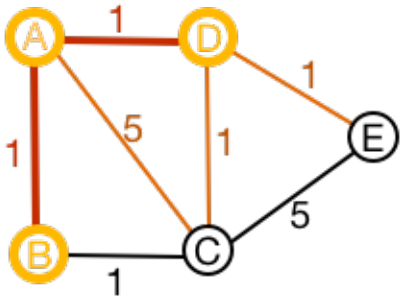
Zbiór: *minSpanningTree* ={ (A – D), (A – B) };

– przypisanie zmiennej *startV* wartości wierzchołka końcowego krawędzi *minEdge*

$$startV = B$$

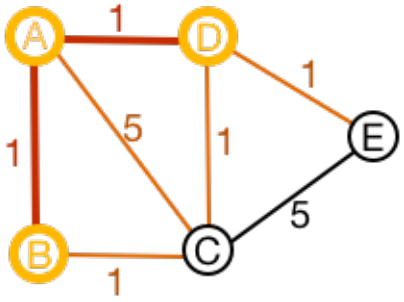
– oznaczenie wierzchołka końcowego krawędzi *minEdge* jako odwiedzony
– usunięcie krawędzi uwzględnionej w *minSpanningTree* – *minEdge* – z listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

Lista wierzchołków odwiedzonych: *visitedVertices* = [A , D , B];
Lista krawędzi incydentnych: *adjacencyEdges* = [(D – C), (D – E), (A – C)];



rys. 2.7. Prim – Krok 7.

– aktualizacja listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*



rys. 2.8. Prim – Krok 7 - aktualizacja listy krawędzi.

Lista krawędzi incydentnych: *adjacencyEdges* = [(B – C), (D – C), (D – E), (A – C)];

Krok 8.

Aktualizacja *minEdge* poprzez wybór pierwszego elementu posortowanej rosnąco listy krawędzi incydentnych *adjacencyEdges*.

$$minEdge = (B - C)$$

Krok 9.

Lista *visitedVertices* nie zawiera jeszcze wierzchołka końcowego krawędzi *minEdge* (C), dlatego:

– Brak cyklu z krawędzią *minEdge*

– Dodanie krawędzi *minEdge* do zbioru krawędzi minimalnego drzewa rozpinającego *minSpanningTree*

Zbiór: *minSpanningTree* ={ (A – D), (A – B), (B – C) };

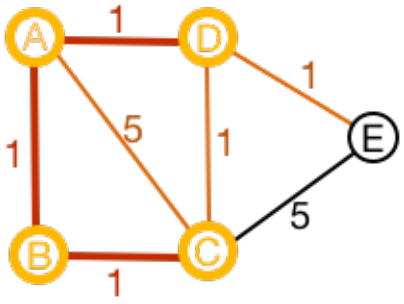
– przypisanie zmiennej *startV* wartości wierzchołka końcowego krawędzi *minEdge*

$$startV = C$$

– oznaczenie wierzchołka końcowego krawędzi *minEdge* jako odwiedzony

– usunięcie krawędzi uwzględnionej w *minSpanningTree* – *minEdge* – z listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

Lista wierzchołków odwiedzonych: *visitedVertices* = [A , D , B, C];
Lista krawędzi incydentnych: *adjacencyEdges* = [(D – C), (D – E), (A – C)];



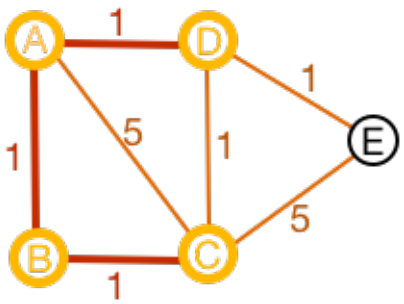
rys. 2.9. Prim – Krok 9.

– aktualizacja listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

Lista krawędzi incydentnych: *adjacencyEdges* = [(D – C), (D – E), (A – C), (C – E)];

Krok 10.

Aktualizacja *minEdge* poprzez wybór pierwszego elementu posortowanej rosnąco listy krawędzi



rys. 2.10. Prim – Krok 9 - aktualizacja listy krawędzi.

incydentnych *adjacencyEdges*.

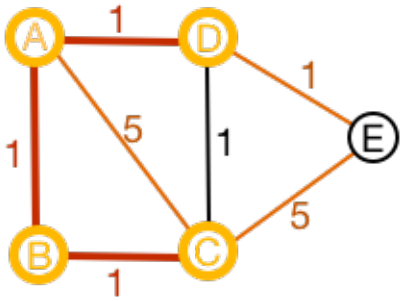
minEdge = (D – C)

Krok 11.

Lista *visitedVertices* zawiera wierzchołek krawędzi *minEdge* (C), dlatego:

– Usunięcie krawędzi *minEdge* z listy krawędzi incydentnych do odwiedzonych wierzchołków

Lista krawędzi incydentnych: *adjacencyEdges* = [(D – E), (A – C), (C – E)];



rys. 2.11. Prim – Krok 11.

Krok 12.

Aktualizacja *minEdge* poprzez wybór pierwszego elementu posortowanej rosnąco listy krawędzi incydentnych *adjacencyEdges*.

$$minEdge = (D - E)$$

Krok 13.

Lista *visitedVertices* nie zawiera jeszcze wierzchołka końcowego krawędzi *minEdge* (E), dlatego:

– Brak cyklu z krawędzią *minEdge*

– Dodanie krawędzi *minEdge* do zbioru krawędzi minimalnego drzewa rozpinającego *minSpanningTree*

Zbiór: *minSpanningTree* ={ (A – D), (A – B), (B – C), (D – E) };

– przypisanie zmiennej *startV* wartości wierzchołka końcowego krawędzi *minEdge*

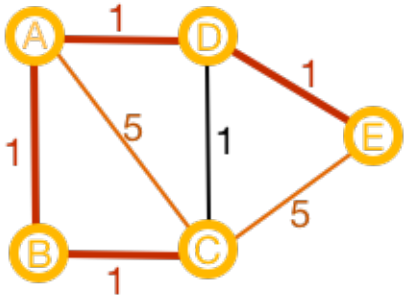
$$startV = E$$

– oznaczenie wierzchołka końcowego krawędzi *minEdge* jako odwiedzony

– usunięcie krawędzi uwzględnionej w *minSpanningTree* – *minEdge* – z listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

Lista wierzchołków odwiedzonych: *visitedVertices* = [A , D , B, C, E];
Lista krawędzi incydentnych: *adjacencyEdges* = [(C – E), (A – C)];

– aktualizacja listy krawędzi incydentnych do odwiedzonych wierzchołków *adjacencyEdges*

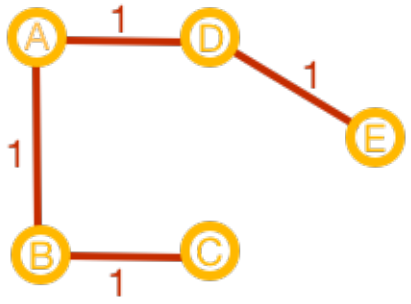


rys. 2.12. Prim – Krok 13.

Lista krawędzi incydentnych: *adjacencyEdges* = [(C – E), (A – C)];

Krok 14.

Lista *visitedVertices* zwiera wszystkie wierzchołki grafu wejściowego. W związku z tym otrzymano następujące drzewo rozpinające:



rys. 2.13. Prim – Krok 14 – otrzymane MST.

2.1.3. Składowe implementacji algorytmu

Składowa	Rodzaj	Opis
rand	Zmienna pomocnicza	Element odpowiedzialny za losowy wybór wierzchołka początkowego
startV	Zmienna typu int	Wierzchołek początkowy minimalnej krawędzi z listy *adjacencyEdgesList*. Na początku przyjmuje wartość losową.
visitedVertices	Lista elementów typu Integer	Lista odwiedzonych wierzchołków grafu
adjacencyEdgesList	Lista elementów typu Edge ArrayList<Edge>	Lista krawędzi incydentnych do odwiedzonych wierzchołków grafu
minSpanningTree	Zbiór Set<Edge>	Zbiór krawędzi, tworzących minimalne drzewo rozpinające
minEdge	Zmienna typu Edge	Obiekt klasy *Edge* (krawędź), zawierający minimalną krawędź z listy *adjacencyEdgesList*
setVertexVisited(int vertex, ArrayList<Integer> list)	metoda	Metoda dodająca wierzchołek *vertex* do listy wierzchołków odwiedzonych *list* jeśli lista ta nie zawiera wierzchołka *vertex*
getAllAdjacencyEdges(int vertex, int[][] graph)	Metoda zwracająca listę obiektów Edge	Metoda odpowiedzialna za ustalenie listy krawędzi incydentnych do odwiedzonych punktów *adjacencyEdgesList*. Dla każdego wierzchołka $i \subseteq V$ grafu weryfikuje czy dany wierzchołek *i* tworzy krawędź z wierzchołkiem *vertex*. Jeśli krawędź taka istnieje i lista *visitedVertices* nie zawiera wierzchołka *i*, to do listy *adjacencyEdgesList* zostaje dodana krawędź (*vertex* – *i*)
minSpanning-Tree.add(minEdge)	Operacja na zbiorze minSpanningTree	Operacja polegająca na dodaniu do zbioru *minSpanningTree* minimalnej krawędzi *minEdge*

tab. 2.1. Składowe implementacji algorytmu Prima

Składowa	Rodzaj	Opis
adjacencyEdges.remove(minEdge)	Operacja na liście *adjacencyEdges*	Operacja polegająca na usunięciu z listy krawędzi *adjacencyEdges* krawędzi *minEdge*
Edge	Klasa	Jest to klasa, której obiekty reprezentują krawędzie grafu. Ich atrybutami są: wierzchołek początkowy i końcowy krawędzi oraz jej waga. Klasa implementuje również metodę odpowiedzialną za sortowanie obiektów Edge znajdujących się w liście (metoda compareTo(Edge arg)) oraz metodę odpowiedzialną za wypisanie danej krawędzi (metoda toString()).

tab. 2.2. Składowe implementacji algorytmu Prima c.d.

Uwagi

- W celu realizacji projektu zaistniała również konieczność implementacji modułu generującego macierz sąsiedztwa reprezentującą graf oraz zapisującego ją do pliku tekstowego na pulpicie. Ponadto należało zaimplementować mechanizm odpowiedzialny za odczyt wygenerowanej macierzy z pliku.
- W procesie implementacji algorytmu Prima bazowano na następujących źródłach: [1] oraz [3]. Źródła te miały wpływ na wybór odpowiednich struktur danych oraz na samo zrozumienie istoty algorytmu.

2.2. Algorytm Kruskala

2.2.1. Pseudokod opisujący działanie algorytmu Kruskala:

Inicjalizacja:

1. Utwórz las L z wierzchołków oryginalnego grafu – każdy wierzchołek jest na początku osobnym drzewem.
2. Utwórz posortowany zbiór S zawierający wszystkie krawędzie oryginalnego grafu.

Tworzenie MST

1. Wybierz i usuń z S jedną z krawędzi o minimalnej wadze.
2. Jeśli krawędź ta łączyła dwa różne drzewa, to dodaj ją do lasu L tak, aby połączyła dwa odpowiadające drzewa w jedno.
3. W przeciwnym wypadku odrzuć ją.
4. Jeżeli wciąż istnieje więcej niż jedno drzewo przejdź do punktu 1.

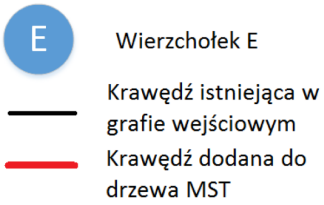
Po zakończeniu algorytmu L jest minimalnym drzewem rozpinającym zgodnie z [4].

2.2.2. Działanie algorytmu na przykładzie

Dany jest graf posiadający następujące składowe:

- Wierzchołki: A, B, C, D, E
- Krawędzie: A B, 1, A D, 1, C D, 1, D E, 1, B C, 1, A C, 5, C E, 5

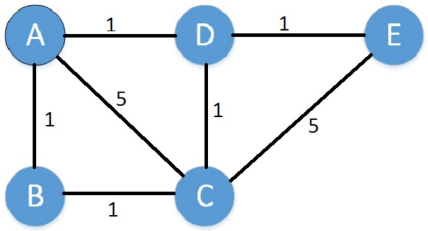
Legenda oznaczeń w poniższym przykładzie:



rys. 2.14. Oznaczenia wykorzystane w opisie algorytmu Kruskala

Przebieg algorytmu

1. Pierwszy etap wykonania algorytmu - ustalenie zbioru wierzchołków S oraz zbioru krawędzi L:
   S: A, B, C, D, E
   L: A B, A D, C D, D E, B C, A C, C E



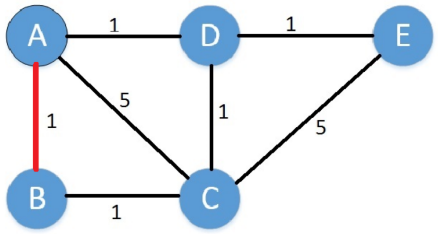
rys. 2.15. Przykładowy graf / pierwszy etap wykonania algorytmu Kruskala

2. Krok 2

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. A B Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, więc dodawana jest krawędź dodawana jest do W.

   S: A, B, C, D, E

   L: A D, C D, D E, B C, A C, C E



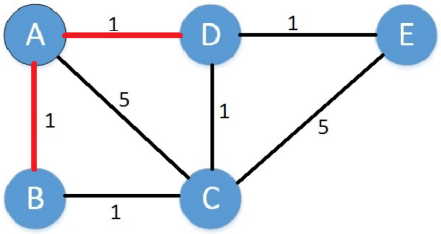
rys. 2.16. Drugi etap wykonania algorytmu Kruskala

3. Krok 3

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. A D Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, D, więc dodawana jest krawędź dodawana jest do W.

   S: A, B, D, C, E

   L: C D, D E, B C, A C, C E



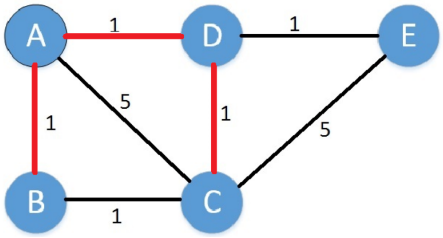
rys. 2.17. Trzeci etap wykonania algorytmu Kruskala

4. Krok 4

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. C D Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, C, D, więc dodawana jest krawędź dodawana jest do W.

   S: A, B, C, D, E

   L: D E, B C, A C, C E



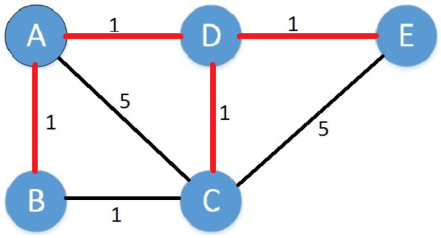
rys. 2.18. Czwarty etap wykonania algorytmu Kruskala

5. Krok 5

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. D E Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, C, D, E, więc dodawana jest krawędź dodawana jest do W. Ponieważ w zbiorze L pozostał tylko jeden zbiór wykonanie algorytmu zostaje zakończone.

   S: A, B, C, D, E

   L: B C, A C, C E



rys. 2.19. Piąty etap wykonania algorytmu Kruskala – uzyskane MST

Jak widać nie zawiera ono żadnych cykli i zawiera tylko krawędzie o minimalnej wadze.

2.2.3. Opis struktur danych i algorytmów

- Dane zostaną pobrane z pliku tekstowego o ustalonym formacie.
- Ponieważ samodzielna implementacja podstawowych algorytmów mija się z celem użyta zostanie jedna z funkcji bibliotecznych języka JAVA.
- Wykrywanie cykli oraz scalanie odbywać się będzie poprzez użycie struktury zbiorów rozłącznych - [5].
- Krawędzie dodane do grafu będą wpisywane do listy w celu zapisania uzyskanego rozwiązania.

2.3. Ogólne założenia dotyczące badań algorytmów

1. Algorytmy zostaną wykonane dla **grafów spójnych, nieskierowanych**
2. Działający program zostanie przetestowany na przypadkach, dla których znane są poprawne rozwiązania
3. Na wejściu każdego z algorytmów dany jest graf w postaci **macierzy wag** o zadanej liczbie wierzchołków N(ang. nodes) oraz E krawędzi (ang. edges), gdzie $1 \leqslant N \leqslant 10000$) oraz $1 \leqslant M \leqslant 100000$.

   Macierz sąsiedztwa

   Graf reprezentujemy za pomocą macierzy kwadratowej *o* stopniu *n*, gdzie n oznacza liczbę wierzchołków w grafie. Macierz tą nazywamy macierzą sąsiedztwa (ang. adjacency matrix). Odwzorowuje ona połączenia wierzchołków krawędziami. Wiersze macierzy sąsiedztwa odwzorowują zawsze wierzchołki startowe krawędzi, a kolumny odwzorowują wierzchołki końcowe krawędzi. Komórka *A[i,j]*, która znajduje się w *i-tym* wierszu i *j-tej* kolumnie odwzorowuje krawędź łączącą wierzchołek startowy vi z wierzchołkiem końcowym v_j. Jeśli *A[i,j]* ma wartość 1, to dana krawędź istnieje. Jeśli *text* ma wartość 0, to wierzchołek v_i nie łączy się krawędzią z wierzchołkiem v_j.
4. Na wyjściu każdego z algorytmów otrzywana zostanie suma wag krawędzi MST
5. W związku z tym, że wejście oraz wyjście obu algorytmów dotyczy tych samych danych, wyniki zostaną porównane

6. Głównym elementem podlegającym porównaniu będzie czas wykonania obu algorytmów
7. **Złożoność czasowa** – ze względu na wykorzystane struktury danych, złożoność czasowa wybranych algorytmów powinna wynosić odpowiednio:
   - Dla algorytmu Prima: $O(E \cdot \log N)$
   - Dla algorytmu Kruskala: $O(E \cdot \log N)$

Spis załączników

1. **Kod źródłowy algorytmu Prima**
2. **Kod źródłowy modułu odpowiedzialnego za odczyt macierzy z pliku**
3. **Link do repozytorium z kodem źródłowym generatora macierzy sąsiedztwa**

https://github.com/joannasia9/Gis-pro

Bibliografia

[1] *http://www.algorytm.org/algorytmy-grafowe/algorytm-prima.html*

[2] *http://algorytmika.wikidot.com/mst*

[3] *http://eduinf.waw.pl/inf/alg/001_search/0141.php*

[4] Jacek Wojciechowski, Krzysztof Pieńkosz: *Grafy i sieci*, Wydawnictwo Naukowe PWN SA, Warszawa 2013

[5] *http://www.ams.org/proc/1956-007-01/S0002-9939-1956-0078686-7/*

Spis rysunków

Załącznik 1. Kod źródłowy algorytmu Prima

```
1  public class PrimAlgorithm{
2  /**
3  * Element losowosci
4  */
5  Random rand;
6  /**
7  * Lista odwiedzonych wierzcholkow
8  */
9  private ArrayList<Integer> visitedVertices;
10
11 /**
12 * Lista wszystkich wierzcholkow
13 */
14 private ArrayList<Edge> adjacencyEdgesList;
15
16
17 /**
18 * Konstruktor
19 * Odpowiada za wykonanie poszczegolnych krokow algorytmu
20 *
21 * @param  graph  macierz sasiedztwa reprezentujaca graf
22 *
23 */
24 public PrimAlgorithm(final int[][] graph){
25 this.adjacencyEdgesList = new ArrayList<>();
26
27 this.visitedVertices = new ArrayList<>();
28
29 /**
30 * Zbior krawedzi
31 * Reprezentuje minimalne drzewo rozpinajace
32 */
33 Set<Edge> minSpanningTree = new HashSet<>();
34
35 /**
36 * Zmienna pomocnicza
37 * Lista krawedzi
38 * Reprezentuje krawedzie incydentne do odwiedzonych wierzcholkow grafu
39 */
40 ArrayList<Edge> adjacencyEdges;
41
42 /**
43 * Zmienna pomocnicza
44 * Element odpowiedzialny za losowy wybor wierzcholka poczatkowego
45 */
46 this.rand = new Random();
47
48 /**
49 * Losowy wybor wierzcholka poczatkowego
50 * Z zakresu od 0 do n-1,
51 * gdzie n-liczba wierzcholkow grafu
```

```
52  */
53  int startV = rand.nextInt(graph.length-1);
54
55  /**
56  * Oznaczenie wierzcholka startowego jako odwiedzony
57  * Dodanie wierzcholka startV do listy odwiedzonych wierzcholkow
58  */
59  setVertexVisited(startV,visitedVertices);
60
61
62  /**
63  * Pobranie do zmiennej adjacencyEdges listy krawedzi incydentnych do startV
64  */
65  adjacencyEdges = getAllAdjacencyEdges(startV,graph, adjacencyEdgesList);
66
67  /**
68  * Petla warunkowa while()
69  * Wykonuje sie dopoki lista odwiedzonych wierzcholkow visitedVertices
70  * nie zawiera wszystkich wierzcholkow grafu
71  */
72  while(visitedVertices.size()!= graph.length) {
73  /**
74  * Zmienna reprezentujaca krawedz z listy krawedzi incydentnych
75  * do punktow odwiedzonych adjacencyEdges, ktora ma minimalna wage
76  */
77  Edge minEdge = adjacencyEdges.get(0);
78  /**
79  * Instrukcja warunkowa if()
80  * Pod warunkiem, ze lista odwiedzonych wierzcholkow visitedVertices
81  * nie zawiera wierzcholka koncowego krawedzi minEdge:
82  *
83  * Do MST zostaje dodana krawedz minEdge
84  * Wierzcholek startV zmienia wartosc na wierzcholek koncowy minEdge
85  * Krawedz minEdge zostaje usunieta z listy krawedzi incydentnych
86  * do rozpatrzenia
87  * Lista krawedzi incydentnych adjacencyEdges zostaje zaktualizowana
88  * o krawedzie
89  * incydentne do wierzcholka startV
90  *
91  * W przeciwnym wypadku:
92  * Krawedz minEdge zostaje usunieta z listy krawedzi do rozpatrzenia
93  */
94  if(!visitedVertices.contains(minEdge.getEnd())){
95  minSpanningTree.add(minEdge);
96  setVertexVisited(minEdge.getEnd(), visitedVertices);
97  startV = minEdge.getEnd();
98  adjacencyEdges.remove(minEdge);
99  adjacencyEdges = getAllAdjacencyEdges(startV,graph,adjacencyEdgesList);
100 } else adjacencyEdges.remove(minEdge);  }
101 /**
102 * Wypisanie otrzymanego MST w konsoli
103 */
104 printMST(minSpanningTree); }
```

```
105 /**
106 * @param vertex
107 * @param list
108 * Oznacza wierzcholek jako odwiedzony poprzez dodanie wierzcholka vertex
109 * do listy wierzcholkow odwiedzonych list
110 */
111 private void setVertexVisited(int vertex, ArrayList<Integer> list){
112 /**
113 * Instrukcja warunkowa if()
114 * Blok instrukcji zostaje wykonany, gdy lista wierzcholkow odwiedzonych
115 * nie zawiera jeszcze wierzcholka vertex
116 */
117 if(!list.contains(vertex))
118 list.add(vertex);
119 }
120
121 /**
122 *
123 * @param vertex
124 * @param graph
125 * @return
126 *
127 * Dodaje do listy adjacencyEdgesList krawedzie incydentne
128 * do wierzcholka vertex, ktore nie tworza cyklu
129 */
130 private ArrayList<Edge> getAllAdjacencyEdges(int vertex,
131         int[][] graph, ArrayList<Edge> adjacencyEdgesList){
132
133 /**
134 * Instrukcja iteracyjna for()
135 * Blok instrukcji jest wykonywany dla kazdego wierzcholka grafu
136 */
137 for(int i=0; i<graph.length; i++){
138 /**
139 * Instrukcja warunkowa if()
140 * Operacja dodania krawedzi incydentnej do wierzcholka vertex
141 * zostaje wykonana pod warunkiem, ze:
142 * Krawedz (vertex-i) istnieje, czyli jej waga jest rozna od 0 oraz
143 * Lista odwiedzonych wierzcholkow visitedVertices
144 * nie zawiera jeszcze i-tego wierzcholka
145 */
146 if(graph[vertex][i]!= 0 && !visitedVertices.contains(i))
147   adjacencyEdgesList.add(new Edge(vertex,i,graph[vertex][i]));
148 }
149
150 /**
151 * Sortowanie otrzymanej listy krawedzi wzgledem wag
152 */
153 Collections.sort(adjacencyEdgesList);
154
155 return adjacencyEdgesList;
156 }
157
```

```
158 /**
159 *
160 * @param set
161 *
162 * Wyswietla w konsoli liste krawedzi otrzymanego drzewa
163 * wraz z suma wag krawedzi
164 */
165 private void printMST(Set<Edge> set){
166 int sum = 0;
167 for(Edge item : set){
168 System.out.println(item);
169 sum+=item.getWeight();
170 }
171 System.out.println("SUMA WAG WYNOSI: " + sum);
172 }
```

Załącznik 1.a. Kod źródłowy klasy Edge

```
1  /**
2  * Klasa Edge, ktorej obiekty reprezentuja krawedzie grafu
3  */
4  private class Edge implements Comparable<Edge> {
5  /**
6  * Wierzcholek poczatkowy krawedzi
7  */
8  private int start;
9  /**
10 * Wierzcholek koncowy krawedzi
11 */
12 private int end;
13 /**
14 * Waga krawedzi (start - end)
15 */
16 private int weight;
17
18 /**
19 *
20 * @param s
21 * @param e
22 * @param w
23 *
24 * Konstruktor krawedzi
25 */
26 Edge(int s, int e, int w){
27 setStart(s);
28 setEnd(e);
29 setWeight(w);
30 }
31
32 private int getWeight(){
33 return weight;
34 }
```

```
35
36 private int getStart() {
37 return start;
38 }
39
40 private void setStart(int start) {
41 this.start = start;
42 }
43
44 private int getEnd() {
45 return end;
46 }
47
48 private void setEnd(int end) {
49 this.end = end;
50 }
51
52 private void setWeight(int weight) {
53 this.weight = weight;
54 }
55
56 /**
57 *
58 * @param e
59 * @return
60 * Metoda odpowiedzialna za porownywanie krawedzi
61 * Wykorzystywana podczas sortowania krawedzi w liscie
62 * @see Collections
63 */
64 @Override
65 public int compareTo(Edge e) {
66 if (getWeight() < e.getWeight()){
67 return -1;
68 } else if (getWeight() > e.getWeight()) {
69 return 1;
70 }
71 return 0;
72 }
73
74 /**
75 * @return
76 *
77 * Metoda odpowiedzialna za postac wypisanej w konsoli krawedzi
78 */
79 @Override
80 public String toString() {
81 return String.format("%d--%d\tw:%d ", start, end, weight);
82 }
83 }
```

Załącznik 2. Kod źródłowy modułu odpowiedzialnego za odczyt macierzy z pliku

```
import java.io.*;

public class GraphFileExtractor {
private Graph graph;
private int[][] adjacencyMatrix;

public void getFileContent(int iterator) {
String home = System.getProperty("user.home");
File file = new File(home + File.separator + "Desktop"
          + File.separator + "G" + iterator + ".txt");

try {
BufferedReader br = new BufferedReader(new FileReader(file.getPath()));
String line = br.readLine();
initiateMatrix(line);

int i = 0;
while (line != null) {
parseLine(i,line);
line = br.readLine();
i+=1;
}

} catch (IOException e) {
e.printStackTrace();
}
}

private void parseLine(int i, String line){
String delim = "\t";
String[] tokens = line.split(delim);

for(int j=0; j<tokens.length;j++){
if(Integer.parseInt(tokens[j])!=0)
graph.addEdge(new Edge(i,j,Integer.parseInt(tokens[j])));
adjacencyMatrix[i][j] = Integer.parseInt(tokens[j]);
}

}

private void initiateMatrix(String line){
String delim = "\t";
String[] tokens = line.split(delim);
graph = new Graph(tokens.length);
adjacencyMatrix = new int[tokens.length][tokens.length];
}

public Graph getGraph(){
return graph;
}

```

```
52  public int[][] getAdjacencyMatrix(){
53  return adjacencyMatrix;
54  }
55  }
```