`Sprawozdanie końcowe`
Generacja grafu i odnalezienie najkrótszej ścieżki pomiędzy węzłami - `graph` - Java

Ulyana Petrashevich, Inga Maziarz

08.06.2022

Spis treści

Opis teoretyczny zagadnienia

Program `Kratka` służy do wyszukiwania najkrótszej ścieżki w grafie ważonym nieskierowanym. Program zawiera funkcje wczytującą i wypisującą graf z/do pliku oraz generator grafu spójnego i niespójnego. Za wyszukiwanie najkrótszej ścieżki odpowiedzialne są funkcje `bfs` oraz `dijkstra`. `BFS`, czyli algorytm przeszukiwania wszerz, działa następująco:

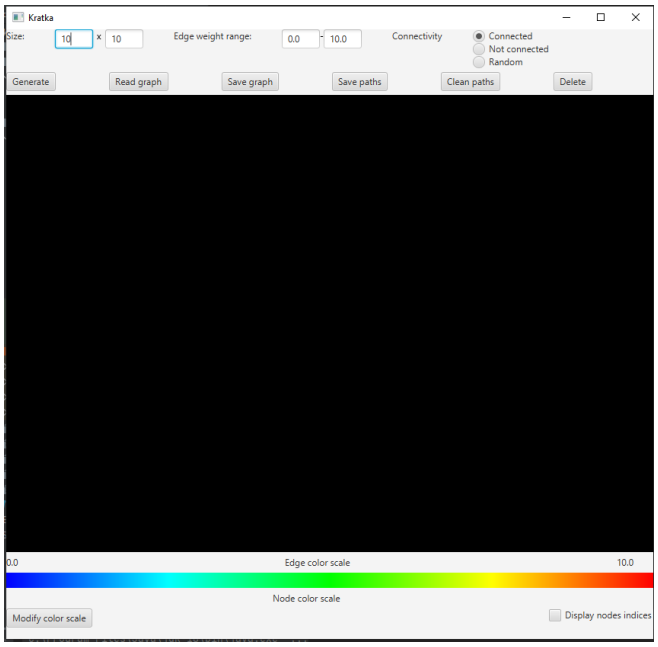
- Węzeł początkowy oznaczany jest jako odwiedzony. Do kolejki dodawane są sąsiadujące z nim węzły (w kolejności od węzła z najmniejszym indeksem).
- Odwiedzony zostaje następny węzeł w kolejce. Postępowanie jest analogiczne do wcześniejszego; do kolejki zostają dodane węzły sąsiadujące z obecnym, jednak tylko te, które nie zostały odwiedzone wcześniej.
- Proces jest powtarzany aż do momentu odwiedzenia wszystkich węzłów z kolejki. Jeżeli na końcu wszystkie węzły w grafie zostały odwiedzone, oznacza to, że przeszukiwany graf jest spójny.

Jeżeli `bfs` zwróci wartość oznaczającą spójność grafu, to program przechodzi do kolejnego kroku, którym jest znalezienie najkrótszej ścieżki algorytmem Dijkstry, który działa w poniższy sposób:

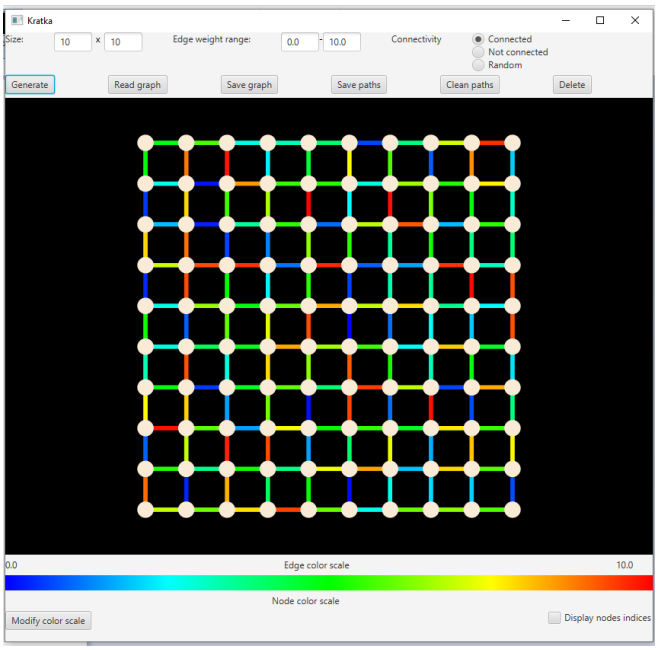
- Długości przy węźle początkowym otrzymują wartość 0. Długość ścieżki do każdego innego węzła zostaje ustawiona na nieskończoność.
- Oznaczamy węzeł początkowy jako odwiedzony. Dla każdego jego sąsiada zostaje przypisana długość równa wadze krawędzi między nimi.
- Nieodwiedzony węzeł o najmniejszej przypisanej długości zostaje oznaczony jako odwiedzony. Dla każdego jego sąsiada zostaje obliczona wartość równa sumie długości przy obecnym węźle i wagi krawędzi między nimi. Jeżeli znaleziona wartość jest mniejsza niż przypisana do sąsiada, to zostaje ona zamieniona.
- Poprzedni krok jest powtarzany aż do odwiedzenia wszystkich węzłów. Ostatecznie każdy węzeł (w tym wybrany jako końcowy) ma przypisaną długość najkrótszej ścieżki od węzła początkowego. Zapamiętana ścieżka może zostać wypisana na ekran.

Opis wywołania i wygląd interfejsu

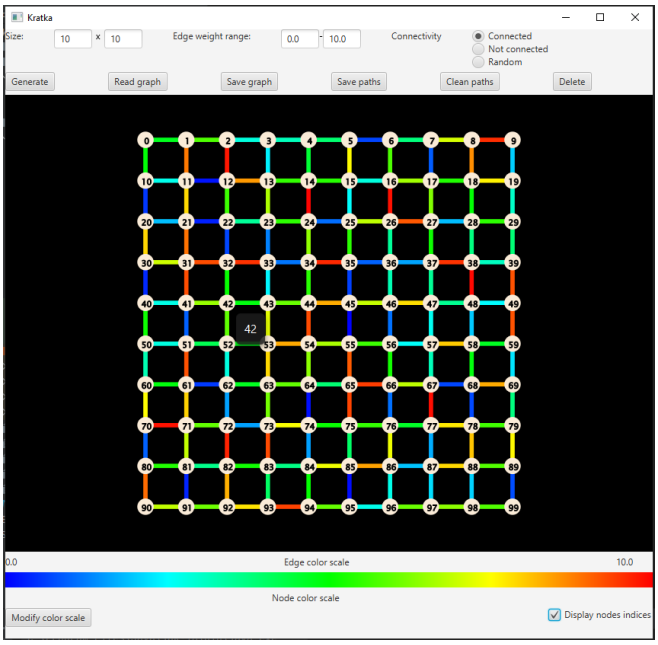
W celu uruchomienia programu należy pobrać pliki źródłowe zachowując pierwotną strukturę podziału plików. Program należy skompilować i uruchomić np. przez środowisko IntelliJ IDEA. Dalsza interakcja pomiędzy użytkownikiem a programem odbywa się za pomocą interfejsu graficznego.



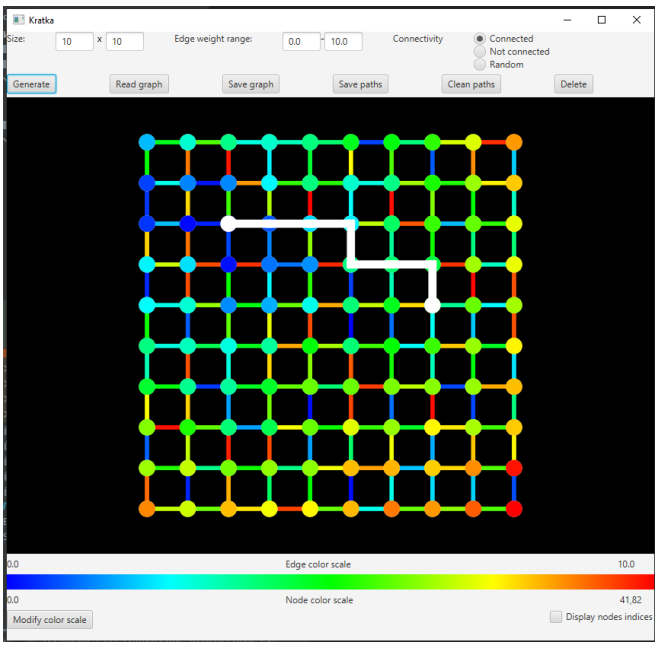
Rys. 1: Wygląd głównego okna interfejsu



Rys. 2: Wygląd głównego okna interfejsu po narysowaniu grafu



Rys. 3: Wygląd okna z pokazanymi numerami wierzchołków



Rys. 4: Wygląd głównego okna interfejsu po narysowaniu ścieżki

W górnym panelu znajdują się pola tekstowe, pozwalające wprowadzić pożądane cechy generowanego grafu lub będą wyświetlane parametry wczytanego grafu, w tym rozmiar, zakres wag krawędzi i spójność i przyciski, pozwalające uruchomić odpowiedni algorytm:

- Generate

  Przycisk wczyta podane przez użytkownika w polach tekstowych i polu wyboru dane, sprawdzi je i wygeneruje odpowiedni graf. W przypadku, gdy poprzednio już został wygenerowany/wczytany graf lub wyliczona ścieżka, poprzednie dane zostaną usunięte.

- Read graph

  Przycisk uruchomi okno wyboru pliku. Postać pliku została opisana w podrozdziale **"Widok pliku"**. W przypadku udanego wczytania grafu program narysuje graf i ustawi odpowiednie wartości do pól tekstowych, etykiet i polu wyboru. W przypadku, gdy poprzednio już został wygenerowany/wczytany graf lub wyliczona ścieżka, poprzednie dane zostaną usunięte.

- Save graph

  Przycisk uruchomi okno wyboru ścieżki do utworzenia pliku i zapisanie do niego grafu w postaci tekstowej. Nie spowoduje usunięcia żadnych danych.

- Save paths

  Przycisk uruchomi okno wyboru ścieżki do utworzenia pliku i zapisanie do niego wszystkich wyznaczonych na rysunku ścieżek w postaci tekstowej. Nie spowoduje usunięcia żadnych danych.

- Clean paths

  Przycisk uruchomi wyczyszczenie wszystkich wyliczonych ścieżek i kosztów dojścia. Lecz nie spowoduje usunięcia grafu.

- Delete

  Przycisk spowoduje usunięcie wszystkich poprzednio wyliczonych danych, w tym grafu, kosztów i ścieżek.

Większą część okna zajmuje pole do wyświetlania grafu w postaci graficznej. Interakcja z programem jest możliwa nie tylko poprzez przyciski, ale również poprzez poniższe akcje dostępne na tym polu.

- **Najechanie myszką** na wierzchołek spowoduje pokazanie jego indeksu.
- **Kliknięcie lewym klawiszem** myszki w okolicy wybranego wierzchołka spowoduje uruchomienie się algorytmu Dijkstry i przypisanie kolorów do każdego z pozostałych wierzchołków. Kolory odzwierciedlają odległość od wybranego wierzchołka do innych wierzchołków. Dany wierzchołek zostaje

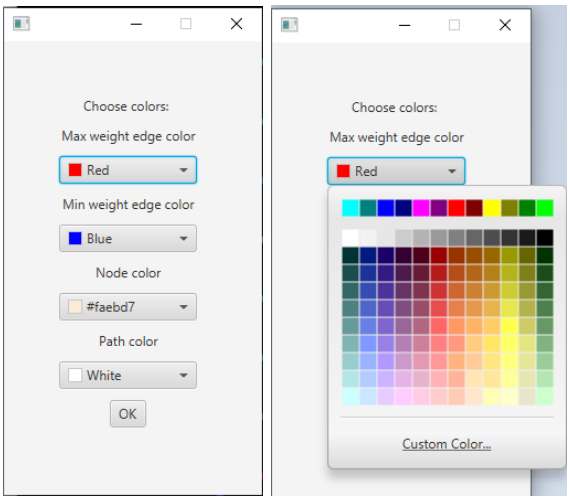
również oznaczony jako wierzchołek startowy w celu późniejszego wyboru ścieżki.

- **Kliknięcie prawym klawiszem myszki** w okolicy innego wierzchołka spowoduje wyliczenie i narysowanie najkrótszej ścieżki do tego wierzchołka od wierzchołka poprzednio wybranego za pomocą lewego klawisza. Możliwe jest wielokrotne wybieranie końcowych wierzchołków; ścieżki będą rysowane od pierwotnie wybranego wierzchołka startowego.

Program pozwala na wyliczenie wielu ścieżek od różnych wierzchołków. W tym celu po wyliczeniu ścieżki od jednego wierzchołka do drugiego, należy znowu wybrać wierzchołek początkowy za pomocą lewego klawisza i wierzchołek końcowy za pomocą prawego klawisza. Wszystkie ścieżki będą jednocześnie widoczne na ekranie aż do momentu ich wyczyszczenia i można będzie zapisać je do pliku.

Dolną część okienka zajmuje mapa kolorów z odpowiadającymi przedziałami wag krawędzi i kosztów dojścia. Przedziały będą się dynamicznie zmieniać wraz ze zmianą danych. Pod mapą znajdują się dwa dodatkowe przyciski akcji:

- przycisk **Modify color scale**, uruchamiający okienko wyboru kolorów wartości minimalnych i maksymalnych dla mapy, koloru wierzchołków i koloru ścieżki. Po zaakceptowaniu zmian kolorów używanych do rysowania grafu, graf i pasek obrazujący skalę kolorów zostają od razu zaktualizowane.
- checkbox **Display nodes indices**, który po zaznaczeniu umożliwa wyświetlanie indeksów każdego z wierzchołków na ekranie. Ze względu na zmniejszenie czytelności wyświeltanych indeksów wierzchołków wraz ze zwiększeniem rozmiaru grafu, włączenie tej opcji jest zalecane tylko dla grafów o wymiarach mniejszych niż 30 x 30.



Rys. 5: Wygląd okienka wyboru kolorów

Widok pliku

Program przyjmuje plik wyłącznie w formacie `.txt`.

Schemat poprawnego formatu danych:

Pierwszą linijkę zajmują dwie liczby: liczba wierszy i liczba kolumn, oddzielone od siebie minimum jedną spacją. Pomiędzy liczbami mogą się znajdować dodatkowe białe znaki (np. tabulator, spacje), ale nie może być to znak nowej linii.

Zaczynając od drugiej linijki, podawane są krawędzie i ich wagi. Numer wierzchołka, od którego zaczyna się krawędź jest równy `numerowi linijki - 2`; np. w 2 linijce, tuż po rozmiarach grafu, będą opisane krawędzie prowadzące do wierzchołka 0. Dane o krawędziach łączących wierzchołek z innymi muszą być podane w odpowiadającej mu linii. Format jest następujący:

```
[indeks wierzchołka] : [waga krawędzi] ...
```

gdzie indeks wierzchołka jest liczbą całkowitą z przedziału [0, [liczba wierszy * liczba kolumn - 1]] i oznacza wierzchołek, do którego prowadzi podana krawędź. Między indeksem wierzchołka a wagą krawędzi musi znaleźć się znak dwukropka `(:)`. Jeśli będzie to inny znak, to program wyświetli komunikat o błędnym formacie wczytywanego grafu. Waga krawędzi jest liczbą rzeczywistą; może być to liczba całkowita lub ułamek dziesiętny, z tą uwagą, że separatorem dziesiętnym musi być kropka `(.)`, a nie na przykład przecinek.

Nie ma potrzeby podawać wagi dwa razy symetrycznie, np. od 0 do 1 i od 1 do 0 – program duplikuje jedną z podanych wag przy wczytaniu.

Jeżeli do wierzchołka nie chcemy przypisywać żadnych krawędzi, pozostawiamy odpowiednią linijkę pustą (choć mogą znajdować się tam białe znaki, np. spacja lub tabulator, z wyłączeniem znaku nowej linii - taka zasada tyczy się do oddzielania od siebie wszystkich danych w pliku).

Przykład poprawnej formy danych:

```
2 2
    2 :2.76543  1 :4.56134

    1 :4.567345
    2 :6.47586
```

W powyższym przykładzie wierzchołek numer 1 nie ma przypisanych krawędzi (choć będzie je miał po zduplikowaniu krawędzi od wierzchołków 0 i 2). Postać symboliczna:

2␣2\n

\t2␣:2.76543␣␣1␣:4.56134\n

\n

\t1␣:4.567345\n

\t2␣:6.47586EOF

Testy

Podczas pisania programu były używane testy jednostkowe.

```
@Test
public void ConnectedGraphTrue1(){


    HashMap<Integer, ArrayList<Edge>> edges = new HashMap<>();
    ArrayList<Edge> tmp = new ArrayList<>();

    Edge edge1 = new Edge(1,1);
    Edge edge2 = new Edge(3,4);
    tmp.add(edge1);
    tmp.add(edge2);
    edges.put(0, (ArrayList<Edge>) tmp.clone());
    tmp.clear();

    edge1 = new Edge(0,1);
    edge2 = new Edge(2,3);
    tmp.add(edge1);
    tmp.add(edge2);
    edges.put(1,(ArrayList<Edge>) tmp.clone());
    tmp.clear();

    edge1 = new Edge(1,3);
    edge2 = new Edge(5,7);
    tmp.add(edge1);
    tmp.add(edge2);
    edges.put(2,(ArrayList<Edge>) tmp.clone());
    tmp.clear();

    edge1 = new Edge(0,4);
    tmp.add(edge1);
    edges.put(3,(ArrayList<Edge>) tmp.clone());
    tmp.clear();

    edge1 = new Edge(5,2);
    tmp.add(edge1);
    edges.put(4,(ArrayList<Edge>) tmp.clone());
    tmp.clear();
```

```
        edge1 = new Edge(4,2);
        edge2 = new Edge(2,7);
        tmp.add(edge1);
        tmp.add(edge2);
        edges.put(5,(ArrayList<Edge>) tmp.clone());
        tmp.clear();

        Graph graph = new Graph(2, 3, edges);

        assertTrue(graph.bfs());
    }

    @Test
    void CheckGenerationOfConnectedGraph() {
        Graph graph = new Graph(3, 2);
        graph.generateGraph(true);
        assertTrue(graph.bfs());
    }

    @Test
    void CheckGenerationOfNotConnectedGraph() {
        Graph graph = new Graph(3, 2);
        graph.generateGraph(false);
        assertFalse(graph.bfs());
    }

    @Test
    void CheckReadingAndSaving(){
        Graph graph = new Graph(3,2);
        graph.generateGraph(true);
        File write;
        write = new File("filename.txt");
        try {
            PrintWriter w = new PrintWriter (write);
            graph.saveGraph(w);
            w.close();
        } catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }
        Graph graph1 = new Graph();
        Reader read;
        try {
            read = new FileReader(write);
            BufferedReader r = new BufferedReader(read);
            graph1.readGraph(r);
            r.close();
```

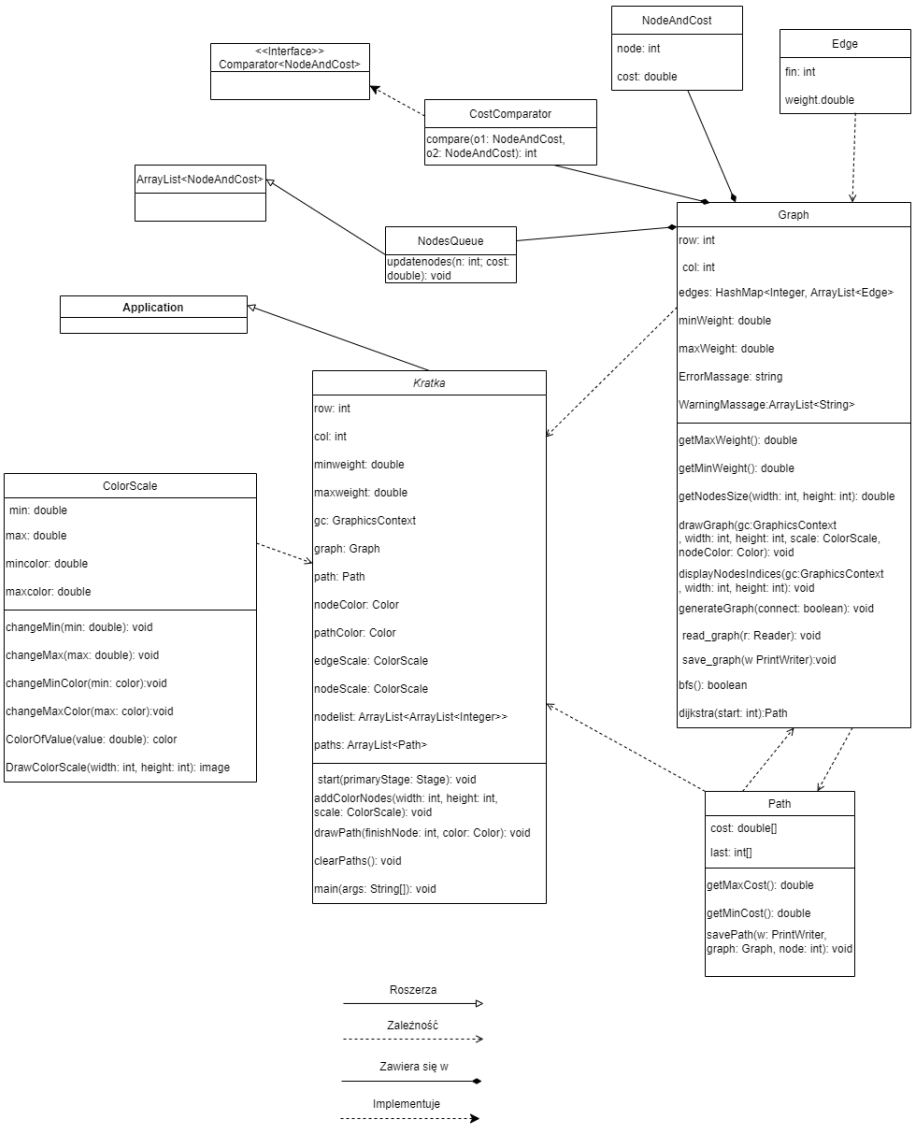
```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        assertEquals(graph.row, graph1.row);
        assertEquals(graph.col, graph1.col);
        for (int i = 0; i < graph.edges.size(); i++)
            for (int j = 0; j < graph.edges.get(i).size(); j++) {
                assertEquals(graph.edges.get(i).get(j).fin,
                                        graph1.edges.get(i).get(j).fin);
                assertEquals(graph.edges.get(i).get(j).weight,
                                        graph1.edges.get(i).get(j).weight);

            }
    }

    @Test
    void checkDijkstra() {
        HashMap<Integer, ArrayList<Edge>> edges = new HashMap<>();
        ArrayList<Edge> tmp = new ArrayList<>();

        Edge edge1 = new Edge(1,1);
        Edge edge2 = new Edge(3,4);
        tmp.add(edge1);
        tmp.add(edge2);
        edges.put(0, (ArrayList<Edge>) tmp.clone());
        tmp.clear();

        edge1 = new Edge(0,1);
        edge2 = new Edge(2,3);
        tmp.add(edge1);
        tmp.add(edge2);
        edges.put(1,(ArrayList<Edge>) tmp.clone());
        tmp.clear();

        edge1 = new Edge(1,3);
        edge2 = new Edge(5,7);
        tmp.add(edge1);
        tmp.add(edge2);
        edges.put(2,(ArrayList<Edge>) tmp.clone());
        tmp.clear();

        edge1 = new Edge(0,4);
        tmp.add(edge1);
        edges.put(3,(ArrayList<Edge>) tmp.clone());
        tmp.clear();
```

```
        edge1 = new Edge(5,2);
        tmp.add(edge1);
        edges.put(4,(ArrayList<Edge>) tmp.clone());
        tmp.clear();

        edge1 = new Edge(4,2);
        edge2 = new Edge(2,7);
        tmp.add(edge1);
        tmp.add(edge2);
        edges.put(5,(ArrayList<Edge>) tmp.clone());
        tmp.clear();

        Graph graph = new Graph(2, 3, edges);

        Path path = graph.dijkstra(0);
        assertArrayEquals(path.cost, new double[]{0.0, 1.0, 4.0, 4.0, 13.0, 11.0});
        assertArrayEquals(path.last, new int [] {-1,0,1,0,5,2});
    }
```

Reszta testów odbywała się poprzez sprawdzanie działania w środowisku graficznym i weryfikowanie, czy program zwraca odpowiednie komunikaty z ostrzeżeniami i błędami oraz czy wygląd grafu jest zgodny z oczekiwanym.

Błędy i wnioski

Podczas przeprowadzania testów w czasie pisania programów zauważyłyśmy, że przedstawienie grafu w postaci macierzy sąsiedztwa nie jest optymalne: za bardzo obciąża projekt. Długość listy odpowiadającej macierzy rosła bardzo szybko wraz ze zwiększaniem rozmiaru grafu - lista posiadała (liczba wierszy * liczba kolumn) do potęgi drugiej elementów. Z tego wynika, że dla grafu o wymiarach 100 x 100 lista zawierała 100 milionów pozycji, a to wymagało długiego czasu na przetworzenie. Dlatego postanowiłyśmy zmienić postać przechowywania grafu. Graf jest przedstawiony w postaci listy z kluczami HashMap, gdzie kluczem jest numer wierzchołka, elementami - lista krawędzi, prowadzonych od niego. Krawędź jest przedstawiona klasą Edge, zawierającą numer wierzchołka końcowego i wagę.



Rys. 6: Aktualny diagram klas

Innym problemem było nakładanie się warstw rysunku w procesie rysowania grafu. Nie udało nam się znaleźć sposobu na utworzenie kontenerów odpowiadającym warstwom, przez co należało zwrócić uwagę, aby funkcje rysujące poszczególne części grafu były wywoływane w odpowiedniej kolejności. Ostatecznie udało się doprowadzić do tego, że rysunek wygląda prawidłowo i czytelnie nawet pomimo używania wielu opcji na jednym rysunku (tym samym nakładania kolejnych warstw), natomiast z tego powodu w kodzie występują redundantne fragmenty.