Specyfikacja implementacyjna programu służącego do analizy grafów (Java).

Filip Sosnowski, Krzysztof Tadeusiak

06.06.2022

Informacje ogólne

Program służy do analizy grafu wygenerowanego na podstawie podanych przez użytkownika argumentów. Umożliwia znalezienie najkrótszej drogi pomiędzy wybraną parą węzłów, pozwala sprawdzić spójność grafu, zapisać go do pliku wyjściowego, jak i odczytać z pliku wejściowego. Wygenerowane dane są zapisywane do pliku tekstowego. W naszym projekcie wykorzystane jest działanie algorytmu Breadth-first Search (BFS) oraz algorytmu Dijkstry.

Parametry

- `x (liczba)` określa liczbę kolumn w grafie;
  Jest to parametr obowiązkowa.
- `y (liczba)` określa liczbę wierszy w grafie;
  Jest to flaga obowiązkowa.
- `n (liczba)` określa liczbę grafów otrzymanych z pierwotnie wygenerowanego grafu;
  domyślnie `n = 1`;
- `min (liczba)` określa minimalną wartość wagi krawędzi między węzłami;
  domyślnie `min = 0`;
- `max (liczba)` określa maksymalną wartość wagi krawędzi między węzłami;
  domyślnie `max = 1`;
- `ps (numer_węzła)` określa punkt startowy;
  domyślnie `ps = 0`;
- `pk (numer_węzła)` określa punkt końcowy;
  domyślnie `pk =` $x \times y$`-1` ;

- `out (nazwa_pliku)` określa nazwę pliku, do którego zostaną zapisane dane;
  domyślnie `out = mygraph`;

Projekt ten może działać także w drugim trybie. Program potrafi przeczytać graf z pliku o ustalonym formacie. Służy do tego flaga:

- `in (nazwa_pliku)` określa nazwę pliku, z którego zostaną odczytane dane;

Zakresy wartości parametrów:

- x > 0, x $\leq 10^8$, x$\times y \leq 10^8$ oraz $x \in C$ (x jest zmienną typu int);
- y > 0, y $\leq 10^8$, x$\times y \leq 10^8$ oraz $y \in C$ (y jest zmienną typu int);
- min $\geq 0, min \leq 1$ oraz min$\in R$ (min jest zmienną typu double);
- max $\geq 0, max \leq 1$ oraz max$\in R$ (max jest zmienną typu double);
- ps $\geq 0, ps \leq$ x$\times y - 1$ oraz $ps \in C$ (ps jest zmienną typu int);
- pk $\geq 0, pk \leq$ x$\times y - 1$ oraz $pk \in C$ (pk jest zmienną typu int);
- n $\geq 1, n \leq \frac{x \times y}{4}$ oraz $n \in C$ (n jest zmienną typu int);

Opis klas

Program będzie składać się z trzynastu Klas:

- **Main** - klasa główna, odpowiedzialna za wyświetlenie interfejsu programu, który oparty jest na bibliotece JavaFX.
- **IncidenceMatrix** - klasa odpowiedzialna za tworzenie trzech wektorów: wektor przechowujący numery kolumn elementów występujących w macierzy incydencji, wektor przechowujący numery wierszy elementów występujących w macierzy incydencji oraz wektor przechowującego wartość przejść pomiędzy wierzchołkami grafu. Wektory te pomieszczą informacje o grafie, którego wymiary wynoszą: $x \times y$, gdzie: x$\leq 10^8$, y $\leq 10^8$, $x \times y \leq 10^8$.
- **BreadthFirstSearch** - klasa odpowiedzialna za przeszukiwanie grafu metodą breadth-first search (przeszukiwanie wszerz). Służy do ustalenia spójności grafu.
- **Djikstra** - klasa odpowiedzialna za znalezienie najkrótszej drogi pomiędzy wybranymi przez użytkownika wierzchołkami.
- **MatrixValues** - klasa odpowiedzialna za wypełnienie trzech wektorów reprezentujących macierz incydencji wartościami pseudolosowymi od 0 do 1 lub zakresami podanymi przez użytkownika.
- **Cohesion** - klasa odpowiedzialna za n-krotne podzielenie grafu gdzie: n ≥ 1, $n \leq \frac{x \times y}{4}$ oraz $n \in C$.

- **ToFile** - klasa odpowiedzialna za zapisywanie parametrów wygenerowanego grafu do pliku.
- **FromFile** - klasa odpowiedzialna za odczytywanie pliku z zapisanym grafem.
- **Connectivity** - klasa odpowiedzialna za sprawdzenie spójności grafu, wykorzystuje BFS.
- **Controller** - klasa, w której znajdują się metody, które niezbędne są do obsługi interfejsu. Są to głównie metody dotyczące przycisków, pól tekstowych oraz komunikatów błędów.
- **Generate** - klasa łącząca, która pozwala przeprowadzić całkowitą generację grafu po wpisaniu w interfejsie użytkownika parametrów. Tworzy graf na ich podstawie oraz zapisuje go do pliku wyjściowego w folderze `dane`.
- **ShortestPath** - klasa odpowiedzialna za uzyskanie wartości najkrótszej ścieżki z algorytmu Dijkstry oraz wypisanie jej w inetrfejsie użytkownika.
- **Zoom** - klasa odpowiedzialna za wizualizację grafu.

Opis metod

Klasa Main:

- plik Main.java
- public void start(Stage stage) throws IOException

Klasa `Main` służy do uruchomienia programu. Tworzy scenę (okno), na której wyświetlony jest interfejs użytkownika dla naszego programu.

Klasa IncidenceMatrix

- plik IncidenceMatrix.java
- public int CountElements(int x, int y)
  Służy do policzenia, ile elementów maksymalnie może być przechowanych w wektorach reprezentujących połączenia w grafie. Zwraca liczbę elementów.
- public void FillArrays(int x, int y, double[] value, int[] row, int[] column)
  Metoda, która wypełnia wektory value, row, column odpowiednimi wartościami: value - wartościami przejść(jako połączenie wstawiana jest cyfra "1"), row - numer wiersza, column - numer kolumny.
  By zaimplementować obie powyższe metody, w zależności od położenia węzłów, należało zapisać odpowiednie warunki, by generacja wektorów reprezentujących macierz sąsiedztwa była poprawna.

Klasa BreadthFirstSearch:

- plik BreadthFirstSearch.java

- private int V - liczba wierzchołków
- private LinkedList<Integer> adj[] - lista liniowa przechowująca połączenia
- BreadthFirstSearch(int v) - konstruktor , tworzy listę liniową adj oraz przypisuje liczbę wierzchołków V równą v.
- void addEdge(int v, int w) - dodaje połączenie między wierzchołkami
- void addAllEdges(int[] row, int[] column) - dodaje wszystkie połączenia opisane w wektorach row oraz column
- public String BFS(int ps) - algorytm BFS, który zwraca zmienną typu String, która zawiera informacje o spójności grafu.

Struktura kolejki jest niezbędnym elementem wykorzystywanym w algorytmie BFS. Wykorzystać należy listę liniową LinkedList<Integer>. Umożliwia ona poruszanie się między węzłami i zapamiętywanie odwiedzone wczesniej węzły. W naszym programie wykorzystywać będziemy kolejkę typu FIFO. Początkowo kolejka składa się tylko z wierzchołka startowego. Dokładani są później do niej sąsiedzi pierwszego wierzchołka. Dopóki kolejka nie jest pusta, pobieramy pierwszej element (usuwając go z kolejki). Następnie sprawdzamy, który z jego sąsiadów nie ma ustawionego poprzednika. Dla każdego nieodwiedzonego dotychczas sąsiada należy ustawić odległość od wybranego elementu z kolejki. Następnie należy ustawić poprzednika na aktualnie wybrany element. Algorytm kończy się gdy kolejka będzie pusta. Jesli liczba odwiedzonych wierzchołków jest równa liczbie wszystkich wierzchołków, to metoda zwraca String "Graf jest spójny", w przeciwnym wypadku "Graf nie jest spójny".

Klasa Djikstra:

- plik Djikstra.java
- public static class AdjacencyList -
  Lista sąsiedztwa, zawiera informacje o wierzchołkach oraz wartościach przejścia między nimi
  - AdjacencyList(int v, double w)
  - int getVertex() - metoda zwraca wierzchołek z listy sąsiedztwa
  - double getWeight() - metoda zwraca wartość krawędzi przejścia między określonymi wierzchołkami z listy sąsiedztwa
- public static ArrayList<Number> dijkstra(int V, ArrayList<ArrayList <AdjacencyList> > graph, int ps, int pk)
  Metoda implementująca algorytm Dijkstry. Zwraca listę, która zawiera informacje o wartości ścieżki między punktami "ps" oraz "pk", a także znajdują się w niej pokonane wierzchołki. V jest liczbą wszystkich wierzchołków grafu.
- void addEdgeWeights(ArrayList<ArrayList<AdjacencyList> > graph, double[] value, int[] row, int[] column)
  Metoda ta dodaje wszystkie połączenia w grafie oraz ich wartości przejść.

Metoda faworyzuje wagi o mniejszej wartości(chętniej je wybieraja i poruszaja się nimi). Dla każdego wierzchołka przypisywane są dwie wartości: dystans oraz poprzednik. Początkowo każdy wierzchołek ma dystans równy Integer.MAX_VALUE, a poprzednik jest nieznany. Wyjątkiem jest wierzchołek początkowy dla którego dystans wynosi 0.

Dalszy etap polega na utworzeniu kolejki priorytetowej wszystkich wierzchołków, która porównuje zmienne typu double z listy AdjacencyList. Nastpnie dopóki kolejka nie jest pusta należy usunąć wierzchołek o najmniejszym dystansie i zbadać dla niego wszystkie połączone wierzchołki. Jeśli połączenie z wybranego wierzchołka do sąsiada ma łącznie mniejszą wartość niż docelowy wierzchołek to należy zaktualizować dystans oraz sąsiada oraz zapamiętać wierzchołek w tablicy "parent", która będzie przechowywać przebyte wierzchołki.

Klasa MatrixValues:

- plik MatrixValue.java
- private double RandomValue(double min, double max)
  Metoda zwraca losową wartość double z przedziału min oraz max.
- public void RandomizeMatrixValues(double min, double max, double[] value, int[] row, int[] column)
  Metoda ta wstawia losowe wartości w odpowiednie miejsca w tablicy value w zależności od położeń wartości z tablic row oraz column.

Moduł Split:

- plik Split.java
- public void cohesion(double[] value, int[] row, int[] column, int start)

Metoda `cohesion` ma za zadanie podzielić graf (reprezentowany przez trzy wektory) tyle razy, ile określił użytkownik poprzez podanie parametru `n`. Dokonane jest to poprzez wybór początkowego punktu podziału położonego na krańcu grafu, następnie poprzez losowe wybranie przejścia po wierzchołkach i utworzenie przecięcia. Przecięcie kończy się po dotarciu do wierzchołka na jednej z krawędzi grafu. Opisany powyżej podział reprezentowany jest poprzez zastąpienie wartości 1 z macierzy incydencji wartościami 0, które odwzorowują brak możliwości dostania się z jednego wierzchołka do drugiego. Podział nie może zakończyć sie na wierzchołku, z którego wychodzą 4 krawędzie. Oznaczałoby to wtedy niepoprawność działania programu i brak podziału grafu. Moduł zwraca zmodyfikowane tablice wejściowe.

Klasa ToFile:

- plik ToFile.java
- public void writeToFile(String fileName, int x, int y, double[] value, int[] row, int[] column)
  Metoda zapisuje do pliku fileName informacje o wygenerowanym grafie. W pierwszej linijce określona jest liczba kolumn oraz wierszy. Kolejne wiersze

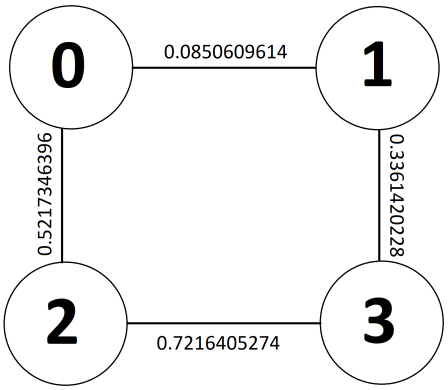
opisują wartości przejść (z tablicy value) pomiędzy konkretnymi węzłami w rozważanym grafie, zaczynając od pierwszego węzła i kończąc na węźle ostatnim.

Plik wyjściowy ma przyjmować następujący format:

```
2 2
        1: 0.0850609614 2: 0.5217346396
        0: 0.0850609614 3: 0.3361420228
        0: 0.5217346396 3: 0.7216405274
        1: 0.3361420228 2: 0.7216405274
```

W pierwszej linijce określona jest liczba kolumn oraz wierszy. Kolejne wiersze opisują wartości przejść pomiędzy konkretnymi węzłami w rozważanym grafie, zaczynając od pierwszego węzła i kończąc na węźle ostatnim.

Jest to odwzorowaniem poniższego grafu przykładowego:



Klasa FromFile:

- plik from_file.java
- private int numberOfLines(String filename)
  Metoda liczy liczbę linii w pliku - jeśli istnieje następna linia, to zwiększa licznik.
- public int numOfLinks(String filename)
  Metoda liczy liczbę połączeń w grafie - jeśli istnieje dwukropek, to zwiększa licznik.
- public int readX(String filename)
  Metoda odczytuje wartość "x" (kolumn grafu) z pliku "filename". Jest to pierwsze słowo występujące w poprawnie sformatowanym pliku.
- int readY(String filename)
  Metoda odczytuje wartość "y" (wierszy grafu) z pliku "filename". Jest to drugie słowo występujące w poprawnie sformatowanym pliku.

- private void readVals(String filename, double[] value, int[] row, int[] column)
  Metoda odczytuje wartości według ustalonego formatownia typu wierzchołek: wartość przejścia. Każda linia odpowiada połączeniom od określonego wierzchołka (zależnego od numeru linii w pliku).
- public boolean checkFileFormat(String filename)
  Metoda sprawdza, czy plik filename ma poprawny format, tzn. np. czy wymiary grafu odpowiadają liczbie linii pliku.
- public void reader(String filename, double[] value, int[] row, int[] column)
  Metoda łączy działanie metody checkFileFormat oraz metody readVals. Najpierw sprawdza czy format pliku jest poprawny, a nastepnie odczytuje wartości z pliku.

W pierwszej linijce określona jest liczba kolumn oraz wierszy. Kolejne wiersze opisują wartości przejść pomiędzy konkretnymi węzłami w rozważanym grafie, zaczynając od pierwszego węzła i kończąc na węźle ostatnim.

Klasa Controller:

- plik Controller.java
  W tej klasie należy zaimplementować wszystkie potrzebne metody, które potrzebne są do obsługi interfejsu użytkownika, tzn. działania po wciśnięciu przycisków czy wpisaniu wartości.

Klasa Generate:

- plik Generate.java
- public void generateGraph(int x, int y, double min, double max, int n, String output)
  Metoda ta pozwala skupić proces generacji w jednej metodzie. Łączy działanie innych metod: CountElements z Incidence Matrix, FillArrays z IncidenceMatrix, RandomizeMatrixValues z MatrixValues, writeToFile z ToFile. Tworzy wektory reprezentujące połączenia w grafie, losuje wartości przejścia z przedzialu min - max oraz zapisuje ten graf do pliku wyjściowego output.

Klasa ShortestPath:

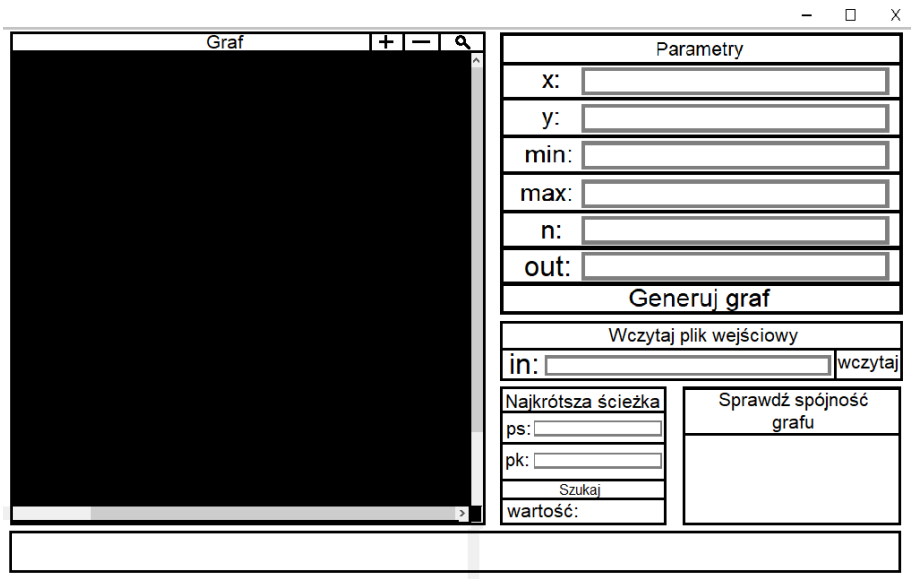
- plik ShortestPath.java
- public ArrayList<Number> getShortestPath(String filename, int ps, int pk)
  Metoda ta pozwala uzyskać ścieżkę z algorytmu Dijkstry oraz jej wartość pomiędzy dwoma punktami - ps oraz pk. Odczytuje graf z pliku, a następnie uruchamia algorytm. Zwracana jest lista, taka jak w metodzie Dijkstra.dijkstra.

Klasa Zoom:

- plik DragContext.java
  Klasa ta odpowiedzialna jest za zapamiętywania i zaczepianie kursora myszy na ekranie.
- plik NodeGestures.java
  Klasa odpowiedzialna za możliwość klikania na wierzchołki.
- plik PannableCanvas.java
  Klasa odpowiedzialna za usatlenie tła pod rysowanie grafu.
- plik SceneGestures.java
  W klasie tej znajdują się wydarzenia związane z przeciąganiem, klikaniem oraz scrolowaniem myszą. Dzieki klikaniu i przeciąganiu jesteśmy w stanie przesuwać canve po ekranie. Scrolowanie myszy jest odpowiedzialne za wbudowany w programie zoom dzięki, któremu możemy zrobić zbliżenie na bardzo małe i gęsto upakowan obiekty.
- plik Zoom.java
  public void start(Stage stage,int x ,int y, int ps, int pk,String filename) Metoda, która wyświetla drugą scenę z wygenerowanymi wezłami i krawedziami. Zaznacza również najkrótszą drogę między wierzchołkiem początkowym a końcowym zmieniając ich kolor na czerwony.

Opis graficznego interfejsu użytkownika

Po uruchomieniu programu pojawi się okno przedstawione poniżej.W klasie Controller zdefiniować należy pola tekstowe, przyciski oraz metody, które odpowiedzialne będą za łączenie elementów interfejsu z metodami innych klas, od których zależne jest działanie konkretnej funkcjonalności, np. znalezienia najkrótszej ścieżki.



Widoczne jest sześć paneli:

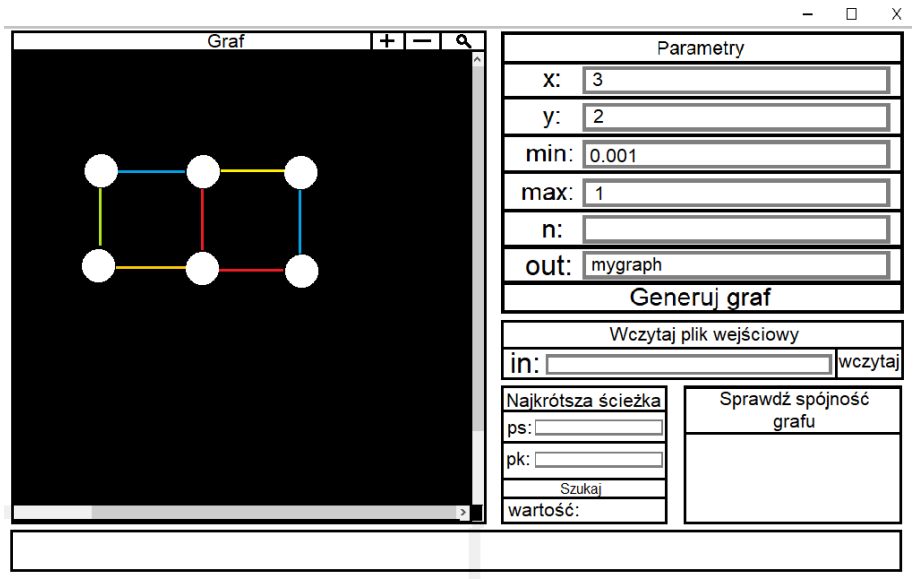
- Parametry - ten panel umożliwia podanie argumentów wejściowych takich jak liczba kolumn (x), liczba wierszy (y), minimalna (min) oraz maksymalna (max) wartość przejścia między węzłami, a także liczba podziału grafu (n) i nazwa pliku wyjściowego (out). Wybrane wartości użytkownik wprowadza w pole tekstowe. By wygenerować graf na podstawie podanych argumentów, należy kliknąć przycisk „Generuj". Przycisk ten połączony jest z działaniem klasy Generate.
- Graf - w tym panelu zostanie zwizualizowany graf, który został wygenerowany lub wczytany przez użytkownika. Umożliwi wybór węzłów do wyznaczenia najkrótszej ścieżki za pomocą myszki, a także pokaże ją na grafie. Graf zostanie przedstawiony w postaci siatki. Skrzyżowania kratek to węzły, a linie pionowe i poziome to krawędzie. Wagę krawędzi odwzorowane zostaną kolorem, gdzie kolor ciemnoniebieski odpowiada minimalnej wartości, a kolor ciemnoczerwony - maksymalnej wartości. Graf można przybliżać za pomocą przycisku "+" oraz oddalać za pomocą przycisku "-". Do przybliżenia można wykorzystać również przycisk z lupą. By przesuwać sie wzdłuż i wszerz grafu można wykorzystać paski przewijania. Rysowanie grafu powinno być zaimplementowane w klasie Zoom, a następnie podłączone do przycisku "Generuj" lub "Wczytaj" w klasie Controller
- Wczytaj plik wejściowy - panel ten odpowiada za wczytywanie pliku wejściowego o ustalonym formacie z grafem. Nazwę pliku należy wpisać w pole tekstowe obok słowa "in:", a następnie wcisnąć przycisk "wczytaj". Po jego wciśnięciu zostaje narysowany graf za pomocą metod zaimplementowanych

w klasie Zoom.

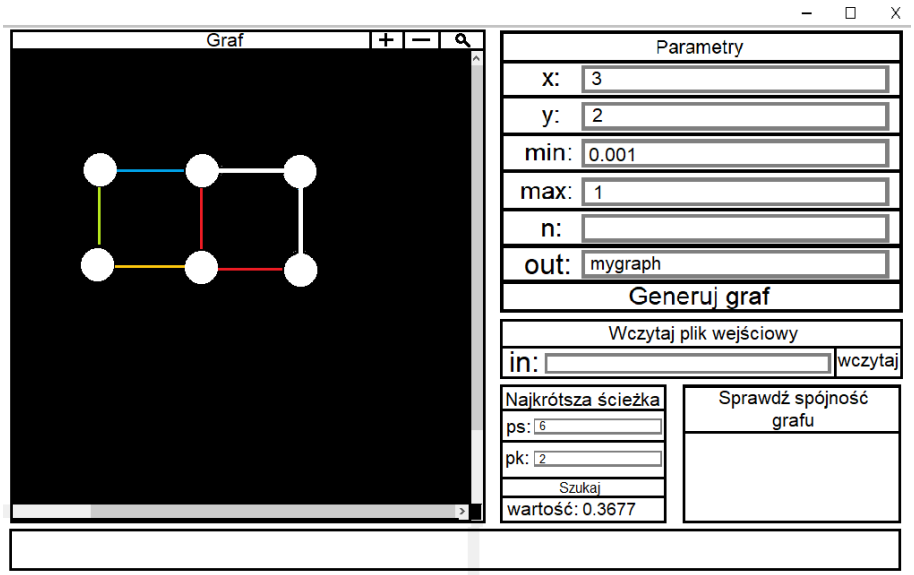
- Najkrótsza ścieżka - umożliwia wyszukanie najkrótszej ścieżki pomiędzy dwoma wybranymi punktami - punktem startowym (ps) oraz punktem końcowym (pk). Odpowiednie numery węzłów należy wpisać w pola tekstowe umieszczone przy "ps" oraz "pk". By zatwierdzić, trzeba wcisnąć przycisk "Szukaj". Jeśli wprowadzone dane są poprawne, wartość najkrótszej ścieżki pomiędzy tymi węzłami zostanie wypisana w polu "Wartość". Wartość ta zostanie uzyskana z metody getShortestPath z klasy ShortestPath.
- Sprawdź spójność grafu - na podstawie algorytmu BFS, panel ten pokaże czy rozważany graf jest spójny. By się tego dowiedzieć, należy wcisnąć przycisk "Sprawdź spójność grafu". Informacja o spójności zostanie wypisana w polu pod przyciskiem. Informacja ta zostanie uzyskana z metody checkConnectivity z klasy Connectivity.
- Komunikaty - panel umieszczony na samym dole okna programu jest odpowiedzialny za wypisywanie błędów oraz informacji.

Przykład działania interfejsu

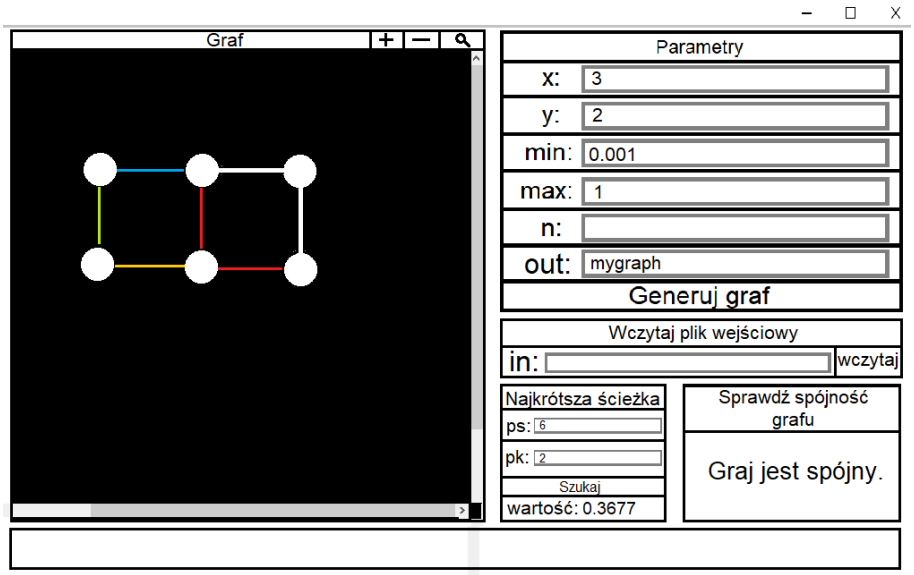
Przykład działania w trybie zapisu do pliku:



Po podaniu w panelu "Parametry" wybranych wartości oraz zatwierdzeniu ich przyciskiem "Generuj", w panelu "Graf" został wygenerowany graf o odpowiednich wymiarach oraz wagach krawędzi przejścia. Graf został zapisany do pliku wyjściowego "mygraph".

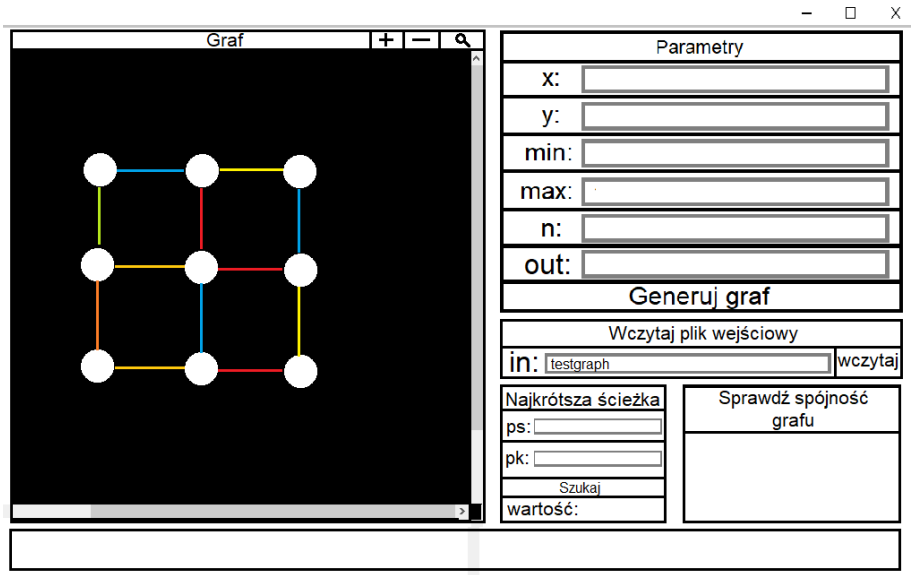


Po podaniu w panelu "Najkrótsza ścieżka" odpowiednich wartości punktu startowego oraz końcowego oraz zatwierdzeniu ich przyciskiem "Szukaj", została wypisana wartość długości tej ścieżki, a w panelu "Graf" kolorem białym zaznaczono krawędzie, które wchodzą w skład tej ścieżki.



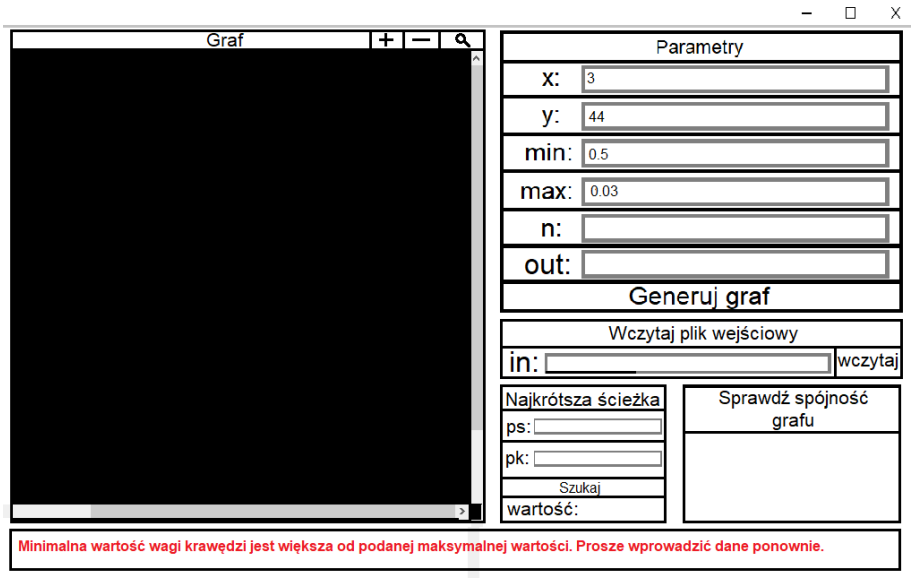
Po wciśnięciu przycisku "Sprawdź spójność grafu" zostaje poniżej tego przycisku wypisana informacja dotycząca spójności grafu, w tym przypadku "Graf jest spójny".

Przykład działania w trybie odczytu z pliku:



Po wpisaniu w panelu "Wczytaj plik wejściowy" nazwy pliku wejściowego z grafem o (odpowiednim formacie) oraz zatwierdzeniu go przyciskiem "Wczytaj", graf ten zostanie zwizualizowany w panelu "Graf". Sprawdzenie najkrótszej ścieżki oraz spójności grafu działa analogicznie do przypadków przedstawionych wyżej.

Przykład działania panelu z komunikatami błędów:



Testowanie

Program ten zostanie przetestowany pod kątem poprawności generowanych plików, odczytu zewnętrznych plików o określonym formacie oraz prawidłowości działania algorytmów analizy grafu. Do testów skorzystamy z JUnit.

Utworzony zostanie test pliku ShortestPath opierający się na wyznaczaniu najkrótszej ścieżki między węzłami w grafie. Wykorzystuje on do tego algorytm Dijkstry. W celu weryfikacji poprawnego działania testu, własnoręcznie obliczyliśmy wartość przejść w losowo wygenerowanym grafie.

Drugim testem jest test klasy Connectivity, a w tym algorytmu BFS. Test ma na celu wyznaczyć spójności grafu. Wykorzystuje on do tego fakt, iż ilość wszystkich wierzchołków w grafie wynosi $x \times y$, a każda inna wartość wynikowa zostanie uznana jako niepoprawna. Wynika to z tego, że graf jest spójny wtedy i tylko wtedy, gdy liczba odwiedzonych wierzchołków równa jest łącznej liczbie wierzchołków.

Kolejnym testem jest test formatu pliku, sprawdzić należy czy dobrze odczytuje poprawność formatu dla pliku zgodnego z ustaleniami oraz dla pliku, który nie spełnia określonych wymagań formatu. Sprawdzimy w tym teście również odczytywanie wartości z pliku.

Dla grafów o wymiarach $x \times y \leq 25$ dokonamy analizy manualnej, to znaczy osobiście przestudiujemy wygenerowany graf oraz określimy spodziewane rezultaty. Określimy najkrótszą drogę pomiędzy wybranymi punktami, a także określimy spójność tego grafu. Test zostanie zakończony sukcesem, jeżeli wyniki te będą równe rozwiązaniom uzyskanym za pomocą naszego programu. Pozwoli to ocenić poprawność zaimplementowanego działania algorytmu BFS oraz Dijkstry.