

JIMP2 Projekt 2025

Dokumentacja implementacyjna - Java

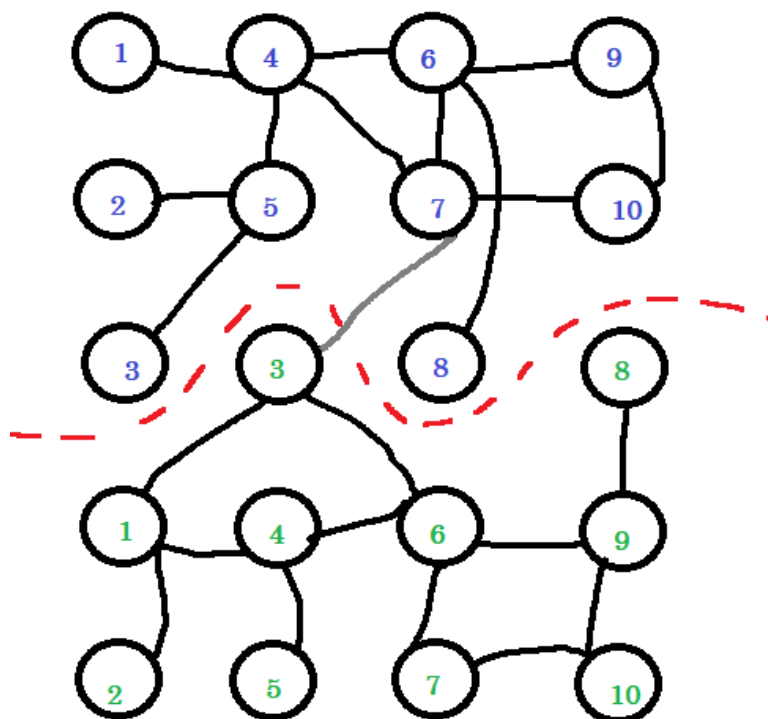
Michał Ludwiczak
GR3

12 maja 2025

Spis treści

1	Cel projektu	2
2	Problem	3
3	Algorytm	3
3.1	Macierz Laplace'a	3
3.2	Obliczenie par własnych	3
3.3	Podział	4
3.3.1	Podział według wektora Fiedlera	4
3.3.2	Klasteryzacja	4
3.4	Wynik	4
4	Interfejs graficzny użytkownika	4
5	Struktura programu	6
6	Pliki wejściowe i wyjściowe dla trybu dzielenia	11
6.1	Plik wejściowy (.csrrg / .bin)	11
6.2	Plik wyjściowy (.csrrg2 / .bin)	11
7	Pliki wejściowe dla trybu wyświetlenia podzielonego grafu	12
7.1	Plik wejściowy w formacie .csrrg	12
7.2	Plik wejściowy w formacie binarnym (.bin)	12
8	Przykłady użycia	12
8.1	Wczytywanie grafu z pliku tekstowego	12
8.2	Wczytywanie grafu z pliku binarnego	13
8.3	Podział grafu na x części z marginesem y	13
8.4	Zapisanie wyniku do pliku wyjściowego	13
8.5	Zapisanie wyniku do pliku binarnego	13

Celem projektu jest stworzenie aplikacji w języku Java, umożliwiającej podział grafu na określoną przez użytkownika liczbę części, z zachowaniem określonego lub domyślnego 10-procentowego marginesu różnicy liczby wierzchołków pomiędzy częściami. Domyślnie graf dzielony jest na dwie części. Celem podziału jest także minimalizacja liczby przeciętych krawędzi pomiędzy powstałymi częściami grafu. Aplikacja będzie wyposażona w graficzny interfejs użytkownika wykonany w technologii Swing. Użytkownik będzie mógł wczytywać już podzielony graf z pliku tekstowego lub binarnego, wczytywać niepodzielone grafy, definiować liczbę części oraz margines podziału, a także zapisywać wynikowy graf wraz z danymi wejściowymi do pliku tekstowego lub binarnego. Każda część grafu będzie prezentowana w interfejsie graficznym w odrębnym kolorze.



Rysunek 1: Przykładowy graf podzielony na 2 równe części

2 Problem

Podział grafu na części w taki sposób, aby liczba przeciętych krawędzi była jak najmniejsza, nie jest problemem łatwym. Znalezienie optymalnego rozwiązania dla dużych grafów jest praktycznie niewykonalne w rozsądnym czasie, ponieważ liczba możliwych podziałów rośnie wykładniczo wraz z liczbą wierzchołków. W przypadku podziału grafu na dwie części istnieje $2^{n-1} - 1$ możliwych sposobów podziału, gdzie n oznacza liczbę wierzchołków. Z tego powodu zamiast sprawdzania wszystkich możliwych podziałów stosuje się algorytmy przybliżone, heurystyki lub algorytmy zachłanne, które pozwalają na szybkie znalezienie dobrego, choć niekoniecznie optymalnego rozwiązania.

3 Algorytm

W programie zdecydowałem się użyć metody spektralnej dzielenia grafu, która wykorzystuje własności spektralne macierzy Laplace’a grafu. Na podstawie spektrum grafu, które opisuje jego strukturę graf możemy podzielić w efektywny i satysfakcjonujący sposób, minimalizując liczbę przeciętych krawędzi.

3.1 Macierz Laplace’a

W podejściu spektralnym do podziału grafu kluczową rolę odgrywa macierz Laplace’a [2]. Wzór na Laplacian klasyczny definiuje się jako:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

gdzie:

- L to **macierz Laplace’a** grafu,
- D to **macierz stopni** to macierz diagonalna, w której elementy na diagonalu d_{ii} odpowiadają stopniowi wierzchołka i , czyli liczbie krawędzi, które są z nim bezpośrednio połączone. W przypadku pętli, każda taka krawędź zwiększa stopień wierzchołka o 2, ponieważ jest traktowana jako dwie krawędzie incydentne z tym samym wierzchołkiem
- A to **macierz sąsiedztwa** grafu, gdzie elementy a_{ij} są równe 1, jeśli istnieje krawędź między wierzchołkami i i j , oraz 0 w przeciwnym przypadku.

3.2 Obliczenie par własnych

Następnym krokiem jest obliczenie $k = p - 1$ (gdzie p to liczba części, na które graf ma być podzielony) najmniejszych wektorów własnych, nie wliczając pierwszego zerowego. Dla bisekcji będzie to tylko jeden wektor, drugi najmniejszy - wektor Fiedlera. Wektory własne macierzy Laplace’a grafu przechowują informacje o połączeniach pomiędzy wierzchołkami. Do obliczenia tych par własnych można posłużyć się biblioteką Smile/ARPACK. Jest oparta na metodzie Implicitly Restarted Arnoldi Method (IRAM) i zaprojektowana specjalnie do obliczania kilku wartości i wektorów własnych bardzo dużych, rzadkich macierzy CSR. Jednak dla macierzy Laplace’a grafu będzie wykorzystywał Implicitly Restarted Lanczos Method (IRLM) - IRAM dla symetrycznych macierzy, który

stosuje metodę Lanczosa zamiast pełnej procedury Arnoldiego, budując tridiagonalną macierz T_k oraz ortonormalną bazę wektorów własnych. Smile/ARPACK wymaga jedynie iloczynu macierz–wektor, co oznacza, że pamięć zajmowana jest proporcjonalnie do liczby niezerowych elementów, dzięki czemu świetnie sprawdza się przy analizie dużych grafów.

3.3 Podział

3.3.1 Podział według wektora Fiedlera

Podział grafu według wektora Fiedlera polega na wykorzystaniu drugiego co do wielkości (drugiego najmniejszego) wektora własnego macierzy Laplasjanu grafu, nazywanego wektorem Fiedlera. Ten wektor koduje informację o strukturze spójności grafu – wierzchołki o podobnych wartościach współrzędnych wektora leżą blisko siebie w grafie.

3.3.2 Klasteryzacja

Natomiast po otrzymaniu macierzy zawierającej k wektorów własnych, każdy wierzchołek grafu jest reprezentowany jako punkt w k -wymiarowej przestrzeni. W celu podziału grafu na p części, można zastosować algorytm centroidów (k-means) [3]. Centroidy są inicjalizowane na podstawie średniej wartości współrzędnych dla głównych osi. Następnie iteracyjnie przypisuje się wierzchołki do najbliższych centroidów, z uwzględnieniem limitu maksymalnej liczby elementów w jednym klastrze (wyznaczonego na podstawie dopuszczalnego marginesu). Po każdej iteracji aktualizowane są położenia centroidów, aż do równowagi.

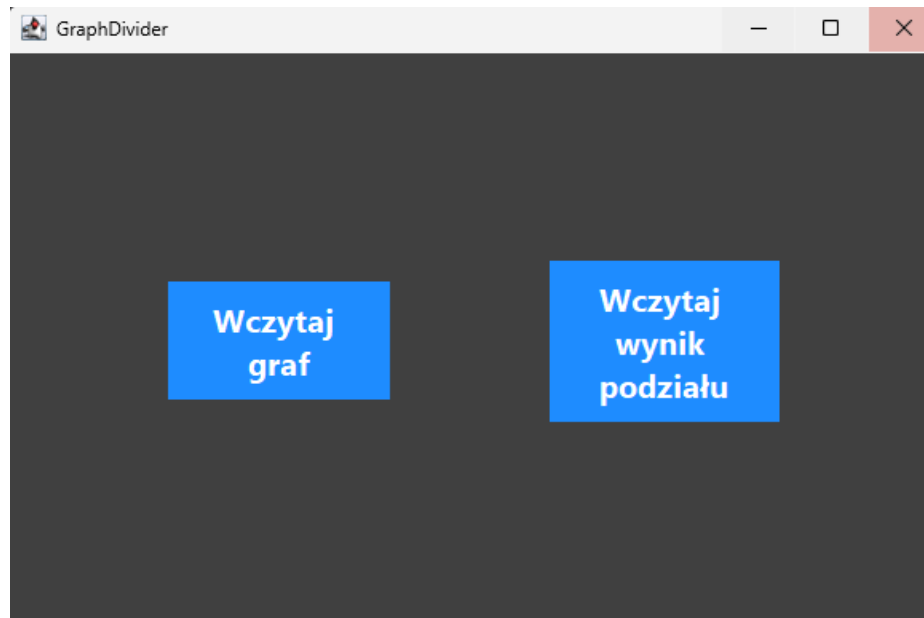
3.4 Wynik

Po otrzymaniu poszczególnych części z wierzchołkami modyfikuje się macierz sąsiedztwa A usuwając krawędzie pomiędzy wierzchołkami przynależącymi do różnych grup. W ten sposób otrzymuje się nową macierz sąsiedztwa, która zawiera już podzielony graf. Macierz sąsiedztwa jest już gotowa do przetworzenia i wypisania na plik wyjściowy.

4 Interfejs graficzny użytkownika

Aplikacja zostanie zaprojektowana z wykorzystaniem biblioteki Swing i oferuje graficzny interfejs użytkownika, który umożliwia interaktywną obsługę procesu podziału grafu. Interfejs składa się z następujących komponentów:

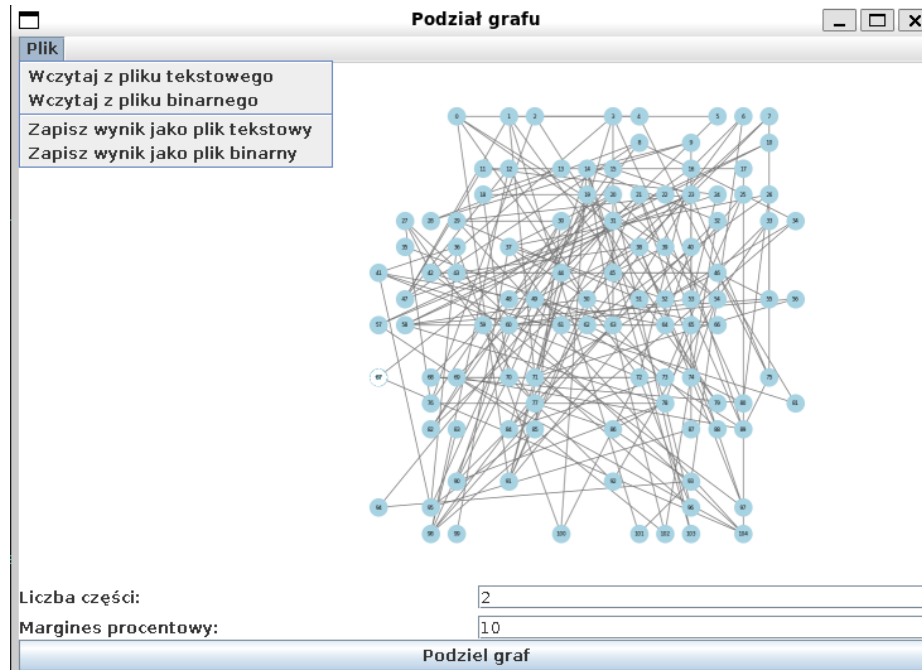
- **Ekran startowy** umożliwiający wybranie:
 - Wczytania grafu z pliku wejściowego, jego podzielenie po wybraniu liczby części i marginesu (w ekranie do definicji podziału), a następnie wyświetlenie
 - Wczytanie podzielonego grafu z pliku wejściowego (wyjściowy z C) i jego wyświetlenie



Rysunek 2: Przykładowy ekran startowy

- **Ekran do definicji podziału** z opcją wyboru liczby części i marginesu
- **Pasek menu lub przyciski** zawierające opcje:
 - Wczytaj z pliku tekstowego
 - Wczytaj z pliku binarnego
 - Zapisz wynik jako plik tekstowy
 - Zapisz wynik jako plik binarny
- **Komponent prezentujący wczytany lub podzielony graf**
Wierzchołki i krawędzie będą rysowane z wykorzystaniem metod graficznych Swing, a poszczególne części grafu po podziale są oznaczane różnymi kolorami.
- **Panel narzędziowy** zawierający:
 - Pole do wprowadzenia liczby części, na które ma zostać podzielony graf (domyślnie 2).
 - Pole do określenia dopuszczalnego marginesu procentowego różnicy w liczbie wierzchołków między częściami (domyślnie 10%).
 - Przycisk **Podziel graf**, który inicjuje proces podziału grafu zgodnie z podanymi parametrami.

Interfejs zostanie zaprojektowany z myślą o intuicyjnej obsłudze, umożliwiając użytkownikowi łatwe wczytywanie grafów, definiowanie parametrów podziału oraz wizualizację wyników.



Rysunek 3: Przykładowy interfejs przed podziałem

5 Struktura programu

W projekcie będę korzystał z Gradle jako głównego systemu budowania, co pozwoli na łatwe zarządzanie zależnościami i automatyzację procesu kompilacji. Wybrany IDE jest IntelliJ IDEA.

- docs/ — katalog z dokumentacją
- build.gradle — główny plik konfiguracyjny Gradle
- settings.gradle — plik ustawień projektu
- gradlew, gradlew.bat, gradle/wrapper/
- build/ — katalog generowany przez Gradle z artefaktami kompilacji
- src/main/java/graphdivider/ — katalog dla właściwego programu w Java
 - gui/ - cały interfejs graficzny z wykorzystaniem biblioteki Swing
 - * navigation/ - nawigacja odpowiedzialna za poszczególne ekrany
 - Navigator.java
 - AppNavigator.java

```

8  /**
9   * Implementation of Navigator using CardLayout to switch between different screens.
10  * <p>
11  * This class creates a single JFrame that contains a JPanel with CardLayout. Each registered s
12  */
13  public class AppNavigator implements Navigator 3 usages  ⚡ Michal Ludwiczak +1 *
14  {
15      /** Main application window. */
16      private final JFrame frame; 9 usages
17      /** Layout manager that treats each screen as a "card" and shows one at a time. */
18      private final CardLayout cardLayout; 3 usages
19      /** Container panel holding all registered screens under CardLayout. */
20      private final JPanel container; 4 usages
21
22      /**
23       * Constructs the AppNavigator:
24       */
25      public AppNavigator() 1 usage  ⚡ Michal Ludwiczak +1
26      { ... }
27
28      /**
29       * Registers a new screen with the navigator.
30       *
31       * @param screenId unique string identifier for the screen
32       * @param screen the JPanel representing the screen's UI
33       */
34      public void register(String screenId, JPanel screen) { container.add(screen, screenId); }
35
36      /** Displays the screen associated with the given ID. ... */
37      @Override 3 usages  ⚡ Michal Ludwiczak
38      public void show(String screenId)
39      { ... }
40
41      /** Provides access to the main JFrame for additional configuration, such as adding menus,
42       */
43      public JFrame getFrame() { return frame; }
44  }

```

Rysunek 4: AppNavigator.java

- * screen/ - poszczególne ekrany z zastosowaniem komponentów
 - StartScreen.java - ekran startowy

```

9  /**
10   * The initial screen shown when the application starts.
11   * <p>
12   * Uses GridBagLayout to place two buttons that expand
13   * proportionally when the window is resized.
14   */
15  public class StartScreen extends Screen 2 usages  ⤴ Michał Ludwiczak
16  {
17      /** Button to load a graph from an input file. */
18      private final Button button1; 3 usages
19      /** Button to load the result file produced by the C program. */
20      private final Button button2; 3 usages
21
22      /**
23       * Constructs the StartScreen and lays out its components.
24       *
25       * @param nav the Navigator used to switch between screens
26       */
27      public StartScreen(Navigator nav) 1 usage  ⤴ Michał Ludwiczak
28      {
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

```

Rysunek 5: StartScreen.java

- DivideScreen.java - ekran podziału grafu
- GraphScreen.java - ekran wyświetlenia grafu
- * widget/ - poszczególne komponenty Swing z zastosowaniem styli dla programu, w tym wizualizacja grafu
 - Screen.java


```

8  /**
9   * Base class for all application screens.
10  * <p>
11  * Provides common layout, styling, and full-screen frame creation.
12  * Subclasses should add their own components and event handlers.
13  */
14  @ public abstract class Screen extends JPanel 2 usages 1 inheritor 3 Michał Ludwiczak
15  {
16      /** Reference to the Navigator for switching between screens. */
17      protected final Navigator navigator; 3 usages
18
19      /**
20       * Constructs a Screen with shared configuration.
21       *
22       * @param navigator the Navigator used to request screen changes
23       */
24      public Screen(Navigator navigator) 1 usage 3 Michał Ludwiczak
25      { ... }
26
27      /**
28       * Initializes the panel with:
29       * 1. A centered GridBagLayout for flexible component placement.
30       * 2. A dark background for consistent theming.
31       * 3. An empty border to provide padding around edges.
32       */
33      private void initScreen() 1 usage 3 Michał Ludwiczak
34      { ... }
35
36      /**
37       * Wraps this panel in a full-screen JFrame.
38       * <p>
39       * Call frame.setVisible(true) to display.
40       *
41       * @return a JFrame ready to show this screen
42       */
43      public JFrame createFrame() no usages 3 Michał Ludwiczak
44      { ... }
45  }

```

Rysunek 6: Screen.java

· Button.java

```

11  /**
12   * A custom JButton with application-wide styling.
13   */
14  public class Button extends JButton 5 usages  ⚡ Michał Ludwiczak +1
15  {
16      /** Minimum and maximum font sizes for dynamic resizing */
17      private static final int MIN_FONT = 20, MAX_FONT = 28; 1 usage
18
19      /** Default colors for button background and text */
20      private static final Color PRIMARY_COLOR = new Color(r: 30, g: 140, b: 255); // 2 usages
21      private static final Color HOVER_COLOR = new Color(r: 24, g: 116, b: 205); // slight
22      private static final Color TEXT_COLOR = Color.WHITE; 1 usage
23
24      public Button(String text) 2 usages  ⚡ Michał Ludwiczak
25      {
26          ...
27      }
28
29      /** Applies the style. */
30      private void initStyle() 1 usage  ⚡ Michał Ludwiczak
31      {
32          ...
33      }
34
35      /** Configures background color and opacity settings. */
36      private void applyBackgroundAndOpacity() 1 usage  ⚡ Michał Ludwiczak
37      {
38          ...
39      }
40
41      /** Removes default borders and adds custom padding. */
42      private void applyBorderAndPadding() 1 usage  ⚡ Michał Ludwiczak
43      {
44          ...
45      }
46
47      /** Adds a mouse listener to handle hover color changes. */
48      private void attachHoverEffect() 1 usage  ⚡ Michał Ludwiczak
49      {
50          ...
51      }
52
53      /** Defines the preferred size based on the text length */
54      @Override  ⚡ Michał Ludwiczak
55      public Dimension getPreferredSize()
56      {
57          ...
58      }
59  }

```

Rysunek 7: Button.java

- Label.java
- Vertex.java
- Edges.java
- Graph.java
- algorithm/ - implementacja logiki algorytmu spektralnego
 - * CSRMmatrix.java - reprezentacja grafu w postaci macierzy Laplace'a w formie CSR, format CSR
 - * algorithm/ - implementacja właściwego algorytmu spektralnego
 - Eigenpairs.java - obliczanie par własnych macierzy Laplace'a grafu
 - Clusterization.java - klasteryzacja - podział grafu

- * `Output.java` - wypisanie do pliku wyjściowego
- `Main.java` - wywoływanie całego programu i poszczególnych modułów
- `src/main/resources/` — katalog z zasobami
 - `input/` - pliki wejściowe
 - `input_c/` - pliki wejściowe (wyjściowe z programu w C)
 - `icon.png` - ikonka programu

6 Pliki wejściowe i wyjściowe dla trybu dzielenia

6.1 Plik wejściowy (`.csrrg` / `.bin`)

Wejście programu może przyjąć dwie formy:

- Plik `.csrrg` z dodatkowymi sekcjami po piątej linii, opisującymi kolejne podgrafy powstałe z podziału grafu. Format pliku składa się z pięciu sekcji zapisanych w kolejnych liniach:
 1. Maksymalna liczba wierzchołków w dowolnym wierszu macierzy sąsiedztwa.
 2. Lista sąsiadów wszystkich wierzchołków zapisana sekwencyjnie.
 3. Wskaźniki (indeksy) na początki list sąsiedztwa dla poszczególnych wierzchołków.
 4. Lista grup wierzchołków połączonych krawędziami (reprezentacja krawędzi).
 5. Wskaźniki na początki grup węzłów z poprzedniej listy.
- Plik binarny `.bin` — binarna forma pliku `.csrrg`, mniej czytelna dla człowieka, ale szybsza w odczycie przez program.

6.2 Plik wyjściowy (`.csrrg2` / `.bin`)

Po przetworzeniu danych program generuje plik wyjściowy w jednym z dwóch formatów:

- `.csrrg2` — tekstowa forma pliku wyjściowego, wzorowana na formacie wejściowym `.csrrg`, ale zawierająca dodatkową linię nagłówkową z wynikiem działania programu.
- `.bin` — binarna wersja pliku tekstowego `.csrrg2`.

W pierwszej linii pliku wyjściowego zapisywany jest rezultat działania programu w formacie:

```
<wynik (S - sukces, F - porażka)> <liczba_części> <liczba_przecięć>
<zachowany_margines>
```

Przykład:

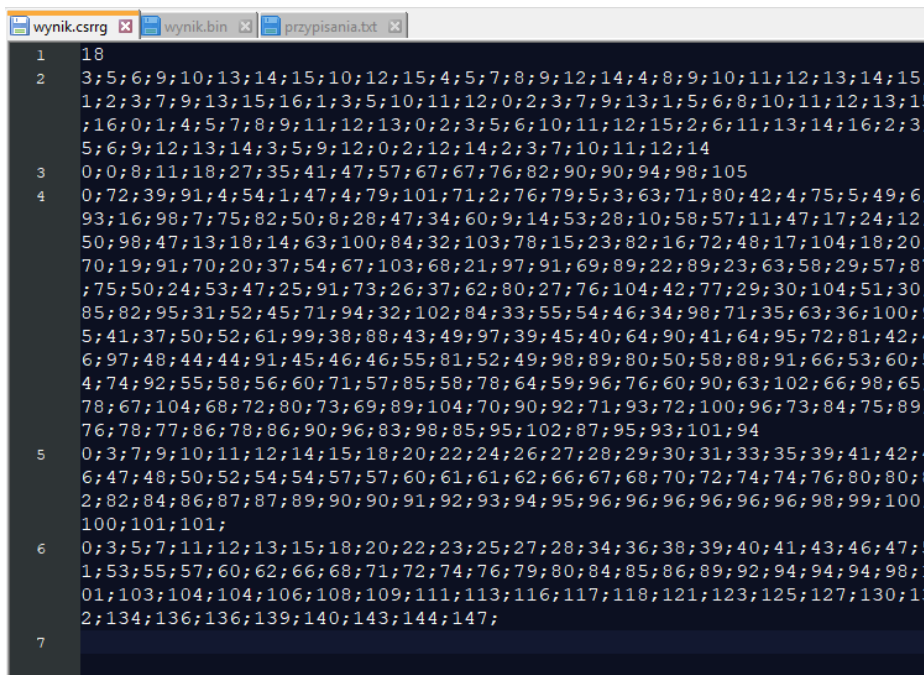
```
S 3 2 5
```

7 Pliki wejściowe dla trybu wyświetlenia podzielonego grafu

(pliki wyjściowe programu w C)

7.1 Plik wejściowy w formacie .csrrg

Plik przyjmuje postać taką jak plik wejściowy w formacie .csrrg z tym, że linijki po 5 linijce zawierają wierzchołki należące do danych kolejnych podgrup.



Rysunek 8: Przykładowy plik wejściowy dla trybu wyświetlenia

7.2 Plik wejściowy w formacie binarnym (.bin)

Zapis taki sam jak w formacie .csrrg.

8 Przykłady użycia

8.1 Wczytywanie grafu z pliku tekstowego

Aby wczytać graf z pliku tekstowego `graf.csrrg`, wybierz opcję `Wczytaj z pliku tekstowego` z paska menu. Program odczyta dane grafu, a następnie wyświetli je w głównym obszarze aplikacji.

8.2 Wczytywanie grafu z pliku binarnego

Aby wczytać graf z pliku binarnego `graf.bin`, wybierz opcję **Wczytaj z pliku binarnego** z paska menu. Program odczyta dane grafu w formacie binarnym i wyświetli je w głównym obszarze aplikacji.

8.3 Podział grafu na x części z marginesem y

Po załadowaniu grafu, w panelu narzędziowym ustaw liczbę części na x oraz margines na $y\%$. Następnie kliknij przycisk **Podziel graf**, aby rozpocząć proces podziału. Program spróbuje podzielić graf na x części, minimalizując liczbę przeciętych krawędzi, z zachowaniem określonego marginesu różnicy liczby wierzchołków.

8.4 Zapisanie wyniku do pliku wyjściowego

Po zakończeniu podziału, aby zapisać wynik do pliku tekstowego, wybierz opcję **Zapisz wynik jako plik tekstowy** z paska menu. Program zapisze dane grafu w formacie `.csrrg2`, zawierającym dodatkową linię nagłówkową z wynikiem działania programu.

8.5 Zapisanie wyniku do pliku binarnego

Po zakończeniu podziału, aby zapisać wynik do pliku binarnego, wybierz opcję **Zapisz wynik jako plik binarny** z paska menu. Program zapisze dane grafu w formacie `.bin`, co pozwoli na szybszy odczyt danych w przyszłości.

9 Wymagania niefunkcjonalne

Poniżej przedstawiono kluczowe wymagania niefunkcjonalne dla aplikacji:

- **Wydajność:** Czas podziału grafu powinien być jak najkrótszy dla danej wielkości grafu. Dla grafów zawierających do 1000 wierzchołków, czas przetwarzania nie powinien przekraczać minuty na standardowym sprzęcie klasy PC.
- **Użyteczność:** Interfejs użytkownika powinien być intuicyjny i zgodny z ogólnie przyjętymi standardami projektowania GUI.
- **Skalowalność:** Aplikacja powinna umożliwiać obsługę grafów o różnej wielkości, od małych do dużych.
- **Przenośność:** Aplikacja powinna działać poprawnie na różnych systemach operacyjnych wspierających środowisko Java.
- **Utrzymywalność:** Kod źródłowy aplikacji powinien być czytelny i dobrze udokumentowany, aby ułatwić przyszłe modyfikacje i rozwój.

Literatura

- [1] Leonid Zhukov, *Lecture 7. Graph partitioning algorithms.*, YouTube, 24 luty 2021, Dostępny na 11 maja 2025 w: https://youtu.be/zZae_C2BU_4
- [2] *Laplacian matrix*, Wikipedia, Dostępne na 11 maja 2025 w: https://en.wikipedia.org/wiki/Laplacian_matrix
- [3] *Algorytm centroidów*, Wikipedia, Dostępne na 11 maja 2025 w: https://pl.wikipedia.org/wiki/Algorytm_centroid%C3%B3w