JIMP2 cz.2 Java
Dokumentacja Końcowa

Mateusz Molęda oraz Zenon Nakamura

2 czerwca 2025

Spis treści

1 Wprowadzenie

Graph Partitioner to aplikacja Java Swing do wizualizacji i partycjonowania grafów przy użyciu algorytmu Kernighana-Lina oraz algorytmu hybrydowego. System umożliwia wczytywanie grafów z różnych formatów plików, interaktywną wizualizację, konfigurację parametrów podziału oraz zapis wyników.

1.1 Główne funkcjonalności

1. **Wczytywanie grafów** z 4 różnych formatów:
   - Tekstowy CSR z macierzą sąsiedztwa
   - CSRRG (Compressed Sparse Row Row Graph)
   - Proste przypisanie tekstowe
   - Proste przypisanie binarne (format bitowy)
2. **Algorytmy partycjonowania**:
   - Modulo
   - Sekwencyjny
   - Losowy
   - DFS (Depth-First Search)
   - Hybrydowy (automatycznie wybiera najlepszą strategię)
3. **Optymalizacja** przy użyciu algorytmu Kernighana-Lina
4. **Interaktywna wizualizacja**:
   - Przeciąganie wierzchołków
   - Zoom (kółko myszy)
   - Pan (prawy przycisk myszy)
   - Kolorowanie części
5. **Zapis wyników** w formatach:
   - Tekstowy (macierz + przypisania)
   - CSRRG

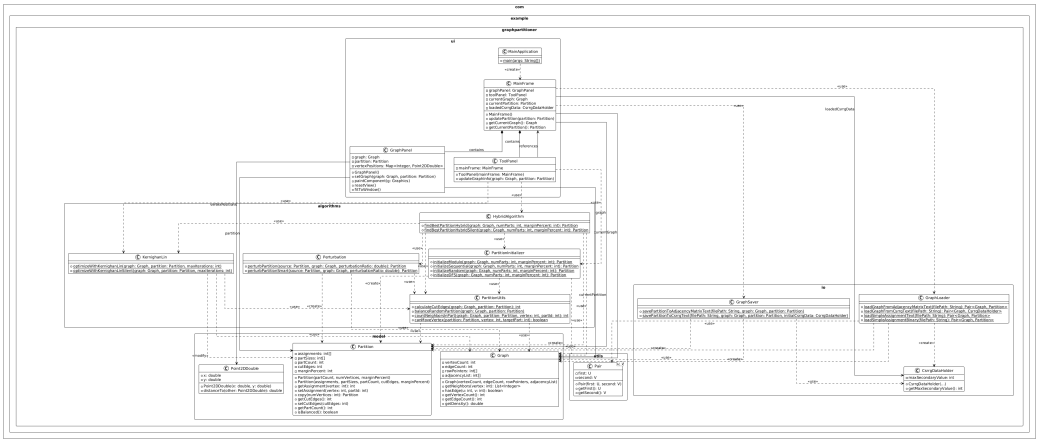
2 Architektura Systemu

2.1 Struktura pakietów

Projekt jest zorganizowany w następującej strukturze pakietów:

- `com.example.graphpartitioner` - pakiet główny
- `com.example.graphpartitioner.ui` - komponenty interfejsu użytkownika
- `com.example.graphpartitioner.model` - klasy modelu danych
- `com.example.graphpartitioner.io` - wczytywanie i zapis plików
- `com.example.graphpartitioner.algorithms` - algorytmy partycjonowania
- `com.example.graphpartitioner.utils` - klasy pomocnicze

2.2 Diagram klas UML



Rysunek 1: Diagram klas UML systemu Graph Partitioner

3 Reprezentacja Danych

3.1 Format CSR (Compressed Sparse Row)

Aplikacja wykorzystuje format CSR do efektywnej reprezentacji grafów rzadkich. Format ten składa się z dwóch tablic:

1. **rowPointers** - tablica wskaźników wierszy o rozmiarze $n + 1$ (gdzie n to liczba wierzchołków)
2. **adjacencyList** - spłaszczona lista sąsiedztwa zawierająca wszystkich sąsiadów

Dla wierzchołka v, jego sąsiedzi znajdują się w tablicy `adjacencyList` w zakresie indeksów:

$$\text{od } rowPointers[v] \text{ do } rowPointers[v+1] - 1$$

3.2 Klasa Graph

```java
package com.example.graphpartitioner.model;

import java.util.ArrayList;
import java.util.List;

public class Graph {
    private final int vertexCount;
    private final int edgeCount;
    private final int[] rowPointers;
    private final int[] adjacencyList;

    public Graph(int vertexCount, int edgeCount,
                 int[] rowPointers, int[] adjacencyList) {
        this.vertexCount = vertexCount;
        this.edgeCount = edgeCount;
        this.rowPointers = rowPointers;
        this.adjacencyList = adjacencyList;
    }

    public List<Integer> getNeighbors(int vertex) {
        if (vertex < 0 || vertex >= vertexCount) {
            return new ArrayList<>();
        }

        List<Integer> neighbors = new ArrayList<>();
        int start = rowPointers[vertex];
```

```
        int end = rowPointers[vertex + 1];

        for (int i = start; i < end; i++) {
            neighbors.add(adjacencyList[i]);
        }

        return neighbors;
    }

    public double getDensity() {
        if (vertexCount <= 1) {
            return 0.0;
        }
        double maxEdges = (double) vertexCount * (vertexCount
    - 1) / 2.0;
        return edgeCount / maxEdges;
    }
}
```

Listing 1: Implementacja klasy Graph w formacie CSR

3.3 Klasa Partition

Klasa `Partition` reprezentuje podział grafu na części:

```
package com.example.graphpartitioner.model;

import java.util.Arrays;

public class Partition {
    private final int[] assignments;     // mapuje ID
   wierzcho ka na ID cz   ci
    private final int[] partSizes;       // rozmiar ka dej
   cz   ci
    private final int partCount;         // liczba cz   ci
    private int cutEdges;                // liczba
   przeci tych kraw dzi
    private final int marginPercent;     // maksymalny
   dozwolony margines procentowy

    public Partition(int partCount, int numVertices, int
   marginPercent) {
        this.partCount = partCount;
        this.marginPercent = marginPercent;
        this.assignments = new int[numVertices];
        this.partSizes = new int[partCount];
        this.cutEdges = 0;

        Arrays.fill(assignments, -1);
```

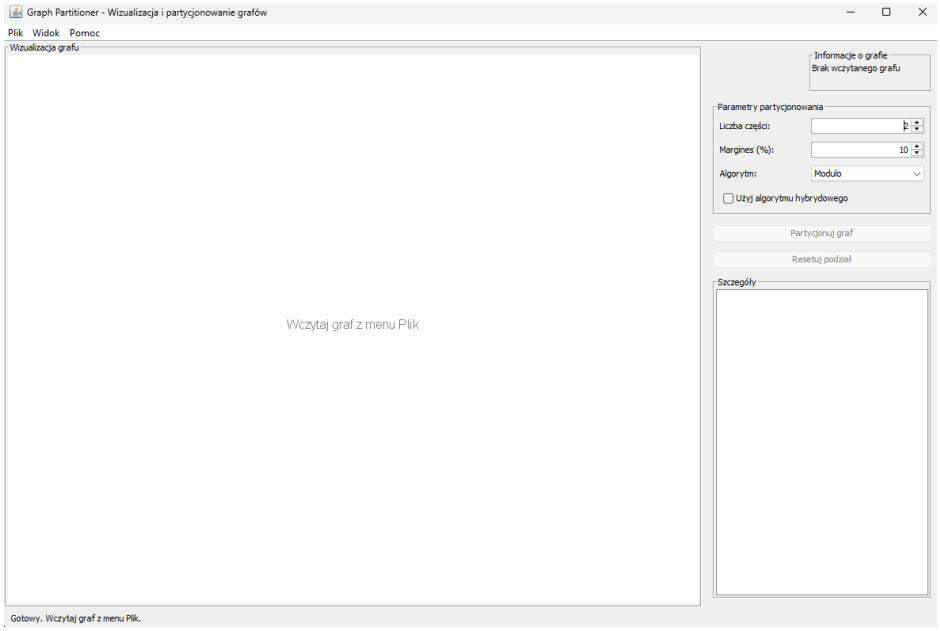
```
20      }
21
22      public int getAssignment(int vertex) {
23          if (vertex < 0 || vertex >= assignments.length) {
24              return -1;
25          }
26          return assignments[vertex];
27      }
28
29      public void setAssignment(int vertex, int partId) {
30          if (vertex < 0 || vertex >= assignments.length ||
31              partId < 0 || partId >= partCount) {
32              throw new IllegalArgumentException("Invalid
    vertex or part ID");
33          }
34
35          int oldPartId = assignments[vertex];
36
37          if (oldPartId >= 0 && oldPartId < partCount) {
38              partSizes[oldPartId]--;
39          }
40          partSizes[partId]++;
41
42          assignments[vertex] = partId;
43      }
44
45      public boolean isBalanced() {
46          int avgSize = getAveragePartSize();
47          int maxImbalance = getMaxImbalance();
48
49          for (int size : partSizes) {
50              if (Math.abs(size - avgSize) > maxImbalance) {
51                  return false;
52              }
53          }
54          return true;
55      }
56  }
```

Listing 2: Implementacja klasy Partition

4 Interfejs Użytkownika

4.1 Główne okno aplikacji (MainFrame)

Klasa `MainFrame` implementuje główne okno aplikacji z menu i panelami:



Rysunek 2: Układ głównego okna aplikacji

4.2 Panel narzędziowy (ToolPanel)

Panel narzędziowy zawiera kontrolki do konfiguracji parametrów podziału:

```
public class ToolPanel extends JPanel {
    private JSpinner numPartsSpinner;
    private JSpinner marginSpinner;
    private JComboBox<String> algorithmComboBox;
    private JCheckBox useHybridCheckBox;
    private JButton partitionButton;

    private void initializeComponents() {
        // Liczba cz   ci
        SpinnerNumberModel numPartsModel = new
   SpinnerNumberModel(2, 2, 100, 1);
        numPartsSpinner = new JSpinner(numPartsModel);

        // Margines procentowy
        SpinnerNumberModel marginModel = new
   SpinnerNumberModel(10, 0, 100, 5);
```

```
        marginSpinner = new JSpinner(marginModel);

        // ComboBox dla algorytmu
        String[] algorithms = {"Modulo", "Sekwencyjny", "
    Losowy", "DFS"};
        algorithmComboBox = new JComboBox<>(algorithms);

        // CheckBox dla algorytmu hybrydowego
        useHybridCheckBox = new JCheckBox("U yj algorytmu
    hybrydowego");
        useHybridCheckBox.addActionListener(e -> {
            algorithmComboBox.setEnabled(!useHybridCheckBox.
    isSelected());
        });

        // Przyciski
        partitionButton = new JButton("Partycjonuj graf");
        partitionButton.setEnabled(false);
    }
}
```

Listing 3: Fragment implementacji ToolPanel

4.3 Panel wizualizacji grafu (GraphPanel)

Panel wizualizacji implementuje interaktywną wizualizację grafu z możliwością manipulacji:

```
public class GraphPanel extends JPanel {
    private Graph graph;
    private Partition partition;
    private Map<Integer, Point2DDouble> vertexPositions;

    // Parametry widoku
    private double scale = 1.0;
    private double panX = 0;
    private double panY = 0;

    // Stan interakcji
    private Point dragStartPoint;
    private Integer draggedVertex;
    private boolean isPanning = false;

    // Kolory dla cz   ci
    private static final Color[] PART_COLORS = {
        new Color(255, 99, 71),   // Tomato
        new Color(30, 144, 255),  // DodgerBlue
        new Color(50, 205, 50),   // LimeGreen
```

```
        new Color(255, 215, 0),   // Gold
        new Color(148, 0, 211),   // DarkViolet
        new Color(255, 140, 0),   // DarkOrange
        new Color(0, 206, 209),   // DarkTurquoise
        new Color(255, 20, 147),  // DeepPink
        new Color(70, 130, 180),  // SteelBlue
        new Color(154, 205, 50)   // YellowGreen
    };

    private void calculateVertexPositions() {
        vertexPositions = new HashMap<>();

        if (graph == null || graph.getVertexCount() == 0) {
            return;
        }

        int n = graph.getVertexCount();
        double centerX = getWidth() / 2.0;
        double centerY = getHeight() / 2.0;
        double radius = Math.min(getWidth(), getHeight()) *
0.35;

        // Grupuj wierzcho ki wed ug cz   ci (je li
istnieje podzia )
        if (partition != null && partition.getPartCount() >
1) {
            // Uk adamy wierzcho ki w grupach wed ug
cz   ci
            Map<Integer, List<Integer>> verticesByPart = new
HashMap<>();
            for (int i = 0; i < partition.getPartCount(); i
++) {
                verticesByPart.put(i, new ArrayList<>());
            }

            for (int v = 0; v < n; v++) {
                int part = partition.getAssignment(v);
                if (part >= 0 && part < partition.
getPartCount()) {
                    verticesByPart.get(part).add(v);
                }
            }

            // Uk adamy ka d   cz      w sektorze ko a
            double anglePerPart = 2 * Math.PI / partition.
getPartCount();

            for (int part = 0; part < partition.getPartCount
(); part++) {
```

```
61                  List<Integer> vertices = verticesByPart.get(
    part);
62                  if (vertices.isEmpty()) continue;
63
64                  double startAngle = part * anglePerPart;
65                  double endAngle = (part + 1) * anglePerPart;
66                  double angleStep = (endAngle - startAngle) /
    vertices.size();
67
68                  for (int i = 0; i < vertices.size(); i++) {
69                      double angle = startAngle + i * angleStep
     + angleStep / 2;
70                      double x = centerX + radius * Math.cos(
    angle);
71                      double y = centerY + radius * Math.sin(
    angle);
72                      vertexPositions.put(vertices.get(i), new
    Point2DDouble(x, y));
73                  }
74              }
75          } else {
76              // Zwyk y  uk ad  ko owy
77              double angleStep = 2 * Math.PI / n;
78
79              for (int i = 0; i < n; i++) {
80                  double angle = i * angleStep;
81                  double x = centerX + radius * Math.cos(angle)
    ;
82                  double y = centerY + radius * Math.sin(angle)
    ;
83                  vertexPositions.put(i, new Point2DDouble(x, y
    ));
84              }
85          }
86      }
87  }
```

Listing 4: Implementacja wizualizacji grafu

5 Algorytmy Podziału Grafu

5.1 Strategie inicjalizacji

Aplikacja oferuje cztery strategie początkowego podziału:

1. **Modulo** - wierzchołek v trafia do części $v \bmod k$
2. **Sekwencyjna** - kolejne bloki wierzchołków do kolejnych części
3. **Losowa** - losowe przypisanie z balansowaniem
4. **DFS** - używa przeszukiwania w głąb do grupowania połączonych wierzchołków

5.2 Algorytm Kernighana-Lina

Algorytm KL optymalizuje początkowy podział poprzez iteracyjne zamiany wierzchołków:

5.3 Algorytm hybrydowy

Algorytm hybrydowy łączy różne strategie i perturbacje:

```
1  public static Partition findBestPartitionHybrid(Graph graph,
2                                                  int numParts,
3                                                  int
       marginPercent) {
4      Partition bestPartition = null;
5      int bestCutEdges = Integer.MAX_VALUE;
6
7      // 1. Wypr buj deterministyczne strategie
8      System.out.println("Krok 1: Wypr bowywanie
       deterministycznych strategii...");
9
10     // Testuj: modulo, sekwencyjn , DFS
11     Partition[] strategies = {
12         PartitionInitializer.initializeModulo(graph, numParts
       , marginPercent),
13         PartitionInitializer.initializeSequential(graph,
       numParts, marginPercent),
14         PartitionInitializer.initializeDFS(graph, numParts,
       marginPercent)
15     };
16
17     for (Partition partition : strategies) {
18         if (partition != null) {
```

Algorithm 1 Algorytm Kernighana-Lina

Input: Graf G, początkowy podział P
Output: Zoptymalizowany podział P'
repeat
  $moves \leftarrow []$
  $cumulativeGain \leftarrow [0]$
  $moved \leftarrow [false] \times |V|$
  for $step = 1$ to $|V|$ **do**
    $bestGain \leftarrow -1$
    $bestVertex \leftarrow null$
    $bestTargetPart \leftarrow null$
    for każdy wierzchołek v not in $moved$ **do**
      for każda część $p \neq P[v]$ **do**
        if można przenieść v do p zachowując balans **then**
          $gain \leftarrow$ oblicz zysk z przeniesienia
          if $gain > bestGain$ **then**
            $bestGain \leftarrow gain$
            $bestVertex \leftarrow v$
            $bestTargetPart \leftarrow p$
          end if
        end if
      end for
    end for
    if $bestVertex = null$ OR $bestGain \leq 0$ **then**
      break
    end if
    Dodaj ruch do $moves$
    Przenieś $bestVertex$ do $bestTargetPart$
    $moved[bestVertex] \leftarrow true$
    $cumulativeGain.append(cumulativeGain[-1] + bestGain)$
  end for
  Znajdź prefiks $moves$ z maksymalnym $cumulativeGain$
  Przywróć oryginalny podział i zastosuj tylko najlepszy prefiks
until brak poprawy

```
19              KernighanLin.optimizeWithKernighanLin(graph,
    partition, 0);
20              if (partition.getCutEdges() < bestCutEdges) {
21                  bestCutEdges = partition.getCutEdges();
22                  bestPartition = partition;
23              }
24          }
25      }
26
27      // 2. Wypr buj losowe inicjalizacje
28      System.out.println("Krok 2: Wypr bowywanie losowych
    inicjalizacji...");
29      int randomTrials = calculateAdaptiveRandomTrials(graph,
    numParts, bestCutEdges);
30
31      for (int seed = 0; seed < randomTrials; seed++) {
32          Partition randomPartition = PartitionInitializer.
    initializeRandom(
33                                      graph, numParts,
    marginPercent);
34          if (randomPartition != null) {
35              KernighanLin.optimizeWithKernighanLin(graph,
    randomPartition, 0);
36              if (randomPartition.getCutEdges() < bestCutEdges)
     {
37                  bestCutEdges = randomPartition.getCutEdges();
38                  bestPartition = randomPartition;
39              }
40          }
41      }
42
43      // 3. Spr buj perturbacji najlepszego rozwi zania
44      if (bestPartition != null) {
45          System.out.println("Krok 3: Testowanie perturbacji...
    ");
46
47          int numPerturbations = graph.getVertexCount() > 1000
    ? 3 : 2;
48
49          for (int i = 0; i < numPerturbations; i++) {
50              double perturbationRatio = (i == 0) ? 0.15 : 0.1;
51
52              Partition perturbed = (i % 2 == 0) ?
53                  Perturbation.perturbPartition(bestPartition,
    graph, perturbationRatio) :
54                  Perturbation.perturbPartitionSmart(
    bestPartition, graph, perturbationRatio);
55
56              if (perturbed != null) {
```

```
57                  KernighanLin.optimizeWithKernighanLin(graph,
    perturbed, 0);
58                  if (perturbed.getCutEdges() < bestCutEdges) {
59                      bestCutEdges = perturbed.getCutEdges();
60                      bestPartition = perturbed;
61                  }
62              }
63          }
64      }
65
66      return bestPartition;
67  }
```

Listing 5: Główna funkcja algorytmu hybrydowego

6 Formaty Plików

6.1 Format 1: Tekstowy CSR z macierzą sąsiedztwa

Plik tekstowy zawierający macierz sąsiedztwa i opcjonalne przypisania:

```
[0. 1. 1. 0.]
[1. 0. 1. 1.]
[1. 1. 0. 1.]
[0. 1. 1. 0.]

0 - 0
1 - 0
2 - 1
3 - 1
```

6.2 Format 2: CSRRG (Compressed Sparse Row Row Graph)

Format 5-liniowy dla grafu głównego:

```
1                           # max secondary value
0;1;2                       # secondary data
0;1;2;3                     # secondary row pointers
1;2;0;2;0;1                 # graph neighbors (lista sąsiedztwa)
0;2;4;6                     # graph row pointers
```

6.3 Format 3: Proste przypisanie tekstowe

Linie w formacie:

```
Wierzchołek 0 -> Podgraf 0
Wierzchołek 1 -> Podgraf 1
Wierzchołek 2 -> Podgraf 0
...
```

6.4 Format 4: Proste przypisanie binarne

Plik binarny zawiera sekwencję bitów reprezentujących przynależność wierzchołków:

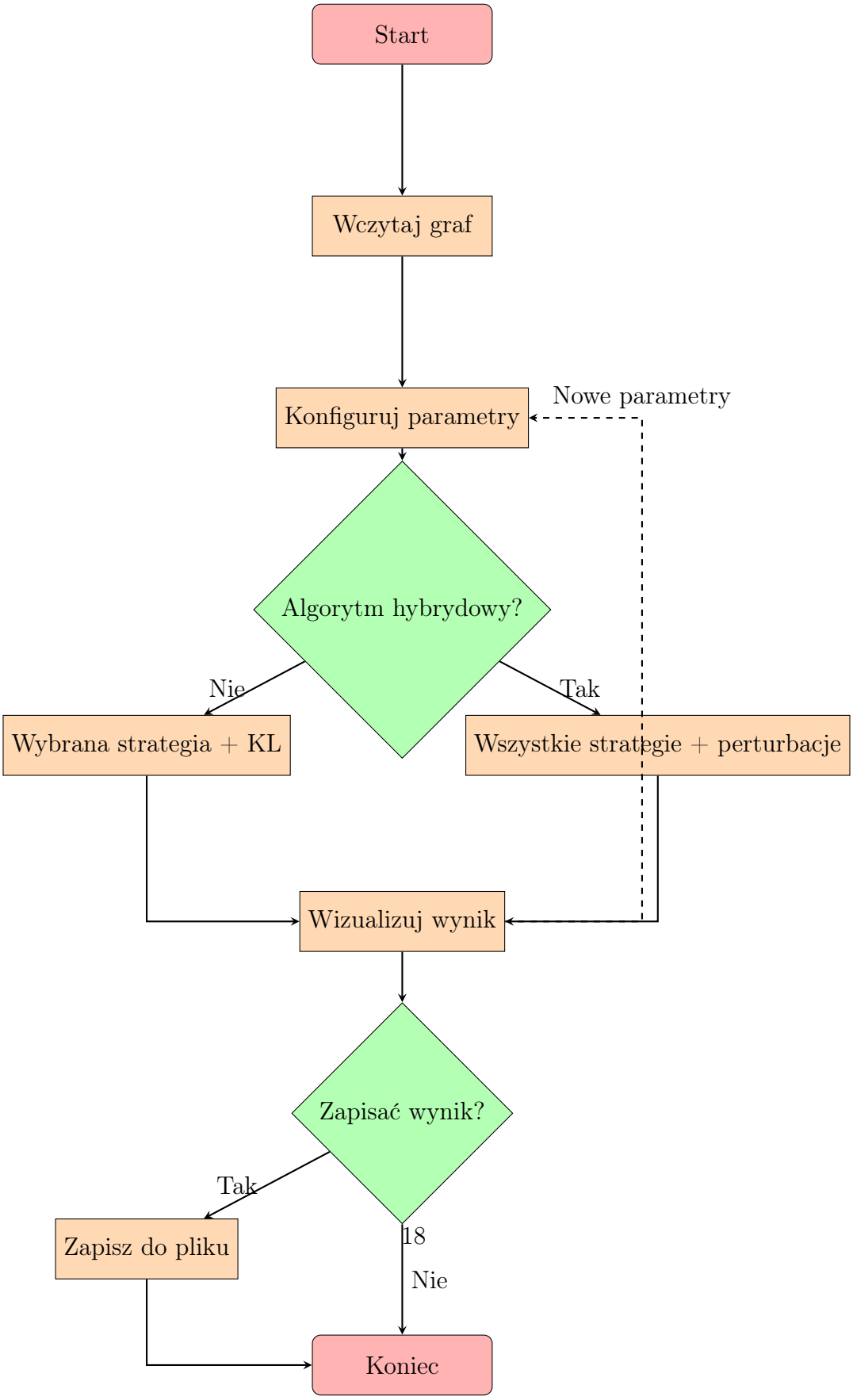
- Każdy bit odpowiada jednemu wierzchołkowi
- Bit = 0: wierzchołek należy do podgrafu 0
- Bit = 1: wierzchołek należy do podgrafu 1
- Bity są pakowane po 8 w bajty (LSB first)

7 Przepływ Pracy w Aplikacji

7.1 Wczytywanie grafu

1. Użytkownik wybiera opcję z menu "Plik":
   - Wczytaj tekstowy CSR/Macierz
   - Wczytaj CSRRG
   - Wczytaj proste przypisanie (tekst)
   - Wczytaj proste przypisanie (binarny)
2. Wyświetlane jest okno dialogowe wyboru pliku
3. Odpowiednia metoda z klasy `GraphLoader` parsuje plik
4. Graf jest wyświetlany w panelu wizualizacji

7.2 Partycjonowanie grafu



8 Wydajność i Optymalizacje

8.1 Optymalizacje algorytmiczne

1. **Format CSR** - minimalizuje zużycie pamięci dla grafów rzadkich
2. **Ograniczenie iteracji KL** - dla dużych grafów (>5000 wierzchołków) limit do 20 iteracji
3. **Adaptacyjna liczba prób** - w algorytmie hybrydowym zależna od:
   - Gęstości grafu
   - Liczby części
   - Dotychczasowej jakości rozwiązania
4. **Inteligentne perturbacje** - skupiające się na wierzchołkach granicznych

8.2 Optymalizacje interfejsu

1. **Lazy loading** - pozycje wierzchołków obliczane tylko przy zmianie grafu
2. **Efficient rendering** - rysowanie tylko widocznych elementów
3. **SwingWorker** - operacje czasochłonne w osobnym wątku

9 Obsługa Błędów

9.1 Walidacja danych wejściowych

System sprawdza poprawność:

- Formatów plików wejściowych
- Parametrów algorytmów (liczba części $\leq$ liczba wierzchołków)
- Indeksów wierzchołków
- Spójności struktur danych

9.2 Komunikaty błędów

Błędy są komunikowane użytkownikowi poprzez:

- Okienka dialogowe `JOptionPane`
- Pasek statusu
- Panel szczegółów w `ToolPanel`

10 Instrukcja Użytkowania

10.1 Wymagania systemowe

- Java 8 lub nowsza
- System operacyjny: Windows, Linux lub macOS
- Minimum 512 MB RAM (zalecane 2 GB dla dużych grafów)

10.2 Kompilacja i uruchomienie

10.2.1 Linux/macOS

```
chmod +x compile.sh run.sh
./compile.sh
./run.sh
```

10.2.2 Windows

```
compile.bat
run.bat
```

10.2.3 VS Code

Po otwarciu folderu w VS Code:

1. Zainstaluj rozszerzenie "Extension Pack for Java"
2. Użyj `Ctrl+Shift+B` do kompilacji
3. Użyj `F5` do uruchomienia

10.3 Typowy scenariusz użycia

1. **Wczytanie grafu**
   - Menu Plik → Wczytaj tekstowy CSR/Macierz...
   - Wybór pliku z grafem
2. **Konfiguracja parametrów**
   - Liczba części: 2-100
   - Margines procentowy: 0-100%
   - Algorytm: Modulo/Sekwencyjny/Losowy/DFS
   - Opcja: Użyj algorytmu hybrydowego
3. **Wykonanie podziału**
   - Kliknięcie "Partycjonuj graf"
   - Obserwacja postępu w panelu szczegółów
4. **Interakcja z wizualizacją**
   - Przeciąganie wierzchołków: lewy przycisk myszy
   - Zoom: kółko myszy
   - Pan: prawy przycisk myszy
   - Menu Widok → Resetuj widok / Dopasuj do okna
5. **Zapisanie wyniku**
   - Menu Plik → Zapisz podział jako tekst/CSRRG
   - Wybór lokalizacji i nazwy pliku

11 Rozszerzalność Systemu

11.1 Dodawanie nowych algorytmów

System umożliwia łatwe dodawanie nowych algorytmów:

1. Utworzenie nowej klasy w pakiecie `algorithms`
2. Implementacja metody partycjonowania
3. Dodanie opcji w `ToolPanel`
4. Aktualizacja logiki w metodzie `performPartitioning`

11.2 Wsparcie dla nowych formatów plików

Dodanie nowego formatu wymaga:

1. Implementacji metody wczytującej w `GraphLoader`
2. Opcjonalnie: metody zapisującej w `GraphSaver`
3. Dodania opcji menu w `MainFrame`

12 Podsumowanie

Graph Partitioner to kompletne narzędzie do wizualizacji i partycjonowania grafów, oferujące:

- **Elastyczność** - wsparcie dla różnych formatów plików i algorytmów
- **Interaktywność** - pełna kontrola nad wizualizacją
- **Wydajność** - optymalizacje dla dużych grafów
- **Rozszerzalność** - modułowa architektura umożliwiająca rozwój
- **Użyteczność** - intuicyjny interfejs graficzny

System może być wykorzystywany w:

- Badaniach naukowych nad algorytmami grafowymi
- Analizie sieci społecznościowych
- Optymalizacji układów VLSI
- Równoważeniu obciążenia w systemach rozproszonych
- Edukacji w zakresie teorii grafów i algorytmów

Aplikacja stanowi przykład dobrze zaprojektowanego oprogramowania w języku Java, wykorzystującego wzorce projektowe MVC (Model-View-Controller) oraz najlepsze praktyki programowania obiektowego.