1. Cel i zakres programu

`przetnij-graf` to aplikacja konsolowa w języku Java służąca do **podziału nieskierowanego, prostego grafu** na zadaną liczbę podgrafów, z zachowaniem równowagi wielkości w granicach określonego marginesu procentowego. Umożliwia to:

- wyodrębnienie spójnych fragmentów grafu,
- analizę każdego podgrafu niezależnie,
- eksport wyników w formacie tekstowym (CSR-like) lub binarnym,
- opcjonalny podgląd w terminalu.

2. Interfejs użytkownika

2.1. Wywołanie

```
java -jar przetnij-graf-1.0-SNAPSHOT-jar-with-dependencies.jar <input file> <N> <M> [-o <out
```

- `<input file>` - plik z grafem w prostym formacie tekstowym:
  ```
  V E
  src  dest
  …
  src_E dest_E
  ```
- `<N>` - liczba podgrafów (N 1),
- `<M>` - margines nierównowagi, interpretowany jako **maksymalny udział procentowy** wierzchołków jednego podgrafu (np. 20.0 oznacza, że żadna część nie może mieć $>$ 20% wszystkich wierzchołków),
- `-o <output file>` - ścieżka pliku wyjściowego (domyślnie `plik.out`),
- `-t` - dodatkowo drukuje reprezentację CSR-like w terminalu,
- `-b` - zamiast formatu tekstowego generuje plik binarny.

2.2. Format wyjściowy

2.2.1. Tekstowy (CSR-like)

1. `rowPos[0..V]` - tablica długości `V+1`, gdzie `rowPos[i]` wskazuje początek listy sąsiadów wierzchołka `i` w tablicy `rowNodes`,
2. `rowNodes[0..2E-1]` - spłaszczona lista wszystkich sąsiadów (graf nieskierowany: każda krawędź zapisana dwukrotnie),
3. `component[0..V-1]` - numer podgrafu (`0…splitCount-1`) dla każdego wierzchołka,
4. `compCounts[0..splitCount-1]` - liczba wierzchołków w każdej części,
5. `-1` - separator kończący dump.

2.2.2. Binarne

- Little-endianowy zapis kolejno:

1. `int` - `numVertices`
2. `int` - `numEdges`
3. Dla każdej z `numEdges` krawędzi:
   - `int src`,
   - `int dest`.

2.3 Interfejs graficzny

Dla użytkowników preferujących pracę w środowisku graficznym, program oferuje prosty interfejs GUI umożliwiający wizualizację grafu i wyników podziału.

Wygląd i funkcje:

1. **Główny panel** - wyświetla wizualizację grafu z kolorowym oznaczaniem podgrafów
2. **Panel kontrolny** - umożliwia zmianę parametrów podziału (liczba części, margines)
3. **Menu File** - opcje:
   - `Open` - wczytaj graf z pliku tekstowego
   - `Save As` - zapisz wynik podziału w formacie tekstowym lub binarnym
   - `Exit` - zamknij aplikację
4. **Menu View** - opcje:
   - `Refresh` - odświeża wizualizację grafu po zmianie parametrów
   - `Show Node Labels` - pokazuje/ukrywa etykiety wierzchołków
   - `Show Edge Weights` - pokazuje/ukrywa wagi krawędzi
5. **Status Bar** - wyświetla informacje o:
   - Liczbie wierzchołków i krawędzi
   - Aktualnie wybranych parametrach podziału
   - Statusie operacji

Obsługa:

1. Po uruchomieniu GUI (np. przez podwójne kliknięcie pliku JAR), wybierz `File > Open` aby wczytać graf
2. Ustaw parametry podziału w panelu kontrolnym
3. Kliknij `Split` aby wykonać podział
4. Wynik będzie widoczny jako kolorowe grupy wierzchołków
5. Użyj `View > Refresh` po zmianie parametrów aby zaktualizować wizualizację
6. Zapisz wynik wybierając `File > Save As`

3. Struktury danych i klasy publiczne

3.1. Klasy

```
public class Edge {
    private final int src;
    private final int dest;
    // konstruktor, gettery, equals, hashCode
}

public class Graph {
    private final int numVertices;
    private final int numEdges;
    private final List<Edge> edges;
    private int splitCount;
    // konstruktor, gettery, settery
}
```

3.2. Kluczowe metody publiczne

Klasa	Metoda	Opis
GraphIO	static Graph fromTextFile(String path)	Wczytuje gr
GraphIO	static void toTextFile(Graph g, String path)	Zapis CSR-l
GraphIO	static void toBinaryFile(Graph g, String path)	Zapis binarn
Graph	String toCSRString()	Generuje CS
GraphSplitter	static Graph splitGraph(Graph orig, int number, float marginPct)	Dzieli graf n

4. Przepływ głównego algorytmu (`splitGraph`)

1. **Przygotowanie**
   - `number++` → pracujemy na `N+1` grupach,
   - `Graph balancedGraph = new Graph(orig)`,
   - `totalV = orig.getNumVertices()`.
2. **Obliczenie `desiredSizes`**

   ```
   int base = totalV / number;
   int rem = totalV % number;
   int[] desiredSizes = new int[number];
   for(int c = 0; c < number; c++) {
       desiredSizes[c] = base + (c < rem ? 1 : 0);
   }
   ```
3. **Wielośródłowy BFS**

- Wylicz stopnie wierzchołków, wybierz `number` wierzchołków-ziaren z najwyższym stopniem,
- Kolejka elementów `(comp, u)`; oznacz `component[u] = comp`, `visited[u] = true`,
- Rozszerzaj BFS aż każda grupa osiągnie `desiredSizes[comp]`.

4. **Dopełnienie**
   - Dla każdej krawędzi `(u,v)`: jeżeli `u` ma `component`, `v` nie - przypisz `v` do tej samej grupy (i odwrotnie).
5. **Filtrowanie krawędzi**
   - Przepisz tylko te krawędzie, których oba końce są w tym samym `component`.
6. **Weryfikacja marginesu**

```
boolean invalid = false;
for(int i = 0; i < number; i++) {
    float perc = compCounts[i] * 100.0f / totalV;
    if(perc > marginPct) {
        invalid = true;
        break;
    }
}
if(invalid) return null;
```

7. **Zakończenie**
   - `balancedGraph.setSplitCount(number-1);`
   - Zwróć `balancedGraph`.

5. Przykłady użycia

5.1. Podział na 3 części, margines 20%, zapis tekstowy

Wejście (`graph.txt`):

```
8 8
0 1
0 2
1 3
2 3
3 4
4 5
5 6
6 7
```

```
java -jar przetnij-graf-1.0-SNAPSHOT-jar-with-dependencies.jar graph.txt 3 20.0 -o out.txt
```

Wygeneruje `out.txt` z CSR-like dump i komunikat:

```
Graph written to text file: out.txt
```

5.2. Podział na 4 części, margines 15%, zapis binarny

```
java -jar przetnij-graf-1.0-SNAPSHOT-jar-with-dependencies.jar graph.txt 4 15.0 -o out.bin -
```

Wygeneruje `out.bin` (little-endian) i komunikat:

```
Graph written to binary file: out.bin
```

6. Obsługa błędów i kody wyjścia

Kod wyjścia	Komunikat na `stderr`	Przyczyna
1	`Usage: java -jar ... <input> <N> <M> [-o <out>] [-t] [-b]`	Zbyt mało argumentów
2	`Failed to load graph from file: <input file>`	Błąd otwarcia / parsow
3	`Failed to split graph into <N> parts with margin <M>%`	Algorytm nie spełnił wa
(inny 0)	–	Błąd zapisu pliku wyjśc

7. Środowisko i zależności

- **Java**: JDK 17 lub nowszy
- **System budowania**: Maven
- **Uruchamianie**: Gotowy plik JAR z dołączonymi zależnościami:
  `java -jar target/przetnij-graf-1.0-SNAPSHOT-jar-with-dependencies.jar ...`

pom.xml (fragment):

```
<dependencies>
    <!-- Brak zewnętrznych zależności -->
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>przetnij.graf.Main</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
```

```
            </configuration>
        </plugin>
    </plugins>
</build>
```

8. Wytyczne stylu kodu

- **Formatowanie**: zgodne z konwencjami Java, 4 spacje wcięcia
- **Nazewnictwo**:
  - Klasy: `PascalCase` (`Graph`, `Edge`)
  - Metody: `camelCase` (`splitGraph`, `fromTextFile`)
  - Zmienne: `camelCase` (`desiredSizes`, `rowPos`)
- **Komentarze**: Javadoc dla publicznych klas i metod
- **Zarządzanie pamięcią**: wykorzystanie garbage collectora, zamknięcie strumieni w blokach try-with-resources

9. Licencja i autorzy

- **Licencja**: MIT
- **Autorzy**:
  - Karol Juszczak (karol.juszczak.stud@pw.edu.pl)
  - Yaroslav Shevchuk (yaroslav.shevchuk.stud@pw.edu.pl)