1. Struktura projektu

```
przetnij-graf-main/
  program/
     MAKEFILE
     main.c
     graph.h
     graph.c
     split.h
     split.c
     graphs/               # przykładowe pliki grafów i wyniki
  documentation/
     binary_graph_documentation.md  # dokumentacja formatu binarnego
  visualize_graph.py       # skrypt do wizualizacji podziału grafu
  README.md                # ogólny opis projektu
  .gitignore
```

2. Moduł `graph` (`graph.h` / `graph.c`)

2.1 Definicje typów

```c
typedef struct Edge {
    int src, dest;
} Edge;

typedef struct Graph {
    int    numVertices;  // liczba wierzchołków
    int    numEdges;     // liczba krawędzi
    Edge  *edges;        // tablica krawędzi (src, dest)
    int    splitCount;   // liczba żądanych podgrafów (ustawiana w split.c)
} Graph;
```

2.2 Funkcje tworzenia i zwalniania

- `Graph *createGraph(int V, int E)` Alokuje i inicjalizuje `Graph`, ustawia `splitCount = -1`.
- `void freeGraph(Graph *g)` Zwalnia tablicę `edges` i samą strukturę `Graph`.
- `Graph *copyGraph(const Graph *orig)` Tworzy głęboką kopię wszystkich pól i tablicy `edges`; kopiuje też `splitCount`.

2.3 Parsowanie i serializacja

2.3.1 Tekstowy format prosty (używany przy wczytywaniu)

- **Wejście** (`graphFromTextFile`):

  ```
  V E
  src1 dest1
  ...
  srcE destE
  ```

2.3.2 Wyjście tekstowe CSR-like (`graphToTextFile` / `graphToString`)

1. **Tablica `rowPos`** długości `V+1`, gdzie `rowPos[i]` = indeks w spłaszczonej liście sąsiadów, od którego zaczynają się krawędzie wierzchołka `i`.
2. **Spłaszczona lista sąsiadów** długości `2·E` (bo graf nieskierowany każda krawędź dwa razy).
3. **Tablica komponentów** długości `V`, zawierająca numer podgrafu (`0…splitCount-1`) dla każdego wierzchołka.
4. **Tablica rozmiarów komponentów** długości `splitCount`, podająca liczbę wierzchołków w każdej grupie.
5. **Separujący `-1`** kończący dump.

Przykład (dla małego grafu):

```
0 2 5 7   # rowPos[0..3] dla V=3, E=3
1 2 0 2 0 1  # sąsiedzi
0 0 1       # component[0..2]
2 1         # rozmiary 2 komponentów
-1
```

2.3.3 Format binarny (`graphToBinaryFile`)

Zapis little-endian:

1. 4 bajty (`int32`): `numVertices`
2. 4 bajty (`int32`): `numEdges`
3. Dla każdej z `numEdges` krawędzi:
   - 4 bajty: `src`
   - 4 bajty: `dest`

   Uwaga: W pliku binarnym **nie** zapisujemy `splitCount`.

3. Moduł `split` (`split.h` / `split.c`)

```
Graph *splitGraph(Graph *originalGraph, int number, float margin);
```

1. **Parametry**

- `originalGraph` – wskaźnik na wczytany graf,
- `number` – liczba podgrafów użytkownika (`N`),
- `margin` – maksymalny procentowy udział wierzchołków w jednym podgrafie (`M`).

2. **Inkrementacja grup**

```
number++; // pracujemy na internalCount = N+1 grupach
```

3. **Kopiowanie oryginału**

```
Graph *balancedGraph = copyGraph(originalGraph);
int totalV = originalGraph->numVertices;
```

4. **Obliczenie rozmiarów docelowych**

```
int baseSize  = totalV / number;
int remainder = totalV % number;
int *desiredSizes = calloc(number, sizeof(int));
for (int c = 0; c < number; c++)
    desiredSizes[c] = baseSize + (c < remainder ? 1 : 0);
```

5. **Wielośródłowy BFS**

- Wyznacz stopnie każdego wierzchołka (`degree[u]`),
- Wybierz `number` wierzchołków-ziaren o największym stopniu, nadaj im kolejne `comp = 0…number-1`,
- Rozszerzanie kolejką: dla każdego entry = (comp,u) dopóki `compSize[comp] < desiredSizes[comp]` oznaczaj sąsiadów.

6. **Dopełnienie nieprzypisanych** Przeglądaj każdą krawędź; jeżeli jeden koniec jest przypisany, a drugi nie, dołącz drugi do tej samej grupy.

7. **Filtracja krawędzi** Przepisz do `balancedGraph->edges` tylko te krawędzie, których oba końce mają ten sam numer `component[]`.

8. **Weryfikacja marginesu**

```
bool valid = true;
for (int i = 0; i < number; i++) {
    float perc = compCounts[i] * 100.0f / totalV;
    if (perc > margin) { valid = false; break; }
}
if (!valid) { freeGraph(balancedGraph); return NULL; }
```

9. **Zapis `splitCount` i zwrot**

```
balancedGraph->splitCount = number - 1; // przywracamy oryginalne N
return balancedGraph;
```

4. `main.c`

1. **Parsowanie argumentów**

```
if (argc < 4) { printf("Usage: %s <input> <N> <M> [-o <out>] [-t] [-b]\n", argv[0]); re
char *inputFile  = argv[1];
int   number     = atoi(argv[2]);  // N
float marginPct  = atof(argv[3]);  // M (%)
// następnie obsługa flag -o, -t, -b
```

2. **Wczytanie grafu**

```
Graph *original = graphFromTextFile(inputFile);
if (!original) { fprintf(stderr,"Failed to load graph from file\n"); return 2; }
```

3. **Podział grafu**

```
Graph *splitG = splitGraph(original, number, marginPct);
if (!splitG) { fprintf(stderr,"Failed to split graph into %d parts with margin %.2f%%\n
```

4. **Zapis wyjścia**

```
if (flagBinary)
    graphToBinaryFile(splitG, outputFile);
else
    graphToTextFile(splitG, outputFile);
```

5. **(Opcjonalne) Wyświetlenie CSR-like**

```
if (flagTerminal) {
    char *dump = graphToString(splitG);
    printf("%s\n", dump);
    free(dump);
}
```

6. **Czyszczenie pamięci**

```
freeGraph(original);
freeGraph(splitG);
```

5. Skrypt wizualizujący (`visualize_graph.py`)

- Wczytuje `.txt` CSR-like, buduje `networkx.Graph`, koloruje wierzchołki wg `component[]` i rysuje za pomocą `matplotlib`.
- Główne funkcje:

```
def read_csr_txt(path): ...
def draw_partitioned_graph(G, comp): ...
```

6. Kompilacja – MAKEFILE

```
CC = gcc
CFLAGS = -Wall -O2

all: przetnij-graf

przetnij-graf: main.o graph.o split.o
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c %.h
    $(CC) $(CFLAGS) -c $<

clean:
    rm -f *.o przetnij-graf
```

7. Zarządzanie pamięcią i złożoność

- Wszystkie struktury dynamiczne (`edges`, `desiredSizes`, BFS-queue, tablice pomocnicze) są zwalniane w ścieżkach sukcesu i błędów.
- Złożoność typowo O(V + E) dla BFS, dopełnienie i filtrowanie krawędzi – O(E) + ewentualne iteracje korekcyjne.

8. Punkt wyjścia do rozbudowy

- Możliwość podmiany heurystyki `split.c` na zaawansowane algorytmy (METIS, spectral).
- Przechowywanie struktury w CSR (ewentualnie adjacency list) zamiast surowej tablicy `Edge[]` – lepsza skalowalność.
- Dodanie testów jednostkowych (CUnit, Unity) dla parsera i algorytmu.