JIMP2 Projekt 2025

Dokumentacja końcowa - C

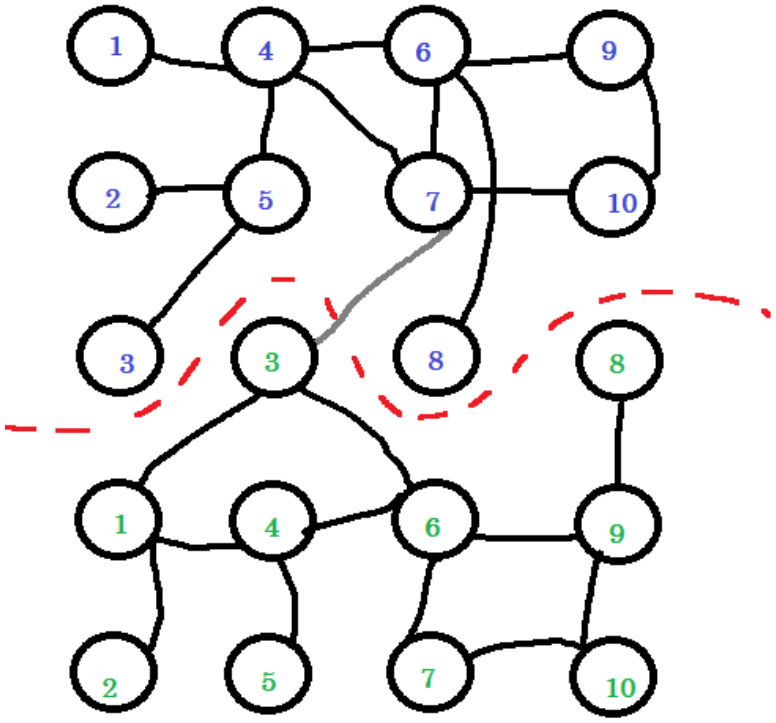
Michał Ludwiczak
GR3

29 kwietnia 2025

Spis treści

1 Cel projektu

Celem projektu jest stworzenie aplikacji w języku C, dokonującej podziału grafu na określoną przez użytkownika lub domyślną liczbę 2 równych części z zachowaniem wybranego lub, ponownie, domyślnego 10-procentowego marginesu różnicy. Liczba wierzchołków w powstałych częściach grafu nie powinna się różnić o więcej niż zadany margines procentowy, a liczba przeciętych krawędzi pomiędzy wynikowymi częściami grafu powinna być jak najmniejsza. Wyjściem programu ma być plik tekstowy lub binarny. Użytkownik ma mieć możliwość wskazać wyjście programu, a więc plik tekstowy lub binarny, zawierający wynik działania programu, oraz dane wejściowe, które mogą być wykorzystane ponownie w kolejnym działaniu programu.



Rysunek 1: Przykładowy graf podzielony na 2 równe części

2 Algorytm

Niestety znalezienie optymalnego podziału grafu należy do klasy problemów NP-trudnych. Dla grafu o n wierzchołkach istnieje aż $2^{n-1} - 1$ możliwych sposobów na bisekcję (wy-

raźnego podziału na 2 grafy). [1] Przy tego typu problemie nie możemy po prostu sprawdzić wszystkich możliwości i wybrać najlepszej z nich - jest to problem optymalizacji kombinatorycznej. W związku z tym opracowano wiele algorytmów zachłannych, metod przybliżonych i heurystyk, które pozwalają na uzyskanie satysfakcjonujących rozwiązań w praktyce. W tym programie zdecydowałem się użyć algorytmu spektralnego, który wykorzystuje własności spektralne macierzy Laplace'a grafu. Na podstawie spektrum grafu, które opisuje strukturę jego strukturę, graf możemy podzielić w efektywny i satysfakcjonujący sposób, minimalizując liczbę przeciętych krawędzi.

2.1 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 diagonali 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.

2.2 Metoda Lanczosa

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. Istnieje metoda Lanczosa [3], którą można zastosować na macierzy Laplace'a, gdyż jest to macierz symetryczna, a więc również i hermitowska. Generuje ona ortonormalną bazę przestrzeni Kryłowa oraz przekształca macierz Laplace'a do macierzy trójdiagonalnej T. Z wartości i wektorów własnych tej uproszczonej macierzy i bazy Kryłowa można obliczyć przybliżenia wartości i wektorów własnych oryginalnej macierzy. Dodatkowo wybór odpowiedniego wektora początkowego jest istotny dla efektywności metody. Często wybiera się go losowo, aby zapewnić, że ma składowe we wszystkich kierunkach przestrzeni własnej macierzy. Metoda Lanczosa może być również jednak podatna na niestabilności numeryczne, zwłaszcza w przypadku dużych macierzy. Aby temu przeciwdziałać, często stosuje się techniki ponownej ortogonalizacji.

2.3 Algorytm QR

W celu obliczenia wartości własnych macierzy trójdiagonalnej **T**, uzyskanej z metody Lanczosa, zastosowano algorytm QR z rotacjami Givensa. Macierz **T** jest rzeczywista,

symetryczna i trójdiagonalna, co umożliwia efektywne wykorzystanie tego algorytmu. Algorytm polega na iteracyjnym rozkładzie macierzy **T** na iloczyn macierzy ortogonalnej **Q** i macierzy górnotrójkątnej **R**, a następnie aktualizacji **T** poprzez mnożenie **RQ**. Rotacje Givensa są stosowane do wyzerowania elementów poddiagonalnych, co zachowuje trójdiagonalność macierzy i redukuje złożoność obliczeniową. W każdej iteracji aktualizowana jest macierz $\mathbf{Q}_{\text{total}}$, która akumuluje iloczyn wszystkich rotacji Givensa, umożliwiając późniejsze obliczenie wektorów własnych. Proces powtarza się, aż do osiągnięcia zbieżności, czyli gdy elementy poddiagonalne macierzy **T** stają się wystarczająco małe. Po zbieżności, wartości własne odczytywane są z diagonali macierzy **T**, a wektory własne są kolumnami macierzy $\mathbf{Q}_{\text{total}}$. Algorytm QR z rotacjami Givensa jest szczególnie efektywny dla macierzy trójdiagonalnych, ponieważ każda rotacja wpływa tylko na dwa wiersze i kolumny, co znacząco redukuje złożoność obliczeniową i pozwala na oszczędność pamięci. Po obliczeniu tych wartości i wektorów własnych oblicza się ich przybliżenia dla macierzy Laplace'a grafu.

2.4 Klasteryzacja

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, zastosowano zmodyfikowaną wersję algorytmu k-średnich (k-means) [4], uwzględniającą ograniczenia na rozmiary klastrów. Centroidy są inicjalizowane na podstawie średniej wartości współrzędnych wzdłuż głównych osi danych. 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. Jeśli algorytm nie osiągnie zbieżności w ustalonej liczbie iteracji, zostaje wyświetlone ostrzeżenie. Proces powtarzany jest wielokrotnie z różnymi inicjalizacjami, aby znaleźć podział minimalizujący liczbę przeciętych krawędzi (tj. krawędzi łączących różne klastry). Wynik jest akceptowany tylko wtedy, gdy zachowany zostaje wymagany margines wielkości klastrów; w przeciwnym przypadku podział jest powtarzany od nowa. Program ma ustaloną maksymalną liczbę prób. Jeżeli podział się nie uda, algorytm wraca do metody Lanczosa.

2.5 Wynik

Po otrzymaniu poszczególnych części z wierzchołkami modyfikuje się macierz sąsiedztwa A wstawiając 0 w pola oznaczające 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.

3 Flagi i argumenty

Program jest uruchamiany z linii poleceń i obsługuje następujące flagi:

<plik_wejściowy>
input

Ścieżka do pliku wejściowego zawierającego opis grafu.
Wymagane jako pierwszy argument

-p `<liczba_części>`
parts
Określa liczbę części, na które ma zostać podzielony graf.
Domyślnie: 2
Minimalna wartość: 2
Maksymalna wartość: zależna od wielkości grafu (podwojona liczba podziałów nie może być większa niż liczba wierzchołków grafu)

-m `<margines>`
margin
Określa dopuszczalny margines procentowy różnicy w liczbie wierzchołków między częściami.
Domyślnie: 10%
Uwaga: Wartości graniczne są ustalane w zależności od liczby wierzchołków grafu. Przykład - nie można podzielić grafu z 7 wierzchołkami z marginesem 5%.

-o `<plik_wyjściowy>`
output
Określa ścieżkę do pliku wyjściowego, w którym zostaną zapisane wyniki.
Domyślnie: output.txt lub output.bin (w zależności od formatu)
Ścieżka: Program zapisuje pliki wyjściowe w folderze output.

-f `<format_pliku_wyjściowego>`
format
Określa format pliku wyjściowego: `txt` dla pliku tekstowego lub `bin` dla pliku binarnego.
Domyślnie: txt
Uwaga: Jeżeli zmieni się format pliku wyjściowego, zmienione zostanie również rozszerzenie pliku z flagi -o.

4 Szczegóły implementacyjne

4.1 Struktura plików

W projekcie obowiązuje poniższa struktura plików:

```
JIMP2-Projekt-2025-C/
├── src/
├── include/
├── input/
├── output/
├── docs/
├── obj/
├── Makefile
└── .gitignore
```

- `src/`
  Przechowywanie plików źródłowych (.c).
- `include/`
  Przechowywanie plików nagłówkowych (.h).
- `input/`
  Przechowywanie plików wejściowych (.csrrg).
- `output/`
  Przechowywanie plików wyjściowych (.txt, .bin).
- `docs/`
  Przechowywanie dokumentacji.
- `obj/`
  Przechowywanie plików wynikowych kompilacji typu object (.o). Folder ten jest tworzony i kasowany przez Makefile.
- `Makefile`
  Organizacja kompilacji programu.
- `.gitignore`
  Przechowywanie listy wyjątków - plików/folderów, które git powinien zignorować.

4.2 Moduły

- `main.c - plik główny`
  Wywołanie modułów w odpowiedniej kolejności.
  Parsowanie argumentów, wczytanie grafu z pliku wejściowego, utworzenie macierzy sąsiedztwa i macierzy Laplace'a. W pętli do sprawdzania sukcesu podziału: obliczenie wektorów własnych i podział grafu. wypisanie wyniku do pliku wyjściowego.
- `config.h / config.c`
  Obsługa argumentów. Walidacja pliku wejścioweg.
- `input.h / input.c`
  Wczytywanie grafu z pliku. Obsługa argumentów w oparciu o dane grafu.
- `mat_vec.h / mat_vec.c`
  Tworzenie macierzy sąsiedztwa grafu w oparciu o dane wejściowe, obliczanie macierzy stopni, obliczanie macierzy Laplace'a grafu. Operacje na wektorach i macierzach: wypisywanie, obliczanie iloczynu skalarnego, obliczanie normy euklidesowej, mnożenia macierzy, ortogonalizacji i normalizacji.
- `eigenvalues.h / eigenvalues.c`
  Implementacja algorytmu Lanczosa, obliczanie wartości i wektorów własnych macierzy trójdiagonalnej i obliczanie przybliżonych wartości wektorów własnych macierzy Laplace'a.

- `clusterization.h / clusterization.c`
  Implementacja algorytmu klasteryzacji k-means do podziału grafu z minimalizacją przeciętych krawędzi i zachowaniem marginesu. Modyfikacja macierzy sąsiedztwa i wypisanie wyniku.

- `test.h / test.c`
  Testy poprawności algorytmu podziału grafu.

- `output.h / output.c`
  Zapisanie wyniku w pliku wyjściowym w odpowiednim formacie - tekstowym lub binarnym.

4.3 Struktury

- `Config`
  Przechowywanie informacji podanych przez użytkownika wsadowo.

```c
// Struktura do przechowywania ustawień
typedef struct
{
    int parts;
    int margin;
    char *input_file;
    char *output_file;
    char *format;
} Config;
```

- `Input`
  Przechowywanie informacji pobranych z pliku wejściowego. Informacje te są później wykorzystywane do utworzenia macierzy sąsiedztwa oraz wypisywania danych do pliku wyjściowego.

```c
// Struktura do przechowywania danych z pliku wejściowego
typedef struct
{
    FILE *in;
    int max_vertices;
    int *row_indices;
    int r_count;
    int *first_vertices;
    int f_count;
    int *vertices_groups;
    size_t g_count;
    int *vertices_ptrs;
    size_t p_count;
    int v_count;
    size_t len;
} Input;
```

- `LanczosEigenV`
  Przechowywanie zmiennych do metody Lanczosa oraz do obliczania wartości wła-

snych, przybliżonych wektorów własnych macierzy Laplace'a grafu oraz do podziału grafu.

```c
// Struktura do przechowywania zmiennych potrzebnych do metody Lanczosa oraz do obliczenia przybliżeń wektorów własnych macierzy Laplace'a grafu
typedef struct
{
    // rozmiar macierzy Laplace'a
    int n;
    // liczba iteracji
    int m;
    // baza ortonormalna V z wektorami v (baza przestrzeni Kryłowa)
    double* V;
    // wektory w (macierz)
    double* W;
    // macierz trójdiagonalna T
    double* alpha;
    double* beta;
    // wartości własne macierzy T
    double* theta;
    // wektory własne macierzy T (macierz)
    double* Y;
    // przybliżone wektory własne macierzy L (macierz)
    double* X;
} LanczosEigenV;
// Nazwy takie same jak na Wikipedii - Lanczos algorithm
// https://en.wikipedia.org/wiki/Lanczos_algorithm
```

- `Result`
  Przechowywanie zmiennych do wypisania wyniku w pierwszej linijce pliku wyjściowego.

```c
// Struktura do przechowywania wyniku
typedef struct
{
    char res;
    int parts;
    int cut_count;
    int margin_kept;
} Result;
```

4.4 Najważniejsze funkcje

- `parse_args`
  Obsługa argumentów. Zapisanie konfiguracji do obiektu struktury Config. Sprawdzenie i poprawa rozszerzenia pliku wyjściowego dla ustawionego formatu.

```c
// Parsowanie argumentów, zwrócenie skonfigurowanej struktury
Config parse_args(int argc, char **argv)
{
    Config c;

    // Ustawienia domyślne
    c.parts = 2;
    c.margin = 10;
    c.input_file = NULL;
    c.output_file = "output.txt";
    c.format = "txt";

    // Parsowanie argumentów
    int opt;
    while ((opt = getopt(argc, argv, "p:m:o:f:")) != -1)···

    // Sprawdzenie rozszerzenia pliku wyjściowego i poprawienie go, jeśli
    jest nieprawidłowe
    if (c.output_file != NULL)···

    return c;
}
```

- `read_input`
  Wczytanie danych z pliku wejściowego do obiektu struktury Input.

```c
// Czytanie pliku linia po linii
void read_input(Input *i)
{
    ssize_t read;
    char *line = NULL;
    int line_number = 0;

    while((read = getline(&line, &i->len, i->in)) != -1)
    {
        line_number++;

        // Odczytanie maksymalnej liczby wierzchołków w wierszu
        if(line_number == 1)
        {
            i->max_vertices = atoi(line);
        }
        // Odczytanie indeksów wierszy
        else if(line_number == 2)···
        // Odczytanie pierwszych wierzchołków wierszy
        else if(line_number == 3)···
        // Odczytanie grup
        else if(line_number == 4)···
        // Odczytanie wskaźników
        else if(line_number == 5)···
    }

    // Zwolnienie pamięci dla odczytu linii
    free(line);

    // Zamknięcie pliku wejściowego
    fclose(i->in);
}
```

- `get_adjacency_matrix`

Zapisanie grafu w formie macierzy sąsiedzywa.

```c
// Wczytanie grafu do macierzy sąsiedztwa A
int* get_adjacency_matrix(Input *i)
{
    printf("\n\tPołączenia dodane do macierzy sąsiedztwa:");
    int *A = calloc(i->v_count * i->v_count, sizeof(int));
    int p = 0;
    int v = 0;
    for(int it = 0; it < (int)i->g_count; it++)
    {
        // Zaktualizuj wierzchołek
        if(p < (int)i->p_count && it == i->vertices_ptrs[p])
        {
            v = i->vertices_groups[it];
            printf("\n\t%d - %d: ", it, v);
            p++;
        }
        else
        {
            printf("\t%d", i->vertices_groups[it]);
            add_edge(A, v, i->vertices_groups[it], i->v_count);
        }
    }
    printf("\n");

    // Test
    //printf("\]n\t%d\n", getv(A, 93, 90, v_count));

    // Wyświetlenie macierzy sąsiedztwa
    if(i->v_count < 30)
    {
        printf("\n");
        printv(A, i->v_count * i->v_count, i->v_count);
    }

    return A;
}
```

- `calc_laplacian`
  Obliczenie macierzy Laplace'a grafu.

```c
// Obliczenie macierzy Laplace'a grafu L
int* calc_laplacian(int* A, int* D, int n)
{
    // = 0
    int* L = calloc(n * n, sizeof(int));
    if(L == NULL)
    {
        fprintf(stderr, "Błąd alokacji pamięci!\n");
        free(A);
        free(D);
    }

    // + D
    for(int i = 0; i < n; i++)
    {
        L[i * n + i] = D[i];
    }

    // - A
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            L[i * n + j] -= A[i * n + j];
        }
    }

    // Wyświetlenie macierzy Laplace'a
    if(n < 30)
    {
        printf("\n");
        printv(L, n * n, n);
    }

    return L;
}
```

- `lanczos`
  Algorytm Lanczosa - przekształcenie problemu obliczenia wartości wektorów własnych macierzy Laplace'a na ich obliczenie dla macierzy trójdiagonalnej.

```
162  // Iteracje metody Lanczosa dla j = 2, ..., m
163  void lanczos(LanczosEigenV *l, int* A)
164  {
165      int n = l->n;
166      int m = l->m;
167      double *V = l->V;
168      double *W = l->W;
169      double *alpha = l->alpha;
170      double *beta  = l->beta;
171
172      // Inicjalizacja wektora tymczasowego dla wj' a potem dla w
173      double *w = (double *)malloc(n * sizeof(double));
174      if(w == NULL)
175      {
176          fprintf(stderr, "Błąd alokacji pamięci dla w.\n");
177          exit(EXIT_FAILURE);
178      }
179
180      // Pętla dla j = 2, ..., m (tu: j = 1, ..., m - 1)
181  >   for (int j = 1; j < m; ++j)···
234
235      free(w);
236  }
```

- `qr_algorithm`
  Obliczenie wartości wektorów własnych dla macierzy trójdiagonalnej.

```
293  // Algorytm QR z rotacjami Givensa do obliczenia wartości własnych macierzy
     trójdiagonalnej T
294  void qr_algorithm(LanczosEigenV *l)
295  {
296      // Inicjalizacja macierzy Q jako macierzy jednostkowej
297      double* Q_total = (double *)calloc(l->m * l->m, sizeof(double));
298  >   if (Q_total == NULL)···
303  >   for (int i = 0; i < l->m; ++i)···
307
308      double* T = build_T(l);
309      int iter = 0;
310      int converged = 0;
311
312      // QR rozkład T = QR (rotacja Givensa)
313  >   while (iter < MAX_ITER && !converged)···
386
387      // Zapisanie wartości własnych z przekątnej macierzy T
388  >   for (int i = 0; i < l->m; ++i)···
392
393      // Wypisanie wartości własnych
394      printf("\n\tWartości własne macierzy T:\n");
395  >   for (int i = 0; i < l->m; ++i)···
399      printf("\tLiczba iteracji: %d\n", iter);
400  >   if (converged)···
404  >   else···
408
409      // Inicjalziacja macierzy Y
410      l->Y = (double *)calloc(l->m * l->m, sizeof(double));
411  >   if (l->Y == NULL)···
418
419      // Zapisanie wektorów własnych macierzy T z Q_total
420  >   for (int i = 0; i < l->m * l->m; ++i)···
424
425      // Wypisanie wektorów własnych
426  >   if (l->m < 20)···
440
441      free(T);
442      free(Q_total);
443  }
```

- `clusterization`
  Algorytm klasteryzacji - podziału wierzchołków na grupy w oparciu o wartości wektorów własnych macierzy Laplace'a grafu.

```c
// Algorytm klasteryzacji centroidów (k-means) z minimalizacją liczby przecięć i modyfikacją macierzy sąsiedztwa
Result *clusterization(double *X, int v_count, int parts, int dimensions, double margin_percentage, int *A)
{
    // Sprawdzenie poprawności danych wejściowych
    if (parts <= 0 || v_count <= 0 || X == NULL || A == NULL || parts > v_count || margin_percentage < 0.0 || margin_percentage > 100.0)...

    int *best_labels = NULL;
    int min_intersection_count = INT_MAX;

    for (int attempt = 0; attempt < MAX_ATTEMPTS; attempt++)...

    // Alokacja pamięci dla wyniku
    Result *r = malloc(sizeof(Result));
    if (!r)...

    r->cut_count = min_intersection_count;
    int max_cluster_size = 0;
    int min_cluster_size = INT_MAX;

    // Obliczenie rozmiarów klastrów
    int *cluster_sizes = calloc(parts, sizeof(int));
    if (!cluster_sizes)...

    for (int i = 0; i < v_count; i++)...

    for (int i = 0; i < parts; i++)...

    free(cluster_sizes);

    // Obliczenie zachowanego marginesu
    int actual_margin = max_cluster_size - min_cluster_size;
    r->margin_kept = (int)((double)actual_margin / v_count * 100.0);
    if (r->margin_kept <= margin_percentage)...
    else...
    r->parts = parts;

    free(best_labels);
    return r; // Zwróć wskaźnik na wynik
}
```

- `write_output`
  Wypisanie pliku wyjściowego z wynikiem i podzielonym grafem w odpowiednim formacie.

```c
// Wypisywanie grafu do pliku
void write_output(char *output_file, Result *r, Input *i, int *A, int n,
char *format)
{
    // Sprawdzenie, czy format jest poprawny
    if (strcmp(format, "txt") != 0 && strcmp(format, "bin") != 0)···

    // Tworzenie folderu output, jeśli nie istnieje
    if (mkdir("output", 0777) == -1 && errno != EEXIST)···

    // Tworzenie ścieżki do pliku wyjściowego w folderze output/
    char output_path[256];
    snprintf(output_path, sizeof(output_path), "output/%s", output_file);

    // Otworzenie pliku wyjściowego
    FILE *out = fopen(output_path, "w");
    if (out == NULL)···

    if(strcmp(format, "txt") == 0)···
    else if (strcmp(format, "bin") == 0)···

    fclose(out);
}
```

4.5 Pliki wejściowe i wyjściowe

4.5.1 Plik wejściowy (`.csrrg`)

Program akceptuje pliki wejściowe z rozszerzeniem `.csrrg`. Format pliku składa się z pięciu sekcji, zapisanych w kolejnych linijkach:

1. Maksymalna liczba wierzchołków w dowolnym wierszu macierzy (graf może mieć wiersze o mniejszej liczbie sąsiadów, ale nie większej).
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.

Przykład:

input > graf.csrrg

```
1  18
2  3;5;6;9;10;13;14;15;10;12;15;4;5;7;8;9;12;14;4;8;9;10;11;12;13;14;15;1;2;3;7;9;13;15;16;1;3;
3  0;0;8;11;18;27;35;41;47;57;67;67;76;82;90;90;94;98;105
4  0;72;39;91;4;54;1;47;4;79;101;71;2;76;79;5;3;63;71;80;42;4;75;5;49;6;93;16;98;7;75;82;50;8;2
5  0;6;12;16;21;23;25;29;33;38;42;45;49;53;55;62;65;68;70;73;76;82;87;89;97;100;103;107;112;116
```

4.5.2 Plik wyjściowy (`.txt` / `.bin`)

Po przetworzeniu danych program zapisuje wynik do pliku wyjściowego w formacie tekstowym lub binarnym.

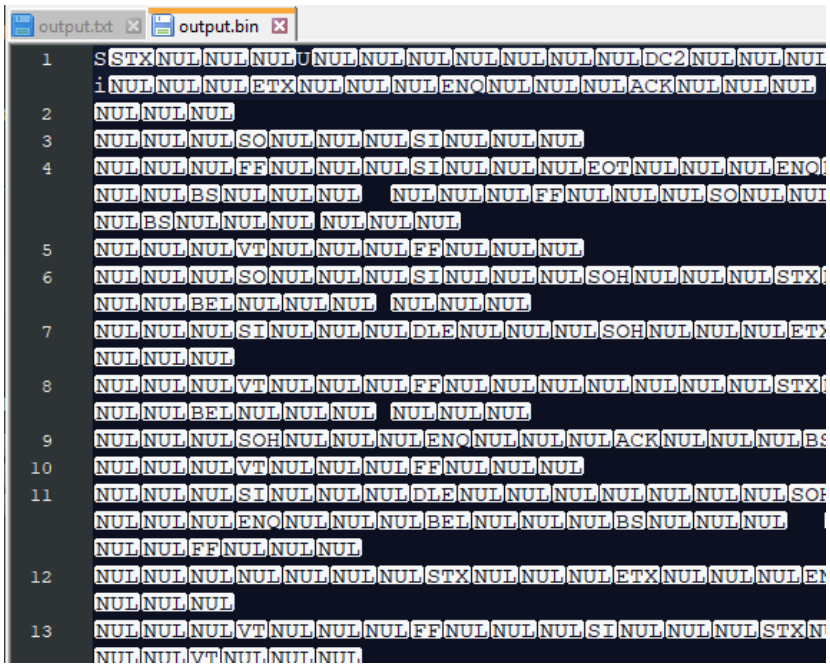
- **Pierwsza linijka** zawiera wynik działania programu w formacie:
  `<wynik (S - sukces, F - porażka)> <liczba_części> <liczba_przecięć> <zachowany_margines>`
  Przykład: `S 3 2 5`
- **Kolejne linie** opisują strukturę grafu w tym samym formacie co plik wejściowy
- **Format binarny** (`.bin`) zapisuje te same informacje co plik tekstowy, lecz w postaci binarnej za pomocą funkcji `fwrite`, co umożliwia szybsze wczytywanie i mniejszy rozmiar pliku. Przed wypisaniem każdej tablicy podawana jest liczba elementów do poprawnego odczytu.

Przykład fragmentu pliku tekstowego:

output.txt

```
1  S 2 90 2
2  18
3  3;5;6;9;10;13;14;15;10;12;15;4;5;7;8;9;12;14;4;8;9;
   1;2;3;7;9;13;15;16;1;3;5;10;11;12;0;2;3;7;9;13;1;5;
   ;16;0;1;4;5;7;8;9;11;12;13;0;2;3;5;6;10;11;12;15;2;
   5;6;9;12;13;14;3;5;9;12;0;2;12;14;2;3;7;10;11;12;14
4  0;0;8;11;18;27;35;41;47;57;67;67;76;82;90;90;94;98;
5  0;4;39;54;72;91;1;4;47;71;79;101;2;5;76;79;3;42;63;
   16;93;98;7;50;75;82;8;28;34;47;60;9;14;28;53;10;57;
   47;50;98;13;18;14;32;63;78;84;100;103;15;23;82;16;4
   70;19;70;91;20;37;54;67;68;103;21;69;89;91;97;22;89
   ;75;87;24;47;53;25;73;91;26;37;62;80;27;42;76;77;10
   30;82;85;95;31;45;52;71;94;32;84;102;33;46;54;55;34
   ;95;100;37;50;52;61;99;38;43;49;88;97;39;45;40;64;9
   2;44;46;48;97;43;44;91;45;46;46;52;55;81;47;48;49;8
   8;91;51;52;53;60;54;74;92;55;58;56;60;71;57;85;58;6
   0;61;62;63;66;98;102;64;65;78;66;67;104;68;72;73;80
   2;71;93;72;96;100;73;84;74;75;89;76;78;77;86;78;86;
   83;98;84;85;95;102;86;87;95;88;89;90;91;92;93;94;10
   9;100;101;102;103;104;
6  0;6;12;16;21;23;25;29;33;38;42;45;49;53;55;62;65;68
   ;97;100;103;107;112;113;117;121;126;129;133;136;138
   57;162;167;168;170;172;176;177;178;182;187;188;189;
   1;204;207;209;210;211;215;216;218;219;221;225;228;2
   ;241;243;245;249;250;251;252;253;255;256;259;260;26
   267;270;271;272;273;274;275;276;277;278;279;280
```

Przykład fragmentu pliku binarnego:



4.6 Testy

4.6.1 Test 1: Funkcje `dot_product`, `norm` i `mat_vec_multiply`

Pierwszy test sprawdza działanie funkcji do obliczania iloczynu skalarnego, normy wektora oraz mnożenia macierzy przez wektor. W ramach testu obliczono:

- Iloczyn skalarny wektorów `v1` i `v2`.
- Normę wektora `v1`.
- Wynik mnożenia macierzy `M` przez wektor `v1`.

4.6.2 Test 2: Funkcje `orthogonalize` i `normalize`

Drugi test weryfikuje działanie funkcji odpowiedzialnych za ortogonalizację i normalizację wektorów. Sprawdzono:

- Ortogonalizację wektora `v` względem wektora `V1`.
- Normalizację wektora `v` oraz wektora `u`.
- Obsługę błędów dla próby normalizacji wektora zerowego.

4.6.3 Test 3: Metoda Lanczosa, algorytm QR i dzielenie

Trzeci test dotyczy implementacji metody Lanczosa oraz algorytmu QR. Sprawdzono:

- Inicjalizację i normalizację wektora `v1`.

- Obliczenie wartości własnych i wektorów własnych macierzy Laplace'a grafu.
- Dzielenie grafu na dwie części.

Wszystkie testy zakończyły się pomyślnie.

5 Uruchomienie

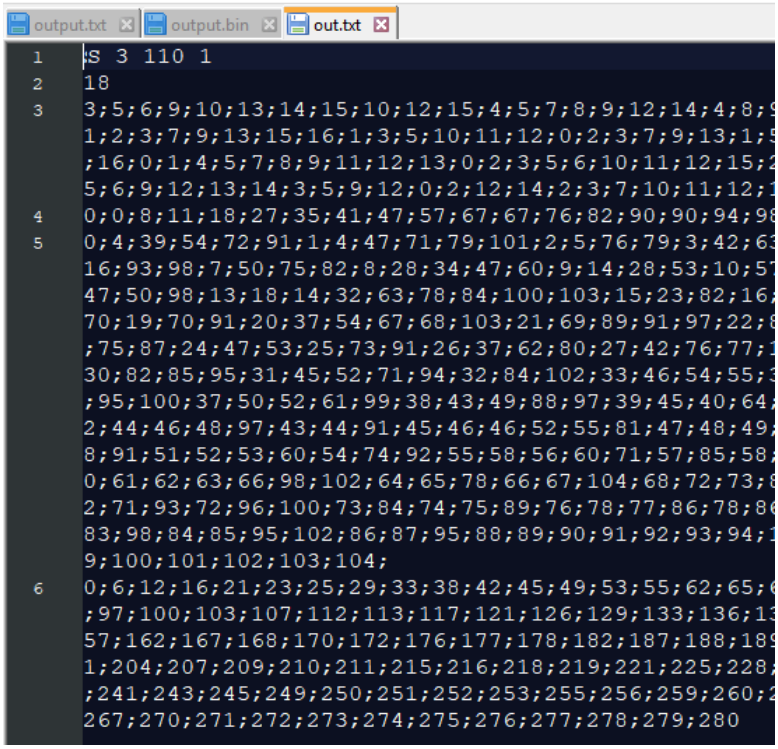
Aby skompilować i uruchomić program, należy posiadać zainstalowane narzędzia:

- `gcc` – kompilator języka C,
- `make` – narzędzie do automatyzacji procesu kompilacji.

Proces kompilacji jest zautomatyzowany za pomocą pliku `Makefile`. Aby skompilować projekt, wystarczy w terminalu wydać polecenie 'make'. Po zakończeniu kompilacji, aby usunąć pliki wygenerowane podczas procesu (np. pliki obiektowe, pliki wykonywalne), można użyć polecenia 'make clear' lub 'make clear'. Polecenie `make clean` usunie pliki, które zostały utworzone w wyniku kompilacji, umożliwiając "czyste"ponowne zbudowanie projektu w przyszłości. Aby uruchomić program, należy postępować zgodnie z instrukcjami przedstawionymi w sekcji Flagi i argumenty. Przykład uruchomienia programu:

```
./graphdivider input/graf.csrrg -p 3 -o out.txt
        Plik wejściowy: input/graf.csrrg
        Liczba części: 3
        Margines: 10%
        Plik wyjściowy: out.txt
        Format: txt

        Limit wierzchołków w wierszu: 18

        Graf jest zbyt duży, aby wyświetlić szczegóły.

        Liczba wierzchołków: 105

        Inicjalizacyjny krok iteracyjny metody Lanczosa:
        alpha1 = 0.828009
        Liczba iteracji: 5
        Nie osiągnięto zbieżności po 5 iteracjach.
        Wynik klasteryzacji:
                Liczba przecięć: 110
                Zachowany margines: 1
                Wynik: S
Liczba prób podziału: 1
```

W pliku wyjściowym zostaje zapisany rezultat oraz informacje o podzielonym grafie:



Przykład uruchomienia programu z wyjściem binarnym:

```
./graphdivider input/graf.csrrg -o out -f bin -m 20
       Plik wejściowy: input/graf.csrrg
       Liczba części: 2
       Margines: 20%
       Plik wyjściowy: out.bin
       Format: bin

       Limit wierzchołków w wierszu: 18

       Graf jest zbyt duży, aby wyświetlić szczegóły.

       Liczba wierzchołków: 105

       Inicjalizacyjny krok iteracyjny metody Lanczosa:
       alpha1 = 0.995677
       Liczba iteracji: 5
       Nie osiągnięto zbieżności po 5 iteracjach.
       Wynik klasteryzacji:
              Liczba przecięć: 76
              Zachowany margines: 10
              Wynik: S
Liczba prób podziału: 1
```

6 Problemy i możliwe usprawnienia

Podczas pracy nad programem napotkano kilka istotnych problemów:

1. **Nieoptymalna liczba przecięć:** Liczba przecięć nie zawsze jest optymalna. Niedokładne przybliżenia wartości własnych mogą prowadzić w konsekwencji do nieoptymalnego podziału.

2. **Problemy z większymi grafami:** Przy zastosowaniu metody Lanczosa do większych grafów mogą wystąpić problemy związane z wydajnością obliczeniową i stabilnością numeryczną. Dzielenie większych grafów trwa długo.

3. **Optymalizacja przechowywania macierzy:** Przechowywanie macierzy w tablicach dynamicznych jest nieefektywne w przypadku macierzy rzadkich, w których przechowuje się mnóstwo zer. Zamiast tego, warto by było rozważyć zastosowanie jednego z popularnych formatów przechowywania macierzy rzadkich, przykładowo Compressed Sparse Row (CSR).

Literatura

[1] Leonid Zhukov, *Lecture 7. Graph partitioning algorithms.*, YouTube, 24 luty 2021, Dostępny na 1 kwietnia 2025 w: `https://youtu.be/zZae_C2BU_4`

[2] *Laplacian matrix*, Wikipedia, Dostępne na 1 kwietnia 2025 w: `https://en.wikipedia.org/wiki/Laplacian_matrix`

[3] *Lanczos algorithm*, Wikipedia, Dostępne na 1 kwietnia 2025 w: `https://en.wikipedia.org/wiki/Lanczos_algorithm`

[4] *Algorytm centroidów*, Wikipedia, Dostępne na 1 kwietnia 2025 w: `https://pl.wikipedia.org/wiki/Algorytm_centroid%C3%B3w`