Sprawozdanie z miniprojektu 3

Mateusz Kamieniecki

SPRAWOZDANIE	**Data wykonania: 30.11.2025**
Tytuł miniprojektu:	*Grafy*
Wykonał:	*Mateusz Kamieniecki*
Sprawdził:	*dr inż. Konrad Markowski*

Spis treści

1 Cel projektu

Celem projektu było stworzenie losowego spójnego grafu, przedstawienie go w postaci macierzy sąsiedztwa, macierzy incydencji oraz listy sąsiedztwa. Program miał wypisywać wyrysowany graf, listę lub macierz odpowiadającą grafu, stopnie wierzchołków oraz podać nam wszystkie łuki i multiłuki w grafie. Ja osobiście uznałem, że łuki i multiłuki będą

podane podczas rysowania grafu. Postanowiłem jeszcze pominąć pętelki, ponieważ wprowadzają one dodatkowe niepewności co do macierzy incydencji oraz stopni wierzchołków.

2 Rozwiązanie problemu

Aby rozwiązać problem musimy najpierw stworzyć spójny graf. W tym celu musimy stworzyć minimalne drzewo rozpinające do którego później będziemy dodawać resztę krawędzi.

2.1 Tworzenie losowego grafu spójnego

Minimalne drzewo rozpinające tworzymy poprzez losowe łączenie wierzchołków, które jeszcze nie miały połączenia. Gdy każdy wierzchołek dostał połączenie z innym wierzchołkiem, mamy gotowe minimalne drzewo rozpinające.

Gdy mamy już minimalne drzewo rozpinające, możemy dodać resztę krawędzi. Robimy to poprzez losowe wybieranie dwóch wierzchołków i dodawanie między nimi krawędzi. Krawędzie teraz mogą się powtarzać, więc możemy mieć multiłuki.

Dla każdej metody reprezentacji grafu (macierz sąsiedztwa, macierz incydencji, lista sąsiedztwa) będziemy korzystać z macierzy sąsiedztwa do tworzenia grafu, ponieważ ona jest najprostsza do zrobienia.

2.2 Macierz sąsiedztwa

2.2.1 Rysowanie grafu

Gdy mamy wygenerowany graf w postaci macierzy sąsiedztwa, możemy go narysować iterując się przez tablicę i wypisać krawędź jak napotkamy niezerową wartość w tablicy. Liczba w danej komórce będzie reprezentować liczbę krawędzi między dwoma wierzchołkami.

2.2.2 Stopnie wierzchołków

Aby obliczyć stopnie wierzchołków, iterujemy się przez macierz sąsiedztwa i dla każdej niezerowej wartości w komórce dodajemy do stopnia wierzchołka wartość z komórki odpowiednio do stopnia wejścia z danego wierzchołka oraz stopnia wyjścia danego wierzchołka.

2.3 Macierz incydencji

2.3.1 Rysowanie grafu

Aby narysować graf z macierzy incydencji, iterujemy się przez kolumny macierzy. Dla każdej kolumny sprawdzamy, które wiersze mają wartość 1 (krawędź wychodząca) oraz -1 (krawędź wchodząca). Gdy zliczyliśmy wszystkie krawędzie, wypisujemy je.

2.3.2 Stopnie wierzchołków

Aby obliczyć stopnie wierzchołków z macierzy incydencji, iterujemy się przez wiersze macierzy. Dla każdego wiersza zliczamy liczbę wystąpień 1 (stopień wyjścia) oraz -1 (stopień wejścia).

2.4 Lista sąsiedztwa

2.4.1 Rysowanie grafu

Aby narysować graf z listy sąsiedztwa, iterujemy się przez każdy wierzchołek i wypisujemy wszystkie krawędzie wychodzące z danego wierzchołka oraz ewentualnie ilość krawędzi.

2.4.2 Stopnie wierzchołków

Aby obliczyć stopnie wierzchołków z listy sąsiedztwa, iterujemy się przez każdy wierzchołek. Stopień wejścia dla danego wierzchołka obliczamy poprzez zliczanie ile razy dany wierzchołek pojawia, a stopień wyjścia to po prostu długość listy sąsiedztwa danego wierzchołka.

3 Szczegóły implementacji

3.1 Wczytywanie danych

Na początku sprawdzamy czy mamy odpowiednie liczby wierzchołków oraz krawędzi, aby utworzyć spójny graf. Następnie w zależności od podanej metody wywołujemy odpowiednią funkcję tworzącą graf.

Listing 1: Wczytywanie danych wejściowych

```
if (n <= 1) {
    fprintf(stderr, "Liczba␣wierzchołków␣musi␣być␣większa␣od␣
       jedynki\n");
    free(method);
    return 1;
}
if (m < 0) {
    fprintf(stderr, "Liczba␣krawędzi␣nie␣może␣być␣ujemna\n");
    free(method);
    return 1;
}
if (n > 1 && m == 0) {
    fprintf(stderr, "Graf␣z␣więcej␣niż␣jednym␣wierzchołkiem␣i␣
       zerową␣"
                    "liczbą␣krawędzi␣nie␣jest␣spójny\n");
    free(method);
    return 1;
}
if (m < n - 1) {
    fprintf(stderr, "Za␣mało␣krawędzi,␣aby␣graf␣był␣spójny\n");
    free(method);
    return 1;
}
if (strcmp(method, "ms") == 0)
    adjacency_matrix(n, m);
else if (strcmp(method, "mi") == 0)
```

```
    incidence(n, m);
else if (strcmp(method, "ls") == 0)
    adjacency_list(n, m);
else {
    printf("Zła metoda\n");
    free(method);
    return 1;
}
free(method);
```

3.2 Sposób przechowywania macierzy sąsiedztwa i macierzy incydencji

Macierz sąsiedztwa i macierz incydencji są przedstawione jako 2d tablice, które są inicjalizowane przy pomocy calloca, aby wartości były na początku wyzerowane, wartości rozmiaru tablicy były dowolne, oraz aby można było używać syntaxu t[][]

Listing 2: Inicjalizacja macierzy sąsiedztwa i incydencji

```
int (*adj_matrix)[n] = calloc(n, sizeof *adj_matrix);
int (*inc_matrix)[m] = calloc(n, sizeof *inc_matrix);
```

3.3 Sposób przechowywania listy sąsiedztwa

Używamy struktury dynamicznej tablicy do przechowywania listy sąsiedztwa. Każdy wierzchołek ma swoją własną tablicę sąsiadów, która może się dynamicznie rozrastać w razie potrzeby.

Listing 3: Struktura dynamicznej tablicy

```
struct vector {
    int *neighbors;
    int size;
    int capacity;
};
```

Listing 4: Inicjalizacja dynamicznej tablicy

```
void vector_init(struct vector *vec) {
    vec->size = 0;
    vec->capacity = 4;
    vec->neighbors = malloc(vec->capacity * sizeof(int));
}
```

Listing 5: Dodawanie elementu do dynamicznej tablicy

```
void vector_push_back(struct vector *vec, int value) {
    // zwiększanie pojemności, jeśli potrzeba
    if (vec->size >= vec->capacity) {
        vec->capacity *= 2;
        vec->neighbors = realloc(vec->neighbors, vec->capacity
            * sizeof(int));
```

```
    }
    vec->neighbors[vec->size++] = value;
}
```

Listing 6: Zwalnianie dynamicznej tablicy

```
void vector_free(struct vector *vec) { free(vec->neighbors); }
```

4 Sposób wykonywania programu

Program jest kompilowany za pomocą GNU make. Aby skompilować program, należy wykonać polecenie `make` w katalogu głównym projektu. Spowoduje to utworzenie pliku wykonywalnego o nazwie `main`.

Aby wyczyścić pliki obiektowe i plik wykonywalny, należy wykonać polecenie `make clean`.

W przypadku braku zainstalowanego GNU make, program można skompilować ręcznie za pomocą następującego polecenia:

Listing 7: Ręczna kompilacja programu

```
gcc -Wall -Wextra -std=c11 -o main 343333.c adjacency_matrix.c
    incidence_matrix.c adjacency_list.c
```

Aby uruchomić program, należy wykonać polecenie `./main` i podać na wejściu liczbę wierzchołków, liczbę krawędzi oraz metodę reprezentacji grafu (ms - macierz sąsiedztwa, mi - macierz incydencji, ls - lista sąsiedztwa). Przykładowe polecenie uruchamiające program z 5 wierzchołkami, 7 krawędziami i reprezentacją w postaci macierzy sąsiedztwa wygląda następująco:

Rysunek 1: Uruchomienie programu dla przykładowych danych wejściowych

5 Wnioski i spostrzeżenia

W ramach projektu udało się zaimplementować trzy różne metody reprezentacji grafów skierowanych: macierz sąsiedztwa, macierz incydencji oraz listę sąsiedztwa. Każda z tych metod ma swoje zalety i wady w kontekście złożoności pamięciowej i czasowej.

5.1 Porównanie metod reprezentacji

Macierz sąsiedztwa jest najprostsza w implementacji i pozwala na szybkie sprawdzenie czy istnieje krawędź między dwoma wierzchołkami w czasie $O(1)$. Wadą jest jednak duże zużycie pamięci $O(n^2)$, co dla rzadkich grafów (mało krawędzi) jest nieefektywne. Łatwo w niej również przechowywać multikrawędzie poprzez zwiększanie wartości w odpowiedniej komórce.

Macierz incydencji zajmuje pamięć $O(n \cdot m)$, co dla gęstych grafów może być bardziej efektywne niż macierz sąsiedztwa. Pozwala na łatwe rozróżnienie krawędzi wchodzących i wychodzących poprzez użycie wartości dodatnich i ujemnych. Wadą jest większa złożoność operacji - sprawdzenie czy istnieje krawędź wymaga przejrzenia całej kolumny.

Lista sąsiedztwa jest najbardziej efektywna pamięciowo dla rzadkich grafów, zajmując $O(n + m)$ pamięci. Iteracja po sąsiadach danego wierzchołka jest bardzo szybka. Wadą jest większa złożoność implementacji wymagająca użycia dynamicznych struktur danych oraz trudniejsze sprawdzanie istnienia konkretnej krawędzi.

5.2 Wnioski dotyczące implementacji

Implementacja generowania losowego grafu spójnego wymagała zapewnienia minimalnego drzewa rozpinającego poprzez pierwsze $n - 1$ krawędzi, które nie mogły się powtarzać. Dopiero kolejne krawędzie mogły tworzyć multikrawędzie. Takie podejście gwarantuje spójność grafu przy dowolnej liczbie krawędzi większej lub równej $n - 1$.

Użycie macierzy sąsiedztwa jako pośredniej struktury dla pozostałych dwóch metod okazało się dobrym rozwiązaniem, upraszczającym kod i zapewniającym spójność danych wejściowych dla wszystkich reprezentacji.

Dynamiczna struktura wektora dla listy sąsiedztwa pozwoliła na elastyczne zarządzanie pamięcią, minimalizując jej zużycie przy jednoczesnym zachowaniu prostoty użycia.