Specyfikacja implementacyjna - Graf

Daria Danieluk, Weronika Zbierowska

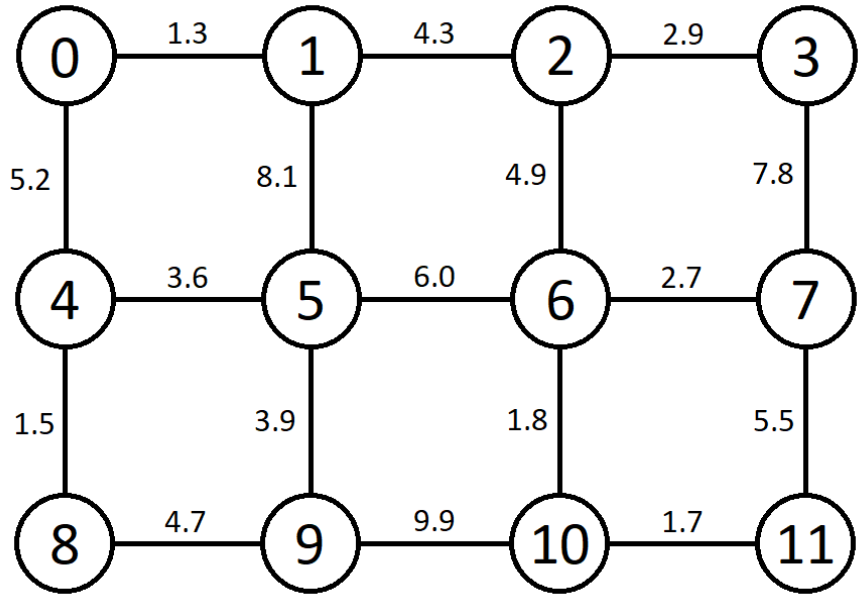
19.03.2022 r.

1 Cel projektu

Celem projektu jest stworzenie programu, który generuje i analizuje prostokątny, nieskierowany, ważony graf. Analiza polega na możliwości sprawdzenia, czy graf jest spójny za pomocą algorytmu BFS (breadth-first search) oraz znalezienia najkrótszej ścieżki pomiędzy dwoma wskazanymi przez użytkownika wierzchołkami grafu za pomocą algorytmu Dijkstry.

Wierzchołki grafu są ponumerowane od 0, od lewej do prawej strony grafu, a następnie przechodzą do wiersza poniżej.

Przykładowy graf:



2 Wstęp teoretyczny

2.1 Algorytm BFS

Algorytm BFS (breadth-first search), czy inaczej algorytm przeszukiwania wszerz, polega na przejściu grafu od wybranego wierzchołka aż do momentu, gdy wszystkie wierzchołki, do których można było dotrzeć zostały odwiedzone. Ze względu na swoją specyfikę może zostać wykorzystany do sprawdzania spójności grafu, ponieważ odwiedzone zostaną jedynie wierzchołki, które znajdują się w tym samym spójnym segmencie, co wierzchołek początkowy.

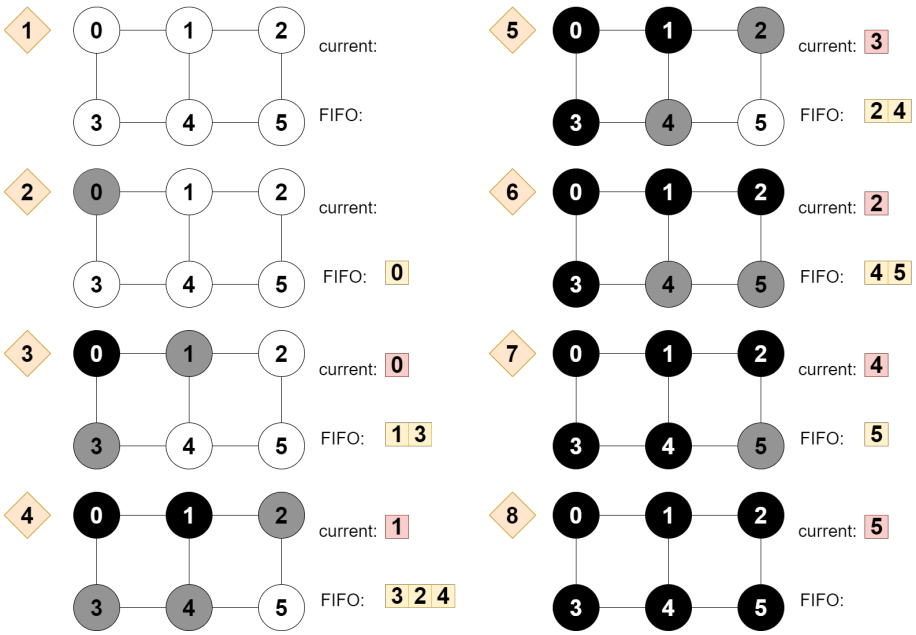
Każdy wierzchołek może się znaleźć w 1 z 3 stanów: nieodwiedzony (biały), odkryty (szary) i odwiedzony (czarny). Na początku wszystkie wierzchołki są białe. Gdy wierzchołek został dodany do kolejki FIFO, staje się szary. Gdy wszystkie jego sąsiednie wierzchołki zostały odkryte lub odwiedzone, staje się on czarny.

Aby sprawdzić spójność grafu, należy przeanalizować stany wszystkich wierzchołków. Jeśli którykolwiek pozostał biały po przejściu algorytmu BFS, to graf jest niespójny.

Lista kroków - algorytm BFS:

1. Pomaluj wszystkie wierzchołki grafu na biało.
2. Pomaluj wierzchołek początkowy na szaro.
3. Dodaj wierzchołek początkowy do kolejki.
4. Dopóki kolejka nie jest pusta wykonuj kroki 5. - 10.
5. Wyjmij z kolejki wierzchołek `u`.
6. Dla każdego wierzchołka sąsiedniego do `u` wykonaj krok 7., następnie przejdź do kroku 10.
7. Jeśli wierzchołek jest biały wykonaj kroki 8. - 9., w przeciwnym wypadku wróć do kroku 6.
8. Pomaluj wierzchołek na szaro.
9. Dodaj wierzchołek do kolejki.
10. Pomaluj wierzchołek `u` na czarno.

Przykład:



2.2 Algorytm Dijkstry

Algorytm Dijkstry służy do znajdowania najkrótszej ścieżki w grafie. Dla każdego wierzchołka w grafie zapamiętywana jest długość aktualnie najkrótszej ścieżki od wierzchołka początkowego oraz poprzednik, czyli sąsiedni wierzchołek przez który prowadzi ta ścieżka. Do uporządkowania wierzchołków według długości ścieżki służy kolejka priorytetowa.

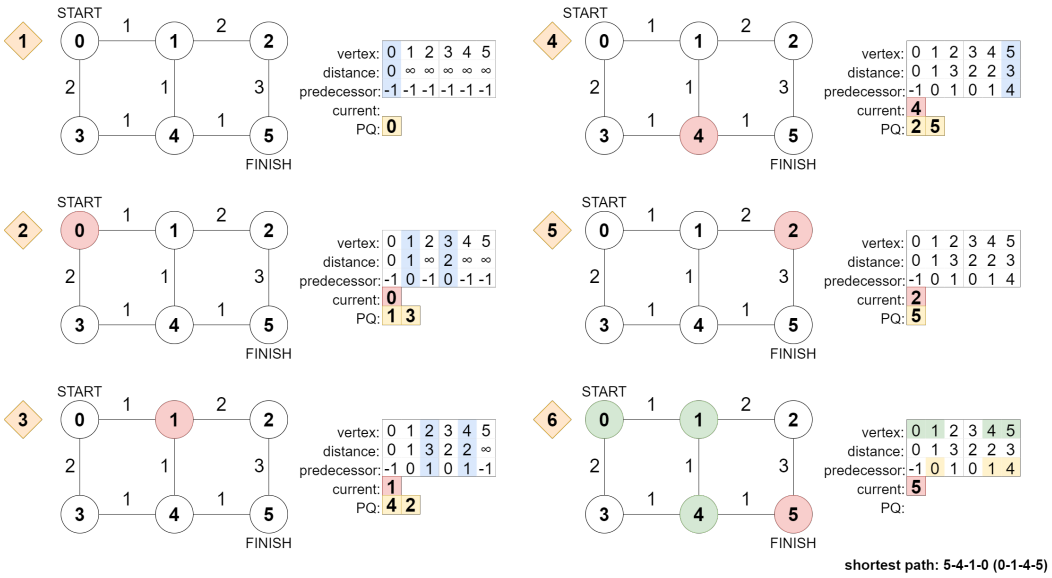
Najkrótszą ścieżkę od wybranego wierzchołka początkowego do wybranego wierzchołka końcowego można odczytać analizując poprzedników rozpoczynając od wierzchołka końcowego, następnie jego poprzednika, itd. aż do momentu dojścia do wierzchołka początkowego.

Lista kroków - algorytm Dijkstry:

1. Ustaw długość ścieżki dla każdego z wierzchołków grafu jako nieskończoność.
2. Ustaw poprzednika dla każdego z wierzchołków jako -1.
3. Ustaw długość ścieżki dla wierzchołka początkowego jako 0.
4. Dodaj wierzchołek początkowy do kolejki.
5. Dopóki kolejka nie jest pusta wykonuj kroki 6. - 11.
6. Wyjmij z kolejki wierzchołek `u`.

7. Dla każdego wierzchołka `v` sąsiedniego do `u` wykonaj krok 8.
8. Jeśli suma drogi do wierzchołka `u` i wagi krawędzi między wierzchołkami `u` i `v` jest mniejsza od obecnie zapamiętanej droga do wierzchołka `v`, to wykonaj kroki 9. - 11.
9. Zapisz nową drogę do wierzchołka `v` jako sumę drogi do wierzchołka `u` i wagi krawędzi między wierzchołkami `u` i `v`.
10. Zaktualizuj pozycję wierzchołka `v` w kolejce.
11. Zapisz wierzchołek `u` jako poprzednika wierzchołka `v`.

Przykład:



3 Kompilacja i uruchomienie

3.1 Argumenty wywołania

Program należy uruchomić z linii poleceń w trybie wsadowym. Zmiana parametrów analizowanego grafu oraz wybór funkcji następuje poprzez użycie flag.

Program akceptuje następujące argumenty wywołania:

- `--size` *`rows columns`* - rozmiar generowanego grafu;
  $rows$, $columns \in N$; $rows \cdot columns \leqslant 10^6$;
  domyślnie: $rows = 100$, $columns = 100$;
  typ danych: `int`;

- `--weight` *`w1 w2`* - zakres losowanych wag dla krawędzi;
  $w1$, $w2 \in R$; $w1$, $w2 \geqslant 0.0$; $w1$, $w2 \leqslant 100.0$; $w1 < w2$;
  domyślnie: $w1 = 0.0$, $w2 = 10.0$;
  typ danych: `double`;

- `--segments` *`n`* - liczba spójnych segmentów, na które jest podzielony graf;
  $n \in N$, $n \leqslant 10$;
  domyślnie: $n = 1$;
  typ: `int`;

- `--in` *`filename`* - plik wejściowy o odpowiednim formacie, z którego ma być odczytany graf;
  parametr wykluczający parametry: `--size, --weight` oraz `--segments`;
  domyślnie: wyłączone;

- `--out` *`filename`* - plik wyjściowy, do którego zostanie zapisany graf;
  domyślnie: `graph.output`;

- `--connectivity` - sprawdzenie spójności grafu;
  domyślnie: wyłączone;

- `--path` *`v1 v2`* - znalezienie najkrótszej ścieżki z wierzchołka $v1$ do $v2$;
  $v1$, $v2 \in N$; $v1$, $v2$ należą do grafu;
  domyślnie: wyłączone;
  typ danych: `int`;

- `--help` - wyświetlenie skróconej instrukcji z opisem sposobu uruchamiania;
  domyślnie: wyłączone;

Opcje kompilacji:

- `-DDEBUG` - wyświetlanie dodatkowych komunikatów w trakcie działania programu;
  domyślnie: wyłączone.

3.2 Makefile

Plik Makefile, który został umieszczony w repozytorium, ma 3 tryby:

- `default` - domyślna kompilacja całego programu do pliku `a.out`;
  `$ make`

- `debug` - kompilacja całego programu do pliku `a.out` z wyświetlaniem dodatkowych komunikatów w trakcie działania programu;
  `$ make debug`

- `clean` - usunięcie wszystkich plików obiektowych i pliku wynikowego `a.out`.
  `$ make clean`

4 Dane wejściowe

Program akceptuje dane wejściowe w formie pliku tekstowego o określonym formacie. Separatorem dziesiętnym jest kropka.

Opis grafu powinien zamierać następujące elementy:

- wymiary grafu (liczba wierszy i liczba kolumn);
- opis krawędzi wychodzących z poszczególnych wierzchołków (numery wierzchołków docelowych wraz z wagami krawędzi).

Ogólny format pliku:

```
#-rows #-columns
    description-of-the-0th-vertex
    description-of-the-1st-vertex
    ...
```

Format opisu każdego wierzchołka:

```
    vertex-number :edge-weight  vertex-number :edge-weight  ...
```

Fragment przykładowego pliku:

```
3 4
    1 :1.3  4 :5.2
    0 :1.3  2 :4.3  5 :8.1
    1 :4.3  3 :2.9  6 :4.9
    2 :2.9  7 :7.8
    0 :5.2  5 :3.6  8 :1.5
    1 :8.1  4 :3.6  6 :6.0  9 :3.9
    2 :4.9  5 :6.0  7 :2.7  10 :1.8
    3 :7.8  6 :2.7  11 :5.5
    4 :1.5  9 :4.7
    5 :3.9  8 :4.7  10 :9.9
    6 :1.8  9 :9.9  11 :1.7
    7 :5.5  10 :1.7
```

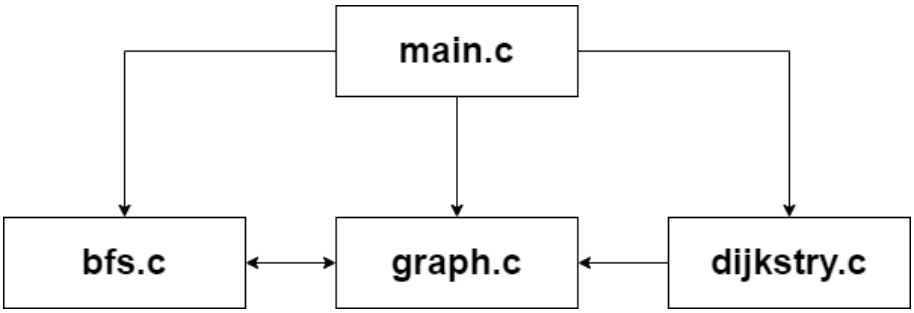
Graf reprezentowany przez ten plik jest przedstawiony na pierwszej stronie.

5 Dane wyjściowe

Program generuje plik wyjściowy o takim samym formacie, jak akceptowany format danych wejściowych. Umożliwia to ponowną analizę wygenerowanego grafu. Domyślnie generowany jest plik o nazwie `graph.output`.

6 Moduły

Diagram modułów:



6.1 main.c

Moduł `main.c` zawiera ogólne sterowanie programem oraz obsługę argumentów wywolania.

Funkcje w `main.c`:

- `int main(int argc, char **argv)` - funkcja główna programu, jej parametrami są parametry wprowadzane przy uruchamianiu programu;
- `void help()` - używana jest przy uruchomieniu programu z flagą `--help`, wyświetla instrukcję obsługi programu.

Struktury z innych modułów wykorzystywane w `main.c`:

- `graph.c`:
  - ```
    typedef struct graph
    {
        double **matrix;
        int rows;
        int columns;
    } * graph_t;
    ```

Funkcje z innych modułów wywoływane w `main.c`:

- `graph.c`:
  - `graph_t initialise_graph(int rows, int columns);`
  - `void generate_graph(graph_t g, double w1, double w2);`
  - `void split_graph(graph_t g, int segments);`
  - `int read_graph(graph_t g, FILE * in);`
  - `void write_graph(graph_t g, FILE * out);`
    ```

  - `void free_graph(graph_t g);`

- `bfs.c`:
  - `int check_connectivity(graph_t g);`

- `dijkstry.c`:
  - `int find_path(graph_t g, int start_vertex, int finish_vertex);`

6.2 graph.h/c

Moduł `graph.h` zawiera definicję struktury oraz deklaracje funkcji związanych z grafem. Moduł `graph.c` zawiera definicje tych funkcji.

Graf jest przechowywany w symetrycznej macierzy sąsiedztwa zaimplementowanej jako dwuwymiarowa tablica dynamiczna. Indeksy tablicy to numery wierzchołków, wartości w tablicy to wartości wag dla każdej krawędzi.

Przykład:
`g->matrix[0][1] = 3.55` znaczy, że krawędź pomiędzy wierzchołkiem 0 a 1 ma wagę 3.55 (wówczas również `g->matrix[1][0] = 3.55`)

Struktury w `graph.h/c`:

- ```
  typedef struct graph
  {
      double **matrix;
      int rows;
      int columns;
  } * graph_t;
  ```

Funkcje w `graph.h/c`:

- `graph_t initialise_graph(int rows, int columns)` - funkcja generująca graf o zadanych rozmiarach, bez krawędzi;
- `void generate_graph(graph_t g, double w1, double w2)` - funkcja generująca krawędzie grafu o wagach z zadanego zakresu;
- `void split_graph(graph_t g, int segments)` - funkcja dzieląca graf na zadaną liczbę segmentów, zawsze przecinane są krawędzie prawa i dolna;
- `int read_graph(graph_t g, FILE * in)` - funkcja czytająca graf z pliku;
- `void write_graph(graph_t g, FILE * out)` - funkcja zapisująca wygenerowany graf do pliku;
- `void free_graph(graph_t g)` - funkcja zwalniająca pamięć zaalokowaną dla grafu.
  ```

Struktury z innych modułów wykorzystywane w `graph.c`:

- `bfs.c`:
  - `typedef enum colour {white, gray, black} colour_t;`

Funkcje z innych modułów wywoływane w `graph.c`:

- bfs.c:
  - `colour_t bfs(graph_t g, int start_vertex);`

6.3 bfs.h/c

Moduł `bfs.h` zawiera definicje struktur oraz deklaracje funkcji związanych z algorytmem BFS. Moduł `bfs.c` zawiera definicje tych funkcji.

Kolejka FIFO zaimplementowana jest za pomocą listy jednokierunkowej.

Struktury w `bfs.h/c`:

- ```
  typedef struct fifo_element
  {
      int vertex;
      struct fifo_element *next;
  } fifo_element_t;
  ```
- ```
  typedef struct fifo
  {
      int count;
      fifo_element_t *front;
      fifo_element_t *back;
  } * fifo_t;
  ```
- `typedef enum colour {white, gray, black} colour_t;`

Funkcje w `bfs.h/c`:

- `fifo_t initialise_fifo()` - funkcja inicjalizująca kolejkę FIFO;
- `void push_fifo(fifo_t q, int vertex)` - funkcja dodająca wierzchołek do kolejki;
- `int pop_fifo(fifo_t q)` - funkcja usuwająca wierzchołek z kolejki, zwraca numer usuniętego wierzchołka;
- `int is_empty_fifo(fifo_t q);` - funkcja sprawdzająca, czy kolejka FIFO jest pusta;
- `void print_fifo(fifo_t q)` - funkcja wypisująca kolejkę FIFO;
- `void free_fifo(fifo_t q)` - funkcja zwalniająca pamięć zaalokowaną dla kolejki FIFO;

- `colour_t bfs(graph_t g, int start_vertex);` - funkcja implementująca algorytm BFS;
- `int check_connectivity(graph_t g)` - funkcja sprawdzająca spójność grafu na podstawie wyników działania algorytmu BFS, wywoływana przy uruchamianiu programu parametrem --connectivity.

Struktury z innych modułów wykorzystywane w `bfs.h/c`:

- `graph.c`:
  - ```
    typedef struct graph
    {
        double **matrix;
        int rows;
        int columns;
    } * graph_t;
    ```

6.4 dijkstry.h/c

Moduł `dijkstry.h` zawiera definicje struktur oraz deklaracje funkcji związanych z algorytmem Dijkstry. Moduł `dijkstry.c` zawiera definicje tych funkcji.

Kolejka priorytetowa zaimplementowana jest za pomocą kopca.

Struktury w `dijkstry.h/c`:

- ```
  typedef struct pq
  {
      int *heap;
      int count;
      double *distance;
      int *position;
  } * pq_t;
  ```
- ```
  struct path
  {
      int *predecessor;
      double *distance;
  } * path_t;
  ```

Funkcje w `dijkstry.h/c`:

- `pq_t initialise_pq(int size);` - funkcja inicjalizująca kolejkę priorytetową;
- `static void heap_up(pq_t q, int child);` - przesiewanie kopca w górę w celu dodania nowego elementu do kolejki priorytetowej;
- `static void heap_down(pq_t q);` - przesiewanie kopca w dół w celu usunięcia elementu z kolejki priorytetowej;
  ```

- `void push_pq(pq_t q, int vertex, double distance);` - funkcja dodająca wierzchołek do kolejki priorytetowej lub aktualizująca jego pozycję na podstawie aktualnej długości ścieżki;
- `int pop_pq(pq_t q);` - funkcja usuwająca wierzchołek z kolejki priorytetowej, zwraca numer usuniętego wierzchołka;
- `int is_empty_pq(pq_t q);` - funkcja sprawdzająca, czy kolejka priorytetowa jest pusta;
- `void print_pq(pq_t q);` - funkcja wypisująca kolejkę priotytetową;
- `void free_pq(pq_t q);` - funkcja zwalniająca pamięć zaalokowaną dla kolejki priorytetowej;
- `path_t dijkstry(graph_t g, int start_vertex)` - funkcja implementująca algorytm Dijkstry;
- `int find_path(graph_t g, int start_vertex, int finish_vertex)` - funkcja szukająca najkrótszej ścieżki między wierzchołkami na podstawie wyników działania algorytmu Dijkstry, wywoływana przy uruchamianiu programu z parametrem --path.

Struktury z innych modułów wykorzystywane w `dijkstry.h/c`:

- `graph.c`:
  -
```
typedef struct graph
{
    double **matrix;
    int rows;
    int columns;
} * graph_t;
```

7 Testy

Testy zostaną wykonane przez nas ręcznie, będziemy sprawdzać działanie funkcjonalności programu oraz zachowania niestandardowe. Dla każdego testu zostało podane planowane wywołanie oraz spodziewany wynik. Pliki wejściowe `connected_graph`, `disconnected_graph` oraz `error_graph` zostały umieszczone w repozytorium.

Planowane testy:

1. Generowanie grafu o zadanych rozmiarach i przedziale wag.
   `./a.out --size 5 6 --weight 1.3 4.8`
   plik `graph.output` z prawidłowo wygenerowanym grafem

2. Sprawdzanie spójności wygenerowanego grafu spójnego.
   `./a.out --segments 1 --connectivity`
   komunikat: `The graph is connected.`

3. Sprawdzanie spójności wygenerowanego grafu niespójnego.
   `./a.out --segments 2 --connectivity`
   komunikat: `The graph is disconnected.`

4. Sprawdzenie spójności wczytanego z pliku grafu spójnego.
   `./a.out --in connected_graph --connectivity`
   komunikat: `The graph is connected.`

5. Sprawdzenie spójności wczytanego z pliku grafu niespójnego.
   `./a.out --in disconnected_graph --connectivity`
   komunikat: `The graph is disconnected.`

6. Szukanie najkrótszej ścieżki w grafie.
   `./a.out --in connected_graph --path 0 6`
   komunikat: `The shortest path between vertices 0 and 6:  0-1-2-6. The total distance:  10.5.`

7. Szukanie najkrótszej ścieżki pomiędzy niespójnymi wierzchołkami.
   `./a.out --in disconnected_graph --path 0 6`
   komunikat: `The vertices 0 and 6 are disconnected.  There is no path between them.`

8. Podanie nieistniejącego parametru.
   `./a.out --sizee 4 5`
   komunikat: `a.out:  Error!  The flag "sizee" does not exist.  For further info please refer to the manual.`

9. Podanie formatu parametru nieakceptowanego przez program.
   `./a.out --size cztery 5.1`
   komunikat: `a.out:  Error!  The flag "size" does not accept the given format of arguments.  For further info please refer to the manual.`

10. Brak podania argumentów dla parametru.
    `./a.out --size`
    komunikat: `a.out: Error! The flag "size" requires arguments. For further info please refer to the manual.`

11. Podanie wartości argumentów wkraczających poza zakresy.
    `./a.out --size -3 -4`
    komunikat: `a.out: Error! The values of arguments for flag "size" are out of the allowed range. For further info please refer to the manual.`

12. Użycie nieprawidłowego separatora dziesiętnego.
    `./a.out --weight 1,1 5,9`
    komunikat: `a.out: Error! The only allowed decimal sepatator is ".". For further info please refer to the manual.`

13. Podanie parametrów wzajemnie się wykluczających.
    `./a.out --size 5 7 --weight 1.2 7.5 --segments 2 --in connected_graph`
    komunikat: `a.out: Error! The flags "size", "weight" and "segments" are mutually exclusive with the flag "in". For further info please refer to the manual.`

14. Wczytanie pliku wejściowego o nieodpowiednim formacie.
    `./a.out --in error_file`
    komunikat: `a.out: Error! Incorrect file format. For further info please refer to the manual.`

8 Zachowania niestandardowe

W przypadku wystąpienia nieprawidłowości program zachowa się w następujący sposób:

- podanie nieistniejącego parametru (np. w postaci literówki) - program wypisuje odpowiedni komunikat o błędzie, kontynuuje działanie ignorując błędny parametr;
  `./a.out --sizee 4 5`
  `a.out: Error! The flag "sizee" does not exist. For further info please refer to the manual.`

- podanie formatu parametru nieakceptowanego przez program - program wypisuje odpowiedni komunikat o błędzie, kontynuuje działanie przyjmując wartość domyślną danego parametru;
  `./a.out --size cztery 5.1`
  `a.out: Error! The flag "size" does not accept the given format of arguments. For further info please refer to the manual.`

- podanie wartości argumentów poza dozwolonym zakresem - program wypisuje odpowiedni komunikat, kontynuuje działanie przyjmując wartość domyślną danego parametru;
  ```
  ./a.out --size -3 -4
  a.out:  Error!  The values of arguments for flag "size" are out
  of the allowed range.  For further info please refer to the manual.
  ```

- użycie nieprawidłowego separatora dziesiętnego - program wypisuje odpowiedni komunikat, kontynuuje działanie przyjmując wartość domyślną danego parametru;
  ```
  ./a.out --weight 1,1 5,9
  a.out:  Error!  The only allowed decimal sepatator is ".".  For
  further info please refer to the manual.
  ```

- podanie parametrów wzajemnie wykluczających się - program wypisuje odpowiedni komunikat, kontynuuje działanie na podstawie danych z pliku wejściowego;
  ```
  ./a.out --size 5 7 --weight 1.2 7.5 --segments 2 --in mygraph
  a.out:  Error!  The flags "size", "weight" and "segments" are
  mutually exclusive with the flag "in".  For further info please
  refer to the manual.
  ```

- wczytanie pliku wejściowego o nieodpowiednim formacie - program wypisuje odpowiedni komunikat, kontynuuje działanie generując graf na podstawie wartości domyślnych;
  ```
  ./a.out --in error_file
  a.out:  Error!  Incorrect file format.  For further info please
  refer to the manual.
  ```

Komunikaty o błędach zostaną wypisane na `stderr`.