GridGraph - Dokumentacja

Skoczek Mateusz, Jędrzejewski Sebastian

12 kwietnia 2022

Streszczenie

Dokument zawiera specyfikację funkcjonalną i implementacyjną dotyczącą projektu *GridGraph* oraz opis testów programu.

Spis treści

Rozdział 1

Specyfikacja funkcjonalna

1.1 Cel projektu

Program `GridGraph` ma na celu wygenerowanie oraz zapis do pliku (lub na standardowe wyjście) grafu siatkowego o podanych paramentrach lub wczytanie grafu z pliku (lub ze standardowego wejścia) i sprawdzenie wybranych jego parametrów. Program działa w trybie wsadowym. Grafy są przedstawiane w plikach w postaci listy sąsiedztwa.

1.2 Opis funkcji

Program może działać w dwóch trybach: zapisu (`write`) i czytania (`read`).

W trybie zapisu program generuje graf o określonej przez użytkownika szerokości (ilości kolumn) (`width`), wysokości (ilości wierszy) (`height`), minimalnej (`edge_weight_min`) i maksymalnej (`edge_weight_max`) wagi krawędzi oraz minimalnej (`edge_count_min`) i maksymalnej (`edge_count_max`) ilości krawędzi wychodzących z jednego wierzchołka, a następnie zapisuje go w formie listy sąsiedztwa do pliku określonego przez użytkownika (lub wypisuje na standardowe wyjście).

Jeżeli graf zostanie pomyślnie zapisany do pliku (lub wypisany na standardowe wyjście), program zwróci `0`. W przeciwnym wypadku zostanie wyświetlony komunikat błędu, a program zwróci `1`.

W trybie czytania program wczytuje graf zapisany (w formie listy sąsiedztwa) w określonym przez użytkownika pliku (lub czyta ze standardowego wejścia), a następnie sprawdza określone przez użytkownika właściwości grafu:

- Spójność grafu (`connectivity`)
- Najkrótsza ścieżka z węzła A do innych węzłów (`shortest_path_a`) lub do określonego węzła B (`shortest_path_a` oraz `shortest_path_b`)

Jeżeli graf został wczytany oraz sprawdzony pomyślnie, zostanie wyświetlony wynik sprawdzania, a następnie program zwróci `0`. W przeciwnym wypadku zostanie wyświetlony komunikat błędu, a program zwróci `1`

Przykład (graf spójny, ścieżka istnieje):

```
Connectivity: connected
Shortest path from 0 to 10 (weight): 0-3-4-6-9-10 (0.778)
```

Przykład (graf niespójny, ścieżka nie istnieje):

```
Connectivity: disconnected
Shortest path from 0 to 10 (weight): path does not exist
```

1.3 Opis wywołania

Jeżeli nie zostanie wybrany tryb (tzn. nie zostanie przekazany argument `--write/-w` lub `--read/-r`) zostanie wyświetlona pomoc.

1.3.1 Tryb zapisu

Wywołanie: `./gridgraph --write/-w [argumenty]`

Argumenty:

- `--width/-xw` (Szerokość grafu - liczba kolumn)

  Typ: Liczba naturalna
  Zakres: > 0
  Wymagany: TAK

- `--height/-xh` (Wysokość grafu - liczba wierszy)

  Typ: Liczba naturalna
  Zakres: > 0
  Wymagany: TAK

- `--edge_weight_min/-Wmin` (Minimalna waga pojedyńczej krawędzi)

  Typ: Liczba rzeczywista
  Zakres: <0, `edge_weight_max`>
  Wymagany: NIE (domyślnie: 0)

- `--edge_weight_max/-Wmax` (Maksymalna waga pojedyńczej krawędzi)

  Typ: Liczba rzeczywista
  Zakres: > `edge_weight_min`
  Wymagany: NIE (domyślnie: 1)

- `--edge_count_min/-Cmin` (Minimalna liczba krawędzi wychodzących z jednego wierzchołka) [1]

  Typ: Liczba naturalna
  Zakres: <0, `edge_count_max`>
  Wymagany: NIE (domyślnie: 0)

- `--edge_count_max/-Cmax` (Maksymalna liczba krawędzi wychodzących z jednego wierzchołka)

  Typ: Liczba naturalna
  Zakres: <`edge_count_min`, 4>
  Wymagany: NIE (domyślnie: 4)

[1] Program będzie dążył do utworzenia co najmniej `edge_count_min` krawędzi, ale nie może tego zagwarantować. Nie jest możliwe wygenerowanie więcej niż 2 krawędzi dla wierzchołków w narożnikach oraz więcej niż 3 dla wierzchołków bocznych. Nie jest możliwe także utworzenie krawędzi, jeżeli wszystkie wierzchołki wokół osiągnęły już swoją nominalną (wylosowaną z podanego przedziału) liczbę krawędzi.

- `--seed/-s` (Ziarno generatora liczb losowych)

  Typ: Liczba całkowita

  Zakres: Zbiór liczb całkowitych

  Wymagany: NIE

- `--file/-f` (Plik w którym ma zostać zapisany graf)

  Typ: Ścieżka do pliku

  Zakres: -

  Wymagany: NIE (domyślnie: standardowe wyjście)

Przykład:

```
./gridgraph -w -xw 6 -xh 6 -Wmin 0.65 -Wmax 0.2 -Cmax 3 -f "/home/user/graph"
```

Powyższy przykład ilustruje wywołanie programu, który generuje graf o 6 kolumnach i 6 wierszach, z wagami krawędzi mieszczącymi się w przedziale od 0.2 do 0.65, gdzie minimalna ilość krawędzi wychodzących z wierzchołka to 0, a maksymalna ilość krawędzi to 3. Program zapisuje graf w odpowiednim formacie do pliku o nazwie `graph` znajdującego się w `/home/user`.

1.3.2 Tryb czytania

Wywołanie: `./gridgraph --read/-r [argumenty]`

Argumenty:

- `--shortest_path_a/-Sa` (Znajduje najkrótszą ścieżkę od wierzchołka A do pozostałych wierzchołków, używając algorytmu Dijkstry)

  Typ: Liczba naturalna

  Zakres: <0, ilość wierzchołków grafu>

  Wymagany: NIE [2] [3]

- `--shortest_path_b/-Sb` (Znajduje najkrótszą ścieżkę od wierzchołka A do wierzchołka B, używając algorytmu Dijkstry)

  Typ: Liczba naturalna

  Zakres: <0, *ilość wierzchołków grafu*> (nie licząc `shortest_path_a`)

  Wymagany: NIE[3]

- `--connectivity/-c` (Sprawdza czy graf jest spójny, używając algorytmu BFS)

  Typ: -

  Zakres: -

  Wymagany: NIE[3]

- `--file/-f` (Plik z którego ma zostać wczytany graf)

  Typ: Ścieżka do pliku

[2] Wymagane jeżeli `shortest_path_b` zostało podane

[3] Wymagany przynajmniej jeden

Zakres: -

Wymagany: NIE (domyślnie: standardowe wejście)

Przykład:

```
./gridgraph -r -c -Sa 0 -Sb 10 -f "/home/user/graph"
```

Powyższy przykład ilustruje wywołanie programu, który czyta plik ze strukturą grafu o nazwie `graph` znajdujący się w `/home/user`, a następnie sprawdza czy ten graf jest spójny oraz wyznacza najkrótszą ścieżkę pomiędzy węzłami numer 0 i 10.

1.4 Format danych wejściowych i wyjściowych

Dane wejściowe i wyjściowe przechowują graf w postaci listy sąsiedztwa. W pierwszej linijce znajdują się dwie liczby, które oznaczają odpowiednio liczbę kolumn i wierszy danego grafu. Każda następna linijka reprezentuje jeden wierzchołek, przy czym wierzchołki numerujemy od 0 od lewej do prawej. Zatem druga linijka w pliku zawiera numery wierzchołków, z którymi połączony jest wierzchołek numer 0, kolejna dotyczy wierzchołka numer 1 itd. Przy każdym numerze wierzchołka po dwukropku podana jest waga krawędzi pomiędzy tymi dwoma wierzchołkami.

Przykład:

```
2 2
    1 :0.54 2 :0.78
    0 :0.54 3 :0.12
    0 :0.78 3 :0.89
    1 :0.12 2 :0.89
```

Powyżej przedstawiona jest przykładowa zawartość pliku przechowującego graf. W pierwszej linijce można odczytać, że jest to graf o dwóch kolumnach i dwóch wierszach. W drugiej linijce przedstawiona jest informacja o tym, że wierzchołek numer 0 połączony jest z wierzchołkiem numer 1, a krawędź ta ma wagę 0.54. Istnieje również krawędź pomiędzy wierzchołkiem 0 a 2 o wadze 0.78. W trzeciej linijce znajdują się numery wierzchołków połączonych z wierzchołkiem numer 1 wraz z wagami itd.

1.5 Opis błędów

W przypadku błędu program wypisuje błąd na standardowy strumień błędów i zwraca 1. Komunikat błędów jest poprzecony słowem `ERROR` oraz nazwą trybu w nawiasie (np. `(Write mode)|`, jeżeli błąd dotyczy konkretnego trybu.

Poniżej przedstawione są komunikaty generowane przez program, gdy ten wykryje błąd, wraz z ich wyjaśnieniem:

WIDTH_NOT_A_NUMBER Został wybrany argument `width`, ale nie została podana wartość lub wartość nie jest liczbą (całkowitą).

HEIGHT_NOT_A_NUMBER Został wybrany argument `height`, ale nie została podana wartość lub wartość nie jest liczbą (całkowitą).

EDGE_WEIGHT_MIN_NOT_A_NUMBER Został wybrany argument `edge_weight_min`, ale nie została podana wartość lub wartość nie jest liczbą.

EDGE_WEIGHT_MAX_NOT_A_NUMBER Został wybrany argument `edge_weight_max`, ale nie została podana wartość lub wartość nie jest liczbą.

EDGE_COUNT_MIN_NOT_A_NUMBER Został wybrany argument `edge_count_min`, ale nie została podana wartość lub wartość nie jest liczbą (całkowitą).

EDGE_COUNT_MAX_NOT_A_NUMBER Został wybrany argument `edge_count_max`, ale nie została podana wartość lub wartość nie jest liczbą (całkowitą).

SEED_NOT_A_NUMBER Został wybrany argument `seed`, ale nie została podana wartość lub wartość nie jest liczbą (całkowitą).

WIDTH_LOWER_OR_EQUAL_TO_ZERO Wartość argumentu `width` jest mniejsza lub równa 0 (musi być większa od 0).

HEIGHT_LOWER_OR_EQUAL_TO_ZERO Wartość argumentu `height` jest mniejsza lub równa 0 (musi być większa od 0).

EDGE_WEIGHT_MIN_LOWER_THAN_ZERO Wartość argumentu `edge_weight_min` jest mniejsza od 0 (musi być większa lub równa 0 i mniejsza lub równa `edge_weight_max`).

EDGE_WEIGHT_MAX_GREATER_THAN_ONE Wartość argumentu `edge_weight_max` jest większa od 1 (musi być mniejsza lub równa 1 i większa lub równa `edge_weight_min`).

EDGE_WEIGHT_MIN_GREATER_THAN_EDGE_WEIGHT_MAX Wartość argumentu `edge_weight_min` jest większa od `edge_weight_max` (musi być większa lub równa 0 i mniejsza lub równa `edge_weight_max`).

EDGE_COUNT_MIN_LOWER_THAN_ZERO Wartość argumentu `edge_count_min` jest mniejsza od 0 (musi być większa lub równa 0 i mniejsza lub równa `edge_count_max`).

EDGE_COUNT_MAX_GREATER_THAN_FOUR Wartość argumentu `edge_count_max` jest większa od 4 (musi być mniejsza lub równa 4 i większa lub równa `edge_count_min`).

EDGE_COUNT_MIN_GREATER_THAN_EDGE_COUNT_MAX Wartość argumentu `edge_count_min` jest większa od `edge_count_max` (musi być większa lub równa 0 i mniejsza lub równa `edge_count_max`).

SHORTEST_PATH_A_NOT_POSITIVE_NUMBER Został wybrany argument `shortest_path_a`, ale nie została podana wartość lub wartość nie jest liczbą (nieujemną).

SHORTEST_PATH_B_NOT_POSITIVE_NUMBER Został wybrany argument `shortest_path_b`, ale nie została podana wartość lub wartość nie jest liczbą (nieujemną).

SHORTEST_PATH_B_WITHOUT_SHORTEST_PATH_A_SPECIFIED Został wybrany argument `shortest_path_a`, ale nie został wybrany argument `shortest_path_b`.

SHORTEST_PATH_B_EQUAL_TO_SHORTEST_PATH_A Argument `shortest_path_b` jest równy `shortest_path_a` (wartości argumentów muszą być różne od siebie).

CHECKING_OPTIONS_NOT_SPECIFIED Nie została wybrana przynajmniej jedna opcja sprawdzająca (przynajmniej jedna wymagana).

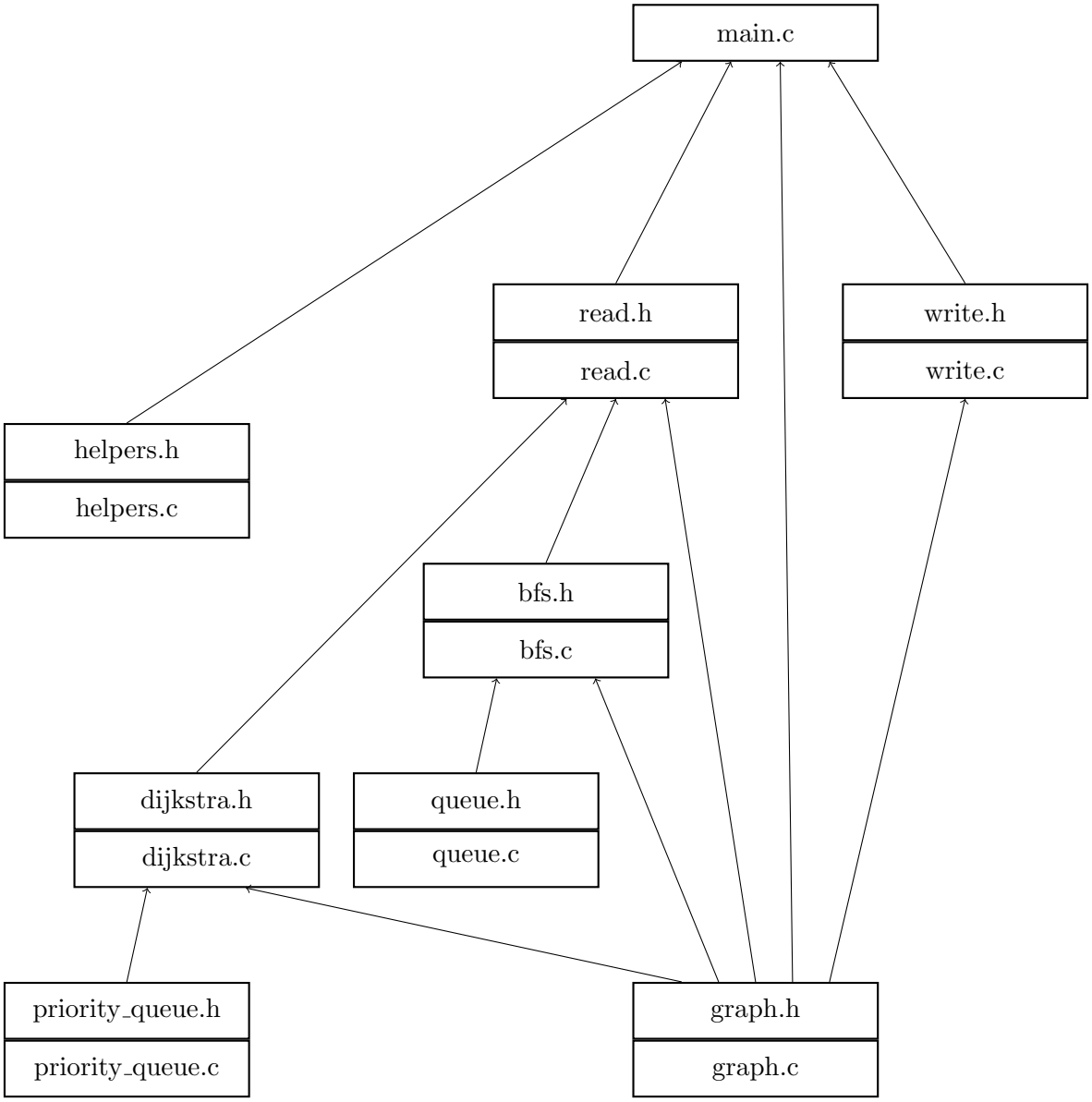
SHORTEST_PATH_A_GREATER_THAN_TOTAL_NUMBER_OF_VERTICES Wartość argumentu `shortest_path_a` jest większa niż całkowita liczba wierzchołków grafu.

SHORTEST_PATH_B_GREATER_THAN_TOTAL_NUMBER_OF_VERTICES Wartość argumentu `shortest_path_b` jest większa niż całkowita liczba wierzchołków grafu.

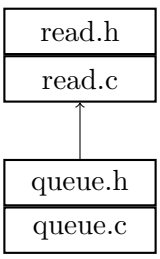
Rozdział 2

Specyfikacja implementacyjna

2.1 Diagram zależności plików źródłowych



Objaśnienie:



Plik `queue.h` jest dołączany do pliku `read.c` (oraz ewentualnie do `read.h`). To znaczy że w pliku `read.c` (oraz ewentualnie w `read.h`) znajduje się dyrektywa `#include "queue.h"`.

2.2 Plik main.c

Plik źródłowy (main.c) zawiera stałe `*_o` i `help` oraz definicje funkcji:

- `write_init`
- `read_init`
- `main`

2.2.1 Stałe `*_o`

`const char* *_o[2]` (np. `const char* file_o[2]`)

Stałe te przechowują nazwy argumentów wywołania, gdzie pierwszym elementem jest długa (normalna) nazwa argumentu, a drugim nazwa skrócona.

2.2.2 Stała `help`

`const char* help`

Stała ta przechowuje pomoc, zawierającą instrukcję wywołania i opisy wszystkich argumentów.

2.2.3 Funkcja `write_init`

`int write_init(int argc, char* argv[])`

Funkcja ta sprawdza argumenty wywołania dla trybu zapisu i ich wartości, a następnie wywołuje funkcję odpowiedzialną za tryb zapisu. Funkcja zwraca wartości `EXIT_SUCCESS` lub `EXIT_FAILURE`.

2.2.4 Funkcja `read_init`

`int read_init(int argc, char* argv[])`

Funkcja ta sprawdza argumenty wywołania dla trybu czytania i ich wartości, a następnie wywołuje funkcję odpowiedzialną za tryb czytania. Funkcja zwraca wartości `EXIT_SUCCESS` lub `EXIT_FAILURE`.

2.2.5 Funkcja `main`

`int main(int argc, char* argv[])`

Funkcja ta sprawdza, który tryb został wybrany, a następnie wywołuje funkcje `write_init` lub `read_init` (lub wyświetla pomoc jeżeli tryb nie został wybrany). Funkcja zwraca wartości `EXIT_SUCCESS` lub `EXIT_FAILURE`.

2.3 Pliki write.c i .h

Plik źródłowy (write.c) zawiera stałą `ec_drawing_weight` oraz definicje funkcji:

- `gen_graph`
- `write`*

Plik nagłówkowy (write.h) zawiera deklaracje zaznaczonych funkcji z pliku źródłowego (*).

2.3.1 Stała `ec_drawing_weight`

```
const int ec_drawing_weight[5]
```

Stała ta przechowuje informację o wagach losowania dla poszczególnych liczb krawędzi. Na przykład wartości $\{1, 2, 3, 4, 5\}$ oznaczają że wraz z liczbą krawędzi rośnie szansa na wylosowanie danej liczby krawędzi, gdyż tablica z której losowana będzie liczba krawędzi będzie wyglądać tak: $\{0, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4\}$.

2.3.2 Funkcja `gen_graph`

```
graph gen_graph(int width, int height, double edge_weight_min, double
edge_weight_max, int edge_count_min, int edge_count_max, int seed)
```

Funkcja ta odpowiada za wygenerowanie grafu o szerokości (ilości kolumn) `width`, wysokości (ilości wierzy) `height`, gdzie każdy wierzchołek będzie miał maksymalnie `edge_count_max` i (w miarę możliwości) minimalnie `edge_count_min` krawędzi [1], z których każda będzie miała wagę mieszczącą się w przedziale od `edge_weight_min` do `edge_weight_max` włącznie. Funkcja zwraca graf.

2.3.3 Funkcja `write`

```
int write(FILE* file, int width, int height, double edge_weight_min,
double edge_weight_max, int edge_count_min, int edge_count_max, int se-
ed)
```

Funkcja ta wywołuje funkcję `gen_graph` (przekazując jej jednocześnie wszystkie swoje parametry poza `file`), a następnie zapisuje uzyskany graf do pliku `file`. Funkcja zwraca wartości `EXIT_SUCCESS` lub `EXIT_FAILURE` (funkcja nie przewiduje wystąpienia błędu, więc w praktyce jest to tylko `EXIT_SUCCESS`).

[1] liczba krawędzi jest losowana; patrz "Stała `ec_drawing_weight`"

2.4 Pliki read.c i .h

Plik źródłowy (read.c) zawiera definicje funkcji:

- `path_fill`
- `path_display`
- `bfs_init`
- `dijkstra_init`
- `read_graph`
- `read*`

Plik nagłówkowy (read.h) zawiera deklaracje zaznaczonych funkcji z pliku źródłowego (*).

2.4.1 Funkcja `path_fill`

```
int path_fill(d_result result, int *nv, int from, int to)
```

Funkcja ta odpowiada za wypełnienie tablicy `nv` numerami wierzchołków, które tworzą ścieżkę od `from` do `to`, na podstawie tablicy poprzedników wygenerowanej przez algorytm Dijkstry i przechowywanej w strukturze `result`. Funkcja zwraca ilość dodanych wierzchołków do tablicy.

2.4.2 Funkcja `path_display`

```
void path_display(int *nv, int l)
```

Funkcja ta odpowiada za wyświetlenie ścieżki pomiędzy dwoma wierzchołkami przechowywanej w tablicy `nv` znając jej długość `l`.

2.4.3 Funkcja `bfs_init`

```
void bfs_init(graph g)
```

Funkcja ta odpowiada za wywołanie algorytmu BFS i wyświetlenie komunikatu o spójności grafu `g`.

2.4.4 Funkcja `dijkstra_init`

```
void dijkstra_init(graph g, int vertex_a, int vertex_b)
```

Funkcja ta odpowiada za wywołanie algorytmu Dijkstry i wyświetlenie najkrótszej ścieżki pomiędzy wierzchołkami `vertex_a` i `vertex_b` lub najkrótszych ścieżek pomiędzy wierzchołkiem `vertex_a` i pozostałymi wierzchołkami grafu `g`.

2.4.5 Funkcja `read_graph`

```
graph read_graph(FILE *f)
```

Funkcja ta odpowiada za odczytanie grafu z pliku `f`, tworzenie struktury grafu i uzupełnienie jej przeczytanymi wierzchołkami i wagami z pliku. Funkcja zwraca stworzoną strukturę grafu.

2.4.6 Funkcja `read`

```
int read(FILE *file, int connectivity, int vertex_a, int vertex_b)
```

Funkcja odpowiada za zarządzanie trybem read. Przyjmuje wskaźnik na plik (bądź stdin), z którego ma być czytany graf, a także czy ma być sprawdzona jego spójność (`connectivity` przyjmuje wartość 1/0) oraz numery wierzchołków, między którymi ma być wyznaczona ścieżka (gdy `vertex_b` jest równy -1, wyznaczana jest ścieżka pomiędzy `vertex_a` i wszystkimi wierzchołkami).

2.5 Pliki dijkstra.c i .h

Plik źródłowy (dijkstra.c) zawiera definicje funkcji:

- `result_init`*
- `result_free`*
- `dijkstra`*

Plik nagłówkowy (dijkstra.h) poza deklaracjami zaznaczonych funkcji z pliku źródłowego (*) zawiera także deklarację struktury wyniku generowanego przez algorytm Dijkstry (`d_result`).

2.5.1 Struktura `d_result`

```
typedef struct r
{
   double *d;
   int *p;
} *d_result;
```

Struktura przechowuje dwie tablice: `d` - zawierającą całkowitą wagę dojścia do danego wierzchołka z wierzchołka początkowego oraz `p` – zawierającą numery poprzedników.

2.5.2 Funkcja `result_init`

```
d_result result_init(int n, int vertex_a, int *visited)
```

Funkcja ta odpowiada za zainicjalizowanie struktury typu `d_result` i wypełnienie jej tablic w taki sposób jaki wymaga tego algorytm Dijkstry (tablica `d` wypełniona nieskończonościami oprócz elementu o indeksie wierzchołka początkowego; tablica `p` wypełniona wartościami -1). Funkcja zwraca utworzoną strukturę.

2.5.3 Funkcja `result_free`

```
void result_free(d_result result)
```

Funkcja ta odpowiada za zwolnienie pamięci zarezerwowanej dla struktury `result`.

2.5.4 Funkcja `dijkstra`

```
d_result dijkstra(graph graph, int vertex_a)
```

Funkcja ta jest implementacją algorytmu Dijkstry wykorzystującego kolejkę priorytetową do uzupełnienia tablic struktury typu `d_result`. Funkcja jako argument przyjmuje `vertex_a` (numer wierzchołka startowego) oraz `vertex_b`. Jeżeli szukana jest ścieżka wyłącznie do jednego wierzchołka, algorytm przerywa działanie w momencie dojścia do wierzchołka docelowego (`vertex_b`). Funkcja zwraca utworzoną strukturę.

2.6 Pliki bfs.c i .h

Plik źródłowy (bfs.c) zawiera definicje funkcji:

- `bfs*`

Plik nagłówkowy (bfs.h) zawiera deklaracje zaznaczonych funkcji z pliku źródłowego (*).

2.6.1 Funkcja `bfs`

```
int* bfs(graph graph, int vertex)
```

Funkcja ta jest implementacją algorytmu BFS (Breadth-first search) w uproszczonej wersji. Sprawdza czy w grafie `graph` od wierzchołka `vertex` istnieją ścieżki do wszystkich pozostałych wierzchołków. Funkcja zwraca tablicę liczb 0/1 (prawda/fałsz) o długości [ilość wierzchołków grafu], w której index oznacza numer wierzchołka a wartość wynik sprawdzenia dla wierzchołka o danym indeksie.

2.7 Pliki graph.c i .h

Plik źródłowy (graph.c) zawiera definicje funkcji:

- `graph_init`*
- `graph_free`*
- `edge_list_add`*
- `edge_list_contains_vertex`*
- `edge_list_length`*

Plik nagłówkowy (graph.h) poza deklaracjami zaznaczonych funkcji z pliku źródłowego (*) zawiera także deklarację struktury grafu w formie listy sąsiedztwa (`graph`) oraz deklarację struktury listy wierzchołków (`edge_list`).

2.7.1 Struktura `graph`

```
typedef struct g
{
   int width;
   int height;
   edge_list **list;
} *graph;
```

Struktura przechowuje szerokość grafu (liczba kolumn) (`width`), wysokość grafu (liczba wierszy) (`height`) oraz tablicę list połączonych wierzchołków dla każdego wierzchołka (lista sąsiedztwa) (`list`).

2.7.2 Struktura `edge_list`

```
typedef struct e
{
   int vertex;
   double weight;
   struct e *next;
} edge_list;
```

Struktura przechowuje numer połączonego wierzchołka (`vertex`), wagę krawędzi (`weight`) oraz wskaźnik na następny element (`next`) (lista jednokierunkowa).

2.7.3 Funkcja `graph_init`

```
graph graph_init(int w, int h)
```

Funkcja ta odpowiada za zainicjalizowanie grafu o szerokości (ilości kolumn) `w` i wysokości (ilości wierszy) `h`. Funkcja zwraca graf.

2.7.4 Funkcja `graph_free`

```
void graph_free(graph g)
```

Funkcja ta odpowiada za zwolnienie pamięci zarezerwowanej dla grafu `g`.

2.7.5 Funkcja `edge_list_add`

```
edge_list* edge_list_add(edge_list *l, int v, double wt)
```

Funkcja ta odpowiada za dodanie na koniec listy połączonych wierzchołów `l` nowego wierzchołka o numerze `v` i wadze krawędzi `wt`. Funkcja zwraca wskaźnik na listę połączonych wierzchołków.

2.7.6 Funkcja `edge_list_contains_vertex`

```
int edge_list_contains_vertex(edge_list* l, int v)
```

Funkcja ta odpowiada za sprawdzenie czy na liście połączonych wierzchołków `l` znajduje się wierzchołek o numerze `v`. Funkcja zwraca wartości 1/0 (prawda/fałsz).

2.7.7 Funkcja `edge_list_length`

```
int edge_list_length(edge_list* l)
```

Funkcja ta odpowiada za sprawdzenie długości listy połączonych wierzchołków `l`. Funkcja zwraca długość listy.

2.8 Pliki queue.c i .h

Plik źródłowy (queue.c) zawiera definicje funkcji:

- `queue_enqueue`*
- `queue_dequeue`*

Plik nagłówkowy (queue.h) poza deklaracjami zaznaczonych funkcji z pliku źródłowego (*) zawiera także deklarację struktury kolejki FIFO (First in, first out) (`queue`).

2.8.1 Struktura queue

```
typedef struct q
{
   int value;
   struct q* next;
} queue;
```

Struktura przechowuje wartość pojedyńczego elementu (`value`) oraz wskaźnik na następny element (`next`) (lista jednokierunkowa).

2.8.2 Funkcja queue_enqueue

```
void queue_enqueue(queue** q, int value)
```

Funkcja ta odpowiada za dodanie na początku kolejki `q` (listy) nowego elementu o wartości `value`.

2.8.3 Funkcja queue_enqueue

```
int queue_dequeue(queue** q)
```

Funkcja ta odpowiada za pobranie wartości elementu z końca kolejki `q` (listy), usunięcie ostatniego elementu z kolejki, a następnie zwrócenie pobranej wartości.

2.9 Pliki priority_queue.c i .h

Plik źródłowy (priority_queue.c) zawiera definicje funkcji:

- `pq_init`*
- `pq_is_empty`*
- `heap_up`*
- `pq_push`*
- `heap_down`
- `pq_pop`*
- `pq_free`*

Plik nagłówkowy (priority_queue.h) poza deklaracjami zaznaczonych funkcji z pliku źródłowego (*) zawiera także deklarację struktury kolejki priorytetowej (`pq`).

2.9.1 Struktura `pq`

```
typedef struct p
{
   int *q;
   int *pn;
   int n;
   int size;
} *pq;
```

Struktura przechowuje tablicę numerów wszystkich wierzchołków (`q`), gdzie priorytetem jest najmniejsza odległość od wierzchołka źródłowego. Tablica `pn` przechowuje pozycję i-tego wierzchołka w kolejce, `n` jest liczbą pozostałych wierzchołków, a `size` maksymalnym rozmiarem kolejki równym liczbie wszystkich wierzchołków w grafie.

2.9.2 Funkcja `pq_init`

```
pq pq_init(int size)
```

Funkcja ta odpowiada za alokowanie miejsca na strukturę `pq` i jej pola. Funkcja zwraca tą strukturę.

2.9.3 Funkcja `pq_is_empty`

```
int pq_is_empty(pq p_queue)
```

Funkcja sprawdza czy kolejka jest pusta. Funkcja zwraca wartości 1/0 (prawda/fałsz).

2.9.4 Funkcja `heap_up`

```
void heap_up(pq p_queue, double *d, int from)
```

Funkcja ta odpowiada za implementację przesiewania kopca w górę. Wykorzystywana do tego jest tablica `d`, która przechowuje odległości wierzchołków od wierzchołka źródłowego, które są priorytetem. Funkcja przyjmuje także argument `from`, który jest indeksem kopca, od którego należy rozpocząć przesiewanie (przydatne w algorytmie Dijkstry).

2.9.5 Funkcja **pq_push**

```
int pq_push(pq p_queue, double *d, int v)
```

Funkcja ta odpowiada za dodanie do kolejki priorytetowej numeru wierzchołka `v` i wywołanie funkcji `heap_up`. Funkcja ta jest używana tylko podczas testów i nie wykorzystuje jej algorytm Dijkstry.

2.9.6 Funkcja **heap_down**

```
static void heap_down(pq p_queue, double *d)
```

Funkcja ta odpowiada za implementację przesiewania kopca w dół. Wykorzystywana do tego jest tablica `d`, która przechowuje odległości wierzchołków od wierzchołka źródłowego, które są priorytetem.

2.9.7 Funkcja **pq_pop**

```
int pq_pop(pq p_queue, double *d)
```

Funkcja ta odpowiada za usunięcie z kolejki priorytetowej wierzchołka znajdującego się w korzeniu i wywołanie funkcji `heap_down`. Funkcja zwraca numer zdjętego wierzchołka.

2.9.8 Funkcja **pq_free**

```
void pq_free(pq p_queue)
```

Funkcja ta odpowiada za zwolnienie pamięci zarezerwowanej dla kolejki `p_queue`

2.10 Pliki helpers.c i .h

Plik źródłowy (helpers.c) zawiera definicje funkcji:

- `str_arr_get_index`*
- `str_is_int`*
- `str_is_double`*

Plik nagłówkowy (helpers.h) zawiera deklaracje zaznaczonych funkcji z pliku źródłowego (*).

2.10.1 Funkcja **`str_arr_get_index`**

```
int str_arr_get_index(char* element, const char* array[], int length)
```

Funkcja ta odpowiada za znalezienie w tablicy napisów `array` o długości `length` konkretnego napisu `element`. Funkcja zwraca indeks tego napisu w tablicy.

2.10.2 Funkcja **`str_is_int`**

```
int str_is_int(char* string)
```

Funkcja ta odpowiada za sprawdzenie czy napis `string` jest liczbą całkowitą (tzn. napis zawiera tylko cyfry i ewentualnie znak minus na początku). Funkcja zwraca wartości 1/0 (prawda/fałsz).

2.10.3 Funkcja **`str_is_double`**

```
int str_is_double(char* string)
```

Funkcja ta odpowiada za sprawdzenie czy napis `string` jest liczbą rzeczywistą (tzn. napis zawiera tylko cyfry oraz ewentualnie jedną kropkę i znak minus na początku). Funkcja zwraca wartości 1/0 (prawda/fałsz).

2.11 Plik makefile

Plik makefile przechowuje instrukcje kompilacji programu oraz instrukcję czyszczenia folderu (`clean`) i instrukcję testu (`test`).

2.11.1 Kompilacja (`make`)

Instrukcja kompilacji składa się z czterech etapów:

1. wygenerowania pliku zawierającego zależności (`cc -MM [pliki źródłowe] > [plik zaw. zależności]`)
2. wczytania go (`-include [plik zaw. zależności]`)
3. skompilowania programu (`cc -o gridgraph [pliki .o]`)
4. czyszczenia pokompilacyjnego (`rm [pliki .o] [plik zaw. zależności]`)

2.11.2 Czyszczenie (`make clean`)

Instrukcja czyszczenia usuwa pliki (wywoływana jest komenda `rm [pliki]`):

- plik programu
- pliki .o
- plik zawierający zależności

2.11.3 Test (`make test`)

Instrukcja testu w pierwszej kolejności wywołuje instrukcję kompilacji, a następnie usuwa folder zawierający wyniki testów (domyślnie: "./test_results"), tworzy go i na koniec wywołuje kolejne testy opisane w rozdziale "Testy".

Rozdział 3

Testy

3.1 Generowanie małego grafu 6x6 z domyślnymi parametrami

Test polega na wygenerowaniu grafu o rozmiarach 6 na 6 z domyślnymi pozostałymi parametrami i zapisaniu go do pliku "test1" w folderze zawierającym wyniki testów.

Plik powinien zawierać 27 linii. W pierwszej powinna być zapisana długość i wysokość grafu (`5 5`), a w następnych 25 lista sąsiedztwa, gdzie dla każdego wierzchołka powinno być od 0 do 4 krawędzi, z których każda powinna mieć wagę mieszczącą się w przedziale od 0 do 1. Ostatnia linia powinna być pusta.

Wywoływane komendy:

```
./gridgraph -w -xw 5 -xh 5 -f [folder zaw. wyn. testów]/test1
```

Przykładowy wynik testu zakończonego powodzeniem:

Plik "test1":

```
5 5
    5 :0.141989 1 :0.588184
    0 :0.588184 2 :0.374213 6 :0.753903
    1 :0.374213 7 :0.012313 3 :0.115006
[... pominięto 19 linii ...]
    21 :0.775892 23 :0.844664
    18 :0.961897 22 :0.844664 24 :0.048494
    19 :0.151187 23 :0.048494
```

3.2 Generowanie dużego grafu 1000x1000 z domyślnymi parametrami

Test polega na wygenerowaniu grafu o rozmiarach 1000 na 1000 z domyślnymi pozostałymi parametrami i zapisaniu go do pliku "test2" w folderze zawierającym wyniki testów.

Plik powinien zawierać 1000002 linie. W pierwszej powinna być zapisana długość i wysokość grafu (`1000 1000`), a w następnych 1000000 lista sąsiedztwa, gdzie dla każdego wierzchołka powinno być od 0 do 4 krawędzi, z których każda powinna mieć wagę mieszczącą się w przedziale od 0 do 1. Ostatnia linia powinna być pusta.

Wywoływane komendy:

```
./gridgraph -w -xw 1000 -xh 1000 -f [folder zaw. wyn. testów]/test2
```

Przykładowy wynik testu zakończonego powodzeniem:

Plik "test2":

```
1000 1000
    1000 :0.337479 1 :0.003876
    0 :0.003876 2 :0.098184 1001 :0.338146
    1 :0.098184 1002 :0.693836 3 :0.853243
[... pominięto 999994 linie ...]
    999996 :0.167250 999998 :0.136315
    998998 :0.988869 999997 :0.136315 999999 :0.805686
    998999 :0.235517 999998 :0.805686
```

3.3 Generowanie grafu 50x50 ze stałym ziarnem generatora liczb losowych

Test polega na wygenerowaniu dwóch grafów o rozmiarach 50 na 50, ziarnem generatora liczb losowych równym 0 i z domyślnymi pozostałymi parametrami i zapisaniu ich odpowiednio do plików "test3" i "test4", a następnie sprawdzeniu ich identyczności oraz identyczności z plikiem zawierającym oczekiwany graf.

Oba pliki powinny zawierać 2502 linie każdy. W pierwszej powinna być zapisana długość i wysokość grafu (`50 50`), a w następnych 2500 lista sąsiedztwa, gdzie dla każdego wierzchołka powinno być od 0 do 4 krawędzi, z których każda powinna mieć wagę mieszczącą się w przedziale od 0 do 1. Ostatnia linia powinna być pusta.

Oba pliki powinny być identyczne. Co więcej pliki powinny być identyczne jak te co w poprzednim wywołaniu testu. Identyczność plików jest sprawdzana programem `cmp` (najpierw identyczność obu plików, a następnie identyczność z plikiem "expected_test3" zawierający spodziewany dla tego testu graf). Po każdym wywołaniu programu `cmp` wypisywany jest kod zakończenia. Wszystkie powinny być równe 0.

Wywoływane komendy:

```
./gridgraph -w -xw 50 -xh 50 -s 0 -f [folder zaw. wyn. testów]/test3
./gridgraph -w -xw 50 -xh 50 -s 0 -f [folder zaw. wyn. testów]/test4
cmp [folder zaw. wyn. testów]/test3 [folder zaw. wyn. testów]/test4
echo $?
cmp [folder zaw. wyn. testów]/test3 ./Tests/expected_test3
echo $?
```

Przykładowy wynik testu zakończonego powodzeniem:

Pliki "test3" i "test4" [1] [2]:

```
50 50
    1 :0.743163 50 :0.092576
    0 :0.743163 51 :0.482107
    3 :0.329285 52 :0.311602
[... pominięto 2494 linie ...]
    2447 :0.383774 2496 :0.584013
    2448 :0.722728 2499 :0.981057
    2449 :0.113450 2498 :0.981057
```

[1] Cała zawartość w pliku "Tests/expected_text3"

[2] Pliki powinny być identyczne jak przedstawiony przykład

3.4 Generowanie grafu 50x50 na pewno niespójnego (granica liczby krawędzi od 0 do 1) i sprawdzenie spójności

Test polega na wygenerowaniu grafu który na pewno będzie niespójny, o rozmiarach 50 na 50, granicą liczby krawędzi od 0 do 1 i z domyślnymi pozostałymi parametrami i zapisaniu go do pliku "test5" w folderze zawierającym wyniki testów, a następnie na sprawdzeniu jego spójności, zapisaniu wyniku sprawdzenia w pliku "test6" i porównaniu wyniku sprawdzenia z oczekiwanym.

Plik "test5" powinien zawierać 2502 linie. W pierwszej powinna być zapisana długość i wysokość grafu (`50 50`), a w następnych 2500 lista sąsiedztwa, gdzie dla każdego wierzchołka powinno być od 0 do 1 krawędzi, z których każda powinna mieć wagę mieszczącą się w przedziale od 0 do 1. Ostatnia linia powinna być pusta. Plik "test6" powinien zawierać 3 linie, z których w pierwszej powinien być wypisany wynik testu spójności, a dwie pozostałe powinny być puste.

Identyczność pliku zawierającego wynik sprawdzenia ("test6") z plikiem zawierającym oczekiwany wynik sprawdzenia ("expected_test6") jest sprawdzana programem `cmp`. Po wywołaniu programu `cmp` wypisywany jest kod zakończenia, ktry powinien by równy 0.

Wywoływane komendy:

```
./gridgraph -w -xw 50 -xh 50 -Cmax 1 -f [folder zaw. wyn. testów]/test5
./gridgraph -r -c -f [folder zaw. wyn. testów]/test5 > [folder zaw.
wyn. testów]/test6
cmp [folder zaw. wyn. testów]/test6 ../Tests/expected_test6
echo $?
```

Przykładowy wynik testu zakończonego powodzeniem:

Plik "test5":

```
50 50
    1 :0.938031
    0 :0.938031
    52 :0.782637
[... pominięto 2494 linie ...]


    2449 :0.331057
```

Plik "test6" [3]:

```
Connectivity: disconnected
```

[3]Plik powinien być identyczny jak przedstawiony przykład

3.5 Generowanie grafu 50x50 na pewno spójnego (granica liczby krawędzi od 4 do 4) i sprawdzenie spójności

Test polega na wygenerowaniu grafu który na pewno będzie spójny, o rozmiarach 50 na 50, granicą liczby krawędzi od 4 do 4 i z domyślnymi pozostałymi parametrami i zapisaniu go do pliku "test7" w folderze zawierającym wyniki testów, a następnie na sprawdzeniu jego spójności, zapisaniu wyniku sprawdzenia w pliku "test8" i porównaniu wyniku sprawdzenia z oczekiwanym.

Plik "test7" powinien zawierać 2502 linie. W pierwszej powinna być zapisana długość i wysokość grafu (`50 50`), a w następnych 2500 lista sąsiedztwa, gdzie każdy wierzchołek powinien mieć 4 krawędzie[4], z których każda powinna mieć wagę mieszczącą się w przedziale od 0 do 1. Ostatnia linia powinna być pusta. Plik "test8" powinien zawierać 3 linie, z których w pierwszej powinien być wypisany wynik testu spójności, a dwie pozostałe powinny być puste.

Identyczność pliku zawierającego wynik sprawdzenia ("test8") z plikiem zawierającym oczekiwany wynik sprawdzenia ("expected_test8") jest sprawdzana programem `cmp`. Po wywołaniu programu `cmp` wypisywany jest kod zakończenia, ktry powinien by równy 0.

Wywoływane komendy:

```
./gridgraph -w -xw 50 -xh 50 -Cmin 4 -f [folder zaw. wyn. testów]/test7
./gridgraph -r -c -f [folder zaw. wyn. testów]/test7 > [folder zaw.
wyn. testów]/test8
cmp [folder zaw. wyn. testów]/test8 ../Tests/expected_test8
echo $?
```

Przykładowy wynik testu zakończonego powodzeniem:

Plik "test7":

```
50 50
    1 :0.719183 50 :0.750156
    0 :0.719183 51 :0.448654 2 :0.136224
    1 :0.136224 3 :0.087748 52 :0.258144
[... pominięto 2494 linie ...]
    2447 :0.796042 2496 :0.107598 2498 :0.043867
    2448 :0.404981 2497 :0.043867 2499 :0.831039
    2449 :0.912295 2498 :0.831039
```

Plik "test8" [5]:

```
Connectivity: connected
```

[4] Jeżeli to możliwe

[5] Plik powinien być identyczny jak przedstawiony przykład

3.6 Generowanie grafu 50x50 o wagach krawędzi od 0.3 do 0.7

Test polega na wygenerowaniu grafu o rozmiarach 50 na 50, o wagach krawędzi mieszczących się w granicach od 0.3 do 0.7 i z domyślnymi pozostałymi parametrami i zapisaniu go do pliku "test9" w folderze zawierającym wyniki testów.

Plik powinien zawierać 2502 linie. W pierwszej powinna być zapisana długość i wysokość grafu (`50 50`), a w następnych 2500 lista sąsiedztwa, gdzie dla każdego wierzchołka powinno być od 0 do 4 krawędzi, z których każda powinna mieć wagę mieszczącą się w przedziale od 0.3 do 0.7. Ostatnia linia powinna być pusta.

Wywoływane komendy:

```
./gridgraph -w -xw 50 -xh 50 -Wmin 0.3 -Wmax 0.7 -f [folder zaw. wyn. testów]/test9
```

Przykładowy wynik testu zakończonego powodzeniem:

Plik "test9":

```
50 50
    1 :0.587673 50 :0.600063
    0 :0.587673 51 :0.479462 2 :0.354490
    1 :0.354490 3 :0.335099
[... pominięto 2494 linie ...]
    2447 :0.401067 2496 :0.697442 2498 :0.340867
    2448 :0.496822 2497 :0.340867
    2449 :0.425698
```

3.7 Generowanie grafu 100x100, o stałym ziarnie generatora liczb losowych, granicy liczby krawędzi od 1 do 3 i wagach krawędzi od 0.4 do 0.9 oraz sprawdzenie najkrótszych ścieżek od wierzchołka 0 do pozostałych wierzchołków

Test polega na wygenerowaniu grafu o rozmiarach 100 na 100, ziarnem generatora liczb losowych równym 0, granicą liczby krawędzi od 1 do 3 i wagach krawędzi od 0.4 do 0.9 i zapisaniu go do pliku "test10" w folderze zawierającym wyniki testów, a następnie sprawdzeniu jego z plikiem zawierającym oczekiwany graf. W ramach testu sprawdzane są także najkrótsze ścieżki od wierzchołka 0 do pozostałych wierzchołków. Wynik sprawdzenia zapisywany jest w pliku "test11", a następnie porównanywany z oczekiwanym.

Plik "test10" powinien zawierać 10002 linie. W pierwszej powinna być zapisana długość i wysokość grafu (`100 100`), a w następnych 10000 lista sąsiedztwa, gdzie dla każdego wierzchołka powinno być od 1 [6] do 3 krawędzi, z których każda powinna mieć wagę mieszczącą się w przedziale od 0.4 do 0.9. Ostatnia linia powinna być pusta. Plik "test11" powinien zawierać 10003 linie, z których w dwóch pierwszych znajduje się nagłówek testu, a w następnych 10000, ścieżki do określonych wierzchołków (wraz z wagami). Ostatnia linia powinna być pusta.

Identyczność plików jest sprawdzana programem `cmp` (najpierw identyczność pliku zawierającego graf ("test10") z plikiem "expected_test10", a następnie identyczność pliku zawierającego wynik sprawdzenia ("test11") z plikiem "expected_test11"). Po każdym wywołaniu programu `cmp` wypisywany jest kod zakończenia. Wszystkie powinny być równe 0.

Wywoływane komendy:

```
./gridgraph -w -xw 100 -xh 100 -Cmin 1 -Cmax 3 -Wmin 0.4 -Wmax 0.9 -
s 0 -f [folder zaw. wyn. testów]/test10
cmp [folder zaw. wyn. testów]/test10 ../Tests/expected_test10
echo $?
./gridgraph -r -sA 0 -f [folder zaw. wyn. testów]/test10 > [folder zaw.
wyn. testów]/test11
cmp [folder zaw. wyn. testów]/test11 ../Tests/expected_test11
echo $?
```

Przykładowy wynik testu zakończonego powodzeniem:

Plik "test10" [7] [8]:

```
100 100
    100 :0.828420
    2 :0.577575 101 :0.736913
    1 :0.577575 3 :0.787888 102 :0.684638
[... pominięto 9994 linie ...]
    9996 :0.871117 9998 :0.714386
    9898 :0.608558 9997 :0.714386
    9899 :0.818561
```

[6] Jeżeli to możliwe

[7] Cała zawartość w pliku "Tests/expected_text10"

[8] Plik powinnien być identyczny jak przedstawiony przykład

Plik "test11" [9] [10]:

```
Shortest paths from 0 to all vertices and their weights:
Path to 0 (weight): 0 (0)
Path to 1 (weight): 0 - 100 - 200 - 201 - 202 - 102 - 2 - 1 (5.07255)
Path to 2 (weight): 0 - 100 - 200 - 201 - 202 - 102 - 2 (4.49497)
[... pominięto 9994 linie ...]
Path to 9997 (weight): 0 - 100 - [... pominięto 1788 znaków ...] - 9998
- 9997 (166.437)
Path to 9998 (weight): 0 - 100 - [... pominięto 1781 znaków ...] - 9898
- 9998 (165.863)
Path to 9999 (weight): 0 - 100 - [... pominięto 1788 znaków ...] - 9899
- 9999 (166.64)
```

[9] Cała zawartość w pliku "Tests/expected_text11"

[10] Plik powinnien być identyczny jak przedstawiony przykład

3.8 Sprawdzenie poprawności działania kolejki priorytetowej

Test polega na posortowaniu 1000 liczb wylosowanych z zakresu od 0 do 1 wykorzystując przy tym kolejkę priorytetową. Wyniki sortowania zapisywane są do pliku „test12" w folderze zawierającym wyniki testów.

Plik powinien zawierać 1003 linie. W pierwszej linii powinna być zapisana informacja o sortowaniu kopcowym. Następne 1000 linii to liczby, które powinny być uporządkowane rosnąco. W 1002 linii powinna pojawić się informacja o tym czy kolejka priorytetowa działa poprawnie (program testujący sprawdza również czy liczby na pewno są posortowane). Ostatnia linia powinna być pusta.

W ramach testu na początku kompilowany jest także program testujący, który na koniec jest usuwany.

Wywoływane komendy:

```
cc -o pq_test priority_queue.c ../Tests/priority_queue_test.c
./pq_test 1000 [folder zaw. wyn. testów]/test12
rm pq_test
```

Przykładowy wynik testu zakończonego powodzeniem:
Plik "test12":

```
Heap sort:
0.000368076
0.000576437
0.00139771
[... pominięto 994 linie ... ]
0.998762
0.99949
0.999512
Priority queue works properly.
```