Specyfikacja programu *graphalgo*

Bartosz Socki
Kacper Wiączek

Marzec 2022

1 Specyfikacja Funkcjonalna

1.1 Nazwa programu

`graphalgo` – program implementujący algorytmy działające na grafie, wypisuje wynik działania na stdout, czyta dane z stdin.

1.2 Sposób wywołania

```
graphalgo -g -r<ROWS> -c<COLS> -n<MIN> -x<MAX> [-s <SEED>]

graphalgo -b -1<VERT_BEG>

graphalgo -d -1<VERT_BEG> -2<VERT_END>
```

1.3 Opis argumentów

`--generate, -g`

generuje graf, należy podać z odpowiednimi flagami ilość kolum, wierszy, zakres wag. (jeżeli nie zostanie wprowadzona wystarczająca ilość opcji to program zwróci błąd na stderr).

`--rows, -r <ROWS>`

podaje ilość wierszy dla grafu, wymagane podanie przy generacji grafu. (musi być większa od 0)

`--cols, -c <COLS>`

podaje ilość kolumn dla grafu, wymagane podanie przy generacji grafu. (musi być większa od 0)

`--min, -n <MIN>`

podaje minimum dla wag generowanych na krawędziach, wymagane podanie przy generacji grafu.

`--max, -x <MAX>`

podaje maksimum dla wag generowanych na krawędziach, wymagane podanie przy generacji grafu.

`--seed, -s <SEED>`

ziarno dla generatora grafu, jeżeli pominięte to przyjmuje za ziarno unix timestamp.

`--vert1, -1 <VERT_BEG>`

wierzchołek startowy.

`--vert2, -2 <VERT_END>`

wierzchołek końcowy.

`--bfs, -b`

uruchamia algorytm bfs na grafie. Sprawdza spójność, zwraca odległości od punktu początkowego do wszystkich innych wierzchołków, tak jak by wagi wynosiły 1.

`--dijkstra, -d`

uruchamia algorytm dijkstry, wypisuje najkrótszą ścieżkę między wierzchołkami zdefiniowanymi przy użyciu opcji –**vert1**=<**VERT_BEG**>, –**vert2**=<**VERT_END**>. Zwraca odległości wszystkich wierzchołków od wierzchołka <**VERT_END**>. Funkcja wymaga podania grafu na standardowe wejście. (jeżeli wierzchołek będzie znajdował się poza grafem, to program zwróci błąd na stderr).

1.4 Format wejścia

Pierwszy wiersz zawiera dwie liczby naturalne, kolejno liczbę wierszy **R** i liczbę kolumn **C**. Potem w **R*C** wierszach opisywane są listy sąsiedztwa dla kolejnych wierzchołków. Format listy sąsiedztwa dla wierzchołka **U** wygląda następująco: $\mathbf{V_0}$: $\mathbf{W_0}$ $\mathbf{V_1}$: $\mathbf{W_1}$ ... $\mathbf{V_n}$: $\mathbf{W_n}$, gdzie, **V** jest wierzchołkiem, a **W** jest wagą krawędzi pomiędzy wierzchołkiem **U**, a wierzchołkiem **V**

1.5 Przykłady

`graphalgo -g -r20 -c20 --min=0 --max=1`

generuje graf o 20 wierszach, 20 kolumnach, o wagach w zakresie 0..1 i wsypisuje go na stdout.

`graphalgo -g -r20 -c20 --min=0 --max=1 -s 100`

generuje graf o 20 wierszach, 20 kolumnach, o wagach w zakresie 0..1 i ziarnie do generowania wag = 100, wypisuje graf na stdout.

`graphalgo -g -r20 -c20 --min=2 --max=2 > out.txt`

generuje graf o 20 wierszach, 20 kolumnach i wagach na krawędziach równych 2, zapisuje graf do pliku out.txt.

`graphalgo -g -r20 -c20 -n0 -x1 | graphalgo -b -1 1`

generuje graf o 20 wierszach, 20 kolumnach, o wagach w zakresie 0..1 i uruchamia na nim algorytm BFS dla wierzchołka o indeksie 1.

`cat out.txt | graphalgo -b -1 1`

czyta graf z stdin i uruchamia na nim algorytm BFS dla wierzchołka o indeksie 1.

`cat out.txt | graphalgo -d -1 2 -2 10`

czyta graf z stdin i wyznacza najkrótszą drogę pomiedzy węzłami 2 i 10.

2 Specyfikacja Implementacyjna

2.1 Struktura plików źródłowych

- `src/` - pliki źródłowe
  - `main.c` - plik główny programu, przetwarza argumenty wywołania, wywołuje odpowiedni algorytm na grafie.
  - `vertex_priority_queue.c, vertex_priority_queue.h` - implementacja kolejki priorytetowej.
  - `graph.c, graph.h` - implementacja grafu.
  - `dijkstra.c, dijkstra.h` - implementacja algorytmu Dijkstry.
  - `bfs.c, bfs.h` - implementacja algorytmu BFS.
- `docs/` - folder z dokumentacją projektu
- `bin/` - pliki wykonywalne
- `objs/` - folder na pliki **.o**
- `tests/` - testy do programu
- `Makefile` - buduje program, uruchamia testy

2.2 Podstawowe struktury danych

2.2.1 Graph

Struktura przechowująca wierzchołki, krawędzie i ich wagi.

```
typedef struct _Graph {
    // gdzie edges[i] to lista krawędzi od wierzchołka i
    Edge** edges;
    size_t rows;
    size_t cols;
} Graph;

// generuje graf z ziarna
Graph* graph_generate_from_seed(int rows, int cols, double
   min, double max, long seed);

// czyta graf z stdin, zwraca błąd przy niepoprawnym formacie
Graph* graph_read_from_stdin();

// wypisuje graf w formacie zdefiniowanym w 1.4
void graph_print_to_stdout(Graph* graph);

// zamienia "współrzedne x, y" wierzchołka na indeks w tablicy edges
int graph_xy_to_index(Graph* graph, int row, int col);

void graph_free(Graph* graph);
```

2.2.2 VertexPriorityQueue

Zaimplementowana przy pomocy kopca minimalnego. Do implementacji została utworzona struktura pomocnicza **QueuedVertex**. Struktura przechowuje wierzchołki grafu. Pozwala na zwrócenie wierzchołka o najmniejszym dystansie, aktualizację priorytetu wierzchołka oraz na dodawanie wierzchołków do kolejki.

```
typedef struct{
    //indeks wierzchołka
    int index;
    // dystans, traktowany jako priorytet
    double dist;
} QueuedVertex;
typedef struct{
    //wielkość kolejki
    int capacity;
    //liczba elementów w kolejce
    int size;
```

```
    //tablica przechowująca informacje o tym gdzie znajduje się
    //wierzchołek o danym indeksie w tablicy verticies
    int * verticies_indexes;

    QueuedVertex ** verticies;
} VertexPriorityQueue;

//inicjalizacja kolejki
VertexPriorityQueue * vertex_priority_queue_initalize(int
   number_of_verticies);

//oczyszczanie pamięci po kolejce
void vertex_priority_queue_free(VertexPriorityQueue * pr);

//dodawanie elementów do kolejki
void vertex_priority_queue_add(VertexPriorityQueue * pr,
   QueuedVertex * item);

//usuwanie elementu z kolejki
QueuedVertex *  vertex_priority_queue_poll(
   VertexPriorityQueue * pr);

//aktualizacja priorytetu elementu
void vertex_priority_queue_update(VertexPriorityQueue * pr,
   int index, double new_dist);
```

2.2.3 Edge

Struktura będzie przechowywała dane o krawędziach między wierzchołkami, przechowywane są waga i wierzchołek końcowy.

```
typedef struct Edge_t {
    // lista krawędzi z wierzchołka
    struct Edge_t* next;
    int end_vertex;
    double weight;
} EdgeNode;

// inicjuje krawędź
Edge* edge_node_init(int connected_vertex, double weight,
   Edge* next);

void edge_node_free(Edge* edge);

// uruchamia bfs na danym grafie
```

```
BFSResult *bfs(Graph* graph, int start_vertex);

// wypisuje wynik bfs
void bfs_print_result(BFSResult *result);

// zwalnia strukture
void bfs_result_free(BFSResult *result);
```

2.2.4 BFSResult

Struktura będzie przechowywała wynik wykonania algorytmu BFS na podanym grafie.

```
typedef struct _BFSResult {
    bool is_connected; // connected = graf jest spójny
    int * pred; //poprzedni wierzchołek
    int * dist; //odległość od punktu początkowego
} BFSResult;
```

2.2.5 DijkstraResult

Struktura będzie przechowywała wynik wykonania algorytmu dijkstry na podanym grafie.

```
typedef struct{
    //index poprzedniego wierzchołka
    int * pred;
    //odległości od wierzchołka początkowego
    double * dist;
    //wierzchołek początkowy
    int source;
    //ilość wierzchołków
    int no_verticies;
} DijkstraResult;

//uruchamia algorytm
DijkstraResult * dijkstra(Graph * graph, int source);

//oczyszcza wynik działania algorytmu
void dijkstra_result_free(DijkstraResult * result);

//wypisuje tabelę zawierającą wierzchołek, jego poprzedni wierzchołek i
   jego odległość od punktu startowego
void dijkstra_print_result(DijkstraResult * res);
```

```
//wypisuje ścieżkę między punktem startowym zdefiniowanym przy
    uruchomieniu algorytmu) , a drugi sprecyzowanym przez użytkownika
void dijkstra_print_path(DijkstraResult * res, int to);
```