Specyfikacja implementacyjna projektu 1

Sebastian Pietrykowski, Paweł Borkowski
Grupa projektowa nr 3

16 marca 2022

1 Opis ogólny

Program zostanie napisany w języku C zgodnie ze standardem C99. Obsługa przez użytkownika będzie możliwa jedynie w trybie wsadowym.

Cel projektu 1 jest zdefiniowany w specyfikacji funkcjonalnej.

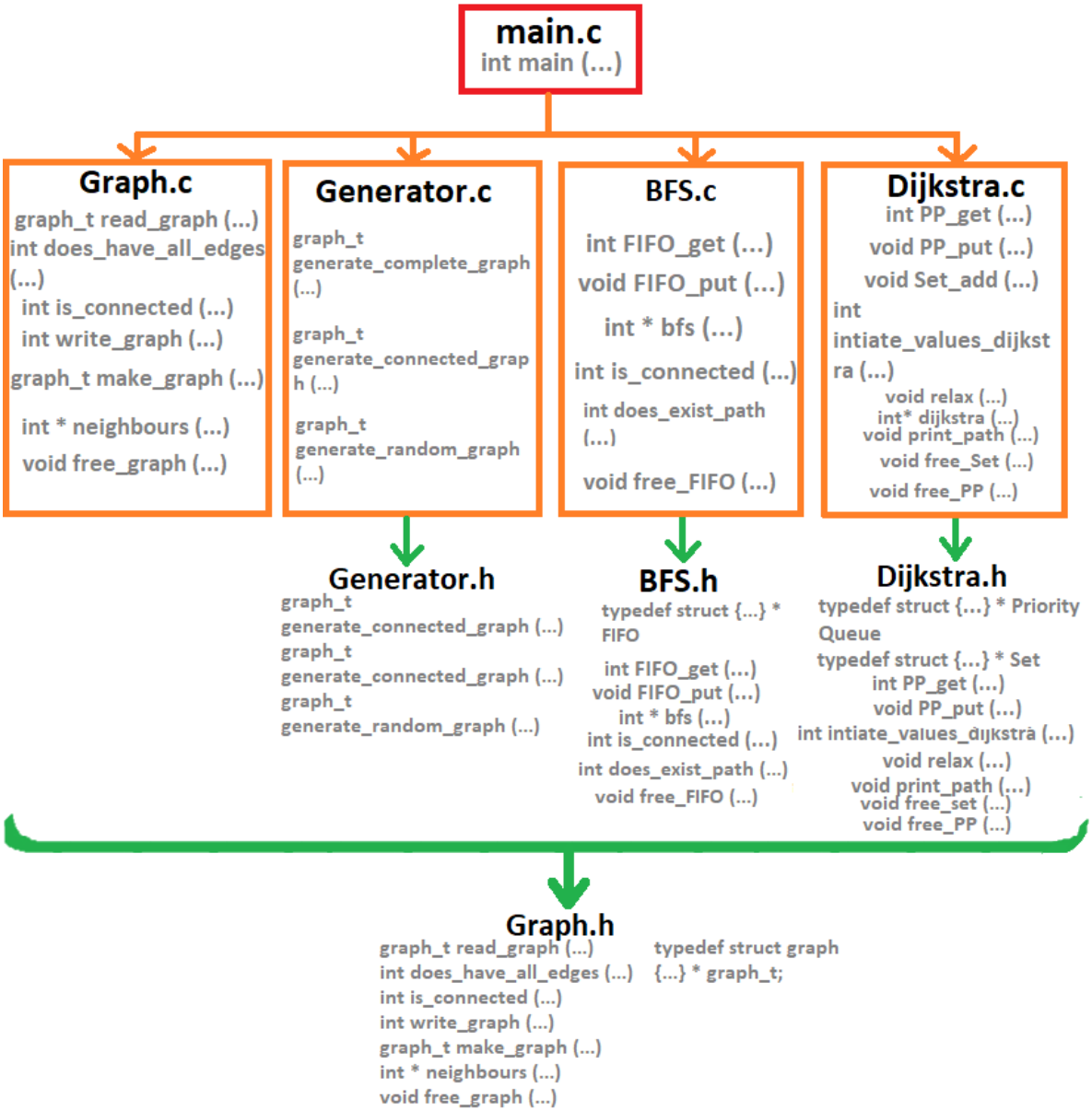
2 Środowisko deweloperskie

Projekt 1 jest tworzony w edytorze tekstu Vim version 8.0, używany jest kompilator GCC version 7.3.0 (Ubuntu 7.3.0-27ubuntu1 18.04). Prace zostaną przeprowadzone zdalnie przy użyciu protokołu ssh na komputerze stud.jimp.iem.pw.edu.pl, należącym do IETiSIP na Politechnice Warszawskiej.

W projekcie 1 wykorzystywana jest konwencja nazewnicza Snake Case.

Do pracy przy projekcie używany jest system kontroli wersji Git. Poszczególne gałęzie noszą nazwy modułów implementowanych w programie (w danej gałęzi implementowany jest jeden dany moduł), są one scalane do gałęzi głównej po prawidłowym przejściu testów.

3 Opis modułów



Rys.1 Diagram modułów.

Program będzie składał się z kilku plików, w których zawarte będą następujące moduły:

Main
Plik główny sterujący działaniem programu.

Graph
Odpowiedzialny za przechowywanie grafu. Posiada funkcje czytające graf z pliku, zapisujące go do pliku oraz sprawdzające pewne warunki.

Generator
Moduł ten jest odpowiedzialny za generowanie grafu. W zależności od trybu wybranego przez użytkownika generowany jest inny graf.

BFS

W tym module znajduje się implementacja algorytmu przeszukiwania wszerz (Breadth–first search – BFS). Jest jednym z najprostszych algorytmów przeszukiwana grafu. Działanie algorytmu głównie polega na dodawaniu kolejnych wierzchołków do kolejki, a następnie usuwaniu ich przy przejściu do kolejnej warstwy grafu. Czynności te powtarzamy aż do przejścia całego grafu.

Dijkstra
W tym module znajduje się implementacja algorytmu Dijkstry, służącego do znajdywania najkrótszej ścieżki pomiędzy jednym wierzchołkiem a wszystkimi innymi osiągalnymi wierzchołkami w grafie. W wynikowym zbiorze znajdzie się więc również najkrótsza możliwa ścieżka do zadanego wierzchołka.

4 Opis plików

Main

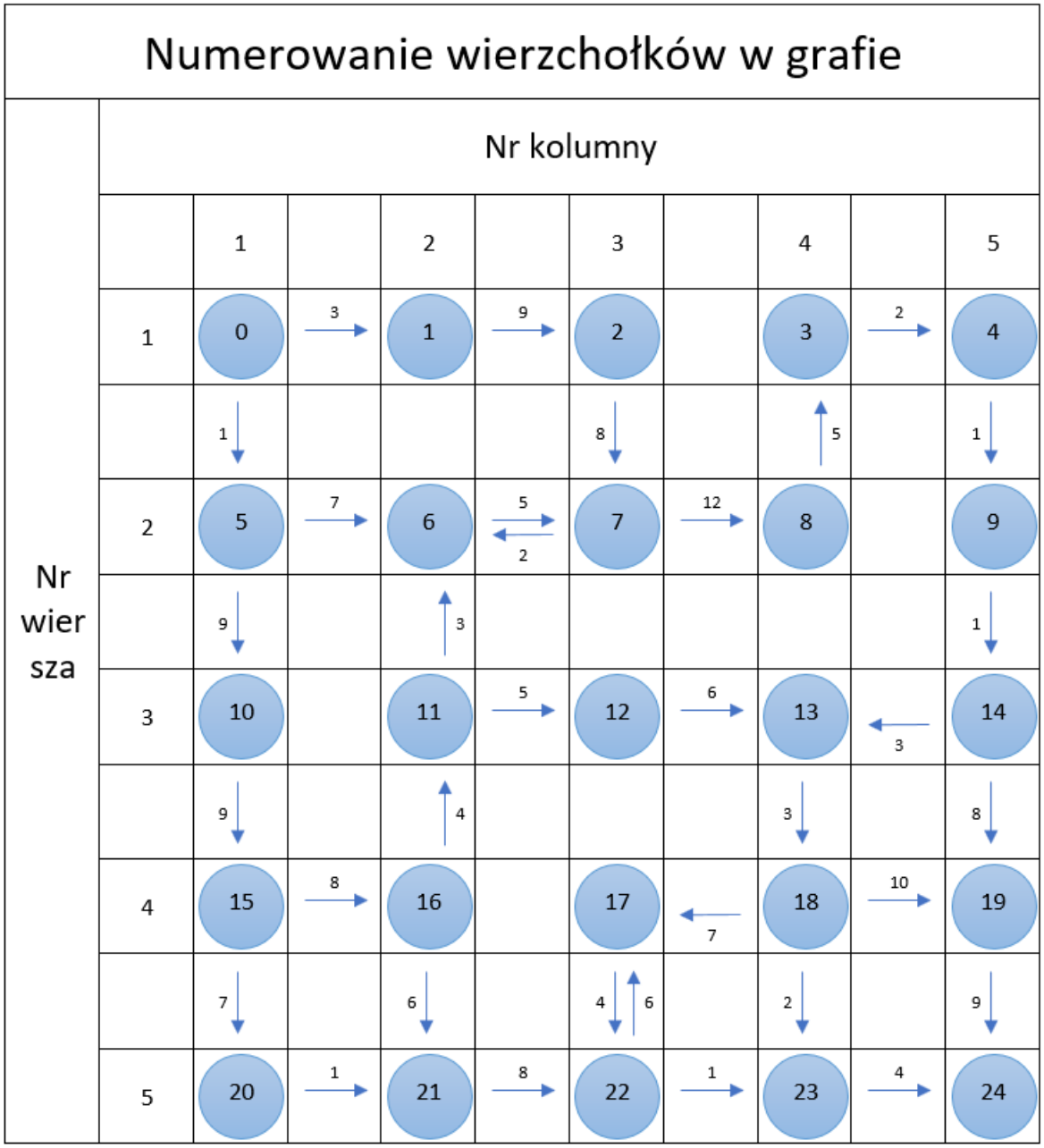
- Funkcje
  - main – funkcja główna sterująca działaniem programu, do interpretacji instrukcji podanych przez użytkownika używa biblioteki getopt.
    - Zmienne
      - graph_t graph – przechowuje graf
      - int opt – znak odczytany przez mechanizm getopt
      - char *inp – nazwa pliku wejściowego
      - char *out – nazwa pliku wyjściowego
      - int columns – liczba kolumn w grafie
      - int rows – liczba wierszy w grafie
      - double from_weight – początek zakresu, z którego będą losowane wagi dla krawędzi, nie włącznie
      - double to_weight – koniec zakresu, z którego będą losowane wagi dla krawędzi, nie włącznie
      - int mode – tryb działania programu, określa sposób generowania grafu lub warunki, które musi spełnić wczytywany plik z grafem, możliwe do wyboru tryby: 1, 2, 3
      - int start_vertex_number – nr wierzchołka, z którego ma zostać wyznaczona najkrótsza możliwa droga
      - int end_vertex_number – nr wierzchołka, do którego ma zostać wyznaczona najkrótsza możliwa droga
      - int does_check_connectivity – określa, czy użytkownik życzy sobie sprawdzić spójność grafu; 1 jeżeli tak, 0 jeżeli nie
      - int does_print_weights – określa, czy użytkownik życzy sobie, aby wypisać wagi krawędzi w najkrótszej możliwej ścieżce; 1 jeżeli tak, 0 jeżeli nie.

Graph

- Struktury
  - graph_t – przechowuje graf, zmienna jest wskaźnikiem
    - int columns – liczbę kolumn
    - int rows – liczba rzędów
    - int no_vertexes – liczba wierzchołków w grafie
    - double * adj_mat – macierz sąsiedztwa (ang. adjacency matrix) przechowująca krawędzie grafu: wskaźnik na zaalokowany blok pamięci o rozmiarze na no_vertexes*no_vertexes elementów; jeżeli istnieje krawędź z wierzchołka i do wierzchołka j, to element na pozycji i*no_vertexes+j jest równy wadze odpowiadającej tej krawędzi jeżeli wartość jest większa od 0, jeżeli krawędź nie istnieje wartość elementu jest równa -1.

- Funkcje
  - graph_t read_graph(FILE * in) – wczytuje graf znajdujący się w pliku in; zwraca wczytany graf w przypadku sukcesu, NULL jeżeli wystąpił błąd
  - int does_have_all_edges(graph_t graph) – sprawdza, czy graf posiada wszystkie możliwe krawędzie (pomiędzy wierzchołkami sąsiadującymi poziomo lub pionowo); zwraca 1 jeżeli tak, 0 jeżeli nie; warunek sprawdzany w trybie 1
  - int is_connected(graph_t graph) – sprawdza, czy graf jest spójny; warunek sprawdzany w trybie 2; zwraca 1 jeżeli tak, 0 jeżeli nie; warunek sprawdzany w trybie 2
  - int write_graph(graph_t graph, FILE * out) – zapisuje graf znajdujący się w zmiennej graph do pliku in; zwraca 0 w przypadku sukcesu, 1 jeżeli wystąpił błąd
  - graph_t make_graph(int columns, int rows) – tworzy nowy graf, wypełnia w nim wartościami zmienne columns, rows i no_vertexes, alokuje pamięć na wektor adj_mat o rozmiarze na no_vertexes*no_vertexes elementów
  - int * neighbors(graph_t graph, int vertex) – zwraca tablicę wierzchołków będącymi sąsiadami wierzchołka vertex – do których istnieje krawędź z wierzchołka vertex
  - void free_graph(graph_t graph) – zwalnia z pamięci graf graph.

Sposób numerowania wierzchołków w grafie został przedstawiony na poniższej grafice:



Rys.2 Sposób numerowania wierzchołków w grafie na przykładzie pewnego grafu.

Sposób numerowania indeksów odpowiadającym krawędziom w macierzy sąsiedztwa dla grafu o 5 wierzchołkach						
		Do wierzchołka				
		0	1	2	3	4
Z wierzchołka	0	5*0+0 = 0	5*0+1 = 1	5*0+2 = 2	5*0+3 = 3	5*0+4 = 4
	1	5*1+0 = 5	5*1+1 = 6	5*1+2 = 7	5*1+3 = 8	5*1+4 = 9
	2	5*2+0 = 10	5*2+1 = 11	5*2+2 = 12	5*2+3 = 13	5*2+4 = 14
	3	5*3+0 = 15	5*3+1 = 16	5*3+2 = 17	5*3+3 = 18	5*3+4 = 19
	4	5*4+0 = 20	5*4+1 = 21	5*4+2 = 22	5*4+3 = 23	5*4+4 = 24

Rys.3 Sposób numerowania indeksów odpowiadającym krawędziom w macierzy sąsiedztwa (adj_mat) dla grafu o 5 wierzchołkach.

Generator

- Funkcje
  - graph_t generate_complete_graph(double from_weight, double to_weight) – zwraca graf wygenerowany zgodnie z trybem 1: z wszystkimi możliwymi krawędziami (pomiędzy wierzchołkami sąsiadującymi poziomo lub pionowo) oraz z losowymi wagami krawędzi z zakresu (from_weight,to_weight)
  - graph_t generate_connected_graph(double from_weight, double to_weight) – zwraca graf wygenerowany zgodnie z trybem 2: spójny z losowymi wagami krawędzi z zakresu (from_weight,to_weight)
  - graph_t generate_random_graph(double from_weight, double to_weight) – zwraca graf wygenerowany zgodnie z trybem 3: z losowo występującymi krawędziami (spójny lub niespójny) oraz z losowymi wagami krawędzi z zakresu (from_weight,to_weight).

BFS

- Struktury
  - FIFO – kolejka First–in, First–out, zmienna jest wskaźnikiem
    - int * vertexes – tablica wierzchołków dodanych do kolejki
    - no_elements – liczba wierzchołków w tablicy vertexes.
- Funkcje
  - int FIFO_get(FIFO fifo) – usuwa z kolejki element na pierwszej pozycji, przesuwa elementy w kolejce fifo o 1 pozycję do przodu, zwraca usunięty element
  - void FIFO_put(FIFO fifo, int vertex) – dodaje do kolejki fifo wierzchołek vertex
  - int * bfs (graph_t graph, int start_vertex_number, int end_vertex_number) – zwraca tablicę poprzedników otrzymanych w wyniku działania algorytmu bfs; bfs[x] przechowuje numer wierzchołka będącego poprzednikiem wierzchołka x, jeżeli wierzchołek nie ma poprzednika, pod jego indeksem przechowywana jest wartość -1
  - int is_connected(int * predecessors) – na wejście przyjmuje tablicę poprzedników, otrzymaną w wyniku działania bsf(graph_t graph); sprawdza, czy ten graf jest spójny, zwraca 1 jeżeli tak, 0 jeżeli nie
  - OPCJONALNIE: int does_exist_path(int * predecessors, int start_vertex_number, int end_vertex_number) – na wejście (predecessors) przyjmuje tablicę poprzedników, otrzymaną w wyniku działania bsf(graph_t graph); zwraca 0 jeśli droga między wierzchołkiem początkowym start_vertex_number a końcowym end_vertex_number w grafie graph nie istnieje lub zwraca 0 jeśli graf jest niespójny i wybrano tryb 2 pracy programu. Zwraca 1 jeśli droga między wybranymi wierzchołkami istnieje
  - void free_FIFO(FIFO fifo) – zwalnia z pamięci kolejkę fifo.

Złożoność czasowa algorytmu BFS wynosi O(V+E).
Gdzie: V – liczba wierzchołków, E – liczba krawędzi w grafie.

Algorytm BFS:

```
szukaj_wszerz( Graf G, Wierzchołek s ):
    inicjuj Color c[ 0...G.liczbaWierzchołków()-1 ]    //kolory:
                BIAŁY – nie odwiedzono wierzchołka,
                SZARY – odwiedzono wierzchołek, ale nie odwiedzono sąsiednich wierzchołków,
                CZARNY – odwiedzono wierzchołek i wszystkie sąsiednie wierzchołki
    inicjuj Integer poprzednik[ 0...G.liczbaWierzchołków()-1 ]    //poprzedniki
    inicjuj Integer l[ 0...G.liczbaWierzchołków()-1 ]    //odległość od punktu START
    inicjuj kolejkę First–In, First–Out FIFO<Wierzchołek>
    dla każdego Wierzchołka w z G.wierzchołki() wykonaj    //inicjuj początkowe wartości
        c[w] ← BIAŁY
        l[w] ← INFINITY
        poprzednik[w] ← -1
    c[s] ← SZARY    //odwiedź wierzchołek s
    l[s] ← 0
    FIFO.put(s)
    dopóki FIFO nie jest pusta wykonuj
        Wierzchołek w ← FIFO.get()
        dla każdego Wierzchołka v z G.sąsiednie(w) wykonaj    //odwiedź sąsiadów wierzchołka w
            jeżeli c[v] = BIAŁY wykonaj
                c[v] ← SZARY
                l[v] ← l[w]+1
                poprzednik[v] ← w
                FIFO.put(v)
```

```
c[w] ← CZARNY    //odwiedzono wierzchołek w i jego sąsiadów
```

Szczegóły implementacyjne: typ Color jest reprezentowany przez char*, zamiast typu Wierzchołek używa się int.

Jeżeli iterując po tablicy poprzednik[] natrafimy na wartość -1 w innym elemencie niż poprzednik[s] (poprzednik wierzchołka START), to znaczy, że graf jest niespójny. Ponadto w tablicy l[] zapisane są odległości poszczególnych wierzchołków od wierzchołka START.

Dijkstra

- Struktury
  - PriorityQueue – kolejka priorytetowa, zmienna jest wskaźnikiem
    - int * vertexes – tablica wierzchołków dodanych do kolejki
    - double * distances – każdy element odpowiada elementowi o takim samym indeksie w vertexes, zawiera odległości do danych wierzchołków
    - no_elements – liczba wierzchołków w tablicy vertexes.
  - Set – zawiera tablicę przechowującą elementy, która w razie potrzeby ulega zmianie, zmienna jest wskaźnikiem
    - int * vertexes – tablica dodanych wierzchołków
    - no_elements – liczba wierzchołków w tablicy vertexes.
- Funkcje
  - int PP_get(PriorityQueue pp) – usuwa z kolejki pp element o najmniejszej odległości, odpowiednio przesuwa elementy w kolejce fifo, zwraca usunięty element
  - void PP_put(PriorityQueue pp, int vertex, double distance) – dodaje do kolejki pp wierzchołek vertex o odległości distance od aktualnie badanego wierzchołka
  - void Set_add(Set set, int vertex) – dodaje do tablicy vertexes w strukturze set zmienną wertex, w razie potrzeby zmienia rozmiar tej tablicy
  - int initiate_values_dijkstra(graph_t graph, int start–vertex–number, int ** p, double ** o) – inicjuje tablice p (poprzedniki) i o (odległości), zgodnie z algorytmem opisanym niżej
  - void relax(graph_t graph, int u, int v, int ** p, double ** o) – potencjalnie dodaje kolejny wierzchołek do tablic o i p, zgodnie z algorytmem opisanym niżej
  - int * dijkstra(grapht_t graph, int start_vertex_number) – funkcja odpowiedzialna za algorytm Dijkstry. Zwraca wskaźnik na tablicę typu int, w której każdy indeks reprezentuje wierzchołek o danym numerze oraz pod tym indeksem przechowywany jest numer wierzchołka będący poprzednikiem. W razie błędu funkcja zwraca NULL
  - void print_path(int * predecessors, int start_vertex_number, int end_vertex_number, int does_print_weights) – jako argument przyjmuje tablicę poprzedników, otrzymaną w wyniku działania algorytmu dijkstra(), wypisuje najkrótszą możliwą ścieżkę miedzy wierzchołkiem start_vertex_number a end_vertex_number; zmienna int does_print_weights określa, czy mają zostać wypisane wagi krawędzi wchodzących w skład drogi – 0 jeżeli nie, 1 jeżeli tak
  - void free_PP(PriorityQueue pp) – zwalnia z pamięci kolejkę priorytetową pp
  - void free_Set(Set set) – zwalnia z pamięci set.

Złożoność czasowa algorytmu Dijkstry wynosi $O(E \cdot \log V)$.
Gdzie: V – liczba wierzchołków, E – liczba krawędzi w grafie.

Funkcje pomocnicze dla algorytmu:

```
inicjujNS1Z( Graf G, Wierzchołek s ):    //inicjuje tablicę poprzedników i odległości
  inicjuj Integer p[ 0...G.liczbaWierzchołków()-1 ]    //poprzedniki
```

```
    inicjuj Double o[ 0...G.liczbaWierzchołków()-1 ]    //odległości od punktu START
    dla każdego Wierzchołka w z G.wierzchołki() wykonaj
        o[w] ← INFINITY
        p[w] ← -1
    o[s] ← 0
    zwróć parę <o,p>

relax( Graf G, Wierzchołek u, Wierzchołek v, Double[] o, Integer[] p ):
    / /wyznacza kolejną część drogi
    // u – sąsiad Wierzchołka v, v – potencjalny poprzednik Wierzchołka u
    // o[u] = INFINITY (ścieżka do punktu nie została wyznaczona) lub nowa ścieżka jest krótsza
    jeżeli o[u] > o[v] + G.waga(u,v) to
        o[u] ← o[v] + G.waga(u,v)
        p[u] ← v
```

Algorytm:

```
aDijkstry( Graf G, Wierzchołek s ):
    inicjuj parę <o,p> ← inicjujNS1Z( G, s )
    inicjuj SET<Wierzchołek> w
    inicjuj PRIOTITY_QUEUE<Wierzcholek> q, zawierającą wszystkie elementy z G.wierzchołki(),
o priotytecie 1/o[]

    dopóki q nie jest pusta wykonuj
        Wierzchołek u ← q.get()    //element o najmniejszym koszcie dojścia (najmniejszym o)
        w.add(u)
        dla każdego Wierzchołka v z G.sąsiednie(u) wykonaj
            jeżeli v nie jest w w, wykonaj
                relax( G, v, u, o, p)
```

Szczegóły implementacyjne: zamiast typu Wierzchołek używa się int, zamiast pary <o,p> odpowiednie tablice zmieniane są przy użyciu wskaźników, nie następuje ich zwrócenie w funkcji inicjujNS1Z.

Drogę z wierzchołka a do wierzchołka b, można znaleźć iterując po poprzednikach b (tablica p) – p[b] ->p [p[b]] -> p[p[p[b]]], aż do napotkania a jako poprzednika. Jeżeli program napotka na wartość -1, takie połączenie nie istnieje.

5 Testowanie

Testy jednostkowe zostaną napisane w języku C. Będą one uruchamiane używając osobnego pliku z metodą main(), w której zostaną wywołane funkcje porównujące zmienne z ich przewidywanymi prawidłowymi wartościami na dany moment. Planuje się następujące testy jednostkowe:

1. Moduł Graph
   (a) void test_read_graph(graph_t correct_graph, FILE * in) – testuje funkcję read_graph(FILE * in)
   (b) void test_does_have_all_edges(int correct_answer, graph_t graph) – testuje funkcję does_have_all_edges(graph_t graph)
   (c) void test_is_connected(int correct_answer, graph_t graph) – testuje funkcję is_connected(graph_t graph)
   (d) void test_write_graph(graph_t graph, FILE * out) – testuje funkcję write_graph(graph_t graph, FILE * out)
   (e) void test_make_graph(graph_t correct_graph, int columns, int rows) – testuje funkcję make_graph(int columns, int rows)

   (f) void test_neighbors(graph_t graph, int vertex) – testuje funkcję neighbors(graph_t graph, int vertex).

2. Moduł Generator

   (a) void test_generate_complete_graph(double from_weight, double to_weight) – testuje funkcję graph_t generate_complete_graph(double from_weight, double to_weight), używając funkcji int does_have_all_edges(graph_t graph)

   (b) void test_generate_connected_graph(double from_weight, double to_weight) – testuje funkcję graph_t generate_connected_graph(double from_weight, double to_weight), używając funkcji int is_connected(graph_t graph)

   (c) void test_generate_random_graph(double from_weight, double to_weight) – testuje funkcję graph_t generate_random_graph(double from_weight, double to_weight).

3. Moduł BFS

   (a) test_FIFO – testuje kolejkę FIFO, w tym metody void FIFO_put(FIFO fifo, int vertex) i void FIFO_get(FIFO fifo)

   (b) test_bfs (graph_t graph, int start_vertex_number, int end_vertex_number) – testuje funkcję int * bfs (graph_t graph, int start_vertex_number, int end_vertex_number)

   (c) void test_is_connected(int * predecessors) – testuje funkcję int is_connected(int * predecessors).

4. Moduł Dijkstra

   (a) test_PriorityQueue(PriorityQueue pp) – testuje strukturę PriorityQueue oraz funkcje int PP_get(PriorityQueue pp) i void PP_put(PriorityQueue pp, int vertex, double distance)

   (b) test_Set(Set set) – testuje strukturę Set wraz z funkcją void Set_add(Set set, int vertex)

   (c) test_dijkstra(grapht_t graph, int start_vertex_number, int end_vertex_number) – testuje funkcje odpowiedzialne za algorytm Dijkstry: initiate_values_dijkstra, relax, dijkstra.

Literatura

[1] Jacek Starzyński. *Prezentacja "Algorytmy dla grafów" na podstawie: Cormen, Leiserson, Rivest, Stein: "Wprowadzenie do algorytmów", WNT 2004*

Źródło Rys.1, Rys.2, Rys.3: rysunek własny