Spis treści

1 Opis ogólny

1.1 Nazwa programu

SUGP -The Shortest Undirected Graph Path (Poszukiwanie najkrótszej ścieżki w grafie nieskierowanych.)

1.2 Przeznaczenie dokumentu

Specyfikacja implementacyjna projektu SUGP jest dokumentem omawiającym tematykę przedstawianego oprogramowania pod kątem implementacji. Wyjaśnia takie aspekty programu jak jego budowę, przeprowadzanie testów i podejście konceptualne przyświecające procesowi tworzenia. Dokument ten stanowi źródło wiedzy dla osób zainteresowanych działaniem oprogramowania *SUGP*.

1.3 Cel programu

Celem programu jest znalezienie najkrótszej ścieżki w grafie nieskierowanym. Graf będzie wczytywany z pliku o ustalonej strukturze lub generowany na podstawie zadanych parametrów wejściowych. Generowany graf będzie zapisywany w postaci pliku.

1.4 Scenariusz działania programu

Program na podstawie podanych argumentów wejściowych ustali czy należy generować graf czy wczytać go z pliku tekstowego.

- Jeżeli dostaje 2 liczby (int) reprezentujące liczbę wierszy i liczbę kolumn lub 4 liczby (dwie pierwsze(int) - liczba wierszy i liczba kolumn, dwie drugie(double) - zakres wartości do wygenerowania wag krawędzi), to
    1. Generuje graf ważony o wagach w zakresie podanym przez użytkownika, lub, jeżeli zakres nie zostanie podany, program generuje wagi w domyślnym zakresie <0,1>.
    2. Zapisuje wygenerowany graf do pliku Graph.txt.
- Jeżeli plik tekstowy (.txt) i 2 liczby(int) okreslające wierzchołek początkowy i wierzchołek końcowy do znajdowania ścieżki - przechodzi do następnego kroku.
    1. Program sprawdza spójność grafu, w przypadku grafu spójnego kontynuje działanie, a w przypadku grafu niespójnego wyświetla komunikat o błędzie.
    2. Program wyszukuje najkrótsze ścieżki pomiędzy wybranymi punktami.

W przypadku, gdy użytkownik chce wygenerować graf oraz znaleźć w nim ścieżkę pomiędzy poszczególnymi punktami, użytkownik powinien uruchomić program dwa razy. Pierwszy raz, żeby wygenerować graf na podstawie podanych parametrów i zapisać go do pliku, drugi - żeby znaleźć najkrótszą ścieżkę we wcześniej wygenerowanym grafie pomiędzy punktami podanymy jako argumenty wywołania.

1.5 Środowisko powstawania

Program *SUGP* jest napisany w języku programowania C. Zintegrowanym środowiskiem programistycznym używanym w procesie tworzenia aplikacji jest „blabla". Dokładnie wersje środowiska programistycznego:

Element śrdowiska	Wersja
Język programowania	C17 (ISO/IEC 9899:2018)
IDE Visual Studio Code	1.65.2

2 Budowa programu

2.1 Przechowywanie danych w pamięci

Dane programu będą wczytywane do list sąsiedztwa. Lista sąsiedztwa umożliwia efektywne (szybkie) wyszukiwanie sąsiadów. Z punktu działania programu jest to bardzo ważna kwestia, gdyż algorytm poszukiwania ścieżki często odwołuje się do wierzchołków sąsiadujących z wierzchołkiem przetwarzanym (połączonym z nim krawędzią). Dodatkowo lista sąsiedztwa jest najwydajniejszym sposobem reprezentacji grafu ze względu na pamięć komputera. Prezentuje się ona znacznie lepiej niż macierz sąsiedztwa (pamięć rzędu O (n^2), gdzie n – liczba wierzchołków) czy macierz incydencji (pamięć rozmiaru O (m x n)), ponieważ zajmuje ona pamięć rzędu O(m), gdzie m to liczba krawędzi grafu.

2.2 Struktura danych

2.2.1 Dane wejściowe

Istnieją dwa możliwe warianty danych wejściowych:

1. Nazwa pliku(*.txt) zawierającego strukturę grafu, dwa parametry(int) określające wierzchołek startowy i końcowy.

   Wywołanie:

   ```
   ./out -nazwa_pliku(*.txt) -nr_wierzcholka_start(int) -nr_wierczholka_koniec(int)
   ```

2. Zestaw parametrów wejściowych na podstawie których zostanie wygenerowany graf.

   Wywołanie:

   ```
   ./out -liczba_wierszy(int) -liczba_kolumn(int) -lewa_granica_zakresu(double)
           - prawa_granica_zakresu(double)
   ```

   Lub 2 wariant wywołania, w tym przypadku wagi krawędzi należy generować w przedziale <0,1> typu double:

   ```
   ./out -liczba_wierszy(int) -liczba_kolumn(int)
   ```

Ad. 1 W przypadku wczytywania grafu dane wejściowe zawarte będą w pliku tekstowym (*.txt), którego nazwę zostanie podana jako argument wywołania.

Struktura pliku:

- Pierwszy wiersz pliku zawierać będzie kolejno: liczbę kolumn i liczbę wierszy grafu.
- Kolejne wiersze pliku dotyczyły będą kolejno wierzchołków od 0 do n(int), gdzie n(int) to liczba wierzchołków pomniejszona o 1 (numeracja wierzchołków zaczyna się od 0). Każdy z wierszy zawierać będzie listę par (numer wierzchołka(int) :wagę krawędzi(double)), gdzie separatorem jest „:”, a liczby zmiennoprzecinkowe wyrażające wagi zapisywane są przy pomocy znaku kropki (zamiast przecinka).

Przykładowo:

```
7 4
1 :0.8864916775696521  4 :0.2187532451857941
5 :0.2637754478952221  2 :0.6445273453144537  0 :0.4630166785185348
6 :0.8650384424149676  3 :0.42932761976709255  1 :0.6024952385895536
7 :0.5702072705027322  2 :0.86456124269257
8 :0.9452864187437506  0 :0.8961825862332892  5 :0.9299058855442358
1 :0.5956443807073741  9 :0.31509645530519625  6 :0.40326574227480094
10 :0.7910000224849713  7 :0.7017066711437372  2 :0.20056970253149548
6 :0.9338390704123928  3 :0.797053444490967  11 :0.7191822139832875
4 :0.7500681437013168  12 :0.5486221194511974  9 :0.25413610146892474
13 :0.8647843756083231  5 :0.8896910556803207  8 :0.4952122733888106
14 :0.5997502519024634  6 :0.5800735782304424  9 :0.7796297161425758
15 :0.3166804339669712  10 :0.14817882621967496  7 :0.8363991936747263
13 :0.5380334165340379  16 :0.8450927265651617  8 :0.5238810833905587
17 :0.5983997022381085  9 :0.7870744571266874  12 :0.738310558943156
10 :0.8801737147065481  15 :0.6153113201667844  18 :0.2663754517229303
19 :0.9069409600272764  11 :0.7381164412958352  14 :0.5723418590602954
```

```
20 :0.1541384547533948  17 :0.3985282545552262  12 :0.29468967639003735
21 :0.7576872377752496  13 :0.4858285745038984  16 :0.28762266137392745
17 :0.6628790185051667  22 :0.9203623808816617  14 :0.8394013782615275
18 :0.6976948178131532  15 :0.4893608558927002  23 :0.5604145612239925
24 :0.8901867253885717  21 :0.561967244435089  16 :0.35835658210649646
17 :0.8438726714274797  20 :0.3311114339467634  25 :0.7968809594947989
21 :0.6354858042070723  23 :0.33441278736675584  18 :0.43027465583738667
27 :0.8914256412658524  22 :0.8708278171237049  19 :0.4478162295166256
20 :0.35178269705930043  25 :0.2054048551310126
21 :0.6830700124292063  24 :0.3148089827888376  26 :0.5449034876557145
27 :0.2104213229517653  22 :0.8159939689806697  25 :0.4989269533310492
26 :0.44272335750313074  23 :0.4353604625664018
```

Ad. 2 W przypadku generowania grafu na podstawie parametrów wejściowych powinne zostać podane następujące dane w odpowiedzniej kolejności:

- Liczba wierszy grafu(int)
- Liczbę kolumn grafu(int)
- Zakres zmienności losowanych wag – 2 liczby(double) rzeczywiste nieujemne w porządku rosnącym (wartości będą losowane w zakresie od pierwszej liczby do drugiej)

Jeżeli nie zostanie podany zakres zmienności losowanych wag przyjęty zostanie przedział domyślny - <0,1>.

Przykładowo:

- 7 4 0 2 – podanie takich argumentów wejściowych oznacza wygenerowanie grafu o 7 wierszach, 4 kolumnach i wylosowanych wagach w przedziale <0,2>
- 2 3 – podanie takich argumentów wejściowych oznacza wygenerowanie grafu o 2 wierszach, 3 kolumnach oraz domyślnym przedziale wartości <0,1>

Dane powinne zostać podane w odpowiednim formacie. Żeby program był w stanie znaleźć rozwiązanie należy podać liczby w zakresie:

- Liczbę wierzchołków labiryntu (int) -liczba całkowita od 2 do 10000
- Liczbę krawędzi(int) - liczba całkowita od 1 do 10000
- Lewa granica zakresu(double) (w przypadku losowania wag krawędzi) - od 0 do 10000
- Prawa granica zakresu(double) (w przypadku losowania wag krawędzi) - od 0 do 10000; Przy tym lewa granica >= prawa granica

2.2.2 Dane wyjściowe

1. Zapis grafu w przypadku jego generacji

   Dane wyjściowe zapisywane będą w takim samym formacie jak dane wejściowe w przypadku wczytywania grafu – plik tekstowy (Graph.txt).

   Struktura pliku:

   - Pierwszy wiersz pliku zawierać będzie kolejno: liczbę kolumn(int) i liczbę wierszy(int) grafu.
   - Kolejne wiersze pliku dotyczyły będą kolejno wierzchołków od 0 do n(int), gdzie n(int) to liczba wierzchołków pomniejszona o 1 (numeracja wierzchołków zaczyna się od 0). Każdy z wierszy zawierać będzie listę par (numer wierzchołka(int) :wagę krawędzi(double), gdzie separatorem jest „:", a liczby zmiennoprzecinkowe wyrażające wagi zapisywane są przy pomocy znaku kropki (zamiast przecinka).

   Przykładowo:

```
2 3
0 :0.2637754478952221  1 :0.6445273453144537  3 :0.4630166785185348
0 :0.8650384424149676  2 :0.42932761976709255  4 :0.6024952385895536
1 :0.9452864187437506  5 :0.8961825862332892
0 :0.5956443807073741  4 :0.31509645530519625
3 :0.7910000224849713  5 :0.7017066711437372  1 :0.20056970253149548
4 :0.9338390704123928  1 :0.797053444490967  2 :0.7191822139832875
```

2. Zapis wyniku działania programu - znalezienie najkrótszej ścieżki.

   Dane wyjściowe z najkrótszą ścieżką do wyjścia zapisywane będą w pliku tekstowym (Result.txt).

   Plik wyjściowy zawierać będzie:

   - sumę wag krawędzi(double), czyli całkowitą "długość" ścieżki
   - listę przejść między komórkami reprezentowanymi w postaci par połączeń wierzchołekPoczątkowy wierzchołekDocelowy wagaKrawędziMiędzyWierzchołakmi.

   Przykładowo:

```
1.3375
0 1 0.3911
1 4 0.3212
4 5 0.6252
```

2.3 Opis Algorytmów

Program SUGP wykorzystuje następne algorytmy:

- Algorytm przeszukiwania grafu wszerz BFS
- Agorytm Dijkstry

2.3.1 Algorytm BFS

Algorytm BFS (ang. breadth-first search) przeszukiwania grafu wszerz zostanie wykorzystany w celu sprawdzenia spójności grafu.

Przechodzenie grafu wszerz polega na odwiedzeniu kolejnych węzłów leżących na następnych poziomach. Kolejno odwiedzony zostaje korzeń drzewa, następnie jego synowie itd. Operacja ta jest kontynuowana, aż do przejścia przez wszystkie węzły drzewa. Takie przejście przez węzły drzewa oznacza konieczność zapamiętywania ich wskazań w kolejce. Kolejka umożliwia odczytywanie zawartych w niej elementów w tej samej kolejności, w jakiej zostały do niej wstawione. Działanie to przypomina bufor opóźniający.

Złożoność pamięciowa algorytmu uzależniona jest od tego w jaki sposób reprezentowany jest graf wejściowy. W przypadku listy sąsiedztwa dla każdego wierzchołka przechowywana jest lista wierzchołków osiągalnych bezpośrednio z niego. W tym wypadku złożoność pamięciowa wynosi O(|V| + |E|), gdzie |V| to liczba węzłów, a |E| to liczba krawędzi w grafie, odpowiadająca sumie wierzchołków znajdujących się na listach sąsiedztwa.

Złożoność czasowa przeszukiwania wszerz wynosi O(|V| + |E|), gdzie |V| to liczba węzłów, a |E| to liczba krawędzi w grafie. Wynika to z faktu, że w najgorszym przypadku przeszukiwanie wszerz musi przejść wszystkie krawędzie prowadzące do wszystkich węzłów.

Działanie programu rozpoczyna się od odwiedzenia wierzchołka startowego. Następnie odwiedzamy wszystkich jego sąsiadów. Dalej odwiedzamy wszystkich nieodwiedzonych jeszcze sąsiadów sąsiadów itd. W programie zostanie użyty dodatkowy parametr *visited*, aby uniknąć zapętlenia w przypadku napotkania cyklu.
Parametr *visited[ ]* określa stan odwiedzin wierzchołka. Wartość *false* posiada wierzchołek jeszcze nie odwiedzony, a wartość *true* - wierzchołek, który algorytm już odwiedził. Parametry te są zebrane w tablicy logicznej *visited [ ]*, która posiada tyle elementów, ile jest wierzchołków w grafie. Element v*isited[i]* odnosi się do wierzchołka grafu o numerze i.

Algorytm BFS dla macierzy sąsiedctwa - lista kroków:

```
Wejście:
    v - numer wierzchołka startowego, v  C.
    visited - wyzerowana tablica logiczna n elementowa
              z informacją o odwiedzonych wierzchołkach.
```

```
    graf - zadany w dowolnie wybrany sposób, algorytm
           tego nie precyzuje.

Wyjście:
    Przetworzenie wszystkich wierzchołków w grafie.
Zmienne pomocnicze:
    Q         -         kolejka.
    u         -         wierzchołek roboczy, u   C.


K01:        Q.push ( v )
#w kolejce umieszczamy numer wierzchołka                                    s
K02:        visited [ v ] ← true

K03:        Dopóki Q.empty( ) = false,
#tutaj jest pętla główna  algorytmu BFS

K04:            v  ← Q.front( )
#odczytujemy z kolejki numer wierzchołka

K05:             Q.pop( )
#odczytany numer usuwamy z kolejki

K06:            Przetwórz wierzchołek v
#tutaj wykonujemy operacje na wierzchołku v

K07:            Dla i = 0, 1, ..., n - 1:
                wykonuj kroki K08...K10
#przeglądamy wszystkich sąsiadów v

K08:                Jeśli ( A [ v ][ i ] = 0 ) v ( visited [ i ] = true ),
                    to następny obieg pętli K07
#szukamy nieodwiedzonego sąsiada

K09:                Q.push ( i )
#numer sąsiada umieszczamy w kolejce

K10:                visited [ i ] ← true
#i oznaczamy go jako odwiedzonego
K11:        Zakończ
```

2.3.2 Algorytm Dijkstry

Przez najkrótszą ścieżkę (ang. shortest path) łączącą w grafie dwa wybrane wierzchołki będziemy rozumieli ścieżkę o najmniejszym koszcie przejścia, czyli o najmniejszej sumie wag tworzących tę ścieżkę. Algorytm Dijkstry pozwala

znaleźć koszty dojścia od wierzchołka startowego v do każdego innego wierzchołka w grafie (o ile istnieje odpowiednia ścieżka). Dodatkowo wyznacza on poszczególne ścieżki. Zasada pracy jest następująca:

Tworzymy dwa zbiory wierzchołków Q i S. Początkowo zbiór Q zawiera wszystkie wierzchołki grafu, a zbiór S jest pusty. Dla wszystkich wierzchołków u grafu za wyjątkiem startowego v ustawiamy koszt dojścia d (u) na nieskończoność. Koszt dojścia d (v) zerujemy. Dodatkowo ustawiamy poprzednik p (u) każdego wierzchołka u grafu na niezdefiniowany. Poprzedniki będą wyznaczały w kierunku odwrotnym najkrótsze ścieżki od wierzchołków u do wierzchołka startowego v. Teraz w pętli, dopóki zbiór Q zawiera wierzchołki, wykonujemy następujące czynności:

- Wybieramy ze zbioru Q wierzchołek u o najmniejszym koszcie dojścia d (u).
- Wybrany wierzchołek u usuwamy ze zbioru Q i dodajemy do zbioru S.
- Dla każdego sąsiada w wierzchołka u, który jest wciąż w zbiorze Q, sprawdzamy, czy
  - d (w) > d (u) + waga krawędzi u–w.
  - Jeśli tak, to wyznaczamy nowy koszt dojścia do wierzchołka w jako:
  - d (w) ← d (u) + waga krawędzi u–w.
  - Następnie wierzchołek u czynimy poprzednikiem w: p (w) ← u.

Agorytm Dijkstry - lista kroków:

```
Wejście:
    n- liczba wierzchołków w grafie, n   C.
    graf - zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje.
    Definicja grafu powinna udostępniać wagi krawędzi.
    v - wierzchołek startowy, v   C.

Wyjście:
    d - n  elementowa tablica z kosztami dojścia od wierzchołka v  do wierzchołka
    i-tego wzdłuż najkrótszej ścieżki. Koszt dojścia jest sumą wag krawędzi,
    przez które przechodzimy posuwając się wzdłuż wyznaczonej najkrótszej ścieżki.
    p - n  elementowa tablica z poprzednikami wierzchołków na wyznaczonej najkrótszej
    ścieżce. Dla i-tego wierzchołka grafu p [ i  ] zawiera numer
    wierzchołka poprzedzającego na najkrótszej ścieżce

Zmienne pomocnicze:
    S - zbiór wierzchołków grafu o policzonych już najkrótszych ścieżkach
    od wybranego wierzchołka v.
    Q - zbiór wierzchołków grafu, dla których najkrótsze ścieżki nie zostały
    jeszcze policzone.
    u, w - wierzchołki, u, w   C.
```

```
K01:        S  ← ∅        zbiór S
#ustawiamy jako pusty

K02:        Q  ← wszystkie wierzchołki grafu

K03:        Utwórz n  elementową tablicę d
#tablica na koszty dojścia

K04:        Utwórz n  elementową tablicę p
#tablica poprzedników na ścieżkach

K05        Tablicę d  wypełnij największą wartością dodatnią

K06:        d [ v  ] ← 0
#koszt dojścia do samego siebie jest zawsze zerowy

K07:        Tablicę p  wypełnij wartościami -1
#-1 oznacza brak poprzednika

K08:        Dopóki Q  zawiera wierzchołki,
            wykonuj kroki K09...K12

K09:        Z Q  do S  przenieś wierzchołek u o najmniejszym d [ u ]

K10:            Dla każdego sąsiada w  wierzchołka u :
                wykonuj kroki K11...K12
#przeglądamy sąsiadów przeniesionego wierzchołka

K11:                Jeśli w  nie jest w Q,
        to następny obieg pętli K10
#szukamy sąsiadów obecnych w Q
K12:                Jeśli d [ w  ] > d [ u  ] + waga krawędzi u-w,
                to:
                    d [ w  ] ← d [ u  ] + waga krawędzi u-w
                    p [ w  ] ← u
#sprawdzamy koszt dojścia.
#Jeśli mamy niższy, to modyfikujemy koszt i zmieniamy poprzednika w na u

K13:        Zakończ
```

3 Kod programu

3.1 Koncepcja nazewnictwa

Pisanie kodu zespołowo wymaga przyjęcia wspólnej koncepcji nazewniczej w celu uzyskania przejrzystego i czytelnego kodu. Cały kod w obrębie projektu powinien być napisany w sposób zapewniający zachowanie następujących zasad:

- Wszystkie nazwy są w języku angielskim,
- Nazwy zmiennych i metod zaczynają się z małych liter, a każde kolejne słowo, które zawierają rozpoczyna się z wielkiej litery. Nazwy powinny być jasno utożsamiane z obiektem, którego dotyczą, z zachowaniem adekwatnego poziomu abstrakcji.
- Każde zagłębienie w kodzie jest symbolizowane przez rosnący akapit
- Znaki rozpoczynające dany blok instrukcji znajdują się w jednej linii z nazwą metody, instrukcji warunkowej lub pętli, jeśli to możliwe
- Adnotacje znajdują się w oddzielnym wierszu, bezpośrednio poprzedzającym metodę, do której się odnoszą

3.2 Sposób wprowadzania zmian

Każdy członek zespołu dokonuje zmian w obrębie kodu, za który odpowiada lub kodu, za który nie odpowiada po uprzedniej konsultacji z autorem. Niemniej jednak, każda wprowadzana zmiana powinna zachowywać zasady przyjęte przy procesie tworzenia i nie zaburzać czytelności kodu.

3.3 System kontroli wersji

System kontroli wersji jest narzędziem używanym przez cały proces tworzenia oprogramowania. Każda zmiana dokonywana przez osobę z zespołu jest umieszczana w repozytorium. Proces ten przebiega przy użyciu systemu kontroli wersji "GitHub". W tym celu utworzono repozytorium do którego odnośnik znajduje się poniżej.

`https://github.com/LidiaLachman/SUGP`

3.4 Struktura plików projektu

Projekt zawierał będzie plik główny programu **main.c** napisany w języku C. W projekcie umieszczone zostaną funkcje odpowiadające między innymi za wczytywanie danych z wejścia do odpowiednich struktur, generowanie losowych grafów oraz implementację algorytmów poszukiwania najkrótszych ścieżek. Projekt będzie również zawierał odpowiednie pliki nagłówkowe, które umożliwią

włączanie wybranych funkcji do programu głównego. Wszelkie dane - generowane przez program bądź wczytywane do niego, znajdą się w katalogu głównym projektu.

4 Komunikaty błędów

Program obsługuje wiele rodzajów błędów - testy zostaną przeprowadzone poprzez uruchomienie programu z odpowiednio przygotowanymi, niepoprawnymi argumentami wywołania i formatowaniem pliku wejściowego, tak aby otrzymać każdy z komunikatów błędów z osobna.

Program powinien wyświetlić następujące komunikaty w razie wystąpienia problemów związanych z działaniem programu:

- Can't open file – Jeżeli program nie jest w stanie otworzyć pliku, nie może znaleźć pliku o podanej nazwie albo plik nie istnieje.
- Wrong data format in file - Jeśli program nie może poprawnie odczytać danych z pliku wejściowego, podanego przez użytkownika, dane zapisane w złym formacie lub podane są niepoprawne wartości. Dane wejściowe powinne być zgodne z informacją zawartą w punkcie XX
- Invalid program invocation arguments - W przypadku, gdy zostały podane złe argumenty lub ich liczba nie jest odpowiednia. Argumenty wejściowe powinny być podane w sposób zaprezentowany w punkcie XX.
- Inconsistent graph - Po sprawdzeniu spójności grafu powinien się wyświetlić ten komunikat, jeżeli podany albo wygenerowany graf jest niespójny, wtedy program nie jest w stanie znaleźć najkrótszą ścieżkę.

Bibliografia

[1] `https://eduinf.waw.pl/inf/alg/001_search/0126.php`

[2] `https://eduinf.waw.pl/inf/alg/001_search/0138.php`

[3] `https://pl.wikipedia.org/wiki/Przeszukiwanie_wszerz`