

## Spis treści

<b>1</b>	<b>Opis ogólny</b>	<b>1</b>
1.1	Nazwa programu . . . . .	1
1.2	Przeznaczenie dokumentu . . . . .	1
1.3	Cel programu . . . . .	1
1.4	Środowisko powstania . . . . .	1
1.5	Scenariusz działania programu . . . . .	2
<b>2</b>	<b>Budowa Programu</b>	<b>7</b>
2.1	Opis klas . . . . .	7
2.2	Struktura danych . . . . .	9
2.2.1	Dane wejściowe . . . . .	9
2.2.2	Dane wyjściowe . . . . .	11
2.3	Opis Algorytmów . . . . .	12
2.3.1	Algorytm BFS . . . . .	12
2.3.2	Algorytm Dijkstry . . . . .	14
<b>3</b>	<b>Kod programu</b>	<b>16</b>
3.1	Koncepcja nazewnictwa . . . . .	16
3.2	Sposób wprowadzania zmian . . . . .	16
3.3	System kontroli wersji . . . . .	16
<b>4</b>	<b>Komunikaty błędów</b>	<b>16</b>

# 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 (Java) jest dokumentem omawiającym opis szczegółowy projektu SUGP oraz związany z nim zakres prac, funkcje programu, interakcje zachodzące między systemem a użytkownikiem i instrukcje użytkownika.

## 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.

Program zostanie zwizualizowany za pomocą interfejsu graficznego. Interfejs graficzny będzie spełniał następujące funkcjonalności: wizualizacja grafu, wybór za pomocą myszki węzłów, między którymi zostanie poprowadzona najkrótsza ścieżka, wyświetlenie najkrótszej ścieżki.

## 1.4 Środowisko powstania

Program „SUGP” jest napisany w obiektowym języku programowania Java. Zintegrowanym środowiskiem programistycznym używanym w procesie tworzenia aplikacji jest „NetBeans IDE 13” . Dokładnie wersje środowiska programistycznego:

Element środowiska	Wersja
Język programowania	Java 18.0.1.1
Java Development Kit	18.0.1.1
NetBeans	13

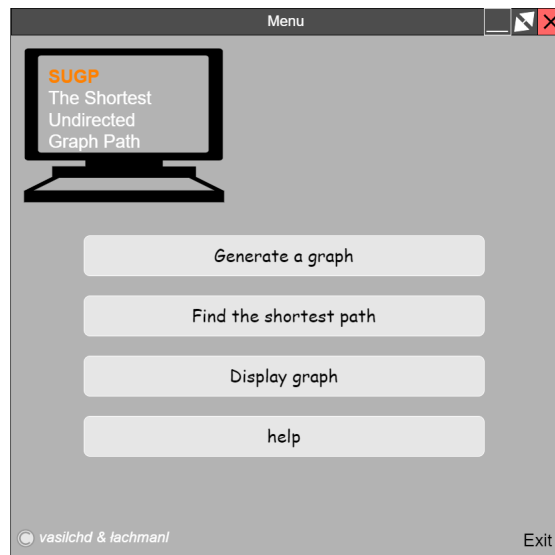
Aplikacja została utworzona jako standardowa aplikacja Java SE.

## 1.5 Scenariusz działania programu

Po uruchomieniu programu zostanie wyświetlone menu główne, w którym użytkownik może wybrać jedną z kilku możliwych opcji działania programu.

Znaczenie poszczególnych opcji :

- Generate a graph - Generowanie grafu
- Find the shortest path - Znalezienie najkrótszej ścieżki
- Display graph - Wyświetlenie grafu



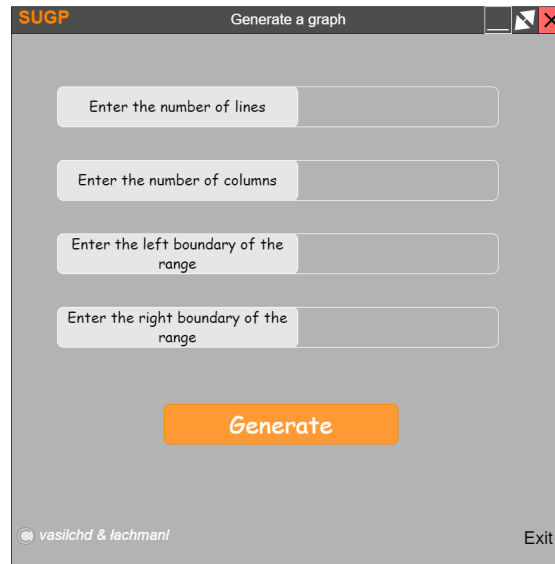
Rysunek 1: Menu główne - wersja poglądowa

1. W przypadku wybrania opcji «Generate a graph» pojawi następujące okienko z zapytaniem o dane wejściowe dla generacji grafu (format danych wejściowych został opisany w p.2.3.1 1)

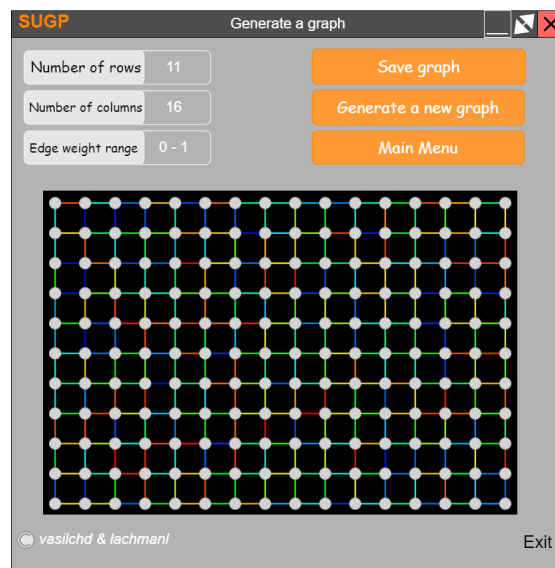
Po wprowadzeniu wszystkich danych trzeba kliknąć przycisk «Generate». Po udanej generacji użytkownik zobaczy wygenerowany graf w postaci rysunku :

Dalej użytkownik ma do wyboru następujące opcje:

- Save graph - zapisuje wygenerowany graf domyślnie do pliku graph.txt
- Generate a new graph - generuje nowy graf z nowymi danymi wejściowymi (czyli wraca do poprzedniego okienka)
- Main Menu - wraca do menu głównego




Rysunek 2: Generate a graph - wersja poglądowa



Rysunek 3: Przykładowy graf - wersja poglądowa, źródło: ISOD

2. W przypadku wybrania opcji «Find the shortest path» użytkownik również zostanie poproszony o podanie danych wejściowych dotyczących ta-

kich informacji jak plik tekstowym z grafem oraz numery wierzchołków startowego i końcowego (format danych wejściowych został opisany w p.2.3.12).



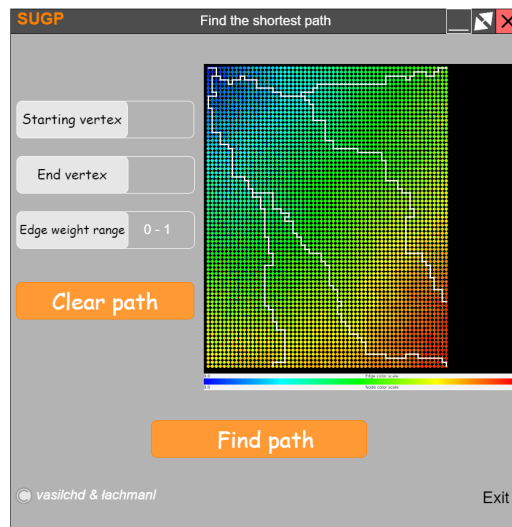
Rysunek 4: Wprowadź dane dla znalezienia najkrótszej ścieżki - wersja poglądowa

Po wprowadzeniu wszystkich danych należy kliknąć przycisk «Find path». Jeżeli podany w pliku graf jest spójny, to program wyświetli graf oraz wyznaczy na nim znaną ścieżkę wraz z jej długością, jeżeli nie to zostanie wyświetlony komunikat o błędzie (p.4)

W przypadku zaistnienia potrzeby wyznaczenia innej ścieżki można dokonać zmiany punktów na dwa sposoby:

- Sposób 1 - należy kliknąć przycisk «Clear path». Następnie wprowadzić nowe dane w okienkach «Starting vertex» oraz «End vertex» oraz zatwierdzić klikając przycisk «Find path».
- Sposób 2 - poprzez kliknięcie myszką w odpowiednie punkty na rysunku w odpowiedniej kolejności, gdzie pierwszy kliknięty punkt zostanie wierzchołkiem startowym, a drugi - końcowym, wtedy w okienkach «Starting vertex» oraz «End vertex» wartości punktów zostaną przypisane automatycznie.

3. W przypadku opcji «Display graph» należy podać ścieżkę do pliku z grafem (format danych wejściowych został opisany w p.2.3.13). Następnie kliknąć przycisk «Display», a na ekranie zostanie wyświetlony narysowany graf.



Rysunek 5: Wprowadź dane dla znalezienia najkrótszej ścieżki - wersja pogłębiona



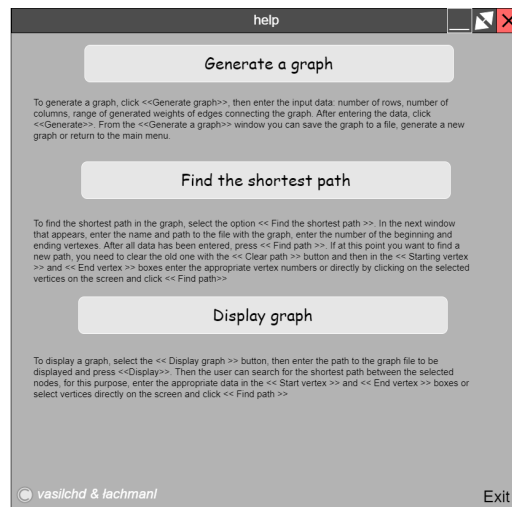
Rysunek 6: Wprowadź dane dla znalezienia najkrótszej ścieżki - wersja pogłębiona

Po wyświetleniu grafu (Rysunek 5) użytkownik może wyszukać w tym grafie najkrótszą ścieżkę poprzez wprowadzenie odpowiednich punktów w miejsca «Starting vertex» oraz «End vertex». Innym wariantem wprowadzenia

dziania punktów jest kliknięcie wybranych miejsca na wyświetlonym grafie. Należy to wykonać w odpowiedniej kolejności: pierwszy kliknięty wierzchołek - wierzchołek startowy, drugi - końcowy. Numery tych wierzchołkom zostaną automatycznie przepisane w okienka «Starting vertex» oraz «End vertex». Dalej należy kliknąć przycisk «Find path».

4. Pozostałe opcje

- Main Menu - Pozwala użytkownikowi wrócić do menu głównego.
- Exit - Kończy działanie programu.
- help - Wyświetla okienko ze strzeszczoną instrukcją.



Rysunek 7: Help - wersja poglądowa

## 2 Budowa Programu

### 2.1 Opis klas

W tym punkcie zostaną opisane klasy zaimplementowane w naszym projekcie.

#### Main

Jedyna klasa, z którą w bezpośrednią interakcję wchodzi użytkownik. Odpowiada za sterowanie przekazywaniem informacji o wciśniętych klawiszach do kolejnych klas.

Metody klasy **Main**:

1. *public static void main(String[] args)* - Metoda główna programu. Od jej wywołania rozpoczyna się działanie programu.

#### Panel

Klasa odpowiedzialna za implementację oraz obsługę graficznego interfejsu użytkownika. Zaimplementowana została tutaj również obsługa zdarzeń związanych z interakcją użytkownika z wykorzystaniem myszy.

Metody klasy **Panel**:

1. *public static void draw(Graph g, int start, int finish)* - Metoda pozwalająca na wyrysowanie grafu na Panelu w celu wizualizacji najkrótszej drogi. Jako argumenty przyjmuje obiekt klasy Graph oraz węzły grafu - startowy oraz docelowy (końcowy).

#### Algorithms

Klasa zawiera implementacje algorytmów wykorzystanych w programie, czyli Algorytm BFS oraz Algorytm Dijkstry.

Metody klasy **Algorithms**:

1. *public boolean BFS(Graph graph, int start)* - Metoda pozwalająca na sprawdzenie spójności grafu. W przypadku, gdy graf jest spójny - zwraca *true*, w przeciwnym razie - *false*. Jako argumenty wejściowe przyjmuje obiekt klasy Graph oraz węzeł początkowy.
2. *public int[] dijkstra(Graph graph, int start, int finish)* - Metoda zawiera implementację algorytmu Dijkstry (poszukiwanie najkrótszej ścieżki w grafie między dwoma zadanymi wierzchołkami). Zwraca tablicę wierzchołków przez które należy przejść aby koszt drogi był najmniejszy. Jako argumenty wejściowe przyjmuje obiekt klasy Graph, numery węzłów - startowego i końcowego.



## Graph

Przechowuje wszystkie informacje o grafie oraz odpowiada za jego wyświetlenie na ekranie, jego budowę i zapis do pliku.

Metody klasy **Graph**:

1. *public void print()* - Metoda pozwala na wypisanie wierzchołków.
2. *public Graph(int rows, int cols)* - konstruktor klasy Graph. Jako argumenty przyjmuje liczbę wierszy i kolumn macierzy sąsiedztwa grafu.
3. *public Graph(int rows, int cols, ArrayList<Node>[] field)* - konstruktor klasy Graph wywoływany w przypadku podania przez użytkownika zmiennej field.
4. *public void printGraphToFile(String filename)* - metoda pozwala na zapis grafu do pliku. Jako argument wejściowy przyjmuje ścieżkę do pliku (lub jego nazwę).

## Generator

Generuje wagi dla krawędzi grafu oraz sam graf.

Metody klasy **Generator**:

1. *public double generateValue(double left, double right)* - Metoda pozwalająca na generowanie losowych wag krawędzi z przedziału podanego jako argumenty wejściowe (left oraz right).
2. *public Graph generateGraph(int rows, int cols, double left, double right)* - Metoda pozwalająca na generowanie grafu. Jako argumenty przyjmuje liczbę wierszy oraz kolumn (rows, cols) oraz zakres zmienności wag (left, right) generowanych z wykorzystaniem funkcji generateValue() opisanej powyżej.

## Node

Jednostka reprezentująca połączenie wierzchołka i krawędzi prowadzącej do niego.

Metody klasy **Node**:

1. *public Node()* - Konstruktor klasy Node.
2. *public Node(int destination, double weight)* - Konstruktor klasy Node wraz z argumentami - destination (numer wierzchołka do którego możemy trafić z bieżącego) oraz weight (dystans do bieżącego wierzchołka).

## Reader

Odpowiada za prawidłowy odczyt grafu z pliku wejściowego. Zawiera również funkcję pozwalającą na oczyszczanie odczytywanych linii z niepożądanych znaków.

Metody klasy **Reader**:

1. *private String normaliseStr(String now)* - Metoda pozwala na oczyszczenie ciągu znaków z niepożądanych elementów. Jako argument wejściowy przyjmuje ciąg znaków now, a zwraca oczyszczony ciąg znaków.
2. *public Graph readGraphFromFile(String filename)* - Metoda pozwala na odczyt grafu z pliku. Jako argument przyjmuje ścieżkę do pliku, a zwraca obiekt typu Graph.

## 2.2 Struktura danych

### 2.2.1 Dane wejściowe

Dane wejściowe różnią się w zależności od wybranych opcji z menu głównego. Dane wejściowe w przypadku wyboru konkretnej opcji:

1. Po wyborze opcji «*Generate a graph*».

Wymagane dane wejściowe do podania przez użytkownika:

- liczba wierszy (*Enter the number of rows*)
- liczba kolumn (*Enter the number of columns*)
- lewa granica zakresu generowanych wag (*Enter the left boundary of the range*)
- prawa granica zakresu generowanych wag (*Enter the right boundary of the range*)

W przypadku niepodania lewej i prawej granicy zakresu generowanych losowo wag program przyjmie domyślny przedział  $<0,1>$ .

2. Po wyborze opcji «*Find the shortest path*».

Wymagane dane wejściowe do podania przez użytkownika:

- nazwa pliku tekstowego (\*.txt) zawierającego graf (*Enter the name of the file with the graph*)
- ścieżka do tego pliku (*Enter the graph filepath*)
- numer wierzchołka startowego (*Enter the start vertex*)
- numer wierzchołka końcowego (*Enter the end vertex*)

3. Po wyborze opcji «*Display a graph*».

Wymagane dane wejściowe do podania przez użytkownika:

- nazwa pliku tekstowego (\*.txt) zawierającego graf (postać pliku identyczna jak w przypadku opcji «*Find the shortest path*») (*Enter the graph filepath*)

Po kliknięciu przycisku «*Display*» na ekranie zostaje wyświetlony graf. Obok wyświetlonego grafu użytkownik ma możliwość wprowadzenia punktów: początkowego i końcowego, dla których zostanie wyznaczona i zwiualizowana najkrótsza ścieżka. Dane wejściowe:

- numer wierzchołka startowego (*Enter the start vertex*)
- numer wierzchołka końcowego (*Enter the end vertex*)

Aby wyznaczyć najkrótszą ścieżkę użytkownik może również wybrać punkty (początkowy i końcowy) poprzez kliknięcie ich na grafie w kolejności: wierzchołek startowy, wierzchołek końcowy. W takim przypadku pola *Enter the start vertex* oraz *Enter the end vertex* zostaną wypełnione automatycznie.

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

- Liczbę wierzchołków(int) -liczba całkowita od 2 do 250000
- Liczbę krawędzi(int) - liczba całkowita od 1 do 250000
- Numer wierzchołka startowego(int) - liczba całkowita od 1 do liczby określającej ilość wierzchołków - 1
- Numer wierzchołka końcowego(int) - liczba całkowita od 2 do liczby określającej ilość wierzchołków
- Lewa granica zakresu(double) - od 0 do 10000
- Prawa granica zakresu(double) - od 0 do 10000; Przy tym lewa granica powinna mieć większą wartość niż prawa granica

Ad. 2 W przypadku wyboru opcji «*Find the shortest path*» użytkownik podaje nazwę pliku tekstowego (\*.txt).

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, gdzie n to liczba wierzchołków pomniejszona o 1 (numeracja wierzchołków zaczyna się od 0). Każdy z wierszy zawierać będzie listę par (numer wierzchołka : wagę krawędzi), gdzie separatorem jest „ : ”, a liczby zmienoprzecinkowe 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. 3 W przypadku wyboru opcji «*Generate a graph*» użytkownik podaje nazwę pliku tekstowego (\*.txt). Struktura pliku jest identyczna jak w punkcie 2.

## 2.2.2 Dane wyjściowe

1. Wybór opcji «*Save graph*» - 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 grafu(int).
- Kolejne wiersze pliku dotyczyły będą kolejno wierzchołków od 0 do  $n(\text{int})$ , gdzie  $n$  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. Pozostałe dane wyjściowe będą wyświetlane przez interfejs graficzny w zrozumiałym dla użytkownika sposób.

## 2.3 Opis Algorytmów

Program SUGP wykorzystuje następujące algorytmy:

- Algorytm przeszukiwania grafu wszerz BFS
- Algorytm 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 wszerek 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 *visited*[*i*] odnosi się do wierzchołka grafu o numerze *i*.

#### Algorytm BFS dla macierzy sąsiedztwa - 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*  
K02: *visited [ v ] ← true*

*startowego*

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  
*#przeładamy wszystkich sąsiadów v*

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

*#szukamy nieodwiedzzonego 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 wierzchołka  $u$ , który jest wciąż w zbiorze  $Q$ , sprawdzamy, czy
  - $d(w) > d(u) + \text{waga krawędzi } u-w$ .
  - Jeśli tak, to wyznaczamy nowy koszt dojścia do wierzchołka  $w$  jako:
    - $d(w) \leftarrow d(u) + \text{waga krawędzi } u-w$ .
    - Następnie wierzchołek  $u$  czynimy poprzednikiem  $w$ :  $p(w) \leftarrow u$ .

#### Algorytm Dijkstry - lista kroków:

Wejście:

$n$  - liczba wierzchołków w grafie,  $n \in \mathbb{C}$ .  
 graf - zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje.  
 Definicja grafu powinna udostępniać wagi krawędzi.  
 $v$  - wierzchołek startowy,  $v \in \mathbb{C}$ .

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 nazewnictwa 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 zawiera 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.

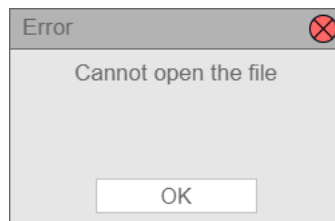
<https://github.com/LidiaLachman/SUGP>

## 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.

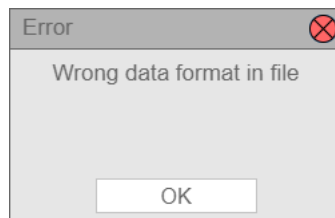
Program powinien wyświetlić następujące komunikaty w razie wystąpienia problemów związanych z działaniem programu:

- Can't open file – Jeżeli program nie jest w stanie otworzyć pliku, nie może znaleźć pliku o podanej nazwie albo plik nie istnieje.



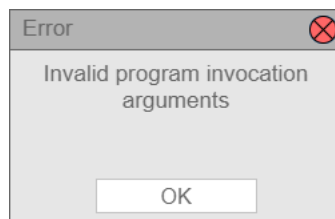
Rysunek 8: Cannot open the file - wersja poglądowa

- 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 powinny być zgodne z informacją zawartą w punkcie 2.3.1



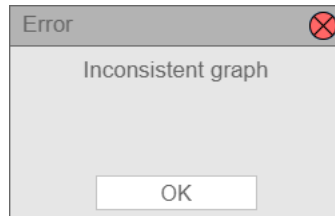
Rysunek 9: Wrong data format in file - wersja poglądowa

- 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 2.3.1.



Rysunek 10: Invalid program invocation arguments - wersja poglądowa

- 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.



Rysunek 11: Inconsistent graph - wersja poglądowa

## Bibliografia

- [1] [https://eduinf.waw.pl/inf/alg/001\\_search/0126.php](https://eduinf.waw.pl/inf/alg/001_search/0126.php)
- [2] [https://eduinf.waw.pl/inf/alg/001\\_search/0138.php](https://eduinf.waw.pl/inf/alg/001_search/0138.php)
- [3] [https://pl.wikipedia.org/wiki/Przeszukiwanie\\_wszerz](https://pl.wikipedia.org/wiki/Przeszukiwanie_wszerz)