

Politechnika Częstochowska
Wydział Elektryczny

Mateusz Iwańczak

Nr albumu 123167

Implementacja metody Dijkstry w języku C++.

Praca dyplomowa inżynierska

Kierunek: Informatyka, studia stacjonarne

Specjalność: Technologie internetowe i techniki multimedialne

Promotor:

Prof. dr hab. Krzysztof Sokalski

Częstochowa, 2017/2018

Spis treści

1. Wstęp	3
2. Cel i zakres pracy	4
3. C++	5
3.1. Historia języka	5
3.2. Filozofia programowania	7
3.3. Pochodzenie	7
3.4. Przenośność i standardy C++	8
3.5. Tworzenie programu	9
4. Problematyka metody Dijkstry	13
4.1. Zasadnicze oznaczenia i pojęcia	13
4.1.1. Termin grafu	13
4.1.2. Graf skierowany i nieskierowany	14
4.1.3. Odległości w grafach	15
4.1.4. Długości krawędzi	16
4.1.5. Relaksacja	17
4.2. Metody reprezentowania grafów w pamięci komputera	17
4.3. Istota algorytmu Dijkstry	20
5. Projekt aplikacji Algorytm Dijkstry	34
5.1. Założenia	34
5.2. Wymagania	34
5.3. Opis aplikacji	35
5.4. Testy programu	44
6. Podsumowanie	55
Literatura	56
Dodatek A Spis zawartości dołączonej płyty CD	56

1. Wstęp

Dzisiejsze czasy są pełne wzorców, które kierują się myślą, aby zrobić jak najwięcej w możliwie najkrótszym czasie. Stawiają ludzie na oszczędność pieniędzy, czasu, a co za tym idzie kosztów. Przemierzając świat, bądź po prostu wyjechać z określonego miejsca, obierają sobie za cel, który może być dobrowolny lub z góry narzucony, jakiejś konkretne miejsce, gdzie mają się udać. Sposób dotarcia do niego może być dowolny: pieszo, rowerem, motorowerem, motocyklem, konno, samochodem, środkiem komunikacji, a czasem nawet samolotem. Wszystkie wymienione przykłady transportu łączy to, jakim kosztem można zmienić położenie z punktu A do B. Pod pojęciem koszt można rozumieć – ten finansowy, lecz nie o nim mowa. Chodzi o ten, określający odległość punktu A do B w sieci, zawierającej zbiór punktów. Stawiając na oszczędność czasu można przyjąć, że osiągnięcie tego celu jest możliwe, podróżując najkrótszą drogą. Nie musi być jedna taka trasa, może być ich kilka. Dochodząc do pojęcia odległości, należy zwrócić jeszcze uwagę na to, przez jakie inne punkty będzie odbywała się podróż?

Odpowiedź na tą głęboką refleksję udziela w swoim algorytmie, inaczej ujmując metodzie holenderski informatyk Edsger Dijkstra, stwarzając schemat obliczający od takiego punktu A do punktu B najkrótszą trasę jedną lub więcej, a także koszt dotarcia do tego finalnego. Do zastosowania jego sposobu wystarczą tylko lokalizacje punktu początkowego A i punktu docelowego B, liczbę punktów tworzących sieć połączeń z tymi dwoma punktami, jako ostatnie potrzebne będą odległości pomiędzy poszczególnymi punktami w tejże sieci, mając na uwadze to, czy występują one pomiędzy sobą lub ich nie ma.

2. Cel i zakres pracy

Celem tej oto pracy jest implementacja algorytmu Dijkstry, służącego do odnajdywania najkrótszej ścieżki w grafie, uwzględniając dane wejściowe, w skład których wchodzi: liczba wierzchołków, długości krawędzi pomiędzy nimi, a także wierzchołkiem zaczynającym i końcowym w rozważaniu danego grafu podaną metodą.

Zakres pracy zawiera takie oto treści:

- Przegląd literatury w zakresie języka programowania ogólnego przeznaczenia C++ - jego historii, filozofii programowania, pochodzenia, przenośności i standardów, jak również tworzenia aplikacji.
- Przegląd piśmiennictwa dotyczącego algorytmu Dijkstry – przewodnich pojęć (graf, graf skierowany i nieskierowany, odległości w grafach, długość krawędzi), metody reprezentowania grafów w pamięci komputera, a na koniec sama istota metody.
- Opracowanie aplikacji, odzwierciedlającej metodę odnajdywania najkrótszej ścieżki w grafie.
- Testowanie programu, będącego implementacją metody Dijkstry.

3. C++

3.1. Historia języka

Przez obecne kilkadziesiąt lat dziedzina informatyki rozwijała się bardzo dynamicznie. Teraźniejszy sprzęt komputerowy ma możliwość przeprowadzać obliczenia sprawniej i magazynować dane w większym zakresie od komputera *mainframe*, który pochodzi z lat 60. zeszłego wieku. W tej kwestii nie podlega to żadnej dyskusji. Nie pozostały bez kroku naprzód języki komputerowe. Nie osiągnęły one w tym przypadku porównywalnej przemiany, lecz miały kluczowe znaczenie. Pecet, który jest obszerniejszy i bardziej wydolny, daje możliwość uruchomienia znaczniejszych i bardziej zawiłych programów, dających niełatwe zadanie w zarządzaniu nimi i renowacji.

Mówiąc o dwudziestym wieku, warto zwrócić uwagę na lata 70., gdzie drzwi do oprogramowania strukturalnego otworzyły użytkownikom przytoczone języki: C i Pascal. Tego rodzaju programy rozpoczęło nowy układ w informatyce, które okazało się niezbędne. Język C odgrywał rolę przyboru programowania strukturalnego, lecz nie było to jedyne jego zastosowanie. Dawał jednoczesną możliwość tworzenia zwięzłych, wydajnych aplikacji i gwarantował obsługę sprzętową, składającą się przykładowo z portów komunikacyjnych, jak również napędów dyskowych. Lata 80. ubiegłego wieku sprawiły tenże język fundamentalnym językiem programowania. Kolejny wzrost na znaczeniu otrzymał nowo powstały paradygmat programowania, rozwinęło się w owym czasie programowanie obiektowe, mające własny wkład w języku C++, a także SmallTalk.

W początkach lat 70. Dennis Ritchie prowadził warsztat nad zrobieniem systemu Unix. W owym przedsięwzięciu miał zapotrzebowanie na nowy język, umożliwiający projektowanie ścisłych i wydajnych programów, panujących nad sprzętem informatycznym.

Większość przypadków programistów korzystało wtedy z assemblera, zależnym od języka maszynowego danego sprzętu. Faktem jest, że tenże język należy do niskopoziomowych, a co za tym idzie działa na zasobach sprzętowych wprost. Rozwijając to stwierdzenie, można przytoczyć takie funkcje jak nawiązywanie do rejestrów procesora, czy unikatowego identyfikatora dla części jednostkowej sprzętu— adresu pamięci. Te informacje o nim przemawiają za tym, iż ma ścisły związek z procesorem

sprzętu. Warunkiem do przeniesienia go na inny podobny sprzęt stanowi odpisanie pełnego oprogramowania w niepodobny assembler.

System, który chciał stworzyć Dennis Ritchie, miał być wieloplatformowy, co z kolei było istotnym argumentem do skorzystania z języka wysokiego poziomu, skierowanego na rozwiązywanie trudności niż zarządzanie maszynami. Kompilator klaruje język wysokiego poziomu na maszynowy wybranego sprzętu, co z pewnością sprzyja korzystaniu z jednego programu na wielu rodzajach komputerów, zrobionych w języku wysokiego poziomu. Odwrotna sytuacja tyczy się kompilatorów. Informatyk był zainteresowany językiem efektywnym z prostym dostępem do komputera na zasadzie takiej, jak ma programowanie niskopoziomowe z cechami programowania wysokiego poziomu. Można wymienić tutaj: ogólność i przenośność.[4]

Ewolucja języka C++ zaczyna się od 1979 roku i języka C. W roku 1979 rozpoczęły się działania nad językiem C, mającym klasy i klasy pochodne, publiczną i prywatną kontrolę dostępu, konstruktory i destruktory, a także deklaracje funkcji, włączając w to weryfikację argumentów. Inicjalna biblioteka sterowała współbieżnością z pominięciem eksmisji i wytwarzania liczb losowych. Modyfikacja nazwy C, mającej klasy, na C++ odbyło się w 1984 roku, gdzie po drodze dorzucono funkcje wirtualne, przeciążanie funkcji i operatorów, referencje, strumień wejścia i wyjścia, jak również biblioteki liczb zespolonych. Dnia 14 października 1985 roku miało miejsce pierwsze wydanie C++, będące wersją komercyjną. W bibliotekę składały się strumienie wejścia i wyjścia, liczby zespolone i zadania. Załączono w 1997 roku do biblioteki zasadniczej szkielet STL, mający w sobie generalne kontenery i algorytmy. Wydano standard ISO C++ w 1998 roku, nazywany też ISO/IEC 14882-1998, czyli C++ 1998. Warsztat, którego kierunkiem przewodnim było poprawienie standardu, potocznie określanym C++0x miało miejsce w roku 2002. Poprawiony standard ISO C++ w 2003 roku zawierał dorzucone składniki do biblioteki standardowej takie jak: wyrażenie regularne, tablice mieszające, a także wskaźniki, służące do kierowania pamięcią. Koniec prac, które zbyły kierowane do zestawu elementów języka C++0x nastąpiło w 2009 roku. Powstały między innymi jednolita inicjacja, semantyka przenoszenia, zmienne argumenty szablonów, wyrażenia lambda, aliasy typów, model pamięci pod kątem współbieżności. Biblioteka standardowa została wzbogacona w wątki, blokady, a także większość elementów z raportu technicznego, wydanego sześć lat wcześniej. W 2011 roku formalnie potwierdzono standard ISO C++11, a rok później ukazała się jego pierwsza kompletna implementacja.[7]

3.2. Filozofia programowania

Programy komputerowe tworzą zwartą całość pomiędzy zagadnieniami algorytmu i dane. Rozpoczynając rozważania nad kierunkiem działania programowania, należy zacząć od ich definicji. Dane stanowią informacje, z których korzysta aplikacja i transformuje je. Z kolei algorytm tworzy sposoby pracy, charakteryzujące określoną aplikację. Język C jest w zbiorze języków proceduralnych, co oznacza zwrócenie szczególnej uwagi na algorytmiczny punkt widzenia w tworzeniu programu. Proceduralne programowanie bazuje na kreowaniu działań, wykonywanych przez sprzęt, potem zapisie ich, korzystając z wyselekcjonowanego języka. Aplikację tworzą składowe, stanowiące zbiór procedur postępowania przeznaczonych dla peceta czy laptopa. Procedury te skutkują zadowalającym efektem pracy. Język C daje pole do tego, by projektować, programowaniem strukturalnym. W tego typu programowaniu jest uszczuplona możliwość modyfikacji realizowanej ścieżki kodu. Nie ma tutaj swobodnego środowiska do przekształceń, lecz są do tego celu poszczególne konstrukcje takie jak:

- a) pętle (do ... while, for, while)
- b) instrukcja warunkowa – wyboru (if else)

Nowinką z zasad stało się też programowanie, określane mianem *od ogółu do szczegółu*. Język, który jest analizowany w tym podrozdziale, porcuje aplikacje na mniejsze części. Celem tego jest lepsza przystępność do ich zrozumienia. W przypadku powtarzającej się sytuacji skomplikowania, pojawia się motyw do kolejnego podzielenia do momentu, kiedy aplikacja będzie tworzyła zestaw znikomych modułów, mających prostotę. W metodzie przytoczonej chodzi o segregację aplikacji na odpowiedzialne za określone czynności funkcje.[4]

3.3. Pochodzenie

Przedstawiając pochodzenie języka C++, należy rozpocząć od tego, iż powstał w Bell Labs tak jak poprzednik - C – przez Bjarne Stroustrup. Nastąpiło to w latach 80. ubiegłego stulecia. Przeznaczeniem języka C++ było to, aby pozbawić konieczności programowania w assemblerze, C, a także innymi przykładami języków wysokiego poziomu ówczesnego okresu. Myślą przewodnią była łatwość i przyjemność z kreowania nowych aplikacji, czy aktualizacji

istniejących. Język miał być wygodny do wykorzystania. Nadrzędne stały się trudności, z jakimi mieli zmierzyć się programiści niż poprawność w teorii i szykowność. Stworzony na podstawie języka C - C++ - zrobiony był ze względu na skondensowanie, użyteczność do celu programowania systemowego, pospolitą osiągalność i korzenie z systemem Unix. Stroustrup rozszerzył język C o obiektowość, uogólnione programowanie bez modyfikacji podstawowej jego struktury, w wyniku tego C++ stanowi nadzbiór tegoż języka. Właściwie utworzony program w C jest tym samym odpowiedni z perspektywy trafności języka C++. Nie mają tu żadnego znaczenia pewne sprzeczności, biblioteki C może bez przeszkód wykorzystywać C++ w swoich aplikacjach. Biblioteki są zbiorami modułów, składających się ze sprawdzonego kodu, służącego do wykonania pewnych zadań, czego zaletą jest z pewnością oszczędność czasu, a co za tym idzie pracy.

Dwa znaki plus (++) mają swoje korzenie w obecnym C, gdzie jest operator inkrementacji, czyli dodanie wartości jeden do określonej zmiennej, co doskonale przemawia za tym, że C++ odzwierciedla rozbudowany język C. C++ jest językiem C z dorzuconą obiektowością, więc można stwierdzić na pierwsze spojrzenie, że można wykorzystywać jedynie cechy C, natomiast odrzucić wszystko, co ma obiektowość. To błędne stwierdzenie, którego urzeczywistnienie skutkuje olbrzymią stratą.

Szablony, dające szansę na programowanie uogólnione, zawarte przez Stroustrup'a stały się mechanizmem porównywalnym z obiektowością. Tezę, mówiącą o stawianiu wygody nadrzędną nad czystością teoretyczną, uzasadnia połączenie programowania obiektowego z ogólnym i tradycyjne nastawienie proceduralne.[4]

3.4. Przenośność i standardy C++

Przenośność i standardy mają fundamentalne znaczenie w języku C++. Spostrzeżenie przenośny jest stosowne, gdy po rekompilacji jest szansa na uruchomienie programu i zwyczajne korzystanie z niego.

Przeszkodą przenośności jest nieuchronnie sprzęt, kiedy aplikacja jest uzależniona od konfiguracji sprzętowej. Kolejną jest zróżnicowanie językowe, ponieważ nie ma spisane go żadnego standardu języka, określającego jego pracę. Jest to wyzwanie niecodzienne, trudne i niemożliwe. W 1990 roku powołany został komitet przez ANSI, mając cel stworzyć standard C++, a niewiele później doszła międzynarodowa organizacja standaryzacyjna ISO, powołując

do tego własny komitet. Doprowadziło to do połączenia starań z zamiarem określenia norm C++.

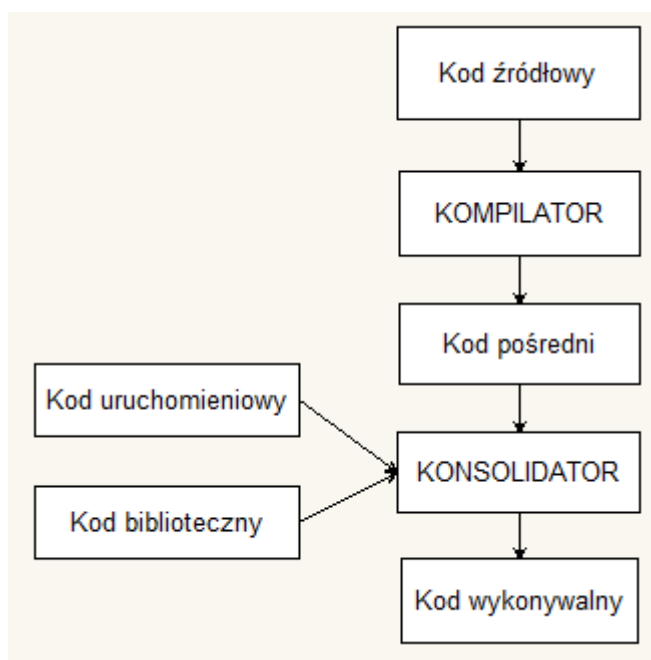
Wspomniane starania poskutkowały standardem międzynarodowym ISO/IEC 14882:1998, zatwierdzonym przez ISO, IEC i ANSI, określany obecnie C++98, precyzującym istniejące charakterystyki języka. Dał mu rozszerzenie w postaci wyjątków, RTTI (mechanizm identyfikacji czasu wykonania), STL (standardową bibliotekę szablonów) wraz z szablonami. Rok 2003 dołożył drugą wersję norm C++, precyzującą bardziej ubiegłego standardu, poprawiając literówki, usunięciem nieoczywistych stwierdzeń. Wersja jest określana jako ISO/IEC 14882:2003, zwyczajowo jest oznaczona C++03. Jest również używane oznaczenie łączne C++98/C++03, ponieważ sam język nie został w żaden sposób zmieniony. W sierpniu 2011 roku komitet ISO ustanowił nowy standard ISO/IEC 14882:2011, konwencjonalną nazwą był C++11, gdzie rozszerzono język i postawił sobie komitet zamiar usunięcia wszelkich niespójności i stworzeniem języka tego prostym do zastosowań i przyswojenia. Tenże standard korzysta ze standardu ANSI C, ponieważ C++ stanowi nadzbiór C. Puenta tegoż stwierdzenia niesie za sobą to, że poprawność aplikacji w C jest równocześnie tą samą opinią w przypadku C++. Standard ANSI C jest nie tyle, że definicją języka, lecz jego standardowej biblioteki, posiadającej wszelką jego implementację. Standard ten ma swoje zastosowanie w języku nie tylko C, ale w C++ jest używany.[4]

3.5. Tworzenie programu

Zasadnicze kroki kreowania programu po jego napisaniu można ująć w trzech punktach. Pierwszy - aplikacja jest tworzona dzięki edytorowi tekstowemu, potem zostaje zapisana w pliku. Następny - dokonuje się kompilacji kodu źródłowego, czyli uruchomienie takiej aplikacji, która przetłumaczy kod źródłowy na język wewnętrzny sprzętu, czyli maszynowy, zapisany zostaje do pliku, określanego mianem kodu wynikowego. Ostatni stanowi konsolidacja kodu wynikowego z uzupełniającymi zasobami. Przykładowo przeciętne programy korzystają z bibliotek, obejmujących kod wynikowy włącznie z paletą procedur, czyli funkcji, pokazujących informacje na ekranie lub wyliczających pierwiastek kwadratowy (drugiego stopnia) z danej wartości.

Termin konsolidacja obejmuje łączenie swojego kodu wynikowego z kodem wynikowym stosowanych funkcji i pomocniczym kodem startowym, umożliwiającym początek działania aplikacji.

Mianem kodu wykonywalnego jest określany taki plik, który składa w sobie efekt końcowy wszystkiego, co tworzy konsolidację.[4]

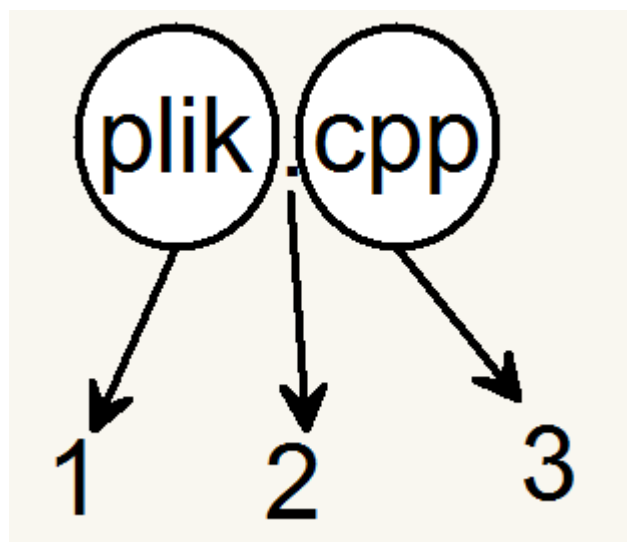


Rysunek 3.1: Kolejne etapy budowy programu

Źródło: [4]

Sporządzenie pliku z kodem źródłowym jest implementacją C++. Liczba plików źródłowych, aby zrobić program, może wynosić jeden lub więcej. Są takie implementacje, gdzie jest dostępne zintegrowane środowisko IDE, doskonałym przykładem jest Microsoft Visual C++. Środowiska te ułatwiają stwarzanie aplikacji od edycji do ostatniej korekty. Zawierają w sobie edytor tekstowy, kompilator i konsolidator łącznie, mają też menedżer projektu i program uruchomieniowy. Inne implementacje, jak GNU C++ w systemie Unix i Linux, IBM XL C/C++ dla systemu AIX, czy bezpłatnych wersjach kompilatorów Borland 5.5 i Digital Mars, mają w pakiecie jedynie kompilację i konsolidację, a użytkownik udziela komend z wiersza poleceń, daje to z kolei możliwość budowania i aktualizowania pliku z kodem źródłowym na jakimkolwiek edytorze.

Nazewnictwo plików ma szczególne znaczenie w przypadku budowania pliku źródłowego. Informację dla kompilatora i użytkownika stanowi to, jaki przyrostek posiada plik. Części składowe przedstawiają się następująco na rysunku 3.2.



*1 – Nazwa bazowa

2* - Kropka

3* - Rozszerzenie pliku

Rysunek 3.2: Składniki nazwy pliku z kodem źródłowym

Źródło: [4]

Składnik, oznaczony jedyneką jest nazwą bazową, dwójką jest kropka, a trójkę stanowi rozszerzenie pliku. Istotną kwestią jest, jakie rozszerzenie ma być zastosowane w poszczególnych systemach. Przykład *plik.C* jest odpowiednim nazewnictwem pliku z kodem źródłowym C++ dla systemu Unix. Rozszerzenia plików z kodem źródłowym dla implementacji C++ przedstawia poniższa tabela 3.1.

Implementacja C++	Rozszerzenie lub rozszerzenia plików z kodem źródłowym
Unix	C, cc, cxx, c
GNU C++	C, cc, cxx, cpp, c++
Digital Mars	cpp, cxx
Borland C++	cpp
Watcom C++	cpp
Microsoft Visual C++	cpp, cxx, cc
Freestyle CodeWarrior	cpp, cp, cc, cxx, c++

Tabela 3.1: Rozszerzenie plików z kodem źródłowym

Źródło: [4]

Początkowo C++ był zaimplementowany jako kompilator kodu C++ na C, nie umożliwiała to przekształcenia go od ręki na kod wynikowy. Program *cfront* transformował

kod źródłowy C++ na C, w konsekwencji był kompilowany prostym kompilatorem języka C, nie zaprzestano jednak na takim rozwiązaniu. Język przeniesiono na inne platformy, upowszechniono C++, a wraz ze wzrostem popularności zaczęto tworzyć kompilatory, z miejsca transformujące kod źródłowy C++ na wynikowy, to dało sprawniejszą kompilację aplikacji i uczyniło go niezależnym językiem programowania. Technika kompilacji aplikacji jest ściśle powiązana z zastosowaną implementacją.

Posługując się przykładem kompilatora dla trybu poleceń w systemie Windows, można przyjąć, że po najmniejszej linii oporu metodę kompilacji aplikacji C++ jest możliwość dokonania jej poprzez pobranie darmowego kompilatora, połączonego z trybem poleceń Windows, a inaczej ujmując oknie systemu MS-DOS. Bezpłatnie można pobrać środowiska z kompilatorem GNU C++ takie jak Cygwin i MinGW, kompilatorem użytym w nich jest g++. Może być również kompilator g++, którego użycie wiąże się z otwarciem okna trybu poleceń, co dzieje się automatycznie w przypadku środowisk Cygwin i MinGW podczas ich startu pracy.

Istotą skompilowania pliku kodu źródłowego wiąże się z wpisaniem w okienko trybu poleceń g++ i po spacji umieszczeniu nazwy pliku, wraz z rozszerzeniem, co przedstawia przykład: „g++ *plik.cpp*”. Powodzenie w kompilacji skutkuje wykreowaniem pliku wykonywalnego o nazwie *plik.exe* .[4]

4. Problematyka metody Dijkstry

Algorytm Dijkstry jest najbardziej znaną metodą poszukiwania najkrótszej ścieżki w grafie, mającą na celu wyznaczyć najmniejsze odległości od konkretnego ustalonego wierzchołka grafu, który może być skierowany lub nieskierowany, do reszty wierzchołków. Utrudnieniem jest w nim to, że obligatoryjne są tutaj wagi dodatnie w grafie, jednak stosując go w kartografii, nie ma to istotnego wpływu. [8] Został opublikowany w 1959 roku.[1]

4.1. Zasadnicze oznaczenia i pojęcia

Pojęcie grafu jest generalnie używane w dziedzinie informatyki, z kolei twierdzenie algorytmu grafowego kwalifikuje się do jednych z fundamentalnych, wchodzących w skład tejże dyscypliny wiedzy.[1]

4.1.1. Termin grafu

Grafy są często stosowaną strukturą danych,[2] które we wstępie podrozdziału zostały określone jako zasadniczo wykorzystywane w branży IT.[1]

Są to abstrakcyjne obiekty matematyczne, mające zastosowanie do modelowania sytuacji, które odzwierciedlają odpowiedzi na refleksje, dotyczące się połączeń pomiędzy parami członów w zróżnicowanych aplikacjach obliczeniowych, co za tym idzie do rozważań nad tym, czy istnieje możliwość dotarcia z jednego elementu do drugiego za pomocą powiązań. Obligatoryjny jest również namysł nad liczbą elementów powiązanych z danym elementem, a także jak przedstawia się najkrótszy łańcuch powiązań pomiędzy określonym elementem a innym wyselekcjonowanym.

Rozwiązania do powyższych wątpliwości może rozwiązać pojęcie algorytmu, ponieważ stanowią punkt wyjścia do badań nad wieloma problemami, są one nieuchronne i niezbędne, natomiast ich brak ukończenie wielu prac nie byłoby w ogóle realne.

Graf definiuje się jako zestaw wierzchołków i paleta krawędzi, gdzie wszystkie wiążą parę wierzchołków. Nazwy wierzchołków są dowolne, choć lepiej jest użyć nazw cyfrowych, ponieważ prostsze staje się pisanie kodu, gdzie w celu uzyskania dostępu do informacji o danym elemencie konieczne będzie podanie tylko indeksu elementu w tablicy. V jest liczbą wierzchołków w rozważanym grafie.

Graf określany jest jako spójny w momencie, gdy jest ścieżka z każdego wierzchołka do każdego innego.

Graf niespójny jest definiowany, jeśli składa się ze spójnych składowych, będącymi maksymalnymi spójnymi podgrafami.[5]

4.1.2. Graf skierowany i nieskierowany

Grafem skierowanym określany jest taki graf, którego opisuje para $G = (V, E)$. Litera V oznacza skończony zestaw wierzchołków w danym grafie, natomiast E stanowi relację binarną zbioru V , gdzie jego elementami są łuki grafu. W przypadku grafu nieskierowanego literka E ma inne znaczenie, gdyż określa zbiór nieuporządkowanych par wierzchołków, określanych mianem krawędzi.

O ile para (u, v) jest łukiem w skierowanym grafie, to znaczy, iż odchodzi z wierzchołka u , kończąc na v ; o tyle w nieskierowanym opisuje się, że jest sąsiednia z danymi wierzchołkami u i v . W wypadku egzystencji w grafie krawędzi (u, v) nazywa się tą przypadłość tak, że wierzchołek v jest sąsiedni wobec wierzchołka u , w odwrotnej kolejności również jest określane na tej samej zasadzie.

Grafy nieskierowane mają to do siebie, że relacja sąsiedztwa jest foremna, co w przypadku skierowanych nie stanowi już regułę, ponieważ nie musi lub może tak być. Rozpatrując graf nieskierowany, uważa się za stopień wierzchołka ilość sąsiadujących z nim krawędzi. Zupełnie inaczej trzeba się odnieść do rozważań grafów skierowanych, w których opisuje się stopień wyjściowy, wejściowy i ten, stanowiący sumę obu.

Droga, czyli ścieżka, posiadająca długość k pomiędzy wierzchołkami p i q , zawierająca się w grafie $G = (V, E)$ jest paletą wierzchołków (v_0, v_1, \dots, v_k) , w taki sposób, iż p jest równe v_0 , q ma tę samą wartość co v_k i (v_{i-1}, v_i) zawiera się w zbiorze E dla $i = 1, 2, \dots, k$.

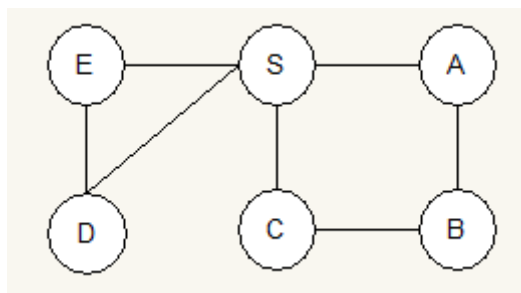
Długość drogi stanowi ilość krawędzi w niej zawartych. Składa się z wierzchołków i krawędzi. Warunkiem do tego, by była prosta, jest to, aby panowała różnorodność pomiędzy wierzchołkami. Poddroga drogi określana jest mianem szeregu następnych jej wierzchołków.

W wypadku grafu skierowanego droga (v_0, v_1, \dots, v_k) kreuje cykl, gdy $v_0 = v_k$, jak również minimum jedną krawędź musi posiadać droga. Cykl ustala się prostym, w przypadku różnych wszystkich wierzchołków v_1, v_2, \dots, v_k . Cykle (v_0, v_1, \dots, v_k) i $(v_1, v_2, \dots, v_k, v_0)$ są jednym tym samym cyklem, a pętla jest cyklem, mającym długość równą jeden. Graf skierowany, aby został utytułowany jako prosty, musi nie mieć w ogóle cykli, co określa taki graf mianem acyklicznego.

Odwołując się ponownie do grafu nieskierowanego, warto wspomnieć, że jeśli jest spójny, to musi mieć minimalnie jedną drogę pomiędzy wszystkimi parami wierzchołków. Uzasadnieniem, że wierzchołek v ma status osiągalnego dla u wtedy, gdy istnieje droga łącząca te wierzchołki. Fragmenty spójności grafu określane są jako podgrupa wierzchołków, jak również krawędzi grafu z tym, że między wszystkimi parami wierzchołków z tego samego podzbioru bytuje minimalnie jedna ścieżka. Rozważania nad grafem skierowanym skłaniają do tego, by uświadomić sobie, że jest bardzo spójny pod warunkiem osiągalności wszystkich dwóch wierzchołków na zasadzie jeden z drugiego. Są zatem klasami abstrakcji wierzchołków, ujmując mianem wzajemnie przystępnych. Rozpatrywany graf w dodatku ma jedną silną spójną składową. Acykliczny graf nieskierowany tytułowany jest jako las, jeśli jest spójny, zwany jest drzewem. Analizując graf nieskierowany warto podkreślić, że warunkiem do tego, by był dwudzielny, konieczne jest to, byle tylko zestaw jego wierzchołków V mógł być rozłączony na podzbiór V_1 i V_2 , na zasadzie przynależności wierzchołka każdej krawędzi do jednego i drugiego podzbioru.[2]

4.1.3. Odległości w grafach

Długości ścieżek odzwierciedlają to, jak daleko od siebie są oddalone dwa wybrane wierzchołki grafu. Odległość pomiędzy dwoma wybranymi wierzchołkami definiuje się długość najkrótszej ścieżki, znajdującej się pośród nich.



Rysunek 4.1: Fizyczny model grafu

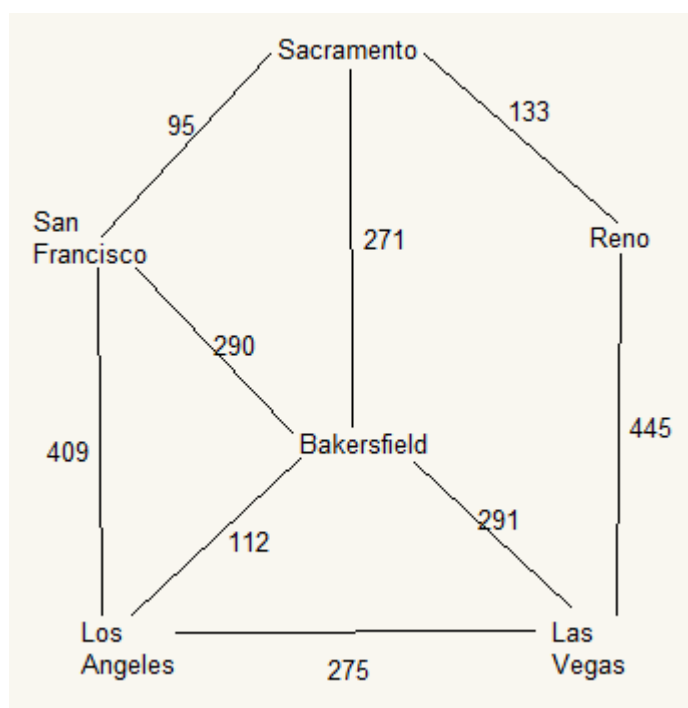
Źródło: [3]

Posługując się przykładem grafu, można zaobserwować, że odległość wierzchołka B od S równa się dwa, ponieważ dają świadectwo dwie najkrótsze drogi.[3]

4.1.4. Długości krawędzi

Termin ścieżka oznacza w grafie zestaw wierzchołków, połączonych za pomocą krawędzi. Na ścieżce prostej nie ma miejsca sytuacja powtarzających się wierzchołków. [5]

W aplikacjach szukających najkrótszych dróg fałszywym stwierdzeniem jest, że wszystkie krawędzie są analizowane z założeniem, że mają jednakową długość. Poniższy przykład ilustrujący, iż długości krawędzi są ważne przedstawia poniższa ilustracja.



Rysunek 4.2. Długości krawędzi często mają znaczenie

Źródło: [3]

Ustalając za punkt początkowy trasy San Francisco i punkt docelowy Las Vegas, należy przyjrzeć się ilustracji na rysunku 4.2, gdzie są ukazane zasadnicze autostrady, służące do ewentualnej trasy. Selekcja tej właściwej jest niczym innym jak wyborem optymalnej, czyli najkrótszej drogi. Długość odcinka każdego, wchodzącego w skład autostrady, nie jest niezbędna. Wartość długości nie w każdym wypadku oznacza fizyczną długość, są alternatywne parametry takie jak: czas, pieniądze itp.[3]

4.1.5. Relaksacja

Metoda relaksacji, a inaczej ujmując osłabiania ograniczeń przedstawiona jest w taki sposób, że dla wszystkich wierzchołków v , należących do V , otrzymana jest własność, którą jest górna granica wagi najkrótszej ścieżki o starcie w źródle s i punkcie docelowym v . Jest tytułowana jako oszacowanie wagi najkrótszej ścieżki. Drugim atrybutem są poprzednicy. Relaksacja krawędzi (u, v) ma na celu weryfikację, czy pokonując u , istnieje krótsza droga od tej już ustalonej do v , a jeśli jest taka, to trzeba zmniejszyć wartość oszacowania i istnieje możliwość modyfikacji poprzednika. Relaksacja może być jednokrotnie lub wielokrotnie wykonywana w zależności od algorytmu. Stanowi jedyny chwyt na edycję poprzedników i oszacowań wag najbardziej optymalnych dróg. Relaksacja w metodzie Dijkstry odbywa się jedynie raz dla każdej z krawędzi.[1]

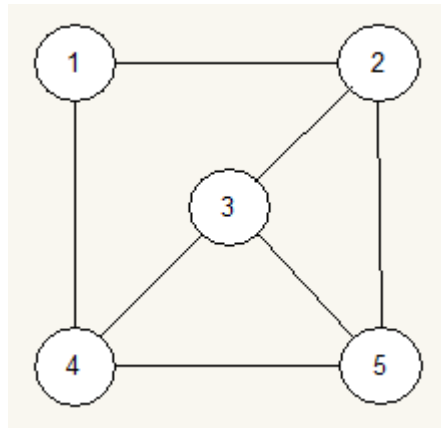
4.2. Metody reprezentowania grafów w pamięci komputera

Graf $G = (V, E)$ posiada dwie zasadnicze metody reprezentowania w pamięci komputera, czyli:

a) lista sąsiedztwa wierzchołków (w skrócie lista sąsiedztwa) – jest utrwalona w tablicy odsyłaczy, każda z niej jest odpowiedzialna za pojedynczy wierzchołek, znajdują się na niej numery (indeksy) wierzchołków z danym elementem sąsiadujących, uporządkowanie wierzchołków nie jest istotne

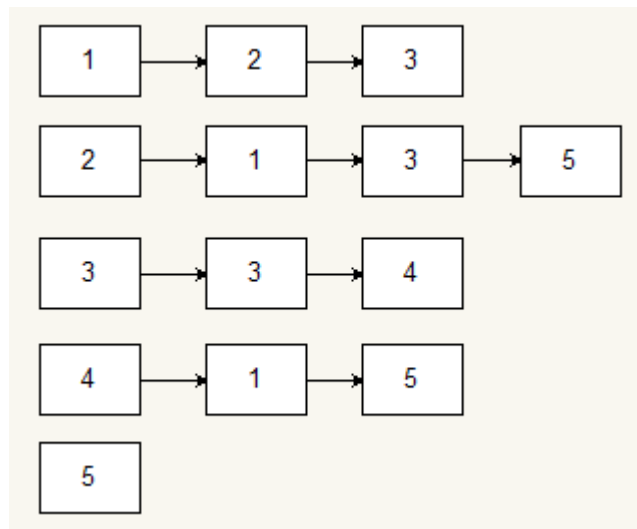
b) macierz sąsiedztwa wierzchołków (w skrócie macierz sąsiedztwa) – jest to taka macierz $A = \{a_{ij}\}$, mająca rozmiar $|V| \times |V|$, gdzie $a_{ij} = 1$ w przypadku, gdy istnieje krawędź (i,j) , natomiast $a_{ij} = 0$, w sytuacji jej braku.

Opis grafu nieskierowanego przedstawia się następująco:



Rysunek 4.3: Graf nieskierowany

Źródło: [2]



Rysunek 4.4: Reprezentacja tego grafu nieskierowanego w postaci list sąsiedztwa

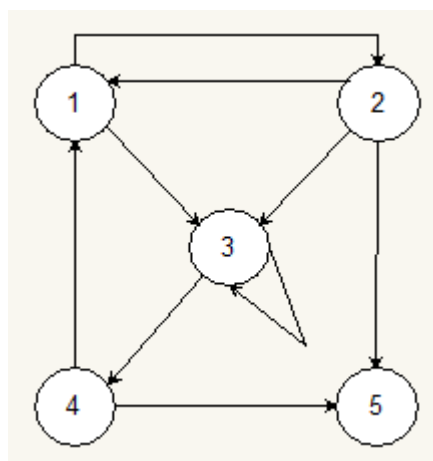
Źródło: [2]

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	1
4	1	0	1	0	1
5	0	1	1	1	0

Tabela 4.1: Macierz sąsiedztwa w tym grafie nieskierowanym

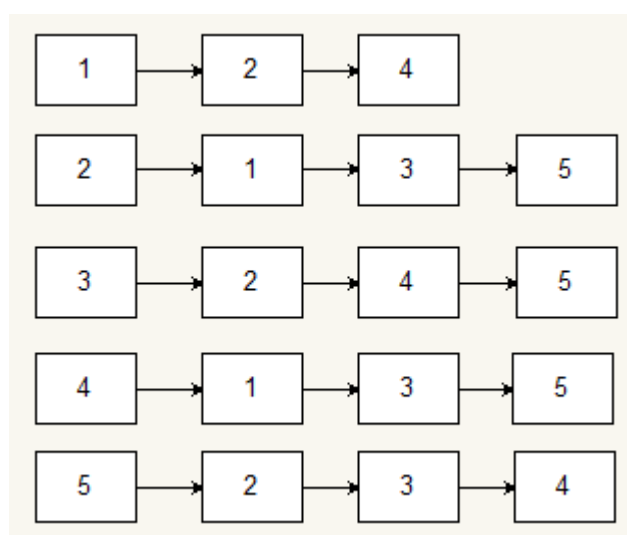
Źródło: [2]

Z kolei opis grafu skierowanego opisuje się tak oto:



Rysunek 4.5: Graf skierowany

Źródło: [2]



Rysunek 4.5: Reprezentacja tego grafu skierowanego w postaci list sąsiedztwa

Źródło: [2]

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	1
3	0	0	1	1	0
4	1	0	0	0	1
5	0	0	0	0	0

Tabela 4.2: Macierz sąsiedztwa w tym grafie skierowanym

Źródło: [2]

Na tym kończą się rozważania metod reprezentowania grafów w pamięci sprzętu IT. [2]

4.3. Istota algorytmu Dijkstry

Zagadnienie najkrótszej ścieżki obejmuje drogę z wierzchołka startowego do wierzchołka końcowego, charakteryzująca się tym, iż nie ma poza nią innej o mniejszej wadze. [5]

Algorytm Dijkstry ma zastosowanie do rozwiązywania problemów, związanych z najkrótszymi ścieżkami z jednym źródłem w grafie ważonym skierowanym $G = (V, E)$, mając na uwadze to, by wagi każdej z krawędzi były większe lub równe zeru, czyli nieujemne[1,2,3] Metoda ta jest przybliżona do algorytmu, który poszykuje minimalne drzewo rozpinające sposobem dorzucania do aktualnie odkrytych fragmentów drzewa następnych krawędzi, gdzie waga wkłada najmniej do wagi całego ogółem drzewa. Przyjmując, że początkowo zbiór S jest pusty, a będzie miejscem, dołączania po kolei wierzchołków z możliwością wyznaczenia na obecną chwilę najkrótszych dróg z wierzchołka startowego do nich. Dla każdego wierzchołka dołączonego do tegoż zbioru długość drogi jest określona i nie zostanie już modyfikowana. Tablica d jest tą, która zawiera aktualne oszacowanie wagi najkrótszych dróg lub drogi (jeśli jest jedna) wiodącej ze startu do wybranego wierzchołka. To właśnie oszacowanie dla danego wierzchołka v posiada element $d[v]$. Omawiając tablicę p , opisuje się ją jako przechowującą poprzedników na trasie z początkowego wierzchołka do innych w sieci. Z kolei $p[v]$ stanowi poprzedni w stosunku do elementu v wierzchołek w tymczasowej najkrótszej ścieżce z wierzchołka inicjującego do v . Operacja relaksacji stanowi osłabienie ograniczeń. Wszystkie wierzchołki z osobna jeśli należą do zestawu V , mają związek z wartością $d[v]$. Ta oto wartość stanowi górną granicę wagi najkrótszej drogi z wierzchołka startu do końca. Nazywana jest oszacowaniem wagi w najkrótszej drodze. Procedura relaksacji łuku (u,v) przy źródłowym wierzchołku s ma zadanie zweryfikować, czy przemierzając wierzchołek u , istnieje szansa na odnalezienie krótszej od bieżącej najkrótszej ścieżki. W przypadku jej istnienia dokonywana jest należna aktualizacja wielkości $d[v]$ i $p[v]$. Polega ona na obniżeniu oszacowania wagi najkrótszej drogi i modyfikacji poprzednika. Relaksacja łuku dokonuje się przez procedurę, ujętą poniżej

```
1  procedure relaksacja( $u, v$ );  
2  begin  
3      { $u, v$  są numerami wierzchołków}  
4  if  $d[u] + waga(u,v) < d[v]$  then  
5       $d[v] := d[u] + waga(u,v);$ 
```

```

6           p[v] := u;
7       end if;
8   end.

```

Listing 4.1: Procedura relaksacji

Źródło: [2]

Posługując się przykładem, gdy $d[u] = 5$, $d[v] = 9$, a łuk (u, v) posiada wagę 2, daje w konsekwencji po procesie relaksacji wartość $d[v]$ zmniejszoną o 7, natomiast $p[v]$ będzie równa u . W hipotetycznej sytuacji, jeśli $d[v]$ przed relaksacją wynosiłoby 6, to procedura nie wprowadziłaby żadnych zmian. Na wejściu w algorytmie jest przyjęte, że dla wszystkich wierzchołków poza źródłowym $d[v]$ jest nieskończonością. Dla źródła s jest ustawione $d[s]$ na wartość 0. Dla wszystkich wierzchołków ich poprzednicy mają $p[v] = 0$, co jest równoznaczne z ich brakiem. W kolejnym etapie są dokonuje się następujące kroki:

- selekcji wierzchołka u niezawierającego się w zbiorze S z minimalnym oszacowaniem wagi dla najkrótszej ścieżki
- dołożeniu wierzchołka u do S
- przeprowadzenie procedury relaksacji dla każdego łuku z początkiem w u , czyli z niego wychodzącego

W celu zapamiętania elementów zestawu różnicy $V - S$ istnieje możliwość skorzystania z kolejki priorytetowej, która została oznaczona w algorytmie literką Q .

Powyższe rozważania przedstawiono w poniższym listingu.

```

1   {s oznacza źródło}
2   procedure Dijkstra(V, E, s);
3   begin
4       d[v] := ∞ dla wierzchołków v należących do zbioru V;
5       d[s] := 0;
6       p[v] := 0 dla wierzchołków v należących do zbioru V;
7       S := zbiór pusty;
8       Q := wszystkie wierzchołki zbioru V;
9       while kolejka Q nie jest pusta do
10          u := wierzchołek z kolejki Q o minimalnej wartości d;
11          S := S + {u};
12          for lista wierzchołków v sąsiadujących z u do
13              relaksacja (u, v);

```

```

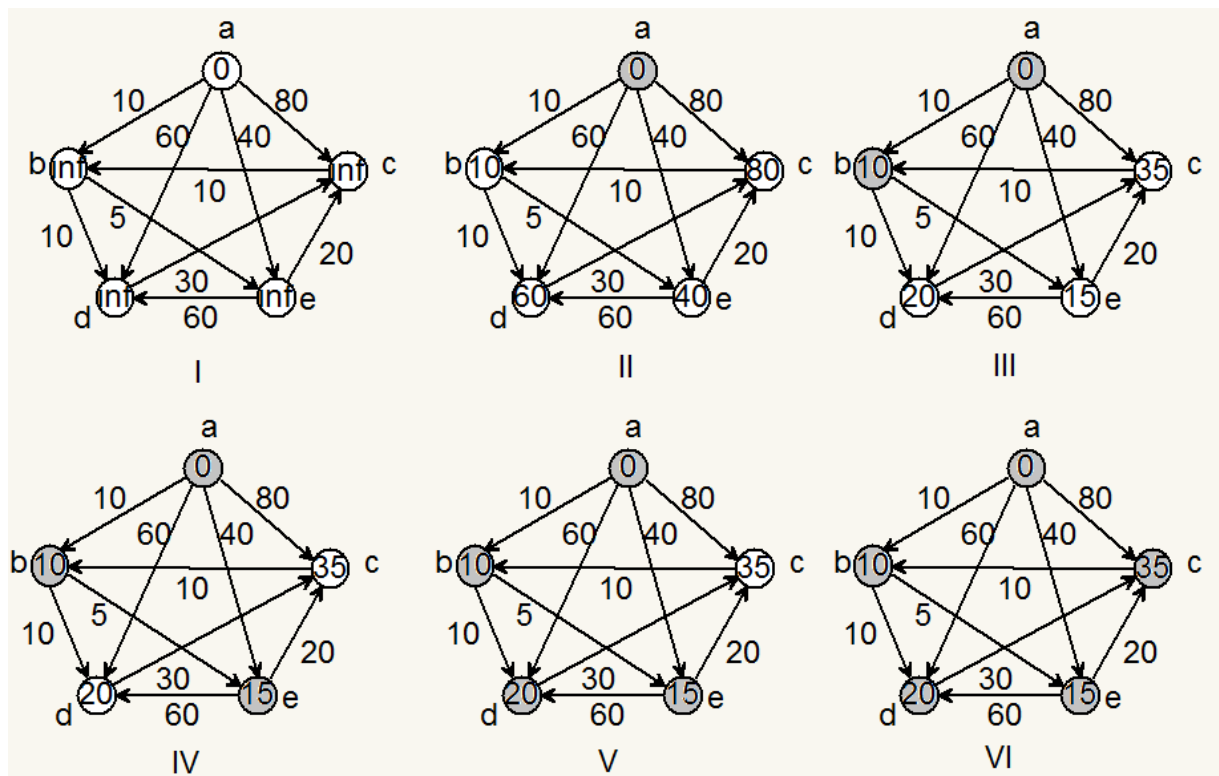
14   end for;
15   end while;
16   end.

```

Listing 4.2: Algorytm Dijkstry

Źródło: [2]

Wierzchołkiem początkowym jest a , natomiast końcowym c . Na rysunku poniżej zobrazowano wynik działania metody Dijkstry dla przykładowego grafu.



Rysunek 4.6: Efekt działania metody Dijkstry

Źródło: [2]

Na rysunku I wierzchołek a ma opisaną wagę drogi o wartości zero, lecz nie jest w zbiorze S . Następny etap prezentuje się tak, iż wierzchołek niniejszy jest wybierany jako ten, który posiada wagę minimalną drogi, w konsekwencji zostaje dodany do zbioru S . Ta sytuacja jest przedstawiona na rysunku II, gdzie wierzchołki należące do zestawu S mają pocieniowane wypełnienie. Potem dokonywana zostaje relaksacja łuków, które wyruszają z wierzchołkiem oznaczonego jako a . Są nimi (a, b) , (a, d) , (a, e) i (a, c) . Są one pocieniowane na rysunku II. Te oto relaksacje łuków prowadzą do obniżenia zredukowania odległości wierzchołków b , d , e i c od wierzchołka oznaczonego jako a na stosowne wartości: 10, 60, 40 i 80, jak na rysunku

II. Później wierzchołek, mający najmniejszą wagę drogi, b zostaje włączony do zestawu S , jak zostało pokazane na rysunku III. Są tam również zaprezentowane łuki, które zostają poddane relaksacji, czyli (b, d) i (b, e) . Skutkuje ona pomniejszeniem odległości wierzchołka e , gdzie wartość 40 ulega zmianie na 15. Tenże wierzchołek dodany jest do zestawu S , potem robiona jest relaksacja łuków (e, d) i (e, c) , co przedstawia rysunek IV. Następne wierzchołki zostają zawarte do zbioru S , czyli d i c . Dla nich sytuacja przedstawia się tak, jak zobrazowano na rysunku V i VI. W wyniku tychże czynności waga najkrótszej drogi z wierzchołka a do c posiada wartość równą 35.

Aspektem, który powinien zostać także rozpatrzony jest złożoność algorytmu. Literką n zostaje oznaczona liczba wierzchołków w zbiorze V , z kolei literką e będzie opisana liczba łuków w zbiorze E . Implementacja kolejki priorytetowej, oznaczona jako Q , ma bardzo ważny wpływ na prędkość czynności poszczególnych w metodzie Dijkstry, która może najłatwiej zostać zaprojektowana w formie tablicy jednowymiarowej z penetracją liniową, gdzie wszystkie operacje selekcji z osobna dla każdego elementu trwa $O(n)$. Którykolwiek z n wierzchołków, znajdujących się w grafie, musi być poddany w tej kolejce, co sumując, daje razem czas $O(n^2)$. Wszelki z n list sąsiedztwa podlegają rozważaniom tylko i wyłącznie jeden raz. Poza tym każdy z łuków sąsiadujący ze wszystkimi z osobna są również jednokrotnie rozpatrywane. Skoro każdy osobny łuk jest badany jednokrotnie, w konsekwencji daje to czas łączny $O(n^2 + e) = O(n^2)$ z uwagi na to, że w najgorszym wypadku liczba łuków wyraża się jako $O(n^2)$.

Analizując grafy, będące rzadkimi, znacznie lepsze rozwiązania są dzięki implementacji kolejki priorytetowej, stosując metodę kopca, nazywanym inaczej zmodyfikowanym algorytmem Dijkstry. Kopiec sporządzany jest w czasie oznaczanym jako $O(n)$. W dalszym ciągu zdarzeń dokonuje się instrukcja iteracyjna, która jest robiona n razy dla wszystkich wierzchołków. Poszczególne operacje selekcji minimalnego elementu kosztuje czas wyrażony jako $O(\log n)$, będący niezbędnym w rekonstrukcji własności kopca, co w konsekwencji dla całkowitej ilości operacji selekcji elementów obejmują czas wynoszący $O(n \log n)$. Każdy łuk ma dokonywaną procedurę relaksacji, dającą w konsekwencji zmianę jakiegokolwiek wartości, znajdującej się w kopcu. Odwrócenie własności kopca trwa $O(\log n)$, a przyglądając się łącznemu trwaniu tego kroku, biorąc pod uwagę liczbę krawędzi e , określa się go jako $O(e \log n)$, sumując wszystkie etapy można stwierdzić, iż czas całego algorytmu wynosi $O((n + e) \log n)$. Odnosząc się do określanego mianem kopca Fibonacciego, dla

operacji odwracania własności kopca jest szansa osiągnąć dla całego algorytmu łączny czas $O(1)$, a całkowitą złożoność $O(n \log n + e)$. [2]

Algorytm Dijkstry posiada następujący pseudokod:

```
Inicjacja:
Dla każdego wezła  $v$  wykonaj
{ $D[v] = \infty$ ;  $Pr[v] = \text{NULL}$ ;}
 $D[s] = 0$  //węzeł źródłowy!
 $S = \text{NULL}$ 
 $Q = \{\text{wstaw wszystkie wierzchołki grafu do kolejki}\}$ 
while not Empty( $Q$ )
     $u = \text{wyjmij z } Q \text{ element o najmniejszym } D[u]$  //kolejka
    priorytetowa!
     $S = S + u$ 
    Dla każdego wierzchołka  $v$  z listy następników wierzchołka  $u$ 
    wykonaj
    {
        //tzw. Relaksacja, sprawdzamy, czy bieżące oszacowanie
        //najkrótszej drogi do  $V$  (tzn..  $D[v]$ ) może zostać ulepszone,
        //jeśli przejdziemy przez  $u$  ( $u$  staje się poprzednikiem  $v$ ):
        if  $D[v] > D[u] + w[u, v]$  then
        {
             $D[v] = D[u] + w[u, v]$ 
             $Pr[v] = u$ 
        }
    }
}
```

Listing 4.3: Pseudokod metody Dijkstry

Źródło: [8]

Oznaczenia w algorytmie:

S – zbiór wierzchołków z obliczoną najkrótszą drogą od punktu startowego

s – punkt startu, inaczej węzeł źródłowy - źródło

t – punkt końcowy, inaczej węzeł docelowy – ujście

$V-S$ – pozostałe wierzchołki

D – tablica, zawierająca najlepsze oszacowania odległości od punktu startowego do każdego pojedynczego wierzchołka v w grafie G

$W[u, v]$ – waga krawędzi $u-v$ w grafie

Pr – tablica poprzedników

Nie wygląda tenże algorytm klarownie, lecz myśl przewodnia jest niewątpliwie jasna:

- Istotna jest tablica oszacowań, gdzie początkowo wszystkie wierzchołki z wyjątkiem źródła z odległością 0, posiadają tymczasowo oszacowanie ∞ .
- Następnie wierzchołki powiązane krawędzią z wierzchołkiem startowym, dostają tymczasowo oszacowania o wartości odległości od s , czyli wartości krawędzi. Dokonywana jest selekcja wierzchołka v o najmniejszej wartości oszacowania $D[v]$ i trzeba przyjrzeć się jego poprzednikom. W przypadku mniejszej długości drogi z s do któregośkolwiek z wierzchołków, przechodząca przez v , posiada mniejszą długość od obecnego oszacowania, to należy ją zmniejszyć, to znaczy dokonać relaksacji.
- Na koniec trzeba odszukać wierzchołek z najmniejszym oszacowaniem chwilowym i podjąć dalsze kroki w algorytmie do czasu dobrnięcia do wierzchołka celu.

Chcąc skompletować metodę Dijkstry obligatoryjnie trzeba spojrzeć na poniższy listing, które jest pseudokodem, przedstawiającym sposób na odczytanie najkrótszej ścieżki z wierzchołka startowego na docelowym kończąc.[8]

```

path (s, t)
{
  S = NULL //pusta sekwencja
  U = t
  while(true)
  {
    S = u + S //wstaw u na początek S
    if (u == s) then break;
    U = Pr[u]
  }
}

```

Listing 4.4: Sposób na odczytanie najkrótszej ścieżki w pseudokodzie

Źródło: [8]

Algorytm przeszukiwania wszerz (BFS) dokonuje inicjacji kolejki Q za pomocą wierzchołka S . Jest on jedynym wyjątkiem, gdzie odległość wynosi zero. Dla wszelkich odległości $d = 1, 2, 3, \dots$ pojawia się chwila, gdy kolejka Q posiada w sobie każdy z wierzchołków o odległości d i na tym się kończy. W przypadku przetwarzania wierzchołków z ciągu sąsiedzi niniejszych wierzchołków są usadowiani na końcu łańcucha.

Stanowi ta myśl implementacje rozważania o modelu grafu (rysunek 4.1), który po uniesieniu do góry wierzchołka oznaczonego jako S , ukazuje podział tego oto grafu na warstwy. Rozwijając to myślenie, chodzi o to, iż w pierwszej warstwie jest wierzchołek S , w drugiej - te oddalone o jeden od S , w trzeciej – oddalone o dwa. Łatwa i efektywna recepta na określenie odległości wierzchołków od S ma postać taką, że rozważane są warstwy kolejno jedna po drugiej. P określeniu wierzchołków o odległościach $0, 1, 2 \dots d$ jest prosto wyznaczyć te o odległości $d + 1$, ponieważ są to te, które nie były przeglądane, a są sąsiadami tych z warstw o odległościach d . W drzewie BFS wszystkie ścieżki, które zmierzają do startu, są możliwie najkrótsze, co można stwierdzić, iż jest to faktycznie drzewo najkrótszych ścieżek.

Algorytm BFS, czyli przeszukiwania wszerz mówi o tym, iż dla każdego $d = 0, 1, 2, \dots$ pojawia się sytuacja, gdy:

- 1) Każdy wierzchołek mający odległość mniejszą lub równą d od S posiadają właściwie opisaną odległość
- 2) Każde z pozostałych wierzchołków posiadają odległość o wartości ∞
- 3) Szereg zawiera jedynie te wierzchołki, mające odległość d

Pseudokod BFS na podstawie wcześniejszych analiz wygląda następująco:

procedura bfs(G, s)

Wejście: graf $G = (V, E)$, skierowany lub nieskierowany;
wierzchołek s należy do V

Wyjście: dla każdego wierzchołka u osiągalnego z s $\text{dist}(u)$
To odległość z s do u .

for każdy wierzchołek u należy do V :
 $\text{dist}(u) = \infty$

$\text{dist}(s) = 0$

$Q = [s]$ (kolejka zawierająca jedynie s)

while Q jest niepuste:

$u = \text{eject}(Q)$

for każda krawędź (u, v) należy do E :

if $\text{dist}(v) = \infty$:

$\text{inject}(Q, v)$

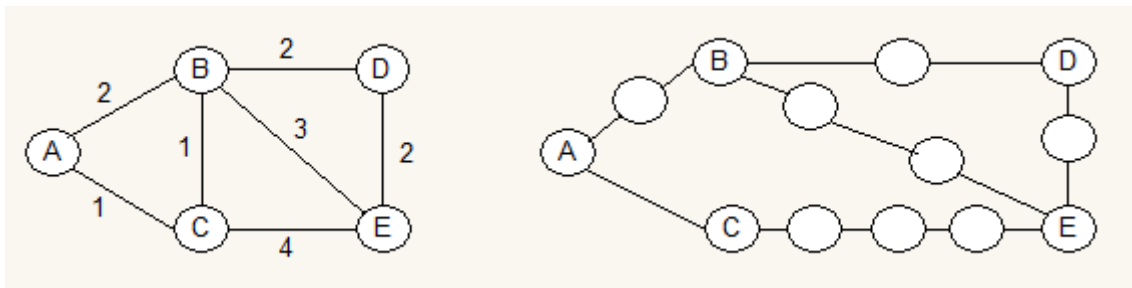
$\text{dist}(v) = \text{dist}(u) + 1$

Listing 4.5: Przeszukiwanie wszerz

Źródło: [3]

Czas przekrojowy wykonywania algorytmu BFS jest liniowy, przedstawia się w postaci $O(|V| + |E|)$. Wszystkie wierzchołki pojedynczo są wrzucane do kolejki dokładnie jeden raz, w przypadku pierwszych odwiedzin, z czego wnioskuje się, że dokonywane jest $2|V|$ operacji na łańcuchu. Resztę pracy metody realizuje najbardziej wewnętrzna pętla, która podczas działania bada pojedynczo wszystkie krawędzie, lecz robi to jedynie raz w przypadku grafów skierowanych, a nieskierowanych – dwa razy. Dzieje się to w czasie określanym jako $O(|E|)$.

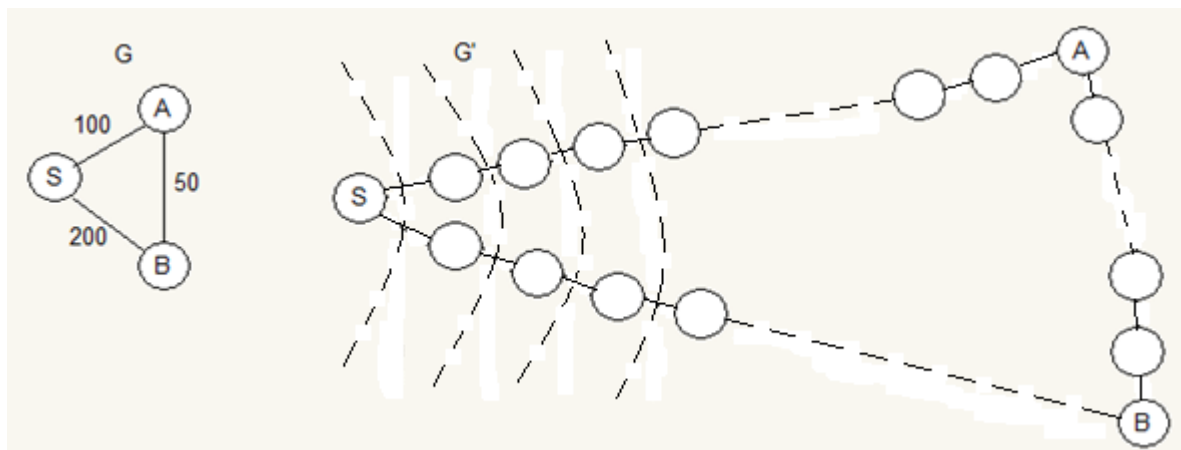
Dokonując przeszukiwania wszerz, jest możliwość odnalezienia optymalnej ścieżki dla wszystkich grafów, gdzie krawędzie posiadają wartość długości równą jeden.



Rysunek 4.7: Dzielenie krawędzi na kawałki jednostkowe

Źródło: [3]

Przekształcenie grafu G w taki sposób, by była możliwość zastosowania algorytmu BFS, należy dodać fikcyjne wierzchołki, które stanowią hipotetyczny podział długości krawędzi większych od jeden na odcinki jednostkowe, jak pokazano na powyższym rysunku. W grafie G' , transformując do jego postaci graf G , odległości między wierzchołkami V z grafu G są takie same, co w tym drugim. Długości krawędzi mają wartość jeden, co umożliwia zastosowanie algorytmu przeszukiwania wszerz na grafie G' . W przypadku bardzo długich krawędzi w grafie G' powstaje więcej wierzchołków fikcyjnych, by były jedynie krawędzie o długości jeden. Prowadzi to w konsekwencji do czasochłonności algorytmu BFS.



Rysunek 4.8: Wykonanie algorytmu BFS na grafie G' jest bardzo „ubogie w zdarzenia”.

Kropkowane linie ilustrują początkowe fale rozprzestrzeniania się algorytmu.

Źródło: [3]

Na powyższym rysunku są przedstawione grafy G i G' , gdzie graf G przedstawia krawędzie z długimi krawędziami. Przeszukiwanie wszczepia się wzdłuż jakiś krawędzi w tymże grafie, gdzie dla wierzchołków stanowiących koniec danej krawędzi jest ustawiony alarm, który uruchomi się po upływie czasu o wartości długości krawędzi. Dopóki nie zadzwoni sygnał, nic ważnego nie ma miejsca, natomiast będzie dzwonił, gdy dotrze do rzeczywistego wierzchołka. Potem algorytm może kontynuować pracę, docierając do następnych rzeczywistych wierzchołków na tej samej zasadzie. Algorytm, który można przyjąć za ten, który zawiera impuls, odzwierciedla algorytm BFS w doskonały sposób:

- Dla wierzchołka startowego s definiuje alarm na wartość równą 0
- Powtórka do sytuacji braku ustawienia jakiegokolwiek alarmu (przykładowo, najbliższy alarm zadzwoni dla wierzchołka u w określonym czasie T , co opisze odległość od s do u jako wartość T ; a dla wszystkich sąsiadów u każde v będzie miało ustawiony sygnał na $T + l(u, v)$ jeśli nie jest ustawiony, a w odwrotnym przypadku i dodatkowo jest późniejszy niż $T + l(u, v)$ to będzie przestawiony na poprzedni czas).

Metoda Dijkstry z użyciem alarmu wylicza odległości dla tych grafów G z długościami całkowitymi, a co najważniejsze dodatnimi. System alarmów implementuje się za pomocą kolejki priorytetowej, gdzie zestaw wierzchołków jest magazynowany wraz z czasami impulsu, innymi słowy kluczami. Na nich robione są następujące operacje:

- Insert (dodanie nowego elementu do zbioru)
- Decrease – key (obsługa zmniejszenia wartości czasu alarmu dla danego wierzchołka)

- c) Delete – min (zwrot wierzchołka z najmniejszym czasem alarmu, następnie usunięcie jego z szeregu)
- d) Make – queue (kreowanie kolejki priorytetowej z określonych wierzchołków i danych kluczy

Operacja Insert i Decrease – key dają możliwość definicji alarmów, Delete – min określa najwcześniejszy sygnał, co w efekcie tworzy algorytm Dijkstry. Procedura Dijkstry przedstawia się w następujący sposób w ujęciu pseudokodu.

```

procedura Dijkstra( $G, L, s$ )
Wejście: graf  $G = (V, E)$ , skierowany lub nieskierowany;
        dodatnie długości krawędzi  $\{l_e: e \text{ należące do } E\}$ ;
        wierzchołek  $s$  należy do  $V$ 

Wyjście: dla każdego wierzchołka  $u$  osiągalnego z  $s$ ,  $\text{dist}(u)$  to
        odległość z  $s$  do  $u$ .

for każdy wierzchołek  $u$  należy do  $V$ :
     $\text{dist}(u) = \infty$ 

 $\text{prev}(u) = \text{nil}$ 
 $\text{dist}(s) = 0$ 

 $H = \text{makequeue}(V)$  (używając wartości z tablicy  $\text{dist}$  jako kluczy)
while  $H$  jest niepuste:
     $u = \text{deletemin}(H)$ 

    for każda krawędź  $(u, v)$  należy do  $E$ :
        if  $\text{dist}(v) > \text{dist}(u) + l(u, v)$ :
             $\text{dist}(v) = \text{dist}(u) + l(u, v)$ 
             $\text{prev}(v) = u$ 
         $\text{decreasekey}(H, v)$ 

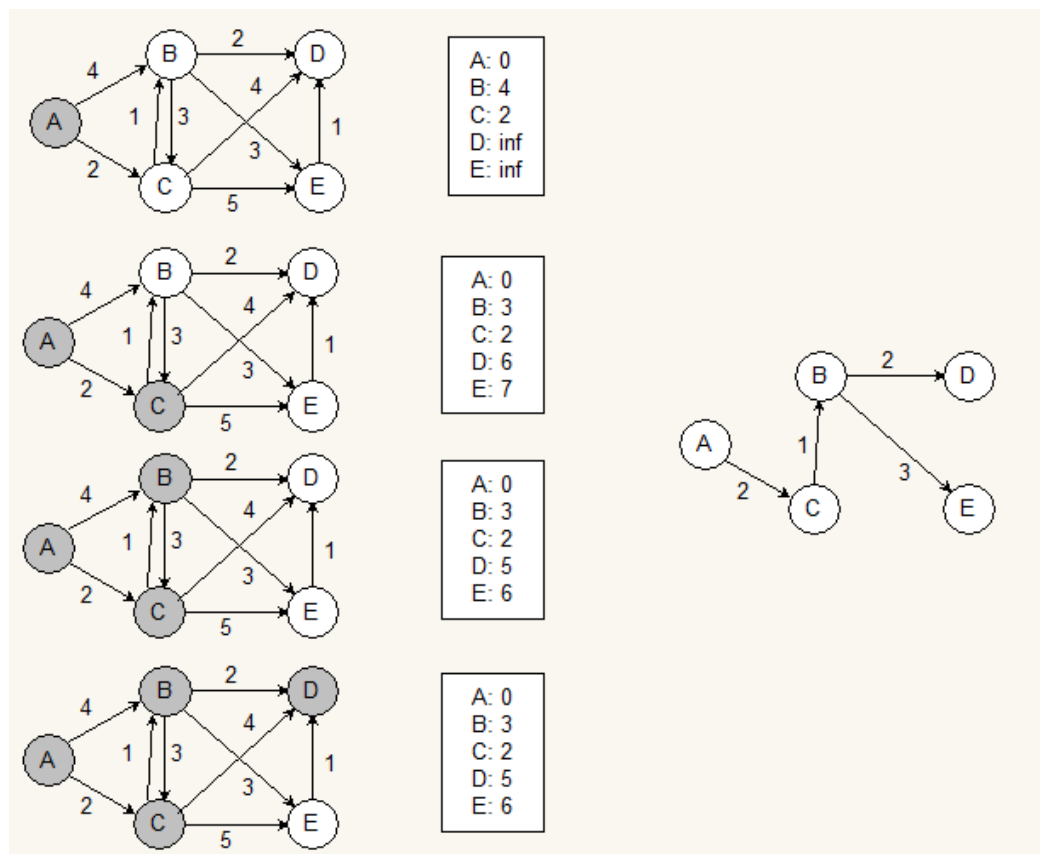
```

Listing 4.6: Algorytm Dijkstry znajdowania najkrótszych ścieżek

Źródło: [3]

W powyższym pseudokodzie $\text{dist}(u)$ stanowi czas aktualnie ustalonego impulsu dla wierzchołka u , a wartość nieskończona oznacza jego brak. Dodatkowa tablica prev

przechowuje identyfikatory wierzchołka, który poprzedza u , na aktualnej najkrótszej ścieżce z s do u . Dzięki powrotnym wskaźnikom można prosto zrekonstruować optymalną drogę, tablica prev jest reprezentacją każdej z najkrótszych tras. Poniższy rysunek przedstawia przykładowo wykonany algorytm na grafie razem z końcowym drzewem z najbardziej optymalnymi rozwiązaniami.



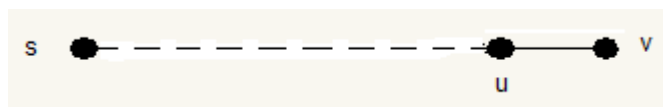
Rysunek 4.9: Przebieg metody Dijkstry z wierzchołkiem A jako tym początkowym. Rysunek ukazuje następane wartości zawarte w tablicy dist, a także efekt końcowy drzewa optymalnych tras

Źródło: [3]

Algorytm BFS stanowi niejako dowód algorytmu Dijkstry, odzwierciedlający zamiast kolejki kolejkę priorytetową, faworyzując dane wierzchołki pod względem długości połączonych z nimi krawędzi.

Inne rozwiązanie, które w żaden sposób nie korzysta z algorytmu BFS, jest bardziej bezpośrednie i abstrakcyjne. W celu znalezienia optymalnej drogi należy zacząć od wyjścia z punktu startu s , następnie po kolei zmierzać do zajęcia coraz obszerniejszej części w grafie, dla których odległości i długości najkrótszych dróg są wiadome. Jest zwiększany

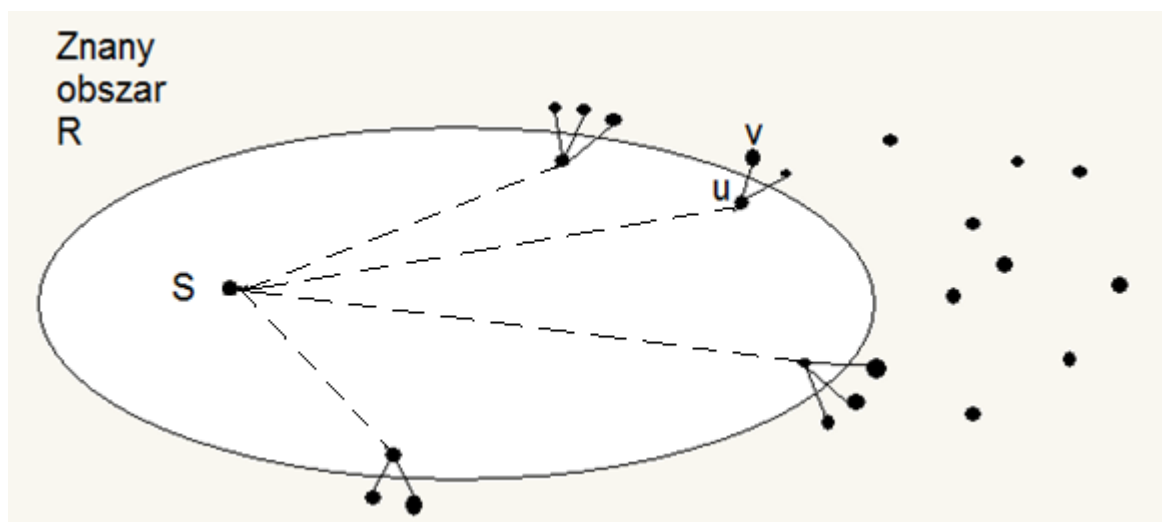
chronologicznie według niniejszego schematu: włączenie wierzchołków najbliższych, potem tych dalszych. W innym ujęciu można powiedzieć, że wiadomy obszar stanowi podzbiór wierzchołków R z wierzchołkiem s , a potem jest dodawany do niego wierzchołek najbliższy startowemu, oznaczony literą v . Odnalezienie tego oto wierzchołka trzeba zrobić taką oto techniką, że w założeniach jest to, iż rozważany wierzchołek u jest bezpośrednim poprzednikiem v , znajdującym się na optymalnej ścieżce od startu s do v .



Rysunek 4.10: Schemat wierzchołka u jako bezpośredniego poprzednika v , znajdującego się na najbardziej optymalnym rozwiązaniu

Źródło: [3]

Zakładając, iż każda z długości jest dodatnia i u koniecznie musi bliżej wierzchołka s niż od v , można wywnioskować, że u znajduje się w zestawie R , bo jeśli tak nie byłoby, doszłoby do sprzeczności, która polega na byciu przez v najbliższym wierzchołkiem s ze zbioru wierzchołków nie zawierającego się w R . Optymalną drogą od s do v jest więc jedna z wiadomych na obecną chwilę najbardziej optymalnych tras, która jest rozszerzona o jedną krawędź.



Rysunek 4.11: Rozszerzenia znanych najkrótszych tras o jedną krawędź

Źródło: [3]

Jak widać na powyższym rysunku, wiele ścieżek utworzonych poprzez zwiększenie znanych najkrótszych dróg jest z pewnością widoczne. Do v prowadzi ta, która jest najkrótsza spośród

tych, które zostały poszerzone. Sytuacją sprzeczną jest z pewnością ta, w której jest ścieżka krótsza, prowadząca do v , z tych rozszerzonych, następny raz doszłoby do niezgodności ze statusem v , wierzchołkiem poza zbiorem R , będący najbliższym do s . Podsumowując, proste jest odnalezienie v , który jest wierzchołkiem niebędącym w zbiorze R , gdzie wartość $\text{dist}(u) + l(u, v)$ rozwiązana dla wszystkich u z R jest wartością najmniejszą, czyli ze wszystkich ścieżek stanowiących rozszerzenie o jedną dokładnie krawędź wiadomych optymalnych ścieżek dokonywany jest wybór tej, która ma najkrótszą długość. Powstaje w ten sposób metoda, która za pośrednictwem przeglądu rozszerzeń bieżącej listy najkrótszych rozwiązań, powiększa obszar R . Warto podkreślić, że na każdym etapie metody nowo powstałe rozszerzenia są produkowane tylko i wyłącznie przez ten wierzchołek, który dorzucono do zbioru R , a reszta rozszerzeń ma oszacowania bez zmian i nie są obligatoryjne ponownej analizy. Poniższy pseudokod mówi, że $\text{dist}(v)$ jest długością bieżącego optymalnego rozwiązania najkrótszej ścieżki, która jest rozszerzona i zmierza ku wierzchołkowi v . Wartość długości dla wierzchołków poza zbiorem R jest ∞ .

```

Ustaw  $\text{dist}(s)$  na 0, a pozostałe  $\text{dist}(\cdot)$  na  $\infty$ 
 $R = \{s\}$  („znany obszar”)
while  $R \neq V$  :
    Wybierz wierzchołek  $v$  nie należący do  $R$  z najmniejszą wartością  $\text{dist}(\cdot)$ 
    Dodaj  $v$  do  $R$ 
    for każda krawędź  $(v, z)$  należy do  $E$ :
        if  $\text{dist}(z) > \text{dist}(v) + l(v, z)$ :
             $\text{dist}(z) = \text{dist}(v) + l(v, z)$ 

```

Listing 4.7: Ustawianie długości $\text{dist}(s)$ i $\text{dist}(\cdot)$

Źródło: [3]

Dokładając do niniejszego pseudokodu operację, działającą na kolejce priorytetowej, wynikiem staje się metoda Dijkstry, jak w pseudokodzie listing 4.6.

W celu udowodnienia wiarygodności algorytmu, trzeba przeprowadzić dowód indukcyjny tak samo, jak w przypadku przeszukiwania wszerz. Hipoteza przedstawia się w następującej postaci.

Koniec wszystkich iteracji pętli while są zachowane warunki:

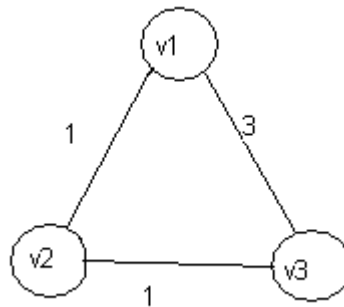
- 1) wartość d jest większa lub równa od odległości od s wszystkich po kolei wierzchołków zawierającego się w zestawie R

- 2) wartość d jest mniejsza lub równa odległości od s wszystkich pojedynczo wierzchołków nie zawartych w zestawie R
- 3) dla każdego wierzchołka u wartość $\text{dist}(u)$ określa długość najbardziej optymalnej trasy z s do u , gdzie wewnętrzne wierzchołki należą do zestawu R , a w przypadku braku jej $\text{dist}(u)$ wynosi ∞ .

W pierwszym etapie, czyli jeśli d wynosi 0, jest otrzymane od razu, natomiast szczegółowe dane etapu indukcyjnego jest szansa uzupełnienia na podstawie powyższych analiz.

Zważywszy na poziom abstrakcji z listingu 4.6 można powiedzieć, że algorytm Dijkstry jest pod względem struktury taki sam jak BFS, ale jego praca jest wolniejsza ze względu na fakt, że operacje kolejki priorytetowej pod względem obliczeń stanowią większe kryteria od działania w stałym czasie operacji eject i inject algorytmu przeszukiwania wszerz. Makequeue ma czas taki, jak $|V|$ operacji takich jak insert/decreasekey, a ich czas potrzebny na pracę jest uzależniony od implementacji. Posługując się przykładem kopca binarnego czas całkowity dla wykonania niniejszej metody jest równy $O((|V| + |E|) \log |V|)$. [3]

Najkrótsza ścieżka wcale nie musi stanowić tej, która ma najmniej ilości krawędzi. Jej długość, czyli odległość wierzchołka startowego do końcowego jest sumą wag krawędzi, występujących na niej. Przykładowo jest graf o trzech wierzchołkach (v_1 , v_2 i v_3), jeżeli krawędź (v_1, v_2) jest równa 1, $(v_1, v_3) = 3$, a (v_2, v_3) ma odległość 1.



Rysunek 4.12: Ilustracja grafu z danymi

Źródło: [6]

Droga, określona przy pomocy algorytmu BFS dla wierzchołka startowego v_1 i końcowego v_3 , będzie zawierała się z krawędzi (v_1, v_3) z daną długością 3, lecz występuje krótsza trasa $v_1-v_2-v_3$, która ma z kolei długość łącznie 2, gdyż jest to suma krawędzi (v_1, v_2) i (v_2, v_3) , mające jednakowe długości 1. [6]

5. Projekt aplikacji Algorytm Dijkstry

Rozdział ten stanowi sedno pracy. W jego skład wchodzi założenia, wymagania i implementację algorytmu Dijkstry w języku programowania ogólnego przeznaczenia, którym jest język C++.

5.1. Założenia

Program sporządzony w ramach pracy daje użytkownikowi następujące możliwości:

- Wytyczyć wszystkie najkrótsze ścieżki w grafie
- Sprecyzować długość najkrótszej ścieżki

Pierwsza wytyczna może dotyczyć jednej najkrótszej drogi z wierzchołka startowego do końcowego, ale możliwe jest też wskazanie więcej niż jednej trasy w przypadku, jeśli takie występują. Tego typu sytuacja ma miejsce, jeśli podczas obliczania wagi dla sąsiadującego wierzchołka z wierzchołkiem aktualnie rozważanym jest równa wadze obliczonej wcześniej. Powstaje sytuacja, kiedy dany wierzchołek przestaje mieć jednego poprzednika w drodze ku wierzchołkowi startowemu. Odczyt najkrótszej trasy odbywa się poprzez czytanie od końca do początku, innymi słowy od wierzchołka końcowego do startowego z tabeli poprzedników.

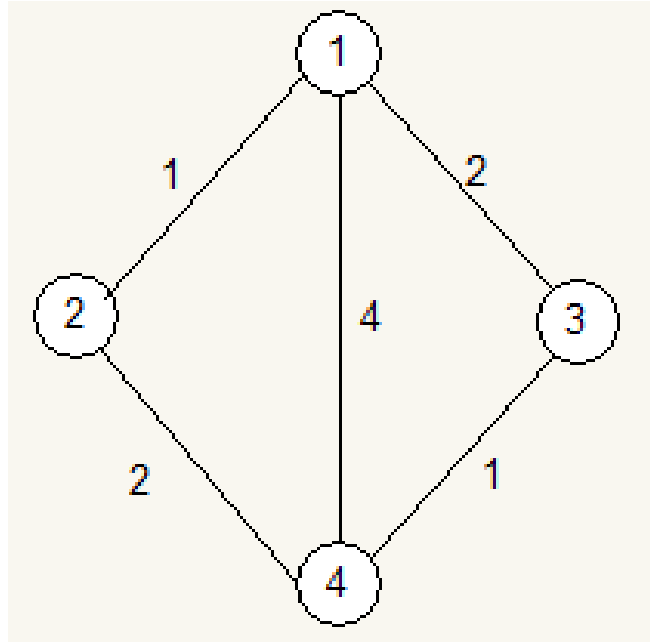
Kolejna dyrektywa określa długość najkrótszej drogi lub dróg, w przypadku ich większej ilości. Jest ona odczytywana z tabeli wag dla wierzchołka końcowego.

5.2. Wymagania

W celu uruchomienia zaprojektowanej aplikacji należy posiadać na własnym urządzeniu, będącym komputerem osobistym PC lub laptopem system operacyjny firmy Microsoft, którym jest Windows. Do kompilacji jest konieczny co najmniej środowisko zawierające kompilator GNU C++, czyli Cygwin lub MinGW z kompilatorem g++. To jest minimalny wymóg, lecz korzystniejsze jest do tego celu środowisko IDE takie jak Dev-C++, które jest bezpłatne. Program sporządzono w środowisku programistycznym Visual Studio 2012 Ultimate, także wybrana wersja VS jest jak najbardziej kompatybilna. Konieczny jest edytor kodu źródłowego i kompilacja, by zmienić nazwę pliku, z którego będą pobierane długości krawędzi i liczba wierzchołków. Nie jest istotna kwestia połączenia sprzętu z Internetem. Brak łączności nie ma na pracę programu żadnego wpływu i nie jest on konieczny.

5.3. Opis aplikacji

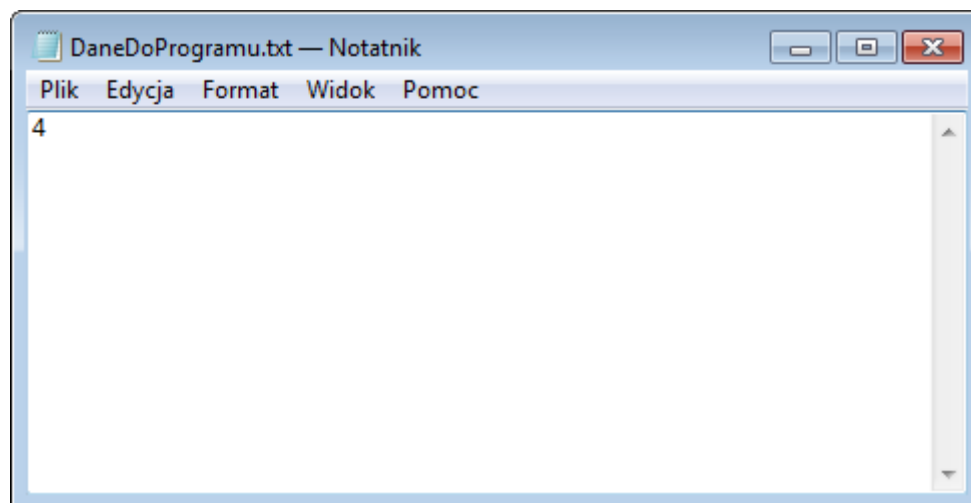
Mając dany graf do rozważań, należy kierować się następującymi czynnościami.



Rysunek 5.1: Przykład prostego grafu

Źródło: opracowanie własne

Na początku należy utworzyć plik z danymi, przykładowa nazwa może być DaneDoProgramu.txt. Edytor tekstowy jest dowolny. Może być nim Notatnik, Notepad++, Microsoft Word, LibreOffice Writer i inne. Ważne jest, aby ten edytor miał możliwość zapisu pliku z rozszerzeniem *.txt. Do niego trzeba wpisać w pierwszym wierszu liczbę wierzchołków, która jednocześnie będzie określała liczbę wierszy i kolumn macierzy, przechowującej długości krawędzi pomiędzy wierzchołkami.



Rysunek 5.2: Wpisanie liczby wierzchołków

Źródło: opracowanie własne

Jak widać na powyższym zrzucie ekranu, została wpisana liczba 4, odpowiadająca rozmiarowi macierzy pod względem liczby wierszy i kolumn.

Kolejnym krokiem jest wpisanie długości krawędzi pomiędzy poszczególnymi wierzchołkami w taki sposób, jak zostało to przedstawione w tabeli poniżej. Indeks kolumny oznacza, do jakiego wierzchołka jest odległość z wierzchołka o indeksie dla danego wiersza.

Wiersz/ Kolumna	1	2	3	4
1	0	1	2	4
2	1	0	∞	2
3	2	∞	0	1
4	4	2	1	0

Tabela 5.1: Tabelka przedstawiająca długości pomiędzy wierzchołkami

Źródło: opracowanie własne

Warto podkreślić dwa aspekty, które są konieczne do objaśnienia. Pierwszym z nich jest kwestia wartości, która ma być wpisana jako odległość wierzchołka do samego siebie. Odpowiedź na tą refleksję jest bardzo prosta, należy wpisać liczbę zero, gdyż z logicznego punktu widzenia odległość punktu do tego samego punktu nie istnieje, czyli charakteryzuje się ten sam punkt. Druga postać tyczy się sytuacji, gdy nie ma połączenia między dwoma wierzchołkami. Błędym myśleniem jest zamiar wpisania liczby zero. Patrząc na ten problem z innej perspektywy, nasuwa się idea, mówiąca o tym, że można dostać się do tego wierzchołka drogą wiodącą przez inne wierzchołki. To z kolei przemawia za tym, by przyjąć,

że odległość w omawianym problemie jest wartością nieskończoną. W tabeli jest ona ujęta jako „inf”, natomiast w pliku tekstowym nie może zostać w ten sposób zdefiniowana niniejsza odległość. Konieczna jest wystarczająco duża wartość, która zostanie przyjęta za wartość nieskończoności. W programie została zdefiniowana jako stała o wartości 9999. Tą oto liczbę należy zamieścić w pliku tekstowym tak, jak zostało to ujęte w tenże tabeli.

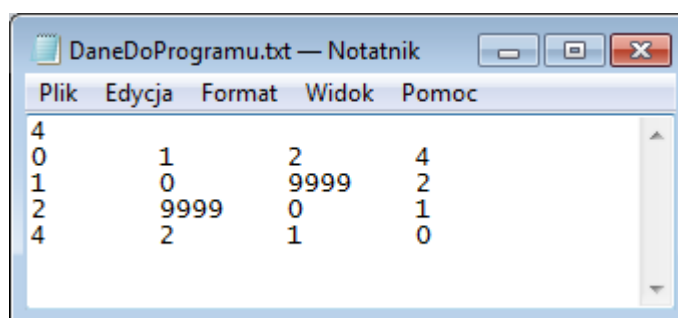
Wiersz/Kolumna	1	2	3	4
1	0	1	2	4
2	1	0	9999	2
3	2	9999	0	1
4	4	2	1	0

*9999 – nieskończoność

Tabela 5.2: Tabelka przedstawiająca długości pomiędzy wierzchołkami według programu

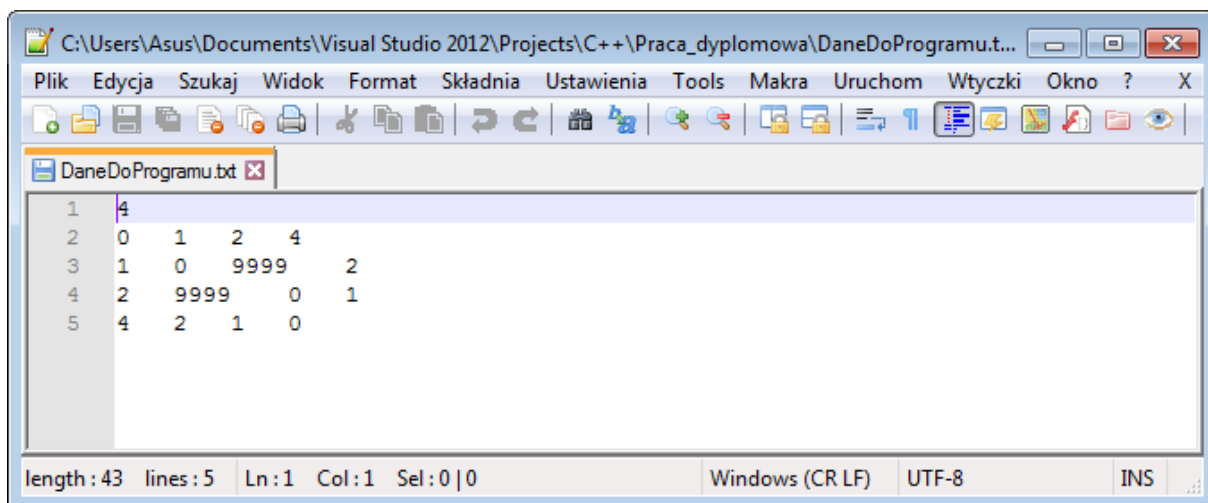
Źródło: opracowanie własne

Uzasadnieniem, że nie można wstawić „inf” w miejsce nieskończoności tylko 9999, jest fakt, iż podczas wczytywania danych z pliku w momencie uruchomienia aplikacji są one pobierane do tablicy przechowującej parametry definiujące odległości. Tablica jest typu całkowitego (int), a napis „inf” jest ciągiem (łańcuchem) znaków, więc jest typu string. To stwierdzenie jednoznacznie wyklucza możliwość zastosowania tego typu zapisu w pliku.



Rysunek 5.3: Dane dla grafu sporządzone w Notatniku

Źródło: opracowanie własne

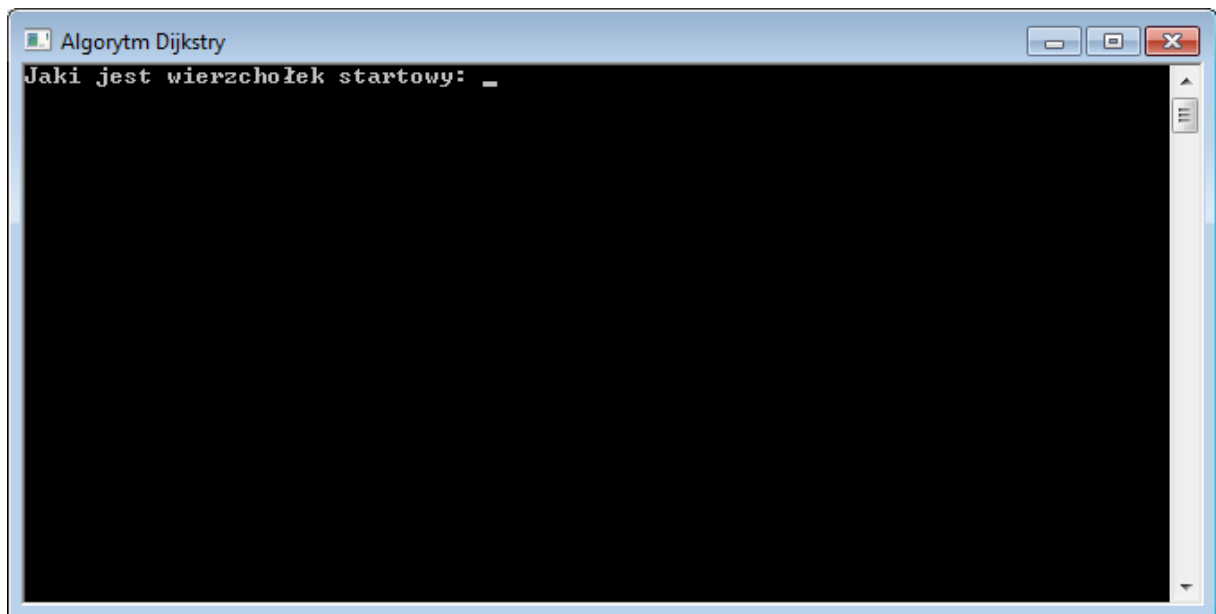


Rysunek 5.4: Dane dla grafu sporządzone w Notepad++

Źródło: opracowanie własne

Warto podkreślić jeszcze jedną techniczną sprawę, dotyczącą oddzielenia poszczególnych wartości w tym samym wierszu. Muszą być oddzielone za pomocą przycisku tabulatora. Połączenie w postaci wprowadzonych danych i użycie tabulacji pomiędzy parametrami najlepiej wygląda i jest przejrzyste w systemowym Notatniku. Porównując zrzuty ekranów można stwierdzić, iż najbardziej przejrzyste i wyrównane wartości, mimo, iż jest wartość nieskończoności w macierzy, najlepiej wygląda efekt z wykorzystaniem systemowego Notatnika niż Notepad++. W pierwszym kolumny są wyrównane, natomiast w drugim ten efekt jest zaburzony, wartości w kolumnach są ułożone nierówno. To jest jedynie ocena wyglądu, który nie ma wpływu jakości pliku, patrząc z perspektywy działania programu na nim.

Po sporządzeniu pliku, przechowującego dane, nie ma przeszkód do przejścia na kolejny etap, w którym trzeba uruchomić aplikację. Po uruchomieniu powinna mieć taką oto postać.



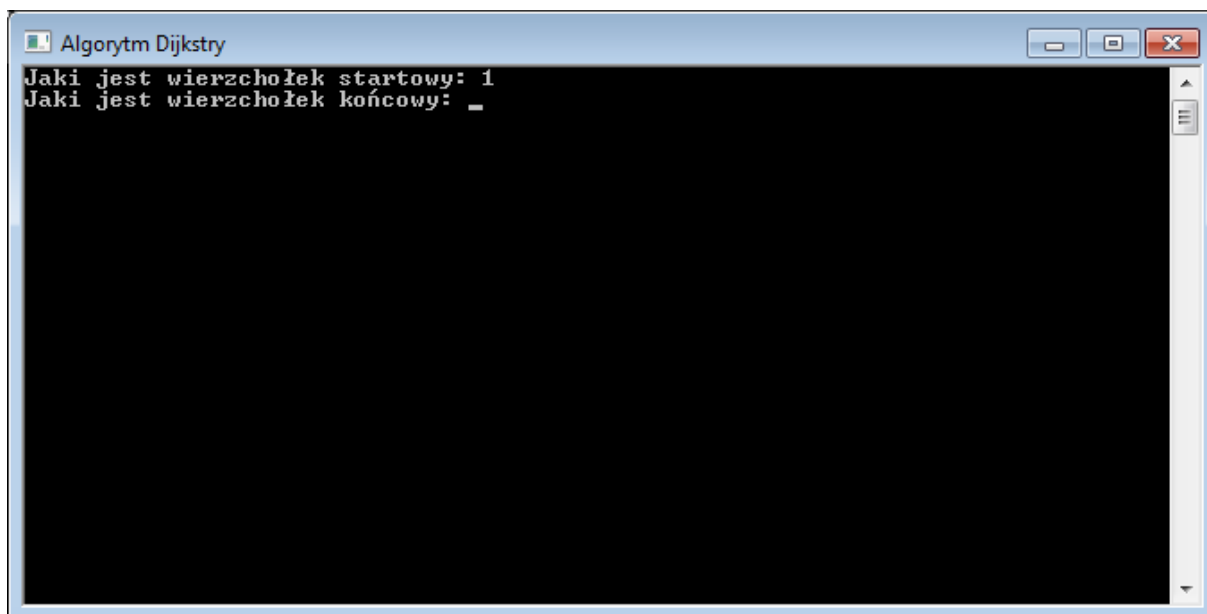
Rysunek 5.5: Aplikacja po uruchomieniu

Źródło: opracowanie własne

Należy wpisać wierzchołek startowy, który nie może być liczbą ujemną, równą zero lub większą od całkowitej liczby wierzchołków. W przypadku złamania przytoczonych zasad program zakończy działanie już w tym kroku, informując przed zamknięciem o tym, co użytkownik uczynił niewłaściwego. Pod pojęciem niewłaściwego kryje się sytuacja:

- a) wpisania liczby zero (indeksy na grafach rozpoczynają się od 1)
- b) zanotowanie wartości ujemnej
- c) napisanie wartości, która jest większa od liczby wierzchołków w grafie (w przykładzie jest ich 4), innymi słowy jest to okoliczność, gdzie użytkownik przekracza dostępne indeksy wierzchołków, a program wychodzi poza zakres.

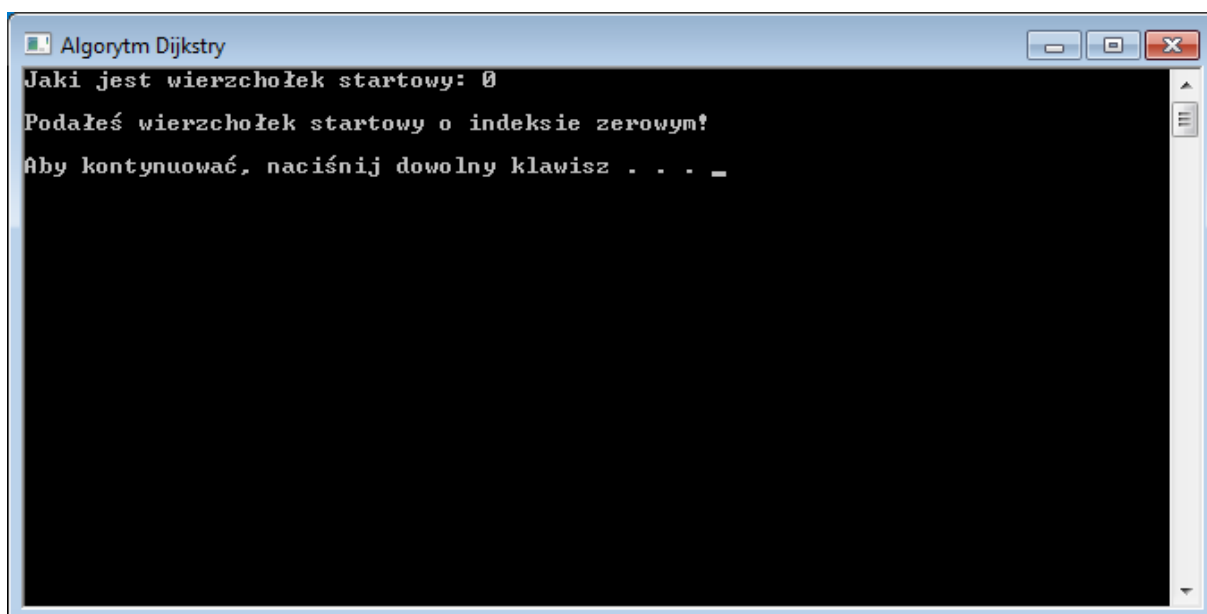
Tak przedstawia się właściwe zdefiniowanie wierzchołka wstępnego.



Rysunek 5.6: Poprawne wpisanie wierzchołka startowego

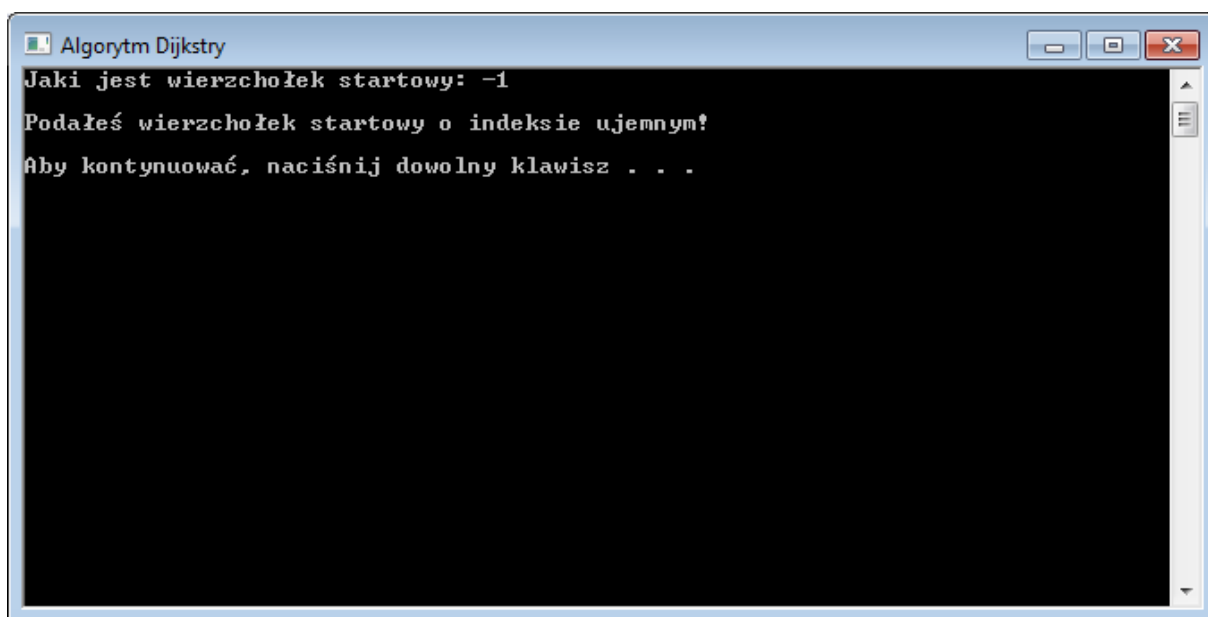
Źródło: opracowanie własne

Jeśli użytkownik zapisze nieodpowiedni wierzchołek zapoczątkowujący algorytm Dijkstry, nastąpi wyświetlenie odpowiedniego komunikatu i zamknięcie programu, jak zostało o tym wspomniane wcześniej. Dalsze rozważania przy określeniu niewłaściwych parametrów na wejściu pozbawiają analizy metodą Dijkstry danego problemu, a co za tym idzie działania programu w późniejszej fazie.



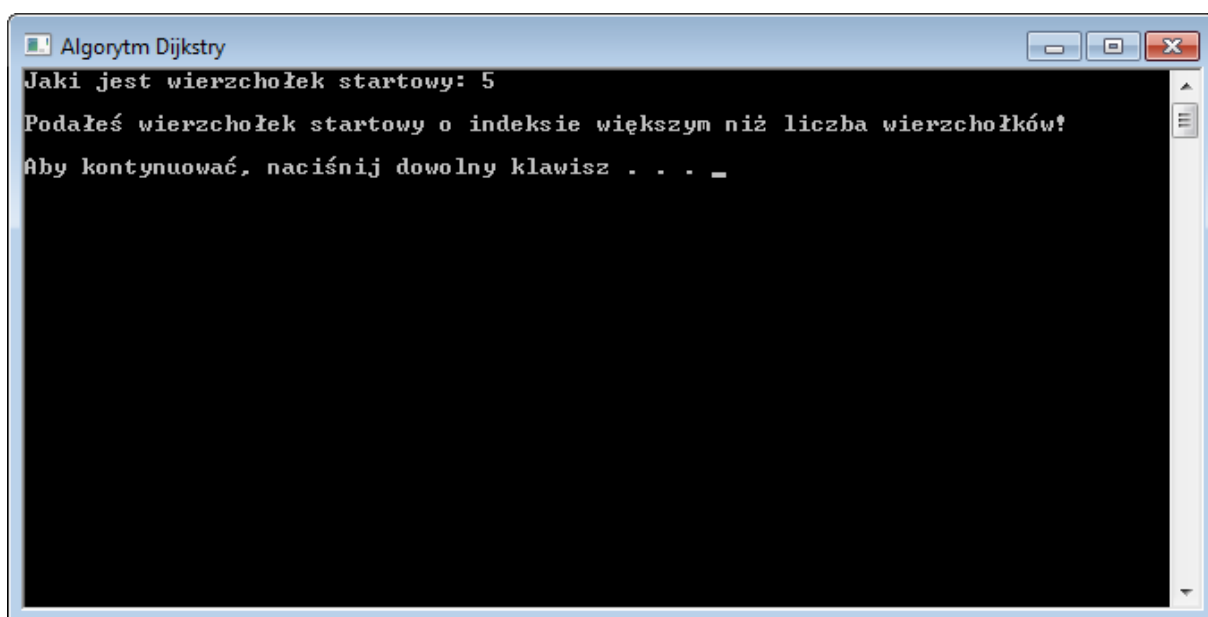
Rysunek 5.7: Błędnie wpisany wierzchołek startowy – przypadek 1

Źródło: opracowanie własne



Rysunek 5.8: Błędnie wpisany wierzchołek startowy – przypadek 2

Źródło: opracowanie własne



Rysunek 5.9: Błędnie wpisany wierzchołek startowy – przypadek 3

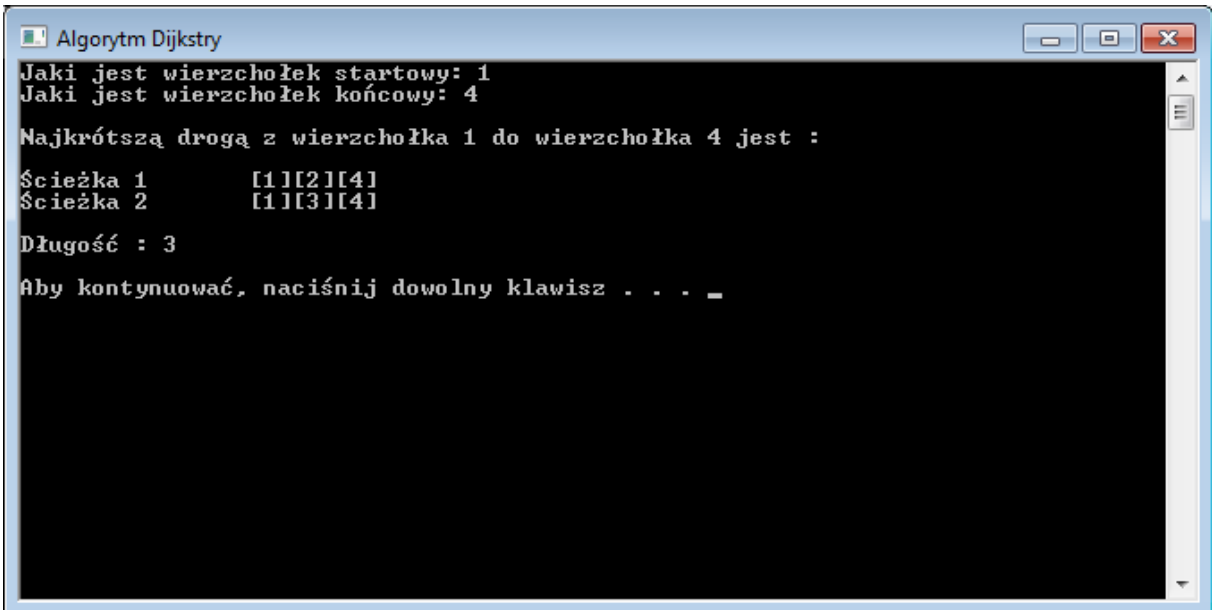
Źródło: opracowanie własne

Jeśli użytkownik wpisze właściwą wartość, może przejść do kolejnej procedury, czyli określenia wierzchołka końcowego. Z tego względu również obowiązują takie same reguły, jakie obowiązują podczas wyznaczania wierzchołka startowego. Wierzchołek docelowy nie może mieć indeksu ujemnego, równego zero, a także wychodzić poza zakres, czyli mieć

większą wartość od liczby wierzchołków. Do tego fragmentu działania programu użytkownik dochodzi, jeśli poprawnie zdefiniuje wierzchołek początkowy. Moment wpisania wierzchołka końcowego kończy zarazem rolę użytkownika jako klienta, wysyłającego zapytanie do pełniącego funkcję serwera. Następuje to bez względu na poprawność wpisania wierzchołka finalnego. Od właściwego określenia tegoż wierzchołka podlega jedynie efekt końcowy działania aplikacji. Jeśli użytkownik wpisze wierzchołek szczytowy, wtedy program wyszczególni wszystkie najkrótsze ścieżki. W przeciwnym razie wypunktuje błąd popełniony przez użytkownika. Jest to oczywiście jeden z trzech przypadków:

- a) zapisanie liczby zero
- b) zanotowanie liczby ujemnej
- c) odnotowanie liczby poza zakresem, czyli większej od liczby wierzchołków

Poprawnie wpisane dane parametry skutkują odpowiednim końcem pracy programu, przedstawiające się w niniejszy sposób:



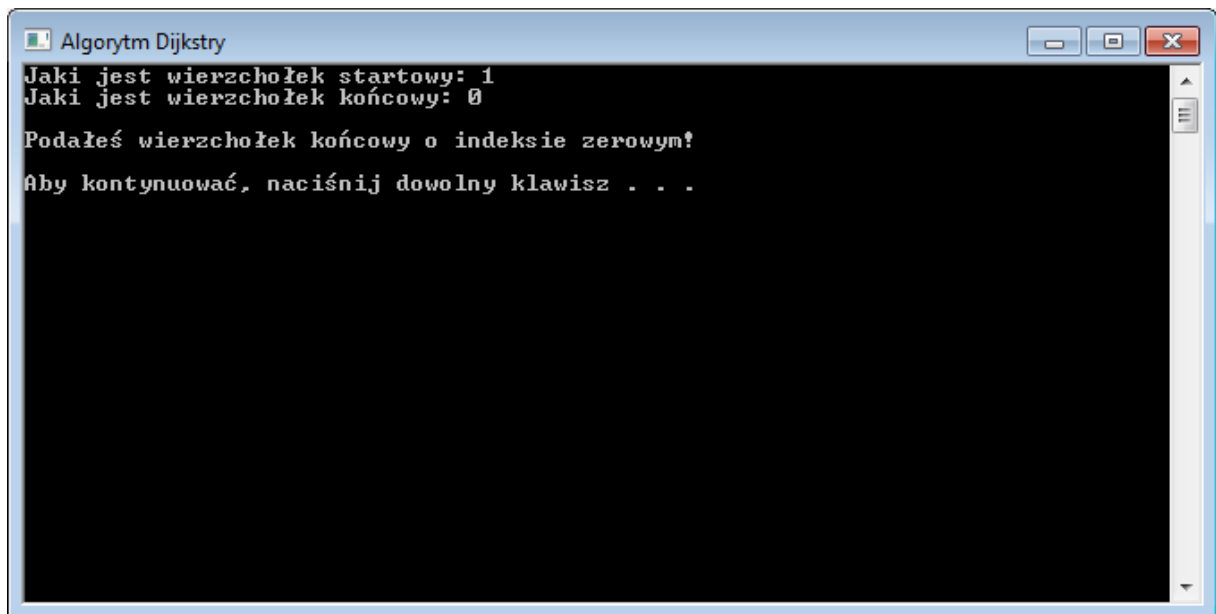
```
Algorytm Dijkstry
Jaki jest wierzchołek startowy: 1
Jaki jest wierzchołek końcowy: 4

Najkrótszą drogą z wierzchołka 1 do wierzchołka 4 jest :
Ścieżka 1      [1][2][4]
Ścieżka 2      [1][3][4]
Długość : 3
Aby kontynuować, naciśnij dowolny klawisz . . . _
```

Rysunek 5.10: Poprawne wpisanie wierzchołka końcowego i adekwatne efekt pracy aplikacji

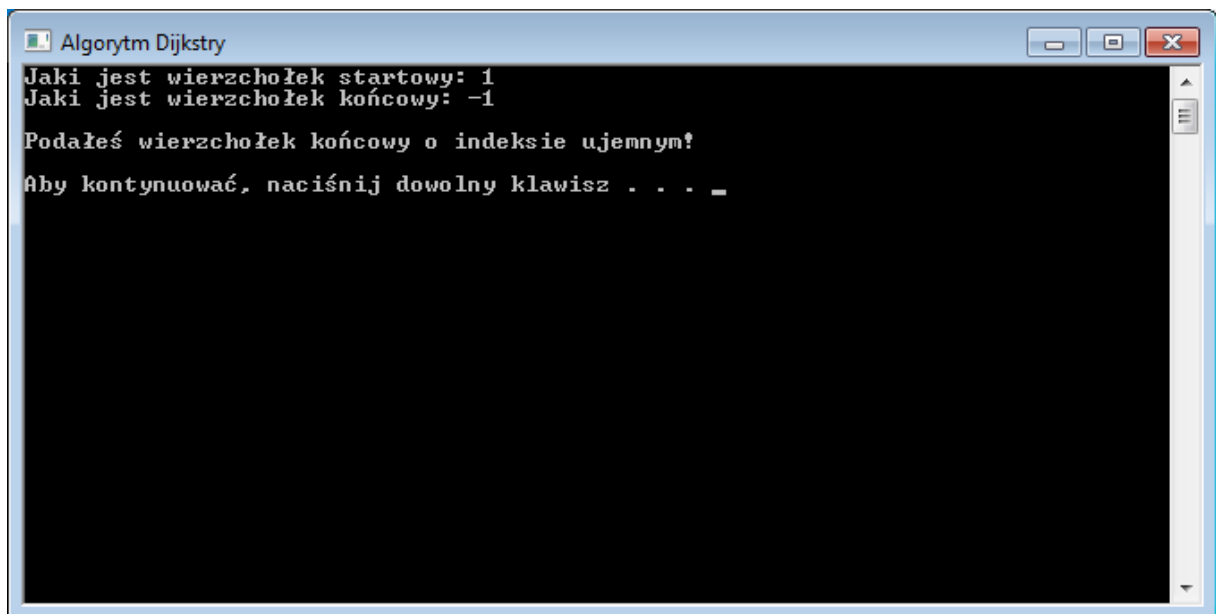
Źródło: opracowanie własne

Błędnie wpisana wartość wierzchołka finalnego skutkuje odpowiednim komunikatem, jak zostało to wspomniane wcześniej.



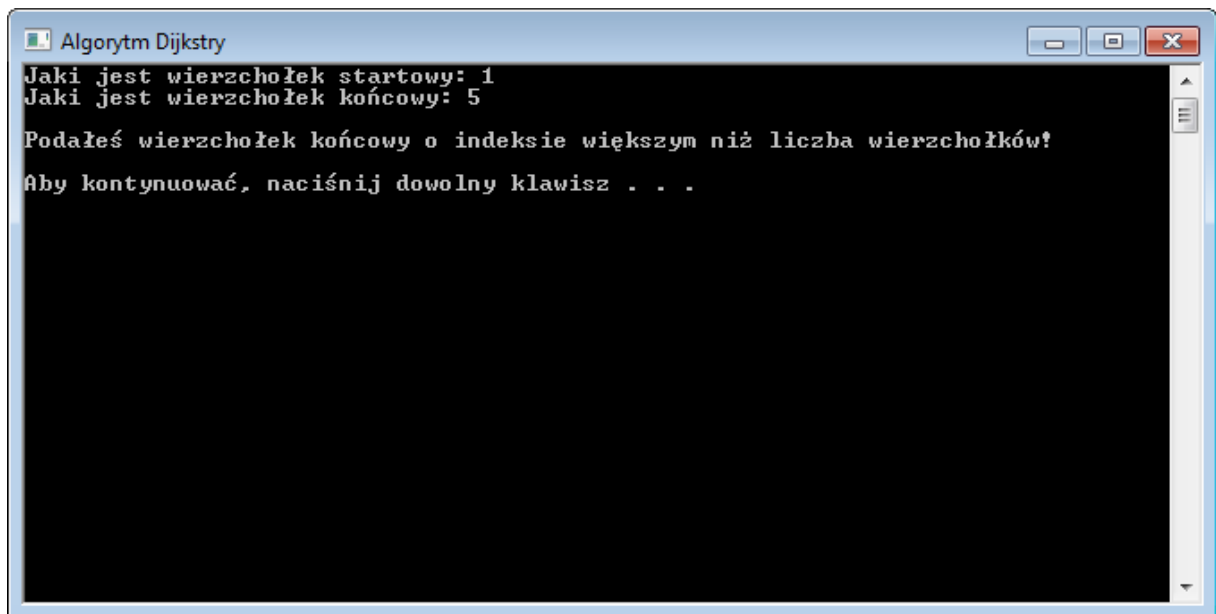
Rysunek 5.11: Błędnie wpisany wierzchołek krańcowy – przypadek 1

Źródło: opracowanie własne



Rysunek 5.12: Błędnie wpisany wierzchołek krańcowy – przypadek 2

Źródło: opracowanie własne



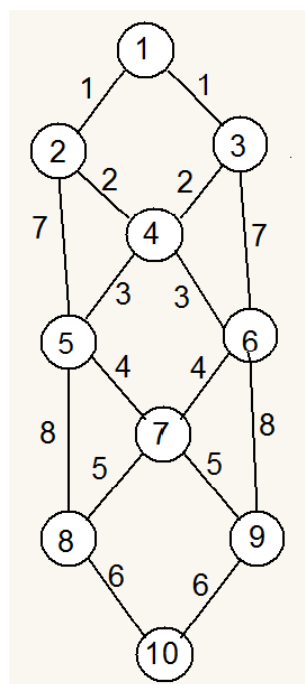
Rysunek 5.13: Błędnie wpisany wierzchołek krańcowy – przypadek 3

Źródło: opracowanie własne

5.4. Testy programu

Bieżący podrozdział przedstawia przeprowadzone testy aplikacji na przykładowych grafach. Zostały stworzone samodzielnie i miały istotny wpływ na to, by zaimplementowanie kodu aplikacji kierowało jej działaniem do końcowego wyświetlenia właściwych wyników zgodnych z założeniami i mechanizmem działania algorytmu Dijkstry.

Graf pierwszy przedstawia się następująco:



Rysunek 5.14: Graf do pliku dane.txt

Źródło: opracowanie własne

Następną kwestią jest zawartość pliku dane.txt, który przechowuje w pierwszej linii liczbę wierzchołków, a od drugiej linii długości pomiędzy poszczególnymi wierzchołkami w postaci macierzy.

dane.txt — Notatnik									
Plik	Edycja	Format	Widok	Pomoc					
10									
0	1	1	9999	9999	9999	9999	9999	9999	9999
1	0	9999	2	7	9999	9999	9999	9999	9999
1	9999	0	2	9999	7	9999	9999	9999	9999
9999	2	2	0	3	3	9999	9999	9999	9999
9999	7	9999	3	0	9999	4	8	9999	9999
9999	9999	7	3	9999	0	4	9999	8	9999
9999	9999	9999	4	4	0	5	5	9999	9999
9999	9999	9999	9999	8	9999	5	0	9999	6
9999	9999	9999	9999	9999	8	5	9999	0	6
9999	9999	9999	9999	9999	9999	9999	6	6	0

Rysunek 5.15: Dane z pliku dane.txt dla grafu 1

Źródło: Opracowanie własne

Na koniec rozwiązanie metodą Dijkstry przeszukiwania najkrótszej ścieżki z wierzchołka startowego 1 do wierzchołka końcowego 10, w którym są cztery najkrótsze drogi. Liczby porządkowe wierzchołków są bardzo ważne w kwestii pokazywania najkrótszych dróg. Jest tu reguła – najpierw najmniejsze indeksy wierzchołków, a potem rosnąco do największych.

```

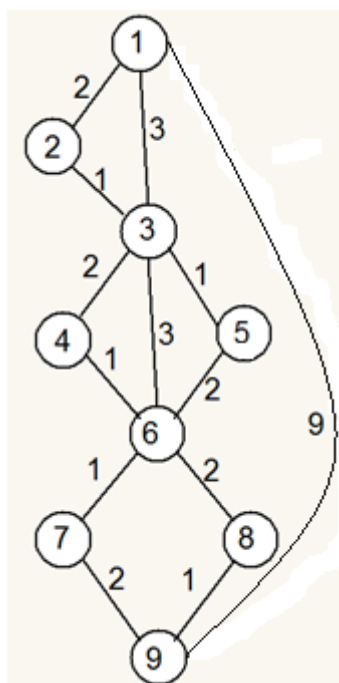
Algorytm Dijkstry
Jaki jest wierzchołek startowy: 1
Jaki jest wierzchołek końcowy: 10
Najkrótszą drogą z wierzchołka 1 do wierzchołka 10 jest :
Ścieżka 1      [1][2][4][5][8][10]
Ścieżka 2      [1][2][4][6][9][10]
Ścieżka 3      [1][3][4][5][8][10]
Ścieżka 4      [1][3][4][6][9][10]
Długość : 20
Aby kontynuować, naciśnij dowolny klawisz . . .

```

Rysunek 5.16: Efekt działania programu dla Grafu pierwszego (dane.txt)

Źródło: opracowanie własne

Graf drugi przedstawia się w następujący sposób:



Rysunek 5.17: Graf do pliku dane2.txt

Źródło: opracowanie własne

Odzwierciedlenie danych wejściowych, czyli liczba wierzchołków i długości krawędzi w postaci macierzowej dla niniejszego grafu przedstawia się w takiej oto postaci.

A screenshot of a Notepad window titled "dane2.txt". The window has a menu bar with "Plik", "Edycja", "Format", "Widok", and "Pomoc". The text content is a 10x10 grid of numbers. The first row is "0 2 3 9999 9999 9999 9999 9999 9". The second row is "2 0 1 9999 9999 9999 9999 9999 9999". The third row is "3 1 0 2 1 3 9999 9999 9999". The fourth row is "9999 9999 2 0 9999 1 9999 9999 9999". The fifth row is "9999 9999 1 9999 0 2 9999 9999 9999". The sixth row is "9999 9999 3 1 2 0 1 2 9999". The seventh row is "9999 9999 9999 9999 9999 1 0 9999 2". The eighth row is "9999 9999 9999 9999 9999 2 9999 0 1". The ninth row is "9 9999 9999 9999 9999 9999 2 1 0".

Rysunek 5.18: Dane z pliku dane2.txt dla grafu 2

Źródło: Opracowanie własne

Koniec rozważań za pomocą aplikacji przedstawia poniższy zrzut ekranu. W tymże grafie jest już znacznie więcej rozwiązań, niż miało to miejsce w poprzedniku. Występuje tutaj również uwzględnienie bezpośredniego połączenia wierzchołków start i koniec w linii ścieżki pierwszej. Ten graf doskonale odzwierciedla sposób, w jaki wyświetlane są wyniki. Podawane są najpierw te ścieżki, które mają w składzie najmniej wierzchołków i odbywa się procedura rosnąco do tych, zawierających najwięcej składowych. Warto podkreślić, że indeksy wierzchołków również mają istotne znaczenie podczas wyświetlania wyników. Podawane są także rosnąco od liczby porządkowej najmniejszej do największej.

```

Algorytm Dijkstry
Jaki jest wierzchołek startowy: 1
Jaki jest wierzchołek końcowy: 9
Najkrótszą drogą z wierzchołka 1 do wierzchołka 9 jest :

Ścieżka 1      [1][9]
Ścieżka 2      [1][3][6][7][9]
Ścieżka 3      [1][3][6][8][9]
Ścieżka 4      [1][3][5][6][7][9]
Ścieżka 5      [1][3][5][6][8][9]
Ścieżka 6      [1][3][4][6][7][9]
Ścieżka 7      [1][3][4][6][8][9]
Ścieżka 8      [1][2][3][6][7][9]
Ścieżka 9      [1][2][3][6][8][9]
Ścieżka 10     [1][2][3][5][6][7][9]
Ścieżka 11     [1][2][3][5][6][8][9]
Ścieżka 12     [1][2][3][4][6][7][9]
Ścieżka 13     [1][2][3][4][6][8][9]

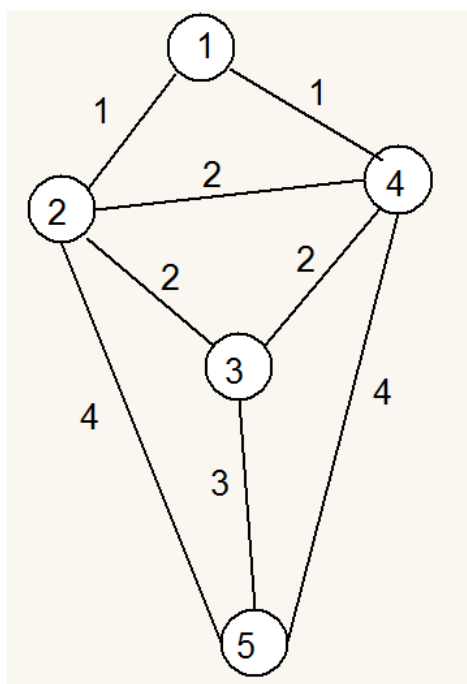
Długość : 9
Aby kontynuować, naciśnij dowolny klawisz . . . _

```

Rysunek 5.19: Efekt działania programu dla Grafu drugiego (dane.txt)

Źródło: opracowanie własne

Graf trzeci wygląda w taki sposób:



Rysunek 5.20: Graf do pliku dane3.txt

Źródło: opracowanie własne

Poniżej znajdują się dane wejściowe w pliku tekstowym. Brak połączenia występuje tylko między wierzchołkiem 1-3 i 1-5.

	1	2	3	4	5
5					
0	1	9999	1	9999	
1	0	2	2	4	
9999	2	0	2	3	
1	2	2	0	4	
9999	4	3	4	0	

Rysunek 5.21: Dane z pliku dane3.txt dla grafu 3

Źródło: opracowanie własne

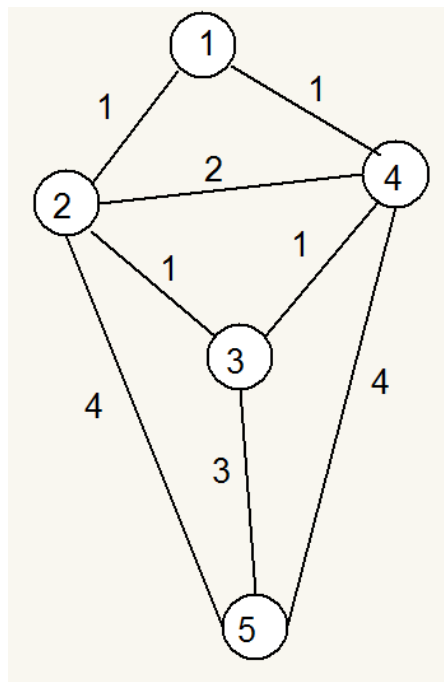
Pod spodem jest ulokowany zrzut ekranowy, przedstawiający jedynie dwa najkrótsze rozwiązania. Tutaj widać, że kolejne liczby porządkowe wierzchołków w wyświetleniu rezultatów odgrywają istotną rolę.


```
Algorytm Dijkstry
Jaki jest wierzchołek startowy: 1
Jaki jest wierzchołek końcowy: 5
Najkrótszą drogą z wierzchołka 1 do wierzchołka 5 jest :
Ścieżka 1      [1][2][5]
Ścieżka 2      [1][4][5]
Długość : 5
Aby kontynuować, naciśnij dowolny klawisz . . . _
```

Rysunek 5.22: Efekt działania programu dla Grafu trzeciego (dane3.txt)

Źródło: opracowanie własne

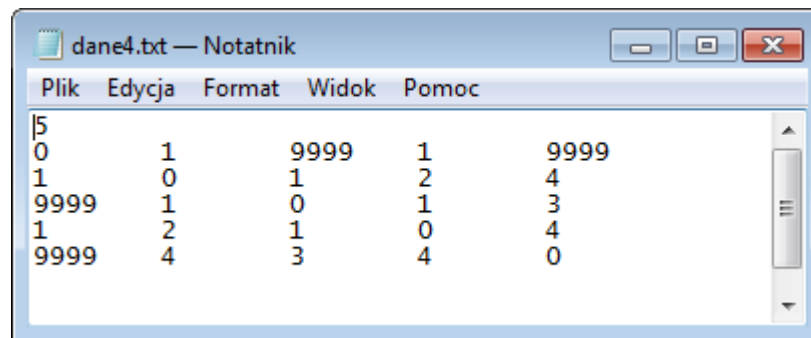
Graf czwarty, choć na pozór taki sam, ma zmienione pewne drobiazgi. Dwa połączenia, którymi są 2-3 i 3-4, mają zmienioną długość z 2 na 1. Jest to działanie celowe, skutkujące tym, że z dwóch rozwiązań są cztery, oczywiście koszt przejazdu równa się wartości 5.



Rysunek 5.23: Graf do pliku dane4.txt

Źródło: opracowanie własne

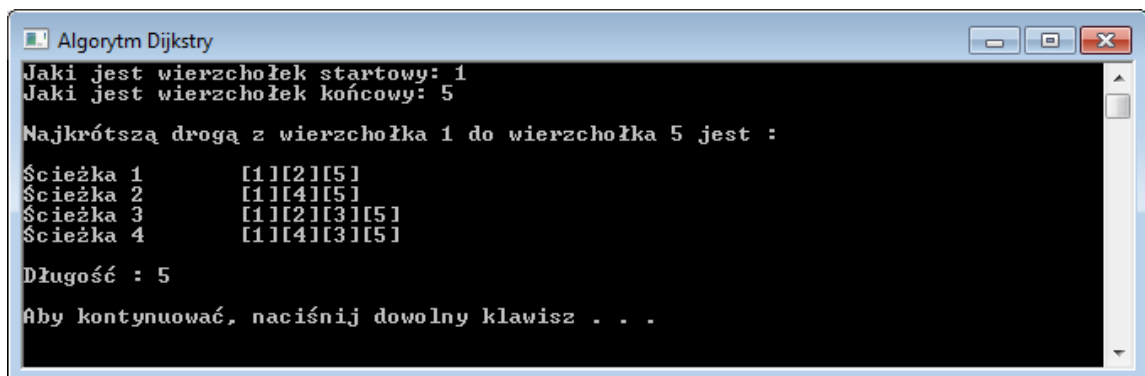
Niżej znajduje się zmodyfikowany plik tekstowy z ważnymi danymi do obliczeń, uwzględniając także zmianę nazwy z dane3.txt na dane4.txt.



Rysunek 5.24: Dane z pliku dane4.txt dla grafu 4

Źródło: opracowanie własne

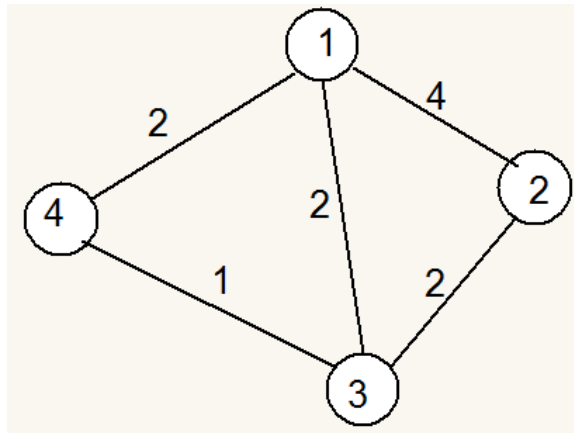
Na dole znajduje się efekt sporządzonych obliczeń, wykonanych przez aplikację, wyświetlone wyniki ponownie ukazują hierarchię wyświetlania najkrótszych dróg, omówionych podczas rozważań nad grafem nr 3. Najpierw decyduje liczba wierzchołków składowych najkrótszego rozwiązania, a potem dopiero ich pojedyncze liczby porządkowe, czyli indeksy.



Rysunek 5.25: Efekt działania programu dla Grafu czwartego (dane4.txt)

Źródło: opracowanie własne

Jako piąty przykład wchodzi na warsztat prosty graf, w którym na wejście wchodzi w odróżnieniu od poprzedników nieco inne dane wejściowe. Rozwijając powyższe stwierdzenie, chodzi konkretnie o wierzchołek końcowy, będący nie tym, co jest ostatni w kolejności, lecz jest ulokowany wewnątrz, czyli nie na krańcowych częściach zbioru.



Rysunek 5.26: Graf do pliku dane5.txt

Źródło: opracowanie własne

Dane wejściowe w pliku tekstowym ukazują informację, iż jest brak połączenia pomiędzy wierzchołkami 2 i 4.

Plik	Edycja	Format	Widok	Pomoc
4				
0	4	2	2	
4	0	2	9999	
2	2	0	1	
2	9999	1	0	

Rysunek 5.27: Dane z pliku dane5.txt dla grafu 5

Źródło: opracowanie własne

Rozwiązanie algorytmu podkreśla w trywialny sposób regułę wyświetlania wyników.

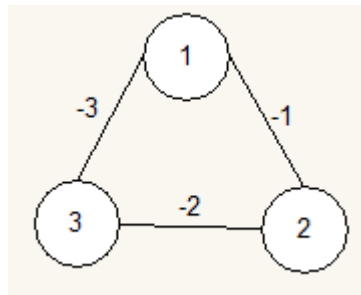
```

Algorytm Dijkstry
Jaki jest wierzchołek startowy: 1
Jaki jest wierzchołek końcowy: 2
Najkrótszą drogą z wierzchołka 1 do wierzchołka 2 jest :
Ścieżka 1      [1][2]
Ścieżka 2      [1][3][2]
Długość : 4
Aby kontynuować, naciśnij dowolny klawisz . . . _
  
```

Rysunek 5.28: Efekt działania programu dla Grafu czwartego (dane4.txt)

Źródło: opracowanie własne

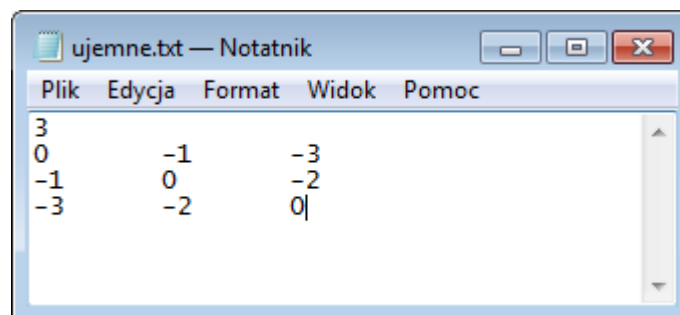
Szósty przykład grafu jest prymitywną siecią, w której są ujemne długości krawędzi. Powstaje zaduma nad tym, jakie będzie rozwiązanie, skoro algorytm Dijkstry nie działa dla takich grafów. Tenże model został celowo sporządzony, by zaprezentować reakcję aplikacji na tę niezgodność.



Rysunek 5.29: Graf do pliku ujemne.txt

Źródło: Opracowanie własne

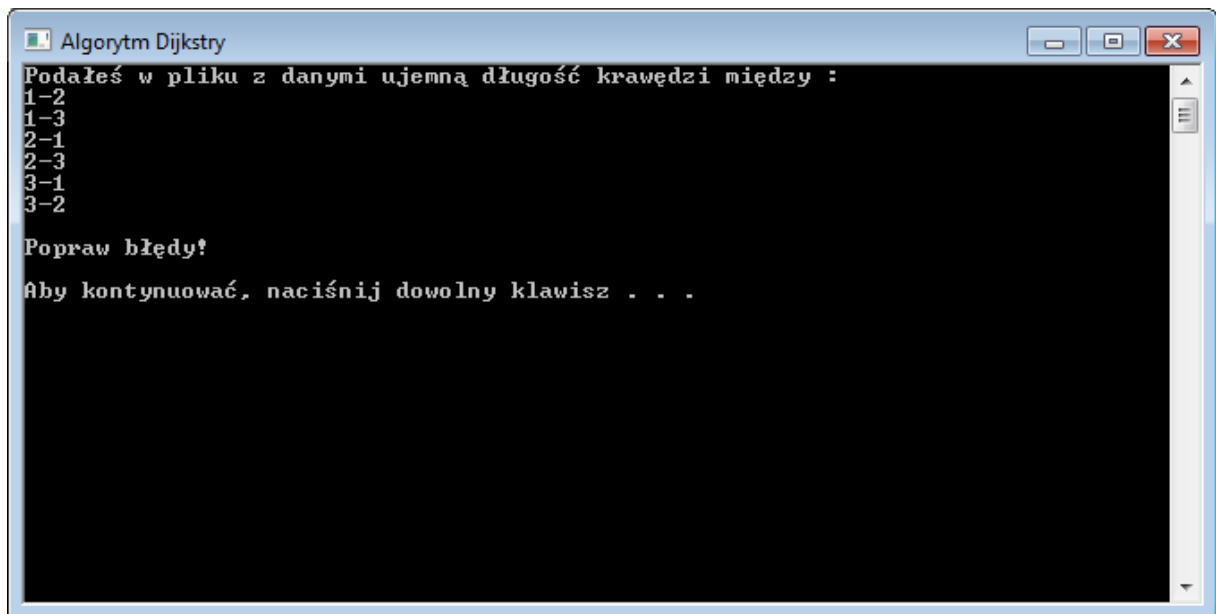
Odpowiednio prezentuje się plik z danymi wejściowymi, gdzie są ujemne wartości krawędzi pomiędzy wierzchołkami.



Rysunek 5.30: Dane z pliku ujemne.txt dla grafu 6

Źródło: opracowanie własne

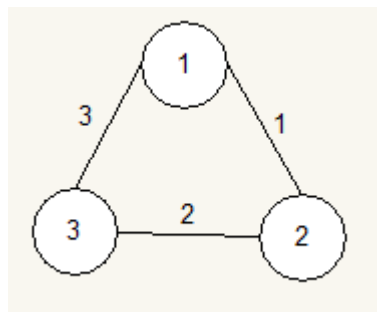
Użytkownik jest poinformowany o tym, że dane są błędne i wyświetla, gdzie są błędy, następnie program zostaje zamknięty.



Rysunek 5.31: Efekt działania programu dla Grafu z ujemnymi długościami krawędzi
(ujemne.txt)

Źródło: opracowanie własne

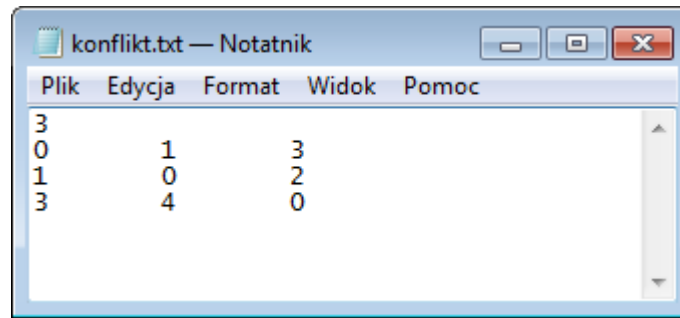
Pozostaje jeszcze jeden, ostatni przykład siódmego grafu, który przedstawia trywialny model, który nie sprawi żadnego problemu aplikacji w obliczeniach, lecz użytkownik popełnił błąd podczas wpisywania wartości odległości i powstaje rozterka nad tym, czy program wykona algorytm czy nie. Otóż nie wykona przy konflikcie danych, a użytkownik musi poprawić swój błąd.



Rysunek 5.32: Graf do pliku konflikt.txt

Źródło: Opracowanie własne

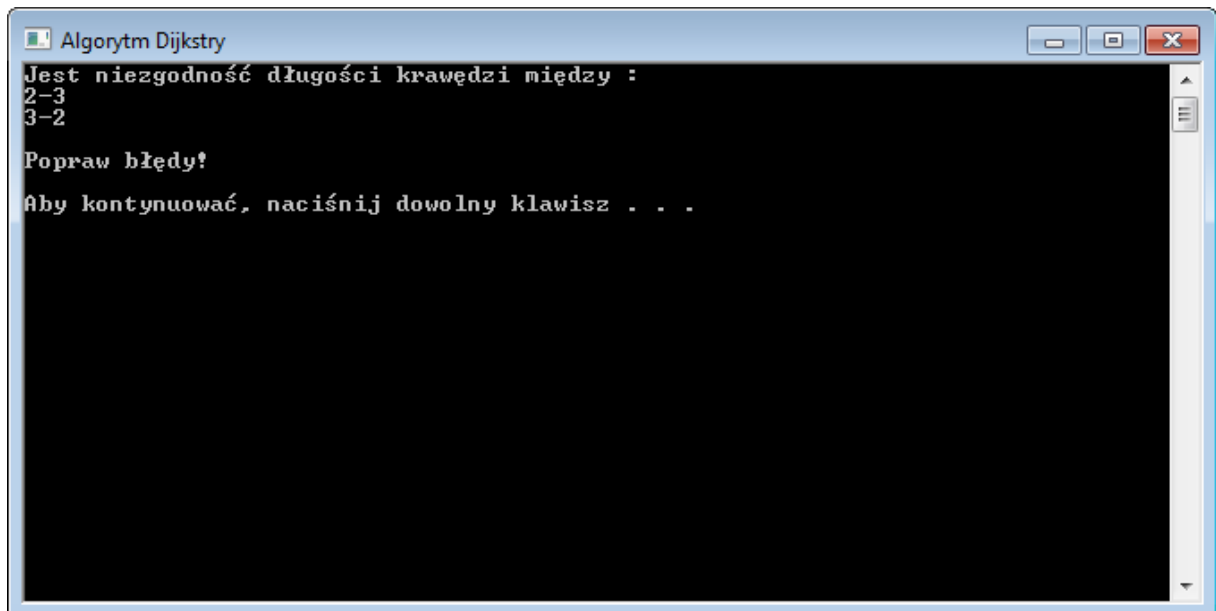
Pora na przedstawienie danych wejściowych i błąd w czwartym wierszu i drugiej kolumnie. Użytkownik pomylił się i wpisał zamiast wartości dwa liczbę cztery.



Rysunek 5.33: Dane z pliku konflikt.txt dla grafu 7

Źródło: opracowanie własne

Stosowny komunikat wyświetli program po niefortunnym błędzie użytkownika.



Rysunek 5.34: Efekt działania programu dla Grafu z konfliktem długości krawędzi (konflikt.txt)

Źródło: opracowanie własne

Aplikacja nie jest doskonała, mogą zdarzyć się grafy, przy których niezgodności mogą nie zostać wychwycone. Jednak podjęto wszelkie kroki, by zmniejszyć to ryzyko do poziomu minimum.

6. Podsumowanie

W tejże pracy zostały poruszone aspekty, dotyczące metody odnajdywania najkrótszych ścieżek w grafie, którą stworzył holenderski informatyk Edsger Dijkstra. Do implementacji tegoż algorytmu posłużył język wysokiego poziomu C++, który został także omówiony.

Wskutek przytoczonego opracowania i dogłębnej analizy literatury, osiągalnych pomocy dydaktycznych powstała aplikacja, stanowiąca odzwierciedlenie schematu odkrywania najkrótszych dróg, sporządzonego przez Dijkstrę. Program wyświetla nie tylko długość, inaczej koszt dotarcia od punktu startu do mety, pokazuje wszystkie rozwiązania, składające się z pojedynczych wierzchołków, tworzących ścieżki. Zaprogramowanie aplikacji, by odnalazła dla danych długości pomiędzy poszczególnymi wierzchołkami w zbiorze o skończonej liczbie elementów i wyświetliła je w oknie konsoli, było największym wyzwaniem i zarazem najtrudniejszym w całym projekcie.

Literatura

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.: *Wprowadzenie do algorytmów.* , Wydanie VII, Wydawnictwo Naukowe PWN, Warszawa 2013, ISBN 978-83-01-16911-4.
2. Czech Z. J., Deorowicz S., Fabian P.: *Algorytmy i struktury danych. Wybrane zagadnienia.* Wydawnictwo Politechniki Śląskiej, Gliwice 2010, ISBN 978-83-7335-668-9.
3. Dasgupta S., Papadimitriou C., Vazirani U.: *Algorytmy.* Wydawnictwo Naukowe PWN, Warszawa 2010, ISBN 978-83-01-16278-8.
4. Prata S.: *Język C++. Szkoła programowania.*, Wydanie VI, Wydawnictwo Helion, Gliwice 2013, ISBN 978-83-246-4336-3.
5. Sedgewick R., Wayne K.: *Algorytmy.*, Wydanie IV, Wydawnictwo Helion, Gliwice 2012, ISBN 978-83-246-3536-8.
6. Stańczyk P.: *Algorytmika praktyczna. Nie tylko dla mistrzów.* Wydawnictwo Naukowe PWN, Warszawa 2009, ISBN 978-83-01-15821-7.
7. Stroustrup B.: *Język C++. Kompendium wiedzy.*, Wydanie IV, Wydawnictwo Helion, Gliwice 2014, ISBN 978-83-246-8530-1.
8. Wróblewski P.: *Algorytmy i struktury danych i techniki programowania.*, Wydanie IV, Wydawnictwo Helion, Gliwice 2010, ISBN 978-83-246-2306-8.

Dodatek A Spis zawartości dołączonej płyty CD

1. Praca inżynierska – niniejszy dokument
2. Kod źródłowy aplikacji wraz z plikami tekstowymi, zawierającymi dane na temat liczby wierzchołków i długościami krawędzi między wierzchołkami grafów, rozważanych w rozdziale Testy programu na przykładowych grafach.