

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Algorytmy w C

Autor: Kyle Loudon

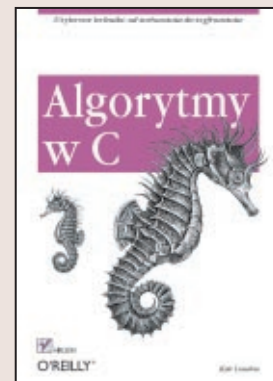
Tłumaczenie: Tomasz Żmijewski

ISBN: 83-7197-912-6

Tytuł oryginału: [Mastering Algorithms with C](#)

Format: B5, stron: 492

[Przykłady na ftp: 272 kB](#)



Książka „Algorytmy w C” jest doskonałą pomocą dla programistów, którym w codziennej pracy potrzebne są sprawdzone rozwiązania. Nie ma tu teoretycznych dywagacji tak charakterystycznych dla większości książek o strukturach danych i algorytmach. Znajdziesz w niej za to przystępnie podane informacje i praktyczne techniki programowania.

Wyjątkowo elegancki styl programowania i pisania autora, Kyle’a Loudona, ułatwia poznanie najważniejszych struktur danych, takich jak listy, stosy, kolejki, zbiory, sterty, kolejki priorytetowe i grafy. Autor prezentuje użycie algorytmów sortujących, wyszukiwania, analiz numerycznych, kompresji danych, szyfrowania danych, typowych algorytmów obsługi grafów oraz geometrii analitycznej. W rozdziałach poświęconych kompresji i szyfrowaniu czytelnik znajdzie nie tylko gotowy, szybki w działaniu kod, ale też informacje przydatne dla osób, które nigdy nie miały czasu ani chęci zagłębiać się w omawiane zagadnienia.

W tekście umieszczono także kody wraz z przykładami zastosowania poszczególnych struktur danych i algorytmów. Komplet kodów źródłowych znajduje się na płycie CD-ROM. Kod ten został napisany w taki sposób, byś łatwo mógł go wykorzystać we własnych aplikacjach.

W książce omówiono:

- Wskaźniki
- Rekurencję
- Analizę algorytmów
- Struktury danych (listy, stosy, kolejki, zbiory, tablice asocjacyjne, drzewa, sterty, kolejki priorytetowe i grafy)
- Sortowanie i wyszukiwanie
- Metody numeryczne
- Kompresję danych
- Szyfrowanie danych
- Algorytmy operujące na grafach



Spis treści

Wstęp	9
Część I Zagadnienia podstawowe	15
Rozdział 1. Wprowadzenie	17
Wprowadzenie do struktur danych	18
Wprowadzenie do algorytmów	19
Odrobina inżynierii oprogramowania	22
Jak tej książki używać	23
Rozdział 2. Użycie wskaźników	25
Podstawowe informacje o wskaźnikach	26
Alokacja pamięci	27
Agregaty i arytmetyka wskaźników	29
Wskaźniki jako parametry funkcji	31
Wskaźniki ogólne i rzutowanie	34
Wskaźniki do funkcji	37
Pytania i odpowiedzi	38
Tematy pokrewne	39
Rozdział 3. Rekurencja	41
Podstawy rekurencji	42
Rekurencja prawostronna	46
Pytania i odpowiedzi	48
Tematy pokrewne	50
Rozdział 4. Analiza algorytmów	51
Analiza najgorszego przypadku	52
O-notacja	53
Złożoność obliczeniowa	55

Przykład analizy: sortowanie przez wstawianie	57
Pytania i odpowiedzi	59
Tematy pokrewne	60
Część II Struktury danych	61
Rozdział 5. Listy powiązane	63
Opis list powiązanych	64
Interfejs list powiązanych	65
Implementacja list powiązanych i analiza kodu	68
Przykład użycia list powiązanych: zarządzanie ramkami	75
Opis list podwójnie powiązanych	78
Interfejs list podwójnie powiązanych	79
Implementacja list podwójnie powiązanych i analiza kodu	81
Opis list cyklicznych	90
Interfejs list cyklicznych	91
Implementacja list cyklicznych i analiza kodu	93
Przykład użycia list cyklicznych: zamiana stron	98
Pytania i odpowiedzi	101
Tematy pokrewne	103
Rozdział 6. Stosy i kolejki	105
Opis stosów	106
Interfejs stosu	107
Implementacja stosu i analiza kodu	108
Opis kolejek	111
Interfejs kolejek	111
Implementacja kolejki i analiza kodu	113
Przykład użycia kolejek: obsługa zdarzeń	116
Pytania i odpowiedzi	118
Tematy pokrewne	119
Rozdział 7. Zbiory	121
Opis zbiorów	122
Interfejs zbiorów	124
Implementacja zbioru i analiza kodu	127
Przykład użycia zbiorów: pokrycie	137
Pytania i odpowiedzi	141
Tematy pokrewne	143
Rozdział 8. Tablice asocjacyjne	145
Opis łańcuchowych tablic asocjacyjnych	147
Interfejs łańcuchowych tablic asocjacyjnych	150
Implementacja łańcuchowej tablicy asocjacyjnej i analiza kodu	152
Przykład użycia łańcuchowych tablic asocjacyjnych: tablice symboli	159
Tablice asocjacyjne z otwartym adresowaniem	162
Interfejs tablic asocjacyjnych z otwartym adresowaniem	166

Implementacja tablicy asocjacyjnej z otwartym adresowaniem i analiza kodu	167
Pytania i odpowiedzi	175
Tematy pokrewne	177
Rozdział 9. Drzewa	179
Drzewa binarne	181
Interfejs drzew binarnych	184
Implementacja drzew binarnych i analiza kodu	187
Przykład użycia drzewa binarnego: przetwarzanie wyrażeń	198
Binarne drzewa wyszukiwania	201
Interfejs binarnych drzew wyszukiwania	203
Implementacja binarnych drzew wyszukiwania i analiza kodu	205
Pytania i odpowiedzi	227
Tematy pokrewne	228
Rozdział 10. Sterty i kolejki priorytetowe	231
Sterty	232
Interfejs stert	233
Implementacja stert i analiza kodu	235
Kolejki priorytetowe	245
Interfejs kolejek priorytetowych	245
Implementacja kolejek priorytetowych i analiza kodu	247
Przykład zastosowania kolejek priorytetowych: sortowanie paczek	248
Pytania i odpowiedzi	250
Tematy pokrewne	251
Rozdział 11. Grafy	253
Grafy	254
Interfejs grafów	260
Implementacja grafów i analiza kodu	264
Przykład użycia grafów: zliczanie kroków w ruchu po sieci	275
Przykład użycia grafów: sortowanie topologiczne	281
Pytania i odpowiedzi	285
Tematy pokrewne	287
Część III Algorytmy	289
Rozdział 12. Sortowanie i przeszukiwanie	291
Sortowanie przez wstawianie	293
Interfejs sortowania przez wstawianie	293
Implementacja sortowania przez wstawianie i analiza kodu	294
Quicksort	296
Interfejs quicksort	297
Implementacja quicksort i analiza kodu	298
Przykład użycia quicksort: zawartość katalogu	303
Sortowanie przez złączanie	306
Interfejs sortowania przez złączanie	307
Implementacja sortowania przez złączanie i analiza kodu	307

Sortowanie ze zliczaniem	312
Interfejs sortowania ze zliczaniem	313
Implementacja sortowania ze zliczaniem i analiza kodu	313
Sortowanie na bazie	317
Interfejs sortowania na bazie	317
Implementacja sortowania na bazie i analiza kodu	318
Wyszukiwanie binarne	321
Interfejs wyszukiwania binarnego	321
Implementacja wyszukiwania binarnego i analiza kodu	322
Przykład użycia przeszukiwania binarnego: kontrola pisowni	324
Pytania i odpowiedzi	326
Tematy pokrewne	328
Rozdział 13. Metody numeryczne	329
Przybliżanie wielomianami	330
Interfejs interpolacji wielomianowej	334
Implementacja interpolacji wielomianowej i analiza kodu	334
Oszacowanie najmniejszymi kwadratami	337
Interfejs oszacowania najmniejszymi kwadratami	339
Implementacja szacowania najmniejszymi kwadratami i analiza kodu	339
Rozwiązywanie równań	340
Interfejs rozwiązywania równań	345
Implementacja rozwiązywania równań i analiza kodu	345
Pytania i odpowiedzi	347
Tematy pokrewne	348
Rozdział 14. Kompresja danych	349
Operatory bitowe	352
Interfejs operacji bitowych	353
Implementacja operacji bitowych i analiza kodu	354
Kodowanie Huffmana	358
Interfejs kodowania Huffmana	362
Implementacja kodowania Huffmana i analiza kodu	362
Przykład użycia kodowania Huffmana: optymalizacja przesyłania przez sieć	376
Algorytm LZ77	378
Interfejs LZ77	382
Implementacja LZ77 i analiza kodu	382
Pytania i odpowiedzi	395
Tematy pokrewne	397
Rozdział 15. Szyfrowanie danych	399
DES	402
Interfejs DES	409
Implementacja DES i analiza kodu	410
Przykład użycia DES: blokowe tryby szyfrowania	420
Algorytm RSA	423
Interfejs RSA	426

Implementacja RSA i analiza kodu.....	427
Pytania i odpowiedzi	430
Tematy pokrewne.....	432
Rozdział 16. Algorytmy operujące na grafach.....	435
Drzewa minimalne	438
Interfejs drzew minimalnych.....	439
Implementacja i analiza kodu.....	441
Najkrótsze ścieżki	446
Interfejs najkrótszych ścieżek	447
Implementacja najkrótszej ścieżki i analiza kodu	449
Przykład użycia najkrótszych ścieżek: tablic tras.....	454
Problem komiwojażera	457
Interfejs problemu komiwojażera	459
Implementacja problemu komiwojażera i analiza kodu	460
Pytania i odpowiedzi	464
Tematy pokrewne.....	465
Rozdział 17. Algorytmy geometryczne	467
Sprawdzanie, czy odcinki się przecinają	470
Interfejs sprawdzania, czy odcinki się przecinają	473
Implementacja sprawdzenia, czy odcinki się przecinają i analiza kodu	473
Obrys wypukły	475
Interfejs obrysów wypukłych	477
Implementacja obrysu wypukłego i analiza kodu	478
Długość łuku na powierzchniach sferycznych	482
Interfejs długości łuku na powierzchniach sferycznych	485
Implementacja długości łuku na powierzchniach sferycznych i analiza kodu	485
Przykład użycia długości łuku: przybliżone określanie odległości na Ziemi	486
Pytania i odpowiedzi	489
Tematy pokrewne.....	492
Skorowidz	493

16

Algorytmy operujące na grafach

Grafy to elastyczne struktury danych pozwalające modelować problemy jako relacje czy powiązania między obiektami (więcej na ten temat w rozdziale 11.). W tym rozdziale przedstawimy algorytmy operujące na grafach. Jak zobaczymy, wiele z tych algorytmów przypomina podstawowe algorytmy przeszukiwania grafów wszerz i w głąb, opisane w rozdziale 11. Te dwie grupy algorytmów są ważne dla innych algorytmów operujących na grafach, jako że pozwalają systematycznie przeglądać graf.

Istotna różnica między algorytmami z tego rozdziału i rozdziału 11. polega na tym, że algorytmy tego rozdziału dotyczą *grafów z wagami*. W takich grafach każdej krawędzi przypisuje się pewną wartość, *wagę*, która wizualnie jest zaznaczana jako niewielka liczba przy krawędzi. Wprawdzie wagi mogą oznaczać różne rzeczy, ale zwykle opisują koszt związany z przejściem danej krawędzi. Grafy z wagami i działające na nich algorytmy pozwalają opisywać ogromną liczbę różnorodnych problemów. Listing 16.1 to nagłówek do algorytmów operujących na grafach, zaprezentowanych w tym rozdziale.

Listing 16.1. Plik nagłówkowy algorytmów operujących na grafach

```
/*****
 *
 * ----- graphalg.h -----
 *
 *****/

#ifndef GRAPHALG_H
#define GRAPHALG_H

#include "graph.h"
#include "list.h"
```

```

/*****
 *
 *   Struktura na węzły drzew minimalnych.
 *
 *****/

typedef struct MstVertex_ {

    void          *data;
    double         weight;

    VertexColor    color;
    double         key;

    struct MstVertex_ *parent;

} MstVertex;

/*****
 *
 *   Struktura na węzły najkrótszej ścieżki.
 *
 *****/

typedef struct PathVertex_ {

    void          *data;
    double         weight;

    VertexColor    color;
    double         d;

    struct PathVertex_ *parent;

} PathVertex;

/*****
 *
 *   Struktura na węzły problemu komiwojażera.
 *
 *****/

typedef struct TspVertex_ {

    void          *data;

    double         x,
                  y;

    VertexColor    color;

} TspVertex;

/*****
 *
 *   ----- Interfejs publiczny -----
 *
 *****/

int mst(Graph *graph, const MstVertex *start, List *span, int (*match)(const
    void *key1, const void *key2));

```

```
int shortest(Graph *graph, const PathVertex *start, List *paths, int (*match)
(const void *key1, const void *key2));

int tsp(List *vertices, const TspVertex *start, List *tour, int (*match)
(const void *key1, const void *key2));

#endif
```

W tym rozdziale omówimy:

Drzewa minimalne

Drzewa będące abstrakcjami wielu problemów związanych z połączeniami. Drzewo minimalne to takie drzewo, które łączy wszystkie węzły w nieskierowanym grafie z wagami możliwie najmniejszym kosztem.

Najkrótsze ścieżki

Wynik rozwiązywania różnego rodzaju problemów najkrótszej drogi. Najkrótsza droga lub ścieżka to droga w grafie skierowanym z wagami dwa węzły przy minimalnym koszcie.

Problem komiwojażera

Zaskakująco trudny problem, w którym szukamy najkrótszej drogi pozwalającej odwiedzić wszystkie węzły zupełnego, nieskierowanego grafu z wagami tak, aby przed powrotem do punktu startowego każdy węzeł został odwiedzony dokładnie raz.

Oto wybrane zastosowania algorytmów operujących na grafach:

Dobrze dobrane sieci przesyłowe

Praktyczne zagadnienie dotyczące przesyłania wody, ropy i innych cieczy. Jeśli punkty odbioru medium z sieci zostaną przedstawione jako węzły grafu, zaś potencjalne połączenia jako krawędzie tego grafu z wagami odpowiadającymi kosztowi połączenia punktów, drzewo minimalne wskazuje optymalny przebieg sieci łączącej wszystkie punkty odbioru.

Tablice tras (przykład w rozdziale)

Tablice używane przez routery do przesyłania danych w Internecie. Zadaniem routera jest przesłanie danych bliżej ich miejsca docelowego. W przypadku jednego ze sposobów wyznaczania tras routery okresowo wyliczają najkrótsze ścieżki między sobą, dzięki czemu każdy z nich potrafi wykonać następny etap przesłania danych.

Usługi dostawcze

Usługi związane zwykle z odwiedzaniem licznych miejsc w celu odbierania i dostarczania paczek. Rozwiązanie problemu komiwojażera pozwala wskazać najlepszą drogę kuriera, który przed powrotem do bazy musi odwiedzić wszystkie punkty.

Sieci komunikacyjne

Sieci zawierające wiele różnego rodzaju sprzętu: linie telefoniczne, stacje przekąźnikowe i systemy satelitarne. Wszystkie te urządzenia muszą zostać optymalnie rozmieszczone. Optymalny rozkład można obliczyć przez wyznaczenie drzewa minimalnego grafu z wagami opisującego sieć.

Kierowanie samolotów

Problem optymalizacyjny szczególnie istotny w liniach lotniczych i agencjach kontroli lotu. Samoloty często nie mogą przemieszczać się bezpośrednio między wyznaczonymi punktami, gdyż muszą korzystać z korytarzy powietrznych (takich lotniczych autostrad), poza tym obowiązują ograniczenia nałożone przez kontrolerów lotów. Optymalna droga między dwoma punktami to ścieżka z najmniejszymi wagami.

System transportu zamkniętego

Systemy, w których pojazdy szynowe lub wózki wielokrotnie poruszają się między tymi samymi punktami. Takie systemy mogą być użyte do rozwożenia części w fabryce lub do przenoszenia towarów w magazynie. Rozwiązanie problemu komiwojażera pozwala znaleźć optymalny układ systemu.

Mostkowanie obwodów elektrycznych

Problem optymalizacyjny związany z produkcją elektroniki. Często trzeba połączyć ze sobą punkty obwodu przy użyciu grafu, w którym poszczególnym punktom odpowiadają węzły. Potencjalne połączenia modelowane są jako krawędzie. Rozwiązanie optymalne znowu wskazuje drzewo minimalne.

Monitorowanie ruchu kołowego

Obserwowanie zmian ruchu zmierzające do wyznaczenia najlepszej trasy poruszania się po mieście. Aby uniknąć nadmiernych opóźnień związanych z korkami, używa się modelu zawierającego informacje o połączeniach i bada się przecięcia z najmniejszym ruchem.

Drzewa minimalne

Wyobraźmy sobie pinezki powbijane w tablicę i połączone sznurkiem. Przy założeniu, że wędrując po sznurku możemy dotrzeć do wszystkich pinezek, wyobraźmy sobie grę, której celem jest usuwanie sznurków dotąd, aż pinezki będą połączone najmniejszą możliwą liczbą sznurków. To jest właśnie idea *drzew minimalnych*. Formalnie, jeśli mamy nieskierowany graf z wagami $G = (V, E)$, jego drzewem minimalnym jest zbiór T krawędzi ze zbioru E łączący wszystkie węzły z V przy możliwie najmniejszym koszcie. Krawędzie z T tworzą drzewo, gdyż każdy węzeł ma dokładnie jednego rodzica — węzeł poprzedni, z wyjątkiem węzła początkowego, który jest korzeniem drzewa.

Algorytm Prima

Jedną z metod wyznaczania drzew minimalnych jest *algorytm Prima*. Algorytm ten rozpina drzewo minimalne dodając kolejno krawędzie według ich chwilowej wartości, zatem zaliczamy go do grupy algorytmów zachłannych (rozdział 1.). Wprawdzie algorytmy zachłanne często zamiast najlepszych rozwiązań dają jedynie ich przybliżenia, jednak algorytm Prima daje faktycznie rozwiązanie optymalne.

Algorytm ten działa poprzez wielokrotne wybieranie wężła i badanie jego krawędzi przylegających w celu sprawdzenia, czy istnieje szybsza droga połączenia zbadanych dotąd węzłów. Algorytm ten przypomina przeszukiwanie wszerz, gdyż badane są wszystkie krawędzie sąsiednie wężła i dopiero wtedy robione jest przejście głębiej w strukturę grafu. Wartość kluczowa wężła, od którego zaczynamy szukanie to 0. Aby wybrać odpowiedni węzeł na każdym etapie działania algorytmu, zapamiętujemy kolor i wartość klucza dla każdego wężła.

Początkowo wszystkie węzły są białe, wszystkie wartości kluczy mają wartość ∞ : dowolnie dużą liczbę, większą od wag wszystkich krawędzi w grafie. Ustawiamy wartość klucza wężła, od którego zaczynamy na 0. W miarę działania algorytmu przypisujemy wszystkim węzłom poza węzłem początkowym rodzica w drzewie minimalnym. Węzeł jest częścią drzewa minimalnego tylko wtedy, gdy stanie się czarny; przed tym jego rodzic może się zmieniać.

Dalej algorytm Prima działa następująco: najpierw, spośród wszystkich białych węzłów grafu wybieramy jeden węzeł u , który ma najmniejszą wartość klucza. Początkowo będzie to węzeł startowy, gdyż ma wartość 0. Po wybraniu wężła malujemy go na czarno. Następnie, dla każdego białego wężła v przylegającego do u , jeśli waga krawędzi (u, v) jest mniejsza od wartości klucza v , ustawiamy wartość klucza v na wagę (u, v) i ustalamy u jako rodzica v . Proces powtarzamy dotąd, aż zaczernione zostaną wszystkie węzły. W miarę wzrostu drzewa minimalnego wchodzi do niego wszystkie krawędzie mające na dowolnym końcu czarne węzły.

Na rysunku 16.1 pokazano wyliczanie drzewa minimalnego za pomocą algorytmu Prima. Wartości klucza i rodzic pokazywane są obok wężła. Wartość klucza jest na lewo od ukośnika, rodzic na prawo. Wyszarzone krawędzie to krawędzie rosnącego drzewa minimalnego. Drzewo minimalne przedstawione na rysunku ma łączną wagę 17.

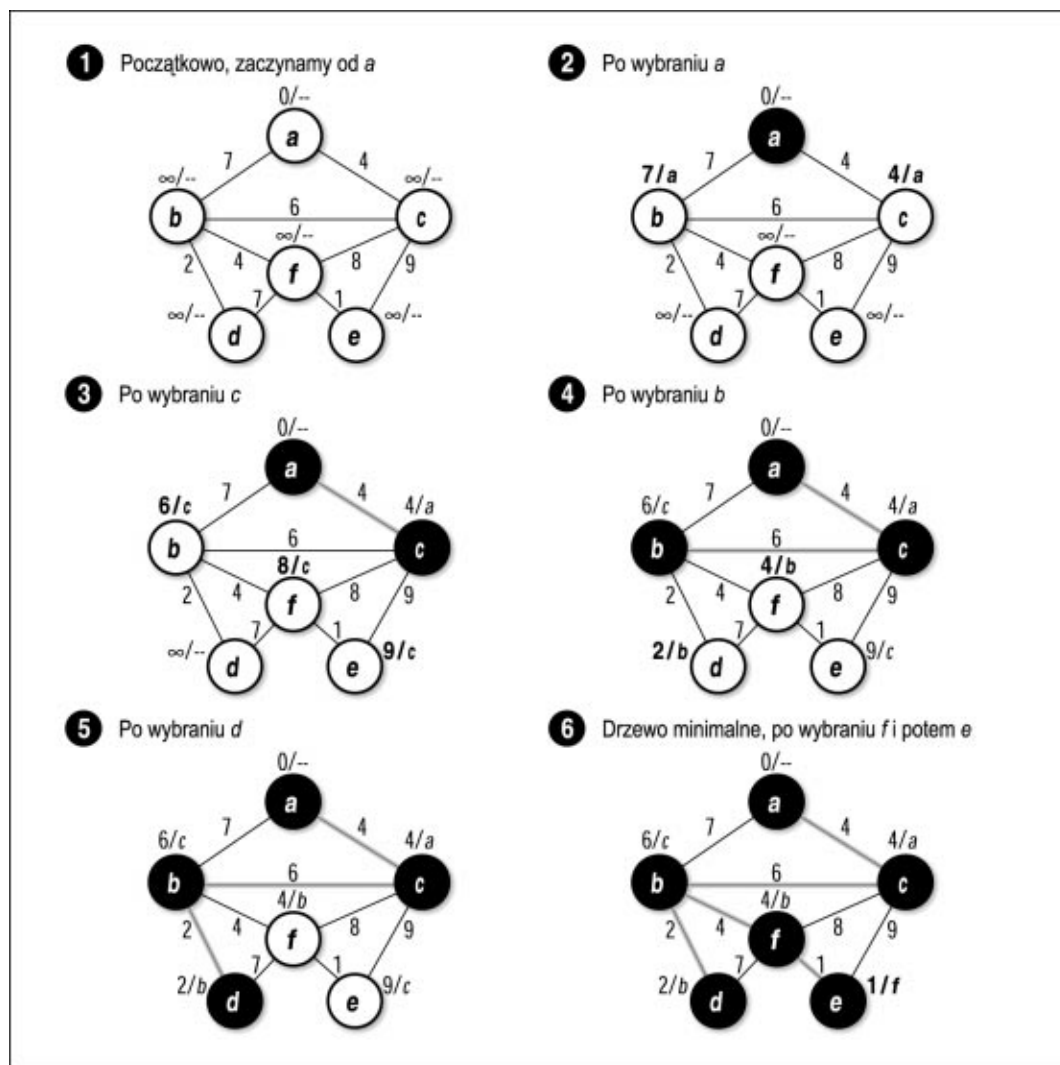
Interfejs drzew minimalnych

mst

```
int mst(Graph *graph, const MstVertex *start, List *span, int (*match)
      (const void *key1, const void *key2));
```

Zwracana wartość: 0, jeśli wyznaczanie drzewa minimalnego się powiodło, i -1 w przeciwnym razie.

Opis: Wyliczane jest drzewo minimalne nieskierowanego grafu z wagami *graph*. Operacja modyfikuje graf, więc w razie potrzeby trzeba zrobić jego kopię. Każdy węzeł grafu *graph* musi zawierać dane typu *MstVertex*. Każdej krawędzi przypisujemy wagę, ustawiając pole *weight* struktury *MstVertex* na wartość przekazaną jako *data2* do *graph_ins_edge*. Korzystając z pola *data* poszczególnych struktur *MstVertex*, zapisujemy dane tego wężła, na przykład jego identyfikator. Funkcja *match* grafu ustawiana przy inicjalizacji grafu za pomocą *graph_init* służy do porównywania jedynie pól *data* struktur *MstVertex*. Jest



Rysunek 16.1. Wyliczanie minimalnego drzewa za pomocą algorytmu Prima

to ta sama funkcja, która przekazywana jest do *mst* jako *match*. Po realizacji algorytmu uzyskane drzewo minimalne jest przekazywane jako *span*; węzeł, którego rodzic to NULL, jest korzeniem całego drzewa. Elementy *parent* pozostałych pól wskazują węzły poprzedzające dany węzeł w drzewie. Węzły ze *span* wskazują faktycznie węzły z grafu *graph*, więc trzeba zapewnić istnienie odpowiednich danych tak długo, jak długo będą potrzebne.

Złożoność: $O(EV^2)$, gdzie V jest liczbą węzłów grafu, a E — liczbą jego krawędzi. Jednak po niewielkiej, przedstawionej dalej poprawce, algorytm działa w czasie $O(E \lg V)$ (zobacz tematy powiązane na końcu tego rozdziału).

Implementacja i analiza kodu

Aby wyznaczyć drzewo minimalne nieskierowanego grafu z wagami, najpierw trzeba zapisać graf z wagami w abstrakcyjnym typie danych przedstawionym w rozdziale 11. Konieczne będzie też pamiętanie informacji potrzebnych w algorytmie Prima, związanych z węzłami i krawędziami. Do tego służy struktura *MstVertex*; używamy jej na węzły grafu, które mają być włączane do drzewa minimalnego (listing 16.2). Struktura ta ma pięć elementów: *data* to dane związane z węzłem, *weight* to waga krawędzi przylegającej do węzła, *color* to kolor węzła, a *key* to wartość klucza węzła, zaś *parent* to rodzic węzła w drzewie minimalnym.

Listing 16.2. Implementacja wyliczania drzewa minimalnego

```

/*****
 *
 * ----- mst.c -----
 *
 *****/

#include <float.h>
#include <stdlib.h>

#include "graph.h"
#include "graphalg.h"
#include "list.h"

/*****
 *
 * ----- mst -----
 *
 *****/

int mst(Graph *graph, const MstVertex *start, List *span, int (*match)(const
    void *key1, const void *key2)) {

    AdjList      *adjlist;

    MstVertex     *mst_vertex,
                  *adj_vertex;

    ListElmt      *element,
                  *member;

    double        minimum;

    int           found,
                  i;

/*****
 *
 *   Inicjalizacja wszystkich węzłów grafu.
 *
 *****/

    found = 0;

    for (element = list_head(&graph_adjlists(graph)); element != NULL; element =
        list_next(element)) {

```

```

mst_vertex = ((AdjList *)list_data(element))->vertex;

if (match(mst_vertex, start)) {

    /*****
    *
    *   Inicjalizacja węzła początkowego.
    *
    *****/

    mst_vertex->color = white;
    mst_vertex->key = 0;
    mst_vertex->parent = NULL;
    found = 1;

}

else {

    /*****
    *
    *   Inicjalizacja węzłów poza węzłem początkowym.
    *
    *****/

    mst_vertex->color = white;
    mst_vertex->key = DBL_MAX;
    mst_vertex->parent = NULL;

}

}

/*****
*
*   Jeśli nie znaleziono węzła początkowego, kończymy.
*
*****/

if (!found)
    return -1;

/*****
*
*   Wyznaczamy drzewo minimalne algorytmem Prima.
*
*****/

i = 0;

while (i < graph_vcount(graph)) {

    /*****
    *
    *   Wybieramy biały węzeł o najmniejszej wartości klucza.
    *
    *****/

    minimum = DBL_MAX;

    for (element = list_head(&graph_adjlists(graph)); element != NULL; element
        = list_next(element)) {

```

```

    mst_vertex = ((AdjList *)list_data(element))->vertex;

    if (mst_vertex->color == white && mst_vertex->key < minimum) {

        minimum = mst_vertex->key;
        adjlist = list_data(element);

    }

}

/*****
*
*   Zaczerniamy wybrany węzeł.
*
*****/

((MstVertex *)adjlist->vertex)->color = black;

/*****
*
*   Przeglądamy wszystkie węzły sąsiadujące z węzłem wybranym.
*
*****/

for (member = list_head(&adjlist->adjacent); member != NULL; member =
    list_next(member)) {

    adj_vertex = list_data(member);

    /*****
    *
    *   Znalezienie węzła sąsiadującego z listy struktur sąsiedztwa.
    *
    *****/

    for (element = list_head(&graph_adjlists(graph)); element != NULL;
        element = list_next(element)) {

        mst_vertex = ((AdjList *)list_data(element))->vertex;

        if (match(mst_vertex, adj_vertex)) {

            /*****
            *
            *   Decydujemy, czy zmieniamy wartość klucza i rodzica węzła
            *   sąsiedniego z listy struktur list sąsiedztwa.
            *
            *****/

            if (mst_vertex->color == white && adj_vertex->weight <
                mst_vertex->key) {

                mst_vertex->key = adj_vertex->weight;
                mst_vertex->parent = adjlist->vertex;

            }

            break;

        }

    }

}

```

```

    }

    /*****
    *
    *   Przygotowujemy się do wyboru następnego wężła.
    *
    *****/

    i++;

}

/*****
*
*   Ładujemy drzewo minimalne na listę.
*
*****/

list_init(span, NULL);

for (element = list_head(&graph_adjlists(graph)); element != NULL; element =
    list_next(element)) {

    /*****
    *
    *   Ładowanie wszystkich czarnych wężłów z listy sąsiedztwa.
    *
    *****/

    mst_vertex = ((AdjList *)list_data(element))->vertex;

    if (mst_vertex->color == black) {

        if (list_ins_next(span, list_tail(span), mst_vertex) != 0) {

            list_destroy(span);
            return -1;

        }

    }

}

return 0;

}

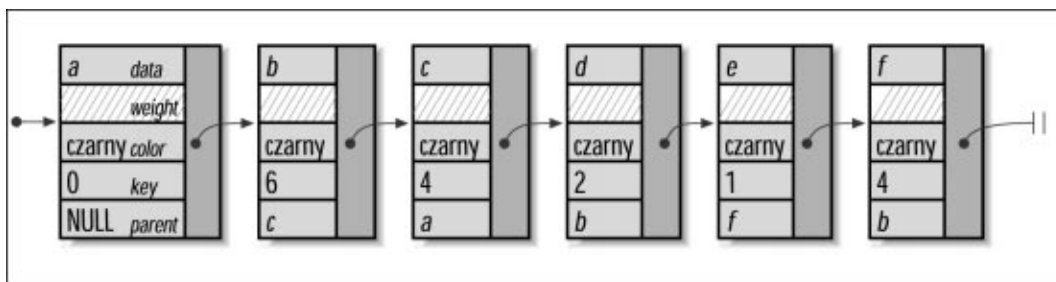
```

Budowa grafu ze struktur *MstVertex* wygląda niemalże tak samo, jak budowa grafu zawierającego inne typy danych. Do wstawienia wężła do grafu używamy *graph_ins_vertex* i przekazujemy jako dane *data* strukturę *MstVertex*. Analogicznie, aby wstawić krawędź, wywołujemy *graph_ins_edge*, i jako *data1* i *data2* przekazujemy struktury *data1* i *data2*. Podczas wstawiania wężła ustawiamy jedynie pole *data* struktury *MstVertex*. Przy wstawianiu krawędzi ustawiamy pole *data* struktury *data1* oraz pola *data* i *weight* struktury *data2*. Występujące w *data2* pole *weight* jest wagą krawędzi od wężła opisywanego przez *data1* do wężła opisywanego przez *data2*. Praktycznie wagi wylicza się i zapisuje zwykle jako liczby zmiennoprzecinkowe. Wartości kluczy wylicza się na podstawie wag, więc też są to liczby zmiennoprzecinkowe.

Operacja *mst* najpierw inicjalizuje wszystkie węzły z listy sąsiedztwa. Początkowe wartości kluczy wszystkich węzłów ustawiamy na *DBL_MAX*, jedynie węzeł początkowy otrzymuje wartość 0.0. Przypomnijmy, że w typie abstrakcyjnym grafów graf jest listą struktur sąsiedztwa, struktura odpowiadająca każdemu węzłowi zawiera ten właśnie węzeł oraz zbiór węzłów doń przylegających (więcej na ten temat w rozdziale 11.). Dla węzła zapisanego w strukturze listy sąsiedztwa zapisujemy jego kolor, wartość klucza i rodzica. Aby wszystkie te informacje zgromadzić w jednym miejscu, korzystamy z samego węzła, a nie z węzłów z jego listy sąsiedztwa. Ten sam węzeł może się pojawić na wielu listach sąsiedztwa, ale każdy węzeł wraz ze swoją listą występuje dokładnie raz.

Najważniejszym punktem algorytmu Prima jest pojedyncza pętla realizująca iterację raz dla każdego węzła grafu. Każdą iterację zaczynamy od wybrania węzła mającego najmniejszą wartość klucza spośród białych węzłów. Zaczerniamy ten węzeł, następnie przechodzimy po węzłach z nim sąsiadujących. Podczas ich przeglądania sprawdzamy kolor i wartość klucza, porównujemy je z kolorem i wartością klucza wybranego węzła. Jeśli węzeł sąsiedni jest biały, a jego klucz ma wartość mniejszą od klucza wybranego węzła, ustawiamy wartość klucza sąsiedniego węzła na wagę krawędzi między węzłem wybranym i sąsiednim, poza tym jako rodzica węzła sąsiedniego ustalamy węzeł bieżący. Aktualizujemy odpowiednie informacje dla węzła sąsiedniego. Cały proces powtarzamy, aż wszystkie węzły będą czarne.

Kiedy kończy się pętla główna algorytmu Prima, wyliczanie drzewa minimalnego jest skończone. W tej chwili wszystkie czarne struktury *MstVertex* z list sąsiedztwa wstawiamy na listę powiązaną *span*. Znajdujący się na tej liście węzeł mający rodzica o wartości NULL jest korzeniem drzewa. Element *parent* wszystkich innych węzłów wskazuje węzeł poprzedzający go w drzewie. Pole *weight* poszczególnych struktur *MstVertex* nie jest wypełniane, gdyż jest ono potrzebne jedynie do zapisywania wag na listach sąsiedztwa. Na rysunku 16.2 pokazano takie struktury *MstVertex*, jakie zostaną zwrócone po wyznaczeniu drzewa minimalnego grafu z rysunku 16.1.



Rysunek 16.2. Lista zwracana przez funkcję *mst* jako drzewo minimalne grafu z rysunku 16.1

Złożoność *mst* wynosi $O(EV^2)$, gdzie V jest liczbą węzłów grafu, zaś E jest liczbą krawędzi. Wynika to ze sposobu działania pętli głównej, w której wybieramy węzły i porównujemy wagi i wartości kluczy. Dla każdego spośród V węzłów najpierw przeglądamy V elementów z list sąsiedztwa w celu sprawdzenia, który biały węzeł ma najmniejszą wartość klucza. Ta część pętli ma zatem złożoność $O(V^2)$. Następnie dla każdego węzła sąsiadującego z wybranym węzłem sprawdzamy listę sąsiedztwa, aby sprawdzić, czy trzeba zmienić

wartość klucza i zmienić rodzica. Dla wszystkich V węzłów E razy sprawdzamy listę — raz dla każdej krawędzi. Każde z tych sprawdzeń wymaga czasu $O(V)$, jest to czas przeszukiwania listy. Wobec tego dla wszystkich wybieranych V węzłów operacja o złożoności $O(V)$ wykonywana jest E razy. W związku z tym ta część pętli ma złożoność $O(EV^2)$, a całkowita złożoność pętli głównej wynosi $O(V^2 + EV^2)$, czyli $O(EV^2)$. Pętle przed pętlą główną i za nią mają złożoność $O(V)$, więc złożoność *mst* wynosi $O(EV^2)$. Przypomnijmy jednak, że niewielkim nakładem pracy można poprawić szybkość działania algorytmu Prima do $O(E \lg V)$ (pokażemy to pod koniec rozdziału).

Najkrótsze ścieżki

Znalezienie *najkrótszej ścieżki* lub *ścieżki o najmniejszej wadze* między wskazanymi węzłami grafu jest istotą wielu problemów związanych ze znajdowaniem drogi. Formalnie, jeśli mamy skierowany graf z wagami $G = (V, E)$, najkrótsza ścieżka z węzła s do węzła t ze zbioru V jest zbiorem S krawędzi z E takim, że s łączy t po minimalnym koszcie.

Znajdując S rozwiązujemy *problem najkrótszej ścieżki między pojedynczą parą węzłów*. W tym celu tak naprawdę rozwiązujemy ogólniejszy *problem najkrótszej ścieżki z pojedynczego źródła*, problem pojedynczej pary rozwiązując niejako „przy okazji”. W przypadku problemu z pojedynczym źródłem, wyznaczamy najkrótsze ścieżki z węzła początkowego s do wszystkich pozostałych węzłów z niego osiągalnych. Robimy tak dlatego, że nie jest znany żaden algorytm, który szybciej rozwiązywałby problem z pojedynczą parą węzłów.

Algorytm Dijkstry

Jednym z rozwiązań problemu najkrótszej ścieżki z pojedynczego źródła jest *algorytm Dijkstry*. Algorytm ten pozwala budować *drzewo najkrótszych ścieżek*, którego korzeniem jest węzeł początkowy s , a gałęziami są najkrótsze ścieżki z s do wszystkich węzłów z G . Algorytm ten wymaga, aby wagi krawędzi były nieujemne. Podobnie jak w przypadku algorytmu Prima, algorytm Dijkstry należy do grupy algorytmów zachłannych, które akurat dają optymalne rozwiązanie. Algorytm ten jest algorytmem zachłannym, gdyż nowe krawędzie dodawane są do drzewa najkrótszej ścieżki według ich oceny w danej chwili.

Podstawą algorytmu Dijkstry jest wielokrotne wybieranie węzłów i badanie krawędzi przylegających do nich w celu określenia, czy najkrótsza ścieżka do poszczególnych węzłów może zostać poprawiona. Algorytm ten jest nieco podobny do przeszukiwania wszerz, gdyż najpierw badane są wszystkie krawędzie przylegające do węzła, dopiero potem przechodzimy do dalszej części grafu. Aby wyznaczyć najkrótszą ścieżkę między s a wszystkimi innymi węzłami, w algorytmie Dijkstry wymaga się, aby w każdym węźle zapisywane były kolor i oszacowanie najkrótszej ścieżki. Zwykle oszacowanie najkrótszej ścieżki zapisuje się w zmiennej d .

Początkowo wszystkim węzłom przypisujemy kolor biały, wszystkie oszacowania ścieżki ustawiamy na ∞ , co oznacza wielkość dowolnie dużą, większą od wagi dowolnej krawędzi grafu. Oszacowanie najkrótszej ścieżki dla węzła początkowego ustawiamy na 0.

W miarę działania algorytmu, wszystkim węzłom poza początkowym przypisujemy rodziców z drzewa najkrótszych ścieżek. Rodzic węzła może zmieniać się przed zakończeniem działania algorytmu wielokrotnie.

Dalej algorytm Dijkstry działa następująco: najpierw spośród wszystkich białych węzłów grafu wybieramy węzeł u z najmniejszym oszacowaniem najkrótszej ścieżki. Wstępnie będzie to węzeł początkowy, którego ścieżka została oszacowana na 0. Po wybraniu węzła zaczerniamy go. Następnie, dla każdego białego węzła v przylegającego do u zwalniamy krawędź (u, v) . Kiedy zwalniamy krawędź, sprawdzamy, czy przejście z u do v poprawi wyznaczoną dotąd najkrótszą ścieżkę do v . W tym celu dodajemy wagę (u, v) do oszacowania najkrótszej ścieżki do u . Jeśli wartość ta jest mniejsza lub równa oszacowaniu najkrótszej ścieżki do v , przypisujemy tę wartość v jako nowe oszacowanie najkrótszej ścieżki i ustawiamy v jako rodzica u . Proces ten powtarzamy dotąd, aż wszystkie węzły będą czarne. Kiedy wyliczone zostanie już drzewo najkrótszych ścieżek, najkrótszą ścieżkę z węzła s do danego węzła t można wybrać poprzez przejście po tym drzewie od węzła t przez kolejnych rodziców, aż do s . Ścieżka o odwrotnej kolejności do uzyskanej jest ścieżką szukaną.

Na rysunku 16.3 pokazano wyznaczanie najkrótszej ścieżki z a do wszystkich innych węzłów grafu. Na przykład, najkrótsza ścieżka z a do b to $\langle a, c, f, b \rangle$, jej waga wynosi 7. Oszacowania najkrótszych ścieżek i ich rodziców pokazano obok poszczególnych węzłów. Oszacowanie najkrótszej ścieżki znajduje się na lewo od ukośnika, rodzic węzła na prawo. Krawędzie zaznaczone na szaro są krawędziami zmieniającego się drzewa najkrótszych ścieżek.

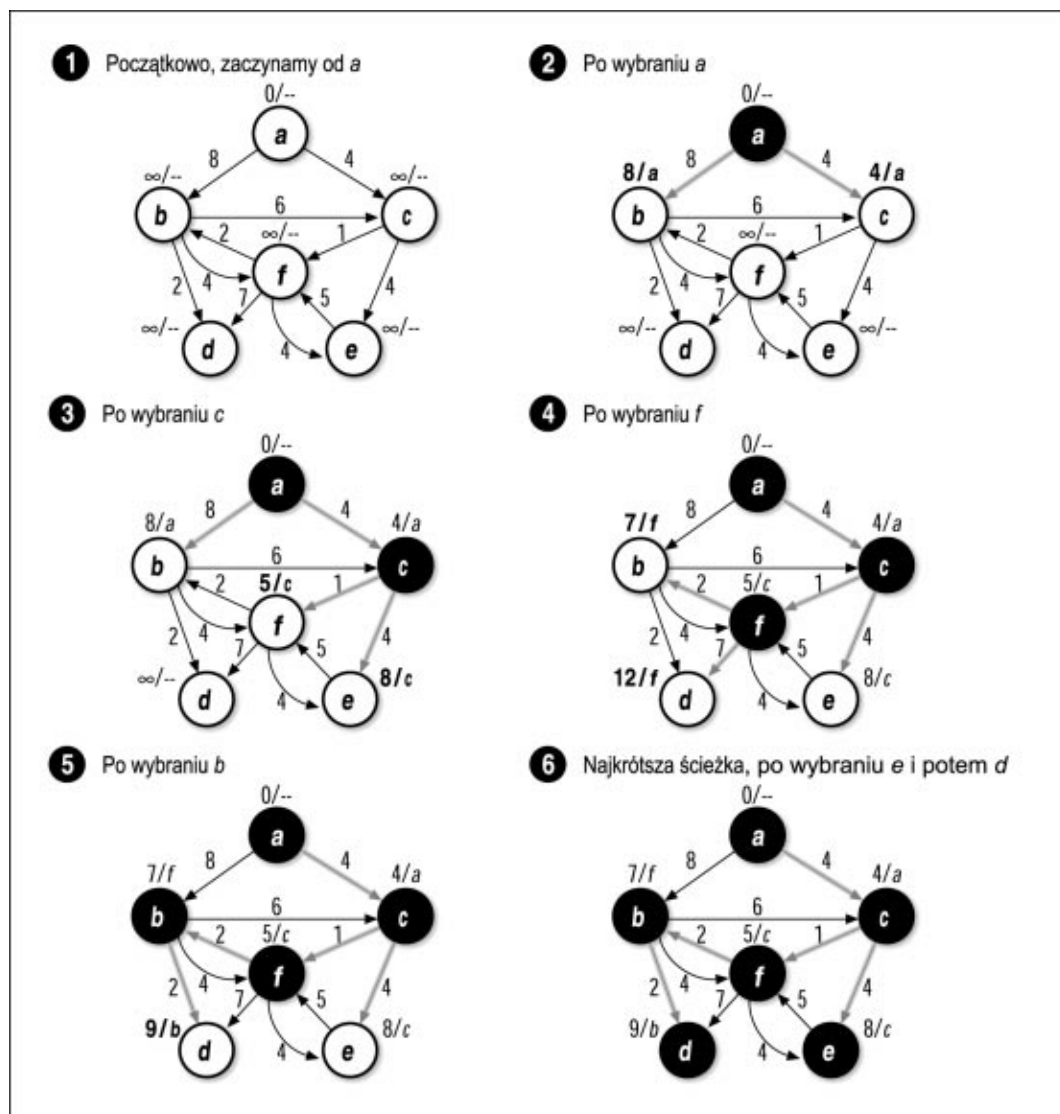
Interfejs najkrótszych ścieżek

shortest

```
int shortest(Graph *graph, const PathVertex *start, List *paths, int (*match)
(const void *key1, const void *key2));
```

Zwracana wartość: 0, jeśli wyznaczanie najkrótszej ścieżki się powiodło i -1 w przeciwnym razie.

Opis: Wylicza najkrótszą ścieżkę między *start* a wszystkimi pozostałymi węzłami grafu skierowanego z wagami *graph*. Operacja modyfikuje *graph*, więc w razie potrzeby należy zrobić jego kopię przed jej wywołaniem. Wszystkie węzły grafu muszą zawierać dane typu *PathVertex*. Wagę poszczególnym krawędziom przypisuje się przez nadanie wartości polu *weight* struktury *PathVertex* przekazanej do *graph_ins_edge* jako *data2*. Pola *data* poszczególnych struktur *PathVertex* używa się do zapisywania danych węzła, na przykład jego identyfikatora. Funkcja *match* grafu, przekazywana podczas inicjalizacji do *graph_init*, używana jest do porównywania pól *data*. Jest to ta sama funkcja, która powinna być przekazana do *shortest* jako *match*. Kiedy najkrótsza ścieżka zostanie wyliczona, zwracana jest w liście *paths*, będącej listą struktur *PathVertex*. W *paths* rodzic węzła początkowego ustawiany jest na NULL, zaś pole *parent* wszystkich pozostałych węzłów wskazuje



Rysunek 16.3. Wyznaczanie za pomocą algorytmu Dijkstry najkrótszych ścieżek

węzeł poprzedzający dany węzeł na ścieżce zaczynającej się od węzła początkowego. Węzły z *paths* wskazują węzły należące do *graph*, wobec czego pamięć zajmowana przez te węzły musi być dostępna tak długo, jak długo korzystamy z *paths*. Potem do usunięcia *paths* używa się operacji *list_destroy*.

Złożoność: $O(EV^2)$, gdzie V jest liczbą węzłów grafu, a E liczbą jego krawędzi. Można uzyskać nieznaczną poprawę szybkości, w postaci wyniku $O(E \lg V)$, podobnie jak w przypadku omawianego wcześniej algorytmu Prima.

Implementacja najkrótszej ścieżki i analiza kodu

Aby wyznaczyć najkrótsze ścieżki z danego wężła do wszystkich węzłów z niego dostępnych w skierowanym grafie z wagami, graf zapisujemy tak samo, jak zrobiliśmy to przy określaniu drzewa minimalnego, jednak zamiast struktury *MstVertex* używamy struktury *PathVertex* (listing 16.3). W strukturze tej możemy zapisać grafy z wagami oraz zapamiętywać informacje o węzłach i grafach wymagane w algorytmie Dijkstry. Struktura ta ma pięć elementów: *data* to dane węzła, *weight* to waga krawędzi przylegającej do węzła, *color* to kolor węzła, *d* jest oszacowaniem najkrótszej ścieżki do węzła, a *parent* to rodzic węzła w drzewie najkrótszych ścieżek. Graf zawierający struktury *PathVertex* tworzymy tak samo, jak wcześniej tworzyliśmy graf zawierający struktury *MstVertex*.

Listing 16.3. Implementacja wyliczania najkrótszych ścieżek

```

/*****
*
* ----- shortest.c -----
*
*****/

#include <float.h>
#include <stdlib.h>

#include "graph.h"
#include "graphalg.h"
#include "list.h"
#include "set.h"

/*****
*
* ----- relax -----
*
*****/

static void relax(PathVertex *u, PathVertex *v, double weight) {

/*****
*
* Zwolnienie krawędzi między węzłami u i v.
*
*****/

if (v->d > u->d + weight) {

    v->d = u->d + weight;
    v->parent = u;

}

return;

}

/*****
*
* ----- shortest -----
*
*****/

```

```

*****/

int shortest(Graph *graph, const PathVertex *start, List *paths, int (*match)
    (const void *key1, const void *key2)) {

    AdjList          *adjlist;

    PathVertex       *pth_vertex,
                    *adj_vertex;

    ListElmt         *element,
                    *member;

    double           minimum;

    int              found,
                    i;

    /*****
    *
    *   Inicjalizacja wszystkich węzłów grafu.
    *
    *****/

    found = 0;

    for (element = list_head(&graph_adjlists(graph)); element != NULL; element =
        list_next(element)) {

        pth_vertex = ((AdjList *)list_data(element))->vertex;

        if (match(pth_vertex, start)) {

            /*****
            *
            *   Inicjalizacja węzła początkowego.
            *
            *****/

            pth_vertex->color = white;
            pth_vertex->d = 0;
            pth_vertex->parent = NULL;
            found = 1;

        }

        else {

            /*****
            *
            *   Inicjalizacja wszystkich węzłów poza początkowym.
            *
            *****/

            pth_vertex->color = white;
            pth_vertex->d = DBL_MAX;
            pth_vertex->parent = NULL;

        }

    }

}

```

```

/*****
*
*   Jeśli nie znaleziono wężła początkowego, kończymy.
*
*****/

if (!found)
    return -1;

/*****
*
*   Za pomocą algorytmu Dijkstry wyznaczamy najkrótsze ścieżki z wężła
*   początkowego.
*
*****/

i = 0;

while (i < graph_vcount(graph)) {

    /*****
    *
    *   Wybieramy biały wężel z najmniejszym oszacowaniem najkrótszej ścieżki.
    *
    *****/

    minimum = DBL_MAX;

    for (element = list_head(&graph_adjlists(graph)); element != NULL; element
        = list_next(element)) {

        pth_vertex = ((AdjList *)list_data(element))->vertex;

        if (pth_vertex->color == white && pth_vertex->d < minimum) {

            minimum = pth_vertex->d;
            adjlist = list_data(element);

        }

    }

    /*****
    *
    *   Zaczerniamy wybrany wężel.
    *
    *****/

    ((PathVertex *)adjlist->vertex)->color = black;

    /*****
    *
    *   Przechodzimy wszystkie wężły sąsiadujące z wężłem wybranym.
    *
    *****/

    for (member = list_head(&adjlist->adjacent); member != NULL; member =
        list_next(member)) {

        adj_vertex = list_data(member);

        /*****

```

```

*                                                                    *
*   Znajdujemy węzły sąsiadujące na listach sąsiedztwa.                *
*                                                                    *
*****/

for (element = list_head(&graph_adjlists(graph)); element != NULL;
    element = list_next(element)) {

    pth_vertex = ((AdjList *)list_data(element))->vertex;

    if (match(pth_vertex, adj_vertex)) {

        /*****
        *
        *   Zwalniamy węzeł sąsiedni w strukturach list sąsiedztwa.
        *
        *****/

        relax(adjlist->vertex, pth_vertex, adj_vertex->weight);

    }

}

}

/*****
*
*   Przygotowujemy się do wybrania następnego węzła.
*
*****/

i++;

}

/*****
*
*   Ładujemy węzły oraz dane o ich ścieżkach na listę.
*
*****/

list_init(paths, NULL);

for (element = list_head(&graph_adjlists(graph)); element != NULL; element =
    list_next(element)) {

    /*****
    *
    *   Ładujemy wszystkie czarne węzły z listy struktur list sąsiedztwa.
    *
    *****/

    pth_vertex = ((AdjList *)list_data(element))->vertex;

    if (pth_vertex->color == black) {

        if (list_ins_next(paths, list_tail(paths), pth_vertex) != 0) {

            list_destroy(paths);
            return -1;
        }

    }

}

```

```

    }

    }

    return 0;

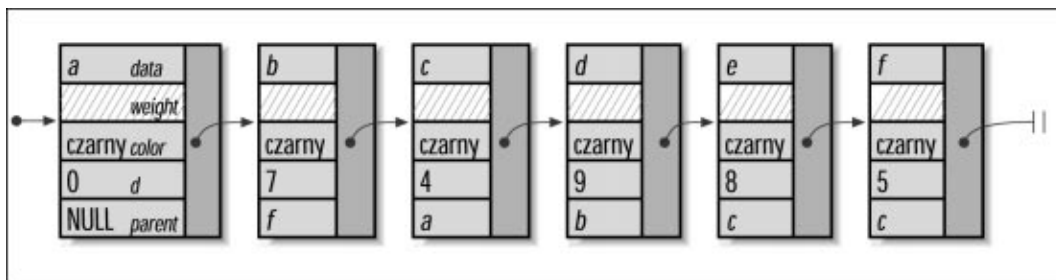
}

```

Operacja *shortest* najpierw inicjalizuje wszystkie węzły na listach sąsiedztwa: oszacowania najkrótszych ścieżek ustawiane są na *DBL_MAX*, z wyjątkiem węzła początkowego, dla którego oszacowaniem jest 0.0. Węzły zapisywane w poszczególnych strukturach list sąsiedztwa używane są do zapamiętywania koloru i oszacowania najkrótszej ścieżki z takich samych powodów, jak w przypadku drzew minimalnych.

Centralnym punktem algorytmu Dijkstry jest pojedyncza pętla wykonująca iterację raz dla każdego węzła grafu. Zaczynamy kolejne iteracje od wybrania węzła mającego najmniejsze oszacowanie najkrótszej ścieżki spośród wszystkich białych węzłów. Zaczerniamy ten węzeł. Następnie przechodzimy do węzłów sąsiadujących z wybranym węzłem. Kiedy przechodzimy do kolejnego węzła, sprawdzamy jego kolor i oszacowanie najkrótszej ścieżki; wywołujemy *relax* w celu zwolnienia krawędzi łączącej wybrany węzeł i węzeł sąsiedni. Jeśli *relax* musi zaktualizować oszacowanie najkrótszej ścieżki i rodzica węzła sąsiedniego, robi to w strukturze listy sąsiedztwa. Proces powtarzany jest dotąd, aż wszystkie węzły będą czarne.

Kiedy pętla główna algorytmu Dijkstry skończy swoje działanie, najkrótsze ścieżki z danego węzła do wszystkich węzłów z niego osiągalnych są już wyznaczone. Wstawiamy wszystkie struktury *PathVertex* czarnych węzłów z listy struktur sąsiedztwa do listy powiązanej *paths*. Znajdujący się na tej liście węzeł początkowy ma rodzica o wartości NULL. Element *parent* wszystkich innych węzłów wskazuje węzeł poprzedzający go w najkrótszej ścieżce. Pole *weight* poszczególnych struktur *PathVertex* nie jest wypełniane, gdyż jest ono potrzebne jedynie do zapisywania wag na listach sąsiedztwa. Na rysunku 16.4 pokazano takie struktury *PathVertex*, jakie zostaną zwrócone po wyznaczeniu najkrótszych ścieżek dla grafu z rysunku 16.3.



Rysunek 16.4. Lista zwracana przez funkcję *shortest* jako najkrótsze ścieżki wyznaczone dla grafu z rysunku 16.3

Złożoność *shortest* wynosi $O(EV^2)$, gdzie V jest liczbą węzłów grafu, zaś E jest liczbą krawędzi. Wynika to ze sposobu działania pętli głównej, w której wybieramy węzły i zwalniamy krawędzie. Dla każdego spośród V węzłów najpierw przeglądamy V elementów

z list sąsiedztwa w celu sprawdzenia, który biały węzeł ma najmniejsze oszacowanie najkrótszej ścieżki. Ta część pętli ma zatem złożoność $O(V^2)$. Następnie dla każdego węzła sąsiadującego z wybranym węzłem sprawdzamy listę sąsiedztwa w celu uzyskania informacji potrzebnych do zwolnienia krawędzi między węzłami. Dla wszystkich V węzłów E razy sprawdzamy listę — raz dla każdej krawędzi. Każde z tych sprawdzeń wymaga czasu $O(V)$, jest to czas przeszukiwania listy. Wobec tego dla wszystkich wybieranych V węzłów operacja o złożoności $O(V)$ wykonywana jest E razy. Dlatego ta część pętli ma złożoność $O(EV^2)$, a całkowita złożoność pętli głównej wynosi $O(V^2 + EV^2)$ czyli $O(EV^2)$. Pętle przed pętlą główną i za nią mają złożoność $O(V)$, więc złożoność *shortest* wynosi $O(EV^2)$. Wynik ten, podobnie jak w przypadku algorytmu Prima, można poprawić do $O(E \lg V)$.

Przykład użycia najkrótszych ścieżek: tablic tras

Jednym z zastosowań, w których najkrótsze ścieżki odgrywają istotną rolę, jest *wyznaczanie tras* danych między punktami Internetu. Wyznaczanie tras polega na podejmowaniu celowych decyzji o sposobie przesyłania danych między punktami. W Internecie przesyła się małe fragmenty danych nazywane *paketami* przez połączone ze sobą punkty nazywane *bramkami*. Kiedy poszczególne pakiety przechodzą przez bramkę, *ruter* sprawdza, gdzie dany pakiet ma dotrzeć i wybiera dla niego następną bramkę. Celem przyświecającym każdemu z ruterów jest przesyłanie pakietu coraz bliżej jego punktu docelowego.

Aby pakiety zbliżały się faktycznie do celu, routery zawierają dane o strukturze, czyli *topologii* sieci. Dane te mają postać *tablicy tras*. Tablica taka zawiera jeden zapis dla każdej bramki, której położenie zna ruter. Wszystkie zapisy wskazują następną bramkę, do której mają być kierowane pakiety przeznaczone dla danej bramki.

Aby pakiety były stale przesyłane możliwie najlepszą drogą, routery okresowo aktualizują swoje tablice tras w celu uwzględnienia w nich zmian w Internecie. W jednym z rodzajów wyznaczania tras, *wyznaczaniu tras pierwszą najkrótszą ścieżką* lub *wyznaczaniu tras SPF*, każdy ruter ma własną mapę sieci, dzięki czemu może aktualizować swoją tablicę tras wyliczając najkrótsze ścieżki między sobą a bramkami docelowymi. Mapa jest skierowanym grafem z wagami, którego węzły odpowiadają bramkom, a krawędzie połączeniom między bramkami. Waga każdej krawędzi jest wyznaczana na podstawie pomiarów szybkości ostatnio przesyłanych pakietów. Okresowo routery wymieniają między sobą dane o topologii i szybkości działania za pomocą specjalnie w tym celu opracowanego protokołu.

Na listingu 16.4. pokazano funkcję *route* wyliczającą dane potrzebne do aktualizacji jednej pozycji tablicy tras przy wyznaczaniu tras SPF. Funkcja pobiera listę informacji o ścieżkach zwracaną przez parametr *paths* funkcji *shortest*. Na podstawie tych informacji decyduje się, do której bramki należy następnie wysłać pakiet, aby jak najszybciej dotarł do miejsca przeznaczenia.

Listing 16.4. Implementacja funkcji aktualizującej zapisy w tablicy tras

```

/*****
*
* ----- route.c -----
*
*****/

```

```

*
*****/

#include <stdlib.h>

#include "graphalg.h"
#include "list.h"
#include "route.h"

/*****
* ----- route -----
*
*****/

int route(List *paths, PathVertex *destination, PathVertex **next, int
        (*match)(const void *key1, const void *key2)) {

    PathVertex    *temp,
                  *parent;

    ListElmt      *element;

    int            found;

/*****
*
*   Znalezienie miejsca docelowego na liście bramek.
*
*****/

    found = 0;

    for (element = list_head(paths); element != NULL; element =
        list_next(element)) {

        if (match(list_data(element), destination)) {

            temp = list_data(element);
            parent = ((PathVertex *)list_data(element))->parent;
            found = 1;
            break;

        }

    }

/*****
*
*   Jeśli miejsce docelowe jest nieosiągalne, kończymy.
*
*****/

    if (!found)
        return -1;

/*****
*
*   Wyliczenie następnej bramki na najkrótszej ścieżce do celu.
*
*****/

    while (parent != NULL) {

```

```

temp = list_data(element);
found = 0;

for (element = list_head(paths); element != NULL; element =
    list_next(element)) {

    if (match(list_data(element), parent)) {

        parent = ((PathVertex *)list_data(element))->parent;
        found = 1;
        break;

    }

}

/*****
*
*   Jeśli miejsce docelowe jest niedostępne, kończymy.
*
*****/

if (!found)
    return -1;

}

*next = temp;

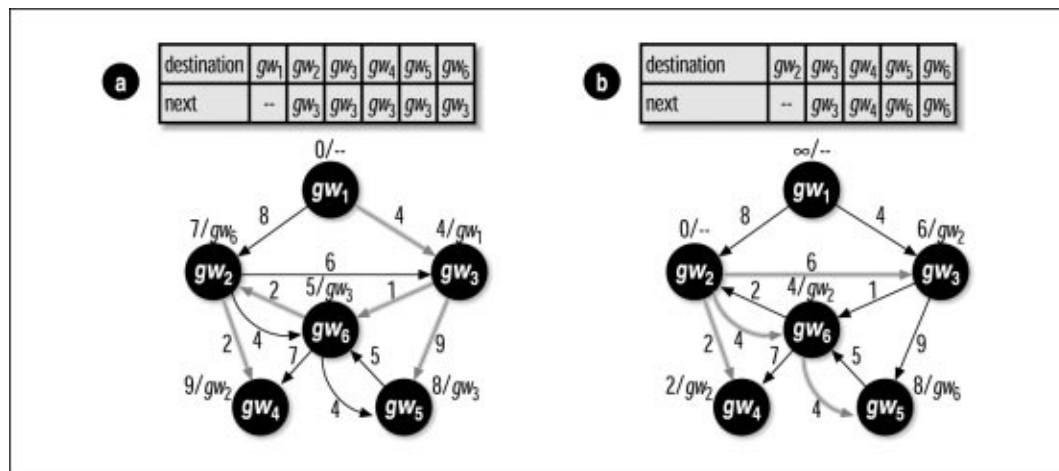
return 0;

}

```

Aby wypełnić całą tablicę w zakresie dotyczącym pewnej bramki, najpierw wywołujemy *shortest* z bramką wskazaną jako *start*. Następnie, dla każdego miejsca przeznaczenia, które ma być uwzględnione w tablicy tras, wywołujemy *route* z miejscem docelowym jako parametrem *destination*. Przekazujemy tę samą funkcję *match*, której użyto w *graph_init* przed wygenerowaniem *paths*. Funkcja *route* śledzi wskaźniki rodziców zapisane w *paths* od punktu docelowego wstecz, do bramki, i wybiera najlepszy sposób przesłania pakietu zwracając go w *next*. Węzeł wskazany w *next* wskazuje węzeł z *paths*, więc pamięć związana z *paths* musi być dostępna tak długo, jak długo korzystamy z *next*.

Na rysunku 16.5a pokazano wyliczanie tablicy tras dla rutera z bramki gw_1 w sieci z grafu pokazanego niżej. Na rysunku 16.5b pokazano wyliczanie tablicy tras dla rutera z bramki gw_2 . Warto zwrócić uwagę, że uzyskiwane najkrótsze ścieżki są różne w zależności od punktu początkowego. Poza tym na rysunku 16.5b nie można sięgnąć do punktu gw_1 , więc w tablicy tras nie ma odpowiedniego zapisu.



Rysunek 16.5. Tablice tras wyznaczone dla internetowych bramek (a) gw₁ i (b) gw₂

Złożoność route to $O(n^2)$, gdzie n jest liczbą bramek w *paths*. Wynika to stąd, że przeszukujemy w *paths* rodziców poszczególnych węzłów na drodze między lokalizacją docelową a punktem początkowym. W najgorszym wypadku, jeśli najkrótsza ścieżka zawiera w *paths* wszystkie bramki, w celu odnalezienia wszystkich rodziców konieczne może być przeszukanie całej listy bramek n razy.

Problem komiwojażera

Wyobraźmy sobie komiwojażera, który odwiedza w ramach swojej pracy szereg miast. Jego celem jest odwiedzenie każdego miasta dokładnie jednokrotnie i powrót do punktu wyjścia oraz pokonanie przy tym jak najkrótszej drogi. Tak właśnie wygląda podstawowe sformułowanie *problemu komiwojażera*.

W grafach cykl polegający na odwiedzeniu każdego węzła dokładnie raz i powrocie następnie do punktu wyjścia nazywany jest *cyklem hamiltonowskim*. Aby rozwiązać problem komiwojażera, jako modelu używa się grafu $G = (V, E)$ i szuka się w nim jak najkrótszego cyklu hamiltonowskiego. G jest zupełnym, nieskierowanym grafem z wagami, V jest zbiorem węzłów reprezentujących punkty, które mają zostać odwiedzone, E to zbiór krawędzi reprezentujących połączenia węzłów. Wagą każdej krawędzi z E jest odległość między węzłami wyznaczającymi tę krawędź. Graf G jest nieskierowany i zupełny, więc E zawiera $V(V - 1) / 2$ krawędzi.

Jednym ze sposobów rozwiązania problemu komiwojażera jest zbadanie wszystkich możliwych permutacji węzłów z G . Każda permutacja odpowiada jednej drodze, więc wystarczy po prostu wybrać permutację dającą najkrótszą drogę. Niestety, podejście takie jest praktycznie nieprzydatne, gdyż jego złożoność nie jest ograniczona wielomianem. Złożoność jest ograniczona wielomianem, jeśli nie przekracza $O(n^k)$, gdzie k jest pewną stałą. Dla zbioru zawierającego V węzłów istnieje $V!$ możliwych permutacji, więc ich zbadanie będzie wymagało czasu $O(V!)$, gdzie $V!$ to silnia liczby V , czyli iloczyn wszystkich liczb od 1 do V włącznie.

W zasadzie unika się stosowania algorytmów o złożoności nieograniczonej wielomianami, gdyż nawet dla małych ilości danych wejściowych problemy szybko stają się zbyt złożone, aby je obliczyć. Problem komiwojażera jest szczególnego rodzaju problemem nieograniczonym wielomianem, jest problemem *NP-zupełnym*. Problemy *NP-zupełne* są to takie problemy, dla których nie są znane algorytmy ograniczone wielomianem, ale dla których nie istnieją też dowody na istnienie takich algorytmów. Niemniej, prawdopodobieństwo znalezienia odpowiedniego algorytmu jest niesłychanie małe. Mając to wszystko na uwadze, problem komiwojażera rozwiązujemy korzystając z algorytmu przybliżającego (więcej na ten temat w rozdziale 1.).

Zastosowanie heurystyki najbliższego sąsiada

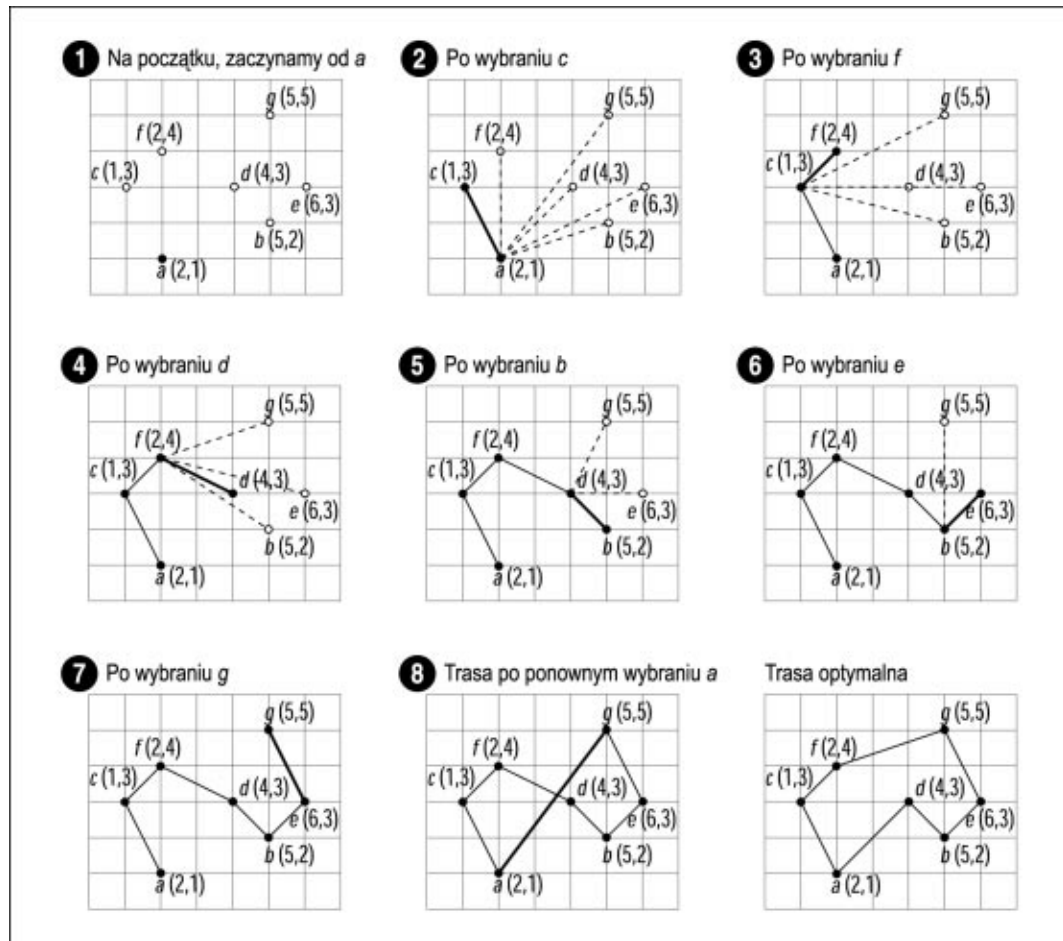
Jednym ze sposobów wyliczenia przybliżonej drogi komiwojażera jest zastosowanie *heurystyki najbliższego sąsiada*. Zaczynamy od drogi zawierającej tylko jeden węzeł, początkowy, który zaczerniamy. Wszystkie pozostałe węzły są białe dotąd, aż zostaną dodane do trasy — wtedy też je zaczernimy. Następnie dla każdego węzła v , który jeszcze nie należy do trasy, wyliczamy wagi krawędzi między ostatnio dodanym do trasy węzłem u a węzłem v . Przypomnijmy, że w opisywanym problemie wagą krawędzi jest odległość między węzłami u i v . Odległości między węzłami wyliczamy przez wyznaczenie odległości między odpowiadającymi im punktami. Odległość r między punktami (x_1, y_1) i (x_2, y_2) wyznacza się za pomocą wzoru:

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Korzystając z powyższego wzoru wybieramy węzeł najbliższy u , zaznaczamy go na czarno i dodajemy do trasy. Proces ten powtarzamy dotąd, aż wszystkie węzły będą czarne. Kiedy wszystkie węzły są już czarne, do trasy dodajemy ponownie punkt początkowy, aby zamknąć cykl.

Na rysunku 16.6 pokazano rozwiązanie problemu komiwojażera za pomocą heurystyki najbliższego sąsiada. Zwykle, kiedy rysujemy graf obrazujący problem komiwojażera, nie pokazujemy krawędzi łączących poszczególne węzły, za to samym węzłom dodaje się ich współrzędne. Linie przerywane na każdym etapie pokazują krawędzie, których długości są porównywane. Najgrubsza linia jest dodawana do trasy. Uzyskana trasa w naszym wypadku ma długość 15,95 jednostki, zaś trasa optymalna 14,71, czyli jest o około 8% krótsza.

Heurystyka najbliższego sąsiada ma pewne interesujące właściwości. Tak jak w przypadku innych algorytmów z tego rozdziału, jest on podobny do przeszukiwania wszerz, gdyż badane są wszystkie węzły sąsiadujące z węzłem ostatnio dodanym do trasy. Opisywane rozwiązanie jest algorytmem zachłannym, za każdym razem do trasy dodawany jest węzeł, który w danej chwili najbardziej nam odpowiada. Niestety, wybór najbliższego sąsiada może negatywnie wpłynąć na dobór dalszej części trasy. Niemniej, opisywany algorytm zawsze zwraca trasę, której długość nie przekracza dwukrotnej długości trasy optymalnej, poza tym zwykle uzyskiwany wynik jest lepszy. Istnieją też inne techniki pozwalające poprawić uzyskiwany wynik. Jedną z nich jest stosowanie *heurystyki wymiany* (zobacz tematy pokrewne na końcu rozdziału).



Rysunek 16.6. Rozwiązywanie problemu komiwojażera za pomocą heurystyki najbliższego sąsiada

Interfejs problemu komiwojażera

tsp

```
int tsp(List *vertices, const TspVertex *start, List *tour, int (*match)
(const void *key1, const void *key2))
```

Zwracana wartość: 0, jeśli wyliczanie trasy komiwojażera się powiodło i -1 w przeciwnym razie.

Opis: Funkcja wylicza przybliżoną trasę komiwojażera przez punkty opisane jako węzły w parametrze *vertices*. Trasa zaczyna się od węzła wskazanego jako *start*. Operacja modyfikuje węzły *vertices*, więc w razie potrzeby, przed wywołaniem operacji, trzeba zrobić kopię. Każdy element z *vertices* musi być typu *TspVertex*. W polu *data* struktury *TspVertex* zapisuje się dane związane z węzłem, na przykład identyfikator. Elementy

x i y struktury określają współrzędne wężła. Funkcja przekazana jako *match* decyduje, czy dwa węzły pasują do siebie. Należy porównywać jedynie pola *data* struktur *TspVertex*. Trasa jest zwracana jako *tour* — listę struktur *TspVertex*. Wszystkie węzły w *tour* występują w takiej kolejności, w jakiej zostały odnalezione. Elementy z *tour* wskazują węzły z *vertices*, więc trzeba zapewnić, że pamięć odpowiadająca *vertices* będzie dostępna tak długo, jak długo będziemy korzystać z trasy *tour*; potem listę tę usuwa się za pomocą operacji *list_destroy*.

Złożoność: $O(V^2)$, gdzie V jest liczbą węzłów odwiedzanych w trasie.

Implementacja problemu komiwojażera i analiza kodu

Aby rozwiązać problem komiwojażera, zaczynamy od zapisania grafu mającego postać listy węzłów. Krawędzie łączące poszczególne węzły nie są zapamiętywane jawnie. Wszystkie węzły z listy mają postać struktur *TspVertex* (listing 16.5). Struktura ta ma cztery pola: *data* na dane węzła, x i y to współrzędne punktu odpowiadającego węzłowi, zaś *color* to kolor węzła.

Listing 16.5. Implementacja problemu komiwojażera

```

/*****
*
* ----- tsp.c -----
*
*****/

#include <float.h>
#include <math.h>
#include <stdlib.h>

#include "graph.h"
#include "graphalg.h"
#include "list.h"

/*****
*
* ----- tsp -----
*
*****/

int tsp(List *vertices, const TspVertex *start, List *tour, int (*match)
      (const void *key1, const void *key2)) {

TspVertex      *tsp_vertex,
                *tsp_start,
                *selection;

ListElmt       *element;

double         minimum,
                distance,
                x,
                y;

int            found,
```

```

        i;

/*****
*
*   Inicjalizacja listy na trasę.
*
*****/

list_init(tour, NULL);

/*****
*
*   Inicjalizacja wszystkich węzłów grafu.
*
*****/

found = 0;

for (element = list_head(vertices); element != NULL; element =
    list_next(element)) {

    tsp_vertex = list_data(element);

    if (match(tsp_vertex, start)) {

/*****
*
*   Początek trasy w węźle początkowym.
*
*****/

        if (list_ins_next(tour, list_tail(tour), tsp_vertex) != 0) {

            list_destroy(tour);
            return -1;

        }

/*****
*
*   Zapis węzła początkowego i jego współrzędnych.
*
*****/

        tsp_start = tsp_vertex;
        x = tsp_vertex->x;
        y = tsp_vertex->y;

/*****
*
*   Zaczerniamy węzeł początkowy.
*
*****/

        tsp_vertex->color = black;
        found = 1;

    }

    else {

/*****

```

```

*                                                                    *
*   Kolorujemy wszystkie inne węzły na białą.                        *
*                                                                    *
*****/

    tsp_vertex->color = white;

}

}

/*****
*                                                                    *
*   Jeśli nie znaleziono węzła początkowego, kończymy.              *
*                                                                    *
*****/

if (!found) {

    list_destroy(tour);
    return -1;

}

/*****
*                                                                    *
*   Za pomocą heurystyki najbliższego sąsiada wyliczamy trasę.      *
*                                                                    *
*****/

i = 0;

while (i < list_size(vertices) - 1) {

    /*****
    *                                                                    *
    *   Wybieramy biały węzeł najbliższy węzłowi dodanemu poprzednio. *
    *                                                                    *
    *****/

    minimum = DBL_MAX;

    for (element = list_head(vertices); element != NULL; element =
        list_next(element)) {

        tsp_vertex = list_data(element);

        if (tsp_vertex->color == white) {

            distance = sqrt(pow(tsp_vertex->x-x,2.0) + pow(tsp_vertex->y-y,2.0));

            if (distance < minimum) {

                minimum = distance;
                selection = tsp_vertex;

            }

        }

    }

}

```

```

/*****
*
*   Zapamiętujemy współrzędne wybranego węzła.
*
*****/

x = selection->x;
y = selection->y;

/*****
*
*   Zaczerniamy wybrany węzeł.
*
*****/

selection->color = black;

/*****
*
*   Do trasy wstawiamy wybrany węzeł.
*
*****/

if (list_ins_next(tour, list_tail(tour), selection) != 0) {
    list_destroy(tour);
    return -1;
}

/*****
*
*   Przygotowujemy się do wybrania następnego węzła.
*
*****/

i++;
}

/*****
*
*   Na koniec do trasy dołączamy ponownie węzeł początkowy.
*
*****/

if (list_ins_next(tour, list_tail(tour), tsp_start) != 0) {
    list_destroy(tour);
    return -1;
}

return 0;
}

```

Operacja *tsp* najpierw koloruje na biało wszystkie węzły poza węzłem początkowym, który od razu jest czarny i od razu dodawany jest do trasy. Współrzędne węzła początkowego są zapamiętywane, aby można było wyliczać odległości między tym i następnym węzłem dołączanym do trasy w pierwszej iteracji w głównej pętli. W pętli tej dodajemy do trasy pozostałe

węzły. W każdej iteracji wyszukujemy węzeł najbliższy w stosunku do węzła dodanego ostatnio, zapisujemy jego współrzędne na potrzeby następnej iteracji i zaczerniamy węzeł. Kiedy kończy się działanie pętli, dodajemy do trasy ponownie węzeł początkowy.

Złożoność tsp wynosi $O(V^2)$, gdzie V jest liczbą węzłów w trasie. Wynika to stąd, że dla każdej z $V - 1$ iteracji pętli głównej przeszukujemy wszystkie węzły i wyliczamy odległości do nich, aby znaleźć najbliższy. Warto zauważyć, że $O(V^2)$ to ogromny zysk w stosunku do wyliczania trasy optymalnej w czasie rzędu $O(V!)$.

Pytania i odpowiedzi

P: W implementacjach wyznaczania drzew minimalnych i najkrótszych ścieżek, grafy z wagami zapisywaliśmy podając wagi krawędzi w samym grafie. Jak można byłoby to zrobić inaczej?

O: W przypadku grafów, których krawędzie mają niezbyt często zmieniające się wagi, jest to dobre rozwiązanie. Jednak ogólniej o wagach krawędzi można myśleć jako o funkcji $W(u, v)$, gdzie u i v są węzłami opisującymi krawędź, dla której wyznaczamy wagę. Aby określić wagę tej krawędzi, po prostu wywołujemy funkcję. Zaletą takiego rozwiązania jest możliwość dynamicznego wyliczania wag, które mogą się często zmieniać. Z drugiej strony, wadą jest to, że jeśli funkcja określająca wagi jest skomplikowana, jej wielokrotne wyliczanie spowolni działanie programu.

P: Kiedy rozwiązywaliśmy problem komiwojażera, stwierdziliśmy, że znajdowanie trasy optymalnej zawierającej więcej niż kilka punktów jest niewykonalne, dlatego użyliśmy algorytmu przybliżającego. Jak inaczej można byłoby przybliżyć trasę komiwojażera? Jaka byłaby złożoność tego rozwiązania? Jak bliskie w stosunku do trasy optymalnej byłyby znajdowane trasy?

O: Innym rozwiązaniem problemu komiwojażera byłoby użycie algorytmu przybliżającego wyznaczającego drzewo minimalne, następnie przejście tego drzewa „po korzeniu” (zobacz: rozdział 9.). Czas wykonania takiego algorytmu wyniosłby $O(EV^2)$, jeśli skorzystalibyśmy z operacji mst . Tak jak w przypadku dobierania najbliższego sąsiada, trasa nie przekraczałaby więcej niż dwukrotnie długości trasy optymalnej. Aby to sprawdzić, przyjmijmy, że T_{MST} jest długością drzewa minimalnego, T_{APP} to długość wyznaczonej w sposób przybliżony trasy, a T_{OPT} to długość trasy optymalnej. Minimalne drzewo i trasa optymalna zawierają wszystkie węzły drzewa i żadne drzewo z wszystkimi węzłami nie jest krótsze od drzewa minimalnego, $T_{MST} \leq T_{OPT}$. Poza tym $T_{APP} \leq 2T_{MST}$, gdyż jedynie w najgorszym razie trasa przybliżona musi przejść po każdej krawędzi drzewa minimalnego dwukrotnie. Wobec tego $T_{APP} \leq 2T_{OPT}$. Ostatecznie zatem:

$$T_{MST} \leq T_{OPT}, T_{APP} \leq 2T_{MST} \Rightarrow T_{APP} \leq 2T_{OPT}$$

P: Jeśli wyliczając drzewo minimalne za pomocą algorytmu Prima zaczniemy od innego węzła, to czy możemy dla tego samego grafu uzyskać inne drzewo?

O: Kiedy algorytm Prima wykonywany jest na dużych grafach, niejednokrotnie zdarza się, że podczas wybierania węzła o najmniejszej wartości klucza okazuje się, że kilka węzłów ma takie same wartości. W takim wypadku wybieramy dowolny z nich. W zależności od

tego, który węzeł wybierzemy, analizujemy inny zbiór krawędzi wychodzących z węzła. Wobec tego w drzewie minimalnym możemy otrzymać inne krawędzie. Jednak, choć do uzyskanego drzewa należeć mogą różne krawędzie, całkowita waga drzewa się nie zmienia i jest stała dla danego grafu.

P: Przypomnijmy, że rozwiązując problem komiwojażera korzystaliśmy z grafu, który badaliśmy szukając w nim minimalnego cyklu hamiltonowskiego. Czy wszystkie grafy zawierają cykle hamiltonowskie?

O: Nie — najprostszym przykładem może być dowolny graf niespójny lub skierowany graf acykliczny. Problem ten jednak nigdy nie występuje w grafach zupełnych, które zawierają wiele takich cykli. Sprawdzenie czy graf zawiera cykl hamiltonowski to całkiem inny problem, który — podobnie jak problem komiwojażera — jest NP-zupełny. Wiele grafów należy do tej klasy.

P: Przedstawiona w tym rozdziale implementacja algorytmu Prima wykonuje się w czasie $O(EV^2)$, ale istnieje lepsza implementacja wykonująca się w czasie $O(E \lg V)$. Jak można poprawić przedstawioną implementację?

O: Przedstawiona w tym rozdziale implementacja algorytmu Prima wykonuje się w czasie $O(EV^2)$, gdyż dla każdego węzła grafu przeszukujemy listę węzłów w celu sprawdzenia, który z nich jest biały i ma najmniejszą wartość klucza. Tę część algorytmu możemy znacząco poprawić korzystając z kolejki priorytetowej (rozdział 10.). Przypomnijmy, że wybieranie najmniejszej wartości z kolejki priorytetowej jest operacją, której złożoność wynosi $O(1)$, zaś utrzymywanie kolejki priorytetowej jest operacją o złożoności $O(\lg n)$, gdzie n jest liczbą elementów. W wyniku tego całkowita złożoność tak zmodyfikowanego algorytmu Prima wyniesie $O(E \lg V)$. Jednak kolejka priorytetowa musi umożliwiać zwiększanie wartości już się w niej znajdujących i szybkie odszukiwanie konkretnej wartości w celu jej zmodyfikowania. Jako że kolejki opisane w rozdziale 10. takich możliwości nie mają, w tym rozdziale implementacja algorytmu Prima została zrealizowana bez tych poprawek.

P: Zwykle gdy wyliczamy drzewo minimalne, robimy to dla grafu spójnego. Co się stanie, jeśli będziemy chcieli zrobić to dla grafu niespójnego?

O: Przypomnijmy, że graf jest spójny, jeśli wszystkie jego węzły są dostępne ze wszystkich innych węzłów. Jeśli będziemy starali się wyznaczyć drzewo minimalne grafu niespójnego, uzyskamy po prostu drzewo minimalne składowej spójnej zawierającej węzeł początkowy.

Tematy pokrewne

Algorytm Bellmana-Forda

Inne rozwiązanie problemu znajdowania najkrótszych ścieżek z pojedynczego źródła. W przeciwieństwie do algorytmu Dijkstry, algorytm Bellmana-Forda obsługuje także grafy z ujemnymi wagami. Jego złożoność wynosi $O(VE)$, gdzie V jest liczbą węzłów grafu, a E liczbą jego krawędzi.

Algorytm Kruskala

Inne rozwiązanie problemu wyznaczania drzew minimalnych. Algorytm działa tak, że najpierw umieszczamy każdy węzeł w osobnym zbiorze. Następnie wybieramy krawędzie w kolejności rosnących wag. Wybierając każdą krawędź, określamy, czy jej węzły należą do różnych zbiorów. Jeśli tak, wstawiamy krawędź do drzewa minimalnego i sumujemy zbiory zawierające oba węzły. W przeciwnym razie po prostu przechodzimy do następnej krawędzi. Proces kontynuujemy dotąd, aż zbadane zostaną wszystkie krawędzie. Złożoność algorytmu Kruskala wynosi $O(E \lg E)$, jeśli do zapisu krawędzi używamy kolejki priorytetowej; E jest liczbą krawędzi grafu.

Problem najkrótszych ścieżek wszystkich par

Inny rodzaj problemu najkrótszej ścieżki, w którym znajdujemy najkrótsze ścieżki między wszystkimi parami węzłów grafu. Jednym ze sposobów rozwiązania tego problemu jest rozwiązanie problemu najkrótszych ścieżek z pojedynczego źródła dla wszystkich węzłów grafu. Jednak można uzyskać szybsze rozwiązanie bezpośrednie.

Heurystyka wymian

Heurystyka pozwalająca poprawić trasy przybliżone trasy komiwojażera, na przykład trasy uzyskane metodą najbliższego sąsiada. Polega ona na wielokrotnych próbach wymiany krawędzi należących już do trasy na inne, które dają lepszy wynik. Przy każdej zamianie krawędzi długość trasy jest ponownie przeliczana w celu sprawdzenia czy uzyskano poprawę.