Grafy i algorytmy grafowe

Biblioteka boost::graph library

Grafy są strukturą danych, która pozwala opisać złożone zależności z otaczającego nas świata. W artykule omawiamy bibliotekę do reprezentacji grafów, wchodzącą w skład kolekcji boost.

Dowiesz się:

- Jak reprezentować grafy w C++;
- Co oferuje biblioteka boost::graph;
- Jak używać algorytmów grafowych oferowanych przez tę bibliotekę.

Powinieneś wiedzieć:

- Jak pisać proste programy w C++;
- Co to jest graf.

Biblioteka boost::graph

Grafy, czyli zbiory wierzchołków powiązane krawędziami (patrz ramka), są bardzo często wykorzystywane do przetwarzania informacji. Są one bardzo ogólną strukturą danych, inne powszechnie używane struktury danych (drzewa, listy) są pewnymi szczególnymi przypadkami grafów.

Biblioteka BGL (ang. *Boost Graph Library*), wchodząca w skład bibliotek boost, zawiera szablony reprezentujące grafy oraz zbiór kilkudziesięciu algorytmów grafowych. Grafy są dostarczane w postaci generycznej, dzięki czemu z wierzchołkiem, krawędzią lub całym grafem można związać obiekt lub obiekty dowolnego typu. Na rysunku 1 pokazano schematycznie graf przechowywany w takiej postaci, gdzie z wierzchołkiem są związane krawędzie z niego wychodzące.

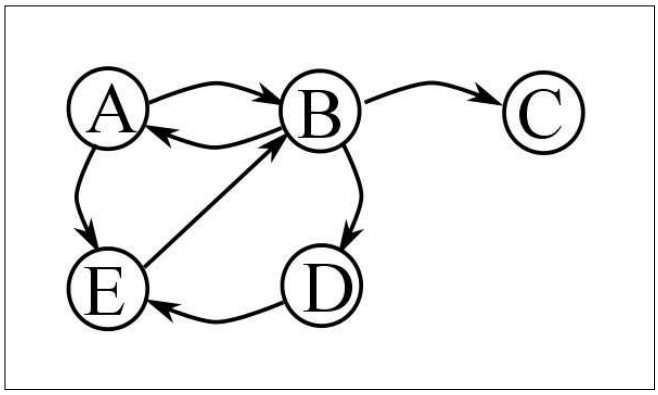
Szablony z biblioteki boost::graph wykorzystują kolekcje ze standardowej biblioteki szablonów (patrz ramka). Graf reprezentowany listą sąsiedztwa generuje się używając szablonu `adjacency_list`, tak jak pokazano na Listingu 1. Trzy pierwsze parametry są nazwami typów, które definiują kolekcje wykorzystane do przechowywania krawędzi i wierzchołków oraz rodzaju grafu, natomiast trzy kolejne są typami związanymi z wierzchołkiem, krawędzią i grafem. Znaczenie tych parametrów, oraz typowe wartości podano na Listingu 1. Biblioteka wykorzystuje typ `struct no_property {}`, który oznacza, że nie przechowujemy obiektów związanych z wierzchołkiem, krawędzią lub grafem.

Graf może być reprezentowany także przez macierz sąsiedztwa (szablon `adjacency_matrix`) albo w postaci skompresowanej (szablon `compressed_sparse_row_graph`), te sposoby nie są wykorzystywane w tym artykule.

W najprostszym przypadku, gdy krawędzie są przechowywane w wektorze, identyfikatorem wierzchołka jest

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Przykłady wykorzystują biblioteki boost (*www.boost.org*). Aby poprawnie je skompilować należy dodać odpowiednie zależności wykorzystywane podczas konsolidacji; dla konsolidatora g++ należy dodać opcje: `-lboost_graph`; dla konsolidatora Visual Studio (program link) biblioteki boost są dodawane automatycznie. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła umieszczono jako materiały pomocnicze.



Rysunek 1. *Przykładowy graf*

liczba całkowita (indeks). W takim przypadku graf można utworzyć dostarczając liczbę wierzchołków oraz tablicę krawędzi, gdzie krawędź to para liczb: indeks wierzchołka z którego krawędź wychodzi oraz indeks wierzchołka do którego krawędź wchodzi. Przykład konstrukcji grafu zawiera Listing 2. Reprezentacja wewnętrzna jest schematycznie przedstawiona na Rysunku 2.

Dodawanie, usuwanie wierzchołków oraz krawędzi jest możliwe dla grafów utworzonych za pomocą szablonu `adjacency_list`. Wymienione modyfikacje grafu wykonuje się za pomocą funkcji `add_vertex` , `add_edge`, `remove_vertex` oraz `remove_edge`. Szczególną uwagę należy zwrócić na usuwanie wierzchołków (`remove_vertex`), powinno się najpierw usunąć wszystkie krawędzie wchodzące i wychodzące z usuwanego wierzchołka. Biblioteka zawiera funkcję `clear_vertex`, która upraszcza to zadanie.

Do grafu, pokazanego na Rysunku 1 dodawany jest nowy wierzchołek. Następnie tworzona jest krawędź pomiędzy wierzchołkiem A, a tym dodanym; później usuwane są wszystkie krawędzie związane z wierzchołkiem D (wchodzące i wychodzące). Na koniec usuwany jest

Grafy

Grafy to zbiory wierzchołków powiązane krawędziami. Każda krawędź zaczyna się i kończy w jednym z wierzchołków. Grafy dzielimy na skierowane (krawędzie mają określony kierunek) oraz nieskierowane. Przykład grafu pokazano na Rysunku 1. Specjalnym rodzajem grafów są drzewa, którymi nazywamy grafy nie zawierające dróg zamkniętych (czyli cykli).

Grafy są bardzo często wykorzystywane do opisywania różnych zależności z otaczającego nas świata; przykładem jest opis sieci połączeń pomiędzy miastami, wtedy wierzchołki mogą opisywać miasta, zaś krawędzie – drogi.

Standardowa biblioteka szablonów

Zbiór kontenerów jednowymiarowych (tablice, listy, słowniki), algorytmów i iteratorów, wchodzi w skład biblioteki standardowej języka C++. Jest bardzo wydajna i uniwersalna (wykorzystuje szablony). Więcej informacji *http://www.sgi.com/tech/stl/*.

Listing 1. *Szablon tworzący graf*

```
adjacency_list<VertexList, //kolekcja przechowująca wierzchołki, znaczenie jest następujące:
                           // vecS - std::vector
                           // listS - std::list
                           // setS - std::set
                           // multisetS - std::multiset
               OutEdgeList,// kolekcja przechowująca krawędzie - parametr ma wartości takie jak
                   VertexList
               Directed,   // rodzaj grafu, znaczenie:
                           // directedS - graf skierowany
                           // undirectedS - graf nieskierowany
               VertexProperties, // typ obiektu związanego z wierzchołkiem, domyślnie no_property
               EdgeProperties,   // typ obiektu związanego z krawdzią, domyślnie no_property
               GraphProperties>  //typ obiektu zwizanego z caym grafem, domyślnie no_property
```

Listing 2. *Tworzenie i modyfikacja grafu*

```
typedef adjacency_list<vecS, vecS, directedS> Graph; //typ grafu
enum { A, B, C, D, E, N }; //identyfikatory wierzchołków, N - liczba wierzchołków
typedef std::pair<int, int> Edge; //para liczb - reprezentuje krawędź
const Edge EDGE_ARRAY[] = { Edge(A,B), Edge(A,E),
                            Edge(B,A), Edge(B,C), Edge(B,D),
                            Edge(D,E), Edge(E,B) };
const int NUM_EDGES = sizeof(EDGE_ARRAY)/sizeof(EDGE_ARRAY[0]);
Graph g(EDGE_ARRAY, EDGE_ARRAY + NUM_EDGES, N);
Graph::vertex_descriptor v = add_vertex(g); //dodaje nowy wierzchołek
add_edge(A, v, g); //dodaje nową krawędź
clear_vertex(D, g); //usuwa krawędzie związane z wierzchołkiem D
remove_vertex(D, g); //usuwa wierzchołek
```

ten wierzchołek. Po tych modyfikacjach będziemy przechowywali graf pokazany na Rysunku 3.

Własne dane opisujące wierzchołki i krawędzie

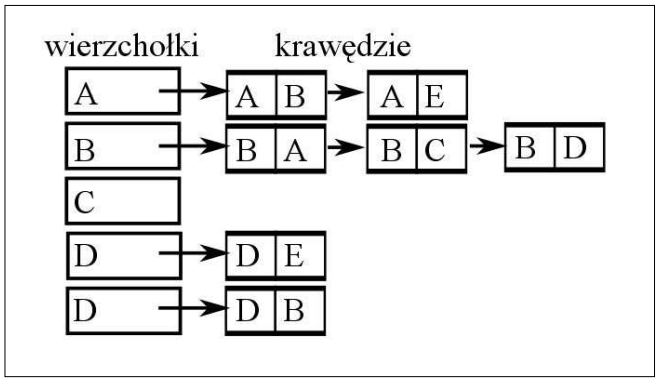
Jeżeli istnieje potrzeba dołączenia własnych danych, które opisują wierzchołek lub krawędź i które mają być zarządzane (tworzone i usuwane) przez klasy reprezentujące wierzchołek lub krawędź, należy dostarczyć odpowiedni typ jako parametr szablonu opisującego graf. Poniżej pokazano graf, który przechowuje strukturę `CityData` wraz z wierzchołkami oraz `RoadData` wraz z krawędziami.

Zamiast tworzyć własne typy, takie jak `CitiData` czy `RoadData`, możemy wykorzystać zbiór etykiet dostarczany przez bibliotekę. Wtedy odpowiedni parametr szablonu generujemy wykorzystując szablon `property`, przykład pokazano na Listingu 4, gdzie w ten sposób dostarcza się liczbę całkowitą przechowywaną w każdej krawędzi.

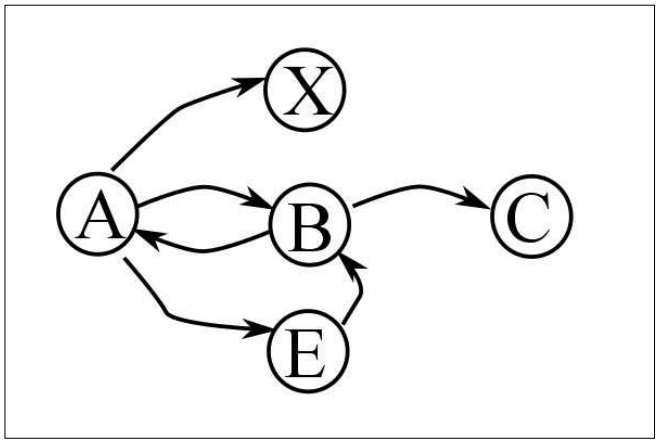
Znajdowanie najkrótszych ścieżek o wspólnym początku – Algorytm Dijkstry

Mając graf składający się z wierzchołków i krawędzi łączących te wierzchołki możemy przyporządkować poszczególnym krawędziom etykiety (liczby) nazywane wagami. Etykiety te oznaczają koszt przejścia pomiędzy wierzchołkami połączonymi krawędzią. Jednym z typowych problemów grafowych jest problem znajdowania najkrótszych ścieżek (o najmniejszej wadze łącznej) z danego wierzchołka, zwanego źródłem, do wszystkich pozostałych wierzchołków w grafie. Problem ten rozwiązuje się zarówno dla grafów skierowanych, jak i nieskierowanych. Będziemy rozwiązywać ten problem przy pomocy biblioteki boost::graph.

Jeżeli wierzchołek będzie reprezentował miasto, krawędź – drogę pomiędzy miastami, zaś waga krawędzi to



Rysunek 2. *Graf przechowywany jako lista sąsiedztwa*



Rysunek 3. *Graf pokazany na Rysunku 1 po modyfikacjach*

długość tej drogi, to wybierając dane miasto i stosując algorytm do znajdowania najkrótszych ścieżek otrzymamy wykaz najkrótszych dróg z zadanego miasta do wszystkich pozostałych. Należy odnotować że waga krawędzi nie musi mieć czysto geometrycznej interpretacji (jaką jest odległość między miastami), wagą krawędzi może

Listing 3. *Graf przechowujący obiekty użytkownika w wierzchołkach i krawędziach*

```
struct CityData { //obiekty związane z wierzchołkiem
   CityData() : name(""), population(0) {}
   string name;
   int population;
};
struct RoadData { //obiekty związane z krawędzią
   RoadData() : length(0) {}
   int length;
};
//graf przechowujący w wierzchołkach obiekty typu CityData oraz obiekty typu RoadData w krawędziach
typedef adjacency_list<vecS, vecS, directedS, CityData, RoadData> Graph;
Graph g(3); //obiekt grafu (trzy wierzchołki, brak krawędzi)
Graph::vertex_descriptor v = *vertices(g).first; //identyfikator pierwszego wierzchołka
g[v].name = "Nowe Miasto"; //g[v] zwraca referencję na obiekt CityData
g[v].population = 2000;
```

Tabela 1: *Algorytmy znajdowania najkrótszych ścieżek*

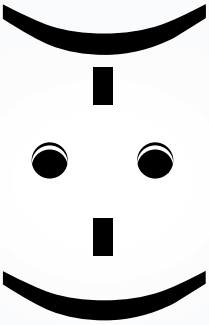
Nazwa algorytmu	W bibliotece BGL	Kiedy stosujemy
Dijkstra	dijkstra_shortest_paths	wagi krawędzi dodatnie
Bellmana-Forda	bellman_ford_shortest_path	wagi krawędzi mogą być ujemne
Przeszukiwanie wszerz (BFS)	breadth_first_search	wagi krawędzi równe 1

Listing 4. *Tworzenie grafu z wagami krawędzi.*

```
typedef adjecency_list<
   vecS,
   vecS,
   directedS,
   no_property, // brak dodatkowych właściwości dla wierzchołków
   property<edge_weight_t, unsigned int> // dodatkowa właściwość dla krawędzi
> Graph;
/* tutaj inicjacja tablicy EDGE_ARRAY tak jak pokazuje Listing 2 */
unsigned int weights[] = { /* tu podajemy wagi krawędzi */ };
Graph graph(
   EDGE_ARRAY,
   EDGE_ARRAY + NUM_EDGES,
   weights, // tablica wag krawędzi
   N);
```

Reklama

Właśnie rozpoczęliśmy rekrutację w Polsce. **I ty możesz być szczęściarzem!**



Właśnie rozpoczęliśmy rekrutację w Polsce. **Więcej wiary w siebie!**

Listing 5. *Obliczanie najkrótszych ścieżek.*

```
vector<int> distance(NUM_VERTICES);
const int START_VERTEX = C;
dijkstra_shortest_paths(
   graph,
   vertex(START_VERTEX, graph),
   distance_map(&distance[0])
);
for(int i = 0; i < NUM_VERTICES; ++i)
   cout << "Odległość do wierzchołka" <<
                  (char)(i+'A')<< "wynosi" <<
                  distance[i] << endl;
```

być też czas podróży, a wtedy rozwiązanie problemu zawiera wykaz najkrótszych czasów połączeń między wybranym miastem, a wszystkimi pozostałymi miastami.

Istnieje wiele algorytmów rozwiązujących problem znajdowania najkrótszych ścieżek, biblioteka boost::graph udostępnia takie, jak przedstawiono w Tabeli 1.

Przedstawimy sposób wykorzystania algorytmu Dijkstry dla prostego grafu. Algorytm ten możemy wykorzystywać dla grafów, w których wagi krawędzi są dodatnie. Typ reprezentujący graf, przedstawiony na Listingu 2, modyfikujemy dodając składową typu `unsigned int` związaną z każdą krawędzią. Składowa ta będzie reprezentowała wagę krawędzi. Wykorzystujemy dostarczany przez bibliotekę szablon `property` oraz tag `edge_weight_t`.

Następnie tworzymy tablicę liczb całkowitych zawierających, wagi krawędzi, w następujący sposób: i-ta wartość tablicy wag odpowiada i-tej krawędzi w tablicy krawędzi `EDGE_ARRAY`, wykorzystywanej przy tworzeniu grafu. Tablice wag krawędzi przekazujemy jako parametr podczas tworzenia grafu, co zostało pokazane na Listingu 4.

W celu obliczenia najkrótszych odległości, musimy wybrać wierzchołek startowy, a także stworzyć dodatkowy kontener, w którym te najkrótsze odległości zostaną zapisane. Po takich przygotowaniach możemy wywołać funkcję `dijkstra_shortest_paths`, obliczającą najkrótsze odległości co zostało pokazane na Listingu 5. Obliczone najkrótsze odległości zostaną zapisane do przekazanego funkcji kontenera. Pozostaje wypisać efekty działania algorytmu.

Podsumowanie

Biblioteka boost::graph zawiera generyczną reprezentację grafu oraz zbiór kilkudziesięciu algorytmów grafowych, między innymi przeglądające grafy (wszerz, w głąb, wykorzystując heurystyki, topologicznie), znajdujące najkrótsze ścieżki, minimalne drzewa rozpinające, badające przepływy, badające spójność i inne metryki, itd. Dostępne są rozszerzenia, m.in. biblioteka umożliwiająca współbieżną (rozproszoną) realizację algorytmów grafowych. Istnieją także wersje tej biblioteki dostępne w innych językach programowania.

ANDRZEJ MICHAŁOWSKI
Student Informatyki Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej oraz Ekonomii Wydziału Nauk Ekonomicznych Uniwersytetu Warszawskiego. Zainteresowania to programowanie, sport, finanse.
Kontakt z autorem:
a.m.michalowski@gmail.com

ROBERT NOWAK
Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji wykorzystujących algorytmy sztucznej inteligencji i fuzji danych. Autor biblioteki faif.sourceforge.net. Programuje w C++ od ponad 15 lat.
Kontakt z autorem:rno@o2.pl

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost, w tym boost::graph;
- *http://home.agh.edu.pl/~horzyk/lectures/pi/ahdydpiwykl9.html* – opis różnych rodzajów grafów;
- *http://osl.iu.edu/~dgregor/bgl-python/* – interfejs do boost::graph z języka Python;
- *http://jung.sourceforge.net/* – biblioteka do grafów w Javie.

Więcej w książce

Omówienie współcześnie stosowanych technik, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, opisano w książce Robert Nowak, Andrzej Pająk „Język C++: mechanizmy, wzorce, biblioteki", BTC 2010.