Algorytmy Grafowe

Dokumentacja zadań grafowych

Paweł Linek, grupa 8
Studia niestacjonarne, Informatyka

Część I

Opis Programu

Zaimplementuj następujące algorytmy grafowe:

- Prima
- Dijkstry
- A*

Program implementuje trzy klasyczne algorytmy grafowe: Dijkstrę, Prima oraz A*. Celem zadań było stworzenie efektywnych rozwiązań do znajdowania najkrótszych ścieżek oraz minimalnego drzewa rozpinającego w grafach. Program umożliwia wprowadzenie grafu reprezentowanego jako lista sąsiedztwa, wykonanie wybranego algorytmu na tym grafie oraz mierzenie czasu wykonania każdego z nich. Dodatkowo, program pozwala na generowanie losowych grafów, co umożliwia przeprowadzanie testów na różnych rozmiarach i strukturach grafów.

Instrukcja Obsługi

Aby uruchomić program, skompiluj plik źródłowy przy użyciu kompilatora C++ (np. g++):

```
g++ -o grafy main.cpp -std=c++11
./grafy
```

Program automatycznie wykonuje algorytmy na predefiniowanych grafach oraz na dużym, losowo wygenerowanym grafie. Wyniki działania algorytmów oraz zmierzone czasy wykonania są wyświetlane w konsoli. Można modyfikować grafy oraz punkty startowe i końcowe bezpośrednio w kodzie, a także dodawać dodatkowe pomiary w sekcji `main()`.

Dodatkowe Informacje

Wymagania:

- Kompilator C++ (obsługujący standard C++11 lub nowszy)
- Środowisko umożliwiające kompilację i uruchomienie programów w C++

Część II

Opis Działania

Algorytmy grafowe są fundamentem w wielu zastosowaniach informatycznych, takich jak nawigacja, sieci komputerowe czy analiza danych. W projekcie zaimplementowano trzy algorytmy:

- **Algorytm Dijkstry** – służy do znajdowania najkrótszej ścieżki z jednego źródła do wszystkich innych wierzchołków w grafie z nieujemnymi wagami krawędzi.
- **Algorytm Prima** – znajduje minimalne drzewo rozpinające (MST) w grafie ważonym, które łączy wszystkie wierzchołki z minimalną sumą wag krawędzi.
- **Algorytm A*** – rozszerza algorytm Dijkstry o heurystykę, co pozwala na bardziej efektywne znajdowanie najkrótszej ścieżki między dwoma wierzchołkami.

Teoria Algorytmów

Algorytm Dijkstry działa na zasadzie iteracyjnego wybierania wierzchołka o najmniejszej znanej odległości od źródła i aktualizowania odległości jego sąsiadów. Wykorzystuje strukturę danych kolejki priorytetowej, co zapewnia efektywne zarządzanie wierzchołkami do przetworzenia.

Data: Graf jako lista sąsiedztwa, wierzchołek źródłowy
Result: Najkrótsze odległości od źródła do wszystkich wierzchołków
Inicjalizuj odległości wszystkich wierzchołków jako INF;
Ustaw odległość źródła na 0;
Inicjalizuj kolejkę priorytetową i dodaj źródło z odległością 0;
while *kolejka nie jest pusta* **do**
  Wybierz wierzchołek u z najmniejszą odległością;
  for *każdego sąsiada v wierzchołka u* **do**
    if $dist[u] + w(u, v) < dist[v]$ **then**
      Zaktualizuj $dist[v]$;
      Dodaj v do kolejki z nową odległością;
    end
  end
end

Algorithm 1: Pseudokod Algorytmu Dijkstry

Algorytm Prima buduje minimalne drzewo rozpinające poprzez dodawanie do drzewa najtańszej dostępnej krawędzi, która łączy drzewo z nowym wierzchołkiem. Również korzysta z kolejki priorytetowej do wyboru kolejnych krawędzi.

Data: Graf jako lista sąsiedztwa
Result: Minimalne drzewo rozpinające (MST)
Inicjalizuj klucze wszystkich wierzchołków jako INF;
Wybierz wierzchołek początkowy i ustaw jego klucz na 0;
Inicjalizuj kolejkę priorytetową i dodaj początkowy wierzchołek z kluczem 0;
Inicjalizuj tablicę rodziców na -1;
while *kolejka nie jest pusta* **do**
    Wybierz wierzchołek u z najmniejszym kluczem;
    Dodaj u do MST;
    for *każdego sąsiada* v *wierzchołka* u **do**
        if $w(u, v) < klucz[v]$ *i* v *nie jest w MST* **then**
            Zaktualizuj klucz v;
            Ustaw rodzica v na u;
            Dodaj v do kolejki priorytetowej z nowym kluczem;
        end
    end
end

Algorithm 2: Pseudokod Algorytmu Prima

Algorytm A* rozszerza Dijkstrę o funkcję heurystyczną, która szacuje odległość od bieżącego wierzchołka do celu. Dzięki temu algorytm A* może skupić się na bardziej obiecujących ścieżkach, co często prowadzi do szybszego znalezienia optymalnej trasy.

Data: Graf jako lista sąsiedztwa, współrzędne wierzchołków, wierzchołek startowy, wierzchołek docelowy
Result: Najkrótsza ścieżka z startu do celu
Inicjalizuj odległości wszystkich wierzchołków jako INF;
Ustaw odległość startu na 0;
Inicjalizuj kolejkę priorytetową i dodaj start z fScore 0;
Inicjalizuj tablicę rodziców na -1;
while *kolejka nie jest pusta* **do**
  Wybierz wierzchołek u z najmniejszym fScore;
  if *u jest celem* **then**
    Przerwij;
  end
  for *każdego sąsiada v wierzchołka u* **do**
    Oblicz tentative_gScore = $dist[u] + w(u, v)$;
    if *tentative_gScore* $< dist[v]$ **then**
      Zaktualizuj $dist[v]$;
      Ustaw rodzica v na u;
      Oblicz fScore = tentative_gScore + heuristic(v, cel);
      Dodaj v do kolejki priorytetowej z fScore;
    end
  end
end

Algorithm 3: Pseudokod Algorytmu A*

Algorytm

Implementacje algorytmów zostały wykonane w języku C++. Każdy z algorytmów korzysta z listy sąsiedztwa do reprezentacji grafu oraz kolejki priorytetowej do efektywnego wybierania kolejnych wierzchołków do przetworzenia. Dodatkowo, algorytmy są zmodyfikowane tak, aby mierzyć czas ich wykonania za pomocą biblioteki `chrono`.

Złożoność Obliczeniowa

- **Algorytm Dijkstry**: $O((V + E) \log V)$, gdzie V to liczba wierzchołków, a E liczba krawędzi. Zastosowanie kolejki priorytetowej opartej na kopcu binarnym pozwala na osiągnięcie tej złożoności. Inne implementacje, takie jak z kopcem Fibonacciego, mogą zmniejszyć złożoność do $O(V \log V + E)$.
- **Algorytm Prima**: $O((V + E) \log V)$, analogicznie do Dijkstry, dzięki użyciu kolejki priorytetowej opartej na kopcu binarnym. Podobnie, zastosowanie innych struktur danych, jak kopiec Fibonacciego, może wpłynąć na efektywność.
- **Algorytm A***: Teoretyczna złożoność jest zbliżona do $O((V + E) \log V)$, jednak w praktyce dzięki zastosowaniu heurystyki algorytm może działać szybciej, kierując się w

stronę celu, co redukuje liczbę przetwarzanych wierzchołków.

Mimo że teoretyczna złożoność algorytmu A* jest podobna do Dijkstry, heurystyka umożliwia bardziej inteligentne przeszukiwanie przestrzeni, co w praktyce często prowadzi do szybszego znalezienia optymalnej ścieżki, zwłaszcza w dużych grafach.

Fragmenty Kodów Algorytmów

Listing 1: Implementacja Dijkstry w C++

```
void dijkstra(const vector<vector<pair<int, int>>>& graph, int source,
    double& duration) {
    // Implementacja algorytmu Dijkstry
    // ...
}
```

Listing 2: Implementacja Prima w C++

```
void prim(const vector<vector<pair<int, int>>>& graph, double& duration)
    {
    // Implementacja algorytmu Prima
    // ...
}
```

Listing 3: Implementacja A* w C++

```
vector<int> a_star(const vector<vector<pair<int, int>>>& graph, const
    vector<pair<int, int>>& coordinates, int start, int goal, double&
    duration, bool shouldIgnoreHeuristic) {
    // Implementacja algorytmu A*
    // ...
}
```

Pełny Kod Aplikacji

Listing 4: Pełny Kod Aplikacji

```
// main.cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <cmath> // Dla funkcji sqrt
#include <chrono> // Dla pomiaru czasu
#include <cstdlib> // Dla rand() i srand()
#include <ctime>   // Dla time()
using namespace std;
using namespace std::chrono;

const int INF = numeric_limits<int>::max();

// Funkcja do generowania losowego grafu
vector<vector<pair<int, int>>> generateRandomGraph(int V, int E) {
    vector<vector<pair<int, int>>> graph(V, vector<pair<int, int>>());
    srand(time(0));
    for (int i = 0; i < E; ++i) {
        int u = rand() % V;
        int v = rand() % V;
        if (u == v) continue; // Unikaj p tli
        int weight = rand() % 10 + 1; // Wagi od 1 do 10
        // Sprawd , czy kraw d   ju  istnieje
        bool exists = false;
        for (auto& edge : graph[u]) {
            if (edge.first == v) {
                exists = true;
                break;
            }
        }
        if (!exists) {
            graph[u].emplace_back(v, weight);
            graph[v].emplace_back(u, weight);
        }
    }
    return graph;
}

vector<vector<pair<int, int>>> generateRandomConnectedGraph(int V, int E
    , const vector<pair<int, int>>& coordinates) {
    vector<vector<pair<int, int>>> graph(V, vector<pair<int, int>>());
    srand(time(0));

    for (int i = 1; i < V; ++i) {
        int u = rand() % i;
        double dist = sqrt(pow(coordinates[i].first - coordinates[u].
            first, 2) +
          pow(coordinates[i].second - coordinates[u].second, 2));
        graph[u].emplace_back(i, static_cast<int>(dist));
```

```
49          graph[i].emplace_back(u, static_cast<int>(dist));
50      }
51
52      // Dodawanie dodatkowych kraw dzi
53      int additionalEdges = E - (V - 1);
54      for (int i = 0; i < additionalEdges; ++i) {
55          int u = rand() % V;
56          int v = rand() % V;
57          if (u == v) continue; // Unikaj p tli
58
59          bool exists = false;
60          for (auto& edge : graph[u]) {
61              if (edge.first == v) {
62                  exists = true;
63                  break;
64              }
65          }
66          if (exists) continue;
67
68          // Oblicz odleg o    euklidesow  jako wag
69          double dist = sqrt(pow(coordinates[v].first - coordinates[u].
                first, 2) +
70            pow(coordinates[v].second - coordinates[u].second, 2));
71          graph[u].emplace_back(v, static_cast<int>(dist));
72          graph[v].emplace_back(u, static_cast<int>(dist));
73      }
74
75      return graph;
76  }
77
78  // Implementacja Dijkstry za pomoc  kolejki priorytetowej
79  void dijkstra(const vector<vector<pair<int, int>>>& graph, int source,
        double& duration) {
80      int n = graph.size(); // Liczba wierzcho k w
81      vector<int> dist(n, INF); // Inicjalizacja odleg o ci na INF
82      dist[source] = 0; // Odleg o    do    rda    wynosi 0
83
84      // Kolejka priorytetowa: (odleg o   , wierzcho ek)
85      priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq
            ;
86      pq.push({0, source}); // Poprawione: najpierw odleg o   , potem
            wierzcho ek
87
88      auto start_time = high_resolution_clock::now();
89
90      while (!pq.empty()) {
91          auto current = pq.top();
92          int newDist = current.first;
93          int node = current.second;
94          pq.pop();
95
96          // Je li znaleziony dystans jest wi kszy ni  obecny, pomijamy
97          if (newDist > dist[node]) {
98              continue;
99          }
```

```
100
101         for (auto& edge : graph[node]) {
102             int v = edge.first;    // S siad
103             int weight = edge.second; // Waga kraw dzi
104
105             if (dist[node] + weight < dist[v]) {
106                 dist[v] = dist[node] + weight;
107                 pq.push({ dist[v], v });
108             }
109         }
110     }
111
112     auto end_time = high_resolution_clock::now();
113     duration = duration_cast<microseconds>(end_time - start_time).count
        () / 1e6;
114
115     if (n < 20) {
116         for (int i = 0; i < n; ++i) {
117             cout << "Vertex: " << i << ", Distance: " << (dist[i] == INF
                ? -1 : dist[i]) << endl;
118         }
119     }
120 }
121
122 // Implementacja Prima za pomoc  kolejki priorytetowej
123 void prim(const vector<vector<pair<int, int>>>& graph, double& duration)
        {
124     int n = graph.size();
125     vector<int> key(n, INF); // klucz do wybrania min kraw dzi
126     vector<bool> inMst(n, false); // vector do sprawdzenia czy
        wierzcho ek ju  zosta  dodany
127     vector<int> parent(n, -1); //tablica 'rodzic w'
128
129     key[0] = 0;
130     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq
        ;
131     pq.push({0, 0}); // Wierzcho ek 0 z kluczem 0
132
133     auto start_time = high_resolution_clock::now();
134
135     while (!pq.empty()) {
136         int u = pq.top().second; //wierzcho ek
137         pq.pop();
138
139         if (inMst[u]) continue;
140         inMst[u] = true;
141
142         for (auto& edge : graph[u]) {
143             int v = edge.first; //s siad
144             int weight = edge.second; //waga
145
146             if (!inMst[v] && weight < key[v]) {
147                 key[v] = weight;
148                 parent[v] = u;
149                 pq.push({ key[v], v });
```

```
150             }
151         }
152     }
153 
154     auto end_time = high_resolution_clock::now();
155     duration = duration_cast<microseconds>(end_time - start_time).count
        () / 1e6;
156 
157     if (n < 20) {
158         cout << "Minimal Spanning Tree (MST):\n";
159         for (int i = 1; i < n; ++i) {
160             if (parent[i] != -1) {
161                 cout << parent[i] << " -- " << i << " (weight: " << key[
                    i] << ")\n";
162             }
163         }
164     }
165 }
166 
167 // Heurystyka: Odleg o    euklidesowa
168 double heuristic(int x1, int y1, int x2, int y2, bool shouldIgnore) {
169     return !shouldIgnore ? sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1
        - y2)) : 0;
170 }
171 
172 // Implementacja algorytmu A*
173 vector<int> a_star(const vector<vector<pair<int, int>>>& graph, const
    vector<pair<int, int>>& coordinates, int start, int goal, double&
    duration, bool shouldIgnoreHeuristic) {
174     int n = graph.size();
175     vector<double> dist(n, INF); // Odleg o ci
176     vector<int> parent(n, -1); // Poprzednicy
177     dist[start] = 0;
178 
179     // Kolejka priorytetowa: (fScore, wierzcho ek)
180     priority_queue<pair<double, int>, vector<pair<double, int>>, greater
        <>> pq;
181     pq.push({0, start});
182 
183     auto start_time = high_resolution_clock::now();
184 
185     while (!pq.empty()) {
186         int u = pq.top().second;
187         pq.pop();
188 
189         if (u == goal) break; // Znaleziono cel
190 
191         // Przegl daj s siad w
192         for (auto& edge : graph[u]) {
193             int v = edge.first;
194             int weight = edge.second;
195 
196             double tentative_gScore = dist[u] + weight;
197             if (tentative_gScore < dist[v]) {
198                 dist[v] = tentative_gScore;
```

```
199                     parent[v] = u;
200
201                     // Oblicz fScore (g + h)
202                     double fScore = dist[v] + heuristic(coordinates[v].first
                            , coordinates[v].second, coordinates[goal].first,
                            coordinates[goal].second, shouldIgnoreHeuristic);
203                     pq.push({ fScore, v });
204                 }
205             }
206         }
207
208         auto end_time = high_resolution_clock::now();
209         duration = duration_cast<microseconds>(end_time - start_time).count
                () / 1e6;
210
211         return parent;
212     }
213
214     // Funkcja do drukowania    cieki
215     void printPath(const vector<int>& parent, int start, int goal) {
216         if (parent[goal] == -1 && start != goal) {
217             cout << "No path found for " << goal << "\n";
218             return;
219         }
220         vector<int> path;
221         for (int at = goal; at != -1; at = parent[at]) {
222             path.push_back(at);
223         }
224         reverse(path.begin(), path.end());
225         cout << "A* Path: ";
226         for (int node : path) {
227             cout << node << " ";
228         }
229         cout << endl;
230     }
231
232     int main() {
233
234         //kod podzielony na sekcje na dole, jezeli chcemy odpali  np
                algorytm dijsktry na ma ym grafie to odkomentowujemy ta sekcje
235
236
237         /*
238         // Przyk adowy graf dla algorytm w Dijkstry
239         vector<vector<pair<int, int>>> graph = {
240             {{1, 4}, {2, 1}}, // Wierzcho ek 0: kraw dzie do 1 (waga 4) i
                    2 (waga 1)
241             {{3, 1}},         // Wierzcho ek 1: kraw d  do 3 (waga 1)
242             {{1, 2}, {3, 5}}, // Wierzcho ek 2: kraw dzie do 1 (waga 2) i
                    3 (waga 5)
243             {}                // Wierzcho ek 3: brak kraw dzi
244         };
245
246         int source = 0;
247         double durationDijkstra = 0.0;
```

```
248     cout << "Dijkstra's Algorithm Results:\n";
249     dijkstra(graph, source, durationDijkstra);
250     cout << "Time taken: " << durationDijkstra << " seconds\n\n";
251     */
252
253
254
255     /*
256     // Przykadowy graf i wsprzdne dla algorytmu A*
257     vector<vector<pair<int, int>>> graphAStar = {
258         {{1, 1}, {2, 4}},  // Wierzchoek 0
259         {{0, 1}, {2, 2}, {3, 5}}, // Wierzchoek 1
260         {{0, 4}, {1, 2}, {3, 1}}, // Wierzchoek 2
261         {{1, 5}, {2, 1}}          // Wierzchoek 3
262     };
263
264     // Wsprzdne wierzchoka w (x, y)
265     vector<pair<int, int>> coordinates = {
266         {0, 0}, {1, 0}, {1, 1}, {2, 1}
267     };
268
269     int start = 0, goal = 3;
270     double durationAStar = 0.0;
271     vector<int> parent = a_star(graphAStar, coordinates, start, goal,
        durationAStar, false);
272     printPath(parent, start, goal);
273     cout << "Time taken: " << durationAStar << " seconds\n\n";
274     */
275
276
277
278     /*
279     // Przykadowy graf dla algorytmu Prima
280     vector<vector<pair<int, int>>> graphPrim = {
281         {{1, 2}, {3, 6}},  // Wierzchoek 0
282         {{0, 2}, {2, 3}, {3, 8}, {4, 5}}, // Wierzchoek 1
283         {{1, 3}, {4, 7}},  // Wierzchoek 2
284         {{0, 6}, {1, 8}},  // Wierzchoek 3
285         {{1, 5}, {2, 7}}   // Wierzchoek 4
286     };
287     double durationPrim = 0.0;
288     prim(graphPrim, durationPrim);
289     cout << "Time taken: " << durationPrim << " seconds\n\n";
290     */
291
292
293
294     // Przykadowe generowanie duego grafu i pomiar czasu
295     /*
296     int V = 1000; // Liczba wierzchoków
297     int E = 5000; // Liczba krawdzi
298     cout << "Generating random graph with " << V << " vertices and " <<
        E << " edges...\n";
299     vector<vector<pair<int, int>>> largeGraph = generateRandomGraph(V, E
        );
```

```
300     cout << "Graph generated.\n\n";
301     */
302
303     /*
304     // Pomiar czasu dla algorytmu Dijkstry na du ym grafie
305     double durationDijkstraLarge = 0.0;
306     cout << "Running Dijkstra's algorithm on large graph...\n";
307     dijkstra(largeGraph, 0, durationDijkstraLarge);
308     cout << "Dijkstra on large graph time: " << durationDijkstraLarge <<
            " seconds\n\n";
309     */
310
311
312
313     /*
314     // Pomiar czasu dla algorytmu Prima na du ym grafie
315     double durationPrimLarge = 0.0;
316     cout << "Running Prim's algorithm on large graph...\n";
317     prim(largeGraph, durationPrimLarge);
318     cout << "Prim on large graph time: " << durationPrimLarge << "
           seconds\n\n";
319     */
320
321
322
323     // Pomiar czasu dla algorytmu A* na du ym grafie (wymaga
           wsp  rz dnych)
324     // Generowanie losowych wsp  rz dnych
325     /*
326     int V = 10000; // Liczba wierzcho k w
327     int E = 50000; // Liczba kraw dzi
328     cout << "Generating random connected graph with " << V << " vertices
            and " << E << " edges...\n";
329
330     vector<pair<int, int>> coordinatesLarge(V, { 0, 0 });
331     for (int i = 0; i < V; ++i) {
332         coordinatesLarge[i].first = rand() % 1000;
333         coordinatesLarge[i].second = rand() % 1000;
334     }
335
336     vector<vector<pair<int, int>>> largeGraph =
           generateRandomConnectedGraph(V, E, coordinatesLarge);
337     cout << "Graph generated.\n\n";
338
339     int startLarge = 0, goalLarge = V - 1;
340     double durationAStarLarge = 0.0;
341     cout << "Running A* algorithm on large graph...\n";
342     vector<int> parentLarge = a_star(largeGraph, coordinatesLarge,
           startLarge, goalLarge, durationAStarLarge, false);
343     printPath(parentLarge, startLarge, goalLarge);
344     cout << "A* on large graph time: " << durationAStarLarge << "
           seconds\n";
345     */
346
347
```

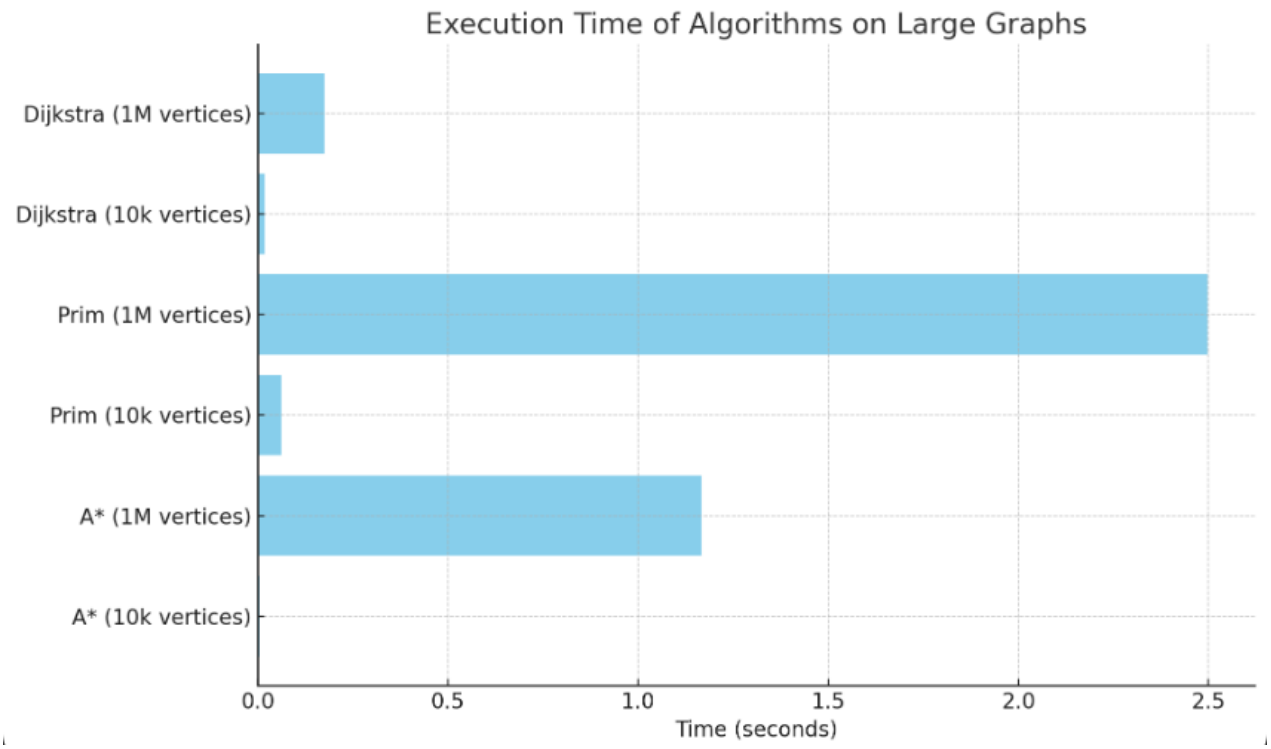
```
348         return 0;
349 }
```

Testy i Eksperymenty

W trakcie projektu przeprowadzono następujące eksperymenty:

- **Wpływ Liczby Wierzchołków na Wydajność**: Analizowano, jak zwiększenie liczby wierzchołków wpływa na czas działania każdego z algorytmów.
- **Porównano szybkości wykonania różnych algorytmów grafowych w stosunku do siebie**

Wszystkie wyniki zostały opisane i poddane analizie statystycznej. Na podstawie przeprowadzonych eksperymentów stwierdzono, że algorytm A* jest zazwyczaj szybszy od Dijkstry i Prima na dużych grafach z odpowiednią heurystyką, natomiast Dijkstra i Prim wykazują stabilną wydajność niezależnie od struktury grafu.



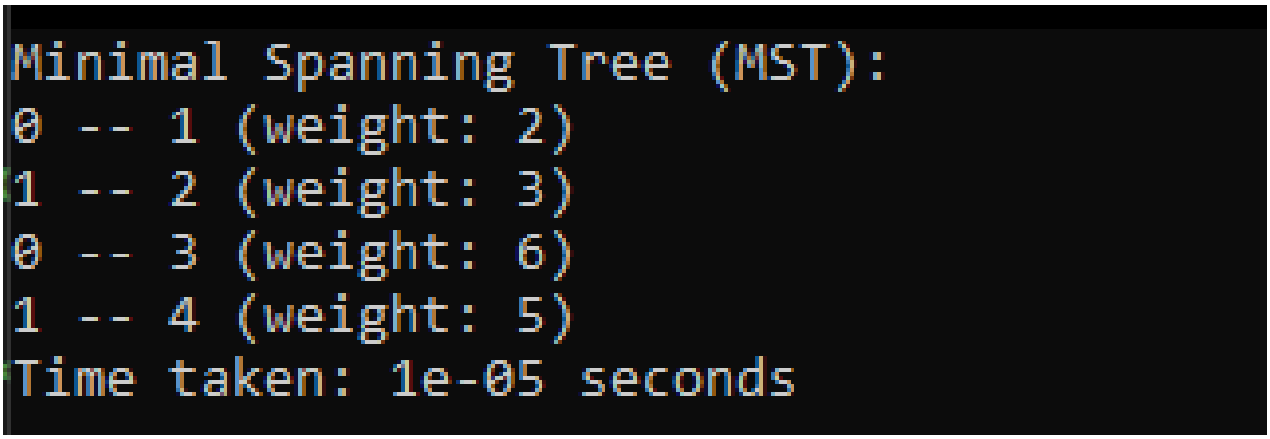
Rysunek 1: Wykres przedstawiający średnie czasy działania algorytmów

```
Dijkstra's Algorithm Results:
Vertex: 0, Distance: 0
Vertex: 1, Distance: 3
Vertex: 2, Distance: 1
Vertex: 3, Distance: 4
Time taken: 6e-06 seconds
```

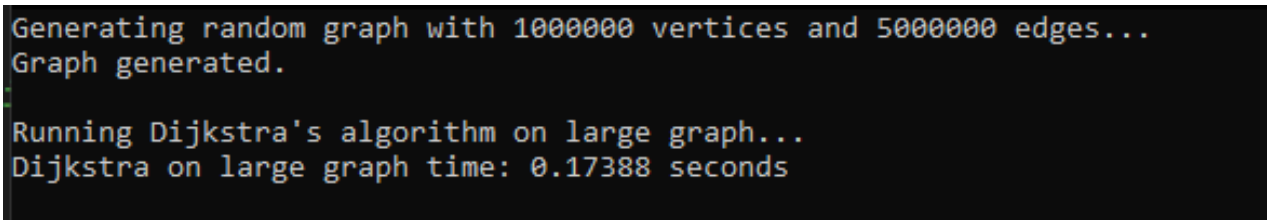
Rysunek 2: Rezultat dla Dijkstry na małym grafie

```
Konsola debugowania programu Microsoft Visual Studio
A* Path: 0 1 2 3
Time taken: 1.3e-05 seconds
```

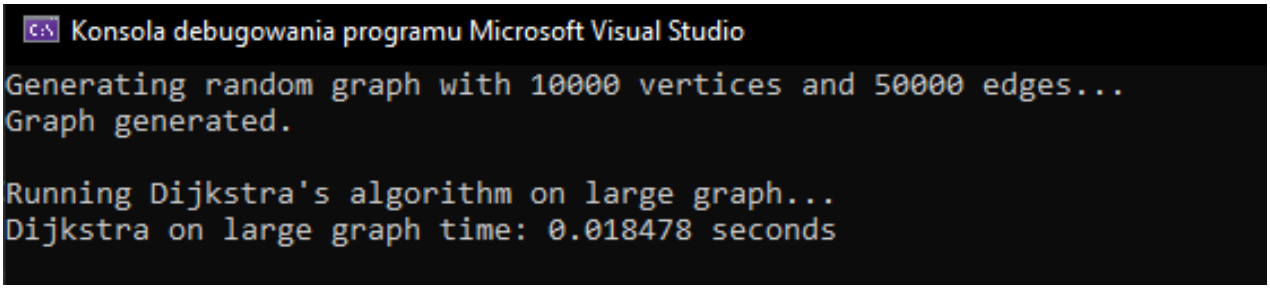
Rysunek 3: Rezultat dla A* na małym grafie



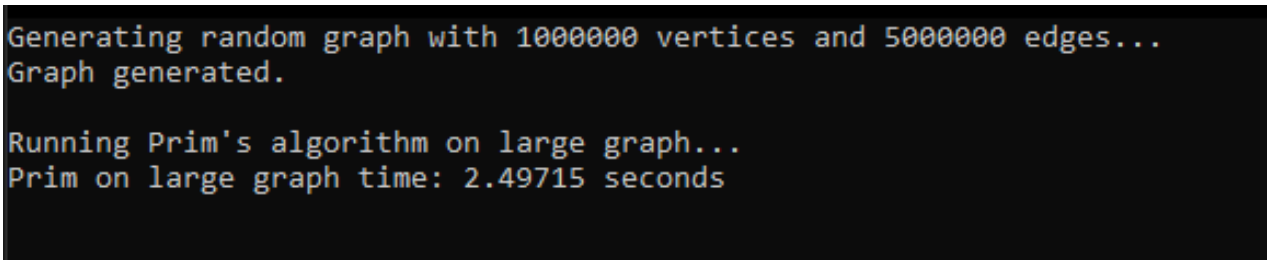
Rysunek 4: Rezultat dla Prima na małym grafie



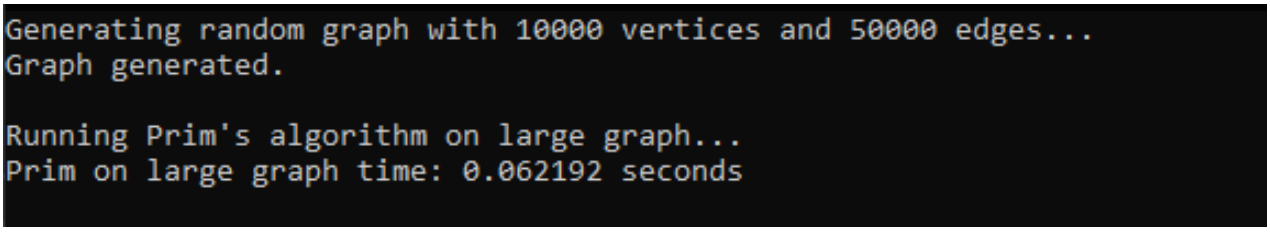
Rysunek 5: Dijkstra (1M vertices)



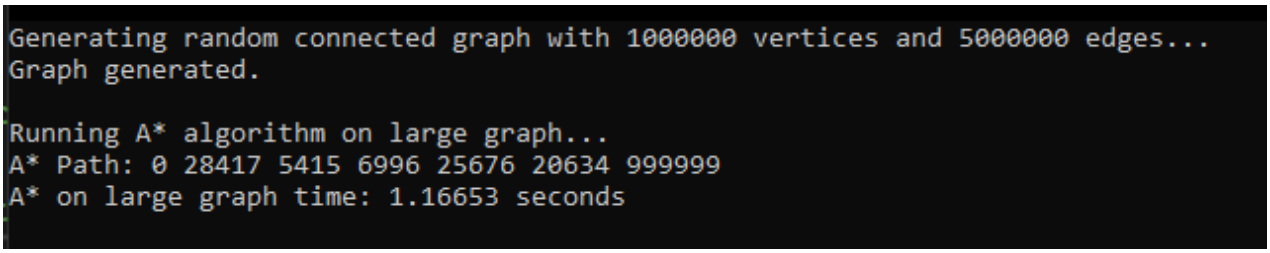
Rysunek 6: Dijkstra (10K vertices)



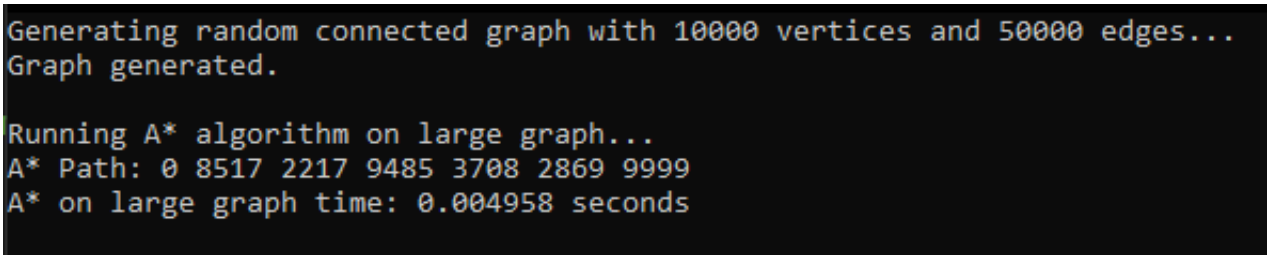
Rysunek 7: Prim (1M vertices)



Rysunek 8: Prim (10K vertices)



Rysunek 9: A* (1M vertices)



Rysunek 10: A* (10K vertices)