Zaawansowane Biblioteki Programistyczne

Sprawozdanie z projektu:

**Analiza porównawcza wydajności algorytmów grafowych biblioteki Boost z implementacją własną**

Skład sekcji:

Michał Krupiński

Mateusz Sulima

# Wstęp

Graf stanowi strukturę *G=(V, E)* składającą się z węzłów (wierzchołków, oznaczanych poprzez *V*) wzajemnie połączonych za pomocą krawędzi (oznaczonych poprzez *E*), które mogą posiadać wagi (oznaczane poprzez *w*).

Grafy dzielone są na skierowane i nieskierowane. Graf skierowany charakteryzuje się występowaniem zwrotu krawędzi *E*.

# Typy grafów

Do badania algorytmów wykorzystaliśmy trzy rodzaje grafów:

1. Graf gęsty, o wypełnieniu 100%
2. Graf o wypełnieniu 50%
3. Graf o topolgii mapy (rzadki) – połączeni wybrani sąsiedzi z 15 najbliższych wierzchołków

Ograniczyliśmy się do grafów nieskierowanych o nieujemnych wagach krawędzi.

# Badane algorytmy

## Przeszukiwanie grafu „w głąb”

Algorytm przeszukiwania grafu w głąb bada drogę aż do jej całkowitego wyczerpania. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka algorytm powraca do wierzchołka, z którego dany wierzchołek został odwiedzony.

Do jego działania potrzebna jest pomocnicza tablica V przechowującą informację o wierzchołkach, które zostały już odwiedzone. Stąd jego złożoność pamięciowa to . Złożoność czasowa zależy od liczby krawędzi i wierzchołków, więc wynosi .

void depth\_first(zbp::distance\_matrix G, bool \*V, size\_t n) {

  size\_t i;

  for (i = 0; i < n; i++) {

    V[i] = 0;

  }

  for (i = 0; i < n; i++) {

    if (V[i] == 0) {

      depth\_first\_visit(G, V, i, n);

    }

  }

}

void depth\_first\_visit(zbp::distance\_matrix G, bool \*V, int i, size\_t n) {

*//jakaś\_operacja(s);*

  V[i] = 1;

  for (size\_t k = 0; k < n; k++) {

    if (G[i][k] != zbp::NO\_EDGE) {

      if (V[k] == 0) {

        depth\_first\_visit(G, V, k, n);

      }

    }

  }

}

void original\_depth\_first(std::shared\_ptr<graph\_generator> graph) {

  bool \*V = new bool[graph->size()];

  depth\_first(graph->original\_graph(), V, graph->size());

  delete[] V;

}

## Przeszukiwanie grafu „wszerz”

Algorytm przeszukiwania grafu wszerz bada najpierw sąsiednie wierzchołki grafu. Kolejne wierzchołki, które mają zostać odwiedzone wstawiane są do kolejki, po czym procedura powtarza się dla następnego wierzchołka z kolejki tak długo, aż wszystkie wierzchołki zostaną odwiedzone.

Do działania również i tego algorytmu potrzebna jest pomocnicza tablica V przechowującą informacje o wierzchołkach, które zostały już odwiedzone oraz dodatkowa pamięć na kolejkę, która jednak nie będzie większa od . Stąd jego złożoność pamięciowa to . Złożoność czasowa zależy od liczby krawędzi i wierzchołków, więc wynosi .

void breadth\_first(zbp::distance\_matrix G, bool \*V, size\_t n) {

  std::queue<int> kolejka;

  for (size\_t i = 0; i < n; i++) {

    V[i] = false;

  }

  kolejka.push(0);

  while (!kolejka.empty()) {

    int s = kolejka.front();

    kolejka.pop();

*//jakaś\_operacja(s);*

    V[s] = 1;

    for (size\_t k = 0; k < n; k++) {

      if (G[s][k] != zbp::NO\_EDGE) {

        if (V[k] == false) {

          V[k] = true;

          kolejka.push(k);

        }

      }

    }

  }

}

void original\_breadth\_first(std::shared\_ptr<graph\_generator> graph) {

  bool \*V = new bool[graph->size()];

  breadth\_first(graph->original\_graph(), V, graph->size());

  delete[] V;

}

## Algorytm Floyda-Warshalla

Algorytm ten pozwala na znalezienie najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków w grafie. Korzysta on z tego, że jeśli najkrótsza ścieżka pomiędzy wierzchołkami i prowadzi przez wierzchołek , to jest ona połączeniem najkrótszych ścieżek pomiędzy wierzchołkami i oraz i .

W trakcie wykonywania algorytmu porównywane są wszystkie możliwe ścieżki pomiędzy każdą parą wierzchołków, stąd jego złożoność obliczeniowa wynosi . Wynikiem działania jest macierz z najkrótszymi ścieżkami pomiędzy parami wierzchołków. W przypadku, gdy algorytm działa w miejscu, nie jest mu potrzebna dodatkowa pamięć. W innym przypadku jego złożoność pamięciowa wynosi .

Przechowujemy graf w postaci macierzy sąsiedztwa, gdzie to waga krawędzi, tak więc w naszej implementacji niepotrzebna jest dodatkowa inicjalizacja struktur danych.

void floyd\_warshall(zbp::distance\_matrix G, size\_t n) {

  for (size\_t k = 0; k < n; k++) {

    for (size\_t i = 0; i < n; i++) {

      for (size\_t j = 0; j < n; j++) {

        G[i][j] = std::min(G[i][j], G[i][k] + G[k][j]);

      }

    }

  }

}

void original\_floyd\_warshall(std::shared\_ptr<graph\_generator> graph) {

  zbp::distance\_matrix m\_original\_graph = graph->original\_graph();

  zbp::distance\_matrix D = new zbp::weight\*[graph->size()];

  for (size\_t i = 0; i < graph->size(); i++) {

    D[i] = new zbp::weight[graph->size()];

    std::copy(m\_original\_graph[i], m\_original\_graph[i] + graph->size(), D[i]);

  }

  floyd\_warshall(D, graph->size());

#ifdef DEBUG

  std::cout << " ";

  for (size\_t k = 0; k < graph->size(); ++k) {

    std::cout << std::setw(5) << k;

  }

  std::cout << std::endl;

  for (size\_t i = 0; i < graph->size(); ++i) {

    std::cout << std::setw(3) << i << " -> ";

    for (size\_t j = 0; j < graph->size(); ++j) {

      if (D[i][j] == zbp::NO\_EDGE) {

        std::cout << std::setw(5) << "x";

      } else {

        std::cout << std::setw(5) << D[i][j];

      }

    }

    std::cout << std::endl;

  }

#endif

  for (size\_t i = 0; i < graph->size(); i++) {

    delete[] D[i];

  }

  delete[] D;

}

## Algorytm Dijkstry

Algorytm wykorzystywany jest do znalezienia najkrótszych ścieżek pomiędzy wybranym wierzchołkiem a wszystkimi pozostałymi. Dodatkowo wyliczany jest koszt przejścia z każdej z tych ścieżek. W algorytmie Dijkstry wagi krawędzi muszą być nieujemne. Algorytm jest podobny do algorytmu znajdującego minimalne drzewo rozpinające metodą dołączania do już znalezionego fragmentu drzewa kolejnych krawędzi, tak aby w każdym kroku drzewo było rozszerzane o tę krawędź, której waga wnosi najmniej do wagi całego drzewa.

Tablica przechowuje bieżące oszacowanie wagi najkrótszej drogi prowadzącej ze źródła *s* do danego wierzchołka.

Tablica *p* przechowuje poprzedników na drodze prowadzącej ze źródła do wierzchołków grafu.

bool dijkstra(zbp::distance\_matrix G, unsigned int source, size\_t n, std::vector<double>& d) {

int u;

int min\_d\_in\_q = INT\_MAX;

std::vector<double> p;

std::set<int> S;

std::set<int> Q;

// Inicjalizacja zmiennych

for (unsigned int i = 0; i < n; ++i) {

d.push\_back(INT\_MAX);

p.push\_back(0);

Q.insert(i);

}

d[source] = 0;

while(!Q.empty()) {

// u := wierzchołek z Q o minimalnej wartości d;

min\_d\_in\_q = INT\_MAX;

for (const int &q : Q) {

int actual\_d\_value = d[q];

if (min\_d\_in\_q == INT\_MAX) {

min\_d\_in\_q = actual\_d\_value;

u = q;

}

else if (actual\_d\_value < min\_d\_in\_q){

min\_d\_in\_q = actual\_d\_value;

u = q;

}

}

// S := S + {u};

if (min\_d\_in\_q != INT\_MAX) {

Q.erase(u);

S.insert(u);

}

// for lista wierzchołków v sąsiadujących z u -> relaksacja

for (unsigned int v = 0; v < n; ++v) {

int edgeWeight = G[u][v];

if (edgeWeight != zbp::NO\_EDGE) {

if (d[u] + edgeWeight < d[v]) {

d[v] = d[u] + edgeWeight;

p[v] = u;

}

}

}

}

return true;

}

## Algorytm Bellmana-Forda

Algorytm rozwiązuje problem najkrótszej ścieżki, to znaczy pozwala znaleźć ścieżkę o najmniejszej wadze pomiędzy dwoma wierzchołkami w grafie ważonym. W przeciwieństwie do algorytmu Dijkstry, algorytm pozwala na występowanie ujemnych wag krawędzi, jednak w grafie nie mogą występować ujemne cykle (o łącznej ujemnej wadze osiągalnej ze źródła). Algorytm działa w czasie O(V\*E).

Oznaczenia tablic są analogiczne do algorytmu Dijkstry.

bool bellman\_ford(zbp::distance\_matrix G, unsigned int source, size\_t n, std::vector<double>& d) {

d.clear();

std::vector<double> p;

// Krok 1:Inicjalizacja zmiennych

for (unsigned int i = 0; i < n; ++i) {

d.push\_back(INT\_MAX);

p.push\_back(0);

}

d[source] = 0;

// Krok 2: powtórna relaksacja

for (unsigned int i = 1; i < n; ++i) {

for (unsigned int u = 0; u < n; ++u) {

for (unsigned int v = 0; v < n; ++v) {

int edgeWeight = G[u][v];

if (edgeWeight != zbp::NO\_EDGE) {

if (d[u] + edgeWeight < d[v]) {

d[v] = d[u] + edgeWeight;

p[v] = u;

}

}

}

}

}

// Krok 3: sprawdzanie wystąpień ujemnych cykli

for (unsigned int u = 0; u < n; ++u) {

for (unsigned int v = 0; v < n; ++v) {

int edgeWeight = G[u][v];

if (edgeWeight != zbp::NO\_EDGE) {

if (d[u] + edgeWeight < d[v]) {

d.clear();

return false;

}

}

}

}

return true;

}

## Algorytm Johnsona

Algorytm służy do znajdowania najkrótszych ścieżek między wszystkimi parami wierzchołków. Działanie algorytmu opiera się na wykorzystaniu algorytmów Dijkstry oraz Bellmana-Forda. Działa w czasie O(V2lg V+VE).

W algorytmie Johnsona wykorzystuje się metodę zmieniających się wag. Polega ona na tym, że jeśli wszystkie wagi *w* krawędzi w grafie *G=(V,E)* są nieujemne, to najkrótsze ścieżki pomiędzy wszystkimi parami wierzchołków można znaleźć wykorzystując algorytm Dijkstry, z każdego wierzchołka oddzielnie. W celu uzyskania nieujemnych wartości wag, wykorzystuje się graf *G’=(V’,E’)*, który tworzy się poprzez dodanie do grafu *G* dodatkowego wierzchołka *s*, połączonego z każdym wierzchołkiem krawędziami wychodzącymi o wagach równych 0. Następnie graf *G’* poddaje się działaniu algorytmu Bellmana-Forda ze źródłem w dodanym wierzchołku *s*, który zwraca wektor odległości *h*, wykorzystywany do uzyskiwania nieujemnych wag grafu *G*. Po użyciu algorytmu Dijkstry dla każdego wierzchołka, macierz odległości jest odpowiednio modyfikowana w oparciu o użyty poprzednio wektor odległości *h*.

bool johnson(zbp::distance\_matrix G, unsigned int source, size\_t n, std::vector<std::vector<double>>& d) {

// Krok 1: Inicjalizacja zmiennych

d.clear();

std::vector<double> h;

// Krok 2: Utworzenie G2 poprzez dodanie nowego wierzcholka S

// z wagami 0 do kazdego wierzcholka

size\_t G2\_size = n + 1;

zbp::distance\_matrix G2 = new zbp::weight\*[G2\_size];

zbp::weight\* s = new zbp::weight[G2\_size] { };

G2[0] = s;

for (unsigned int i = 0; i < n; ++i) {

G2[i + 1] = new zbp::weight[G2\_size];

for (unsigned int j = 0; j < n; j++)

G2[i + 1][j + 1] = G[i][j];

// Do wierzcholka S nie wchodzi zadna krawedz

G2[i + 1][0] = zbp::NO\_EDGE;

}

// Krok 3: sprawdzenie wystepowania ujemnych cykli

if (!bellman\_ford(G2, 0, G2\_size, h))

return false;

// Krok 4: nadanie nieujemnych wartosci krawedzi w grafie G

for (unsigned int u = 0; u < n; ++u) {

for (unsigned int v = 0; v < n; ++v) {

int edgeWeight = G[u][v];

if (edgeWeight != zbp::NO\_EDGE) {

G[u][v] = G[u][v] + h[u + 1] - h[v + 1]; // +1 z uwagi na dodany wierzcholek S w G2

}

}

}

// Krok 5: obliczenie wartosci macierzy d

for (unsigned int u = 0; u < n; ++u) {

std::vector<double> k;

d.push\_back(k);

dijkstra(G, u, n, d[u]);

for (unsigned int v = 0; v < n; ++v) {

d[u][v] = d[u][v] + h[v + 1] - h[u + 1];

}

}

return true;

}

# Wykorzystane struktury danych

## Prosta macierz sąsiedztwa

Macierz sąsiedztwa jest to macierz kwadratowa, w której , jeśli istnieje krawędź pomiędzy wierzchołkami i lub w innym przypadku .

We wszystkich naszych programach wykorzystaliśmy najprostszą dla nas pod względem wykorzystania strukturę, będącą tablicą dwuwymiarową. Dodatkowo dokonaliśmy modyfikacji polegającej na tym, że jest równe wadze krawędzi pomiędzy wierzchołkami i lub, jeśli taka krawędź nie istnieje, to wybranej przez nas bardzo dużej liczbie.

typedef unsigned int weight;

typedef weight\*\* distance\_matrix;

## Macierz sąsiedztwa z biblioteki Boost

Dla algorytmów przeszukiwania w głąb i wszerz z użyciem biblioteki Boost wykorzystaliśmy implementację macierzy sąsiedztwa z tejże biblioteki:

 typedef boost::adjacency\_matrix<boost::undirectedS> BoostSimpleGraph;

## Nieskierowana lista sąsiedztwa z biblioteki Boost

Dla algorytmów Floyda-Warshalla, Dijkstry, Bellmana-Forda oraz Johnsona musieliśmy wykorzystać reprezentację grafu w postaci listy sąsiedztwa. Lista sąsiedztwa jest zazwyczaj implementowana w postaci tablicy wskaźników na listy jednokierunkowe, gdzie -ty element tablicy wskazuje na listę wierzchołków połączonych krawędzią z wierzchołkiem .

Ponieważ potrzebowaliśmy grafu ważonego, musieliśmy również zdefiniować własność krawędzi, określającą jej wagę:

 typedef boost::property<boost::edge\_weight\_t, zbp::weight> WeightProperty;

typedef boost::adjacency\_list<boost::vecS, boost::vecS, boost::undirectedS, boost::no\_property, WeightProperty> BoostWeightedGraph;

## Skierowana lista sąsiedztwa z biblioteki Boost

W testach pamięci sprawdziliśmy również implementację Boosta listy sąsiedztwa dla grafu nieskierowanego.

 typedef boost::adjacency\_list<boost::vecS, boost::vecS, boost::directedS, boost::no\_property, boost::property<boost::edge\_weight\_t, int, boost::property<boost::edge\_weight2\_t, int>>> BoostDirectedGraph;

# Testy zajętości pamięci przez wykorzystane struktury danych

Dokonaliśmy testów sprawdzających w jaki sposób zależy ilość pamięci operacyjnej wykorzystywanej przez program od rodzaju przechowywanego w niej grafu i wybranej struktury danych.

Do wykonania testów posłużył laptop z 8 GB pamięci RAM. Testy zostały wykonane dla liczb wierzchołków w zakresie od 100 do 12000. Górna granica liczby wierzchołków została podyktowana pojemnością pamięci RAM.

Wykresy przedstawiają różnicę pomiędzy wolną pamięcią operacyjną przed i po wygenerowaniu danego typu grafu (bez wykonywania żadnego z algorytmów). Są to wartości uśrednione dla 4 pomiarów. Oznaczenia etykiet wykresów mają następujące znaczenie:

* Original – zaimplementowana przez nas macierz sąsiedztwa
* Boost - adjacency\_matrix – macierz sąsiedztwa z biblioteki Boost
* Boost - adjacency\_list undirected – nieskierowana lista sąsiedztwa z biblioteki Boost
* Boost - adjacency\_list directed – nieskierowana lista sąsiedztwa z biblioteki Boost, w której ustawiono tylko krawędź .
* Boost - adjacency\_list double-directed – nieskierowana lista sąsiedztwa z biblioteki Boost, w której ustawiono krawędzi oraz , czyli teoretycznie jest to taka sama lista jak dla grafu nieskierowanego.

## Grafy o losowym wypełnieniu

## 100% wypełnienia

## 50% wypełnienia

## Wnioski

* Obie struktury oparte na macierzy sąsiedztwa zachowują (w przybliżeniu) ten sam rozmiar dla różnego wypełnienia grafu. Są też o wiele bardziej efektywne.
* Macierz sąsiedztwa z biblioteki Boost jest dużo bardziej efektywna od naszej implementacji. Tak dużej różnicy tej nie tłumaczy nawet to, że dla grafu nieskierowanego wystarczy przechowywać połowę macierzy (znajdującą się pod przekątną).
* Zajętość pamięci dla list sąsiedztwa jest liniowo zależna od liczby krawędzi. Wykresy przyjmują kształt paraboli ponieważ dla grafu z losowym wypełnieniem liczba krawędzi jest proporcjonalna do liczby wierzchołków. Można zauważyć, że dla wypełnienia 100% listy zajmują w przybliżeniu 2 razy więcej pamięci niż dla wypełnienia 50%.
* Zaskoczyło nas, że autorzy biblioteki Boost nie wykorzystali do zaimplementowania skierowanej listy sąsiedztwa gotowej implementacji listy nieskierowanej i faktu, że jeśli istnieje krawędź to istnieje też . Takie rozwiązanie poskutkowało tym, że implementacja ta jest o wiele mniej efektywna niż *Boost – adjacency\_list directed*.

## Graf o topologii mapy

## Wnioski

* Z niezrozumiałych dla nas powodów, nasza implementacja, pomimo takiej samej liczby wierzchołków co wcześniej, zajmuje dużo więcej miejsca w pamięci.
* Macierze sąsiedztwa z biblioteki Boost zajmuje tyle samo miejsca co dla grafu o losowym wypełnieniu.
* Implementacji w oparciu o listę sąsiedztwa zajmują stosunkowo mało pamięci. Jest to naturalne, gdyż dla grafu o losowym wypełnieniu 100% i wierzchołków będzie miliony krawędzi, a dla grafu o topologii mapy i połączonych sąsiednich wierzchołkach liczba krawędzi będzie równa tysięcy.

# Testy czasów wykonania algorytmów

Dokonaliśmy testów sprawdzających czasy wykonania poszczególnych algorytmów w zależności od liczby wierzchołków i typu wykresu. Zakres liczby wierzchołków użytych do testów nie jest identyczny dla każdego algorytmu, z uwagi na różną złożoność obliczeniową algorytmów. W celu uzyskania wiarygodnych wyników, generowanych było losowo 5 grafów o zadanej liczbie wierzchołków, dla których przeprowadzano testy. Typy wykorzystanych grafów zawarte są na początku niniejszej pracy, w sekcji „typy grafów”.

Do wykonania testów posłużył komputer stacjonarny z procesorem Intel Core i5-3570K @ 3.40GHz i 8GB RAM DDR3 1600Mhz.

## Przeszukiwanie grafu „w głąb”

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Original - 100% wypełnienia** | **Boost - 100% wypełnienia** | **Original - 50% wypełnienia** | **Boost - 50% wypełnienia** | **Original - mapa** | **Boost - mapa** |
| **1000** | 0,0042 | 0,0024 | 0,011 | 0,0028 | 0,0068 | 0,001 |
| **2000** | 0,0142 | 0,0092 | 0,0178 | 0,0116 | 0,0058 | 0,0034 |
| **3000** | 0,0354 | 0,0202 | 0,0448 | 0,029 | 0,02 | 0,0088 |
| **4000** | 0,0651 | 0,0362 | 0,0771 | 0,0492 | 0,0208 | 0,015 |
| **5000** | 0,0915 | 0,0567 | 0,1123 | 0,0797 | 0,0206 | 0,0266 |

## Przeszukiwanie grafu „wszerz”

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Original - 100% wypełnienia** | **Boost - 100% wypełnienia** | **Original - 50% wypełnienia** | **Boost - 50% wypełnienia** | **Original - mapa** | **Boost - mapa** |
| **1000** | 0,0098 | 0,0026 | 0,01 | 0,003 | 0,0072 | 0,0008 |
| **2000** | 0,0144 | 0,0098 | 0,0206 | 0,0136 | 0,0052 | 0,0036 |
| **3000** | 0,0418 | 0,0208 | 0,0468 | 0,0298 | 0,0118 | 0,0078 |
| **4000** | 0,0643 | 0,038 | 0,0741 | 0,0553 | 0,0176 | 0,016 |
| **5000** | 0,0989 | 0,0617 | 0,1119 | 0,0865 | 0,022 | 0,0282 |

## Algorytm Floyda-Warshalla

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Original - 100% wypełnienia** | **Boost - 100% wypełnienia** | **Original - 50% wypełnienia** | **Boost - 50% wypełnienia** | **Original - mapa** | **Boost - mapa** |
| **100** | 0,0012 | 0,0228 | 0,0006 | 0,0112 | 0,0006 | 0,0064 |
| **200** | 0,0048 | 0,1465 | 0,0052 | 0,1061 | 0,0048 | 0,04 |
| **300** | 0,0122 | 0,4254 | 0,0122 | 0,2997 | 0,0124 | 0,1099 |
| **400** | 0,029 | 0,9879 | 0,0286 | 0,7067 | 0,0286 | 0,2594 |
| **500** | 0,0524 | 1,4804 | 0,0557 | 1,3745 | 0,0543 | 0,4961 |

## Algorytm Dijkstry

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Original - 100% wypełnienia** | **Boost - 100% wypełnienia** | **Original - 50% wypełnienia** | **Boost - 50% wypełnienia** | **Original - mapa** | **Boost - mapa** |
| **250** | 0,0008 | 0,3031 | 0,0006 | 0,0599 | 0,0004 | 0,0096 |
| **500** | 0,0034 | 0,3093 | 0,0034 | 0,2356 | 0,0018 | 0,0346 |
| **750** | 0,0048 | 0,4572 | 0,0054 | 0,3143 | 0,0036 | 0,0715 |
| **1000** | 0,0076 | 1,2948 | 0,0082 | 0,5489 | 0,0056 | 0,1263 |
| **1250** | 0,0122 | 1,3445 | 0,0126 | 0,9811 | 0,0082 | 0,1964 |

## Algorytm Bellmana-Forda

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Original - 100% wypełnienia** | **Boost - 100% wypełnienia** | **Original - 50% wypełnienia** | **Boost - 50% wypełnienia** | **Original - mapa** | **Boost - mapa** |
| **250** | 0,0046 | 0,1904 | 0,0122 | 0,0683 | 0,0034 | 0,0092 |
| **500** | 0,0364 | 0,2823 | 0,0979 | 0,229 | 0,0182 | 0,0338 |
| **750** | 0,1189 | 0,5137 | 0,3427 | 0,39 | 0,0573 | 0,0827 |
| **1000** | 0,2791 | 1,2806 | 0,8038 | 0,5575 | 0,1485 | 0,1429 |
| **1250** | 0,5796 | 1,4021 | 1,6197 | 0,9017 | 0,3611 | 0,0775 |

## Algorytm Johnsona

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Original - 100% wypełnienia** | **Boost - 100% wypełnienia** | **Original - 50% wypełnienia** | **Boost - 50% wypełnienia** | **Original - mapa** | **Boost - mapa** |
| **100** | 0,0046 | 0,0452 | 0,0048 | 0,0216 | 0,0042 | 0,0066 |
| **200** | 0,023 | 0,4957 | 0,0368 | 0,1553 | 0,021 | 0,0452 |
| **300** | 0,0591 | 0,4324 | 0,0891 | 0,2544 | 0,0561 | 0,1207 |
| **400** | 0,1175 | 0,7787 | 0,1872 | 0,2679 | 0,1085 | 0,1433 |
| **500** | 0,2016 | 0,8676 | 0,3437 | 0,4716 | 0,174 | 0,2867 |

# Podsumowanie

Wyniki przeprowadzonych testów zajętości pamięci wykazują, że dla grafów gęstych lepiej stosować jako strukturę grafu macierz sąsiedztwa, natomiast dla rzadkich - listę sąsiedztwa