## Wstęp

Co to jest graf, bla bla bla

Oznaczenia V – liczba wierzchołków, E – liczba krawędzi

# Typy grafów

1. Gęsty – 100%
2. ??? – 50%
3. Mapa (rzadki) – sąsiedzi wybrani z 15 najbliższych wierzchołków

Ograniczyliśmy się do grafów nieskierowanych o nieujemnych krawędziach.

# 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;

}

# 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.

# Podsumowanie

Dla grafów gęstych lepiej macierz sąsiedztwa, dla rzadkich listę sąsiedztwa.