**Санкт-Петербургский национальный исследовательский университет**

**информационных технологий, механики и оптики**

**Кафедра информатики и прикладной математики**

Алгоритмы и структуры данных

Лабораторная работа №3

Вариант 3

Выполнил: Гхази Даниэль

Группа P3218

Преподаватель: Зинчик А.А.

2017 г.

**Текст задания**

Вариант 3

* А − алгоритм Прима,
* В − алгоритм Краскала;

1. Написать программу, реализующую алгоритм А и алгоритм В.
2. Написать программу, реализующую алгоритм А и алгоритм В, для проведения экспериментов, в которых можно выбирать:

* число n вершин и число m ребер графа,
* натуральные числа q и r, являющиеся соответственно нижней и верхней границей для весов ребер графа.

Выходом данной программы должно быть время работы ТА алгоритма А и время работы ТВ алгоритма В в секундах.

1. Провести эксперименты на основе следующих данных:
   1. n = 1, … ,104+1 с шагом 100, q = 1, r =106, количество ребер: а) m ≈ n2/10, б) m ≈ n2 (нарисовать графики функций TА(n) и ТВ(n) для обоих случаев);
   2. n = 101, … ,104+1 с шагом 100, q = 1, r  = 106, количество ребер: а) m ≈ 100⋅n, б) m ≈ 1000⋅n (нарисовать графики функций TА(n) и ТВ(n) для обоих случаев);
   3. n = 104+1, m = 0, … ,107 с шагом 105, q = 1, r = 106 (нарисовать графики функций TА(m) и ТВ(m) );
   4. n = 104+1, q = 1, r = 1, … ,200 с шагом 1, количество ребер: а) m ≈ n2, б) m ≈ 1000⋅n (нарисовать графики функций TА(r) и ТВ(r) для обоих случаев).
2. Сформулировать и обосновать вывод о том, в каких случаях целесообразно применять алгоритм А, а в каких − алгоритм В.

**Текст программы**

#include "../Headers/main.h"

int main()

{

// Keeps user-specified parameters for graph creation.

GraphParameters graphParameters;

// Pointers to lists of adjacent vertices.

std::vector<ElementOfAdjacencyList\*> ADJ;

// Keeps edges of MST.

std::vector<Edges> MST;

// Prevents usage of functions until graph was created.

bool graphWasCreated = false;

// Used as flag of exit, if user chooses corresponding menu item.

bool exit = false;

MenuItems menuItem = EXIT;

while (!exit)

{

system("cls");

menuItem = getMenuItem();

doChosenAction(menuItem, &exit, &graphWasCreated, ADJ, &graphParameters, MST);

if (!exit) system("PAUSE");

}

return 0;

}

MenuItems getMenuItem()

{

std::string menuItem = checkInputValidness(constants::menuPhrase, constants::menuRegex);

return (MenuItems)std::stoi(menuItem);

}

void doChosenAction(MenuItems menuItem, bool \*exit, bool \*graphWasCreated, std::vector<ElementOfAdjacencyList\*> &ADJ,

GraphParameters \*graphParameters, std::vector<Edges> &MST)

{

switch (menuItem)

{

case CREATE\_GRAPH:

\*graphParameters = getGraphParameters();

formGraph(graphWasCreated, ADJ, graphParameters, MST);

std::cout << "Graph successfully created!" << std::endl;

break;

case FIND\_MST:

if (\*graphWasCreated)

{

Timings timings;

findMST(ADJ, graphParameters, MST, &timings);

std::cout << "Minumum spanning tree found."<< std::endl;

std::cout << "Boruvka's algorithm timing: " << timings.timingOfAlgorithmA << " seconds" << std::endl <<

"Kruskal's algorithm timing: " << timings.timingOfAlgorithmB << " seconds" << std::endl;

// Should be set to false, because graph is broken after Kruskal's algorithm.

\*graphWasCreated = false;

}

else std::cout << "Graph was not created." << std::endl;

break;

case EXPERIMENT\_WITH\_CHANGING\_NUMBER\_OF\_EDGES:

{

bool success = experiment(ADJ, MST, graphWasCreated);

if (success) std::cout << "Experiment was successful. Results are in file." << std::endl;

break;

}

case EXIT:

\*exit = true;

break;

}

}

// Experiment 3.3

// 100 runs through graph with different number of edges.

// Saves data into excel file.

bool experiment(std::vector<ElementOfAdjacencyList\*> &ADJ, std::vector<Edges> &MST, bool \*graphWasCreated)

{

// Assigning graph parameters and forming graph.

GraphParameters graphParameters;

graphParameters.numberOfVertices = 10'000;

graphParameters.lowestPossibleWeight = 1;

graphParameters.highestPossibleWeight = 1'000'000;

// Preparing file for results.

std::ofstream timingsExcelFile;

timingsExcelFile.open("timings.csv");

if (!timingsExcelFile)

{

std::cerr << "Unable to open file." << std::endl;

return false;

}

// Running algorithms.

std::vector<double> timingsOfA(constants::numberOfTestsInExperiment);

std::vector<double> timingsOfB(constants::numberOfTestsInExperiment);

std::vector<long> numberOfEdges(constants::numberOfTestsInExperiment);

Timings timings;

doExperiment(ADJ, MST, timingsOfA, timingsOfB, &graphParameters, graphWasCreated, &timings, numberOfEdges);

// Putting results in file.

// ';' is a delimeter to put data in next cell.

for (int i = 0; i < timingsOfA.size(); i++)

{

timingsExcelFile << numberOfEdges[i] << ";";

timingsExcelFile << timingsOfA[i] << ";";

timingsExcelFile << timingsOfB[i] << std::endl;

}

return true;

}

// Running both algorithms.

void doExperiment(std::vector<ElementOfAdjacencyList\*> &ADJ, std::vector<Edges> &MST, std::vector<double> &timingsOfA,

std::vector<double> &timingsOfB, GraphParameters \*graphParameters, bool \*graphWasCreated, Timings \*timings,

std::vector<long> &numberOfEdges)

{

int indexOfNextFreeSlot = 0;

for (long m = 10'000; m <= 1'000'000; m += 10'000)

{

graphParameters->numberOfEdges = m;

formGraph(graphWasCreated, ADJ, graphParameters, MST);

findMST(ADJ, graphParameters, MST, timings);

timingsOfA[indexOfNextFreeSlot] = timings->timingOfAlgorithmA;

timingsOfB[indexOfNextFreeSlot] = timings->timingOfAlgorithmB;

numberOfEdges[indexOfNextFreeSlot++] = m;

}

}

// Getting user-specified parameters of graph.

GraphParameters getGraphParameters()

{

GraphParameters graphParameters;

//graphParameters.numberOfVertices = 10;

//graphParameters.numberOfEdges = 20;

//graphParameters.lowestPossibleWeight = 1;

//graphParameters.highestPossibleWeight = 10;

std::string graphParameterInput = "";

// Input number of vertices and edges.

bool validInput = false;

while (!validInput)

{

// Number of vertices.

graphParameterInput = checkInputValidness(constants::numberOfVerticesPhrase, constants::numberOfVerticesRegex);

graphParameters.numberOfVertices = std::stoi(graphParameterInput);

// Number of edges.

graphParameterInput = checkInputValidness(constants::numberOfEdgesPhrase, constants::numberOfEdgesRegex);

graphParameters.numberOfEdges = std::stoi(graphParameterInput);

validInput = checkOfMutualVerticesAndEdgesAppropriateness(&graphParameters);

}

// Input lowest and highest possible weights.

validInput = false;

while (!validInput)

{

// Heighest possible weight.

graphParameterInput = checkInputValidness(constants::heighestPossibleWeightPhrase, constants::heighestAndLowestPossibleWeightRegex);

graphParameters.highestPossibleWeight = std::stoi(graphParameterInput);

// Lowest possible weight.

graphParameterInput = checkInputValidness(constants::lowestPossibleWeightPhrase, constants::heighestAndLowestPossibleWeightRegex);

graphParameters.lowestPossibleWeight = std::stoi(graphParameterInput);

// Prevents lowest possible weight being higher than highest possible weight.

if (graphParameters.lowestPossibleWeight > graphParameters.highestPossibleWeight)

{

std::cout << "Lowest possible weight can not be higher than heighest possible input." << std::endl;

system("PAUSE");

}

else validInput = true;

}

return graphParameters;

}

// Gets user input and checks its validness. Returns only valid input.

std::string checkInputValidness(std::string whatToInputPhrase, std::regex regex)

{

std::cout << whatToInputPhrase;

std::string input = "";

bool validInput = false;

while (true)

{

getline(std::cin, input);

validInput = std::regex\_match(input, regex);

if (!validInput)

{

std::cout << "Invalid input." << std::endl << std::endl;

continue;

}

else return input;

}

}

// Prevents existens of vertices without adjacency.

bool checkOfMutualVerticesAndEdgesAppropriateness(GraphParameters \*graphParameters)

{

// Some of arithmetic progression: ((a1 + (n - 1)) / 2) \* (n - 1), where n is number of vertices.

long upperLimit = ((1 + graphParameters->numberOfVertices - 1) / 2) \* (graphParameters->numberOfVertices - 1);

long lowerLimit = graphParameters->numberOfVertices - 1;

if ((graphParameters->numberOfEdges < lowerLimit) ||

(graphParameters->numberOfEdges > upperLimit))

{

if (upperLimit > constants::maxNumberOfEdges) upperLimit = constants::maxNumberOfEdges;

std::cout << "Number of edges, when there are " << graphParameters->numberOfVertices << " vertices, should be within [" <<

lowerLimit << "; " << upperLimit << "]" << std::endl << std::endl;

system("PAUSE");

return false;

}

else return true;

}

**Результаты эксперимента**

**Вывод**

В результате выполнения лабораторной работы был создан генератор случайных графов и реализованы алгоритмы Борувки и Крускала, а также проведены эксперименты для сравнения двух алгоритмов с разным количеством рёбер графа.

В ходе эксперимента были получены неожиданные результаты. При малом количестве рёбер алгоритм Крускала работает в десятки раз быстрее. Это объясняется не самой лучше реализацией алгоритма Борувки.

В теории алгоритмы должны работать за схожее время , но этого не происходит, поскольку алгоритм Борувки был реализован не совсем так, как подразумевает теория, хоть и берет за основу идею этого алгоритма. Особенности реализации алгоритма Борувки таким образом позволили находить минимальное остовное дерево значительно быстрее, чем алгоритм Борувки, при количестве рёбер больше 3’600’000. Алгоритма Борувки работает быстрее в подавляющем количестве случаев при большом количестве ребер.

Реализация алгоритма Крускала вызвала меньше проблем, чем реализация алгоритма Борувки.