**Санкт-Петербургский национальный исследовательский университет**

**информационных технологий, механики и оптики**

**Кафедра информатики и прикладной математики**

Алгоритмы и структуры данных

Лабораторная работа №2

Вариант 3

Выполнил: Гхази Даниэль

Группа P3218

Преподаватель: Зинчик А.А.

2017 г.

**Текст задания**

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 "main.h"

int main()

{

// Keeps user-specified parameters for graph creation.

GraphParameters graphParameters;

// Pointers to lists of adjacent vertices.

std::vector<ElementOfAdjacencyList\*> ADJ;

// Represents final shortest distance from vertex 's' to vertex 'name[i]'.

std::vector<unsigned long long> dist;

// Keeps penultimate vertex in finished path from 's' to 'name[i]'.

std::vector<unsigned long> up;

// Prevents usage of path finding function (ldgDijkstraDHeap), until graph was created.

bool graphWasCreated = false;

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

bool exit = false;

MenuItems menuItem = EXIT;

Timings timings;

while (!exit)

{

system("cls");

menuItem = getMenuItem();

doChosenAction(menuItem, &exit, &graphWasCreated, ADJ, &graphParameters, dist, up, &timings);

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<unsigned long long> &dist, std::vector<unsigned long> &up,

Timings \*timings)

{

switch (menuItem)

{

case CREATE\_GRAPH:

\*graphParameters = getGraphParameters();

formGraph(graphWasCreated, ADJ, graphParameters);

std::cout << constants::graphCreationSuccess << std::endl;

break;

case FIND\_SHORTEST\_PATHS:

if (\*graphWasCreated)

{

ldgDijkstraDHeap(ADJ, dist, up, graphParameters);

std::cout << constants::pathFindingSuccess << std::endl;

}

else std::cout << constants::graphNotCreatedError << std::endl;

break;

case ALGORITHMS\_COMPARISON:

if (\*graphWasCreated) algorithmsComparison(ADJ, dist, up, graphParameters, timings);

else std::cout << constants::graphNotCreatedError << std::endl;

break;

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

experimentWithChangingNumberOfEdges(ADJ, dist, up, graphWasCreated);

break;

case EXIT:

\*exit = true;

break;

}

}

// Getting user-specified parameters of graph.

GraphParameters getGraphParameters()

{

GraphParameters graphParameters;

//graphParameters.numberOfVertices = 10'001;

//graphParameters.numberOfEdges = 1'000'000;

//graphParameters.lowestPossibleWeight = 1;

//graphParameters.highestPossibleWeight = 1'000'000;

//graphParameters.d = 4;

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 << constants::invalidWeightLimitsInputError << std::endl;

system("PAUSE");

}

else validInput = true;

}

// Input d of d-heap.

graphParameterInput = checkInputValidness(constants::dOfDHeapPhrase, constants::dOfDHeapRegex);

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

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 << constants::invalidInputError << std::endl << std::endl;

continue;

}

else return input;

}

}

// Prevents existens of vertices without adjacency.

bool checkOfMutualVerticesAndEdgesAppropriateness(GraphParameters \*graphParameters)

{

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

long lowerLimit = graphParameters->numberOfVertices - 1;

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

(graphParameters->numberOfEdges > upperLimit))

{

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;

}

// Comparing two algorithms, saving their timings.

void algorithmsComparison(std::vector<ElementOfAdjacencyList\*> &ADJ, std::vector<unsigned long long> &dist,

std::vector<unsigned long> &up, GraphParameters \*graphParameters, Timings \*timings)

{

clock\_t start;

clock\_t stop;

start = clock();

ldgDijkstraDHeap(ADJ, dist, up, graphParameters);

stop = clock();

timings->timingOfAlgorithmA = ((double)(stop - start) / CLK\_TCK);

(graphParameters->d)++;

start = clock();

ldgDijkstraDHeap(ADJ, dist, up, graphParameters);

stop = clock();

timings->timingOfAlgorithmB = ((double)(stop - start) / CLK\_TCK);

std::cout << "Algorithm A timing: " << timings->timingOfAlgorithmA << " seconds" << std::endl <<

"Algorithm B timing: " << timings->timingOfAlgorithmB << " seconds" <<std::endl;

(graphParameters->d)--;

}

// Experiment 3.3

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

// Saves data into excel file.

void experimentWithChangingNumberOfEdges(std::vector<ElementOfAdjacencyList\*> &ADJ, std::vector<unsigned long long> &dist,

std::vector<unsigned long> &up, bool \*graphWasCreated)

{

// Assigning graph parameters and forming graph.

GraphParameters graphParameters;

graphParameters.numberOfVertices = 10'001;

graphParameters.numberOfEdges = 1'000'000;

graphParameters.lowestPossibleWeight = 1;

graphParameters.highestPossibleWeight = 1'000'000;

graphParameters.d = 4;

// Preparing file for results.

std::ofstream timingsExcelFile;

timingsExcelFile.open("timings.csv");

if (!timingsExcelFile)

{

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

return;

}

// Running algorithms.

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

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

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

doExperiment(ADJ, dist, up, timingsOfA, timingsOfB, numberOfEdges, &graphParameters, graphWasCreated);

// 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;

}

}

// Running Dijkstra's algorithm.

void doExperiment(std::vector<ElementOfAdjacencyList\*> &ADJ, std::vector<unsigned long long> &dist,

std::vector<unsigned long> &up, std::vector<double> &timingsOfA, std::vector<double> &timingsOfB,

std::vector<long> &numberOfEdges, GraphParameters \*graphParameters, bool \*graphWasCreated)

{

clock\_t start;

clock\_t stop;

int indexOfNextFreeSlot = 0;

for (long m = 100'000; m <= 10'000'000; m += 100'000)

{

graphParameters->numberOfEdges = m;

formGraph(graphWasCreated, ADJ, graphParameters);

// Algorithm A.

start = clock();

ldgDijkstraDHeap(ADJ, dist, up, graphParameters);

stop = clock();

timingsOfA[indexOfNextFreeSlot] = ((double)(stop - start) / CLK\_TCK);

// Algorithm B

start = clock();

ldgDijkstraDHeap(ADJ, dist, up, graphParameters);

stop = clock();

timingsOfB[indexOfNextFreeSlot] = ((double)(stop - start) / CLK\_TCK);

numberOfEdges[indexOfNextFreeSlot++] = m;

}

}

**Результаты эксперимента**

**Вывод**

В результате выполнения лабораторной работы был создан генератор случайных графов и реализован алгоритм Дейстры, а также проведены эксперименты для сравнения двух алгоритмов с разными значениями параметра d у d-кучи.

В ходе эксперимента явного доминирования ни один из алгоритмов не показал. Они оба выполняются за соразмерное время. Вероятно, для выявления наиболее эффективного алгоритма необходимо протестировать алгоритм с другими значениями d.