**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**

* А − алгоритм Борувки,
* 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/findingMST.h"

void findMST(std::vector<ElementOfAdjacencyList\*> &ADJ, GraphParameters \*graphParameters, std::vector<Edges> &MST, Timings \*timings)

{

clock\_t start;

clock\_t stop;

start = clock();

boruvka(ADJ, graphParameters, MST);

stop = clock();

timings->timingOfAlgorithmA = ((double)(stop - start) / CLOCKS\_PER\_SEC);

start = clock();

kruskal(ADJ, graphParameters, MST);

stop = clock();

timings->timingOfAlgorithmB = ((double)(stop - start) / CLOCKS\_PER\_SEC);

}

// BORUVKA'S ALGORITHM

void boruvka(std::vector<ElementOfAdjacencyList\*> &ADJ, GraphParameters \*graphParameters, std::vector<Edges> &MST)

{

// Keeps vertices connected by edges of all subtrees.

std::vector<std::vector<Edges> > edgesSubsets(graphParameters->numberOfVertices + 1, std::vector<Edges>());

// Each element is a part of some subtree. -1 is a marker of removed tree.

std::vector<std::vector<int> > listsOfComponents(graphParameters->numberOfVertices + 1, std::vector<int>());

// Initializing first row with all vertices.

for (int column = 1; column < listsOfComponents.size(); column++)

{

listsOfComponents[column].push\_back(column);

}

// Continue executing algorithm until there are only 1 component left, which is MST.

int numberOfComponents = graphParameters->numberOfVertices;

// When entering 'findLowestWeightElementInComponent()' function first time, some extra initializations must be done.

bool firstSearch = true;

while (numberOfComponents > 1)

{

// Findig lowest element for each component and adding it to corresponding list.

for (int component = 1; component < listsOfComponents.size(); component++)

{

if (listsOfComponents[component][0] == -1) continue;

int elementWithLowestWeight = findLowestWeightElementInComponent(listsOfComponents, ADJ, component, &firstSearch, edgesSubsets);

listsOfComponents[component].push\_back(elementWithLowestWeight);

}

firstSearch = false;

// Finds same elements in different lists and puts them into one, keeping one subtree and marking as irrelevant another.

for (int component = 1; component < listsOfComponents.size(); component++)

{

if (numberOfComponents < 2) break;

if (listsOfComponents[component][0] == -1) continue;

// Used to start from only new elements.

int firstNewElementIndex = listsOfComponents[component].size() - 1;

findListHeaderAsNewElement(listsOfComponents, component, edgesSubsets, &numberOfComponents);

findNewAndAddedElementsFromOtherComponents(listsOfComponents, component, edgesSubsets, firstNewElementIndex, &numberOfComponents);

}

}

// Says which column keeps MST.

int columnWithMST = 0;

for (int component = 1; component < listsOfComponents.size(); component++)

{

if (listsOfComponents[component][0] != -1)

{

columnWithMST = component;

break;

}

}

cleaningRedundantEdgeOfMST(edgesSubsets, columnWithMST, MST);

}

int findLowestWeightElementInComponent(std::vector<std::vector<int> > &listsOfComponents, std::vector<ElementOfAdjacencyList\*> &ADJ,

int component, bool \*firstSearch, std::vector<std::vector<Edges> > &edgesSubsets)

{

ElementOfAdjacencyList \*elementWithLowestWeight = new ElementOfAdjacencyList;

elementWithLowestWeight->weight = INT32\_MAX;

Edges newEdge;

for (int elementOfComponent = 0; elementOfComponent < listsOfComponents[component].size(); elementOfComponent++)

{

int elementWhoseAdjacentEdgesAreToBeChecked = listsOfComponents[component][elementOfComponent];

ElementOfAdjacencyList \*currentElementWithLowestWeight = ADJ[elementWhoseAdjacentEdgesAreToBeChecked];

bool lowestWeightElementExists = false;

// Checks that there is at least one element, that is not in least already.

do

{

if (!(std::find(listsOfComponents[component].begin(), listsOfComponents[component].end(),

currentElementWithLowestWeight->name) != listsOfComponents[component].end()))

{

lowestWeightElementExists = true;

break;

}

currentElementWithLowestWeight = currentElementWithLowestWeight->next;

} while (currentElementWithLowestWeight != NULL);

ElementOfAdjacencyList \*currentElementToCompareTo = NULL;

if (currentElementWithLowestWeight != NULL) currentElementToCompareTo = currentElementWithLowestWeight->next;

// Search of lowest weight edge.

while (currentElementToCompareTo != NULL)

{

if (!(std::find(listsOfComponents[component].begin(), listsOfComponents[component].end(),

currentElementToCompareTo->name) != listsOfComponents[component].end()))

{

if (currentElementToCompareTo->weight < currentElementWithLowestWeight->weight)

{

currentElementWithLowestWeight = currentElementToCompareTo;

}

}

currentElementToCompareTo = currentElementToCompareTo->next;

}

// Checking element to have lowest weight across all comonent.

if ((lowestWeightElementExists) && (currentElementWithLowestWeight->weight < elementWithLowestWeight->weight))

{

elementWithLowestWeight = currentElementWithLowestWeight;

newEdge.firstVertex = elementWhoseAdjacentEdgesAreToBeChecked;

newEdge.secondVertex = elementWithLowestWeight->name;

}

}

// Used to supply header element with an edge.

if (\*firstSearch)

{

edgesSubsets[component].push\_back(newEdge);

}

edgesSubsets[component].push\_back(newEdge);

return elementWithLowestWeight->name;

}

// Searches if new added element to lists after current is same as header of current list.

// If so, adds all elements, that are not already in list, and sets -1 marker on it.

void findListHeaderAsNewElement(std::vector<std::vector<int> > &listsOfComponents, int workingComponent,

std::vector<std::vector<Edges> > &edgesSubsets, int \*numberOfComponents)

{

int workingElement = listsOfComponents[workingComponent][0];

// Searching for presense of current list header as new element of other components.

// Adding 1 is used to start from component following current.

for (int nextComponent = workingComponent + 1; nextComponent < listsOfComponents.size(); nextComponent++)

{

if (listsOfComponents[nextComponent][0] == -1) continue;

int lastElementOfComponent = listsOfComponents[nextComponent].back();

if (workingElement == lastElementOfComponent)

{

addElementFromAnotherComponent(listsOfComponents, nextComponent, workingComponent, edgesSubsets);

// Marking subtree as unexisting anymore.

listsOfComponents[nextComponent][0] = -1;

(\*numberOfComponents)--;

}

}

}

// Searches if subsequent lists have new element or newly added elements from other components in subsequent components.

// If so, adds all elements, that are not already in list, and sets -1 marker on it.

void findNewAndAddedElementsFromOtherComponents(std::vector<std::vector<int> > &listsOfComponents, int workingComponent,

std::vector<std::vector<Edges> > &edgesSubsets, int workingElementIndex, int \*numberOfComponents)

{

do

{

int workingElement = listsOfComponents[workingComponent][workingElementIndex];

// Searching for presense of new elements in subsequent components.

// Adding 1 is used to start from component following current.

for (int nextComponent = workingComponent + 1; nextComponent < listsOfComponents.size(); nextComponent++)

{

if (listsOfComponents[nextComponent][0] == -1) continue;

// If element found - adds all elements, that are not already in list.

if (std::find(listsOfComponents[nextComponent].begin(), listsOfComponents[nextComponent].end(),

workingElement) != listsOfComponents[nextComponent].end())

{

addElementFromAnotherComponent(listsOfComponents, nextComponent, workingComponent, edgesSubsets);

// Marking subtree as unexisting anymore.

listsOfComponents[nextComponent][0] = -1;

(\*numberOfComponents)--;

}

}

} while (listsOfComponents[workingComponent].back() != listsOfComponents[workingComponent][workingElementIndex++]);

}

// Adding each element that is not at current working list and transfering edges.

void addElementFromAnotherComponent(std::vector<std::vector<int> > &listsOfComponents, int nextComponent,

int workingComponent, std::vector<std::vector<Edges> > &edgesSubsets)

{

for (int elementOfComponent = 0; elementOfComponent < listsOfComponents[nextComponent].size(); elementOfComponent++)

{

// If element not found in current working list - add it.

if (!(std::find(listsOfComponents[workingComponent].begin(), listsOfComponents[workingComponent].end(),

listsOfComponents[nextComponent][elementOfComponent]) != listsOfComponents[workingComponent].end()))

{

listsOfComponents[workingComponent].push\_back(listsOfComponents[nextComponent][elementOfComponent]);

}

}

// Transfering all edges to new list.

for (int edgeOfSubset = 0; edgeOfSubset < edgesSubsets[nextComponent].size(); edgeOfSubset++)

{

edgesSubsets[workingComponent].push\_back(edgesSubsets[nextComponent][edgeOfSubset]);

}

}

void cleaningRedundantEdgeOfMST(std::vector<std::vector<Edges> > &edgesSubsets, int columnWithMST, std::vector<Edges> &MST)

{

MST.push\_back(edgesSubsets[columnWithMST][0]);

// Going through each accumulated edge.

for (int edge = 1; edge < edgesSubsets[columnWithMST].size(); edge++)

{

int numberOfCopiesNotFound = 0;

// Going through each edge already in MST vector.

for (int edgeInMST = 0; edgeInMST < MST.size(); edgeInMST++)

{

// If no direct copies.

if (!((MST[edgeInMST].firstVertex == edgesSubsets[columnWithMST][edge].firstVertex) &&

(MST[edgeInMST].secondVertex == edgesSubsets[columnWithMST][edge].secondVertex)))

{

// If no mirror copies.

if (!((MST[edgeInMST].firstVertex == edgesSubsets[columnWithMST][edge].secondVertex) &&

(MST[edgeInMST].secondVertex == edgesSubsets[columnWithMST][edge].firstVertex)))

{

numberOfCopiesNotFound++;

}

}

}

if (numberOfCopiesNotFound == MST.size()) MST.push\_back(edgesSubsets[columnWithMST][edge]);

}

}

// KRUSKAL'S ALGORITHM

void kruskal(std::vector<ElementOfAdjacencyList\*> &ADJ, GraphParameters \*graphParameters, std::vector<Edges> &MST)

{

std::priority\_queue <Edges, std::vector<Edges>, myComparator > sortedEdges;

createPriorityQueueOfEdges(sortedEdges, ADJ);

formingMST(sortedEdges, graphParameters, MST);

}

// Putting elements into ascending priority queue base on minimum heap.

void createPriorityQueueOfEdges(std::priority\_queue <Edges, std::vector<Edges>, myComparator > &sortedEdges,

std::vector<ElementOfAdjacencyList\*> &ADJ)

{

for (int list = 1; list < ADJ.size(); list++)

{

ElementOfAdjacencyList \*element = &ADJ[list][0];

Edges edgeToAdd;

while (element != NULL)

{

// If element = removed reflection element.

if (element->name == -1)

{

element = element->next;

continue;

}

// Removing reflection element.

ElementOfAdjacencyList \*elementToDelete = &ADJ[element->name][0];

while (elementToDelete != NULL)

{

if (elementToDelete->name == list)

{

elementToDelete->name = -1;

break;

}

elementToDelete = elementToDelete->next;

}

// Adding edge

edgeToAdd.firstVertex = list;

edgeToAdd.secondVertex = element->name;

edgeToAdd.weight = element->weight;

sortedEdges.push(edgeToAdd);

element = element->next;

}

}

}

// Picking the smallest edge. Checking if it forms a cycle with the spanning tree

// formed so far. If cycle is not formed, include this edge. Else, discard it.

void formingMST(std::priority\_queue <Edges, std::vector<Edges>, myComparator > sortedEdges, GraphParameters \*graphParameters,

std::vector<Edges> &MST)

{

// Keeps disjoint sets of edges, so we can prevent cycling in graph.

std::vector<int> setsOfEdges(graphParameters->numberOfVertices + 1, -1);

// Adding edges to MST and removing those, which make cycles.

// Number of edges of MST must be one less than the number of vertices.

for (int i = 0; i < graphParameters->numberOfVertices; i++)

{

// Taking minimum edge.

Edges currentEdge = sortedEdges.top();

sortedEdges.pop();

// Checking if cycle will be created.

// Finding first vertex set.

int setLeaderOfFirstVertex = currentEdge.firstVertex;

while (setsOfEdges[setLeaderOfFirstVertex] != -1)

{

setLeaderOfFirstVertex = setsOfEdges[setLeaderOfFirstVertex];

}

// Finding second vertex set.

int setLeaderOfSecondVertex = currentEdge.secondVertex;

while (setsOfEdges[setLeaderOfSecondVertex] != -1)

{

setLeaderOfSecondVertex = setsOfEdges[setLeaderOfSecondVertex];

}

// If elements are in same set - addition of this edge would result in cycle.

// If they are in different sets - add it to MST and unite with other set.

if (setLeaderOfFirstVertex == setLeaderOfSecondVertex) continue;

else

{

MST.push\_back(currentEdge);

setsOfEdges[setLeaderOfFirstVertex] = setLeaderOfSecondVertex;

}

}

}

**Результаты эксперимента**

**Вывод**

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

В ходе эксперимента были полученынеожиданные результаты. При малом количестве рёбер алгоритмКрускала 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.