**ИТМО Кафедра Информатики и прикладной математики**

# Отчет по лабораторной работе №3 «Поиск минимального остовного дерева» Вариант 7

**Выполнил: студент группы P3217**
**Плюхин Дмитрий**
**Преподаватель: Зинчик А. А.**

**2017 год**

## 1. Задание

1. Написать программу, реализующую алгоритм Прима и алгоритм Борувки.
2. Написать программу, реализующую алгоритмы, 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 $T_A$ алгоритма Прима и время работы $T_B$ алгоритма Борувки, в секундах.

3. Провести эксперименты на основе следующих данных:

- $n = 10^4+1$
- $m = 0, \dots, 10^7$ с шагом $10^5$
- $q = 1$
- $r = 10^6$
- нарисовать графики функций $T_A(m)$ и $T_B(m)$;

4. Сформулировать и обосновать вывод о том, в каких случаях целесообразно применять алгоритм A, а в каких алгоритм B.

## 2. Исходный код

**Алгоритм Прима**

```
unsigned* mst_prim(unsigned numberOfVertexes, unsigned numberOfEdges, GraphAdjRecordPointer
graphAdj, unsigned maxWeight, EdgePointer edges){
  unsigned seed = 1;
  unsigned* minimalIncidentEdgeWeight =
(unsigned*)malloc((numberOfVertexes+1)*sizeof(unsigned));
  unsigned* edgesMST = (unsigned*)malloc((numberOfVertexes+1)*sizeof(unsigned));
  unsigned* anotherVertexOfMinimalIncidentEdgeWeight =
(unsigned*)malloc((numberOfVertexes+1)*sizeof(unsigned));
  unsigned* inMST = (unsigned*)malloc((numberOfVertexes+1)*sizeof(unsigned));
  unsigned edgesMSTIndex = 0;
  for (int i = 1; i <= numberOfVertexes; i++){
    inMST[i] = 0;
    minimalIncidentEdgeWeight[i] = maxWeight + 1;
    anotherVertexOfMinimalIncidentEdgeWeight[i] = 0;
  }
  inMST[seed] = 1;
  GraphAdjRecordPointer currentAdjRecord = &graphAdj[seed];
  while(currentAdjRecord->next != NULL){
    currentAdjRecord = currentAdjRecord->next;
    minimalIncidentEdgeWeight[currentAdjRecord->name] = currentAdjRecord->weight;
    anotherVertexOfMinimalIncidentEdgeWeight[currentAdjRecord->name] = seed;
  }
  unsigned nextVertex;
  unsigned preVertex;
  unsigned vertexNum;
  while (edgesMSTIndex < numberOfVertexes - 1){
    nextVertex = findNextVertex(minimalIncidentEdgeWeight, inMST, numberOfVertexes, maxWeight);
    if (minimalIncidentEdgeWeight[nextVertex] > maxWeight){
      cout << "Graph is not connected" << endl;
      return edgesMST;
    }
    preVertex = anotherVertexOfMinimalIncidentEdgeWeight[nextVertex];
    inMST[nextVertex] = 1;
    edgesMSTIndex ++;
    vertexNum = getEdgeNum(nextVertex, preVertex, edges, numberOfEdges);
    edgesMST[edgesMSTIndex] = vertexNum;
    GraphAdjRecordPointer currentAdjRecord = &graphAdj[nextVertex];
    while(currentAdjRecord->next != NULL){
      currentAdjRecord = currentAdjRecord->next;
      if (inMST[currentAdjRecord->name] == 0){
        if (minimalIncidentEdgeWeight[currentAdjRecord->name] > currentAdjRecord->weight){
          minimalIncidentEdgeWeight[currentAdjRecord->name] = currentAdjRecord->weight;
          anotherVertexOfMinimalIncidentEdgeWeight[currentAdjRecord->name] = nextVertex;
        }
      }
    }
  }
```

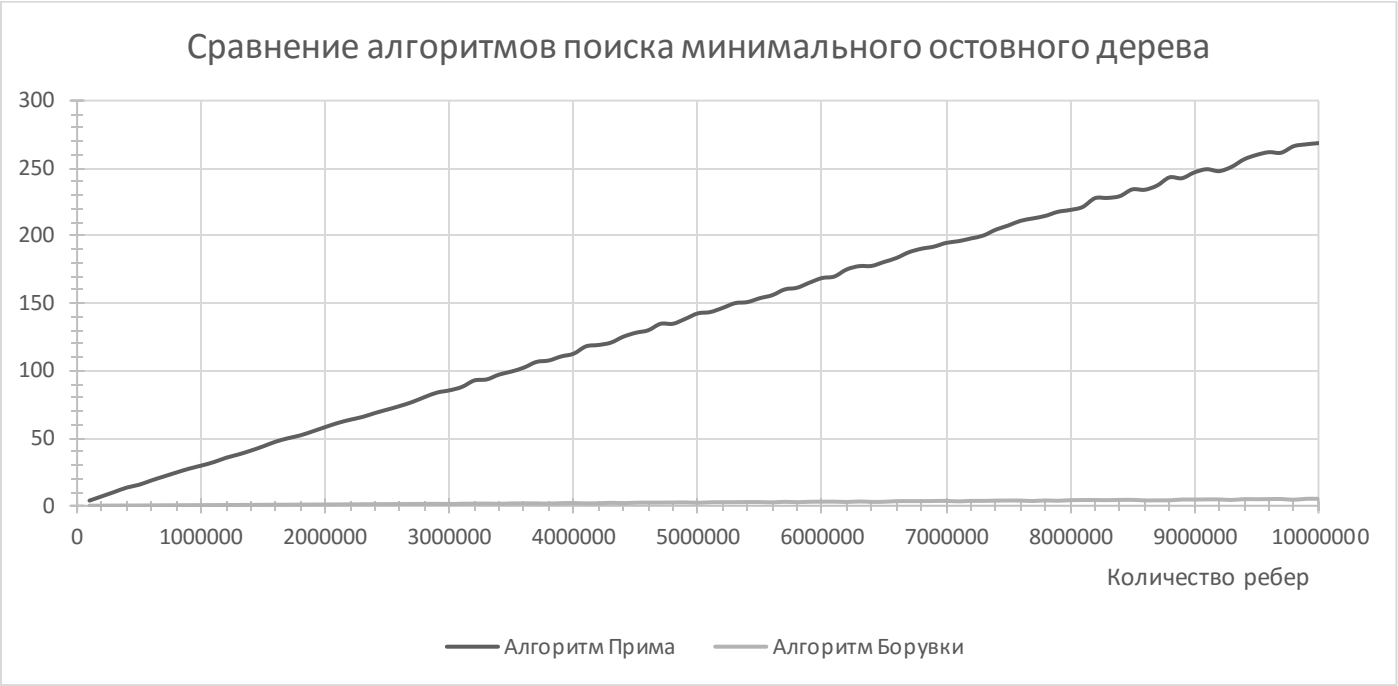
```
  return edgesMST;
}
unsigned findNextVertex(unsigned* minimalIncidentEdgeWeight, unsigned* inMST, unsigned
numberOfVertexes, unsigned maxWeight){
  unsigned minimalWeight = maxWeight + 1;
  unsigned indexOfMinimal = 0;
  for (int i = 1; i <= numberOfVertexes; i++){
    if ((minimalIncidentEdgeWeight[i] < minimalWeight) && (inMST[i] == 0)){
      minimalWeight = minimalIncidentEdgeWeight[i];
      indexOfMinimal = i;
    }
  }
  return indexOfMinimal;
}
```

**Алгоритм Борувки**

```
unsigned* mst_boruvka(unsigned numberOfEdges, unsigned numberOfVertexes, EdgePointer edges,
unsigned* weights){
  unsigned edgesMSTIndex = 0;
  unsigned* minimalIncidentEdge = (unsigned*)malloc((numberOfVertexes+1)*sizeof(unsigned));
  DisjointSetObjectPointer* sets =
(DisjointSetObjectPointer*)malloc((numberOfVertexes+1)*sizeof(DisjointSetObjectPointer));
  unsigned* edgesMST = (unsigned*)malloc((numberOfVertexes+1)*sizeof(unsigned));
  EdgeExPointer edgeExs = (EdgeExPointer)malloc((numberOfEdges+1)*sizeof(EdgeEx));
  for (int i = 1; i <= numberOfVertexes; i++){
    minimalIncidentEdge[i] = 0;
    sets[i] = makeSet(i);
  }
  for (int i = 1; i <= numberOfEdges; i++){
    edgeExs[i].anode = sets[edges[i].anode];
    edgeExs[i].bnode = sets[edges[i].bnode];
  }
  DisjointSetObjectPointer anode;
  DisjointSetObjectPointer bnode;
  unsigned firstSetName;
  unsigned secondSetName;
  while (findMinimalIncidentEdge(minimalIncidentEdge, sets, numberOfVertexes, numberOfEdges,
edgeExs, weights) || (edgesMSTIndex < numberOfVertexes - 1)){
    for (int i = 1; i <= numberOfVertexes; i++){
      if (minimalIncidentEdge[i] > 0){
        anode = edgeExs[minimalIncidentEdge[i]].anode;
        bnode = edgeExs[minimalIncidentEdge[i]].bnode;
        firstSetName = findSet(anode)->value;
        secondSetName = findSet(bnode)->value;
        if (firstSetName != secondSetName){
          edgesMSTIndex++;
          edgesMST[edgesMSTIndex] = minimalIncidentEdge[i];
          makeUnion(findSet(anode),findSet(bnode));
        }// if it is not one set
        minimalIncidentEdge[i] = 0;
      }// if there is minimal edge incident the vertex
    }// for each vertex
  }// while find new minimal edges
  return edgesMST;
}
bool findMinimalIncidentEdge(unsigned* minimalIncidentEdge, DisjointSetObjectPointer* sets,
unsigned numberOfVertexes, unsigned numberOfEdges, EdgeExPointer edgeExs, unsigned* weights){
  bool result = false;
  DisjointSetObjectPointer anode;
  DisjointSetObjectPointer bnode;
  unsigned firstSetName;
  unsigned secondSetName;
  for (int i = 1; i <= numberOfEdges; i++){
    anode = edgeExs[i].anode;
    bnode = edgeExs[i].bnode;
    firstSetName = findSet(anode)->value;
    secondSetName = findSet(bnode)->value;
    if (firstSetName != secondSetName){
      if ((minimalIncidentEdge[firstSetName] == 0) || (weights[i] <
weights[minimalIncidentEdge[firstSetName]])){
        minimalIncidentEdge[firstSetName] = i;
        result = true;
      }
      if ((minimalIncidentEdge[secondSetName] == 0) || (weights[i] <
weights[minimalIncidentEdge[secondSetName]])){
        minimalIncidentEdge[secondSetName] = i;
        result = true;
      }
    }
```

```
  }
  return result;
}
```

## 3. Результаты экспериментов




## 4. Вывод

Итак, касательно рассмотренных алгоритмов поиска минимального остовного дерева можно сделать вывод, что алгоритм Борувки работает гораздо эффективнее алгоритма Прима, который к тому же оказывается неэффективным в плане используемой памяти – для его работы требуется представление графа в виде списка смежности, в то время как алгоритм Борувки получает на вход два массива, общий размер которых выигрывает при сравнении со структурой данных, используемой алгоритмом Прима. Отсюда можно сделать вывод, что применение алгоритма Прима «в чистом виде» неэффективно и требует использования, с одной стороны, эффективного представления графа (например, в виде матрицы смежности), с другой – подходов, увеличивающих скорость работы алгоритма (например, использование очереди с приоритетами при поиске следующей вершины). При всех возможных оптимизациях асимптотически алгоритм Прима будет вести себя так же, как и алгоритм Борувки, однако использование алгоритма Прима будет предпочтительным в том случае, если преобразование данных в специфичную форму для алгоритма Борувки требует больших затрат (или, наоборот, граф изначально представлен как список ребер и преобразование его в матрицу смежности требует дополнительных ресурсов – в этом случае более рационально использование алгоритма Борувки).