Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Мегафакультет компьютерных технологий и управления

Кафедра информатики и прикладной математики

Алгоритмы и структуры данных

Лабораторная работа №3

«Нахождение минимального остовного дерева»

Группа: P3218

Студент: Петкевич Константин

Преподаватель: Зинчик А. А.

2017г

Задание

1. Написать программу, реализующую алгоритм Прима и алгоритм Краскала.
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 Выходом данной программы должно быть время работы Ta алгоритма А и время работы Tb алгоритма В.

3. Провести эксперименты и нарисовать графики функций Ta(n) и Tb(n) для обоих случаев на основе следующих данных: n=1, … m 10^4+1 с шагом 100 ,q=1, r=10^6, кол-во рёбер: m=n^2/10 и m=n^2
4. Сформулировать и обосновать вывод о том, в каких случаях целесообразно применять алгоритм А и Б

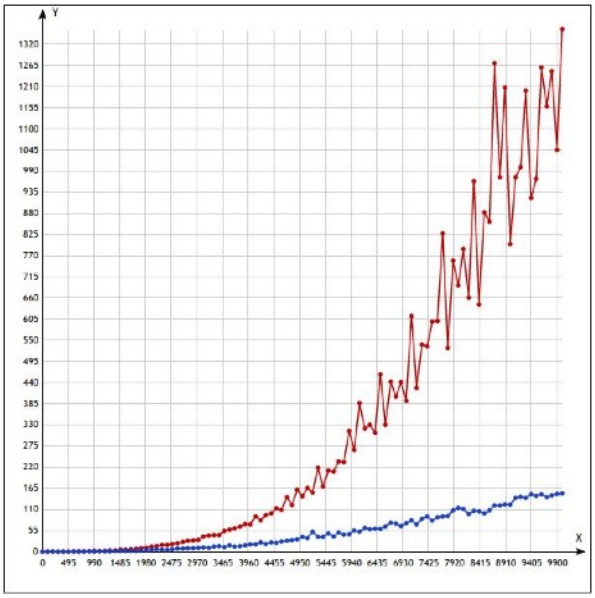
Листинг кода

```
// Lab 3
bool contains(vector<Vertex*> vertices, size_t id) {
   for (vector<Vertex*>::iterator it = vertices.begin(); it != vertices.end(); it++)
      if ((*it) != nullptr && (*it)->id == id)
         return true;

   return false;
}
Graph* Prim(Graph* graph) {
   GraphBuilder builder(graph->getVerticesAmount());
   vector<Vertex*> vertices = graph->getAllVertices();
   multiset<Edge> allAvailableEdges;

   // preparations
   int startId = 0; //rand() % vertices.size();
   builder.addVertex(startId);
   for(list<Edge*>::iterator startEdgesIt = vertices[startId]->neighborhood.begin(); startEdgesIt != vertices[startId]-
>neighborhood.end(); startEdgesIt++ )
      allAvailableEdges.insert(**startEdgesIt);

   // main loop
   while(builder.getResult()->getVerticesAmount() != graph->getVerticesAmount() && !allAvailableEdges.empty()) {
      vector<Vertex*> currentVertices = builder.getResult()->getAllVertices();
      multiset<Edge>::iterator lightweight = allAvailableEdges.begin();

      if (!contains(currentVertices, lightweight->destination->id)) {
         builder.addVertex(lightweight->destination->id);
         for(list<Edge*>::iterator neighboursIt = lightweight->destination->neighborhood.begin(); neighboursIt !=
lightweight->destination->neighborhood.end(); neighboursIt++ ) {
            if (!(*neighboursIt)->isOpposite(*lightweight))
               allAvailableEdges.insert(**neighboursIt);
         }

         builder.addUndirectedEdge(lightweight->source->id, lightweight->destination->id, lightweight->weight);
      }

      allAvailableEdges.erase(lightweight);
   }
```

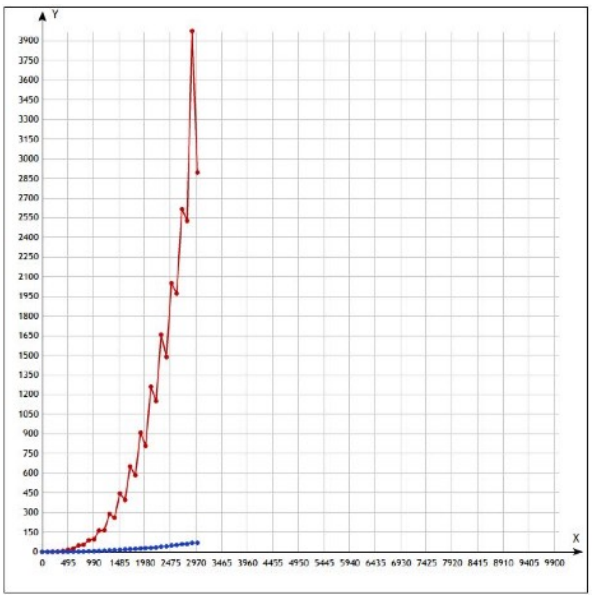
```
    return builder.getResult();
}
Graph* Kruskal(Graph* graph) {
    GraphBuilder builder(graph->getVerticesAmount());

    GraphBuilder preparator(graph);
    preparator.removeLoops();
    preparator.removeDoubles();

    multiset<Edge> allEdges;
    vector<Vertex*> vertices = graph->getAllVertices();
    for(vector<Vertex*>::iterator vIt = vertices.begin(); vIt != vertices.end(); vIt++)
        for(list<Edge*>::iterator eIt = (*vIt)->neighborhood.begin(); eIt != (*vIt)->neighborhood.end(); eIt++)
            allEdges.insert(**eIt);

    // Now add vertices and edge between them from left to right of edge list if they add new vertices
    while(builder.getResult()->getVerticesAmount() != graph->getVerticesAmount() && !allEdges.empty()) {
        Graph* constructedGraph = builder.getResult();
        multiset<Edge>::iterator min = allEdges.begin();

        if (!constructedGraph->haveCycle(*min)) {
            Graph* currentGraph = builder.getResult();

            if (currentGraph->getVertex(min->source->id) == nullptr)
                builder.addVertex(min->source->id);
            if (currentGraph->getVertex(min->destination->id) == nullptr)
                builder.addVertex(min->destination->id);

            builder.addUndirectedEdge(min->source->id, min->destination->id, min->weight);
        }

        // remove his opposite too;
        allEdges.erase(min);
        for(multiset<Edge>::iterator it = allEdges.begin(); it != allEdges.end(); it++)
            if (it->isOpposite(*min)) {
                allEdges.erase(it);
                break;
            }
    }

    return builder.getResult();
}
```

Эксперименты:

n = m^2/10



n = m^2



Вывод

Алгоритм Краскала для моей модели графа многим сложнее реализовать из-за поиска циклов при добавлении очередного ребра. Его суммарная сложность = Удалить все петли O(e) + Удалить все повторяющиеся рёбра O(e) + Добавить все рёбра в список всех рёбер = O(e) + Для каждого ребра O(e) сделать следующее: добавить к MST, если нет цикла O(DFS) = O(e) + Удалить соответсвующее неориентированное ребро = O(1).

В сумме сложность : O (t) = O (3 e) + O (e * (e + 1)) = O (e ^ 2) = **O (n ^ 4)**

С Примом всё гораздо проще, т.к. он подходит к моей реализации графа, и, как видно, он выдаёт свои O(e*log(v))