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

Кафедра информатики и прикладной математики

Алгоритмы и структуры данных

Лабораторная работа №3

“Нахождение минимального оставного дерева”

Вариант 3

Проверил: **Зинчик А.А.**

Старался: **Шкаруба Н.Е.**

Группа **P3218**

2016г

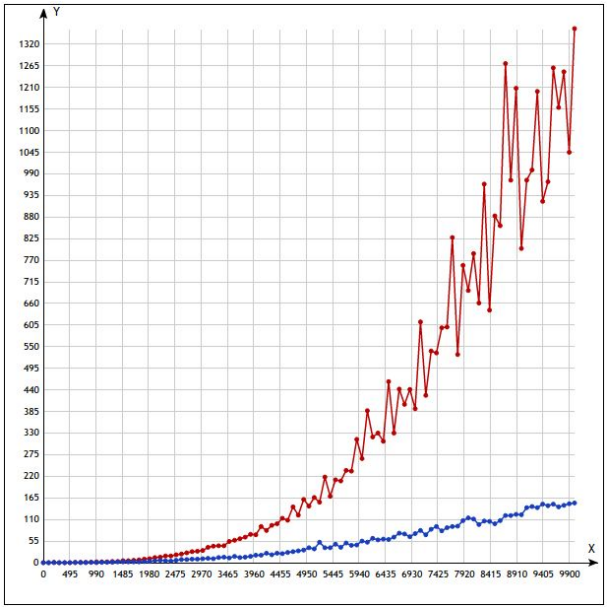
Требования:

1. Написать программу, реализующую алгоритм Прима и алгоритм Краскала.
2. Написать программу, реализующую алгоритм A и алгоритм B, 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

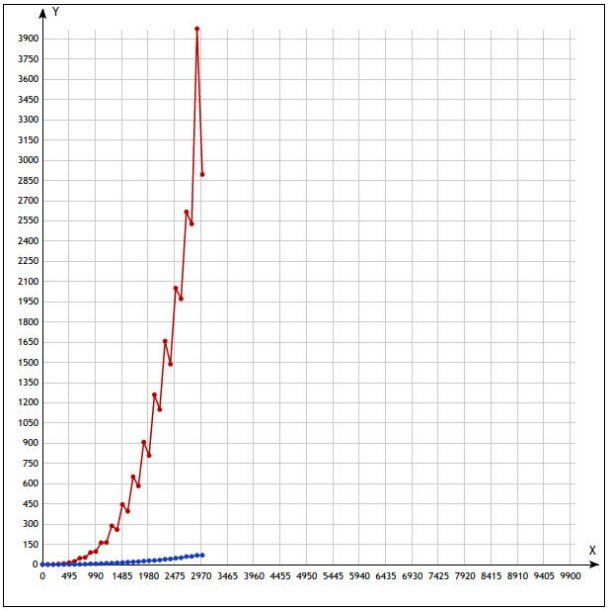
Выходом данной программы должно быть время работы Ta алгоритма A и время работы Tb алгоритма B.
3. Провести эксперименты и нарисовать графики функций Ta(n) и Tb(n) для обоих случаев на основе следующих данных: n = 1, … m 10^4+1 с шагом 100, q = 1, r = 10^6, кол-во рёбер: m = n^2/10 и m = n^2
4. Сформулировать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

Эксперименты:

m = n^2/10



m = n^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 вторящиеся 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 всё это выдаёт сложность: O(t) = O(3e) + 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))

Код: Почему работает медленно: ООП

```
struct Vertex {
   Vertex(size_t id) {}

   const size_t id;
   list<Edge*> neighborhood;
};
```

```
struct Edge {
   Edge(Vertex* v1, Vertex* v2, int weight);

   Vertex* source;
   Vertex* destination;
   int weight;
   bool isDirected;
};
```

```
class Graph {
   // Graph - законченная структура данных, в нём определены операции поиска вершины
   // GraphBuilder - класс, изменяющий граф, в нём определены операции удаления, добавления
   friend class GraphBuilder;

Public:
   vector<Vertex*> getAllVertices();
   Vertex* getVertex(size_t id);
   Vertex* getRandomVertex();

   size_t getVerticesAmount();
   size_t getEdgesAmount();

private:
   // Приватный конструктор, доступный лишь для GraphBuilder
   Graph(size_t verticesAmount);

   vector<Vertex*> vertices;
   size_t verticesAmount;
   size_t edgesAmount;
};
```

```
class GraphBuilder {
public:
   GraphBuilder(size_t graphSize);   // Начать конструировать новый граф
   GraphBuilder(Graph* graph);       // Изменить существующий граф

   void generateRandomDirectedGraph(size_t verticesCount, size_t edgesCount, int minEdgeWeight,
int maxEdgeWeight);
   void generateRandomUndirectedGraph(size_t verticesCount, size_t edgesCount, int
minEdgeWeight, int maxEdgeWeight);

   // Сгенерировать тестовые графы, на которых я теоретически знаю, как работают алгоритмы
   void generateDijkstraTestGraph();
   void generateBellmanFordTestGraph();
   void generatePrimTestGraph();
   void generateKruskalTestGraph();

   void addVertex(size_t id);
   void addEdge(size_t sourceId, size_t destinationId, int weight);
   void addUndirectedEdge(size_t firstId, size_t secondId, int weight);

   void removeLoops();
   void removeDoubles();

   Graph* getResult();

private:
   Graph* constructed;
};
```