Санкт-Петербургский Государственный Университет
Факультет Прикладной Математики и Процессов Управления

Лабораторная работа по курсу «Алгоритмы и анализ сложности»

Эмпирический анализ алгоритма Каргера

Студент: Герасимов Владислав Юрьевич
Преподаватель: Никифоров Константин Аркадьевич
Группа: 16.Б13-пу

Санкт-Петербург
2018

Содержание

1 Краткое описание алгоритма

Алгоритм Каргера (англ. Karger's algorithm) [1] — вероятностный алгоритм нахождения минимального разреза связного графа (т.е. минимального числа рёбер, которое необходимо удалить для нарушения связности).

Был разработан Девидом Каргером (David Karger) и опубликован в 1993 году. После дополнительных оптимизаций Каргера и Штейна (Karger, Stein, 1996) алгоритм стал одним из самых эффективных методов решения задачи о минимальном разрезе.

Примеры областей применения:

- Определение коммуникаций и разделения различных социальных групп.
- Определение уязвимостей у потенциального противника с целью разрушения его транспортной сети.
- Сегментация изображений.

Основной операцией алгоритма Каргера является одна из форм стягивания ребра. Для выполнения этой операции на произвольном ребре $e = (u, v)$ происходит объединение вершин графа u и v в одну uv. Если удаляется вершина v, то каждое ребро вида (v, x) заменяется на ребро вида (u, x). Петли удаляются и после операции граф не содержит петель.

Алгоритм представляет собой равновероятный выбор случайного имеющегося ребра и объединение вершин согласно описанной операции. Результатом работы алгоритма является количество ребёр в разрезе графа. Этот разрез может быть не минимальным, но вероятность того, что этот разрез минимальный существенно больше, чем для случайно выбранного разреза.

Псевдокод алгоритма:

```
0. getCut():
1. повторить n - 2 раза
2.    выбрать случайно ребро e
3.    стянуть ребро e
4. результат ← число рёбер между двумя последними вершинами
```

2 Математический анализ алгоритма

Алгоритм правильно определяет минимальный разрез с вероятностью:
$P \geq \frac{2}{n(n-1)} \geq \frac{2}{n^2}$ [2]

Проведя $n^2 \cdot ln(n)$ независимых запусков с обновлением минимума после каждой итерации, мы получим вероятность получения неверного ответа $\leq \frac{1}{n^2}$, что гарантирует практическую корректность результата работы алгоритма даже при небольших значениях n.

Сложность алгоритма вычисления минимального разреза зависит от конкретной реализации. Известны имплементации, работающие за $O(n^2)$, $O(m \cdot ln(m))$, $O(m)$ время. В разделе «Реализация» будет приведена реализация алгоритма на языке Java, работающая за $O(n^2)$, использующая список смежности для работы с графом. Основной операцией при таком выборе алгоритма является операция умножения целых чисел, а общая теоретическая оценка сложности равна $O(n^4 \cdot ln(n))$.

3 Описание эксперимента

Входными данными описываемого алгоритма является связный граф. Теоретическое время работы алгоритма зависит только от размера входных данных - количества вершин. Измерения трудоёмкости алгоритма будут производиться в диапазоне $n = [8, 128]$ количества вершин с шагом 8. Единица измерения - миллисекунда.

Программа эксперимента написана на языке Java с использованием библиотеки JMH [3] и будет выполнена в вычислительной среде со следующими характеристиками:

- Windows 10.0.17134 Pro
- Intel64 Family 6 Model 42 Stepping 7 GenuineIntel 1600 MHz
- java.version=1.8.0.144
- java.vm.name=Java HotSpot(TM) 64-Bit Server VM
- java.vm.specification.name=Java Virtual Machine Specification
- java.vm.specification.vendor=Oracle Corporation
- java.vm.specification.version=1.8
- java.vm.vendor=Oracle Corporation
- java.vm.version=25.144-b01
- -Xms2G -Xmx2G
- 1 thread

Для очередной итерации n - аргумента количества вершин - программа создает 2 независимых профиля JVM (Forks), проводит 10 тестов с 10 различными графами, полученными с помощью генератора, описанном в разделе «Генерация данных». Первые 3 теста являются «разогревочными», их результаты не учитываются. В качестве результата берется средняя из полученной выборки $(10 - 3) \cdot 2 = 14$ размера. Время генерации графов не учитывается.

4 Реализация

Далее представлена реализация алгоритма на языке Java 1.8. Помимо встроенных средств языка была использована библиотека fastutil [4], расширяющая стандартный набор структур данных.

```
public class MinCut {
    public static int minCut(Graph graph) {
        int ans = Integer.MAX_VALUE;
        int iterations = countIterations(graph);

        for (int i = 0; i < iterations; i++) {
            int currCut = getCut(graph);

            if (currCut < ans) {
                ans = currCut;
            }
        }

        return ans;
    }

    private static int countIterations(Graph graph) {
        int vertices = graph.vertexNumber();
        return vertices * (vertices - 1) * (int) Math.log(vertices);
    }

    private static int getCut(Graph graph) {
        Graph g = new Graph(graph);

        while (g.vertexNumber() > 2) {
            g.contract();
        }

        return g.edgeNumber();
    }
}



public class Graph {
    private static final Random random = new Random();

    private Int2ObjectMap<VertexMeta> adjList;
    private int edges;

    public Graph(Int2ObjectMap<VertexMeta> adjList) {
        this.adjList = adjList;

        for (VertexMeta v : this.adjList.values()) {
            this.edges += v.edges;
        }

        this.edges /= 2;
    }

    public Graph(Graph o) {
        this.edges = o.edges;
        this.adjList = new Int2ObjectOpenHashMap<>(o.adjList.size());

        for (Int2ObjectMap.Entry<VertexMeta> e : o.adjList.int2ObjectEntrySet()) {
            this.adjList.put(e.getIntKey(), new VertexMeta(e.getValue()));
        }
    }

    public int vertexNumber() {
        return adjList.size();
    }

    public int edgeNumber() {
        return edges;
    }

    public void contract() {
        IntPair removed = pickRandomEdge();

        int first = removed.getF();
        int second = removed.getS();
```

```
        VertexMeta cToF = adjList.get(first);
        VertexMeta cToS = adjList.remove(second);

        for (IntPair v : cToS.adjVertices) {
            int edges = v.getS();

            if (edges != 0 && v.getF() != first) {
                VertexMeta cToV = adjList.get(v.getF());

                cToF.addEdges(v.getF(), edges);
                cToV.addEdges(first, edges);
                cToV.addEdges(second, -edges);
            }
        }

        edges -= cToF.getEdges(second);
        cToF.setEdges(second, 0);
    }

    private IntPair pickRandomEdge() {
        int index = random.nextInt(edges * 2);

        for (Int2ObjectMap.Entry<VertexMeta> e : adjList.int2ObjectEntrySet()) {
            if (index - e.getValue().edges < 0) {
                for (IntPair i : e.getValue().adjVertices) {
                    if (index - i.getS() < 0) {
                        return new IntPair(e.getIntKey(), i.getF());
                    }

                    else index -= i.getS();
                }
            }

            else index -= e.getValue().edges;
        }

        throw new AssertionError();
    }
}


public class VertexMeta {
    public IntPair[] adjVertices;
    public int vertex;
    public int edges;

    public void addEdges(int toVertex, int numOfEdges) {
        this.adjVertices[toVertex].increaseS(numOfEdges);
        this.edges += numOfEdges;
    }

    public void setEdges(int toVertex, int numOfEdges) {
        this.edges -= (this.adjVertices[toVertex].getS() - numOfEdges);
        this.adjVertices[toVertex].setS(numOfEdges);
    }

    public int getEdges(int toVertex) {
        return this.adjVertices[toVertex].getS();
    }

    public boolean containsEdge(int toVertex) {
        return this.adjVertices[toVertex].getS() != 0;
    }

    public VertexMeta(int vertex, int vertices) {
        this.vertex = vertex;
        this.adjVertices = new IntPair[vertices];

        for (int i = 0; i < vertices; i++) {
            this.adjVertices[i] = new IntPair(i, 0);
        }
    }

    public VertexMeta(VertexMeta o) {
```

```
        this.vertex = o.vertex;
        this.edges = o.edges;
        this.adjVertices = new IntPair[o.adjVertices.length];

        for (int i = 0; i < o.adjVertices.length; i++) {
            this.adjVertices[i] = new IntPair(o.adjVertices[i]);
        }
    }
}



public class IntPair {
    private int f, s;

    public IntPair(int f, int s) {
        this.f = f;
        this.s = s;
    }

    public IntPair(IntPair p) {
        this.f = p.f;
        this.s = p.s;
    }

    public int getF() {
        return f;
    }

    public int getS() {
        return s;
    }

    public void setS(int s) {
        this.s = s;
    }

    public void increaseS(int by) {
        this.s += by;
    }
}
```

5 Генерация данных

Опишем схему работы и приведём реализацию генератора случайного связного графа с заданным количеством вершин и ребёр.

На первом шаге генератор добавляет в граф нужное количество вершин, создавая на каждой итерации новую вершину и новое ребро, соединяющее новую вершину со случайной из текущего набора. Таким образом обеспечивается условие связности графа.

На втором шаге алгоритм дополняет граф недостающим количеством ребёр, исключая наличие петель и кратных связей.

```
public class GraphGenerator {
    private static final Random random = new Random();

    public static Graph generate(int verticesN, int edgesN) {
        if (verticesN > edgesN) throw new RuntimeException();

        Int2ObjectMap<VertexMeta> adjList = new Int2ObjectOpenHashMap<>();
        int edges = 0;

        adjList.put(0, new VertexMeta(0, verticesN));

        for (int i = 1; i < verticesN; i++) {
            int to = random.nextInt(adjList.size());
            adjList.put(i, new VertexMeta(i, verticesN));
            adjList.get(i).addEdges(to, 1);
            adjList.get(to).addEdges(i, 1);
            edges++;
        }

        while (edges < edgesN) {
            int from, to;

            do {
                from = random.nextInt(verticesN);
                to = random.nextInt(verticesN);

            } while (from == to || adjList.get(from).containsEdge(to));

            adjList.get(from).addEdges(to, 1);
            adjList.get(to).addEdges(from, 1);
            edges++;
        }

        return new Graph(adjList);
    }
}
```

6 Эксперимент

Проведём описанный ранее вычислительный эксперимент в исследуемом диапазоне размеров входных данных. Далее приведены программа и результаты эксперимента.

```
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@BenchmarkMode(Mode.AverageTime)
@State(Scope.Benchmark)
@Fork(value = 2, jvmArgs = {"-Xms2G", "-Xmx2G"})
@Warmup(iterations = 3)
@Measurement(iterations = 7)
public class AlgorithmBenchmark {
    private Graph graph;

    @Param({ "8", "16", "24", "32", "40", "48", "56", "64", "72", "80", "88", "96", "104", \\
        \\ "112", "120", "128" })
    private int vertices;

    @Setup(Level.Trial)
    public void setup() {
        this.graph = GraphGenerator.generate(vertices, vertices * (vertices - 1) / 4);
    }

    @Benchmark
    public void test(Blackhole bh) {
        bh.consume(MinCut.minCut(graph));
    }

    public static void main(String[] args) throws Exception {
        Options options = new OptionsBuilder()
                .resultFormat(ResultFormatType.JSON)
                .result("results" + System.currentTimeMillis() + ".json")
                .build();

        new Runner(options).run();
    }
}
```

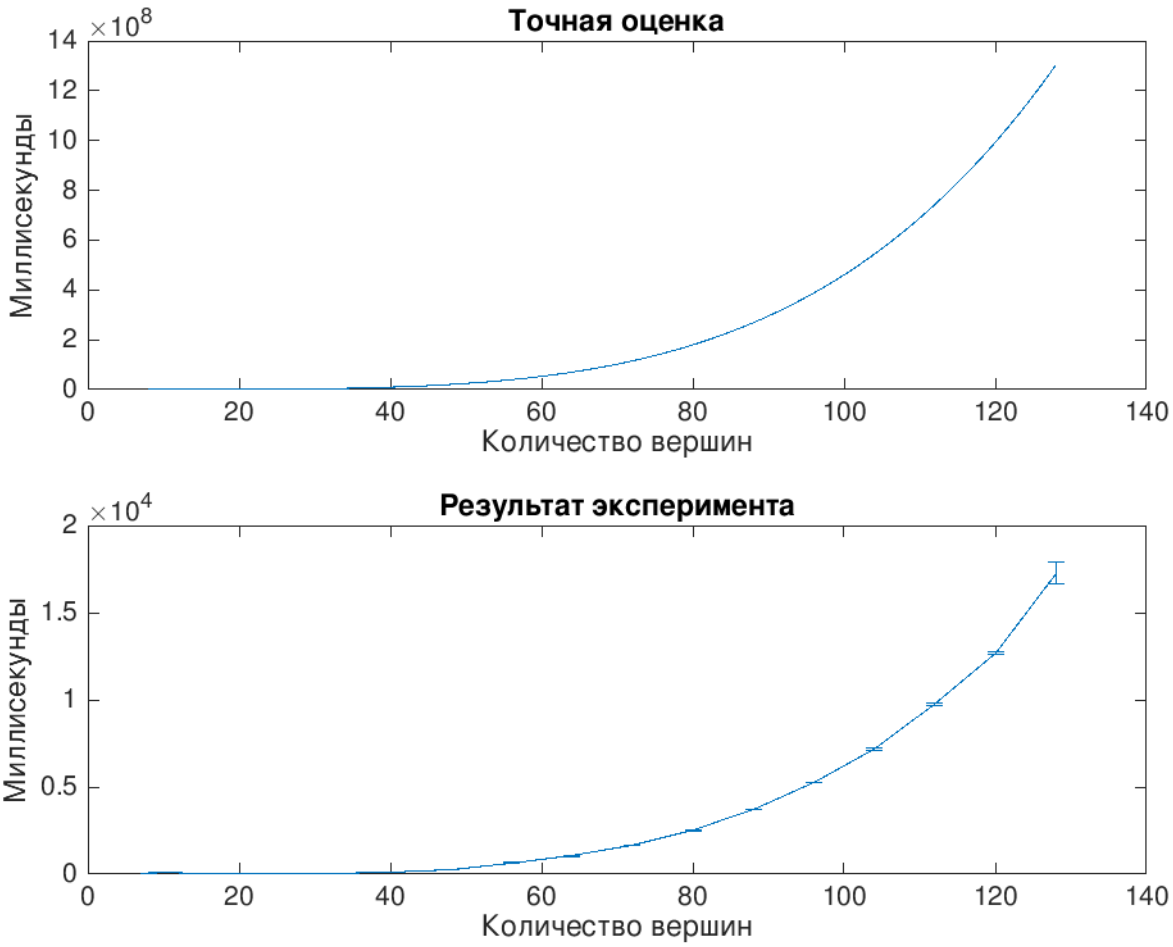
Полученные результаты:

```
# Run complete. Total time: 00:23:10

Benchmark                 (vertices)  Mode  Cnt       Score      Error  Units
AlgorithmBenchmark.test            8  avgt   14       0,156      0,016  ms/op
AlgorithmBenchmark.test           16  avgt   14       2,336      0,267  ms/op
AlgorithmBenchmark.test           24  avgt   14      17,012      0,428  ms/op
AlgorithmBenchmark.test           32  avgt   14      52,169      3,168  ms/op
AlgorithmBenchmark.test           40  avgt   14     121,468      6,123  ms/op
AlgorithmBenchmark.test           48  avgt   14     262,459      6,864  ms/op
AlgorithmBenchmark.test           56  avgt   14     634,884     28,760  ms/op
AlgorithmBenchmark.test           64  avgt   14    1037,073     21,383  ms/op
AlgorithmBenchmark.test           72  avgt   14    1651,745     19,123  ms/op
AlgorithmBenchmark.test           80  avgt   14    2521,044     21,111  ms/op
AlgorithmBenchmark.test           88  avgt   14    3709,148     19,156  ms/op
AlgorithmBenchmark.test           96  avgt   14    5261,755     19,892  ms/op
AlgorithmBenchmark.test          104  avgt   14    7173,079    102,922  ms/op
AlgorithmBenchmark.test          112  avgt   14    9757,983     89,831  ms/op
AlgorithmBenchmark.test          120  avgt   14   12691,223     91,408  ms/op
AlgorithmBenchmark.test          128  avgt   14   17267,708    619,535  ms/op
```

7 Анализ экспериментальных данных

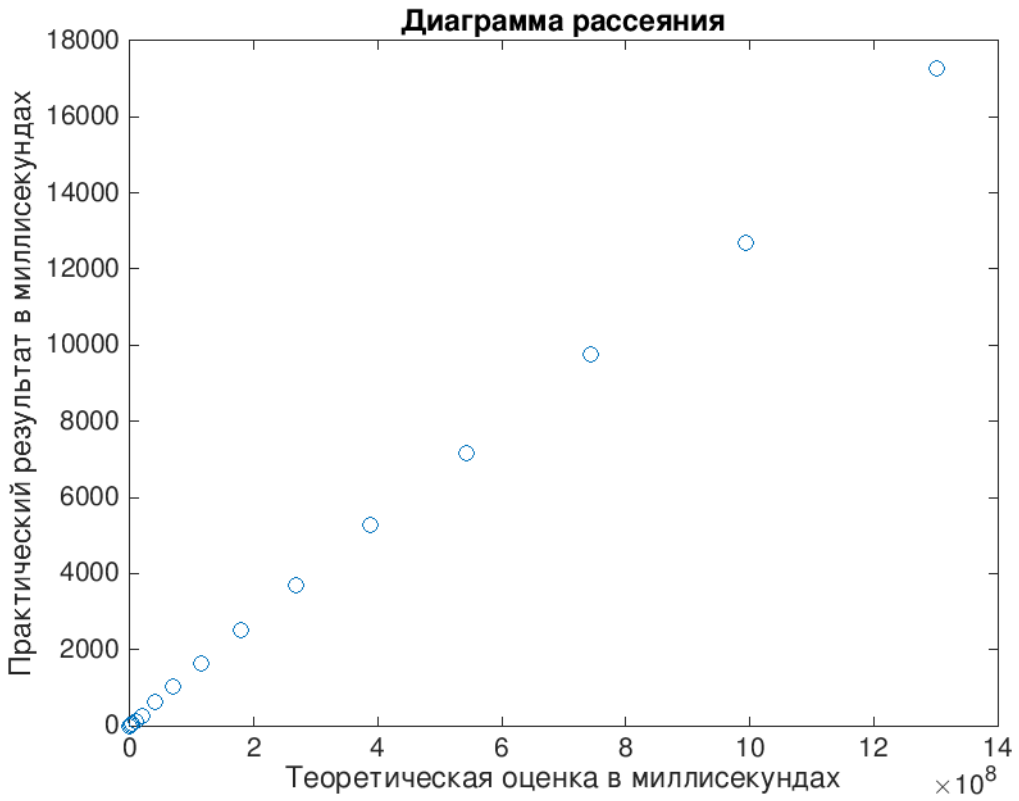
Проанализируем результаты, полученные в результате экперимента. Для этого сравним графики значений теоретической функции временной сложности алгоритма и практических данных в заданном диапазоне количества вершин.

```
val = jsondecode(fileread('results.json'));
vertices = arrayfun(@(x) str2double(x.params.vertices), val);
time = arrayfun(@(x) x.primaryMetric.score, val);
err = arrayfun(@(x) x.primaryMetric.scoreError, val);

x = linspace(vertices(1), vertices(end), 100);

f = @(n) (n .^ 4 .* log(n));

figure;
subplot(2, 1, 1);
plot(x, f(x));

subplot(2, 1, 2);
errorbar(vertices, time, err);
```



Построим диаграмму рассеяния и рассчитаем коэффициент корреляции величин.

```
corr = corrcoef(f(vertices), time);
plot(f(vertices), time, 'o');
title('Диаграмма рассеяния');
```



Коэффициент корреляции равен 0.999711, что говорит о наличие сильной линейной взаимосвязи величин. Исходя из этого можно сделать вывод о практическом совпадении теоретической оценки алгоритма и результатов проведенного эксперимента в исследуемом диапазоне размера входного графа.

8 Ссылки

Github проект: https://github.com/gRastaSsS/kargers-algorithm.

[1] https://en.wikipedia.org/wiki/Karger%27s_algorithm

[2] https://nickhar.wordpress.com/2012/02/06/lecture-10-minimum-cuts-by-the-contraction-algorithm/

[3] https://openjdk.java.net/projects/code-tools/jmh/

[4] http://fastutil.di.unimi.it/