Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное

учреждение высшего профессионального образования

Кубанский государственный технологический университет

(ФГБОУ ВПО КубГТУ )

Факультет компьютерных технологий и автоматизированных систем

Кафедра информационных систем и программирования

**КУРСОВАЯ РАБОТА**

По дисциплине «Дискретная математика»

На тему: «Задачи и алгоритмы дискретной математики»

Выполнил студент группы 12-КБ-ПИ1 Сафаров Роман Исмаилович

(Ф.И.О.)

Руководитель работы \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ Е.А. Симоненко

(подпись, дата, расшифровка подписи)

Защищена \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата)

Члены комиссии \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.А. Симоненко

(подпись, дата, расшифровка подписи)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.Г. Волик

(подпись, дата, расшифровка подписи)

(подпись, дата, расшифровка подписи)

Краснодар

2013

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное

учреждение высшего профессионального образования

Кубанский государственный технологический университет

(ФГБОУ ВПО КубГТУ)

Факультет компьютерных технологий и автоматизированных систем

Кафедра информационных систем и программирования

УТВЕРЖДАЮ

Зав. кафедрой ИСП

профессор Л.А. Видовский

«\_\_\_» 2013 г.

**ЗАДАНИЕ**

на курсовую работу

Студенту Сафаров Р.И. группы 12-КБ-ПИ-1 2 курса

факультета Компьютерных технологий и автоматизированных систем

направление 230700.62 – Прикладная информатика

Тема работы: "Задачи и алгоритмы дискретной математики"

Содержание задания: Разработать Windows-приложение, для решения задач и алгоритмов дискретной математики

(коротко раскрыть основной смысл задания на курсовую работу)

Объём курсовой работы:

а) пояснительная записка 22 стр.;

б) программа.

Рекомендуемая литература Скиена С. Алгоритмы. Руководство по разработке. Фленов М.Е.

Срок выполнения проекта: с «2» сентября 2013г. по «21» декабря 2013г.

Срок защиты: «28» декабря 2013г.

Дата выдачи задания: «2» сентября 2013г.

Дата сдачи работы на кафедру: с «2» декабря 2013г. по «28» декабря 2013г.

Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.А. Симоненко

(подпись)

Задание принял студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Р.И. Исмаилович

(подпись)

**РЕФЕРАТ**

Пояснительная записка к курсовой работе 22 страницы, 9 рисунков, 5 источников, 2 приложения.

ВЕРШИННАЯ СВЯЗНОСТЬ, ОСТОВНЫЕ ДЕРЕВЬЯ, АЛГОРИТМ БОРУВКИ, НЕОРИЕНТИРОВАННЫЙ ГРАФ

В данной курсовой работе рассмотрены вопросы решения по задачам и алгоритмам дискретной математики. Были рассмотрены такие темы как: «Вершинная связность графа» и «Алгоритм Борувки».

Основными моментами проведённого исследования были:

* изучение алгоритмов решения задач;
* изучение графов;
* изучение вершинной связности графов;
* изучение поиска точек сочленения;
* создание приложения на основе полученных данных.

Проделанная работа даст представление о способах решения приведенных задач и их наглядной реализации.

Содержание

[Введение 5](#_Toc375882727)

[1 Нормативные ссылки 6](#_Toc375882728)

[2. Вершинная связность 7](#_Toc375882729)

[2.1 Связь между вершинной, реберной связностью и минимальной степенью вершины 7](#_Toc375882730)

[2.2 Нахождение реберной связности 9](#_Toc375882731)

[2.3 Точки сочленения 10](#_Toc375882732)

[3 Минимальные остовные деревья: алгоритм Борувки 13](#_Toc375882733)

[Список используемых источников 18](#_Toc375882734)

[Приложение А – Листинг программы «Проверка графа на наличие вершинной связности. Поиск точек сочленения» 19](#_Toc375882735)

[Приложение Б – Листинг программы «Минимальные остовные деревья: алгоритм Борувки» 20](#_Toc375882736)

# Введение

Вершинной связностью \lambda графа G называется наименьшее число вершин, которое нужно удалить, чтобы граф перестал быть связным.

Остовное дерево – ациклический [связный](http://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D0%B3%D1%80%D0%B0%D1%84) [подграф](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%B3%D1%80%D0%B0%D1%84) данного связного [неориентированного графа](http://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B3%D1%80%D0%B0%D1%84), в который входят все его вершины. Неформально говоря, остовное дерево состоит из некоторого подмножества рёбер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим рёбрам, и в нём нет циклов, то есть из любой вершины нельзя попасть в саму себя, не пройдя какое-то ребро дважды.

Неориентированный граф - это совокупность объектов со связями между ними.

Работа алгоритма Борувки состоит из нескольких итераций, каждая из которых состоит в последовательном добавлении рёбер к остовному лесу графа, до тех пор, пока лес не превратится в дерево, то есть, лес, состоящий из одной компоненты связности.

# Нормативные ссылки

В данной пояснительной записке использованы ссылки на следующие стандарты.

ГОСТ Р 50739-95 Средства вычислительной техники. Защита от несанкционированного доступа к информации. Общие технические требования

ГОСТ 2.105-95 Общие требования к текстовым документам

ГОСТ 2.301-68 Единая система конструкторской документации. Форматы

ГОСТ 7.32-2001 СИБИД. Отчет о научно-исследовательской работе. Структура и правила оформления

ГОСТ 7.1-2003 СИБИД. Библиографическая запись. Библиографическое описание документа. Общие требования и правила составления

ГОСТ 7.80-2000 СИБИД. Библиографическая запись. Заголовок. Общие требования и правила составления

ГОСТ 19.701-90 СТД. (ИСО 5807-85) ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения

ГОСТ 34.601-90 Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания

ОК 015-94 (МК002-97) Общероссийский классификатор единиц измерения

МР КубГТУ 4.4.3-2004 СМК. УМД. Выпускные квалификационные работы

# 2. Вершинная связность

# 2.1 Связь между вершинной, реберной связностью и минимальной степенью вершины

Вершинной связностью \kappa графа G называется наименьшее число вершин, удаление которых приводит к несвязному или тривиальному графу.

Пускай минимальная степень вершины графа G обозначается буквой \delta. Тогда:

Для любого графа G справедливо следующее неравенство: \kappa \le\lambda \le \delta

1. Проверим второе неравенство. Если в графе G нет ребер, то \lambda = 0. Если ребра есть, то несвязный граф получаем из данного, удаляя все ребра, инцидентные вершине с наименьшей степенью. В любом случае \lambda \le \delta.
2. Чтобы проверить первое неравенство нужно рассмотреть несколько случаев.
   1. Если G - несвязный или тривиальный граф, то \kappa = \lambda = 0.
   2. Если G связен и имеет мост x, то \lambda = 1. В последнем случае \kappa = 1, поскольку или граф G имеет точку сочленения, инцидентную ребру x, или же G=K_2.
   3. Наконец, предположим, что граф G содержит множество из \lambda \ge 2 ребер, удаление которых делает его несвязным. Ясно, что удаляя \lambda - 1 ребер из этого множества получаем граф, имеющий мост x = uv. Для каждого из этих \lambda - 1 ребер выберем какую-либо инцидентную с ним вершину отличную от u и v. Удаление выбранных вершин приводит к удалению \lambda - 1 (а возможно, и большего числа) ребер. Если получаемый после такого удаления граф не связен, то \kappa < \lambda; если же он связен, то в нем есть мост x, и поэтому удаление вершины u или v приводит либо к несвязному, либо к тривиальному графу. В любом случае \kappa \le \lambda.

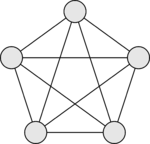


Рисунок 1 – Полный граф

Для любых натуральных чисел a, b, c, таких что a \le b \le c, существует граф G, у которого \kappa  = a, \lambda  = b и \delta = c

Рассмотрим граф G, являющийся объединением двух полных графов G_1 и G_2, содержащих c + 1вершину. Отметим b вершин, принадлежащих подграфу G_1 и a вершин, принадлежащих подграфу G_2. Добавим в граф G b ребер так, чтобы каждое ребро было инцидентно помеченной вершине, лежащей в подграфе G_1 и помеченной вершине, лежащей в подграфе G_2, причем не осталось ни одной помеченной вершины, у которой не появилось хотя бы одно новое ребро, инцидентное ей. Тогда:

1. Поскольку b \le c, то было как минимум две непомеченные вершины, поэтому \delta = c, так как минимальные степени вершин графов G_1 и G_2 были равны c, а степени их вершин не уменьшались.
2. Заметим, что между двумя вершинами графа G существует не меньше a вершинно-непересекающихся простых цепей, следовательно по [теореме Менгера](http://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B5%D0%BE%D1%80%D0%B5%D0%BC%D0%B0_%D0%9C%D0%B5%D0%BD%D0%B3%D0%B5%D1%80%D0%B0) \kappa \ge a. Однако если удалить из графа G помеченные вершины его подграфа G_2, то граф G потеряет связность. Значит, \kappa = a.
3. Аналогично рассуждению пункта 2, легко убедится, что \lambda = b.

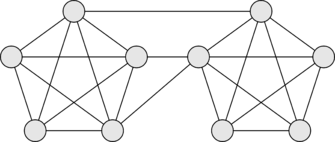


Рисунок 2 - Граф, в котором \delta = 4, \lambda = 3, \kappa = 2.

# 2.2 Нахождение реберной связности

Граф  G является реберно l - связным \Leftrightarrow любая пара его вершин соединена по крайней мере l - реберно непересекающимися путями.

**Реберной связностью** графа называется \lambda(G) = \max \{ l | G реберно l - связен \}, для тривиального графа считаем \lambda (K_1) = 0.

Для нахождения реберной связности нужно перебрать все пары вершин s и t, найти количество непересекающихся путей из s в t и выбрать минимум. Пусть он равен l. По утверждению, граф является l - связным, причем такое l - максимально (ведь мы явно нашли количество путей). А значит, по определению, реберная связность равна l.

Для нахождения количества непересекающихся путей из s в t воспользуемся алгоритмом нахождения максимального потока. Сопоставим каждому ребру пропускную способность, равную 1 и найдем максимальный поток. Он и будет равен количеству путей. Действительно, если провести декомпозицию потока, то получим набор реберно непересекающихся путей из s в t, по которым поток неотрицателен и равен 1 (т.к. пропускная способность всех ребер равна 1). А значит, если поток равен flow, то и количество путей равно flow.

ans = INF

for s \in V:

for t \in V:

flow = find\_flow(s, t)

ans = min(ans, flow)

Используя аналогичные утверждения и определения для вершинной связности придем к такому же алгоритму с тем отличием, что понадобится искать вершинно-непересекающиеся пути. Искать их можно тем же способом, если сопоставить каждой вершине пропускную способность, равную 1. Для этого воспользуемся известным трюком:

Разобьем каждую вершину v графа на две вершины v_1 и v_2. Все ребра, которые входили в v будут входить в v_1. Все ребра, которые выходили из v будут выходить из v_2. Так же добавим ребро (v_1, v_2) с пропускной способностью 1.

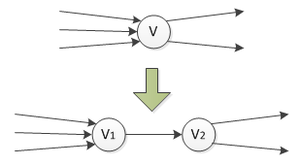


Рисунок 3 -  Вершинно непересекающиеся пути

После этого для нахождения количества вершинно непересекающихся путей в исходном графе будем искать количество реберно непересекающихся в новом графе.

Тем самым сведя задачу к нахождению реберной связности.

# 2.3 Точки сочленения

(1) Точка сочленения [графа](http://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F:_%D0%B3%D1%80%D0%B0%D1%84,_%D1%80%D0%B5%D0%B1%D1%80%D0%BE,_%D0%B2%D0%B5%D1%80%D1%88%D0%B8%D0%BD%D0%B0,_%D1%81%D1%82%D0%B5%D0%BF%D0%B5%D0%BD%D1%8C,_%D0%BF%D0%B5%D1%82%D0%BB%D1%8F,_%D0%BF%D1%83%D1%82%D1%8C,_%D1%86%D0%B8%D0%BA%D0%BB) G — вершина, принадлежащая как минимум двум блокам G.

(2) Точка сочленения графа G — вершина, при удалении которой в G увеличивается число [компонент связности](http://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D1%82%D0%BD%D0%BE%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D0%B8,_%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D1%8B_%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D0%B8).

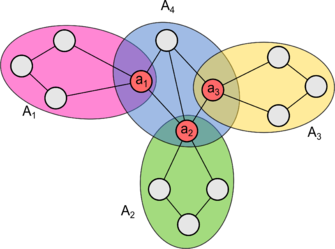


Рисунок 4 - Вершины a_1, a_2, a_3 - точки сочленения графа G

Определения (1) и (2) эквивалентны.

|  |
| --- |
|  |
| 1 \Rightarrow 2 Пусть вершина v принадлежит некоторым блокам A и B. Вершине v инцидентны некоторые ребра e=uv \in A и f=wv \in B. Ребра e и fнаходятся в различных блоках, поэтому не существует двух непересекающихся путей между их концами. Учитывая, что один из путей между концами - путь из v в эту же вершину, получаем, что любой путь, соединяющий u и w, пройдет через v. При удалении v между u и w не останется путей, и одна из компонент связности распадется на две.  2 \Rightarrow 1 Пусть v принадлежала только одному блоку C. Все вершины u_1...u_n, смежные с v, также лежат в C (в силу рефлексивности отношения вершинной двусвязности). Между каждой парой u_i, u_j вершин из u_1...u_n существует как минимум два вершинно непересекающихся пути. Теперь удалим v. Это разрушит путь u_{i}vu_{j}, но не разрушит любой оставшийся, так как иначе он тоже прошел бы через v.  Рассмотрим D — компоненту связности, в которой лежала v. Пусть между вершинами u, w \in D существовал путь, проходящий через v. Но он проходил также через некоторые вершины из u_1...u_n, связность которых не нарушилась, поэтому есть как минимум еще один путь, отличный от удаленного. Противоречие: число компонент связности не увеличилось.  Следующие утверждения эквивалентны:  (1) v — точка сочленения графа G;  (2) существуют такие вершины u и w, отличные от v, что v принадлежит любому простому пути из u в w;  (3) существует разбиение множества вершин V \setminus \{v\} на такие два подмножества U и W, что для любых вершин u \in U и w \in W вершина v принадлежит любому простому пути из u в w.  1 \Rightarrow 3 Так как v — точка сочленения графа G, то граф G \setminus v не связен и имеет по крайней мере две компоненты. Образуем разбиение V \setminus \{v\}, отнеся к Uвершины одной из этих компонент, а к W — вершины всех остальных компонент. Тогда любые две вершины u \in U и w \in W лежат в разных компонентах графа G \setminus v. Следовательно, любой простой путь из u в w графа G содержит v.  3 \Rightarrow 2 Следует из того, что (2) - частный случай (3).  2 \Rightarrow 1 Если v принадлежит любому простому пути в G, соединяющему u и w, то в G нет простого пути, соединяющего эти вершины в G \setminus \{v\}. Поскольку G \setminus \{v\} не связен, то v — точка сочленения графа G. |

# Минимальные остовные деревья: алгоритм Борувки

Впервые был опубликован в 1926 году Отакаром Борувкой в качестве метода нахождения оптимальной электрической сети в Моравии. Несколько раз был переоткрыт, например Флореком,Перкалом и Соллином. Последний, кроме того, был единственным западным ученым из этого списка, и поэтому алгоритм часто называют алгоритмом Соллина, особенно в литературе попараллельным вычислениям.

Работа алгоритма состоит из нескольких итераций, каждая из которых состоит в последовательном добавлении рёбер к остовному лесу графа, до тех пор, пока лес не превратится в дерево, то есть, лес, состоящий из одной компоненты связности.

В псевдокоде, алгоритм можно описать так:

1. Изначально, пусть T — пустое множество ребер (представляющее собой остовный лес, в который каждая вершина входит в качестве отдельного дерева).
2. Пока T не является деревом (что эквивалентно условию: пока число рёбер в T меньше, чем V − 1, где V — число вершин в графе):
   * Для каждой компоненты связности (то есть, дерева в остовном лесе) в подграфе с рёбрами T, найдём самое дешёвое ребро, связывающее эту компоненту с некоторой другойкомпонентой связности. (Предполагается, что веса рёбер различны, или как-то дополнительно упорядочены так, чтобы всегда можно было найти единственное ребро с минимальным весом).
   * Добавим все найденные рёбра в множество T.
3. Полученное множество рёбер T является минимальным остовным деревом входного графа.

На каждой итерации число деревьев в остовном лесу уменьшается по крайней мере в два раза, поэтому всего алгоритм совершает не более O(log V) итераций. Каждая итерация может быть реализована со сложностью O(E), поэтому общее время работы алгоритмы составляет O(Elog V) времени (здесь V и E — число вершин и рёбер в графе, соответственно).

Однако для некоторых видов графов, в частности, планарных, оно может быть уменьшено до O(E). Существует также рандомизированный алгоритм построения минимального остовного дерева, основанный на алгоритме Борувки, работающий в среднем за линейное время.

Итак, пусть дан связный неориентированный граф *G*(*V*;*E*) и на нем задана весовая функция . Пусть *A* – промежуточный остовный лес для графа *V*. На первом шаге *A* состоит из всех вершин *G* и пустого множества ребер. В начале очередной фазы алгоритма Борувки, для каждой компоненты связности промежуточного остовного леса выбирается **лидер** («leader» node, Erickson **[7]**) или **корень** - вершина, сопоставляемая каждой компоненте. Сделать это можно в простейшем случае с помощью обхода *A* в глубину: вершина, с которой начинается обход очередной компоненты, и будет ее лидером. В алгоритме за это отвечает процедура CHOOSE-LEADERS. Лидер для вершины *v* хранится в переменной leader(*v*).

После того, как лидеры выбраны, для каждой компоненты связности находится безопасное для нее ребро, по существу методом грубой силы. Это работа процедуры FIND-SAFE\_EDGES. Безопасное ребро для лидера *v'* хранится в переменной safe(*v'*). Как только все такие ребра отобраны, они добавляются к*A*. Процесс продолжается до тех пор, пока в *A* присутствует больше одной компоненты связности.

# g_boruvka00.gif

Рисунок 5 - Начальная фаза. Минимальный покрывающий лес состоит из всех вершин графа и пустого множества ребер

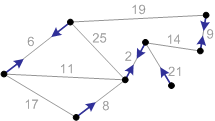


Рисунок 6 - Для каждой компоненты связности (для каждой вершины) находим безопасные ребра. Они отмечены стрелками от компоненты вдоль безопасного ребра

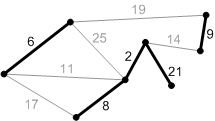


Рисунок 7 - Добавляем безопасные ребра к минимальному остовному лесу

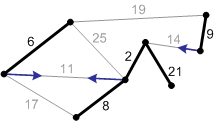


Рисунок 8 - Находим безопасные ребра для каждой компоненты связности

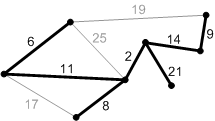


Рисунок 9 - Добавляем найденные ребра. Минимальное остовное дерево построено

Каждое обращение к FIND-SAFE-EDGES требует O(E) времени, т.к. внутри процедуры происходит проверка каждого ребра. Так как мы считаем граф связным, в нем максимум |E|+1 вершин. Тогда в случае, если граф представлен связным списком, требуется O(E) времени для выполнения каждой итерации основного цикла алгоритма Борувки MST-BORUVKA. Каждая итерация уменьшает количество компонент как минимум в два раза (худший случай, если все компоненты разбиваются на пары). Таким образом, если изначально количество компонент было равно V, главный цикл алгоритма выполняется O(logV). Общее время работы алгоритма получается равным O(ElogV).

**Заключение**

В ходе выполнения курсовой работы были изучены следующие вопросы:

* изучение алгоритмов решения задач;
* изучение графов;
* изучение вершинной связности графов;
* изучение поиска точек сочленения;

Достоинствами данной программы являются:

* минимальные системные требования;
* простота использования программы и удобство интерфейса пользователя.

Недостатками данной программы являются:

* отсутствие встроенных плагинов;
* открытый код программы;
* отсутствие дополнительных элементов управления;
* малое функционирование программы.

Была составлена программа для вершинной связности графов, а также для алгоритма Борувки. После проведённых тестов, был сделан вывод, что программы работают корректно, следовательно, поставленная задача выполнена.

# Список используемых источников

1. Г.Шилд. Самоучитель С++, 2005 г., 688 с.
2. Бьерн Страуструп. Язык программирования С++. Специальное издание, 2008 г., 1104 с.
3. Стивен Дьюхерст. С++. Священные знания. Издательство: Символ. 2008 г., 234с.
4. Д. Блох. [Java. Эффективное программирование](http://forcoder.ru/java/java-effektivnoe-programmirovanie-739). Издательство: Лори. 2002 г., 224 с.
5. Брюс Эккель. Философия Java. 2-е издание. Издательство: Питер. 2009 г., 640с.

# Приложение А – Листинг программы «Проверка графа на наличие вершинной связности. Поиск точек сочленения»

#include <iostream>

#include <vector>

    int timer;

vector < int > d;

vector < int > up;

vector < int > used;

vector < int > child;

void dfs\_cp (int &v, vector < vector < int > > &e, vector < int > &is)

{

    used[v] = 1;

    ++ timer;

    d[v] = timer;

    up[v] = timer;

    int i;

    for (i = 0; i < e[v].size(); ++ i)

    {

        int to = e[v][i];

        if (used[to] == 0)

        {

            ++ child[v];

            dfs\_cp (to, e, is);

            up[v] = min (up[v], up[to]);

            if (up[to] >= d[v])

                is[v] = 1;

        }

        else

            up[v] = min (up[v], d[to]);

    }

}

void cut\_points (vector < vector < int > > &e, vector < int > &is)

{

    int i, j;

    int n = e.size ();

    is.assign (n, 0);

    timer = 0;

    d.assign (n, 0);

    up.assign (n, 0);

    used.assign (n, 0);

    child.assign (n, 0);

    for (i = 0; i < n; ++ i)

        if (used[i] == 0)

        {

            dfs\_cp (i, e, is);

            is[i] = (child[i] > 1);

        }

}

# Приложение Б – Листинг программы «****Минимальные остовные деревья: алгоритм Борувки****»

public class Boruvka  
{  
    // private representations  
    /\*\*  
     \* Array of edges, which form the MST of the graph  
     \*/  
    private Edge[] mst;  
    /\*\*  
     \* Edges not yet discarded and not yet in the MST  
     \*/  
    private Edge[] wannabes;  
    /\*\*  
     \* Each component's nearest neighbor with find component numbers as indices  
     \*/  
    private Edge[] neighbors;  
    /\*\*  
     \* Graph representation on which we are searching for MST  
     \*/  
    private Graph g;  
    /\*\*  
     \*  
     \*/  
    private UnionFind uf;  
    // constructors and methods  
    /\*\*  
     \* constructor  
     \* @param G Graph  
     \*/  
    public Boruvka(Graph G) {  
        this.g = G;  
    }  
    /\*\*  
     \* Boruvka's algorithm  
     \*  
     \*  
     \* @return minimal spanning tree - edges that form it  
     \*/  
  
    public Edge[] BoruvkaMSTalg()  
    {  
        Edge hlpEdge = new Edge(g.getMaxWeight(), 0, 0);  
        this.uf = new UnionFind(g.getCountVerteces());  
        this.wannabes = new Edge[this.g.getCountEdges()];  
  
         /\*\*  
         \* Get all edges from the graph G to the array edges  
         \*/  
        for (int i=0; i < g.getCountEdges(); i++)  
            this.wannabes[i] = g.getEdgeAt(i);  
  
  
        this.neighbors = new Edge[this.g.getCountVerteces()];  
        this.mst = new Edge[this.g.getCountVerteces()+1];  
  
        /\*\*  
         \* index, used to store those edges being saved for the next phase  
         \*/  
        int nxtPhase;  
        int k=1;  
  
        for (int i=this.g.getCountEdges(); i!=0; i=nxtPhase)  
        {  
            int l, m, n;  
  
            for (int o=0; o<this.g.getCountVerteces(); o++)  
                this.neighbors[o] = hlpEdge;  
  
            for (n=0, nxtPhase=0; n<i; n++) {  
                Edge e = this.wannabes[n];  
                l = this.uf.find(e.getSVIndex()-1);  
                m = this.uf.find(e.getDVIndex()-1);  
  
                if ( l==m )  
                    continue;  
                if ( e.getWeight() < this.neighbors[l].getWeight() )  
                    this.neighbors[l] = e;  
                if ( e.getWeight() < this.neighbors[m].getWeight() )  
                    this.neighbors[m] = e;  
  
                this.wannabes[nxtPhase++] = e;  
            }  
  
            for (n=0; n<this.g.getCountVerteces(); n++)  
                if ( this.neighbors[n] != hlpEdge ) {  
                    l = this.neighbors[n].getSVIndex();  
                    m = this.neighbors[n].getDVIndex();  
  
                    if ( !this.uf.find(l,m) ) {  
                        this.uf.unite(l,m);  
                        this.mst[k++] = this.neighbors[n];  
                    }  
                }  
        }  
        System.out.println("MST by Boruvka successful");  
        return this.mst;  
    }  
}