СОДЕРЖАНИЕ

|  |  |  |
| --- | --- | --- |
| ВВЕДЕНИЕ…………………….………………………………...................... | | 5 |
| 1 | ОПИСАНИЕ ПОНЯТИЙ НЕОБХОДТМЫХ ДЛЯ РЕАЛИЗАЦИИ АЛГОРИТМА КРУСКАЛА……………………………...……………. | 6 |
| * 1. Базовые понятия о графах……………………………………….   1.2 Матрица смежности..…………………………………….…….. | | 6  8 |
| 1.3 Список смежности………………………………………..….… | | 9 |
| 1.4 Матрица инцидентности….……………………………….…… | | 10 |
| 1.5 Список ребер…………………………………………………... | | 11 |
| 2 | СРАВНЕНИЕ РАЗЛИЧНЫХ АЛГОРИТМОВ НАХОЖДЕНИЯ МИНИМАЛЬНОГО ОСТОВОГО ДЕРЕВА…………………..……… | 13 |
| 2.1 Алгоритм Борувки…………………..………………….……….. | | 13 |
| 2.2 Алгоритм Дейкстры-Прима………….……………….…….….. | | 14 |
| 2.3 Алгоритм Крускала…………………………….………..……… | | 16 |
| 3 | ОПИСАНИЕ РЕАЛИЗАЦИИИ ПРОГРАММЫ И ЕЕ ФУНКЙИЙ… | 19 |
| ВЫВОДЫ………………………………………………………………………  СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ………………………….. | | 24  25 |
| ПРИЛОЖЕНИЕ А……………………………………………………………..  ПРИЛОЖЕНИЕ Б……………………………………………………………..  ПРИЛОЖЕНИЕ В…………………………………………………………….. | | 26  27  29 |
|  | |  |

ВВЕДЕНИЕ

Главной задачей данной курсовой работы является анлиз алгоритма Крускала для нахождение минимального остового дерева, а также его программная реализация и визуализация. Для этого сначала будут данные основные определения теории графов. Также были рассмотрены все виды представления графов, данная временная оценка данных реализаций. Затем аргументированно был выбран один из способов представления графов.

Фундаментальным разделом курсовой работы является сравнение различных алгоритмов нахождения минимального остового дерева, в которым были рассмотрены три основных алгоритма для нахождения минимального остового дерева: алгоритм Борувки, алгоритм Прима и алгоритм Крускала. Каждый из них содержит в себе ряд особенностей, которые были разобраны, и проведен их анализ.

Далее будет написана программа для визуализации алгоритма Крускала, которая будет в себя включать отрисовку начального графа, а также пошаговую отрисовку данного алгоритма. После написания программы будет подробно разобран алгоритм программы, а также разобраны все использованные функции. В конце будут даны развернутые выводы.

1 ОПИСАНИЕ ПОНЯТИЙ НЕОБХОДТМЫХ ДЛЯ РЕАЛИЗАЦИИ АЛГОРИТМА КРУСКАЛА

* 1. Базовые понятия о графах

Граф – совокупность непустого множества вершин и наборов пар вершин.

Графы бывают двух видов: ориентированные, неориентированные и смешанные. Начнем с неориентированных.

Граф или неориентированный граф G – это упорядоченная пара G: = (V, E), для которой выполнены следующие условия:

V это множество вершин или узлов, E это множество пар (в случае неориентированного графа – неупорядоченных) различных вершин, называемых рёбрами.

V (а значит и E) обычно считаются конечными множествами. Многие хорошие результаты, полученные для конечных графов, неверны (или каким-либо образом отличаются) для бесконечных графов. Это происходит потому, что ряд соображений становятся ложными в случае бесконечных множеств.

Вершины и рёбра графа называются также элементами графа, число вершин в графе | V | – порядком, число рёбер | E | – размером графа.

Вершины u и v называются концевыми вершинами (или просто концами) ребра e = {u,v}. Ребро, в свою очередь, соединяет эти вершины. Две концевые вершины одного и того же ребра называются соседними.

Два ребра называются смежными, если они имеют общую концевую вершину.

Два ребра называются кратными, если множества их концевых вершин совпадают.

Ребро называется петлёй, если его концы совпадают, то есть e = {v,v}.

Степенью degV вершины V называют количество рёбер, для которых она является концевой (при этом петли считают дважды).

Ориентированный граф - (или сокращенно орграф) G = (V, Е) состоит из множества вершин V и множества дуг Е. Вершины также называют узлами, а дуги – ориенти­рованными ребрами. Дуга представима в виде упорядоченной пары вершин (v, w), где вершина v называется началом, a w - концом дуги. Дугу (v, w) часто записывают как v → w. Говорят, также, что дуга v → w ведет от вершины v к вершине w, а вершина w смежная с вершиной v.

Вершины орграфа можно использовать для представления объектов, а дуги – для отношений между объектами. Например, вершины орграфа могут представлять горо­да, а дуги – маршруты рейсовых полетов самолетов из одного города в другой.

Путем в орграфе называется последовательность вершин v1, v2, ….vn, для которой существуют дуги v1 → v2,v2 → v3, ...,vn-1 → vn . Этот путь начинается в вершине v1 и, проходя через вершины v2, v3, ..., vn-1 заканчивается в вершине vn Длина пути – количество дуг, составляющих путь, в данном случае длина пути равна n - 1. Как особый случай пути рассмотрим одну вершину v как путь длины 0 от вершины v к этой же вершине v. На рис. 1. последовательность вершин 1, 2, 4 образуют путь длины 2 от вершины 1 к вершине 4.

Путь называется простым, если все вершины на нем, за исключением, может быть, первой и последней, различны. Цикл – это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине. На рис. 1. верши­ны 3, 2, 4, 3 образуют цикл длины 3.

Во многих приложениях удобно к вершинам и дугам орграфа присоединить какую-либо информацию. Для этих целей используется помеченный орграф, т.е. орграф, у кото­рого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

Минимальное остовое дерево (MST) в связанном взвешенном неориентированном графе – это остовое дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов, входящих в него рёбер.

Существует множество вариантов реализовывать граф в программе. Четыре основных метода это: матрица смежности, матрица инцидентности, список дуг и список смежности.

1.2 Матрица смежности

Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа. Одним из наиболее общих представлений орграфа G = (V, Е) является матрица смежности. Предположим, что множество вершин V = {1, 2, ..., n}. Матрица смежности для орграфа G – это матрица А разме­ра n х n со значениями булевого типа, где A[i][j] = true тогда и только тогда, когда существует дуга из вершины i в вершину j. Часто в матрицах смежности значение true заменяется на 1, а значение false – на 0. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в которых надо часто проверять существование данной дуги.

Если дуги от верши­ны i к вершине j не существует, то значение A[i][j] не может быть значением какой-либо допустимой метки, а может рассматриваться как "пустая" ячейка.

Основной недостаток матриц смежности заключается в том, что она требует Ω(n2) объема памяти, даже если дуг значительно меньше, чем n2. Поэтому для чтения мат­рицы или нахождения в ней необходимого элемента требуется время порядка O(n2), что не позволяет создавать алгоритмы с временем O(n) для работы с орграфами, имеющими порядка O(n) дуг. Временные сложности сведены в Таблице 2.1.

Таблица 1.1 – Временная сложность операций в матрице смежности

|  |  |
| --- | --- |
| Операция | Временная сложность |
| Проверка смежности вершин x и y | О(1) |
| Перечисление всех вершин смежных с x | О(V) |
| Определение веса ребра (x, y) | О(1) |
| Перечисление всех ребер (x, y) | О(V2) |

1.3 Список смежности

Списком смежности для вершины i называется список всех вершин, смежных с вер­шиной i, причем определенным образом упорядоченный. Таким образом, орграф G можно представить посредством массива HEAD (Заголовок), чей элемент HEAD[i] является указателем на список смежности вершины i. Представление орграфа с по­мощью списков смежности требует для хранения объем памяти, пропорциональный сумме количества вершин и количества дуг. Если количество дуг имеет порядок O(n), то и общий объем необходимой памяти имеет такой же порядок. Но и для спи­сков смежности время поиска определенной дуги может иметь порядок O(n), так как такой же порядок может иметь количество дуг у определенной вершины. На рисунке в Приложение А на (Рисунок А.1) продемонстрировано данное представление графов.

Для вставки и удаления элементов в списках смежности необходимо иметь массив HEAD, содержащий указатель на ячейки заголовков списков смежности, но не сами смежные вершины. Но если известно, что граф не будет подвергаться изменениям (или они будут незначительны), то предпочтительно сделать массив HEAD массивом курсоров на массив ADJ (от adjacency – смежность), где ячейки ADJ[HEAD[i]], ADJ[HEAD[i]+ 1] и т.д. содержат вершины, смежные с вершиной i, и эта последовательность смежных вершин заканчивается первым встреченным нулем в массиве ADJ.

Рассмотрим временные сложности, которые приведены в Таблице 2.2.

Таблица 1.2 – Временные сложности операций в списке смежности

|  |  |
| --- | --- |
| Операция | Временная сложность |
| Проверка смежности вершин x и y | О(E) |
| Перечисление всех вершин смежных с x | О(E) |
| Определение веса ребра (x, y) | О(E) |
| Перечисление всех ребер (x, y) | О(E) |
| Поиск i-ой дуги | О(1) |

1.4 Матрица инцидентности

Говорить о том, что ребро g и каждая из вершин u и y инцидентна g, стоит лишь в том случае, если g соединяет u и y. Уяснив это, перейдем к рассмотрению данного метода. Матрица инцидентности строиться по похожему, но не по тому же принципу, что и матрица смежности. Так если последняя имеет размер n×n, где n – число вершин, то матрица инцидентности – n×m, здесь n – число вершин графа, m – число ребер. То есть теперь чтобы задать значение какой-либо ячейки, нужно сопоставить не вершину с вершиной, а вершину с ребром.

В каждой ячейки матрицы инцидентности неориентированного графа стоит 0 или 1, а в случае ориентированного графа, вносятся 1, 0 или -1. То же самое, но наиболее структурировано:

Неориентированный граф

1 – вершина инцидентна ребру

0 – вершина не инцидентна ребру

Ориентированный граф

1 – вершина инцидентна ребру, и является его началом

0 – вершина не инцидентна ребру

-1 – вершина инцидентна ребру, и является его концом

Пример матрицы приведен в Приложение А на (Рисунок А.2).

Каждый столбец отвечает за какое-либо одно ребро, поэтому граф, описанный при помощи матрицы инцидентности, всегда будет иметь следующий признак: любой из столбцов матрицы инцидентности содержит две единицы, либо 1 и -1 когда это ориентированное ребро, все остальное в нем – нули.

В программе матрица инцидентности задается, также как и матрица смежности, а именно при помощи двумерного массива. Его элементы могут быть инициализированы при объявлении, либо по мере выполнения программы. Временная сложность всех операций будет такая же как и у матрицы смежности.

1.5 Список ребер

Список, в каждой строке которого записаны две смежные вершины и вес, соединяющего их ребра, называется списком ребер графа. Возьмем связный граф G=(V, E), и множество ребер E разделим на два класса d и k, где d – подмножество, включающее только неориентированные ребра графа G, а k – ориентированные. Предположим, что некоторая величина p соответствует количеству всех ребер, входящих в подмножество d, а s – тоже относительно k. Тогда для графа G высота списка ребер будет равна s+p\*2. Иными словами, количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных ребер с неориентированными, увеличенными вдвое. Это утверждение следует из сказанного ранее, а именно, что данный способ представления графа предусматривает хранение в каждой строке двух смежных вершин, а неориентированное ребро, соединяющее вершины v и u, идет как из v в u, так и из u в v. Программная реализация списка ребер похожа на реализацию списка смежности. Также как и в последней, в ней изначально необходимо организовать ввод данных, которые при помощи специальной функции будут распределяться по массивам. Второй шаг – вывести получившийся список смежности на экран.

В списке смежности хранились только смежные вершины, а здесь, помимо них, будут храниться веса инцидентных этим вершинам ребер, и для этой цели понадобиться дополнительный массив. К тому же, данный метод требует более строго порядка вывода вершин, так как если в списке смежности последовательность нарушалась лишь в строках, то использование аналогичного способа построения приведет к нарушению порядка в столбцах. Поэтому функцию добавления ребер следует организовать иначе.

Для графа с Е ребрами такой способ хранения требует порядка Е памяти, но при этом основные операции — проверка наличия ребра между двумя вершинами и перебор вершин, смежных с данной — выполняются также за время порядка Е, поскольку требуют полного перебора всех списков ребер.

Иногда удобно хранить каждое ребро неориентированного графа дважды: как [u, v] и как [v, u].

Временные сложности сведены в Таблицу 2.3.

Таблица 1.3 – Временные сложности операций в списке ребер

|  |  |
| --- | --- |
| Операция | Временная сложность |
| Проверка смежности вершин x и y | О(E) |
| Перечисление всех вершин смежных с x | О(E) |
| Определение веса ребра (x, y) | О(E) |
| Перечисление всех ребер (x, y) | О(E) |
| Поиск i-ой дуги | О(1) |

Для алгоритма Крускала используется неориентированный, связный, взвешенный граф (следовательно, матрица инцидентности нам не потребуется). В качестве варианта реализации был выбран список ребер, так как у нас используется Алгоритм Крускала, то структура, которая хранит граф должна быть либо отсортирована по возрастанию веса ребра, либо в ней должен осуществляться поиск ребра с наименьшим весом (как например в матрице смежности). Наиболее быстро это делается именно с помощью списка ребер представленным в виде двумерного массива 3\*N, где N количество ребер графа.

2 СРАВНЕНИЕ РАЗЛИЧНЫХ АЛГОРИТМОВ НАХОЖДЕНИЯ МИНИМАЛЬНОГО ОСТОВОГО ДЕРЕВА

Построение MST-дерева – одна из наиболее популярных задач при программирование деревьев. Основные принципы ее решения были известны задолго до разработки современных структур данных и современных технологий анализа производительности алгоритмов, еще в те времена, когда поиск MST для графа, скажем, с тысячей ребер, представлял собой крайне сложную задачу. Как мы увидим, некоторые новые алгоритмы построения MST отличаются от старых, главным образом, способами использования и реализации алгоритмов и структур данных для основных задач, которые (наряду с мощью современных компьютеров) позволяют вычислять MST-деревья с миллионами и даже миллиардами ребер.

2.1 Алгоритм Борувки

Следующий алгоритм вычисления MST, который мы рассмотрим, также является самым старым. Мы строим MST, добавляя ребра в расширяющийся лес MST-поддеревьев, но делаем это поэтапно, добавляя в MST по нескольку ребер. На каждом этапе мы отыскиваем наиболее короткое ребро, которое соединяет каждое MST-поддерево с каким-то другим, а затем включаем все такие ребра в MST.

Вначале мы строим вектор, индексированный именами вершин, который для каждого MST-поддерева определяет ближайшего соседа. Затем для каждого ребра графа мы выполняем следующие операции:

Если оно соединяет две вершины одного и того же дерева, отбрасываем его.

Иначе проверяем расстояния между вершинами деревьев, которые соединяет это ребро и при необходимости изменяем их.

После просмотра всех ребер графа вектор ближайших соседних вершин содержит информацию, которая нужна для соединения поддеревьев. Для каждого индекса вершины мы выполняем операцию объединить, чтобы соединить ее с ближайшей соседней вершиной. На следующем этапе мы отбрасываем все более длинные ребра, которые соединяют другие пары вершин в уже соединенных MST-поддеревьях. Работа нашего алгоритма продемонстрирована в Приложение Б на (Рисунок Б.1). Задан начальный граф G. Каждая вершина является компонентой (синие окружности). На первой итерации внешнего цикла для каждой компоненты были добавлены минимальные сопряженные ребра. Некоторые ребра добавлены несколько раз (AD и CE). Осталось две компоненты. На последней итерации внешнего цикла было добавлено минимальное ребро, соединяющее две оставшиеся компоненты (ребро BE). Осталась одна компонента. Минимальное остовое дерево графа G построено.

На каждой итерации число деревьев в остовом 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(E log V) времени (здесь V и E - число вершин и рёбер в графе, соответственно).

Как было сказано, алгоритм Борувки – самый старый алгоритм из рассматриваемых нами: его идея впервые была выдвинута в 1926 г. для управления распределением электроэнергии. Затем он был заново открыт Сойеном (Sollin) в 1961 г.; а позже он привлек к себе внимание как основа для алгоритмов вычисления MST с эффективной асимптотической производительностью и как основа для алгоритмов параллельного построения MST.

2.2 Алгоритм Дейкстры-Прима

Алгоритм Прима, похоже, наиболее прост для реализации из всех алгоритмов поиска MST и рекомендуется для насыщенных графов. В нем используется сечение графа, состоящее из древесных вершин (выбранных для MST) и недревесных вершин (еще не выбранных в MST-дерево). Вначале мы выбираем в качестве MST-дерева произвольную вершину, затем помещаем в MST минимальное перекрестное ребро (которое превращает недревесную вершину MST-дерева в древесную) и повторяем эту же операцию V- 1 раз, пока все вершины не окажутся в дереве.

Из этого описания непосредственно следует примитивная реализация алгоритма Прима. Чтобы найти очередное ребро для включения в MST, необходимо просмотреть все ребра, которые выходят из древесной вершины в недревесную вершину, а затем выбрать из них самое короткое и включить его в MST.

Единичным шагом в алгоритме Прима является добавление вершины в MST-дерево, и прежде чем приступать к реализации, стоит хорошенько разобраться в его сути. Здесь главное – найти кратчайшее расстояние от каждой недревесной вершины до дерева. При присоединении вершины v к дереву единственным возможным изменением для недревесных вершин w является приближение w к дереву. То есть не нужно проверять расстояние от вершины w до всех вершин дерева: достаточно знать минимальное расстояние в каждый момент и проверять, изменяет ли добавление вершины v в дерево это минимальное расстояние.

Для реализации этой идеи нам потребуются такие структуры данных, которые предоставляют следующую информацию: ребра дерева, самое короткое дерево, соединяющее недревесную вершину с деревом, длина этого ребра.

Простейшей реализацией каждой из этих структур данных будет вектор, индексированный именами вершин (этот вектор можно использовать для хранения древесных ребер, выполняя индексацию по вершинам при их добавлении в дерево).

После включения в дерево нового ребра (и вершины) нужно выполнить еще две задачи:

-Проверить, приблизит ли добавление нового ребра какую-либо из недревесных вершин к дереву.

-Найти следующее ребро для включения в дерево.

Пример работы данного алгоритма показан в Приложение Б (Рисунок Б.2), далее опишем, что и как происходи на данном рисунке. Сначала определяется корень остового дерева, в данном случае – это вершина ноль. После этого находится ребро с минимальным весом исходящее из включенной вершины. Это ребро ноль-два. Далее определяется ребро с минимальным весом исходящим уже из одной из двух включенных в остовое дерево вершин – это ребро ноль-семь. С помощью данного принципа, избегая циклов, строится конечное остовое дерево.

Первым шагом вычисления MST-дерева по алгоритму Прима в это дерево заносится вершина ноль. Затем мы находим все ребра, которые соединяют 0 с другими вершинами (еще не включенными в это дерево), и выбираем из них самое короткое (слева вверху). Ребра, соединяющие древесные вершины с недревесными (накопитель), заштрихованы и перечислены под каждым чертежом графа. Для простоты ребра из накопителя перечисляются в порядке возрастания их длины, то есть самое короткое ребро – первое в этом списке. В различных реализациях алгоритма Прима используются различные структуры данных для хранения этого списка и определения минимального ребра. Вторым шагом самое короткое ребро 0-2 переносится (вместе с его конечной вершиной) из накопителя в дерево (вторая диаграмма сверху слева). На третьем шаге ребро 0-7 переносится из накопителя в дерево, в накопителе, ребро 0-1 заменяется на 7-1, ребро 0-6 на 7-6 (поскольку включение вершины 7 в дерево приближает к дереву вершины 1 и 6), а ребро 7-4 заносится в накопитель (поскольку добавление вершины 7 в дерево превращает 7-4 в ребро, которое соединяет древесную вершину с недревесной) (третья диаграмма сверху слева). Далее, мы переносим в дерево ребро 7-1 (слева внизу). В завершение вычислений мы исключаем из очереди ребра 7-6, 7-4, 4-3 и 3-5, обновляя накопитель после каждой вставки для отражения обнаруженных более коротких или новых путей (справа, сверху вниз).

2.3 Алгоритм Крускала

Алгоритм Прима строит минимальное остовое дерево по одному ребру, находя на каждом шаге ребро, которое присоединяется к единственному растущему дереву. Алгоритм Крускала также строит MST, добавляя к нему по одному ребру, но в отличие от алгоритма Прима, он отыскивает ребро, которое соединяет два дерева в лесу, образованном растущими MST-поддеревьями. Построение начинается с вырожденного леса из V деревьев (каждое состоящее из одной вершины), а затем выполняется операция объединения двух деревьев (самыми короткими ребрами), пока не останется единственное дерево - MST.

Пример работы данного алгоритма приведн в Приложение Б на (Рисунок Б.3).

Ребра AD и CE имеют минимальный вес, равный 5. Произвольно выбирается ребро AD (выделено на рисунке). Теперь наименьший вес, равный 5, имеет ребро CE. Так как добавление CE не образует цикла, то выбираем его в качестве второго ребра. Аналогично выбираем ребро DF, вес которого равен 6. Следующие ребра – AB и BE с весом 7. Произвольно выбирается ребро AB, выделенное на рисунке. Ребро BD выделено красным, так как уже существует путь (зелёный) между A и D, поэтому, если бы это ребро было выбрано, то образовался бы цикл ABD. Аналогичным образом выбирается ребро BE, вес которого равен 7. На этом этапе красным выделено гораздо больше ребер: BC, потому что оно создаст цикл BCE, DE, потому что оно создаст цикл DEBA, и FE, потому что оно сформирует цикл FEBAD. Алгоритм завершается добавлением ребра EG с весом 9. Минимальное остовое дерево построено.

Пусть задан список ребер графа в произвольном порядке (левый список ребер). На первом шаге алгоритма Крускала они сортируются по весам (правый список ребер). Затем мы просматриваем ребра этого списка в порядке возрастания их весов, добавляя в MST ребра, которые не создают в нем циклов. Сначала мы добавляем ребро 5-3 (самое короткое ребро), потом 7-6 (слева), затем 0-2 (справа вверху) и 0-7 (справа, вторая диаграмма сверху). Ребро 0-1 со следующим по величине весом создает цикл и поэтому не добавляется в дерево. Ребра, которые не включаются в MST, выделены в отсортированном списке серым цветом. Затем мы добавляем ребро 4-3 (справа, третья диаграмма сверху). Далее мы отбрасываем ребро 5-4, поскольку оно образует цикл, и потом добавляем 7-4 (справа внизу). Когда MST-дерево готово, любое ребро с большим весом образует цикл и поэтому будет отброшено (алгоритм останавливается, когда в MST будут включены V - 1 ребер). В отсортированном списке эти ребра помечены звездочками.

Ребра добавляются в MST-дерево в порядке возрастания их длины – таким образом, лес содержит вершины, соединенные друг с другом относительно короткими ребрами. В любой момент выполнения алгоритма каждая вершина расположена ближе к некоторой вершине своего поддерева, чем к любой другой вершине, не входящей в это дерево.

Обратите внимание, что существуют два способа окончания работы алгоритма Крускала. Если мы найдем V- 1 ребер, то мы уже построили остовное дерево и можем остановиться. Если мы просмотрим все вершины и не найдем V– 1 древесных ребер, то это означает, что граф не является связным.

Значения времени выполнения рассмотренных базовых алгоритмов вычисления MST приведены в таблице 3.1. Эти данные показывают, что реализация алгоритма Прима для матрицы смежности лучше подходит для насыщенных графов, что все другие методы отличаются по производительности от наилучшего результата лишь небольшими постоянными множителями (время на извлечение ребер) для графов средней насыщенности, и что метод Крускала сводит задачу к сортировке для разреженных графов.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 2.1 – Временные сложности алгоритма Крускала   |  |  |  | | --- | --- | --- | | **Алгоритм** | **Затраты в худшем случае** | **Примечание** | | Прима (стандартный) | V2 | Оптимален для насыщенных графов. | | Прима (PFS, пирамидальное d-дерево) | E\*log(d)V | Линейное время выполнения на всех графах, кроме очень разреженных. | | Крускала | E lgE | Превалируют затраты на сортировку. | | Крускала (частичная сортировка) | E + X lgV | Затраты зависят от веса самого длинного ребра. | | Борувки | E lgE | Осторожная верхняя граница. | | | |
|  |  |  |
|  |  |  |

Проанализировав данный раздел можно сказать, что коренным отличием алгоритма Крускала от алгоритма примы, является преодоление проблемы пересекающихся множеств, а структура алгоритма Борувки полностью отличается от структуры алгоритма Крускала.

3 ОПИСАНИЕ РЕАЛИЗАЦИИК ПРОГРАММЫ И ЕЕ ФУНКЙИЙ

Для написания программы был выбран язык C#, так как реализовывать алгоритм Крускла (а именно решить проблемы системы не пересекающихся множеств) на данном языке проще. Потому, что в систему непересекающихся множеств можно реализовывать с помощью массива строк, а в данном языке строки представлены объектами поэтому с ними намного проще работать чем в каком-либо другом языке программирования. Так же выбранный способ визуализации (Windows Forms) наиболее адаптирован для данного языка.

Наша программа состоит из двух файлов: «My Classes.cp» и «Form1.cp». Начнем с описания «My Classes.cp». В данном файле храниться класс Graph, который является нашим графом. Данный класс содержит только один спецификатор private, который является двумерный массив три на двадцать (так как большее количеств вершин не поместить на экране при визуализации). Все остальные методы класса объявлены со спецификатором publiс.

В данном классе первый метод, это – AddVerch, который принимает четыре параметра. Первый «rebra» хранит в себе номер ребра, которое мы добавляем, переменная «а1» из какой вершины исходит ребро, переменная «а2» в каую вершину и переменная «ves» хранит значение веса ребра. Данные значения соответственно передаются в список ребер. Программный код данного метода приведен в Таблице 3.1.

Таблица 3.1 – Метод AddVerch

|  |
| --- |
| public void AddVer(int rebra,int a1, int a2, int a3)  {  mas[0,rebra] = a1;  mas[1,rebra] = a2;  mas[2,rebra] = a3;  } |

Далее идет три функции, которые возвращают значение переменной, которая хранит откуда выходит ребро после значение переменной, которая хранит куда идет ребро ну и значение переменной, которая хранит вес ребра. Данный методы требуются для реализации алгоритма Крускала, как именно, будет описано позже.

Последний метод – это метод sort, который сортирует наш список ребер по возрастанию веса ребра, это так же требуется для алгоритма. В данном случае используется сортировка пузырьком, так как массив имеет небольшой размер, и данный вид сортировке наиболее прост к пониманию. Программный код метода приведен в Таблице 3.2.

Таблица 3.2 – Метод sort

|  |
| --- |
| public void sort(int size)  {  for (int i = 0; i < size; i++)  for (int j = i + 1; j < size; j++)  {  if (mas[2,i] > mas[2,j])  {  int temp;  temp = mas[2,i];  mas[2,i] = mas[2,j];  mas[2,j] = temp;    temp = mas[1,i];  mas[1,i] = mas[1,j];  mas[1,j] = temp;    temp = mas[0,i];  mas[0,i] = mas[0,j];  mas[0,j] = temp;  }  }  } |

Перейдем к файлу «Form1.cp». В данном файле реализовано все, что связанно с визуализацией. Внешний вид приложения приводится в Приложении В на (Рисунке В.1).

В начале данного файла, объявлен объект класса Graph, переменная «N» которая будет хранить значение количества вершин графа, переменная «rebra», которая будет хранить значения количества ребер нашего графа (ей присвоено значение «0» т.к изначально у нас нету ребер), а так же два массива («cordx» и «cordy»), которые будут хранить значения координат вершин графа.

Кнопка «Задать количество вершин» требуется для того, чтобы задать количество вершин графа. В листинге данной кнопки считывается значение из textBox1 и присваивается переменной «N», после чего выделяется память размера «N», для массивов координат. Программный код данной кнопки приведен в Таблице 3.3.

Таблица 3.3 – Листинг кнопки «Задать количество вершин»

|  |
| --- |
| private void button1\_Click(object sender, EventArgs e)  {  N=Convert.ToInt32(textBox1.Text);  this.textBox1.Text = "";  cordy = new int[N];  cordx = new int[N];  } |

Далее следует нажать кнопку, «Нарисовать граф», для того, чтобы на PictureBox1 отобразились вершины графа. В начале листинга данной кнопки объявляются переменный для работы с графикой (типов Graphics, Pen и Font (для работы с текстом)). Затем с помощью цикла, по кругу, рисуются вершины графа (круг с соответствующим номером внутри него) сохраняя координаты вершин в массивы «cordx» и «cordy». Программный код данной кнопки приведен в Таблице 3.4.

Таблица 3.4 – Листинг кнопки «Нарисовать граф»

|  |
| --- |
| private void button3\_Click(object sender, EventArgs e)  {  Graphics u = pictureBox1.CreateGraphics();  Pen k = new Pen(Color.Black, 1);  System.Drawing.Font shr = new System.Drawing.Font(FontFamily.GenericSansSerif, 9.0F, FontStyle.Regular);  double a = (2 \* 3.14) / N;  u.FillRectangle(Brushes.White, 0, 0, 300, 300);  for (int p = 0; p < N; p++)  {  int x = Convert.ToInt32(80 \* Math.Cos(a \* (p))) + 120;  int y = Convert.ToInt32(80 \* Math.Sin(a \* (p))) + 90;  cordx[p] = x + 10;  cordy[p] = y + 10;  u.DrawEllipse(k, x, y, 20, 20);  u.DrawString(Convert.ToString(p + 1), shr, Brushes.Blue, x + 5, y);  } } |

С помощью кнопки «Добавить новые ребра» добавляются ребра в граф. В листинге кнопки объявлены три переменных, которые будут отвечать за: «а1» и «а2» за номера вершин между которыми рисуется ребро, «а3» за вес данного ребра. Эти переменные принимают значения, вписанные в textBox1, textBox2, textBox3 соответственно. После чего объявляются стандартные переменные для работы с графикой и, рисуется соответствующее ребро и его вес, с координатами, взятыми из массивов координат, по индексу на один меньше чем введенные в текст боксы (т.к в массивах элементы хранятся с нулевого элемента, а номера вершин графа начинаются с первой). Программный код данной кнопки приведен в Таблице 3.5.

Таблица 3.5 – Листинг кнопки «Добавить новые ребра»

|  |
| --- |
| private void button2\_Click(object sender, EventArgs e)  {  int a1, a2, a3;  a1 = Convert.ToInt32(textBox2.Text);  a2 = Convert.ToInt32(textBox3.Text);  a3 = Convert.ToInt32(textBox4.Text);  this.textBox2.Text = "";  this.textBox3.Text = "";  this.textBox4.Text = "";  g.AddVer(rebra,a1,a2,a3);  rebra++;  Graphics u = pictureBox1.CreateGraphics();  Pen k = new Pen(Color.DarkRed, 2);  System.Drawing.Font shr = new System.Drawing.Font(FontFamily.GenericSansSerif, 8.0F, FontStyle.Regular);      u.DrawLine(k, cordx[a1 - 1], cordy[a1 - 1], cordx[a2 - 1], cordy[a2 - 1]);  u.DrawString(Convert.ToString(a3),shr, Brushes.DarkOrange,(cordx[a2 - 1] + cordx[a1 - 1]) / 2+7, (cordy[a2 - 1] + cordy[a1 - 1]) / 2+3);  u.FillEllipse(Brushes.White, cordx[a1 - 1] - 9, cordy[a1 - 1] - 9, 18, 18);  u.FillEllipse(Brushes.White, cordx[a2 - 1] - 9, cordy[a2 - 1] - 9, 18, 18);  u.DrawString(Convert.ToString(a1), shr, Brushes.Blue, cordx[a1 - 1] - 5, cordy[a1 - 1]-9);  u.DrawString(Convert.ToString(a2), shr, Brushes.Blue, cordx[a2 - 1] - 5, cordy[a2 - 1]-9); } |

Последняя кнопка – «Прорисовка алгоритма», изображает на PictureBox2 минимальное остовое дерево. После объявление стандартных графических переменных, рисуются по кругу вершины графа. Далее вызывается метод sort для нашего графа. Далее идет реализация алгоритма Крускала. Вначале объявляется массив строк, который будет хранить различные множества ребер, с размером N/2 (т.к одно ребро соединяет две вершины, то максимально количество множеств ребер вдвое меньше количества вершин графа). Затем объявляется переменная, которая хранит значение количества множеств. После чего начинается цикл который выполняется столько раз, сколько всего было добавлено ребер. Внутри этого цикла объявляются три переменных, которым присваиваются значения, которые возвращают методы GiveRebrIn, GiveRebrTo, GiverebrVes соответственно. Объявляется переменная типа String, которой присваивается значение «а1» и «а2» (с пробелом между ними), преобразованное в строчный тип. Данная переменная – это новое множество вершин. Далее идет условие, если рисуется первое ребро, тогда добавить новое множество в массив. Иначе идет проверка, что делать с новым множеством. С помощью цикла проверяется входит ли один из концов ребра в одно из старых множеств. Если обе вершины входят в одно множество (то есть при его добавление возникнет цикл), тогда переменной «yes» присваивается значение false. Если обе переменные не входят в множество, тогда переменная «shetchik» инкриминируется (после данного цикла есть проверка, если обе вершины не входят ни в одно и з уже добавленных множеств, тогда добавить новое множество в массив). Если же одна из вершин входит в одно из добавленных множеств, тогда в данное множество добавляется другой конец ребра, а если один конец ребра входит в одно множество, а другой конец в другое, тогда мы объединяем два этих множества. После окончания первичного цикла, в случае если переменная «yes» равняется true, рисуется соответствующее ребро. Результаты работы программы показаны на рисунках в Приложении В (Рисунок В.2), (Рисунок В.3).

ВЫВОДЫ

В ходе данного курсового проекта были детально описаны основные понятия теории графов, также были приведены основные виды представления графов при программировании. Для реализации графа в программе был выбран список дуг, так как данный метод предназначен для неориентированных взвешенных графов.

Также был проанализирован Алгоритм Крускала, приведен и описан пример работы данного алгоритма. Помимо этого, были объяснены принципы работ других похожих алгоритмов, таких как алгоритм Борувки и алгоритм Прима, и к ним также приведены примеры. После этого, было описано основное отличие алгоритма Крускала от других, которая заключается в решении задачи пересекающихся множеств.

Далее была создана и проанализирована программа реализующая и визуализирующая алгоритм Крускала, она состояла из двух файлов. В первом файле представлен граф, а во втором его визуализация и визуализация алгоритма Прима. Все методы класса реализующего граф, а также функционал интерфейса, который включал себя различные кнопки, каждая из которых выполняла определенную задачу, были подробно описаны, а также был продемонстрирован их код или же скриншот. При запуске данной программы требуется ввести начальные данные, после чего нажать соответствующую кнопку и произойдет визуализация алгоритма Прима для заданного графа.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Акимов О.Е.Дискретная математика. Логика, группы, графы. – 2-е изд. – М.: Лаборатория базовых знаний, 2001. – 376 с.
2. Аляев Ю.А. Тюрин С.Ф. Дискретная математика и математическая логика. – М.: Финансы и статистика, 2006. – 368 с.
3. Андерсон, Джеймс А. Дискретная математика и комбинаторика. – Пер. с англ. – М.: Издательский дом Вильямс, 2004. – 960 с.
4. Джефри Макконнелл Анализ алгоритмов. Активный обучающий подход – М.: Издательство: Техносфера, 2001. – 1068 с.
5. Стивен С. Скиена. Алгоритмы. Руководство по разработке – СПб.: Издательство: БХВ, 2005. – 926 с.
6. Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн. Алгоритмы. Построение и анализ – Третье издание – М.: Издательство Вильямс, 2003. – 1328 с.

ПРИЛОЖЕНИЕ А

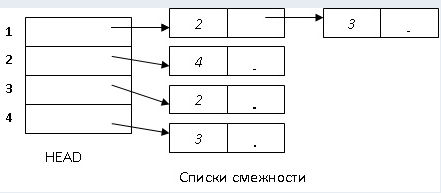


Рисунок А.1 – Список смежности

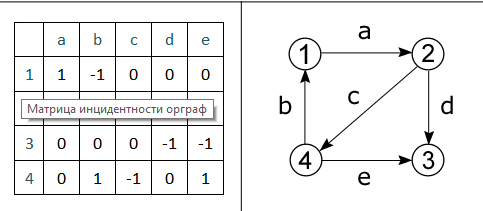


Рисунок А.2 – Матрица инцидентности

ПРИЛОЖЕНИЕ Б

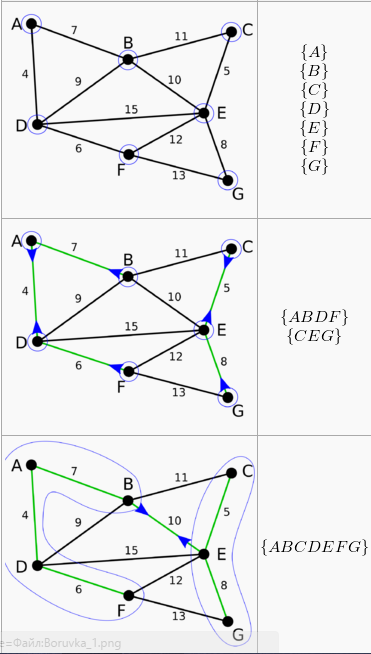


Рисунок Б.1 – Алгоритм Борувки

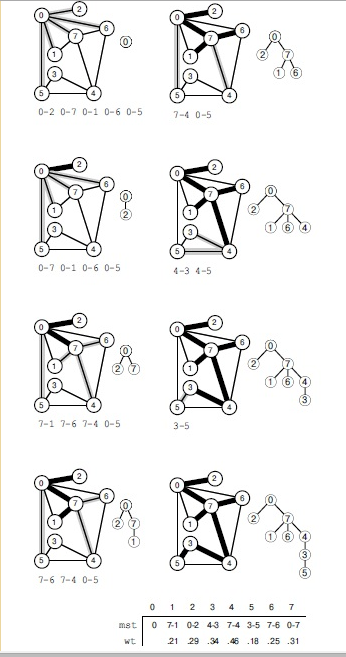


Рисунок Б.2 – Алгоритм Дейкстры-Прима

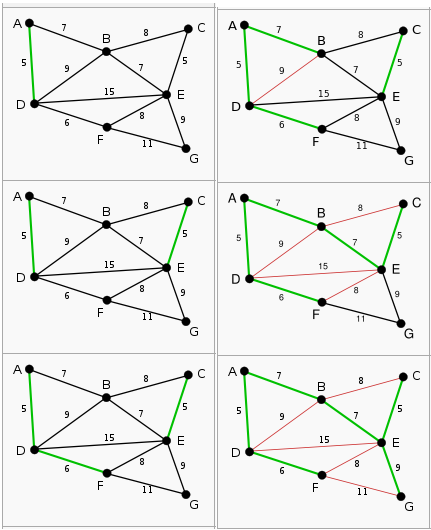


Рисунок Б.3 – Алгоритм Крускала

ПРИЛОЖЕНИЕ В

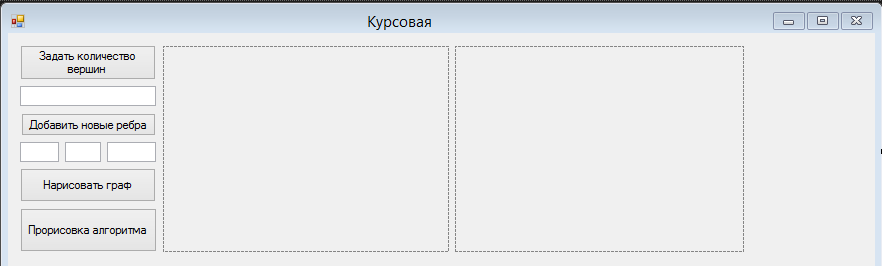


Рисунок В.1 – Внешний вид приложения

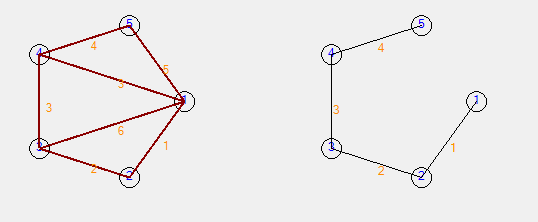


Рисунок В.2 – Прорисовка алгоритма пример первый

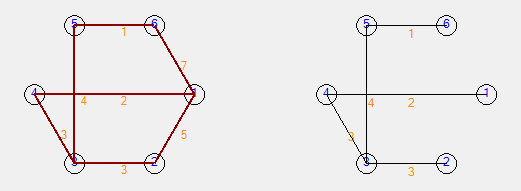


Рисунок В.3 – Прорисовка алгоритма пример второй