Содержание

Содержание 2

Введение 3

1. Алгоритм 4

1.1 Описание алгоритма 4

1.2 Псевдокод 7

1.3. Анализ сложности алгоритма 7

2. Инструкция пользователя 7

3 Тестовые примеры 9

Заключение 15

Список использованных источников 16

Введение

Задачей данной курсовой работы является разработка программы, которая реализовывала бы быстрый алгоритм поиска минимального остовного дерева (Алгоритм Крускала). Остовное дерево графа — это дерево, подграф данного графа, с тем же числом вершин, что и у исходного графа. Неформально говоря, остовное дерево получается из исходного графа удалением максимального числа рёбер, входящих в циклы, но без нарушения связности графа. Наиболее известными алгоритмами для решения данной задачи являются алгоритм Дейкстры, алгоритм Прима и алгоритм Крускала. Однако алгоритм Крускала достаточно прост в своей идее и реализации. Он заключается в сортировке всех рёбер в порядке возрастания длины, и поочерёдному добавлению их в минимальный остов, если они соединяют различные компоненты связности. Хотя оба алгоритма работают за O(MlogN), существуют константные различия в скорости их работы. На разреженных графах (количество рёбер примерно равно количеству вершин) быстрее работает алгоритм Крускала, а на насыщенных (количество рёбер примерно равно квадрату количеству вершин) - алгоритм Прима (при использовании матрицы смежности).

На практике чаще используется алгоритм Крускала.

На рисунке 1 приведен пример решения данной задачи:

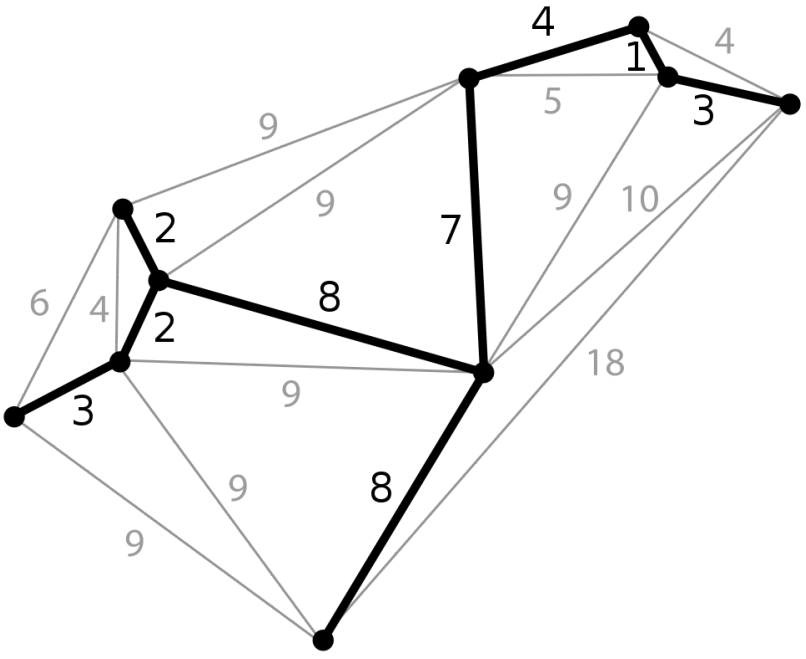


Рисунок 1. “Пример работы алгоритма”

1. Алгоритм

1.1 Описание алгоритма

Он подпадает под класс алгоритмов, называемых *«жадными» алгоритмами* , которые находят локальный оптимальный вариант в надежде найти глобальный оптимальный вариант.

Мы начинаем с ребер с наименьшим весом и продолжаем добавлять ребра, пока не достигнем нашей цели.

Шаги для реализации алгоритма Крускала следующие:

1. Сортировать все ребра по возрастанию.
2. Возьмите ребро с наименьшим весом и добавьте его в остовное дерево. Если добавление ребра создало цикл, то отклоните это ребро.
3. Продолжайте добавлять ребра, пока не достигнете всех вершин.

Более формально: пусть мы уже нашли некоторые рёбра, входящие в минимальный остов. Утверждается, что среди всех рёбер, соединяющих различные компоненты связности, в минимальный остов будет входить ребро с минимальной длиной.

Для реализации алгоритма Крускала необходимо уметь сортировать рёбра по возрастанию длины. Также любой минимальный алгоритм остовного дерева вращается вокруг проверки, создает ли ребро цикл или нет. Наиболее распространенный способ выяснить это - алгоритм Union FInd . Алгоритм Union-Find разделяет вершины на кластеры и позволяет нам проверить, принадлежат ли две вершины одному кластеру или нет, и, следовательно, решить создает ли добавление ребра цикл.

Визуализация работы алгоритма Крускала:

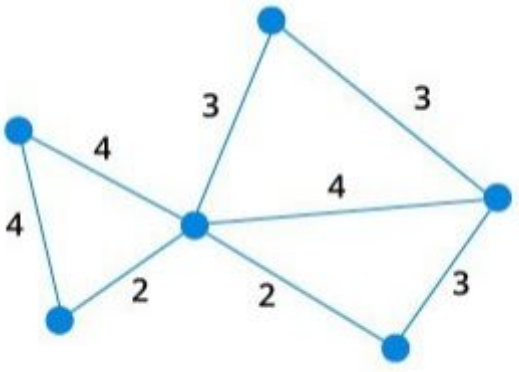


Рисунок 2.1.

Начните с взвешенного графа.

Рисунок 2.2.

Выберите ребро с наибольшим весом,если же их больше одного,то выберите любое.

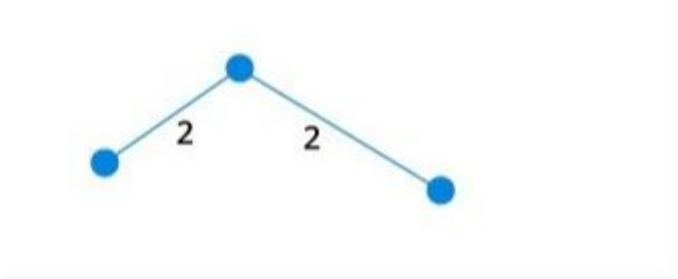


Рисунок 2.3.

Выберите следующее ребро с наименьшим весом и добавьте его.

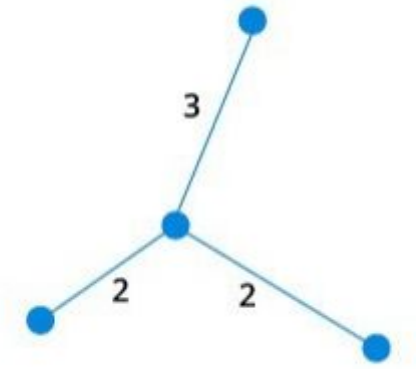


Рисунок 2.4.

Выберите следующее ребро с наименьшим весом, которое не создает цикл и добавьте его.

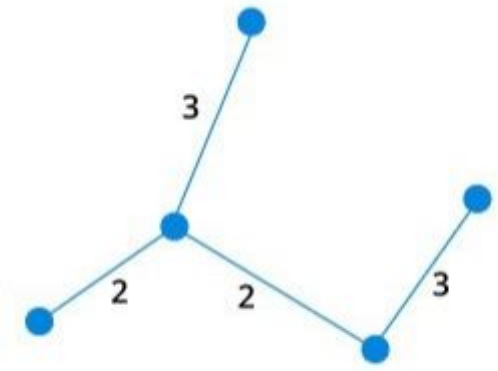


Рисунок 2.5.

Выберите следующее ребро с наименьшим весом, которое не создает цикл и добавьте его.

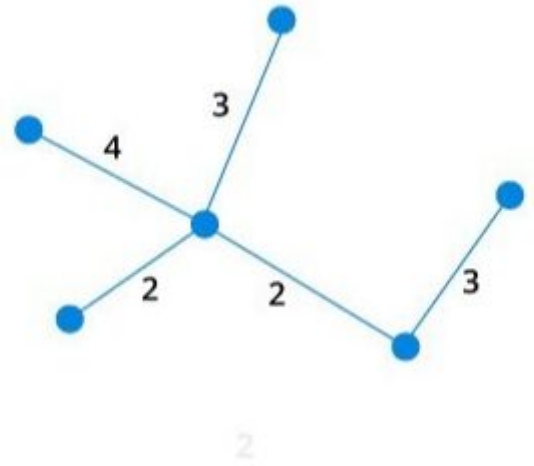


Рисунок 2.6.

Повторяйте до тех пор, пока не получите остовное дерево.

1.2 Псевдокод

string text - текст, в котором будет происходить поиск.

string str - строка, которую мы будем искать.

vector<int> index - результирующий вектор с позициями

```
void Graph::kruskal()
{
  int i, uRep, vRep;
  сортировка длин рёбер по
возрастанию();
  for (для каждого ребра) { //O(M)
    uRep = ищем связь(начало ребра); //
O(logN)
    vRep = ищем связь(конец ребра);
    if (uRep не соответствует vRep) {
      добавляем i-ое ребро(); //
добавляем в дерево
      объединение вершин(uRep, vRep);
    }
  }
}
```

1.3. Анализ сложности алгоритма

В худшем случае время работы алгоритма Крускала — O(MlogN), где M - количество рёбер графа, а N - количество вершин, так как сложность функции find_set составляет O(logN). Таким образом, наиболее оптимальным вариантом использования алгоритма является применение его к разреженным графам, то есть к графам, у которых количество рёбер примерно равно количеству вершин.

2. Инструкция пользователя

Запуск программы производится через командную строку. От пользователя требуется подать в программу путь к четырём файлам. Первый файл является входным и содержит сам граф. Второй и третий файл используются для вывода результата работы программы. Во втором будет содержаться вывод, совместимый с программой Graphviz (файл temp.dot). В третьем будет автоматически прописываться консольная

команда генерации изображения графа, после чего данный файл нужно будет просто запустить. (MyGraph.cmd). Четвертым указывается путь к расположению программы Graphviz. После этого в той же папке появится файл с остовным деревом графа, который наглядно показывает результат работы программы (файл temp.png).

Шаблон команды:

Kruskal.exe <input> <output1> <output2> <output3>, где:

input - путь к файлу, где граф представлен списком рёбер.

output1 - путь к файлу, куда будет выводиться граф для построения в Graphviz. (temp.dot)

output2 - путь к консольному файлу MyGraph.cmd, который при запуске генерирует требуемое нам изображение.

output3 - путь к расположению самой папки Graphviz.

3 Тестовые примеры

CD D:\myprojects\Kruskal\kruskal\Debug
d:\\myprojects\\Kruskal\\kruskal\\kruskal\\graph.txt d:\\Graphviz\\temp.dot
d:\\Graphviz\\MyGraph.cmd cd D:\Graphviz

Пример 1. “Веса некоторых ребер совпадают”.

Ввод:

```
6
0 1 4
0 2 4
1 2 2
1 0 4
2 0 4
2 1 2
2 3 3
2 5 2
2 4 4
3 2 3
3 4 3
4 2 4
4 3 3
5 2 2
5 4 3
```

Вывод:

Консоль:

```
done, file for graphviz was created
```

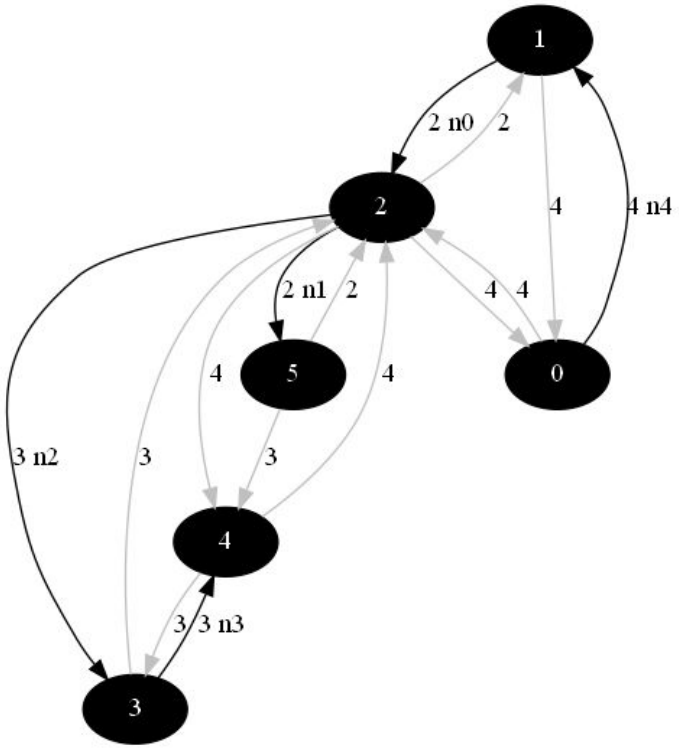
temp.dot:

temp – Блокнот

Файл Правка Формат Вид Справка

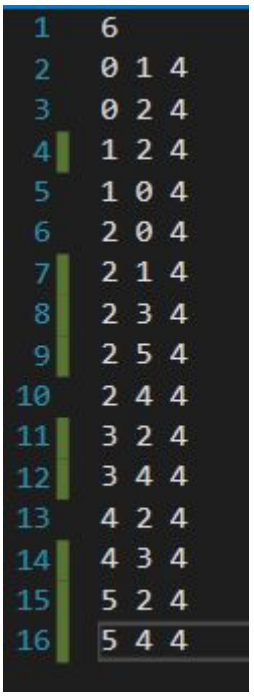
```
digraph G{
node[style = "filled", fillcolor = "black", fontcolor="#FFFFFF"];
1 -- 2[label="2 n0"];
2 -- 5[label="2 n1"];
2 -- 3[label="3 n2"];
3 -- 4[label="3 n3"];
0 -- 1[label="4 n4"];
2 -- 1[label="2",color="gray"];
5 -- 2[label="2",color="gray"];
3 -- 2[label="3",color="gray"];
4 -- 3[label="3",color="gray"];
5 -- 4[label="3",color="gray"];
0 -- 2[label="4",color="gray"];
1 -- 0[label="4",color="gray"];
2 -- 0[label="4",color="gray"];
2 -- 4[label="4",color="gray"];
4 -- 2[label="4",color="gray"];
}
```

Граф:



Пример 2.“Веса всех ребер одинаковые”.

Ввод:



Вывод:

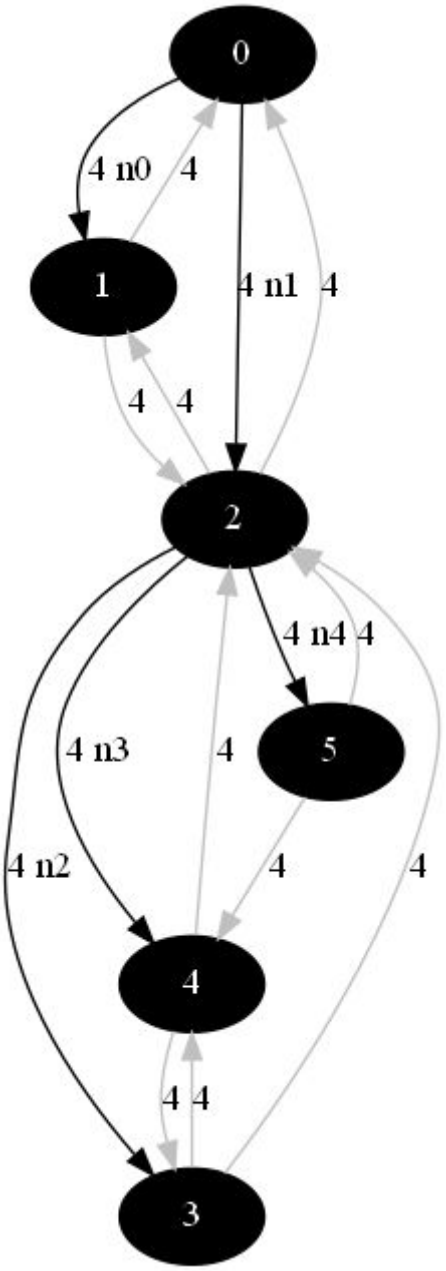
Консоль:

```
done, file for graphviz was created
```

temp.dot:

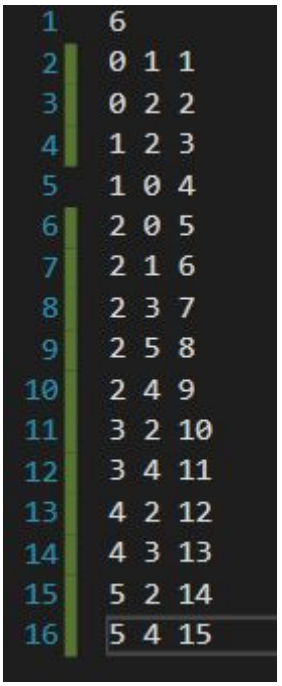
```
digraph G{
node[style = "filled", fillcolor = "black", fontcolor="#FFFFFF"];
0 -> 1[label="4 n0"];
0 -> 2[label="4 n1"];
2 -> 3[label="4 n2"];
2 -> 4[label="4 n3"];
2 -> 5[label="4 n4"];
1 -> 0[label="4",color="gray"];
1 -> 2[label="4",color="gray"];
2 -> 0[label="4",color="gray"];
2 -> 1[label="4",color="gray"];
3 -> 2[label="4",color="gray"];
3 -> 4[label="4",color="gray"];
4 -> 2[label="4",color="gray"];
4 -> 3[label="4",color="gray"];
5 -> 2[label="4",color="gray"];
5 -> 4[label="4",color="gray"];
}
```

Граф:



Пример 3.“Веса всех ребер разные”.

Ввод:



Вывод:

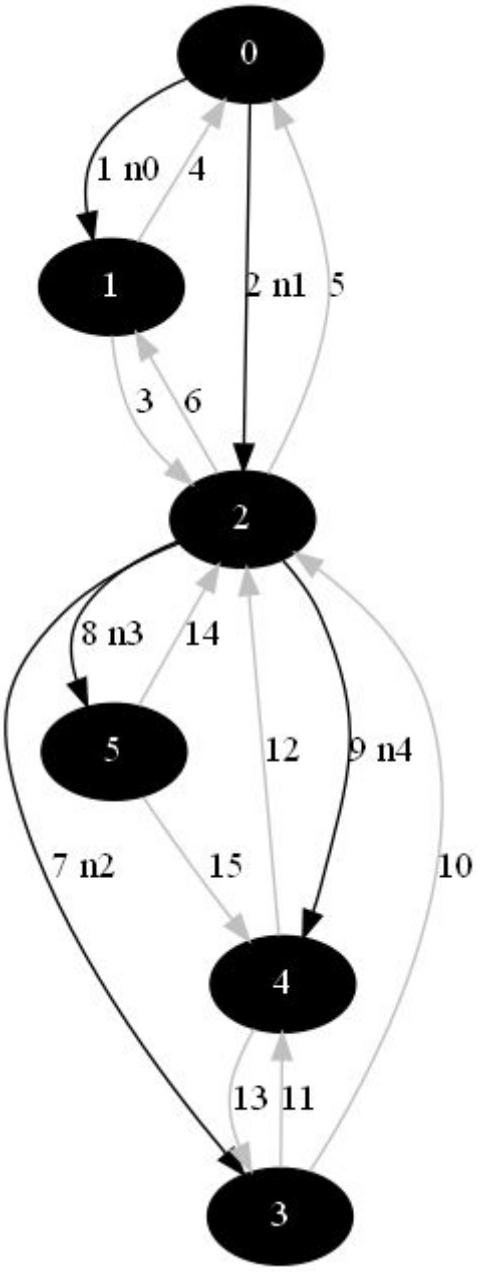
Консоль:

```
done, file for graphviz was created
```

temp.dot:

```
digraph G{
node[style = "filled", fillcolor = "black", fontcolor="#FFFFFF"];
0 -> 1[label="1 n0"];
0 -> 2[label="2 n1"];
2 -> 3[label="7 n2"];
2 -> 5[label="8 n3"];
2 -> 4[label="9 n4"];
1 -> 2[label="3",color="gray"];
1 -> 0[label="4",color="gray"];
2 -> 0[label="5",color="gray"];
2 -> 1[label="6",color="gray"];
3 -> 2[label="10",color="gray"];
3 -> 4[label="11",color="gray"];
4 -> 2[label="12",color="gray"];
4 -> 3[label="13",color="gray"];
5 -> 2[label="14",color="gray"];
5 -> 4[label="15",color="gray"];
}
```

Граф:



Заключение

Результатом курсовой работы является реализованный алгоритм Крускала. Он предназначен для поиска построения минимального остовного дерева. Представленный алгоритм имеет сложность O(MlogN), где M- количество рёбер графа, а N - количество его вершин. В ходе данного отчета мы смогли убедиться, что данный алгоритм действительно является достаточно эффективным для достижений этой цели, нежели аналогичные, такие как алгоритм Прима и алгоритм Дейкстры, обладающие более высокой сложностью при определённых условиях. Помимо построения минимального остовного дерева, алгоритм Крускала также используется для нахождения некоторых приближений задачи Штейнера. Важно заметить, что остовных деревьев может быть несколько, но будет представлен только самый первый подходящий под условие решения задачи вариант.

Разработанная программа способна справиться со своей задачей. В перспективе возможны доработка и оптимизация алгоритма, представленного в данной курсовой работе.

Список использованных источников

1. (Кормен Ю.Б., Лейзерсон Ч.Э., Ривест Р.Л, Штайн К.Э., 2013 год)
2. (“Алгоритм Крускала”, 2020)
3. (”Минимальное остовное дерево. Алгоритм Крускала”, 2008)
4. (”Алгоритм Крускала”, 2019)