Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования

Санкт-Петербургский государственный университет
Факультет прикладной математики и процессов управления

Проектная работа

По дисциплине: *Алгоритмы и анализ сложности*

Алгоритм Борувки нахождения минимального остовного дерева в графе

Выполнила:
Студентка 3 курса группы 18.Б13-пу
бакалаврской программы
“Программирование и информационные технологии”
Неумоина Елизавета Петровна

Проверил:
к.ф.-м.н. Никифоров Константин Аркадьевич

Санкт-Петербург
2020 г.

Содержание

Введение

Алгоритм Борувки – это алгоритм нахождения минимального остовного дерева в графе. Был впервые опубликован в 1926 году чешским математиком Отакаром Борувкой в качестве метода нахождения оптимальной электрической сети в Моравии. Стоит отметить, что алгоритм хорошо параллелизуется и является основой для распределенного алгоритма GHS.

Минимальные остовные деревья позволяют решить задачу, в которой требуется соединить множество точек (представляющих города, дома, перекрестки и др. объекты) наименьшим объемом дорожного полотна, труб, проводов и т.п.

Алгоритм следует “жадной” стратегии: на каждом этапе выбираются самые легкие ребра. Не для всех задач такой выбор ведет к оптимальному решению, однако для задачи о покрывающем дереве это так.

Алгоритм

1. В самом начале каждая вершина графа G является фрагментом (тривиальным деревом), а ребра не принадлежат ни одному из них.
2. Пока T не является деревом (что эквивалентно условию: пока число ребер в T меньше, чем V – 1, где V – количество вершин в графе / пока количество фрагментов больше 1):
   2.1. Для каждого фрагмента T определяется минимальное по весу инцидентное ему ребро.
   2.2. Минимальные ребра добавляются в MST, а соответствующие фрагменты объединяются.
3. Получившийся граф T является MST.

Данный алгоритм может работать неправильно, если в графе есть равные по весу ребра. Избежать эту проблему можно, например, выбирая в пункте 2.1. среди ребер, равных по весу, ребро с наименьшим номером. [1]

Псевдокод [2]

algorithm Borůvka is

input: граф *G*, ребра которого имеют различные веса.

output: *T* – минимальное остовное дерево графа *G*.

Инициализировать лес *T* как набор одновершинных деревьев, одно для каждой вершины графа.

while *T* имеет больше одной компоненты **do**

Найти связные компоненты *T* и обозначить каждую вершину *G* ее компонентом

Инициализировать самое легкое ребро для каждого компонента в "*None*"

for each ребра *{u-v}* в *G* **do**

if *u* и *v* имеют разные ветки компонентов:

if *{u-v}* легче, чем самое легкое ребро для компоненты *u*, **then**

Сделать *{u-v}* самым легким ребром компоненты *u*

if *{u-v}* легче, чем самое легкое ребро для компоненты *v*, **then**

Сделать *{u-v}* самым легким ребром компоненты *v*

for each компоненты, чье самое легкое ребро не “*None*” **do**

Добавить их самые легкие ребра в *T*.

Математический анализ алгоритма

***Свойство*.** Время прогона алгоритма Борувки с целью вычисления дерева MTS заданного графа есть $O(E * logV * logE)$.

Док-во*:** Поскольку число деревьев в лесе уменьшается наполовину на каждом этапе, число этапов не превышает значения $logV$. Время выполнения каждого этапа, самое большее пропорционально затратам на выполнение *E* операций *find*, что меньше *E*logE,* или линейно с точки зрения практических приложений. Функция *find* необходима для того, чтобы присвоить индекс каждому поддереву с таким расчетом, чтобы можно было быстро определить, к какому поддереву принадлежит вершина. [3] Т.е. эта функция определяет родительскую вершину для каждой вершины (в каждом поддереве одна родительская вершина). На каждом этапе эта функция вызывается сначала пропорциональное *E* (количеству ребер) раз, а затем количество раз, не превышающее *logE*. (см. подробнее код реализации ***boruvkaAlgorithm.py)

Время прогона, оценка которого дается свойством, представляет собой консервативную верхнюю границу, поскольку оно не учитывает существенное уменьшение ребер на каждом этапе. В каждой итерации количество деревьев уменьшается, по крайней мере вдвое, что дает нам MST после, самое большее, *logV* итераций, каждая из которых выполняется за линейное время. В целом это нам дает алгоритм с временем исполнения ***O(E*logV).*** [4]

Если зафиксировать количество ребер *E=4*V,* то получим сложность ***O(V*logV).***

Таким образом, мы получим **логарифмически линейный** класс эффективности.

Эмпирический анализ алгоритма

Воспользуемся общим планом эмпирического анализа алгоритма: [5]

1. Цель эксперимента: проверка точности теоретических выводов об эффективности алгоритма.

2. Измеряемая метрика: трудоемкость алгоритма.
   Единицы измерения: время выполнения программы.

3. Характеристики входных данных:
   3.1. Количество вершин *V*: [10, 100] с шагом 1
   3.2. Количество ребер *E*: [V – 1; V*(V-1)/2]. Так как сложность алгоритма зависит от количества ребер *E*, будем считать *E=4*V*
   3.3. Веса ребер: [1, 10]

4. Программная реализация алгоритма:
   Алгоритм принимает на вход массив ребер графа (ребро представляется в виде *[u, v, w]*, где *u* и *v* – вершины графа, *w* – вес ребра), на выходе выдает массив ребер MST и его вес MSTWeight.

   Код алгоритма находится в файле ***boruvkaAlgorithm.py***

5. Генератор образца входных данных:
   На вход принимает количество вершин *V* и ребер *E*, на выходе дает массив ребер (ребра в виде *[u, v, w]*).

   Основные шаги:
   5.1. Создается пустой массив (наш граф, в котором в дальнейшем будут храниться ребра).
   5.2. Для удобства задается матрица инцидентности размерности *[V x V]*, заполненная нулями. В ней будем хранить информацию о наличии ребра в графе.
   5.3. Случайным образом строится массив ребер (включая случайное определение веса ребра в заданном диапазоне), соединяющий все вершины (по сути список). Это необходимо, чтобы на выходе мы получили связный граф.
   5.4. Случайным образом выбираются оставшиеся *V - 1 – E* ребер:
      5.4.1. Случайным образом выбираются две вершины.
      5.4.2. В матрице инцидентности проверяется информация о наличии ребра, и, если оно отсутствует и вершины различны, случайным образом решается, будет ли ребро в графе (*[False, True]*).
      5.4.3. В случае *True* случайным образом выбирается вес в заданном диапазоне. Ребро добавляется в массив ребер графа, в матрицу инцидентности заносится информация о наличии ребра в графе.

Код генератора находится в файле ***boruvkaAlgorithm.py*** (функция ***generate_graph***).

6. Выполнение алгоритма над образцом входных данных.

Заданное *m* (количество образцов входных данных при фиксированном *n*): 30

Было решено повторять алгоритм для каждого *n* заданное количество раз (***repeats*** *= 30*) для получения усредненного времени выполнения. Это поможет избежать проблему получения нулевых значений времени работы алгоритма.

Измеренные значения трудоемкости (время выполнения):

n	10	11	12	13	14	15	16	17	18	19	20
f(n) (мс)	0.13514	0.13951	0.12306	0.1408	0.15720	0.21416	0.26810	0.25957	0.22629	0.32852	0.68306
21	22	23	24	25	26	27	28	29	30	31	32
0.35258	0.29587	0.30605	0.32156	0.36280	0.35210	0.44997	0.45957	0.57864	0.51723	0.44073	0.94534
33	34	35	36	37	38	39	40	41	42	43	44
0.69675	0.66751	0.59398	0.72147	0.62330	0.98494	1.26139	1.03948	1.05933	0.79701	0.91588	0.83840
45	46	47	48	49	50	51	52	53	54	55	56
1.37614	0.98419	0.84122	0.90845	0.91437	1.32676	1.00313	1.00675	1.04658	1.19265	1.71601	1.01573
57	58	59	60	61	62	63	64	65	66	67	68
1.05202	1.35315	1.31423	1.62676	1.28447	1.12683	1.26221	2.10612	1.26404	1.56744	1.36432	1.76997
69	70	71	72	73	74	75	76	77	78	79	80
1.54695	1.52209	1.91176	1.52810	1.59463	1.40063	2.22734	2.03686	1.64490	2.34527	1.74697	1.62548
81	82	83	84	85	86	87	88	89	90	91	92
2.74836	1.88234	2.26950	1.94409	1.90812	2.48014	1.63120	2.22448	2.53443	2.27045	2.44850	2.00849
93	94	95	96	97	98	99	100				
1.91653	2.53857	2.10686	3.17119	1.98594	4.17679	4.65084	4.32012				

Таб. 1. Измеренные значения трудоемкости при заданном n (размер входных данных – количество вершин)

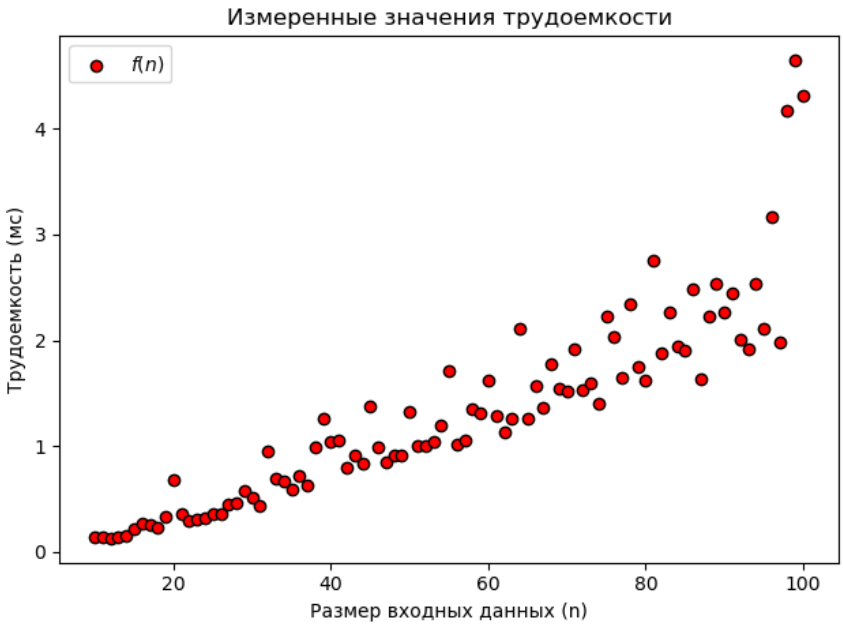

Рис.1. Измеренные значения трудоемкости

7. Анализ полученных результатов

7.1. Для проверки теоретической оценки $\Theta(n * \log(n))$ воспользуемся определением:

$$f(n) = \theta\big(g(n)\big)\ if\ \exists\ n_0 \in N, \exists c_1, c_2 > 0: \forall n \geq n_0: c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Для $n_0 \in [10, 100]$ найдем константы c_1, c_2 по принципу:

- $c_1 = \min\left(\frac{f(n)}{n*log(n)}\right)$ для $n \in [n_0; 100]; c_1 = 0.00286, n_0 = 10$
- $c_2 = \max\left(\frac{f(n)}{n*log(n)}\right)$ для $n \in [n_0; 100]; c_2 = 0.00790, n_0 = 10$
- $c_1 > 0$ и $c_2 > 0$

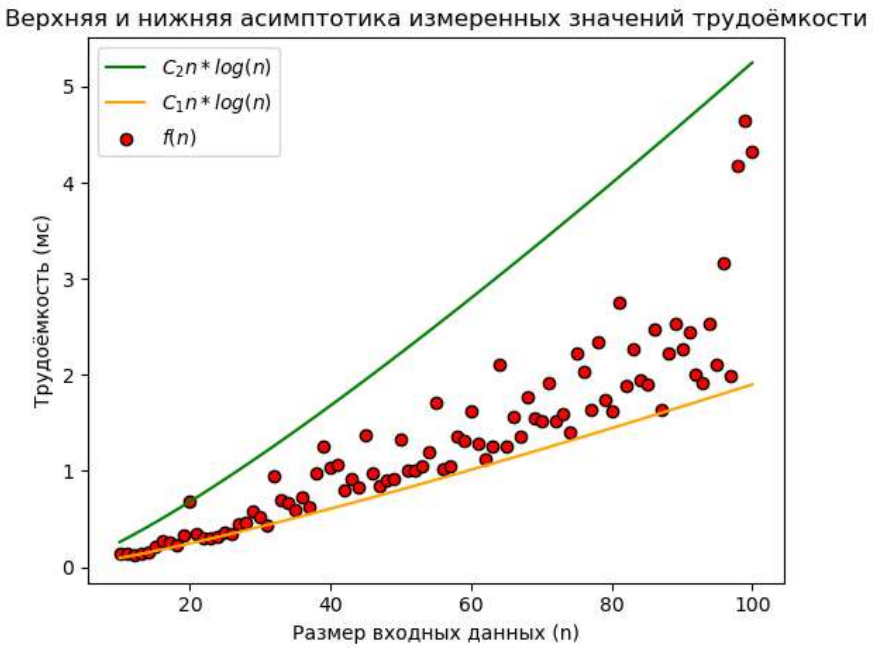

Рис. 2. Верхняя и нижняя асимптотика измеренных значений трудоемкости

7.2. Построим график $\frac{f(2n)}{f(n)}$ для $n \in [10, 50]$ с шагом 1.

Анализируя теоретическую оценку: $\lim_{n\to\infty} \frac{2n*\log(2n)}{n*\log(n)} = \lim_{n\to\infty} \frac{2\,(\log(2)+\log(n))}{\log(n)} = \lim_{n\to\infty} 2\left(\frac{1}{\log(n)}+1\right) = 2$ и график, делаем вывод, что с увеличением n происходит затухание к 2.

Рис. 3. Отношение измеренных значений трудоемкости при удвоении размера входных данных

Код анализа находится в файле ***empiricalAnalysis.py***

8. Вычислительная среда и оборудование:
    - Процессор *Intel® Core™ i5-7200U CPU @ 2.50GHz 2.71 GHz*
    - Установленная память (ОЗУ): 8 ГБ
    - Тип системы: 64-разрядная операционная система, процессор x64
    - Язык программирования: Python
    - Используемые библиотеки:
        - *NumPy* для векторных вычислений
        - *Random* для генерации случайных значений
        - *MatPlotLib* для визуализации результатов
        - *Time* для вычисления трудоемкости алгоритма

Весь код доступен по ссылке: https://github.com/Elizaneu/EmpiricalAnalysis

Литература

1. Университет ИТМО. Алгоритм Борувки [Электронный ресурс] - Алгоритм Борувки — Викиконспекты (ifmo.ru)
2. Wikipedia. The Free Encyclopedia. Borůvka’s algorithm. Pseudocode. - Borůvka's algorithm - Wikipedia
3. *Седжвик Р.* Часть 5. Алгоритмы на графах. Глава 20. Минимальные остовные деревья. 20.5. Алгоритм Борувки // Фундаментальные алгоритмы на C++. Алгоритмы на графах: Пер. с англ./Роберт Седжвик. – СПб: ООО “ДиаСофтЮП”, 2002. – С. 266 – 270. – ISBN 5-93772-054-7.
4. *Скиена С.* Глава 15. Задачи на графах с полиномиальным временем исполнения. 15.3. Минимальные остовные деревья // Алгоритмы. Руководство по разработке. – 2-у изд.: Пер. с англ. – СПб: БХВ-Петербург. 2011. – С. 500 – 505. – ISBN 978-5-9775-0560-4.
5. *Левитин А.В.* Глава 2. Основы анализа эффективности алгоритмов: Эмпирический анализ алгоритмов // Алгоритмы. Введение в разработку и анализ – М.: Вильямс, 2006. – С. 127 - 134. – ISBN 5-8459-0987-2.