Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Программная инженерия

Суханова Анжела Кирилловна

Реализация алгоритма поиска минимального остовного дерева с использованием библиотеки SuiteSparse

Курсовая работа

Научный руководитель:
к. ф.-м. н., доцент Григорьев С. В.

Санкт-Петербург
2020

Оглавление

Введение

Графы являются одним из наиболее мощных инструментов для моделирования сложных задач благодаря своей простоте и универсальности, и алгоритмы, работающие с графами, имеют колоссальное значение в современной жизни. Анализ графструктурированных данных постепенно становится всё более популярным и находит применение в различных областях, таких как биоинформатика, анализ социальных сетей, молекулярный синтез и планирование маршрутов. В этих условиях очень важным оказывается совершенствование представлений алгоритмов на графах: в частности, поскольку графы могут содержать миллиарды вершин, возникает необходимость в распараллеливании этих алгоритмов.

К сожалению, высокопроизводительные реализации графовых алгоритмов сложно представлять на новом параллельном оборудовании (например, графических процессорах). Исследования по разработке программ для параллельного оборудования позволили ускорить графовые алгоритмы [**?**, **?**], но улучшение их производительности произошло за счёт усложнения модели программирования. В результате возникло несоответствие между языками высокого уровня, с которыми бы предпочли иметь дело пользователи и разработчики, (например, Python), и языками и стандартами программирования для параллельного оборудования (например, C++, CUDA, OpenMP или MPI) [**?**].

На устранение этой проблемы направлено несколько инициатив. Одна из них, GraphBLAS [**?**], представляет собой открытый инновационный стандарт, определяющий структурные блоки графовых алгоритмов на языке линейной алгебры. GraphBLAS лежит в основе библиотеки, имеющей непосредственное отношение к этой работе, а именно SuiteSparse. SuiteSparse: GraphBLAS[1] [**?**] — это первая полная реализация стандарта GraphBLAS (тщательно протестированная для точного соответствия спецификации), которая предоставляет множество опе-

[1]Репозиторий с открытым исходным кодом SuiteSparse: GraphBLAS: `https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/master/GraphBLAS` [Accessed: 31th May, 2020].

раций над разреженными матрицами с использованием полуколец и практически неограниченного разнообразия операторов и типов. Последнее достигается за счёт того, что SuiteSparse позволяет создавать собственные типы, операторы и другие объекты и поддерживает работу с ними. Применительно к разреженным матрицам смежности эти алгебраические операции эквивалентны вычислениям на графах. Параллелизм SuiteSparse: GraphBLAS (начиная с версии 3) основан на OpenMP (GPU-версия находится в разработке) [**?**].

В число графовых алгоритмов, применяемых в разных практических и теоретических областях, входит алгоритм поиска минимального остовного дерева[2]. Например, компания, желающая поставлять в несколько магазинов определённый товар с одного склада, может использовать MST для расчёта кратчайших путей к каждому фирменному магазину[3]. Эта работа посвящена сравнению разных решений задачи нахождения MST, основанных на линейной алгебре.

[2]В этой работе вместо "минимальное остовное дерево" может быть использовано сокращение "MST" (от англ. "minimum spanning tree").

[3]В этом примере магазины и склад представлены в виде вершин (склад — начальная вершина), дорожные связи между ними — в виде рёбер, а вес каждого ребра равен длине соответствующего дорожного соединения.

1. Постановка задачи

Целью данной работы является реализация алгоритма поиска минимального остовного дерева на SuiteSparse. Для её достижения были поставлены следующие задачи.

- Собрать и запустить SuiteSparse, изучить её: ознакомиться с документацией по функциональности библиотеки, запустить существующие реализации и разобраться в их исходном коде.
- Реализовать алгоритм поиска минимального остовного дерева на SuiteSparse: GraphBLAS.
- Провести экспериментальное исследование реализации: осуществить замеры и анализ её производительности, сравнить их с данными исследований представлений того же алгоритма на других библиотеках высокопроизводительной обработки графов.

2. Обзор

2.1. SuiteSparse: GraphBLAS

Структуры графов на основе линейной алгебры были впервые разработаны с помощью Combinatorial BLAS[4] (CombBLAS) [**?**], расширяемой библиотеки графов на основе процессоров с распределенной памятью. CombBLAS предлагает небольшой, но мощный набор примитивов линейной алгебры, предназначенный для аналитики графов. GraphBLAS, вдохновением к созданию которого послужили BLAS, основан на идее, что для реализации многих графовых алгоритмов можно использовать четыре понятия: вектор, матрицу, операцию и полукольцо [**?**].

- **Вектор** — подмножество вершин некоторого графа.
- **Матрица** — матрица смежности некоторого графа.
- **Операция** — математическая операция, определенная в спецификации GraphBLAS. Эти операции обычно действуют на матрицы и векторы, используя элементы, определенные в терминах алгебраического полукольца.
- **Полукольцо** — объект, инкапсулирующий вычисления на вершинах и рёбрах графа.

SuiteSparse: GraphBLAS предоставляет набор методов для создания, извлечения и использования и освобождения представителей следующих девяти типов объектов [**?**]:

- **Тип** (GrB_Type). Библиотека поддерживает 11 встроенных типов данных (логический, знаковые и беззнаковые целые размером 8, 16, 32 и 64 бита, а также с плавающей запятой одинарной и двойной точности).
- **Унарный оператор** (GrB_UnaryOp): представляет из себя функцию вида z = f(x).

[4]BLAS — Basic Linear Algebra Subprograms (рус. базовые подпрограммы линейной алгебры).

- **Бинарный оператор** (GrB_BinaryOp): представляет из себя функцию вида z = f(x, y).

- **Оператор выбора** (GxB_SelectOp): используется в операции GxB_select для выбора подмножества записей из матрицы.

- **Моноид** (GrB_Monoid) — это ассоциативный, коммутативный бинарный оператор z = f (x, y) (типы x, y и z одинаковы), для которого существует нейтральный элемент. Скалярное сложение при умножении матриц заменяется моноидом.

- **Полукольцо** (GrB_Semiring): состоит из моноида и оператора «умножения». Вместе эти операторы определяют произведение матриц C = AB, где при умножении матриц моноид используется в качестве аддитивного оператора, а оператор «умножения» полукольца — в качестве скалярного умножения.

- **Дескриптор** (GrB_Descriptor): используется для настройки параметров операций GraphBLAS.

- **Вектор** (GrB_Vector) — одномерный массив значений любого типа. $2^{60} \times 1$ — размер самого большого GrB_Vector, который может быть построен. Первый элемент вектора имеет индекс 0.

- **Матрица** (GrB_Matrix) — двумерный массив значений любого типа. $2^{60} \times 2^{60}$ — размер наибольшей GrB_Matrix. Индексы строк и столбцов матрицы начинаются с 0.

Также пользователь может определять свои собственные типы данных, операторы, моноиды и полукольца. С полными возможностями SuiteSparse: GraphBLAS можно ознакомиться в руководстве пользователя [**?**].

2.2. Библиотеки высокопроизводительной обработки графов

Алгоритмы поиска минимального остовного дерева были реализованы на нескольких библиотеках. В их числе:

- **GBTL** [?] — полная реализация спецификации GraphBLAS C API[5] на C++, библиотека, содержащая наиболее распространённые графовые алгоритмы, использующие GraphBLAS.
- **LAGraph** [?] — проект, целью которого является сбор графовых алгоритмов, построенных на основе GraphBLAS, в единую структуру.

Сравнение производительности реализации алгоритма поиска MST на SuiteSparse с существующими реализациями на вышеупомянутых библиотеках представлено в разделе **??**.

Стоит отметить, что изначально планировалось представить алгоритм поиска MST на библиотеке примитивов для анализа графов на основе линейной алгебры **GraphBLAST**. GraphBLAST [?]— первая реализация стандарта GraphBLAS на графическом процессоре с открытым исходным кодом[6], предназначенная для высокопроизводительных вычислений. Однако в ходе изучения GraphBLAST в её реализации обнаружились некоторые недочёты, о которых было сообщено разработчикам библиотеки:

- Не реализовано поэлементное сложение (операция eWiseAdd) разрежённых матриц и векторов.
- Реализация алгоритма поиска максимального независимого набора работает некорректно на всех представленных в библиотеке примерах входных графов.

[5]GraphBLAS C API — интерфейс прикладного программирования, полностью определяющий типы, объекты, литералы и другие элементы связывания C с GraphBLAS.

[6]Репозиторий с открытым исходным кодом GraphBLAST: `https://github.com/gunrock/graphblast` [Accessed: 31th May, 2020].

- Конструктор матриц плохо обрабатывал графы без рёбер[7].

Помимо этого многие библиотечные тесты выдают предупреждения о некорректной работе и обнаружении ошибок. В связи с этим было принято решение о переходе на SuiteSparse.

2.3. Алгоритмы поиска минимального остовного дерева

Поскольку нахождение MST является широко распространенной задачей в теории графов, существует много последовательных алгоритмов для ее решения. Требования к производительности решений практических проблем стали причиной распараллеливания известных алгоритмов поиска MST. В основу этой работы лёг алгоритм Борувки в силу естественности его параллельного представления и преимущества над другими алгоритмами во временной сложности в случае работы с разрежёнными графами. Это жадный алгоритм поиска минимального остовного дерева (или минимального остовного леса в случае несвязного графа). В таблице **??** приведено сравнение сложности алгоритма Борувки с одним из самых известных алгоритмов поиска MST — алгоритмом Прима [**?**].

Таблица 1 — Сравнение алгоритмов поиска MST, где n — количество вершин входного графа, а m — количество рёбер.

Алгоритм	Сложность канонической реализации	Сложность реализации на основе линейной алгебры (ЛА)	Критический случай (для ЛА-реализации)
Прима	$\Theta(m + n * \log n)$	$\Theta(n^2)$	$\Theta(n)$
Борувки	$\Theta(m * \log n)$	$\Theta(m * \log n)$	$\Theta(\log^2 n)$

Кратко суть алгоритма Борувки можно изложить так:

1. Изначально каждая вершина графа — тривиальное дерево, а ребра не принадлежат никакому дереву.

[7] К настоящему моменту эта ошибка исправлена.

2. Для каждого дерева находится минимальное инцидентное ему ребро, все такие рёбра добавляются к деревьям.

3. Второй шаг повторяется, пока в графе не останется только одно дерево.

Подробнее параллельная версия алгоритма Борувки описана в разделе **??**.

3. Реализация

3.1. Алгоритм Борувки

Рассмотрим псевдокод алгоритма Борувки (Input_matrix — симметричная матрица смежности входного графа):

```
1:  function MST(Input_matrix)
2:      Result = ∅
3:      M = Input_matrix
4:      E = Matrix_nvals(M)
5:      N = Matrix_nrows(M)
6:      v1 = Vector_build([0..N-1])
7:      while E ≠ 0 do
8:          (w, v2) = Mxv(M, v1, min, (,))
9:          for all i ∈ [0..N-1] do
10:             v1[i].pointer = v2[i]
11:         for all i ∈ [0..N-1] do
12:             if v1[i].pointer == (v1[i].pointer).pointer then
13:                 super = super ∪ Min(v1[i], v1[i].pointer)
14:         Result = Result ∪ Select(M, select_edge(v1, v2))
15:         (w, v2) = Mxv(Result, v1, min, (,))
16:         Result = Matrix_build(v1, v2, w)
17:         for all v ∈ super do
18:             for all u ∈ component(v) do
19:                 u.pointer = v
20:         Select(M, diff_components())
21:         E = Matrix_nvals(M)
22:     return Result
```

Данный алгоритм повторяет следующие фазы вычислений, итеративно добавляя выбранные рёбра в MST и удаляя некоторые рёбра из дубликата исходного графа, пока все его рёбра не будут удалены [**?**]:

1. Для каждой вершины выполняется поиск инцидентного ей реб-

ра минимального веса. Если несколько рёбер имеют одинаковый вес, то среди них выбирается ребро с меньшим номером второй вершины. Другими словами, на первом шаге происходит выбор минимальной пары вида (вес, номер вершины), то есть исходная вершина меняет указатель на ту вершину, с которой она соединена этим ребром. Выбор ребра осуществляется посредством умножения входной матрицы на вектор номеров вершин с помощью моноида с выбором минимума в качестве бинарной операции "сложения" и конструктора пары вида (вес, номер вершины) в качестве операции "умножения" (строки 8-10).

2. Вершины и их рёбра образуют связные подграфы, каждый из которых представляет из себя два дерева, две вершины которых соединены циклом. В каждом цикле выбирается ребро, ведущее в вершину с меньшим номером (строка 13). Назовём её супервершиной, а все остальные вершины в подграфе — подвершинами.

3. Выбранные рёбра добавляются в MST с помощью оператора выбора (строка 14). Рёбра-дубликаты удаляются из MST(строки 15-16). Последнее осуществляется посредством умножения матрицы MST на вектор номеров вершин.

4. Каждая подвершина находит супервершину подграфа, к которому она относится, то есть меняет указатель на неё (строки 17-19).

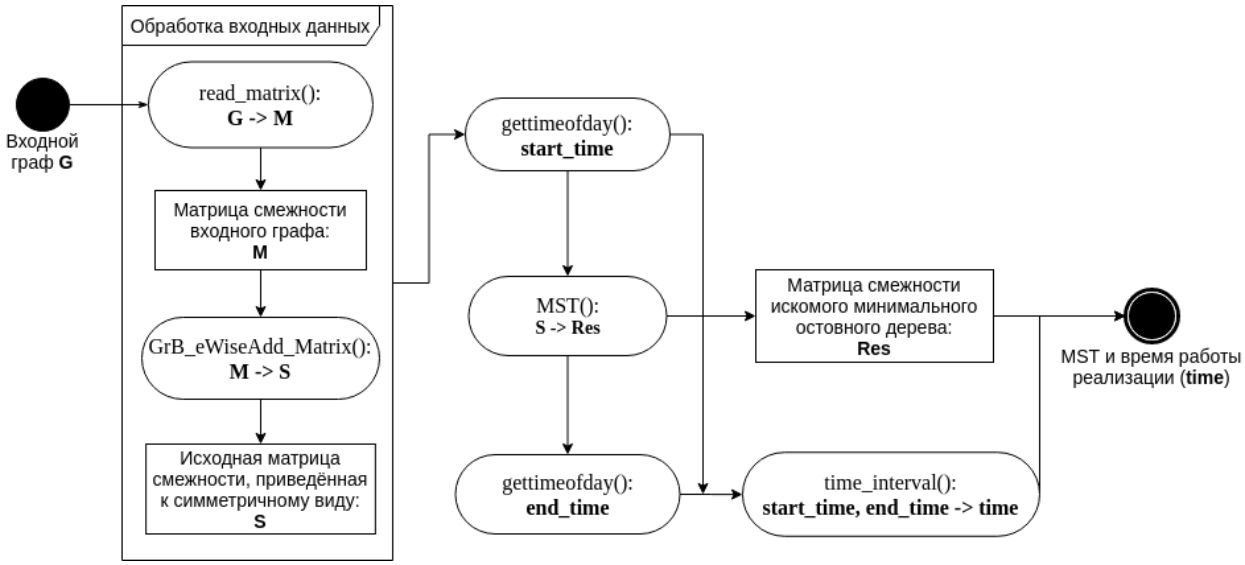
5. Рёбра, обе вершины которых указывают на одну и ту же супервершину, удаляются с помощью оператора выбора (строка 20).

3.2. Архитектура решения

Для выполнения поставленной задачи была разработана следующая архитектура (рис. **??**).

На вход программы подаётся файл, задающий исследуемый граф, то есть набор троек целых чисел, где первые два числа — номера вершин описываемого ребра, а третье число — вес этого ребра. Входной файл

Рисунок 1 — Архитектура решения



считывается функцией read_matrix(), взятой из демо SuiteSparse[8]. Функция read_matrix() составляет матрицу смежности по заданному графу. С помощью определённой в SuiteSparse операции поэлементного сложения матриц (GrB_eWiseAdd_Matrix) матрица смежности приводится к симметричному виду. Важно отметить, что реализации, с которыми сравнивалась данная программа, также приводят входные матрицы к симметричному виду. Затем матрица смежности и пустая матрица для записи результата подаются в качестве параметров функции MST(), которая представляет собой реализацию алгоритма Борувки. Результат работы MST() — искомое минимальное остовное дерево, записанное в переданную функции пустую матрицу. Матрица с MST выводится с помощью библиотечной функции GxB_Matrix_fprint. Время работы функции MST() замеряется с использованием функции gettimeofday(), описанной в заголовочном файле типов и методов работы с временем sys/time.h. Достоинством этого способа измерения является большая точность (10^{-6} сек). Результат замера времени поиска MST тоже выводится.

[8]Код функции read_matrix() можно найти здесь: `https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/master/GraphBLAS/Demo/Source` [Accessed: 31th May, 2020].

4. Экспериментальные исследования

В таблице **??** приведено сравнение реализации алгоритма поиска MST на SuiteSparse с реализациями на GBTL и LAGraph по времени работы.

Таблица составлена по результатам запуска программ на тестах, отличающихся друг от друга количеством вершин, рёбер и диапазоном весов рёбер входных графов (все тесты описаны в таблице **??**). Матрицы смежности графов из тестов 1-6 — разрежённые, из тестов 7-12 — плотные. В каждой группе тесты расположены в порядке возрастания количества вершин и рёбер для наблюдения поведения программ при увеличении нагрузки. Тесты 1, 2 и 7 запускались для исследования работы реализаций на маленьких графах. Пары тестов 3, 4 и 10, 11 имеют одинаковое количество вершин и рёбер, но разный диапазон весов рёбер, а пары 4, 5 и 8, 9, напротив, рассматриваются для изучения работы программ на графах с одинаковым диапазоном весов, но разной плотностью или количеством вершин. Тесты 6 и 12 были созданы для исследования работы реализаций на большом графе.

Таблица 2 — Описание тестов.

№ теста	Число рёбер	Число вершин	Диапазон весов
1	16	10	1
2	196	100	1
3	19996	10^4	[1, 1000]
4	19996	10^4	[1, 10^4]
5	99900	10^4	[1, 10^4]
6	9999900	10^6	[1, 10^6]
7	8	4	1
8	499500	1000	[1, 5000]
9	12497500	5000	[1, 5000]
10	40495500	9000	[1, 1000]
11	40495500	9000	[1, 10^4]
12	49995000	10^4	[1, 10^5]

Заметим, что замерялось только время поиска MST без учёта ввода входного графа и вывода искомого дерева. В таблице **??** представлены средние значения и дисперсия по пяти запускам (в секундах). Время

работы реализации алгоритма поиска MST на GBTL для тестов 6, 11 и 12 не приводится, так как на них полная работа программы (с чтением входного графа) занимает более 5 часов. Замеры проводились на ноутбуке с Ubuntu 18.04, Intel Core i5-7300HQ CPU, 2.50GHz, DDR4 64Gb RAM.

Таблица 3 — Сравнение времени работы различных реализаций (сек, число потоков: 4).

№ теста	Данная реализация		На GBTL		На LAGraph	
	Среднее значение	D	Среднее значение	D	Среднее значение	D
1	0.0013	< 0.0001	0.0008	< 0.0001	0.0009	< 0.0001
2	0.0057	< 0.0001	0.0395	< 0.0001	0.0011	< 0.0001
3	1.0661	0.0001	131.8307	0.0404	0.0224	0.0001
4	1.0650	0.0002	132.1019	0.7902	0.0232	< 0.0001
5	1.0856	0.0010	160.2648	1.9344	0.0226	< 0.0001
6	160.1692	0.2206			2.2857	0.0310
7	0.0008	< 0.0001	0.0002	< 0.0001	0.0008	< 0.0001
8	0.1303	< 0.0001	19.6429	0.0007	0.0771	< 0.0001
9	1.4475	0.0199	2270.5567	4.4393	1.3501	< 0.0001
10	1.7971	0.0051	13090.2499	91.5499	3.8008	0.0007
11	3.5753	0.0052			5.2451	0.0010
12	5.7335	0.0002			8.0566	0.0045

По таблице **??** видно, что при расширении диапазона весов вершин на тестах с разрежёнными графами нагрузка на программы не увеличивается, а с плотными — значительно растёт. Это ожидаемое наблюдение: при увеличении диапазона весов снижается вероятность повторения весов рёбер, то есть это изменение почти не влияет на результат, если в графе мало рёбер, но для плотных матриц смежности оказывается принципиальным. Важно заметить, что на GBTL представлен алгоритм Прима, а на LAGraph — алгоритм Борувки, то есть реализация алгоритма Прима ощутимо отстаёт от реализаций алгоритма Борувки (хотя для совсем маленьких задач она быстрее). Представление алгоритма Борувки на LAGraph заметно обгоняет представление на SuiteSparse почти на всех тестах, но отстаёт на больших плотных

графах, что может быть связано с особенностями обработки рёбер реализацией на LAGraph.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

- Собрана и изучена библиотека высокопроизводительной обработки графов SuiteSparse.
- Реализован алгоритм поиска MST на SuiteSparse : GraphBLAS.
- Проведено исследование производительности реализации, в том числе сравнение с представлениями алгоритмов поиска MST на других библиотеках, основанных на GraphBLAS.

Также были осуществлены сборка библиотеки GraphBLAST и изучение существующих на ней реализаций. В ходе ознакомления с GraphBLAST обнаружились недочёты в её реализации, о которых было сообщено разработчикам библиотеки.

GraphBLAST — молодая библиотека, и некоторые графовые алгоритмы на ней ещё не представлены[9]. В их число входит алгоритм поиска минимального остовного дерева, поэтому следующие задачи могут лечь в основу дальнейших исследований:

- Совершенствование реализации.
- Продолжение работы с GraphBLAST и перевод реализации на неё.

[9]Список алгоритмов, которые уже представлены на GraphBLAST или нуждаются в реализации: https://github.com/gunrock/graphblast/issues/2