Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентрованное программирование»

Выполнил
студент группы 3331506/90401 _____ О. А. Пугина

Преподаватель _____ М. С. Ананьевский

«__»_____2022 г

Санкт-Петербург
2022

Оглавление

1 ВВЕДЕНИЕ

Математические модели в виде графов широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор типовых алгоритмов обработки графов. Рассмотрению вопросов теории графов, алгоритмов моделирования, анализу и решению задач на графах посвящено достаточно много различных изданий.

Далее мы рассмотрим способ реализации алгоритмов на графах на примере задачи выделения минимального охватывающего дерева (остова) графа.

В работе будет рассмотрен алгоритм Прима.

Алгоритм Прима — это алгоритм минимального остовного дерева (дерево графа, которое имеет самый маленький допустимый вес или же сумму весов всех его рёбер), что принимает граф в качестве входных данных и находит подмножество ребер этого графа, который формирует дерево, включающее в себя каждую вершину, а также имеет минимальную сумму весов среди всех деревьев, которые могут быть сформированы из графа. Он подпадает под класс алгоритмов, называемых «жадными» алгоритмами, которые находят локальный оптимум в надежде найти глобальный оптимум.

2 ИДЕЯ АЛГОРИТМА

Задача поиска всех кратчайших путей

Исходной информацией для задачи является взвешенный граф G=(V,R), содержащий n вершин (|V|=n), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать ориентированным, т.е. если из вершины i есть ребро в вершину j, то из этого не следует наличие ребра из j в i. В случае если вершины все же соединены взаимообратными ребрами, веса,

приписанные им, могут не совпадать. Рассмотрим задачу, в которой для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

Охватывающим деревом (или остовом) неориентированного графа G называется подграф T графа G, который является деревом и содержит все вершины из G. Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, под минимально охватывающим деревом (МОД) T будем понимать охватывающее дерево минимального веса. Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием соединительных линий связи минимальной длины.

Пример взвешенного неориентированного графа и соответствующего ему минимально охватывающего дерева приведен на рис. 1.

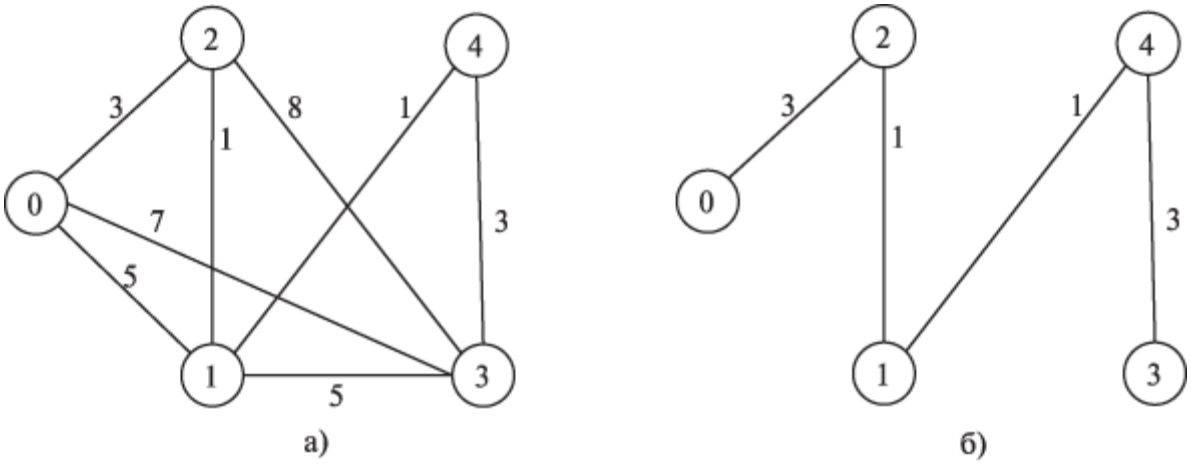

Рис. 1. Пример взвешенного неориентированного графа (а) и соответствующему ему минимально оставного дерева (б)

Дадим общее описание алгоритма решения поставленной задачи, известного под названием метода Прима.

Последовательный алгоритм Прима

Алгоритм начинает работу с произвольной вершины графа, выбираемой в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Пусть V_T есть множество вершин, уже включенных алгоритмом в МОД, а величины d_i, 1<=i<=n, характеризуют дуги минимальной длины от вершин, еще не включенных в дерево, до множества V_T, т.е.

$$\forall i \notin V_T \Rightarrow d_i = \min\{w(i,u) : u \in V_T, (i,u) \in R\}$$

(если для какой-либо вершины $i \notin V_T$ не существует ни одной дуги в V_T, значение d_i устанавливается равным ∞). В начале работы алгоритма выбирается корневая вершина МОД s и полагается V_T={s}, d_s=0.

Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:

- определяются значения величин d_i для всех вершин, еще не включенных в состав МОД;
- выбирается вершина t графа G, имеющая дугу минимального веса до множества $V_T t : d_t,\ i \notin V_T$;
- вершина t включается в V_T.

Говоря простыми словами, шаги для реализации алгоритма следующие:

- Инициализация минимального остовного дерева с произвольно выбранной вершиной.
- Нахождение всех ребер, которые соединяют дерево с новыми вершинами, нахождение минимума и добавление его в дерево.

- Повторение шага 2, пока не будет достигнут результат с минимальным остовным деревом.

Пример работы алгоритма Прима показан на рис.2.

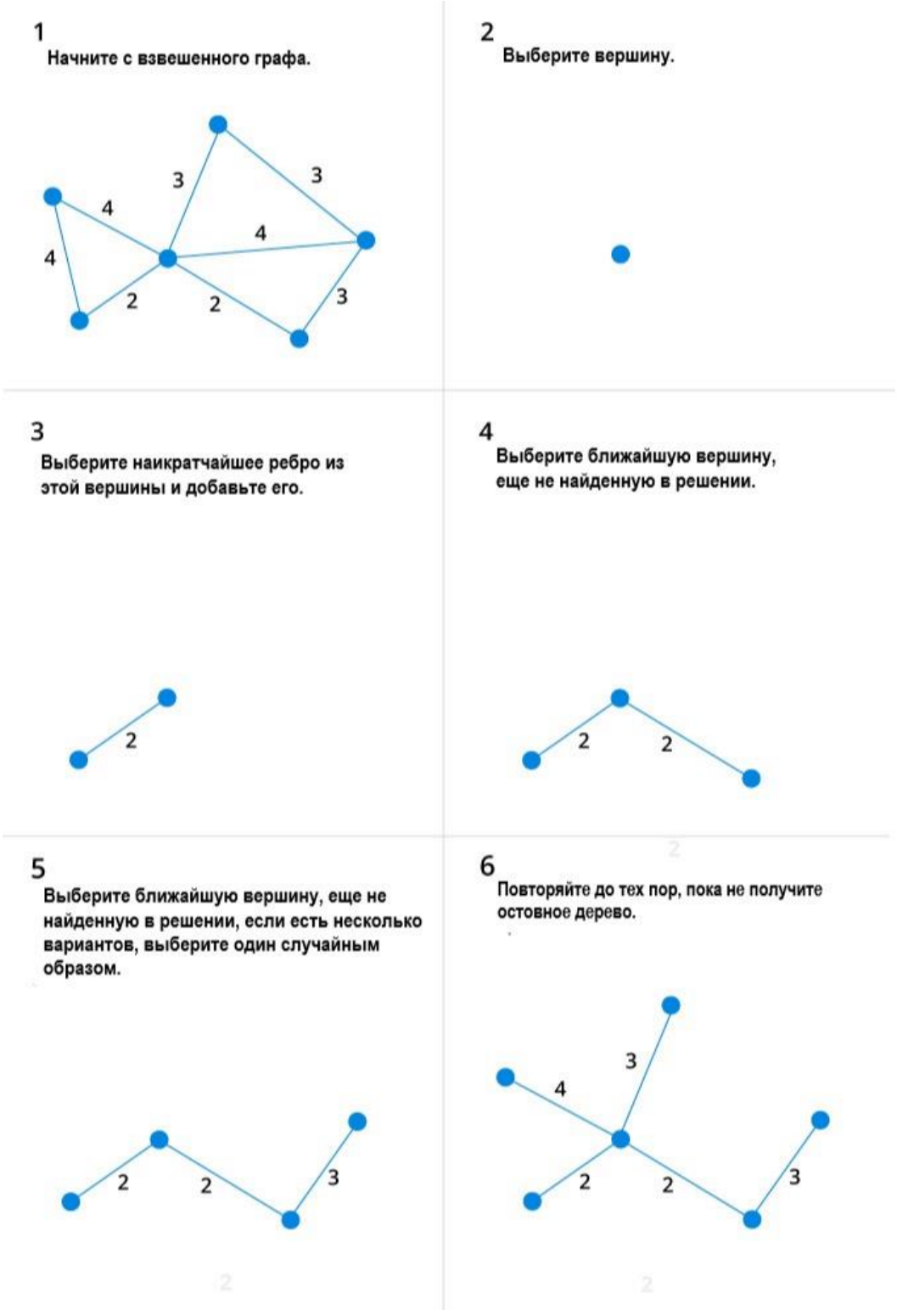

Рис. 2. Пример работы алгоритма

После выполнения n-1 итерации метода МОД будет сформировано. Вес этого дерева может быть получен при помощи выражения

$$W_T = \sum_{i=1}^{n} d_i.$$

Трудоемкость нахождения МОД характеризуется квадратичной зависимостью от числа вершин графа $O \sim n^2$.

Использование заглавной буквы O (или так называемая O-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

O (log n) — логарифмическая сложность

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

O(n2) — квадратичная сложность

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место

очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т. е. n2.

Разделение вычислений на независимые части

Оценим возможности параллельного выполнения рассмотренного алгоритма нахождения минимально охватывающего дерева.

Итерации метода должны выполняться последовательно и, тем самым, не могут быть распараллелены. С другой стороны, выполняемые на каждой итерации алгоритма действия являются независимыми и могут реализовываться одновременно. Так, например, определение величин d_i может осуществляться для каждой вершины графа в отдельности, нахождение дуги минимального веса может быть реализовано по каскадной схеме и т.д.

Распределение данных между процессорами вычислительной системы должно обеспечивать независимость перечисленных операций алгоритма Прима. В частности, это может быть реализовано, если каждая вершина графа располагается на процессоре вместе со всей связанной с вершиной информацией. Соблюдение данного принципа приводит к тому, что при равномерной загрузке каждый процессор p_j, 1<=j<=p, должен содержать:

- набор вершин

$$V_j = \{\nu_{i_j+1}, \nu_{i_j+2}, \ldots, \nu_{i_j+k}\},\ i_j = k \cdot (j-1), k = \lceil n/p \rceil;$$

- соответствующий этому набору блок из k величин

$$\Delta_j = \{d_{i_j+1}, d_{i_j+2}, \ldots, d_{i_j+k}\};$$

- вертикальную полосу матрицы смежности графа G из k соседних столбцов

$$A_j = \{\alpha_{i_j+1}, \alpha_{i_j+2}, \ldots, \alpha_{i_j+k}\} (\alpha_s \text{ есть } s\text{-й столбец матрицы } A);$$

- общую часть набора V_j и формируемого в процессе вычислений множества вершин V_T.

Можем заключить, что базовой подзадачей в параллельном алгоритме Прима может служить процедура вычисления блока значений Δ_j для вершин V_j матрицы смежности A графа G.

Выделение информационных зависимостей

С учетом выбора базовых подзадач общая схема параллельного выполнения алгоритма Прима будет состоять в следующем:

- определяется вершина t графа G, имеющая дугу минимального веса до множества V_T. Для выбора такой вершины необходимо осуществить поиск минимума в наборах величин d_i, имеющихся на каждом из процессоров, и выполнить сборку полученных значений на одном из процессоров;
- номер выбранной вершины для включения в охватывающее дерево передается всем процессорам;
- обновляются наборы величин d_i с учетом добавления новой вершины.

Таким образом, в ходе параллельных вычислений между процессорами выполняются два типа информационных взаимодействий: сбор данных от всех процессоров на одном из процессоров и передача сообщений от одного процессора всем процессорам вычислительной системы.

Масштабирование и распределение подзадач по процессорам

По определению количество базовых подзадач всегда соответствует числу имеющихся процессоров, и, тем самым, проблема масштабирования для параллельного алгоритма не возникает. Распределение подзадач между

процессорами должно учитывать характер выполняемых в алгоритме Прима коммуникационных операций. Для оптимальной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должна обеспечивать эффективное представление в виде гиперкуба или полного графа.

3 ЭФФЕКТИВНОСТЬ

Анализ эффективности параллельных вычислений

Общий анализ сложности параллельного алгоритма Прима для нахождения минимального охватывающего дерева дает идеальные показатели эффективности параллельных вычислений:

$$S_p = \frac{n^2}{(n^2/p)} = p \quad \text{и} \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1.$$

При этом следует отметить, что в ходе параллельных вычислений идеальная балансировка вычислительной нагрузки процессоров может быть нарушена. В зависимости от вида исходного графа G количество выбранных вершин в охватывающем дереве на разных процессорах может оказаться различным, и распределение вычислений между процессорами станет неравномерным (вплоть до отсутствия вычислительной нагрузки на отдельных процессорах). Однако такие предельные ситуации нарушения балансировки в общем случае возникают достаточно редко, а организация динамического перераспределения вычислительной нагрузки между процессорами в ходе вычислений является интересной, но одновременно и очень сложной задачей.

Для уточнения полученных показателей эффективности параллельных вычислений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

При выполнении вычислений на отдельной итерации параллельного алгоритма Прима каждый процессор определяет номер ближайшей вершины из V_j до охватывающего дерева и осуществляет корректировку расстояний d_i после расширения МОД. Количество выполняемых операций в каждой из этих вычислительных процедур ограничивается сверху числом вершин, имеющихся на процессорах, т.е. величиной $\lceil n/p \rceil$. Как результат, с учетом общего количества итераций n время выполнения вычислительных операций параллельного алгоритма Прима может быть оценено при помощи соотношения:

$$T_p(calc) = 2n\lceil n/p \rceil \cdot \tau$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Операция сбора данных от всех процессоров на одном из процессоров может быть произведена за $\lceil log_2 p \rceil$ итераций, при этом общая оценка длительности выполнения передачи данных определяется выражением:

$$T_p^1(comm) = n(\alpha \log_2 p + 3w(p-1)/\beta),$$

где α – латентность сети передачи данных, β – пропускная способность сети, а w есть размер одного пересылаемого элемента данных в байтах (коэффициент 3 в выражении соответствует числу передаваемых значений между процессорами – длина минимальной дуги и номера двух вершин, которые соединяются этой дугой).

Коммуникационная операция передачи данных от одного процессора всем процессорам вычислительной системы также может быть выполнена за $\lceil log_2 p \rceil$ итераций при общей оценке времени выполнения вида:

$$T_p^2(comm) = n \log_2 p(\alpha + w/\beta)$$

С учетом всех полученных соотношений общее время выполнения параллельного алгоритма Прима составляет:

$$T_p = 2n\lceil n/p \rceil \cdot \tau + n(\alpha \cdot \log_2 p + 3w(p-1)/\beta + log_2 p(\alpha + w/\beta)).$$

4 РЕЗУЛЬТАТЫ РАБОТЫ АЛГОРИТМА

Результаты вычислительных экспериментов

Для анализа работы алгоритма будем опираться на данные вычислительных экспериментов для оценки эффективности параллельного алгоритма Прима, которые проводились на вычислительном кластере Нижегородского университета на базе процессоров Intel Xeon 4 *EM64T*, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft *Compute Cluster* Server.

Для оценки длительности τ базовой скалярной операции проводилось решение задачи нахождения минимального охватывающего дерева при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 4,76 нсек. Все вычисления производились над числовыми значениями типа int, размер которого на данной платформе равен 4 байта (следовательно, w=4).

Результаты вычислительных экспериментов даны в таблице 1 и рис.3. Эксперименты проводились с использованием двух, четырех и восьми процессоров. Время указано в секундах.

Таблица 1. Результаты вычислительных экспериментов для параллельного алгоритма Прима

Кол-во вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,044	0,248	0,176	0,932	0,047	1,574	0,028
2000	0,208	0,684	0,304	1,800	0,115	2,159	0,096
3000	0,485	1,403	0,346	2,214	0,219	3,195	0,152
4000	0,873	1,946	0,622	3,324	0,263	5,431	0,161
5000	1,432	2,665	0,736	2,933	0,488	4,119	0,348
6000	2,189	2,900	0,821	4,291	0,510	7,737	0,283
7000	3,042	3,236	0,940	6,327	0,481	8,825	0,345
8000	4,150	4,462	0,930	6,993	0,593	10,390	0,399
9000	5,622	5,834	0,964	7,475	0,752	10,764	0,522
10000	7,512	6,990	1,075	8,597	0,874	14,095	0,533

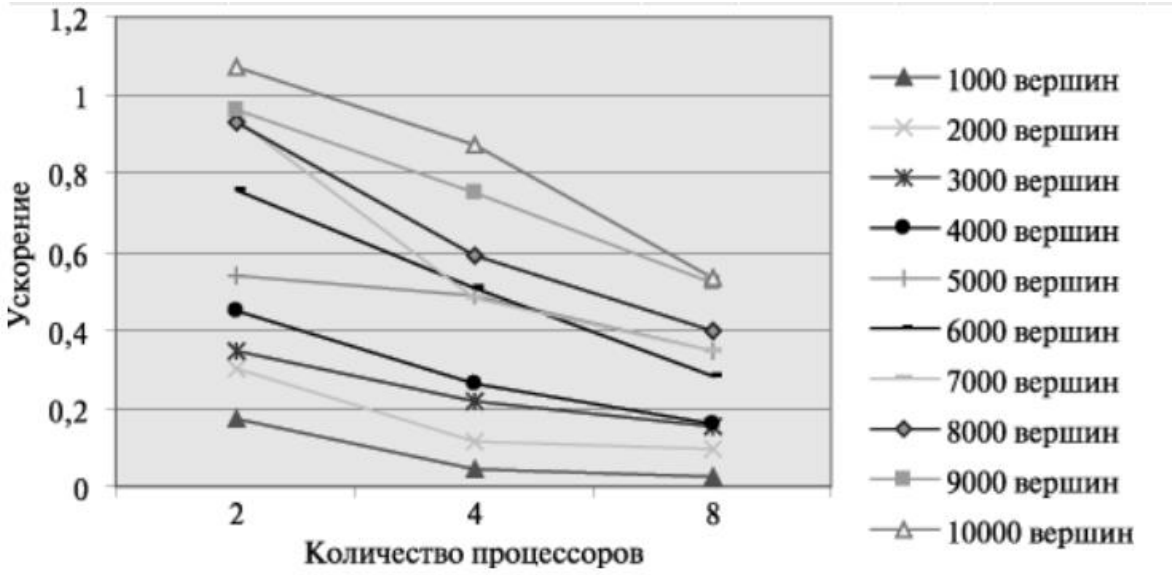

Рис. 3. Графики зависимости ускорения параллельного алгоритма Прима от числа используемых процессоров при различном количестве вершин в модели

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p приведено в таблице 2 и на рис. 4.

Таблица 2. Сравнение экспериментального и теоретического времени работы алгоритма Прима

Количество вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
	T_1^*	T_2	T_2^*	T_4	T_4^*	T_8	T_8^*
1000	0,044	0,405	0,248	0,804	0,932	1,205	1,574
2000	0,208	0,820	0,684	1,613	1,800	2,412	2,159
3000	0,485	1,245	1,403	2,426	2,214	3,622	3,195
4000	0,873	1,679	1,946	3,245	3,324	4,834	5,431
5000	1,432	2,122	2,665	4,068	2,933	6,048	4,119
6000	2,189	2,575	2,900	4,896	4,291	7,265	7,737
7000	3,042	3,038	3,236	5,728	6,327	8,484	8,825
8000	4,150	3,510	4,462	6,566	6,993	9,705	10,390
9000	5,622	3,991	5,834	7,408	7,475	10,929	10,764
10000	7,512	4,482	6,990	8,255	8,597	12,155	14,095

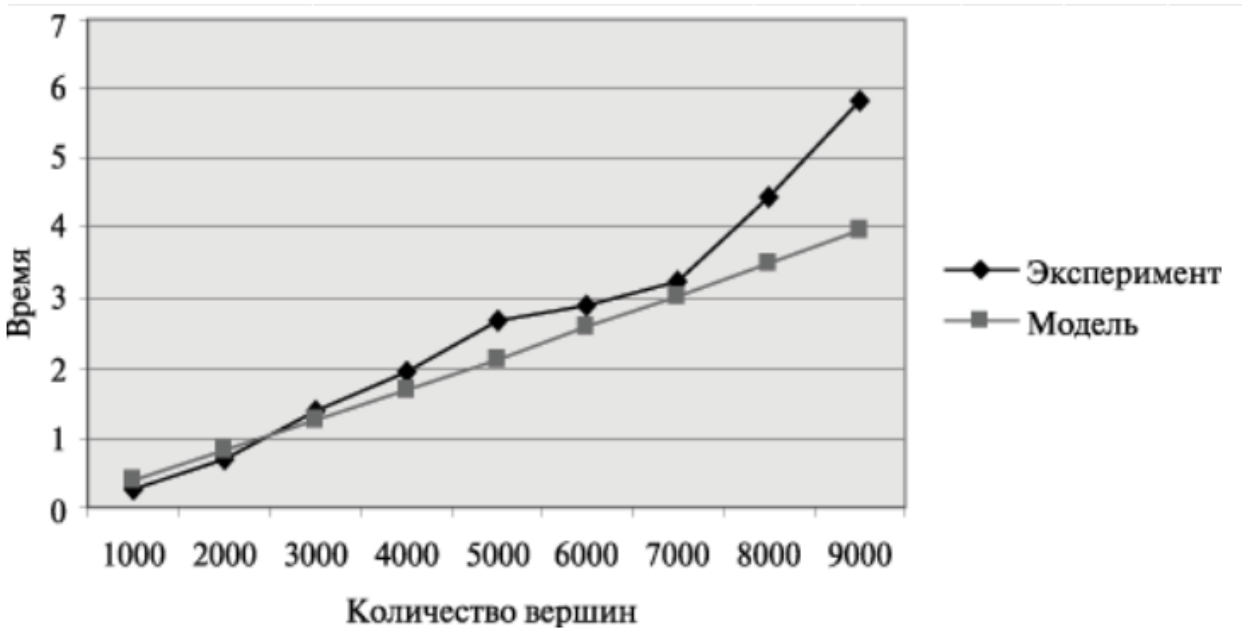

Рис. 4. Графики экспериментально установленного времени работы параллельного алгоритма Прима и теоретической оценки в зависимости от количества вершин в модели при использовании двух процессоров

Как можно заметить из табл. 2 и рис. 4, теоретические оценки определяют время выполнения алгоритма Прима с достаточно высокой погрешностью. Причина такого расхождения может состоять в том, что модель Хокни менее точна при оценке времени передачи сообщений с небольшим объемом передаваемых данных.

Таким образом, алгоритм Прима применяется для таких задач, связанных с графами, как поиск кратчайших путей между всеми парами пунктов назначения и задач выделения минимального охватывающего дерева (остова) графа.

5 СПИСОК ЛИТЕРАТУРЫ

1) https://ru.wikipedia.org/wiki/Алгоритм_Прима
2) https://intuit.ru/studies/courses/1156/190/lecture/4960?page=4
3) https://habr.com/ru/post/569444/
4) https://www.youtube.com/watch?v=vPHUm874EoA

6 ПРИЛОЖЕНИЕ

Приложение 1

Реализация алгоритма Прима в С ++

```
#include <iostream>
#include <vector>
using namespace std;

class prims
{
private:
    int no_of_edges, no_of_nodes;
    vector <vector<long long int>> graph;
    vector<int> visited;
    long long int INF;
public:

    void add_edge(int i, int j, int cost);
    void output();
    void spanningtree();
    prims(int length, int nEdges, int nNodes)
    {
        this->no_of_edges =nEdges;
        this->no_of_nodes = nNodes;
        this->graph.resize(length);
        for (int i = 0; i < length; i++)
        {
            this->visited.push_back(0);
            this->graph[i].resize(length);
            for(int j = 0; j < length; j++)
            {
                this->graph[i].push_back(0);
            }
        }
    }

};

void prims::add_edge(int i, int j, int cost)
{
    this->graph[i][j]=this->graph[j][i]=cost;
}

void prims::output()
{
    for (int i = 0; i < this->no_of_nodes; i++)
    {
        cout << endl;
        for (int j = 0; j < this->no_of_nodes; j++)
        { if (this->graph[i][j]!= INF){
                cout.width(4);
                cout << this->graph[i][j];
            }else{
                cout.width(4);
                cout << 0;
```

```
            }
        }
    }
}

void prims::spanningtree()
{
    this->INF = 9999999999;
    int n = this->no_of_nodes;
    for(int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            if (this->graph[i][j]==0){
                graph[i][j]= INF;
            }
        }
    }
    vector<bool> used (n);
    vector<int> min_e (n, INF), sel_e (n, -1);
    min_e[0] = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j=0; j<n; ++j)
            if (!used[j] && (v == -1 || min_e[j] < min_e[v]))
                v = j;
        if (min_e[v] == INF) {
            cout << "No MST!";
            exit(0);
        }

        used[v] = true;
        if (sel_e[v] != -1)
            cout << v << " " << sel_e[v] << endl;

        for (int to=0; to<n; ++to)
            if (this->graph[v][to] < min_e[to]) {
                min_e[to] = graph[v][to];
                sel_e[to] = v;
            }
    }


    long long int total = 0;
    cout<<endl;
    cout<<"Minimum distance peth is ";
    for (int i = 0; i < n - 1; i++)
    {
        cout << min_e[i] << " ";
        total = total + min_e[i];

    }
    cout << endl << "Total path cost is " << total;

}

int main()
{
```

```
    prims obj {200, 20, 20};
    for (int i = 0; i<20; i++){
        int j=0;
        int k=0;
        while(j==k){
            j = rand() % 20;
            k = rand() % 20;
        }
        obj.add_edge(j, k, rand());
    }
    obj.spanningtree();
    obj.output();

    return 0;
}
```

Результат работы кода:

```
Minimum distance peth is 1842 21726 5436 5705 3902 1869 2995 6868 12316 12382 15724 16827 19718 28703 1410065407 1410065
407 1410065407 1410065407 1410065407
Total path cost is 7050483048
   0   0   0   0   0   0   0   0   015724   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   016827   06334   0   0   02995   0   0   0   019718   0   0   0
   0   0   0   05705   0   05436   0   0   030333   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   09741
   0   05705   0   0   0   032662   0   0   03902   015141   0   0   0   0   0   0
   016827   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   063345436   032662   0   0   0   0   0   0   026299   0   021726   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
15724   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   012316   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   01842   0   0   0   0
   0299530333   03902   0   0   0   0   0   0   0   06868   0   0   0   01869   0
   0   0   0   0   0   0   026299   0   0   0   0   012382   0   0   0   0   0   0
   0   0   0   015141   0   0   0   0   0   0686812382   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   028703   0   0   0   0
   0   0   0   0   0   0   021726   0   01842   0   0   028703   0   0   0   0   0
   019718   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   012316   01869   0   0   0   0   0   0   0   0
   0   0   09741   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
Process finished with exit code 0
```