МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

01.03.02 ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

БАКАЛАВРСКАЯ РАБОТА

на тему **Сравнительный анализ параллельных алгоритмов поиска кратчайших путей на графах**

Студент Р.Р. Идрисов _____

Руководитель М.А. Тренина _____

Консультант по аннотации Н.В. Ященко _____

Допустить к защите
Заведующий кафедрой к.т.н., доцент А.В. Очеповский _____

«_____» _____ 20___ г.

Тольятти 2017

Аннотация

Темой данной бакалаврской работы является «Сравнительный анализ параллельных алгоритмов поиска кратчайших путей на графах».

Работа выполнена студентом Тольяттинского государственного университета, института математики, физики и информационных технологий, группы ПМИб-1301, Идрисова Руслана Ринатовича.

Объект работы: параллельные алгоритмы поиска кратчайших путей на графах.

Предмет исследования: сравнительный анализ параллельных реализаций алгоритмов поиска кратчайших путей на графах на языке программирования C++.

Цель работы: реализовать параллельные версии алгоритмов поиска кратчайших путей на графах, выявить их достоинства и недостатки.

Для достижения цели работы необходимо решить следующие задачи:

1) Рассмотреть основные понятие теории графов и алгоритмы решения классической задачи на графах.

2) Реализовать параллельные версии распространенных и востребованных алгоритмов на языке C++.

3) Выявить достоинства, каждой из реализаций алгоритмов.

Отчет состоит из введения, трех глав и заключения.

В первой главе представлены теоретические аспекты теории графов, основные алгоритмы для решения задач с графами.

Во второй главе описана реализация параллельных алгоритмов Флойда и Джонсона на языке C++, и тестирование их работы.

В третьей главе производится сравнительный анализ реализаций.

Бакалаврская работа представлена на 47 страницах, включает 8 иллюстраций, 2 приложения, список используемой литературы содержит 20 источников.

ABSTRACT

The title of the given graduation work is «Comparative Analysis of Parallel Algorithms for Searching the Shortest Paths in Graphs».

This graduation work deals with the search for an effective solution of the classical problem of graph theory.

The aim of the work is the implement parallel algorithms for finding shortest paths in graphs, to identify their advantages and disadvantages.

The subject of this work is comparative analysis of implementations of algorithms for solving the finding the shortest path in a graph problem in the programming language C++.

The shortest path search is an important task that arises in different fields of science and technology: in the economy (with traffic optimization), in robotics (when searching for the optimal route by the robot), in computer games (when moving troops in a labyrinth in real-time strategies) etc.

We study in details the theory of graphs and algorithms of Floyd and Johnson that are suitable for solving our problem. Successive versions of these algorithms were developed and modified to parallel ones.

Comparison of algorithms in this graduation work is based on the results of simulation experiments for various algorithms and configurations. This simulation allows avoiding long-term field experiments and ensuring reproducibility of the results.

In conclusion we'd like to stress with the growth of multiprocessor architectures, we got a powerful tool for efficient calculation of the required distances - we were able to run these algorithms on several computational cores.

However, effective use of system resources does not have a sufficient variety of solutions.

The graduation project consists of an explanatory note on 47 pages, including 8 figures, the list of 20 references including 10 foreign sources and 2 appendices.

Оглавление

Введение

Алгоритмы поиска кратчайших путей на графах нашли свое применение в различных областях и сферах деятельности человека. Такие алгоритмы используются в картографических сервисах, при построении пути GPS-навигатора, для представления и анализа дорожной сети и во многих других областях.

При проектировании компьютерных сетей, телевизионных линий, трубопроводов и строительстве дорог необходимо минимизировать затраты на прокладку коммуникаций. Прежде всего, целесообразно выбрать минимальный по длине маршрут прокладки коммуникаций. Например, необходимо соединить телефонным или оптоволоконным кабелем несколько зданий, расстояния между которыми различны. Возникает задача определения маршрута прокладки кабеля минимальной длины, но при этом к каждому зданию. Для решения таких задач часто используют теорию графов.

В настоящее время существует большое число алгоритмов и подходов, которые решают данные задачи. Кроме того, в большинстве своем алгоритмы можно логически разделить на два класса – алгоритмы поиска кратчайшего расстояния от одной вершины до всех остальных и алгоритмы поиска кратчайших расстоянии между каждой парой вершин. Из первого класса самыми яркими представителями являются различные модификации алгоритмов Дейкстры и Беллмана-Форда. Для решения задач второго класса часто используются алгоритмы Флойда-Уоршелла и алгоритм Джонсона.

Наиболее распространённой библиотекой, использующейся для решения этих задач, является BGL (Boost Graph Library). Также для работы с графами используется библиотека LEDA. Однако эти библиотеки включают в себя только последовательные версии алгоритмов для решения задач на графах.

Актуальность:

С ростом многопроцессорных архитектур мы получили мощный инструмент для эффективного расчета искомых расстояний - мы получили возможность запускать эти алгоритмы на нескольких вычислительных ядрах. При этом в контексте с параллельными алгоритмами на графах встал вопрос об эффективном использовании ресурсов системы. Эта задача, однако, не имеет такого высокого разнообразия решений, как в случае однопоточного алгоритма. Существующие же решения довольно специфичны и не всегда работают эффективно на всех графовых структурах.

Объект работы: параллельный алгоритм нахождения кратчайших путей в графе.

Предмет исследования: сравнительный анализ параллельных реализаций алгоритмов нахождения кратчайшего пути на графе на языке программирования C++.

Цель работы: реализовать параллельные версии алгоритмов нахождения кратчайших путей на графе, выявить их достоинства и недостатки.

Для достижения цели работы необходимо решить следующие задачи:

1) Рассмотреть основные понятие теории графов и алгоритмы решения классической задачи на графах.

2) Реализовать параллельные версии распространенных и востребованных алгоритмов на языке C++.

3) Проверить работоспособность и корректность работы программ.

4) Выявить достоинства, каждой из реализаций алгоритмов.

Отчет состоит из введения, трех глав и заключения.

В первой главе представлены теоретические аспекты теории графов, основные алгоритмы для решения задач с графами.

Во второй главе описана реализация параллельных алгоритмов Флойда и Джонсона на языке C++, и тестирование их работы.

В третьей главе производится сравнительный анализ реализаций.

Глава 1. Поиск кратчайших расстояний в графах

1.1 Основные понятия теории графов

Графом G(V, E) называется совокупность двух множеств — непустого множества V (множества вершин) и множества E двухэлементных подмножеств множества V (E — множество рёбер). Связи между элементами изображаются на графе линиями. Если линия направленная, то она называется дугой. Если нет, то это ребро.

Последовательность $v_1e_1v_2e_2v_3\ldots e_kv_{k+1}$, (где $k\geq1$, $v_i\in V$, $i=1,...k+1$, $e_i\in X$, $j=1,...,k$), в которой чередуются вершины и рёбра (дуги) и для каждого $j=1,...,k$ ребро (дуга) e_j имеет вид $\{v_j, v_{j+1}\}$ (для ориентированного графа (v_j, v_{j+1})), называется маршрутом (длиной пути), соединяющим вершины v_1 и v_{k+1} (путём из v_1 в v_{k+1}).

Неориентированный граф G(V, E) состоит из конечного множества вершин V и множества рёбер E. В отличие от ориентированного графа, здесь ребро (v, w) соответствует неупорядоченной паре вершин: если (v, w) — неориентированное ребро, то (v, w) = (w, v). Неориентированный граф без петель называется полным, если каждая его вершина смежна со всеми остальными вершинами. Многое из терминологии ориентированных графов применимо к неориентированным графам.

Ориентированный граф (или сокращённо орграф) — граф, в котором связи изображены дугами. Дуга представима в виде упорядоченной пары вершин (v, w), где вершина v называется началом, а w — концом дуги. Дугу (v, w) часто записывают как v→w.

Цикл — это замкнутая цепь в неориентированном графе, в которой все вершины различны, длинной не менее 1, которая начинается и заканчивается в одной и той же вершине. Неориентированный граф G называется связным,

если каждая пара вершин vi и vj в графе связана цепью. Любой максимальный связный подграф (то есть не содержащийся в других связных подграфах) графа G называется компонентой связности. Несвязный граф имеет, по крайней мере, две компоненты связности.

Если каждой дуге (ребру) графа приписано неотрицательное число (вес), то он называется взвешенным графом (взвешенная сеть). Вес графа (сети) равен сумме весов его рёбер. Содержательно вес ребра может означать стоимость передачи информации между узлами, соответствующими его концам, величину потока передаваемой информации по этой дуге или еѐ пропускную способность и т.д. задач. Взвешенные графы применяются для решения различных транспортных задач.

Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа. Для наглядного представления графа используют диаграммы (см. рисунок 1.1), а для математических расчётов удобнее использовать представление графа G (V, E) в форме матрицы смежности.

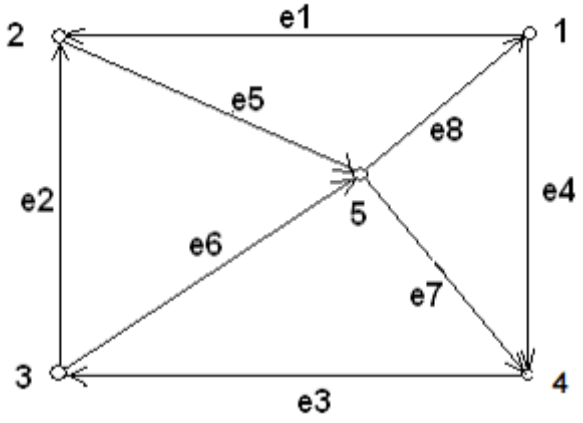

Рис.1.1 – Граф в форме диаграммы

Матрицу смежности можно представить в виде таблицы, строки и столбцы которой соответствуют номерам вершин графа. Если вершины

смежны, то элементы матрицы смежности равны 1, если вершины не смежны, то элементы матрицы равны 0. Диагональные элементы матрицы равны 0, т.к. вершины сами с собой не смежны (их соединяет ребро).

1.2 Задачи поиска кратчайших путей на графах

Поиск кратчайшего пути – важная задача, возникающая в разных областях науки и техники: в экономике (при оптимизации перевозок), в робототехнике (при поиске роботом оптимального маршрута), в компьютерных играх (при перемещении отрядов в лабиринте в стратегиях реального времени) и так далее. На этапе моделирования традиционно используются графы, вершины которых соответствуют пунктам назначения, а ребра – прямым маршрутам из одного пункта в другой. Часто ребру ставится в соответствие число, характеризующее условную «стоимость» перемещения. Выделяют четыре задачи поиска кратчайших путей на графе:

- между двумя вершинами (задача 1);
- от заданной вершины ко всем вершинам (задача 2);
- от всех вершин до заданной вершины (задача 3);
- от всех вершин ко всем вершинам (задача 4).

Возможный вариант решения состоит в том, чтобы решить четвертую задачу, сохранить результаты и использовать ее далее для решения задач 1–3. При этом поиск решения задачи 4 занимает существенное время и выполняется заранее, но задачи 1–3 решаются в режиме реального времени (к примеру, можно использовать таблицы с вычислимым входом для хранения оптимальных путей).

В работе рассматривается задача поиска кратчайших путей от всех вершин ко всем вершинам. Выполняется программная реализация и распараллеливания классических алгоритмов Джонсона и Флойда.

1.3. Методы решения задачи поиска кратчайшего расстояния от фиксированной вершины

В контексте решения задачи поиска кратчайших расстояний от одной вершины до всех остальных существует два наиболее распространенных решения алгоритм Дейкстры и алгоритм Беллмана-Форда.

1.3.1 Алгоритм Дейкстры

Сложность алгоритма Дейкстры зависит от способа нахождения вершины, а также способа хранения множества не посещённых вершин и способа обновления длин.

Если для представления графа использовать матрицу смежности, то время выполнения этого алгоритма имеет порядок $O(n^2)$, где n – количество вершин графа.

Вначале расстояние для начальной вершины полагается равным нулю, а расстояния до всех остальных понимаются бесконечными. Массив флагов, обозначающих то, пройдена ли вершина, заполняется нулями. Затем на каждом шаге цикла ищется вершина с минимальным расстоянием до изначальной и флагом равным нулю. Для неё устанавливается флаг и проверяются все соседние вершины. Если рассчитанное ранее расстояние от исходной вершины до проверяемой больше, чем сумма расстояния до текущей вершины и длины ребра от неё до проверяемой вершины, то расстояние до проверяемой вершины приравниваем к расстоянию до текущей + ребро от текущей до проверяемой. Цикл завершается, когда флаги всех вершин становятся равны 1, либо когда расстояние до всех вершин с флагом 0 бесконечно. Последний случай возможен тогда и только тогда, когда граф несвязный.

Существует множество модификаций алгоритма Дейкстры, которые в зависимости от используемой структуры данных работают $O(V^2 + E)$, $O(Elog(V))$ или $O(Vlog(V) + E)$. Однако такие алгоритмы не работают на графах с отрицательным весом ребер.

1.3.2. Алгоритм Беллмана-Форда

Классический алгоритм Беллмана-Форда работает на графах с произвольных весомы ребер. Важным его недостатком является асимптотика -O(V E). Однако за счет возможности ранней остановки алгоритма и различных оптимизаций на практике на некоторых графовых структурах он может давать результаты, лучшие чем алгоритм Дейкстры. Также известны специализированные алгоритмы, такие как алгоритм A* и D*, которые оперируют большими специализированными графами и используют ряд эвристик для поиска расстояний.

В основе алгоритма лежит идея динамического программирования. После k итерации алгоритма утверждается, что будут корректно посчитаны и обработаны пути длиной не более k. И после V итерации расстояние до каждой из вершин посчитано корректно.

1.4. Методы решения задачи поиска кратчайших расстояний между каждой парой вершин

В данном разделе представлен обзор существующих решений задачи поиска кратчайших расстояний между каждой парой вершин – алгоритм Флойда и алгоритм Джонсона, в основе которого лежат рассмотренные ранее алгоритм Дейкстры и алгоритм Беллмана.

1.4.1 Алгоритм Флойда

Одним из наиболее известных алгоритмов, который применяется для решения данной задачи является алгоритм Флойда. Этот алгоритм использует подход динамического программирования и выполняется за $O(V^3)$. Основная идея состоит в обновлений пути между двумя текущими вершинами выбором некоторой вершины, через которую может пройти потенциальный кратчайший путь.

В основе алгоритма поиска кратчайших путей используется структура кратчайших путей. Пусть дан граф G = (V, E), где V – множество вершин графа, а E – множество ребер. Рассмотрим путь $p = (v_1, v_2,\ldots,v_i\ldots, v_i\ldots,v_{k-1})$. Пусть данный путь оптимальный. Тогда оптимальными будут так же пути $p_{1,i} = (v_1, \ldots , v_i)$, $p_{i,j} = (v_i, \ldots , v_j)$, и $p_{j,k-1} = (v_j, \ldots , v_{k-1})$. Этот факт легко обосновать на основании того, что стоимость пути складывается из суммы весов его частей.

Пусть к пути p необходимо добавить вершину u_k. Тогда существует два варианта:

1 Стоимость пути $p_{i,j}$ меньше стоимости пути $p'_{i,j} = (v_i, v_k, v_j)$, тогда в этом случае оптимальный путь не изменится.

2 Стоимость пути $p_{i,j}$ больше стоимости пути $p'_{i,j} = (v_i, v_k, v_j)$, тогда оптимальным путем будет $p = (v_0, v_1,\ldots,v_i,v_k, v_i\ldots,v_{k-1})$. Данный факт опять же следует из того условия, что вес пути складывается из стоимости путь $p'_{i,j}$ и оставшихся частей.

Используя данные утверждения, алгоритм можно построить следующим образом. Пусть $d^k_{i,j}$ – стоимость оптимального пути из вершины i в вершину j, с проверенной возможностью прохождения через вершину k. При k=0 значения $d^0_{i,j} = w_{i,j}$ совпадают с весом перехода из вершины i в

вершину j. Если ребро $e_{j,i}$ отсутствует, то $d^0_{i,j} = \infty$. Далее рекуррентное соотношение можно определить следующим образом:

$$d^k_{i,j} = \begin{cases} w_{i,j} & , если \quad k = 0 \\ \min(d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j} & , если \quad k > 0 \end{cases}$$

Используя данное рекуррентное соотношение, можно найти матрицу $D^n_{i,j} = (d^n_{i,j})$, содержащую веса кратчайших путей для всех пар вершин $(i,j) \in V$.

Для поиска кратчайших путей часто вычисляют матрицу предшествования П. Для этого существуют простые рекуррентные соотношения:

$$\pi^0_{i,j} = \begin{cases} NULL & , если \quad w_{i,j} = \infty \\ i & , i = j \\ i & , если \quad w_{i,j} < \infty \end{cases}$$

$$\pi^k_{i,j} = \begin{cases} \pi^{k-1}_{i,j} & , если \quad d^{k-1}_{i,j} \le d^{k-1}_{i,k} + d^{k-1}_{k,j} \\ \pi^{k-1}_{k,j} & , если \quad d^{k-1}_{i,j} > d^{k-1}_{i,k} + d^{k-1}_{k,j} \end{cases}$$

Приведём псевдокод для алгоритма Флойда. Пусть граф задан матрицей смежности A[n][n]. Функция min, возвращающая минимум из 2 чисел, должна учитывать способ указания в матрице смежности несуществующих дуг графа. В данной реализации длина несуществующих дуг полагается равной бесконечности.

```
void Floyd (A)
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        if (A[i][k] ≠ ∞)
        for (j = 0; j < n; j++)
            if (A[k][j] ≠ ∞)
```

```
                A[i][j]   =   min(A[i][j],   A[i][k]   +
        A[k][j]);
```

1.4.2 Алгоритм Джонсона

Алгоритм Джонсона позволяет найти пути между всеми парами вершин за время O(Vlog(V) + VE). Если данный алгоритм применять для разреженных графов, таких как карты дорог, то алгоритм Джонсона работает значительно быстрее алгоритма возведения матриц в степень количества вершин и алгоритма Флойда.

Идея алгоритма Джонсона состоит в применении алгоритма Дейкстры для каждой вершины. Алгоритм Дейкстры позволяет быстро найти расстояние от источника до всех вершин, но у него есть существенное ограничение – алгоритм применим только для графов с неотрицательными весами ребер.

Чтобы свести граф с отрицательными весами ребер (но не содержащий циклов с отрицательным весом) к графу без ребер с отрицательными весами применяется алгоритм Беллмана-Форда.

Рассмотрим алгоритм более детально. Чтобы перейти от графа с отрицательными весами к графу без отрицательных весов, можно ввести следующую весовую функцию:

$$w'(u,v) = w(u,v) + h(u) - h(v)$$

Данная замена обладает одним замечательным свойством. Пусть дан путь $p = (v_1, v_2, \ldots, v_k)$, тогда вес пути до и после замены связаны следующим соотношением:

$$\partial(v_1, v_k) = \partial'(v_1, v_k) - h(v_k) + h(v_1)$$

Используя данное свойство, можно найти кратчайшие пути для графа с весами w' и затем вернуться к исходному графу. Для применения алгоритма Дейкстры необходимо найти такую функцию h, чтобы веса ребер стали положительными и при этом кратчайшие пути не изменились.

Теоретически показано, что для поиска функции h применима следующая процедура. Пусть дан исходный граф G=(V,E). Построим новый граф G'=(V',E') следующим образом:

1. $V' = V \cup \{s\}$
2. $E' = E \cup \{(s,v), \text{для всех } v \in V\}$
3. $w(s,v) = 0$, для всех $v \in V$

В полученном графе необходимо найти веса кратчайших путей из вершины во все. Так как сконструированный граф все еще содержит ребра с отрицательными весами, то алгоритм Дейкстры по-прежнему не применим, но для поиска можно применить алгоритм Беллмана-Форда.

Алгоритм возвращает «истину», если в графе нет циклов с отрицательным весом. Можно показать, что в качестве h можно взять кратчайшее расстояние до вершины $v \in V$ из источника s , то есть:

$$h(v) = d[v]$$

Далее приведём псевдокод алгоритма Джонсона. Пусть G – обрабатываемый граф, s – добавляемая вершина для построения графа G' (без рёбер отрицательного веса). После выполнения алгоритма массив d будет содержать длины кратчайших путей между всеми парами вершин, а массив path – сами пути.

```
bool Johnson (G, s, d, path)
```

```
Построить G'
f = BellmanFord (G', s, h)
if (f = false) then
    print('Граф содержит циклы с отрицательным
весом')
    return false
for (u,v) ∈ E do
    w(u,v) = w(u,v) + h(u) – h(v)
for u ∈ V do
    Dijkstra (G, u, d[u], path[u])
for (u,v) ∈ E do
    d(u,v) = d(u,v) – h(u) + h(v)
return true
```

Глава 2. Реализация и тестирование алгоритмов

2.1 Исходные данные

2.1.1 Формат данных

Перед кодированием и написанием программного кода определимся с данными, на которых будут проводиться эксперименты, и с форматом хранения графов.

Графы для экспериментов можно генерировать с помощью специальных генераторов или получить на основании реальных данных, например граф сети дорог, содержащий расстояния или времена между узловыми точками.

В данной работе мы остановимся на втором варианте, взяв граф карты дорог. Граф задаётся в файле и имеет текстовый формат. Файл содержит строчки следующих типов:

1. Комментарий (строка начинается с символа 'c').
2. Формат и описание графа (строка начинается с символа 'p'). Например, "p sp 2000 6000", означает, что граф разреженный и содержит 2000 вершин, 6000 рёбер.
3. Рёбра (строка начинается с символа 'a'). Например: "a 596 959 78", означает ребро из вершины 596 в вершину 959 с весом 78.

Строка с описанием графа должна предшествовать перечислению списка рёбер.

Граф карты дорог имеет разреженный формат, поэтому для хранения графа в оперативной памяти компьютера будем использовать разреженный формат аналогичный тому.

Результатом работы программы будет плотная матрица расстояний между каждой парой вершин. Для любого алгоритма поиска кратчайших путей эта матрица должна содержать одинаковые значения при одинаковых

входных данных.

2.1.2 Обработка данных

Для начала необходимо создать структуру для хранения графа в разряженном формате. Она будет выглядеть следующим образом:

```
#ifndef DATA_STRUCT
#define DATA_STRUCT
struct GraphMatrix
{
 int *pointerB; // указатели на начало списка связаных
 // ребер
 int *column; // индексы связанных вершин
 int *value; // веса ребер
 int sizeV; // количество вершин
 int sizeE; // количество ребер
};
#endif
```

Далее объявим макрос BUF_SIZE, который будет содержать количество элементов массива, необходимого для разбора строк файла. Объявим структуру TEdge, которая описывает ребро графа, и реализуем функцию, позволяющую сравнивать две структуры типа TEdge.

```
#define BUF_SIZE 200
struct TEdge
{
 int row;
 int col;
 int val;
};
bool CmpEdges( TEdge arg1, TEdge arg2 )
{
 if(arg1.row<arg2.row)
 return true;
 else
 if(arg1.row == arg2.row)
 {
```

```
 if(arg1.col<arg2.col)
 return true;
 }
 return false;
}
```

Теперь приступим к реализации функции загрузки графа из файла - ParseGraph. Эта функция будет принимать имя файла и двойной указатель на структуре хранения разреженного графа.

Для начала проверим, доступен ли указанный файл с графом, открыв его на чтение. Если функция fopen вернёт 0, то завершим функцию ParseGraph с кодом ошибки -1. Выделим память под структуру GraphMatrix и создадим контейнер vector, в котором будут временно содержаться ребра графа. Теперь загрузим из файла все ребра в контейнер edges и отсортируем, используя для сравнения функцию CmpEdges. Далее удаляем дубликаты рёбер (оставляем ребро минимального веса). Заполняем структуру GraphMatrix. Выводим информацию о загруженном файле на экран и закрываем файл с графом.

2.2 Реализация алгоритма Флойда

2.2.1 Последовательный алгоритм

Функция main() будет выполнять следующую последовательность действий:

1. разбор аргументов командной строки (для уменьшения объѐ мам кода будем использовать класс string из библиотеки STL);
2. загрузку графа из файла и запись графа в разреженном формате;
3. запуск алгоритма поиска кратчайших путей и измерение времени его работы;

4 вывод матрицы кратчайших путей в файл (запись файла осуществляется в бинарном формате) при наличии ключа “-o” в аргументах командной строки

```
int main(int argc, char **argv){
LARGE_INTEGER freq;
LARGE_INTEGER sQP, fQP;
QueryPerformanceFrequency(&freq);
if (argc < 2){
printf("\nUsage: program.exe <graph file> [-o]\n");
return 1;
}
GraphMatrix *gr;
int *up;
int *dist;
bool printOutput = false;
ParseGraph(argv[1], &gr);
if (argc == 3)
if(string(argv[2]) == string("-o"))
printOutput = true;
```

Далее выделяем память под матрицы расстояний и поиска кратчайших путей (pi-функцию).

```
// pi - функция
up = new int[gr->sizeV * gr->sizeV];
// дистанция до вершины
dist = new int[gr->sizeV * gr->sizeV];
```

После выполненных действий можно запустить алгоритм Флойда-Варшалла с замером времени работы алгоритма.

```
QueryPerformanceCounter(&sQP);
FloydWarshall(gr, up, dist);
QueryPerformanceCounter(&fQP);
printf("Floyd-Warshall time: %f\n",
(fQP.QuadPart-sQP.QuadPart) / (double) freq.QuadPart );
if(printOutput)
{
FILE *distFile=fopen("05_dist.dat", "wb");
fwrite(dist, sizeof(int), gr->sizeV * gr->sizeV,
```

```
distFile);
fclose(distFile);
printf("File (05_dist.dat) written.\n" );
}
```

Реализовав основную функцию программы, перейдем к реализации самого алгоритма Флойда-Варшалла. Объявление функции выглядит следующим образом:

```
void FloydWarshall(graphMatrix *gr, int *up, int *dist)
```

где:

- graphMatrix – входное значение, содержащее взвешенный граф
- int *up - pi-функция
- int *dist - растояние до вершин

Теперь необходимо разработать реализацию алгоритма. Для этого необходимо сначала сформировать исходную матрицу расстояний. Исходная матрица расстояний совпадает с матрицей смежности исследуемого графа. При построении все элементы на диагонали равны 0, то есть стоимость перехода в текущую вершину равна 0. Все остальные элементы формируются следующим образом. Если в матрице смежности графа существует ребро, то значение берется из графа, иначе в качестве расстояния берется некое очень большое число.

```
int i, j;
// переменные прохода по окрестности вершины графа
int okr_s, okr_f, okr_i;
int n = gr->sizeV;
for (i = 0; i < n * n; i++)
dist[i] = INT_MAX;
for (i = 0; i < n; i++)
{
dist[i * n + i] = 0;
up [i * n + i] = i;
}
for(i = 0; i < n; i++)
```

```
{
okr_s = gr->pointerB[i];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
if(dist[i * n + j] > gr->value[okr_i])
dist[i * n + j] = gr->value[okr_i];
up[i * n + j] = i;
```

Далее над полученной матрицей расстояний проводятся рекуррентные действия, описанные в алгоритме Флойда-Варшалла.

```
for(int k=0; k < n; k++)
for(int i=0; i < n; i++)
for(int j=0; j< n; j++)
if (dist[i * n + j] - dist[k * n + j] >
dist[i * n + k])
{
dist[i * n + j] =
dist[i * n + k] + dist[k * n + j];
up [i * n + j] = up[k * n + j];
}
```

Пример выполнения программы представлен на рис. 2.1.

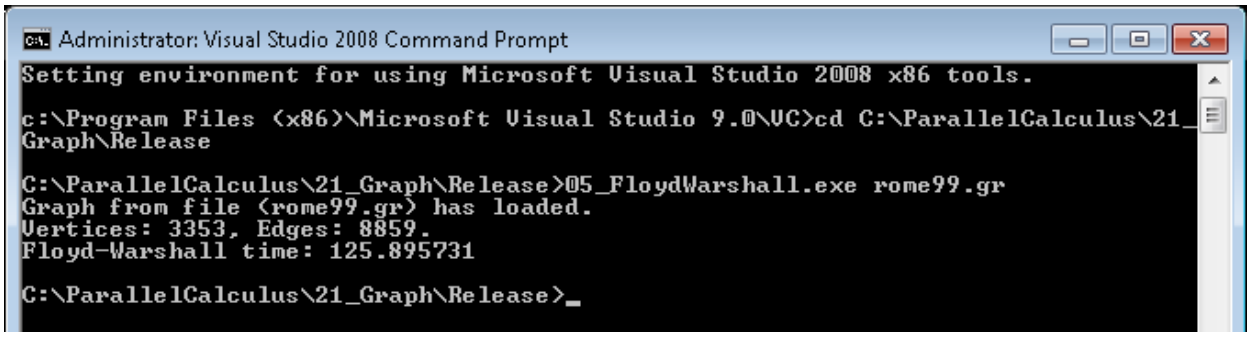

Рис. 2.1. Результаты работы последовательного алгоритма Флойда-Варшалла.

2.2.2 Параллельный алгоритм

Параллелизм, применимый для алгоритма Флойда – итерационный. Эффективный способ распараллеливания данного алгоритма состоит в

одновременном обновлении значений матрицы смежности, т.е. для каждой промежуточной вершины k вычислять путь из произвольной вершины i в произвольную вершину j параллельно. Данный способ параллельности корректен и не приведет к коллизиям.

Доказательство корректности. Необходимо доказать, что на каждой итерации внешнего цикла обновление матрицы смежности может выполняться независимо. Т.е. необходимо показать, что на итерации k не изменяются значения A[i][k], A[k][j] для любых i, j, так как все вычисления на данной итерации зависят только от этих величин.

A[i][j] = min(A[i][j], A[i][k] + A[k][j]):

• для i = k: A[i][k] = min(A[i][k], A[i][k] + A[k][k]). A[k][k] = 0

=> A[i][k] не изменится.

• для j = k: A[k][j] = min(A[k][j], A[k][k] + A[k][j]). A[k][k] = 0

=> A[k][j] не изменится.

Приведём псевдокод для параллельного алгоритма Флойда. Пусть граф задан матрицей смежности A[n][n]. Функция min, возвращающая минимум из 2 чисел, должна учитывать способ указания в матрице смежности несуществующих дуг графа. В данной реализации длина несуществующих дуг полагается равной бесконечности.

```
void Floyd (A)
for (k = 0; k < n; k++)
parallel_for (i = 0; i < n; i++)
if (A[i][k] ≠ ∞)
for (j = 0; j < n; j++)
if (A[k][j] ≠ ∞)
A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
```

(Здесь `parallel_for` – параллельный цикл.)

Сложность последовательной версии данного алгоритма – $O(n^3)$, параллельной – $O(n^3/p)$, где p – число процессоров. Алгоритм Флойда работает с матрицей смежности, следовательно его эффективнее при-

менять для графов, близких к полным.

2.3 Реализация алгоритма Джонсона

2.3.1 Последовательный алгоритм

Реализацию главной функции алгоритма Джонсона, как и раньше, следует начать с загрузки графа из файла.

```
int main(int argc, char **argv){
LARGE_INTEGER freq;
LARGE_INTEGER sQP, fQP;
QueryPerformanceFrequency(&freq);
if (argc < 2){
printf("\nUsage: program.exe <graph file> [-o]\n");
return 1;
}
GraphMatrix *gr;
int *up;
int *dist;
bool printOutput = false;
ParseGraph(argv[1], &gr);
if (argc == 3)
if(string(argv[2]) == string("-o"))
printOutput = true;
```

Далее должен последовать вызов функции поиска кратчайших путей с предварительным выделением памяти и замером времени.

```
// pi - функция
up = new int[gr->sizeV * gr->sizeV];
// дистанция до вершины
dist = new int[gr->sizeV * gr->sizeV];
QueryPerformanceCounter(&sQP);
Johnson(gr, up, dist);
QueryPerformanceCounter(&fQP);
printf("Johnson time: %f\n",
(fQP.QuadPart-sQP.QuadPart)/ (double)freq.QuadPart );
if(printOutput)
{
FILE *distFile=fopen("08_dist.dat", "wb");
fwrite(dist, sizeof(int), gr->sizeV * gr->sizeV,
distFile);
```

```
fclose(distFile);
printf("File (08_dist.dat) written.\n" );
}
```

Для реализации алгоритма Джонсона необходимо реализовать два алгоритма поиска кратчайших путей из источника во все вершины: алгоритм Дейкстры и алгоритм Беллмана-Форда.

Начнем с реализации алгоритма Дейкстры. Объявление функции выглядит следующим образом:

```
void Dijkstra(graphMatrix *gr, int givenNode, int *up, // int *dist)
```

где:

- graphMatrix – входное значение, содержащее взвешенный граф
- int givenNode - вершина источник
- int *up - pi-функция
- int *dist - растояние до вершин

Далее необходимо реализовать соответствующий алгоритм. Для этого разработаем вспомогательную структуру, позволяющую хранить пару элементов – вершину и оценку расстояния до нее. Также для работы с приоритетной очередью из библиотеки STL нам потребуется перегруженный оператор сравнения для введенной структуры.

```
struct Pair
{
int dist;
int node;
};
bool operator<(Pair p1, Pair p2)
{
// более приоритетны меньшие элементы!
return (p1.dist > p2.dist);
}
```

Прежде всего, в реализации алгоритма необходимо установить исходные значения для массива расстояний – расстояние до всех вершин

должно быть проинициализировано очень большим числом, а источник нулем.

```
void Dijkstra(GraphMatrix *gr, int givenNode, int *up,
int *dist)
{
//все вершины бесконечно удалены
for (int i=0; i < gr->sizeV; i++)
dist[i] = INT_MAX;
dist[givenNode] = 0;
up[givenNode] = givenNode;
```

Далее следует создать приоритетную очередь и поместить в нее начальную вершину.

```
priority_queue<Pair> pq;
//помещаем стартовую вершину в приоритетную очередь
Pair t = {0,givenNode};
pq.push(t);
```

Затем необходимо реализовать цикл, в котором, пока очередь не пуста, из нее изымается элемент с минимальным расстоянием, и с помощью выбранного элемента производится попытка уменьшить расстояния до всех смежных вершин (данная модификация алгоритма Дейкстры известна так же как алгоритм Мура).

```
//поиск кратчайших путей
while(!pq.empty())
{
// изымаем элемент из очереди
Pair s = pq.top();
pq.pop();
// для изъятой вершины проверяем окрестность
// на предмет уменьшения длин пути
int okr_s = gr->pointerB[s.node];
int okr_f = gr->pointerB[s.node + 1];
for(int okr_i = okr_s; okr_i < okr_f; okr_i++)
{
int j = gr->column[okr_i];
//релаксация
if (dist[j] > (dist[s.node] + gr->value[okr_i]))
```

```
{
dist[j] = (dist[s.node] + gr->value[okr_i]);
up[j] = s.node;
//помещаем вершину в очередь, т.к. она может
// уменьшить путь
Pair p = {dist[j],j};
pq.push(p);
} } }
```

Теперь займемся реализацией алгоритма Беллмана-Форда. Объявим соответствующую функцию:

bool BellmanFord(GraphMatrix *gr, int givenNode, int *dist)

где:

- graphMatrix – входное значение, содержащее взвешенный граф
- int givenNode - вершина источник
- int *dist - растояние до вершин

Для реализации Беллмана-Форда проинициализируем начальные расстояния до вершин. Как и в алгоритме Джонсона, начальные расстояния до всех вершин равны очень большому числу, а у стартовой вершины оно нулевое.

```
int okr_s, okr_f, okr_i;
int i, j, k;
// пока все вершины бесконечно удалены
for (i=0; i < gr->sizeV; i++)
dist[i] = INT_MAX;
// растояние до начальной вершины равно 0
dist[givenNode] = 0;
```

Далее размещаем реализацию алгоритма поиска путей Беллмана-Форда.

```
//поиск циклов и расстояний до вершин
for (k=0; k < gr->sizeV - 1; k++)
{
```

```
for (i=0; i < gr->sizeV; i++)
{
//для изъятой вершины проверяем окрестность
// на предмет уменьшения длин пути
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
//релаксация
if (dist[j] - gr->value[okr_i] > dist[i])
{
dist[j] = dist[i] + gr->value[okr_i];
} }
} }
```

И, наконец, реализуем проверку на наличие циклов с отрицательными весами.

```
for (i = 0; i < gr->sizeV; i++)
{
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
if (dist[j] > (dist[i] + gr->value[okr_i]))
return false;
}
}
return true;
}
```

Теперь, используя алгоритмы Дейкстры и Беллмана-Форда, реализуем алгоритм Джонсона. Объявим соответствующую функцию:

```
void Johnson(GraphMatrix *gr, int *up, int *dist)
```

где:

- graphMatrix – входное значение, содержащее взвешенный граф
- int *up - pi-функция

- int *dist - растояние до вершин

Вначале необходимо создать граф с дополнительной вершиной. У введенной вершины должны быть ребра до всех вершин в графе с весами равными нулю.

```
int i, j, n;
int okr_s, okr_f, okr_i;
int *h;
GraphMatrix gr_h;
n = gr->sizeV;
gr_h.sizeV = n + 1;
gr_h.sizeE = gr->sizeE + n;
gr_h.column = new int [gr->sizeE + n];
gr_h.value = new int [gr->sizeE + n];
gr_h.pointerB = new int [n + 2];
h = new int [n + 1];
for(i = 0; i < gr->sizeE; i++)
{
gr_h.column[i + n] = gr->column[i] + 1;
gr_h.value [i + n] = gr->value [i];
}
gr_h.pointerB[0] = 0;
for(i = 0; i < n; i++)
{
gr_h.pointerB[i + 1] = gr->pointerB[i] + n;
gr_h.column[i] = i;
gr_h.value [i] = 0;
}
gr_h.pointerB[i + 1] = gr->pointerB[i] + n;
```

Для полученного графа, применим разработанную программную реализацию алгоритма Беллмана-Форда. Алгоритм позволит определить, есть ли в графе циклы с отрицательными весами, а также величины, которые необходимо прибавить к весам ребер исходного графа, чтобы избавиться от отрицательных весов и одновременно не изменить кратчайшие пути.

```
bool f = false;
f = BellmanFord(&gr_h, 0, h);
if (!f)
{
```

```
printf("exist negativ circle\n");
return;
}
```

Используя полученные веса, модифицируем исходный граф.

```
for (i = 0; i < n; i++)
{
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
gr->value[okr_i] += h[i + 1] - h[j + 1];
}
}
```

Далее применяем алгоритм Дейкстры с источником во всех вершинах поочередно.

```
for(i = 0; i < n; i++)
Dijkstra(gr, i, up + n * i, dist + n * i);
```

Для получения решения возвращаем веса ребер и веса путей к весам исходного графа.

```
for (i = 0; i < n; i++)
{
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
gr->value[okr_i] -= h[i + 1] - h[j + 1];
}
}
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
dist[n * i + j] += h[j + 1] - h[i + 1];
delete []h;
}
```

Пример выполнения программы представлен на рис. 2.2.

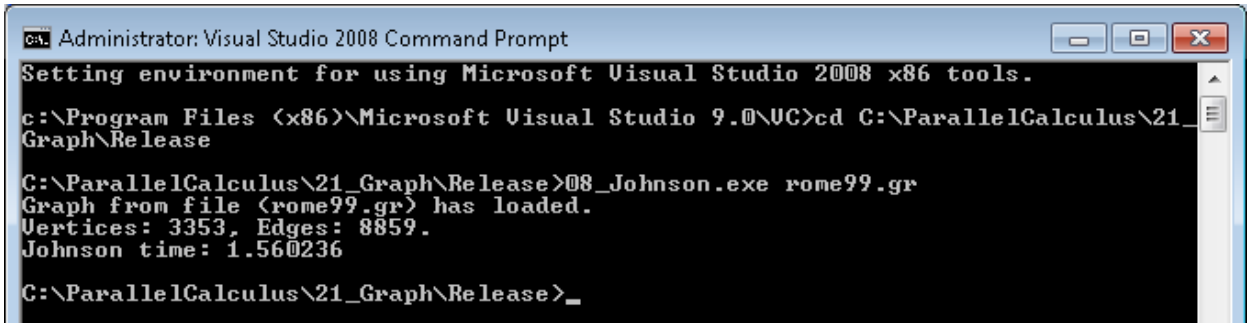

Рис. 2.2. Результаты работы последовательного алгоритма Флойда.

2.3.2 Параллельный алгоритм

Преобразование исходного графа к графу без рёбер отрицательного веса занимает незначительное время от общего времени работы алгоритма, поэтому его распараллеливание нецелесообразно. Эффективный способ распараллеливания состоит в одновременном применении алгоритма Дейкстры для каждой вершины. Алгоритм Дейкстры может работать с разными вершинами параллельно, так как граф не изменяется во времени и массивы расстояний и предшествования независимы.

Далее приведём псевдокод параллельного алгоритма Джонсона. Пусть G – обрабатываемый граф, s – добавляемая вершина для построения графа G' (без рёбер отрицательного веса). После выполнения алгоритма массив d будет содержать длины кратчайших путей между всеми парами вершин, а массив path – сами пути.

```
bool Johnson (G, s, d, path)
Построить G'
f = BellmanFord (G', s, h)
if (f = false) then
print('Граф содержит циклы с отрицательным весом')
return false
for (u,v) ∈ E do
w(u,v) = w(u,v) + h(u) – h(v)
parallel_for u ∈ V do
Dijkstra (G, u, d[u], path[u])
```

for (u,v) ∈ E do
d(u,v) = d(u,v) – h(u) + h(v)
return true

Сложность последовательной версии данного алгоритма – O(V(Vlog(V) + Elog(V)) + VE), тогда как сложность алгоритма Беллмана-Форда – O(VE), а алгоритма Дейкстры – O(Vlog(V) + Elog(V)), и при этом алгоритм Дейкстры нужно применить к каждой вершине. Сложность параллельной версии алгоритма Джонсона равна O(V(Vlog(V) + Elog(V))/p + VE), где p – число процессоров. Данный алгоритм работает со списком смежности, поэтому он эффективнее для разрежённых графов.

Глава 3. Сравнительный анализ реализаций

3.1 Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Процессор	восьмиядерный процессор AMD FX-8320 (3.5 GHz)
Память	16 Gb
Операционная система	Microsoft Windows 10
Среда разработки	Microsoft Visual Studio 2008: • Version 9.0.21022.8 • Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86

Результатом работы программы будет плотная матрица расстояний между каждой парой вершин. Для любого алгоритма поиска кратчайших путей эта матрица должна содержать одинаковые значения при одинаковых входных данных.

3.2 Практическое сравнение реализаций

При разработке программ были предусмотрены функции, отвечающие за подсчет времени работы алгоритмов в миллисекундах.

Основным вопросом исследования стало ускорение, которое позволяет получить соответствующий параллельный алгоритм по сравнению с последовательным аналогом, то есть отношение времени работы последовательного алгоритма к времени работы параллельного.

3.2.1 Параллельный алгоритм Флойда

Результат работы двух реализаций алгоритма Флойда на разном количестве потоков, представлен на рисунках 3.1 и 3.2.

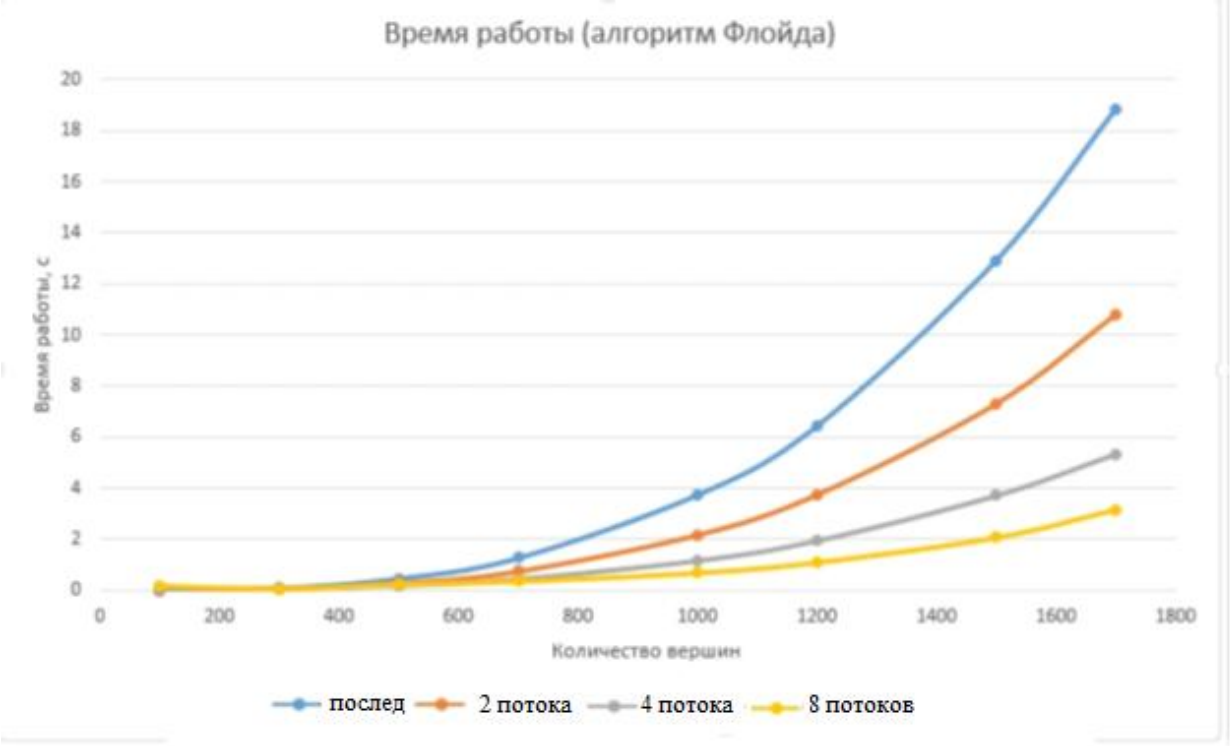

Рис 3.1. – Время работы алгоритма Флойда.

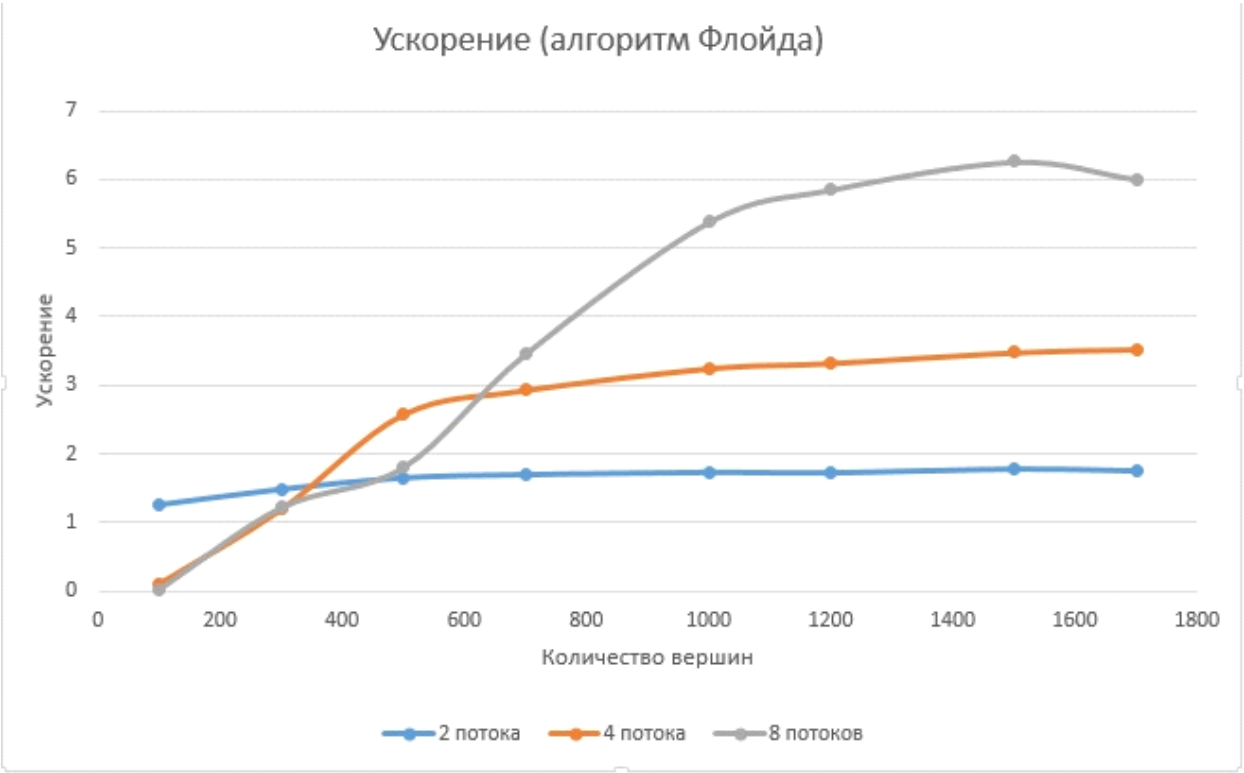

Рис 3.2. – Ускорение алгоритма Флойда.

Исходя из результатов экспериментов, можно сделать вывод, что применение параллельного алгоритма Флойда на полных графах целесообразно. Однако максимальное ускорение можно получить на графах с количеством вершин не менее 1000. Для графов с меньшим количеством вершин использование параллельного алгоритма Флойда не позволяет получить максимальное ускорение, так как в этом случае существенны накладные расходы. Также можно выделить тенденцию, что с возрастанием количества вершин графа наибольшее ускорение получается при использовании наибольшего количества потоков. Напротив, для небольших графов выгоднее использовать меньшее количество потоков.

3.2.2 Параллельный алгоритм Джонсона

Результат работы двух реализаций алгоритма Джонсона на разном количестве потоков, представлен на рисунках 3.3 и 3.4.

Рис 3.3. – Время работы алгоритма Джонсона.

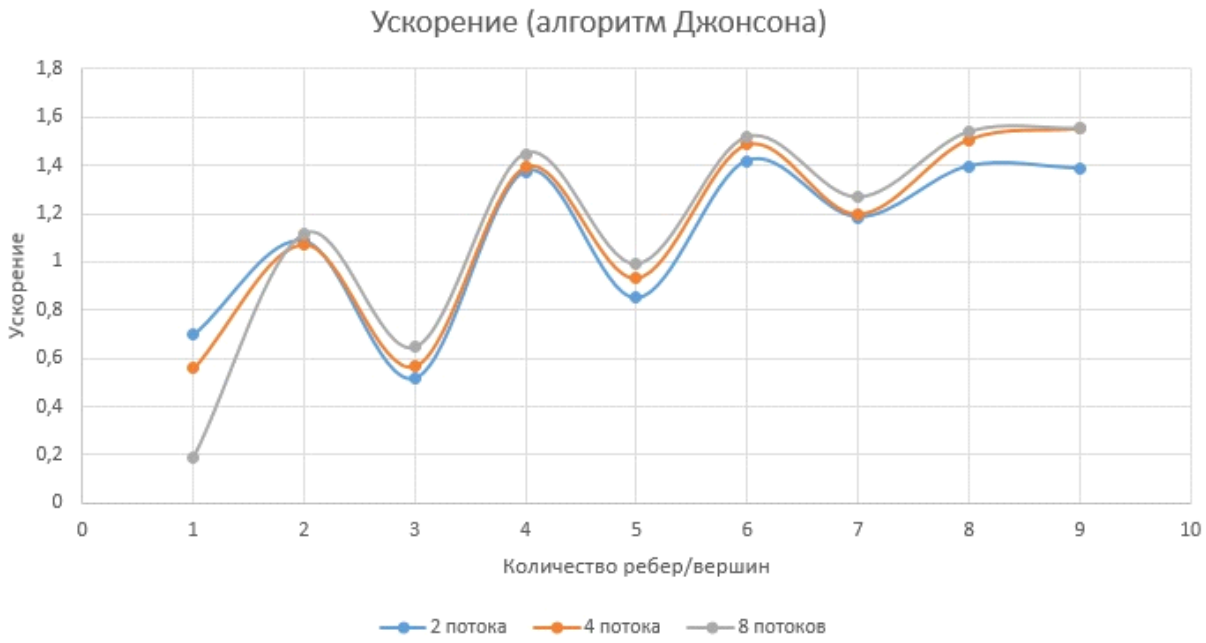

Рис 3.4. – Ускорение алгоритма Джонсона.

Исходя из результатов экспериментов, можно сделать вывод, что применение параллельного алгоритма Джонсона позволяет получить только слабое ускорение. Следует отметить, что для полных графов ускорение незначительно лучше.

3.2.3 Сравнение алгоритмов между собой

Поскольку сложность алгоритма Флойда $O(V^3)$, он является эффективным для расчёта всех кратчайших путей в плотных графах, когда имеет место большое количество пар рёбер между парами вершин.

Сложность Джонсона $O(V^2\log(V) + VE)$ следовательно, эффективней применять его для разреженных графов, таких как карты дорог, в таком случае алгоритм Джонсона работает значительно быстрее алгоритма Флойда. Результаты сравнения отображены на рисунке 3.5.

Рис. 3.5. – Сравнение Джонсона и Флойда.

Заключение

В ходе выполнения бакалаврской работы рассмотрены теоретические аспекты теории графов, алгоритмы решения классической задачи нахождение кратчайшего пути, достоинства и недостатки алгоритмов, а также реализации параллельных алгоритмов Беллмана-Форда и Джонсона на языке программирования C++.

Поиск кратчайшего пути – важная задача, возникающая в разных областях науки и техники: в экономике, робототехнике, в компьютерных играх и так далее. Исходя из того, что для решения всех четырех классических типов задач поиска кратчайших путей на графе, достаточно решить задачу 4 типа (нахождения кратчайшего пути от всех вершин ко всем вершинам), были реализованы параллельные версии классических алгоритмов для решения задач 4 типа.

В результате работы были пошагово реализованы параллельные алгоритмы Флойда, Джонсона на языке программирования C++ с применением технологии параллельного программирования OpenMP.

Вычислительные эксперименты, состоящие в сравнении времени работы данных параллельных алгоритмов, с последовательными аналогами, показали, что параллельный алгоритм Флойда позволяет получить значительно лучшее ускорение, чем параллельный алгоритм Джонсона.

Алгоритм Флойда так же показал, что с возрастанием количества вершин графа наибольшее ускорение получается при использовании наибольшего количества потоков. И напротив, для небольших графов выгоднее использовать меньшее количество потоков.

Однако в рамках рассмотренной задачи с графом карты дорог, алгоритм Джонсона значительно выигрывает, за счет того, что в основе лежит алгоритм Дейкстры эффективно работающий на разряженных графах.

Список используемой литературы

Научная и методическая литература

1. Громович, Ю. Детерминированные подходы к алгоритмизации труднорешаемых задач. Часть I: Основные определения / Ю. Громович - Эвристические алгоритмы и распределенные вычисления. – 2014. – Т.1, №2. – С. 30-32.
2. Макаркин, С. Б. Геометрические методы решения псевдогеометрической версии задачи коммивояжера / С. Б. Макаркин, Б. Ф. Мельников - Стохастическая оптимизация в информатике. - 2013. - Т. 9, № 2. - С. 54-72.
3. Емеличев, В. А. Лекции по теории графов / О. И. Мельников, В. И. Сарванов, Р. И. Тышкевич - Либроком – Москва. - 2012. - 392 с.
4. Берцун, В. Н. Математическое моделирование на графах. Часть 2: Томск/ В. Н. Берцун - Томский университет, - 2013. − 88 с.
5. Черноскутов, М. А Балансировка нагрузки в ГПУ-реализации поиска в ширину на графе / М. А. Черноскутов - Вычисл. методы и программирование. - 2013. - Т. 14. - С. 54–62.
6. Овчинников, В. Графы в задачах анализа и синтеза структур сложных систем / В. Овчинников – МГТУ им. Н. Э. Баумана. - 2014. – 424 с.
7. Лафоре, Р. Программирование в C++ / Р. Лафоре – Питер. – 2015. – 928 с.
8. Поляков, И. В. Алгоритмы поиска путей на графах большого размера / И. В. Поляков, А. А. Чеповский, А. М. Чеповский – НИУ Высшая школа экономики. – 2014. – Т. 19.
9. Кормен, Т. Алгоритмы: построение и анализ. 2-е издание / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн – М.: Вильямс. – 2013. – 1296 с.
10. Гергель, В. П. Теория и практика параллельных вычислений. / В. П. Гергель - Интуит Бином. Лаборатория знаний. – 2014. - 424 с.

Литература на иностранном языке

11. Gaurav Hajela M. P. Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford / International Journal of Computer Applications. 2014.

12. Julian Shun G. E. B. Ligra: A Lightweight Graph Processing Framework for Shared Memory / ACM SIGPLAN Notices. - 2013.

13. Umut Acar Arthur Charguerard M. R. Fast Parallel Graph-Search with Splittable and Catenable Frontiers. - 2015.

14. Umut Acar Arthur Charguerard M. R. Theory and Practice of Chunked Sequences / Algorithms-ESA 2014. - 2014.

15. Takaaki Hiragushi, Daisuke Takahashi. Efficient Hybrid Breadth-First Search on GPUs. Algorithms and Architectures for Parallel Processing / Lecture Notes in Computer Science. - 2013. - Vol. 8286. - P. 40–50.

16. Koji Ueno, Toyotaro Suzumura. Parallel distributed breadth first search on gpu - IEEE Intern. / Conf. on High Performance Computing (Bangalore, 18–21 Dec., 2013). Bangalore, - 2013. - P. 314–323.

17. Merrill D., Garland M., Grimshaw A. Scalable GPU graph traversal / Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New Orleans, 25–29 Febr., 2012). - New York. – 2012. – P. 117–128.

18. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths / IEEE 28th Intern. Parallel and Distributed Processing Symposium (Phoenix, 19–23 May, 2014). – Washington. – 2014. - P. 349–359.

19. Sedgewick R. Algorithms in C++ Part 5: Graph Algorithms, 3rd Edition. / Addison-Wesley Professional. – 2001. – 528 p.

20. Pape U. Implementation and efficiency of moor-algorithms for the shortest route problem / Mathematical programming 7. - 2012. – P. 212-222.

Приложение А

Листинг кода, реализации алгоритма Флойда

```
#ifndef FLOYD_WARSHALL_H
#define FLOYD_WARSHALL_H
#include "routine.h"
#include <algorithm>
//API
// void FloydWarshall(graphMatrix *gr, int *up, int *dist);
// Алгоритм Флойда-Уоршелла поиска кратчайших путей между
// каждой парой вершин.
//
//INPUT
// graphMatrix - взвешенный граф
//
//OUTPUT
// int * - pi-функция
// int * - растояние до вершин
void FloydWarshall(GraphMatrix *gr, int *up, int *dist);
#endif
//API
// void FloydWarshall(graphMatrix *gr, int *up, int *dist);
// Алгоритм Флойда-Уоршелла поиска кратчайших путей между // каждой
парой вершин
//
//INPUT
// graphMatrix - взвешенный граф
//
//OUTPUT
// int * - pi-функция
// int * - растояние до вершин
void FloydWarshall(GraphMatrix *gr, int *up, int *dist)
{
int i, j;
// переменные прохода по окрестности вершины графа
int okr_s, okr_f, okr_i;
int n = gr->sizeV;
for (i = 0; i < n * n; i++)
dist[i] = INT_MAX;
for (i = 0; i < n; i++)
{
dist[i * n + i] = 0;
```

```
up [i * n + i] = i;
}
for(i = 0; i < n; i++)
{
okr_s = gr->pointerB[i];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
if(dist[i * n + j] > gr->value[okr_i])
dist[i * n + j] = gr->value[okr_i];
up[i * n + j] = i;
}
}
for(int k=0; k < n; k++)
parallel_for(int i=0; i < n; i++)
for(int j=0; j< n; j++)
if (dist[i * n + j] - dist[k * n + j] >
dist[i * n + k])
{
dist[i * n + j] =
dist[i * n + k] + dist[k * n + j];
up [i * n + j] = up[k * n + j];
}
```

Приложение В

Листинг кода, реализации алгоритма Джонсона

```
#ifndef JOHNSON_H
#define JOHNSON_H
#include "Dijkstra.h"
//API
// bool BellmanFord(graphMatrix *gr, int givenNode,
// int *dist);
// Алгоритм поиска кратчайшего пути во взвешенном графе из
// источника во все вершины
//
//INPUT
// graphMatrix - взвешенный граф
// int - вершина источник
//
//OUTPUT
// int * - растояние до вершин
bool BellmanFord(GraphMatrix *gr, int givenNode,
int *dist);
//API
// void Johnson(graphMatrix *gr, int *up, int *dist);
// Алгоритм Джонсона поиска кратчайших путей между каждой
// парой вершин
//
//INPUT
// graphMatrix - взвешенный граф
//
//OUTPUT
// int * - pi-функция
// int * - растояние до вершин
void Johnson(GraphMatrix *gr, int *up, int *dist);

#endif

bool BellmanFord(GraphMatrix *gr, int givenNode,
int *dist){

// переменные прохода по окрестности вершины графа

int okr_s, okr_f, okr_i;
int i, j, k;
```

```
// пока все вершины бесконечно удалены
for (i=0; i < gr->sizeV; i++)
dist[i] = INT_MAX;
// растояние до начальной вершины равно 0

dist[givenNode] = 0;

//поиск циклов и расстояний до вершин
for (k=0; k < gr->sizeV - 1; k++)
{
for (i=0; i < gr->sizeV; i++)
{
//для изъятой вершины проверяем окрестность
// на предмет уменьшения длин пути
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
//релаксация
if (dist[j] - gr->value[okr_i] > dist[i])
{
dist[j] = dist[i] + gr->value[okr_i];
}
}
}

}

for (i = 0; i < gr->sizeV; i++)
{
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
if (dist[j] > (dist[i] + gr->value[okr_i]))
return false;
}
}
return true;

}

void Johnson(GraphMatrix *gr, int *up, int *dist)
```

```
{
int i, j, n;
int okr_s, okr_f, okr_i;
int *h;
GraphMatrix gr_h;
n = gr->sizeV;
gr_h.sizeV = n + 1;
gr_h.sizeE = gr->sizeE + n;
gr_h.column = new int [gr->sizeE + n];
gr_h.value = new int [gr->sizeE + n];
gr_h.pointerB = new int [n + 2];
h = new int [n + 1];
for(i = 0; i < gr->sizeE; i++)
{
gr_h.column[i + n] = gr->column[i] + 1;
gr_h.value [i + n] = gr->value [i];
}
gr_h.pointerB[0] = 0;
for(i = 0; i < n; i++)
{
gr_h.pointerB[i + 1] = gr->pointerB[i] + n;
gr_h.column[i] = i;
gr_h.value [i] = 0;
}

gr_h.pointerB[i + 1] = gr->pointerB[i] + n;

bool f = false;
f = BellmanFord(&gr_h, 0, h);
if (!f)

{

printf("exist negativ circle\n");
return;

}

delete [] gr_h.column;
delete [] gr_h.value;

delete [] gr_h.pointerB;

for (i = 0; i < n; i++)
{
okr_s = gr->pointerB[i ];
```

```
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
gr->value[okr_i] += h[i + 1] - h[j + 1];
}

}

for(i = 0; i < n; i++)

Dijkstra(gr, i, up + n * i, dist + n * i);

for (i = 0; i < n; i++)
{
okr_s = gr->pointerB[i ];
okr_f = gr->pointerB[i + 1];
for(okr_i = okr_s; okr_i < okr_f; okr_i++)
{
j = gr->column[okr_i];
gr->value[okr_i] -= h[i + 1] - h[j + 1];
}
}
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
dist[n * i + j] += h[j + 1] - h[i + 1];
delete []h;

}
```