|  |  |  |
| --- | --- | --- |
|  | ***Федеральное агентство по рыболовству***  ***Федеральное государственное бюджетное образовательное учреждение***  ***высшего образования***  ***«Астраханский государственный технический университет»***  **Система менеджмента качества в области образования, воспитания, науки и инноваций сертифицирована DQS**  **по международному стандарту ISO 9001:2015** | |
| Институт информационных технологий и коммуникаций  Направление подготовки 09.03.04 Программная инженерия  Профиль «Разработка программно-информационных систем»  Кафедра «Автоматизированные системы обработки информации и управления» | | |
| **КУРСОВОЙ ПРОЕКТ**  **Учебно-демонстрационная программа по параллельному поиску всех путей в графе**  по дисциплине «Параллельное программирование» | | |
| Допущен к защите  «\_\_» \_\_\_\_\_\_\_\_20\_\_г.  Руководитель  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Оценка, полученная на защите «\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_» | | Проект выполнен  обучающимся группы ДИПРб-41 Самарским В.В. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Руководитель  к.т.н., доцент Белов С.В.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Члены комиссии:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Белов С.В.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лаптев В.В.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Морозов А.В. | |  |

**Астрахань** **– 2020**

**ФЕДЕРАЛЬНОЕ АГЕНТСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ ПО РЫБОЛОВСТВУ**

**АСТРАХАНСКИЙ ГОСУДАРСТВЕННЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

|  |  |
| --- | --- |
| **УТВЕРЖДАЮ** | Кафедра «Автоматизированные системы  обработки информации и управления» |
| Заведующий кафедрой  д.т.н., профессор  Т.В. Хоменко\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

**ЗАДАНИЕ**

**на выполнение курсового проекта**

Обучающийся ***Самарский Владислав Валерьевич***

Группа ***ДИПРб-41***

Дисциплина ***Параллельное программирование***

Тема ***Учебно-демонстрационная программа по параллельному поиску всех путей в графе***

Дата получения задания «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.  
Срок представления обучающимся КП на кафедру «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Руководитель ***к.т.н., доцент***\_\_\_\_\_\_\_\_\_\_\_ ***Белов С.В.*** «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

должность, степень, званиеподписьФИО

Обучающийся \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ***Самарский В.В.*** «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

подпись ФИО

**Задачи**

Реализовать последовательный алгоритм поиска всех путей в графе. Разработать тестовый набор исходных данных и произвести замер производительности, на основе которой выполнить оценку эффективности алгоритма.

Разработать параллельный алгоритм поиска всех путей в графе. Выполнить замеры производительности на том же наборе тестовых данных. Произвести сравнение скорости выполнения последовательного и параллельного алгоритмов, выполнить анализ полученных данных.

**Список рекомендуемой литературы**

1. Гданский Н.И. Основы теории и алгоритмы на графах. Учебное пособие . 4-е изд. – М.: Инфра-М, 2020 – 206 с.
2. Кормен Т. Алгоритмы, построение и анализ. 3-е изд. – М.: Вильямс, 2019 – 1328 с.

|  |  |
| --- | --- |
| **УТВЕРЖДАЮ** | К заданию на курсовой проектпо дисциплине  «Параллельное программирование» |
| Заведующий кафедрой  д.т.н., профессор  Т.В. Хоменко\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20 \_\_\_г. |

**КАЛЕНДАРНЫЙ ГРАФИК**

курсового проектирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Разделы, темы и их содержание, графический материал | Дата сдачи | Объем, % |
| 1 | Выбор темы | 05.09.2020 | 1 |
| 2 | Техническое задание | 12.10.2020 | 3 |
| 3 | Разработка модели, проектирование системы   * *введение,* * *технический проект,* * *программа и методика испытаний,* * *литература* | 22.10.2020 | 25 |
| 4 | Программная реализация системы   * *работающая программа,* * *рабочий проект* * *скорректированное техническое задание (при необходимости)* | 15.11.2020 | 40 |
| 5 | Тестирование и отладка системы, эксперименты   * *работающая программа с внесёнными изменениями,* * *окончательные тексты всех разделов* | 05.12.2020 | 50 |
| 6 | Компоновка текста  Подготовка презентации и доклада   * *пояснительная записка* * *презентация* * *электронный носитель с текстом пояснительной записки, исходным кодом проекта, презентацией и готовым программным продуктом* | 18.12.2020 | 59 |
| 7 | Защита курсового проекта | 26.12.2020-29.12.2020 | 60-100 |

С графиком ознакомлен «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_\_г.

Самарский В.В. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, обучающийся группы ДИПРб-41

(фамилия, инициалы, подпись)

График курсового проектирования выполнен

без отклонений / с незначительными отклонениями / со значительными отклонениями

нужное подчеркнуть

Руководитель курсового проекта\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ к.т.н., доцент Белов С.В.

подпись, ученая степень, звание, фамилия, инициалы

**СОДЕРЖАНИЕ**

[Введение 5](#_Toc59621012)

[1 Технический проект 6](#_Toc59621013)

[1.1 Анализ предметной области 6](#_Toc59621014)

[1.1.1 Граф 6](#_Toc59621015)

[1.1.2 Способы представления графа 7](#_Toc59621016)

[1.1.2 Методы обхода графа 9](#_Toc59621017)

[1.2 Технология обработки информации 10](#_Toc59621018)

[1.2.1 Описание метода распараллеливания обхода графа в ширину 11](#_Toc59621019)

[1.2.2 Описание проектируемых интерфейсов 13](#_Toc59621020)

[1.3 Входные и выходные данные 13](#_Toc59621021)

[1.4 Системные требования 14](#_Toc59621022)

[2 Рабочий проект 15](#_Toc59621023)

[2.1 Общие сведения о работе системы 15](#_Toc59621024)

[2.2 Функциональное назначение программного продукта 15](#_Toc59621025)

[2.3 Инсталляция и выполнение программного продукта 15](#_Toc59621026)

[2.4 Общий алгоритм программного продукта 16](#_Toc59621027)

[2.4.1 Описание реализации алгоритмов обхода 16](#_Toc59621028)

[2.4.1 Описание реализации методов сравнения производительности 17](#_Toc59621029)

[2.5 Разработанные меню и интерфейсы 18](#_Toc59621030)

[2.6 Сообщения системы 19](#_Toc59621031)

[2.7 Замеры времени выполнения 19](#_Toc59621032)

[3 Программа и методика испытаний 23](#_Toc59621033)

[Заключение 24](#_Toc59621034)

[Список использованных источников 25](#_Toc59621035)

[Приложение 1 Листинг реализации параллельного поиска в ширину 26](#_Toc59621036)

[Приложение 2 Листинг реализации последовательного поиска в ширину 27](#_Toc59621037)

# ВВЕДЕНИЕ

Одной из основных проблематик современного программирования является решение задачи повышения скорости и качества обработки информации. Первоначально любые данные представляли из себя слабо упорядоченные куски информации.

В ходе стремительного развития человечества количество информации неуклонно росло. Довольно быстро пришло осознание, что всю информацию необходимо классифицировать, сортировать и хранить определенным образом, чтобы сократить время, затрачиваемое на получение доступа к ней. Так информация аккумулировалась в различных источниках, таких как книги, уставы, законы, летописи, картины, скульптуры. Всё зависит от категории информации.

Современное общество обрабатывает огромное количество данных, для их хранения реализованы представления, которые позволяют ускорить доступ информации. Одним из способов такого представления является, так называемая, древовидная структура или, другими словами, граф.

Граф позволяет представить информацию в виде совокупности узлов, связанных нитями по определенному правилу или принципу. И чтобы эту информацию прочитать или изменить, необходимо провести анализ содержимого. Для этого реализованы различные алгоритмы, которые позволяют получить доступ к информации или дать характеристику графа в целом.

Проблематикой текущего проекта является ускорение обхода графа с целью нахождения всех возможных путей из заданного узла. Механизмом ускорения служит многоядерная архитектура современных процессов и эффективное использования ресурсов процессора с помощью высокоуровневых языков программирования.

Целью разработки является исследование механизмов распараллеливания алгоритмов и анализ влияния многоядерной обработки данных на скорость выполнения.

# 1 ТЕХНИЧЕСКИЙ ПРОЕКТ

## **1.1 Анализ предметной области**

### **1.1.1 Граф**

Граф — абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, то есть соединений между парами вершин. Например, за множество вершин можно взять множество аэропортов, обслуживаемых некоторой авиакомпанией, а за множество рёбер взять регулярные рейсы этой авиакомпании между городами.

Для разных областей применения виды графов могут различаться направленностью, ограничениями на количество связей и дополнительными данными о вершинах или рёбрах. Многие структуры, представляющие практический интерес в математике и информатике, могут быть представлены графами.

Выделяют два основных вида графов:

* + - * ориентированный;
      * неориентированный.

Неориентированный граф **G**— это упорядоченная пара **G:=(V,E)**, где **V**— непустое множество вершин или узлов, а **E** — множество пар вершин, называемых рёбрами. Пусть (v,w) – дуга из вершины **V1** в **V2**, можно сказать, что дуга v w ведет от вершины **V1** до **V2**. Для неориентированного графа тогда справедливо следующее утверждение, что эта же дуга (v,w) ведет из **V2** в **V1**.



Рисунок 1.1 – Неориентированный граф

Ориентированный граф (сокращённо орграф) **G** — это упорядоченная пара **G:=(V,A)**, где **V** — непустое множество вершин или узлов, и **A** — множество (упорядоченных) пар различных вершин, называемых дугами или ориентированными рёбрами.

Пусть (v,w) — это дуга. Тогда вершину v называют её началом, а w — концом. Можно сказать, что дуга vw ведёт от вершины v к вершине w. Обратное, в данном случае, неверно.

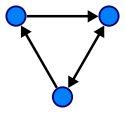


Рисунок 1.2 –Ориентированный граф

### **1.1.2 Способы представления графа**

Граф можно представить с помощью различных моделей, основные:

* + - * матрица смежности;
      * матрица инцидентности;
      * список смежности;
      * список рёбер.

Матрица смежности – способ представления, в котором граф изображается в виде таблицы **NxN**, где **N** – количество вершин, а каждый элемент матрицы указывает на наличие или отсутствие ребра между вершина **i** и **j**: **M[i,j] = 0** | **1**, где **M** – матрица смежности, **0** – дуги нет, **1** – дуга есть, 1 ≤ **i** ≤ **N**, 1 ≤ **j** ≤ **N**.

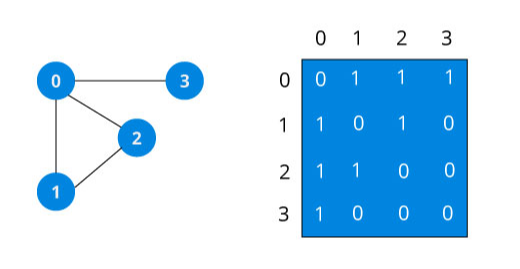


Рисунок 1.3 – Матрица смежности

Матрица инцидентности – способ представления, в котором граф изображается в виде таблицы, где строки соответствуют вершинам графа, а столбцы рёбрам. В ячейку матрицы на пересечении строки i и j записывается:

* + - * 1 в случае, если связь j выходит из вершины i;
      * -1 в случае, если связь j входит в вершину i (в случае неориентированного 1);
      * 0 во всех остальных случаях.

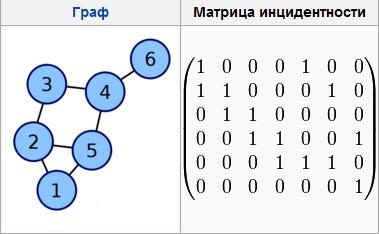


Рисунок 1.4 – Матрица инцидентности

Список смежности – представление графа в виде списка длинной **N**, где **N** – количество вершин. Каждый элемент списка содержит список смежных к себе вершин. Является наиболее удобным способом представления графа.

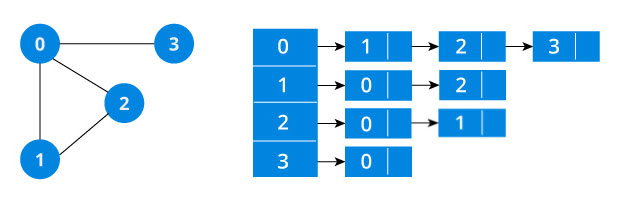


Рисунок 1.5 – Список смежности

Список рёбер – представление графа в виде списка длинной **M**, где **M** – количество рёбер. Каждый элемент списка содержит две вершины, инцидентных ребру. Является самым простым и компактным способом представления, поэтому часто применяется для внешнего хранения.

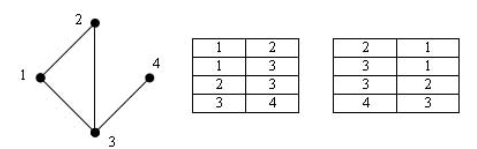


Рисунок 1.6 – Список рёбер

### **1.1.2 Методы обхода графа**

Выделяют два основных алгоритма обхода графа:

* + - * поиск в ширину (BFS, Breadh-First Search);
      * поиск в глубину (DFS, Depth-First Search).

Это не полный список алгоритмов, с помощью которых можно осуществить обход вершин графа, но перечисленные являются самыми популярными и используемые на практике.

Поиск в ширину — один из методов обхода графа. Пусть задан граф **G=(V,E)** и выделена исходная вершина **s**. Алгоритм поиска в ширину систематически обходит все ребра **G** для «открытия» всех вершин, достижимых из **s**, вычисляя при этом расстояние (минимальное количество рёбер) от **s** до каждой достижимой из **s** вершины. Алгоритм работает как для ориентированных, так и для неориентированных графов.

Неформально алгоритм можно описать следующим образом:

* + - 1. Поместить узел **s** в пустую очередь;
      2. Извлечь из начала очереди узел **u** и пометить как «достигнутый»;
      3. Добавить «недостигнутых» преемников узла **u** в очередь;
      4. Если очередь пуста, то завершить алгоритм, иначе вернуться к пункту 2.

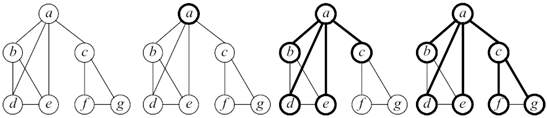


Рисунок 1.7 – Обход в ширину

Для хранения состояний достигнутых вершин можно использовать массив, в котором каждая достигнутая вершина помечается флагом.

Поиск в глубину – второй популярный метод обхода графа. В отличии от поиска в ширину, данный алгоритм пытается идти «вглубь» графа, насколько это возможно. Чаще всего, описывается рекурсивно из-за удобства реализации. Осуществляется перебор всех исходящих из рассматриваемой вершины рёбра. Если ребро ведет в вершину, которая не была рассмотрена ранее, алгоритм запускается от этой вершины. Концом выполнения рекурсии является отсутствие смежных нерассмотренных вершин, тогда управление возвращается к вызываемой функции и перебор осуществляется из ещё нерассмотренных узлов.

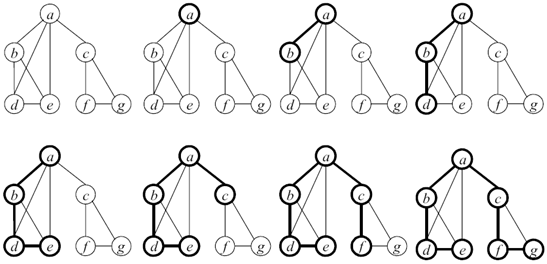


Рисунок 1.8 – Обход в глубину

На больших графах существует риск переполнения стек вызова рекурсией. Поэтому используется нерекурсивная версия алгоритма, неформальное описание которой приведено ниже:

* + - 1. Поместить узел **s** в пустой стек;
      2. Взять из стека узел **u** (не извлекая);
      3. Если вершина «недостигнута», пометить как «просмотренная» и добавить всех «недостгнутых» смежных в стек;
      4. Если вершина «просмотрена», то извлечь из стека и пометить как «достигнута»;
      5. Если очередь пуста, то завершить алгоритм, иначе вернуться к пункту 2.

## **1.2 Технология обработки информации**

Анализ предметной области показал, что алгоритм поиска в ширину больше подходит для распараллеливания. В BFS можно явно выделить этапы, которые выполняются независимо друг от друга, например, поиск соседних «непосещенных» вершин.

Главной проблемой является сохранение состояний узлов. Главной задачей алгоритма, в данном случае, является поиск всех путей, и если до какой либо вершины уже существует путь, то нет необходимости искать путь до этой же вершины из других смежных вершин к искомой из тех, что уже были «посещены».

Для реализации последовательного алгоритма обхода в ширину необходимо три структуры. Одна для представления списка смежности, вторая для списка посещённых вершин, третья представляет очередь, в которую будут помещаться смежные вершины. Общий алгоритм описан в 1.1.3.

Узким местом данного алгоритма является перебор всех смежных вершин. Алгоритм последовательно проверяет каждую из них и все смежные к просматриваемой на наличие флага «посещено».

Данную часть алгоритма можно распараллелить. Одним из вариантов решения является использования уровней, вместо очереди, в которую помещаются нерассмотренные вершины.

Проектируемая система должна выполнять следующие действия: предоставлять наглядное сравнение производительности для различных реализаций обхода графа в ширину и сохранять замеры на внешнем, по отношению к системе, хранилище. Для системы были сформированы следующие функциональные возможности, отражённые в диаграмме UML вариантов использования на рисунке 1.9.



Рисунок 1.8 – Диаграмма вариантов использования

### **1.2.1 Описание метода распараллеливания обхода графа в ширину**

Идея распараллеливания заключается в том, что начиная с начальной вершины **s** просмотр смежных вершин осуществляется по уровням, начиная с нулевого. Так, для вершины s распараллеливание не применяется, т.к. вершина всего одна. На следующем уровне в список добавляются все вершины, смежные s. Просмотр смежных вершин к каждой из них может осуществляться независимо, каждый параллельный процесс заполняет список нерассмотренных смежных вершин для следующего уровня.

Для фиксирования уровня, к которому принадлежат вершины, был разработан список размером **N**, где **N** – количество вершин. А значение каждой вершины может принимать следующие значения:

* -1 для непосещенных (недостигнутых) вершин;
* 0 для начальной вершины;
* От 1 до **N** для уровня, к которому принадлежит текущая вершина.

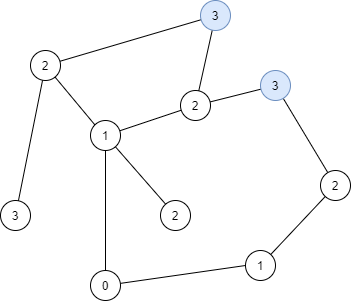


Рисунок 1.8 – Представление уровней в обходе графа в ширину

Уровни выдаются следующим образом: если смежная вершина к текущей является нерассмотренной, то выдать этой вершине уровень на 1 больше текущего и добавить вершину в список для обработки следующего уровня. На рисунке 1.8 серым отмечены пара вершин уровня 3. В случае параллельного доступа может возникнуть ситуация, что две различные вершины уровня два попробуют одновременно получить доступ к этой вершине и добавить в очередь на следующую итерацию цикла. Для предотвращения данной ситуации необходимо использовать средства синхронизации.

Алгоритм можно представить в следующем неформальном виде:

Создать список непосещенных вершин, отметить вершину s = 0;

Добавить в пустой список текущего уровня изначальную вершину s;

Текущий уровень level = 0;

Для всех вершин из списка текущего уровня параллельно выполнить;

4.1 Если сосед вершины не посещен, то добавить в следующий уровень;

4.2 Выбрать следующего соседа и повторить пункт 4.1;

Список текущего уровня = список следующего уровня;

Очистить список следующего уровня для следующей итерации;

level++;

Если список пуст, то выйти из алгоритма, иначе вернуться к пункту 4.

В данной реализации в пункте 4.1 осуществляется проверка «посещалась» вершина или нет. Для обеспечения безопасного доступа к списку уровней необходимо обеспечить потокобезопасный доступ, например через средство lock языка C#.

Таким образом исключаться повторные вызовы обхода вершин, которые были просмотрены другим потоком, но не успели быть отмеченными для других. Полное описание реализации алгоритма содержится в приложении 1.

### **1.2.2 Описание проектируемых интерфейсов**

Для демонстрации различия времени выполнения параллельного и последовательного алгоритмов необходимо разработать интерфейс, который будет выводить следующие данные:

* объём исходных данных (количество вершин и узлов графа);
* название алгоритма;
* время выполнения;
* сравнение результатов работы алгоритмов.

Результаты замеров времени выполнения выводить на экран и в лог-файл. На основе анализа требуемых интерфейсов функциональные возможности консольного вывода являются приемлемыми.

**1.3 Входные и выходные данные**

Входные данные:

* документ \*.edges или другого расширения, который хранит информацию в plain/text виде, внутри которого хранится граф в виде списка рёбер;
* входные параметры при запуске программы: имя файла с графом и номер исходной вершины.

Выходные данные:

* замеры времени выполнения последовательного и параллельного вариантов алгоритма;
* проверка корректности результатов в сравнении с эталонным;
* объём данных, обработанных алгоритмами;
* лог-файл с замерами времени выполнения.

**1.4 Системные требования**

Рекомендуемая конфигурация:

* Intel-совместимый процессор с частотой не менее 1,6 ГГц и не менее 2 логическими ядрами;
* не менее 512 МБ ОЗУ;
* не менее 50 MБ свободного места на диске;
* дисковод CD-ROM/DVD-ROM.

Операционная система: ОС с поддержкой .NET Runtime v5.0 .

Среда разработки – JetBrains Rider 2020.3.

# 2 РАБОЧИЙ ПРОЕКТ

## **2.1 Общие сведения о работе системы**

Проектируемый программный продукт представляет из себя приложение с консольным интерфейсом для замера производительности работы различных алгоритмов обхода в ширину. Разрабатывается в среде JetBrains Rider 2020.3 в виде проекта на платформе SDK .NET v5.0 с использованием языка С# 8.0. Версия 5.0 позволяет производить компиляцию программы в ОС Linux и Windows x64/x86 без внесения изменений, что требуется для кроссплатформенного приложения. И позволяет производить сравнительные замеры времени работы при допустимых по равенству условиях.

## **2.2 Функциональное назначение программного продукта**

Проектируемая система предназначена для последовательного запуска несколько вариаций алгоритма обхода в ширину и произведения сравнительных замеров производительности. Выводом программы является время работы каждого из реализованного алгоритмов в текстовом виде в консоли и в лог-файл.

Функциональные ограничения:

* Невозможно добавить дополнительную вариацию алгоритма обхода в ширину в систему без связи с разработчиком системы;
* нельзя изменять порядок выполнения вариаций алгоритма обхода;
* нельзя задать имя лог-файла.

**2.3 Инсталляция и выполнение программного продукта**

Для того, чтобы установить программный продукт на компьютер, необходимо сохранить на носитель папку bfs\_measurer. Удостовериться, что папка содержит сопроводительную документацию, исполнительный файл bfs и пример графа в файле 0.edges.

Установить .Net 5.0 Runtime при необходимости и через консоль запустить исполнительный файл:

* «.\bfs 0.edges 100» для ОС Linux;
* «bfs.exe 0.edges 100» для ОС Windows.

## **2.4 Общий алгоритм программного продукта**

Программный продукт на вход принимает имя файла, который должен содержать представление графа в виде списка рёбер, где в первой строке указано количество вершин, а все последующие пары вершин, которые соединены ребром. Вторым аргументом является стартовая вершина.

После запуска программы на исполнение производится инициализация структуры, которая хранит граф в системе. Класс «MergeList» представляет граф в виде списка смежности, где каждая вершина хранит список всех смежных к себе вершин. В таблице 2.1 представлено описание данного класса.

Таблица 2.1 – Описание класса MergeList

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| public List<List<int>> Nodes { get; set; } | Представление списка смежности |
| **Метод** | **Описание** |
| void InitDist() | Вспомогательный метод конструктора |
| List<int> SerialCheck (start\_point) | Последовательная реализация с проверкой |
| void Serial (start\_point) | Последовательная реализация |
| List<int> LeveledCheck(vertex) | Параллельная реализация, основанная на уровнях, с проверкой |
| void Leveled(start\_point) \_ | Параллельная реализация, основанная на уровнях |

Далее последовательно выполняются обходы графа через последовательную и параллельную версию алгоритма.

### **2.4.1 Описание реализации алгоритмов обхода**

Последовательная версия реализована с применением списка типа «bool», длинной **N**, где **N** – количество вершин и очереди, в которую помещаются все непросмотренные «соседи» рассматриваемой вершины. Цикл последовательно извлекает вершины из очереди и проверяет все смежные, если вершина не рассмотрена, то она добавляется в очередь. Сигналом окончания цикла является пустая очередь.

Параллельная версия алгоритма основывается на применение списка глубины, вместо очереди. Каждой вершине в соответствии с итерацией цикла, на которой она была достигнута, ставится номер уровня, которому она принадлежит. Если вершина ещё не была достигнута, или же пути к этой вершине нет, то значения для данной вершины является -1, в противном случае вершине присваивается уровень, следующий за текущим, а сама вершина добавляется в список для следующей итерации цикла.

В цикле, реализованном с помощью механизма .Net Parallel.For(), производится параллельный запуск просмотра всех вершин, смежных к списку рассматриваемых. Для обеспечения потокобезопасного доступа к списку глубины происходит блокировка попыток множественного обращения потоков, допуская только один. Блокировка реализована через общую переменную и механизм «lock». Листинг реализации распараллеливания обхода графа представлен в приложении 1.

### **2.4.1 Описание реализации методов сравнения производительности**

Для сравнения качества выполнения алгоритмов были реализованы механизмы сравнения полученных результатов на соответствие эталонным (эталонным в данном случае является последовательная реализация алгоритма). Для этого результирующий список посещенных вершин сравнивается между другими такими же списками из параллельных реализаций алгоритма.

Для замера времени выполнения алгоритмов были разработаны два метода, принимающие в качестве параметра метод на выполнение, краткое описание и стартовую вершину.

Таблица 2.2 – Описание методов замера времени

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| static List<int> StandMeasure(Func<int,List<int>> method, int start\_point, string name = null) | Метод замера времени, принимающий другой метод с 1 параметром и возвращающий результирующий список |
| static Stopwatch StandMeasure(Action<int> method, int start\_point, string name = null) | Метод замера времени, принимающий другой метод с 1 параметром и возвращающий void |

**2.5 Разработанные меню и интерфейсы**

Для программного продукта был разработан интерфейс, представленный на рисунке 2.1.

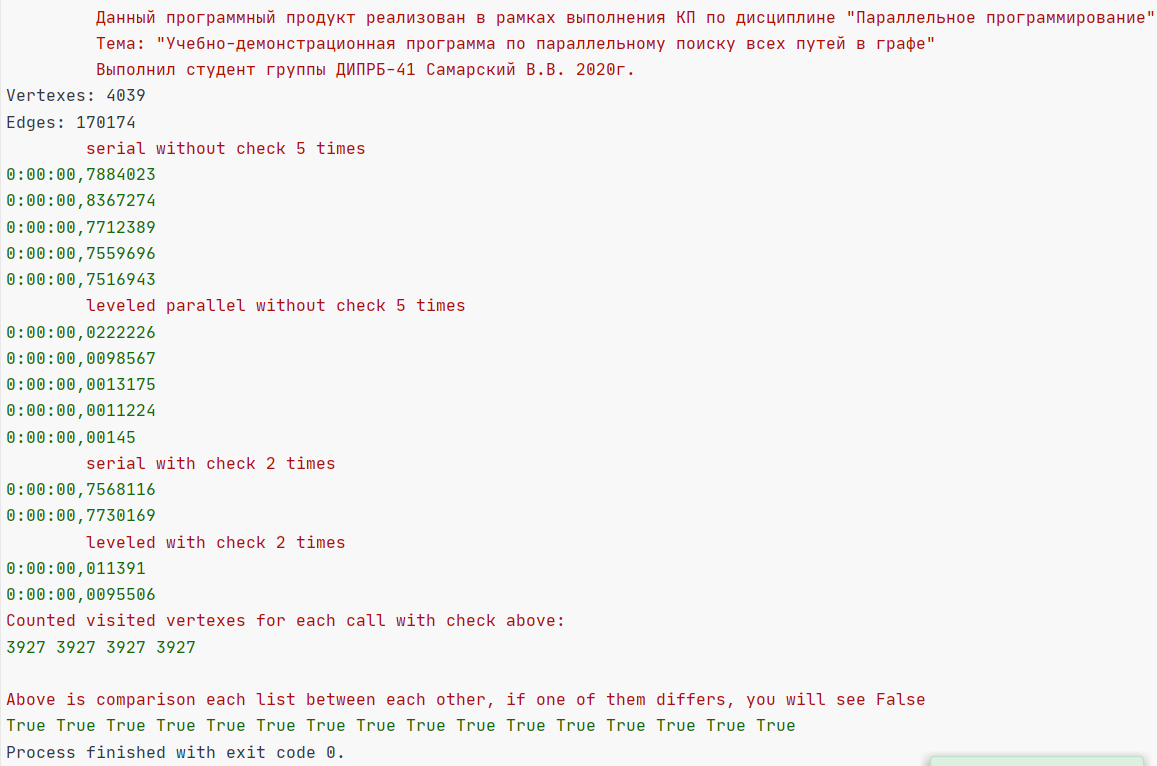


Рисунок 2.1 – Интерфейс разработанной системы

Для сохранения истории выполнения реализовано логирование событий замера времени работы. Каждый запуск алгоритма создаёт запись в лог-файле «log.txt», который содержит:

* дата сообщения;
* содержание сообщения.

Если файла не существует, он будет создан, содержимое файла автоматически не удаляется.

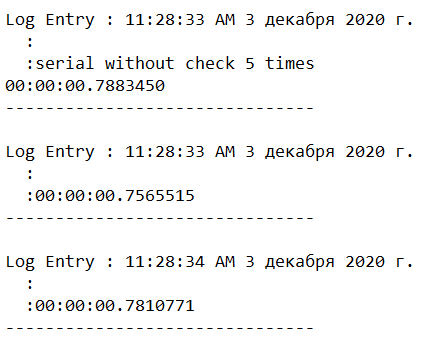


Рисунок 2.2 – Пример содержимого файла log.txt

## **2.6 Сообщения системы**

В таблице 2.3 приведены сообщения системы.

Таблица 2.3 – Сообщения отладочного интерфейса

|  |  |  |
| --- | --- | --- |
| **№ п/п** | **Сообщение** | **Причина возникновения** |
| 1 | give correct args [0] - filename [1]- correct index | Некорректные аргументы командой строки |
| 2 | check max vertex index and edges. each vertex should be < maxVertexIndex | Индекс начальной вершины не принадлежит графу |
| 3 | error in parsed file | Некорректный формат файла |
| 4 | Cannot detect file [filepath] | Не удалось найти файл по пути |

## **2.7 Замеры времени выполнения**

Замеры времени выполнения последовательного и параллельного вариантов обхода графа в ширину производились в двух операционных системах: Windows и Linux.

В качестве тестового графа был использован неориентированный вариант с количество узлов, равное 4039 и количеством рёбер равным 170174. Было проведено несколько испытаний, на основе которых были получены следующие данные:

Таблица 2.4 – результаты замеров последовательного выполнения

|  |  |  |  |
| --- | --- | --- | --- |
| **№ прохода** | **ОС Windows** | **OC Linux** | **Разница времени выполнения в Windows относительно Linux** |
| **1** | 0:00:00,7884023 | 0:00:00,8429484 | -0,0545461 |
| **2** | 0:00:00,8367274 | 0:00:00,8274209 | +0,0093065 |
| **3** | 0:00:00,7712389 | 0:00:00,8255038 | -0,0542649 |
| **4** | 0:00:00,7559696 | 0:00:00,8362848 | -0,0803152 |
| **5** | 0:00:00,7516943 | 0:00:00,8315301 | -0,0798358 |

Таблица 2.5 – результаты замеров параллельного выполнения при применении Paralell.for()

|  |  |  |  |
| --- | --- | --- | --- |
| **№ прохода** | **ОС Windows** | **OC Linux** | **Разница времени выполнения в Windows относительно Linux** |
| **1** | 0:00:00,0222226 | 0:00:00,0067783 | +0,0154443 |
| **2** | 0:00:00,0098567 | 0:00:00,0010095 | +0,0088472 |
| **3** | 0:00:00,0013175 | 0:00:00,0012265 | +0,0000910 |
| **4** | 0:00:00,0011224 | 0:00:00,0013520 | -0,0002296 |
| **5** | 0:00:00,0014500 | 0:00:00,0012632 | +0,0001868 |

Результаты замеров занесены в графики на рисунках 2.3 и 2.4.

Рисунок 2.3 – График времени выполнения последовательной реализации обхода в ширину на 5 итерациях

Рисунок 2.4 – График сравнения времени выполнения параллельной реализации обхода в ширину на 5 итерациях

Анализ полученных результатов показал, что за счёт организации независимого обхода вершин можно достичь значительного прироста производительности. Выигрыш времени напрямую зависит от связанности графа. Чем сильнее связан граф, тем меньше итераций выполняются последовательно, одновременно каждая итерация успевает обработать большее количество смежных нерассмотренных вершин.

Таблица 2.6 – показатель множителя разницы времени выполнения между линейной и параллельной реализацией алгоритма обхода графа в ширину (показатели округлены до 3 знаков после запятой).

|  |  |  |
| --- | --- | --- |
| **№ прохода** | **OC Windows** | **OC Linux** |
| **1** | 35,477 | 124,359 |
| **2** | 84,889 | 819,6341 |
| **3** | 585,380 | 673,056 |
| **4** | 673,529 | 618,553 |
| **5** | 518,409 | 658,272 |

При запуске параллельной версии алгоритма на небольших объёмах данных или на слабосвязанном графе получается обратный эффект. Организация процесса распараллеливания и синхронизации накладывает дополнительные расходы, в результате чего время выполнения полного обхода графа от заданной вершины дольше, чем у последовательной вариации этого же алгоритма. Для демонстрации был использован точно такой же граф, но запуск производился от вершины 90, которая связана всего с 4 соседними вершинами. Результаты тестов приведены в таблицах 2.7 и 2.8.

Таблица 2.7 – результаты замеров последовательного выполнения на небольшом наборе данных

|  |  |  |  |
| --- | --- | --- | --- |
| **№ прохода** | **ОС Windows** | **OC Linux** | **Разница времени выполнения в Windows относительно Linux** |
| **1** | 0:00:00,0013596 | 0:00:00,0017083 | -0,0003487 |
| **2** | 0:00:00,0000602 | 0:00:00,0000349 | +0,0000253 |
| **3** | 0:00:00,000034 | 0:00:00,0000366 | -0,0000026 |
| **4** | 0:00:00,000028 | 0:00:00,0000364 | -0,0000084 |
| **5** | 0:00:00,0000275 | 0:00:00,0000363 | -0,0000088 |

Таблица 2.8 – результаты замеров параллельного выполнения при применении Paralell.for() на небольшом наборе данных.

|  |  |  |  |
| --- | --- | --- | --- |
| **№ прохода** | **ОС Windows** | **OC Linux** | **Разница времени выполнения в Windows относительно Linux** |
| **1** | 0:00:00,0209966 | 0:00:00,0069041 | +0,0140925 |
| **2** | 0:00:00,0005138 | 0:00:00,0002417 | +0,0002721 |
| **3** | 0:00:00,0002851 | 0:00:00,0000878 | +0,0001973 |
| **4** | 0:00:00,0001745 | 0:00:00,0000688 | +0,0001057 |
| **5** | 0:00:00,0001733 | 0:00:00,0000901 | +0,0000832 |

При выполнении в среде линукс на небольшом наборе данных заметно значительное отличие во времени выполнения, в среднем ровно в 2 раза. Выполнение последовательной версии алгоритма относительно параллельной реализации показывает более высокую производительность.

**3 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ**

* + - 1. Открыть консоль в папке с проектом: bfs\_measure.
      2. Запустить программу bfs\_parallel с параметрами «37.edges 100».
      3. Удостовериться, что было высвечено сообщение 4, удостовериться в дублировании сообщения в лог файле log.txt.
      4. Запустить программу bfs с параметрами «0.edges 100».
      5. Удостовериться, что не было сообщений системы1-3.
      6. Удостовериться, что система автоматически закончила работу.
      7. Удостовериться, что сравнение всех списков достигнутых вершин закончилось с результатом «True».
      8. Удостовериться, что количество достигнутых вершин во всех сценариях равно 3927.
      9. Удостовериться, что параллельная версия алгоритма работает быстрее.
      10. Запустить программу bfs с параметрами «0.edges 90».
      11. Удостовериться, что количество достигнутых вершин по всех сценариях равно 3.
      12. Удостовериться, что параллельная версия алгоритма выполняется дольше во всех сценариях.
      13. Не закрывая консоль открыть файл log.txt, предварительно удостоверившись, что файл создан, иначе проверьте уровень разрешений у пользователя к файловой системе (linux).
      14. Завершить работу программы.

**ЗАКЛЮЧЕНИЕ**

В результате выполнения курсовой работы был разработан метод распараллеливания обхода графа в ширину, были проведены сравнительные анализы на основе тестов быстродействия и сделаны выводы о целесообразности применения распараллеливания обхода путей графа.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Белов С.В., Лаптев В.В., Морозов А.В., Толасова В.В., Мамлеева А.Р. Требования к оформлению студенческих работ. – Астрахань, АГТУ, 2017. 104 с.
2. Плаксин М.А. Тестирование и отладка программ для профессионалов будущих и настоящих [Электронный ресурс] 2-е изд. (эл.) – М.: БИНОМ. Лаборатория знаний, 2013. – 167 с.: ил.
3. Кнут Д. Искусство программирования, том 1 – М.: Вильямс, 2007 – 160 с.
4. Эспосито Д. Programming ASP.NET Core .:Microsoft Press 1-st edition 2018 – 416 c.
5. Гданский Н.И. Основы теории и алгоритмы на графах. Учебное пособие . 4-е изд. – М.: Инфра-М, 2020 – 206 с.
6. Stanford Network Analysis Project [Электронный ресурс] URL: <https://snap.stanford.edu/data/ego-Facebook.html> (дата обращения 02.12.2020)

# ПРИЛОЖЕНИЕ 1

**ЛИСТИНГ РЕАЛИЗАЦИИ ПАРАЛЛЕЛЬНОГО ПОИСКА В ШИРИНУ**

*// init distances for our vertex*var dist = new List<int>();  
for (int i = 0; i < Nodes.Count; i++)  
 dist.Add(-1);  
  
dist[initial\_vertex] = 0;  
*// set level to zero*var level = 0;  
  
var this\_level = new List<int>{initial\_vertex};  
var next\_level = new List<int>();  
  
while (this\_level.Count != 0)  
{  
 Parallel.For(0, this\_level.Count, node =>  
 {  
 var vert = this\_level[node];  
foreach (var neighbor in Nodes[vert])  
 {  
 *// всех непосещенных соседей помечаем* if (dist[neighbor] == -1)  
 lock ("neighbor")  
 {  
 dist[neighbor] = level + 1;   
 next\_level.Add(neighbor);  
 }  
 }});  
 this\_level = next\_level;  
 next\_level = new List<int>();  
 level++;  
}

# ПРИЛОЖЕНИЕ 2

**ЛИСТИНГ РЕАЛИЗАЦИИ ПОСЛЕДОВАТЕЛЬНОГО ПОИСКА В ШИРИНУ**

var queue = new Queue<int>();  
var visited = new List<bool>();  
for (int i = 0; i < Nodes.Count; i++)  
 visited.Add(false);  
  
queue.Enqueue(vertex);  
  
while (queue.Count != 0)  
{  
 var v = queue.Dequeue();  
 visited[v] = true;  
 foreach (var neighb in Nodes[v])  
 {  
 if (!visited[neighb])  
 queue.Enqueue(neighb);  
 }  
}