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

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование кафедры)
01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)
Системное программирование и компьютерные технологии
(направленность (профиль)/специализация

БАКАЛАВРСКАЯ РАБОТА

на тему **Параллельные алгоритмы решения задачи коммивояжера методом ветвей и границ**

Студент	Д.Д. Саушкин	_____
	(И.О. Фамилия)	(личная подпись)
Руководитель	М.А. Тренина	_____
	(И.О. Фамилия)	(личная подпись)
Консультанты	Н.В. Андрюхина	_____
	(И.О. Фамилия)	(личная подпись)

Допустить к защите

Заведующий кафедрой к.т.н., доцент, А.В. Очеповский _____
(ученая степень, звание, И.О. Фамилия) (личная подпись)
«_____»_____20_____г.

Тольятти 2019

Аннотация

Темой данной бакалаврской работы является «Параллельные алгоритмы решения задачи коммивояжера методом ветвей и границ».

Работа выполнена студентом Тольяттинского государственного университета, института математики, физики и информационных технологий, группы ПМИб-1502, Саушкиным Денисом Дмитриевичем.

Объект работы: алгоритмы решения задачи коммивояжера.

Предмет исследования: сравнительный анализ реализаций алгоритма ветвей и границ на языке программирования Python.

Цель работы: реализовать параллельные алгоритмы решения задачи коммивояжера методом ветвей и границ, сравнить их эффективность с другими реализациями метода ветвей и границ.

Для достижения цели работы необходимо решить следующие задачи:

1) Рассмотреть вариации алгоритма ветвей и границ.

2) Реализовать параллельный алгоритм ветвей и границ на языке Python.

3) Выявить достоинства параллельной реализации алгоритма ветвей и границ, провести сравнительный анализ.

Отчет состоит из введения, трех глав и заключения.

В первой главе представлены теоретические аспекты теории графов, описание задачи коммивояжера, алгоритмы, используемые для её решения, рассмотрены аспекты теории параллельных алгоритмов.

Во второй главе описан алгоритм ветвей и границ, представлены варианты его параллельных реализации.

В третьей главе производится сравнительный анализ реализаций метода.

Бакалаврская работа представлена на 47 страницах, включает 16 иллюстраций, список используемой литературы содержит 20 источников.

ABSTRACT

The title of the given graduation work is « Parallel algorithms for solving the traveling salesman problem by the branch and bound method».

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

The aim of the work is to implement parallel branch and bounds algorithm to find the most profitable route that passes through the specified cities at least once and then returns to the source city.

The object of the graduation work is the solution of the travelling salesman problem.

The subject of this work is comparative analysis of implementations of branch and bounds algorithm and it’s variations for solving the travelling salesman problem in the programming language Python.

The solution of this problem has found its application in different areas of science and technology: in the traffic optimization, in the manufacture of microchips, in the DNA sequence analysis, etc.

We study in details the theory of graphs, branch and bounds and branch and cuts algorithms 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.

The results of the study showed that paralleling of branch and bounds algorithm had a positive impact on the execution time of the algorithms on similar input data.

The graduation project consists of an explanatory note on 47 pages, including 16 figures, the list of 20 references including 3 foreign sources.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача коммивояжера имеет множество применений в различных областях и сферах деятельности человека. Примерами могут служить доставка посылок, сбор детей школьным автобусом, сбор заказов на складе.

Одним из основных подходов к решению этой задачи является метод ветвей и границ. Его отличают следующие особенности, существенные с точки зрения распараллеливания: неизвестный заранее информационный граф и необходимость обмена информацией между вычислительными потоками. В работе предлагается подход к распараллеливанию метода ветвей и границ. Мы сравниваем этот подход с последовательными реализациями, а также исследуем влияние различных способов организации вычислительного процесса на производительность приложения.

Актуальность темы бакалаврской работы обусловлена тем, что в последнее время основным способом повышения производительности вычислительных устройств стало увеличение числа вычислительных ядер в процессорах. Поэтому особоё значение приобретает разработка и улучшение параллельных реализаций для ресурсоёмких задач, к которым относится задача коммивояжера.

Объект работы: алгоритмы решения задачи коммивояжера.

Предмет исследования: сравнительный анализ реализаций алгоритма ветвей и границ и различных его вариаций на языке программирования Python.

Цель работы: реализовать параллельные алгоритмы решения задачи коммивояжера методом ветвей и границ, сравнить их эффективность с другими реализациями метода ветвей и границ.

Для достижения цели работы необходимо решить следующие задачи:

1) Рассмотреть вариации алгоритма ветвей и границ.

2) Реализовать параллельные алгоритмы ветвей и границ на языке Python.

3) Выявить достоинства и недостатки параллельных реализации алгоритма ветвей и границ, провести сравнительный анализ.

1 ОПИСАНИЕ АЛГОРИТМОВ

1.1 Общая характеристика задачи о коммивояжере

1.1.1 Основные понятия теории графов

Простым графом G называется пара (V(G), E(G)), где V(G) – это непустое конечное множество элементов, называемых вершинами или узлами графа, а E(G) – это конечное множество неупорядоченных пар различных элементов из V(G), которые называют линиями или ребрами. V(G) часто называют множеством вершин, а E(G) - множеством ребер графа G.

На рисунке 1.1 изображен простой граф G, множество вершин V(G) у которого представлено в виде множества {u, v, w, z}, а множество ребер E(G) состоит из пар {u, v}, {v, w}, {u, w}, {w, z}. Вершины u и v соединены ребром {u, v}. Исходя из того, что E(G) является множеством, в котором элементы не повторяются, то в простом графе одну пару вершин может соединить не более чем одно ребро.

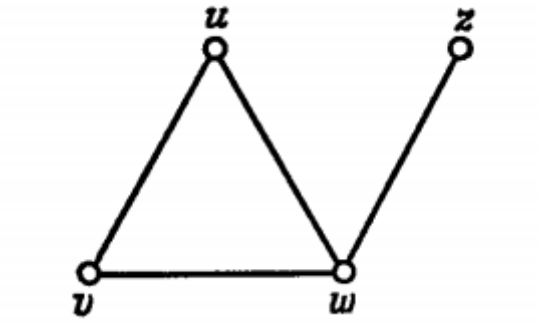

Рисунок 1.1 – Простой граф

Общий граф, который также часто называют просто графом, получают из простого графа, если снять ограничение, состоящее в том, что ребро должно соединить две различные вершины, и допустить существование петель. Петли – ребра, соединяющие вершину с ней самой. Так же снимается ограничение на количество ребер, соединяющих две вершины – вершины могут быть соединены более чем одним ребром. Не каждый граф можно назвать простым

графом, но каждый простой граф является графом. Общий граф изображен на рисунке 1.2.

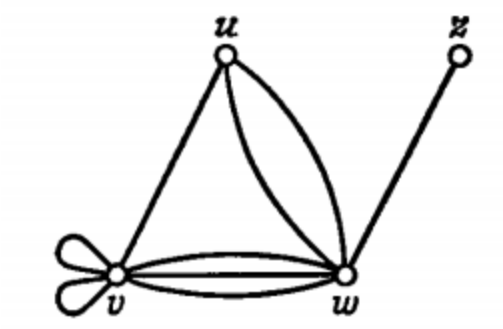

Рисунок 1.2 – Общий граф

Исходя из вышесказанного, графом G называется пара (V(G), E(G)), где V(G) – непустое конечное множество элементов, которое называют вершинами, а E(G) – конечное семейство не обязательно различных неупорядоченных пар элементов из V(G), называемых ребрами.

Семейством будем называть множество, в котором допускается кратность элементов, то есть в нашем случае допускается кратность ребер соединяющих одни и те же вершины.

V(G) будем называть множеством вершин, а E(G) – семейством ребер графа G. На рисунке 1.2 V(G) представлено множеством {u, v, w, z}, а E(G) – это семейство, представленное ребрами {v, w}, {v, w}, {v, w}, {u, w}, {u, w}, {u, v}, {v, v}, {v, v} и {w, z}.

Маршрут – его так же иногда называют путём, это последовательность рёбер графа $e_1, e_2, \dots, e_n$, где $n \geq 1$, для которой существует последовательность вершин $v_0, v_1, \dots, v_n$, такого вида, что для каждого i = 1, 2, …, n ребро e_i инцидентно вершинам v_{i-1} и v_i. Маршрут определяется последовательностью рёбер, чтобы можно было однозначно его определить, если в графе присутствуют кратные рёбра. Если кратных рёбер в графе нет, то маршрут можно однозначно задавать последовательностью вершин.

Ориентированный граф или орграф D – это пара (V(D), A(D)), где V(D) представляет собой непустое конечное множество элементов, которое называют

вершинами, а A(D) обозначает конечное семейство упорядоченных пар элементов из V(D), которые называют дугами или ориентированными дугами. Ориентированный граф изображен на рисунке 1.3.

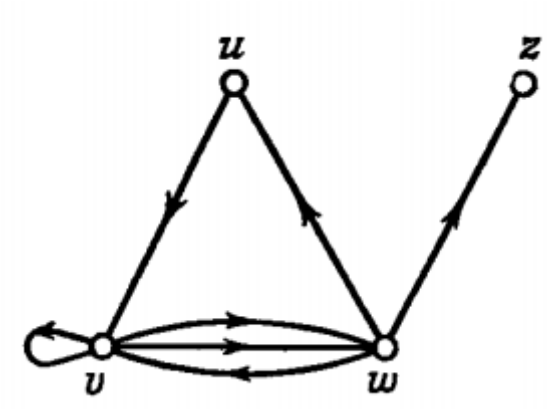

Рисунок 1.3 – Ориентированный граф

Цикл в ориентированном графе – это путь, где начальная вершина пути совпадает с последней вершиной.

Цикл в неориентированном графе – это путь, где начальная вершина пути совпадает с последней вершиной и каждое последующее ребро не совпадает с предыдущим.

Взвешенный граф – это граф, в котором каждому ребру присваивается некоторое число, называемое весом этого ребра. Такие веса могут представлять собой стоимость, длину или любую другую величину в зависимости от решаемой задачи.

Матрица смежности – для графа G с множеством вершин V представляет собой квадратную матрицу A размером $V \times |V|$, что её элемент a_{ij} равен числу рёбер в графе, соединяющих вершины i и j. Меняя обозначения вершин данного графа, можно получить несколько различных матриц смежности. Это в свою очередь приведёт к изменению порядка строк и столбцов матрицы A. Но в результате всегда получается симметричная матрица из неотрицательных чисел, сумма чисел в любой строке или столбце которой равна степени соответствующей вершины. Каждая петля учитывается в степени вершины один раз. По любой заданной симметричной матрице из неотрицательных чисел

можно построить граф, для которого данная матрица будет являться матрицей смежности.

1.1.2 Постановка задачи коммивояжера

Задача коммивояжёра (Travelling salesman problem, TSP) — возможно наиболее известная из задач комбинаторной оптимизации. Формулировка задачи звучит следующим образом: коммивояжер должен отправиться из начального города, посетить каждый из n порученных ему городов по одному разу и вернуться в начальный город. Расстояния между каждым из городов известны заранее. Необходимо найти маршрут, при котором путь коммивояжера будет кратчайшим.

Задачу коммивояжера так же можно сформулировать, используя теорию графов. Задан ненаправленный граф и цена для каждого ребра этого графа. Необходимо найти Гамильтонов цикл с минимальной общей стоимостью.

Существует несколько классификаций данной задачи:

1. Симметричная – в которой расстояния заданы между любыми двумя городами и равны в обоих направлениях.
2. Ассиметричная – в которой расстояния между городами могут отличаться, в зависимости от направления движения, и иногда могут отсутствовать пути между некоторыми городами.
3. С частичным упорядочиванием – в которой требуется, чтобы некоторые города посещались раньше других.
4. С гамильтоновым циклом – требующая нахождения замкнутых путей, проходящих через одну вершину не более одного раза.

Идеи, относящиеся к задаче коммивояжера, появились уже очень давно. В 1736 году Леонард Эйлер занимался проблемой нахождения кольцевого маршрута через семь мостов Кенигсберга. В 1832 была выпущена книга с названием «Коммивояжёр — как он должен вести себя и что должен делать для того, чтобы доставлять товар и иметь успех в своих делах — советы старого

курьера», которая включала в себя примеры маршрутов. В 1850-ых годах, Уильям Гамильтон изучал гамильтоновы циклы на графах. Он также выпустил игру с названием «Икосиан», целью которой было пройти все вершины додекаэдра ровно один раз от вершины к соседней, и вернуться в начало. Впервые, в качестве математической задачи, задача коммивояжера была предложена в 1930-ом году Карлом Менгером. Позже Хасслер Уитни предложил известное на данный момент название «задача странствующего торговца».

Задача коммивояжера применима в логистике, составлении расписаний работы различных станков, работе и перенастройке спутников, телескопов, микроскопов, лазеров, проектировании телекоммуникационных сетей.

Сложности, связанные с задачей коммивояжера, заключаются в том, что если вы начнете свой маршрут в определенном городе и будете двигаться к ближайшим оставшимся городам, то скорее всего не получите оптимальный маршрут. Ещё одной проблемой является то, что невозможно решить задачу коммивояжера простым перебором всех имеющихся маршрутов уже на количестве узлов близком к 20, так как алгоритмическая сложность задачи составляет (n-1)! для ассиметричной задачи и (n-1)!/2 для симметричной.

Для задачи коммивояжера создано большое количество методов и алгоритмов решения, но все они являются либо очень затратными по времени вычислений, либо решают только узкий класс задач, либо являются приближенными. Поэтому разработка алгоритмов решения задачи коммивояжера по-прежнему является актуальной задачей.

1.1.3 Математическая модель задачи коммивояжера

Задача коммивояжера может быть сформулирована в виде задачи целочисленного программирования. Существует несколько её формулировок. Наиболее известные формулировки Миллера, Такера и Землина (Miller-Tucker-Zemlin, MTZ) и Данцика, Фулкерсона, и Джонсона (Dantzig-Fulkerson-Johnson, DFG). Вторая формулировка задачи более устойчивая, но первая до сих пор

используется с некоторыми изменениями.

Рассмотрим формулировку задачи MTZ. Обозначим все города с помощью номеров от 1 до n и определим, что переменная x_{ij}, которая обозначает наличие пути между городами, будет равна 1, если такой путь есть, или 0, если такого пути нет.

$$x_{ij} = \begin{matrix} 1, \text{если междду городами } i \text{ и } j \text{ есть путь} \\ 0, \text{если между городами } i \text{ и } j \text{ нет пути} \end{matrix} \quad (1)$$

Для переменной i, принимающей значения от 1 до n, введём фиктивную переменную u_{ij}, а так же введём переменную c_{ij} для обозначения расстояния от города i до города j.

Тогда задача коммивояжера может быть представлена как задача целочисленного программирования.

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}: \quad (2)$$

$$x_{ij} \in \{0, 1\}, i, j = 1, \dots, n; \quad (3)$$

$$u_i \in Z, i = 1, \dots, n; \quad (4)$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, j = 1, \dots, n; \quad (5)$$

$$\sum_{j=1, i \neq j}^{n} x_{ij} = 1, i = 1, \dots, n; \quad (6)$$

$$u_i - u_j + n x_{ij} \leq n - 1, 2 \leq i \neq j \leq n; \quad (7)$$

$$0 \leq u_i \leq n - 1, 2 \leq i \leq n. \quad (8)$$

Первый набор равенств требует того, чтобы каждый город был посещён только из какого-либо одного города. Второй набор равенств требует, чтобы каждый город мог быть точкой отправления только в один какой-либо другой город. Последние ограничения предусматривают, что может быть только один единственный маршрут, который будет покрывать все города, а не два или более разрозненных маршрута, которые коллективно будут охватывать все города. Каждому маршруту, покрывающему все города, присваивается фиктивная

переменная, которая удовлетворяет ограничениям.

Чтобы доказать, что каждое допустимое решение содержит только одну замкнутую последовательность городов, достаточно показать, что каждый маршрут проходит через первый город, Эти равенства гарантируют, что может быть только один такой маршрут. В этом можно убедиться, если мы просуммируем все неравенства, соответствующие $x_{ij} = 1$ для каждого маршрута, состоящего из k шагов, не проходящих через первый город, то мы получим следующее ограничение:

$$nk \leq \ n - 1 \ k, \tag{9}$$

что в свою очередь будет являться противоречием.

Теперь необходимо доказать, что для каждого отдельного тура, охватывающего все города, существуют значения для фиктивных переменных u_i, которые удовлетворяют ограничениям. Без потери общности, определим начало и конец маршрута в первом городе. Обозначим $u_i = t$, если город i посетили на шаге t (i, t=1, 2, …, n). В этом случае

$$u_i - u_j \leq n - 1, \tag{10}$$

поскольку u_i не может быть больше чем n и u_j не может быть меньше 1. Следовательно, ограничения выполняются всякий раз, когда $x_{ij} = 0$. Для $x_{ij} = 1$ мы получаем:

$$u_i - u_j + nx_{ij} = \ t \ - \ t + 1 \ + n = n - 1, \tag{11}$$

что удовлетворяет ограничениям.

Теперь рассмотрим формулировку задачи DFG. Обозначим все города с помощью номеров от 1 до n и определим, что переменная x_{ij}, которая обозначает наличие пути между городами, будет равна 1, если такой путь есть, или 0, если такого пути нет.

$$x_{ij} = \begin{matrix} 1, \text{если междду городами } i \text{ и } j \text{ есть путь} \\ 0, \text{если между городами } i \text{ и } j \text{ нет пути} \end{matrix} \tag{12}$$

Обозначим c_{ij} как расстояние от города i до города j. Тогда задача коммивояжера может быть записана в виде задачи целочисленного линейного

программирования:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}: \tag{13}$$

$$0 \leq x_{ij} \leq 1, i, j = 1, \dots, n; \tag{14}$$

$$\sum_{i=1, j \neq i}^{n} x_{ij} = 1, j = 1, \dots, n; \tag{15}$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, i = 1, \dots, n; \tag{16}$$

$$\sum_{i \in Q} \sum_{j \in Q} x_{ij} \leq |Q| - 1, \forall Q \subseteq \{2, \dots, n\} \tag{17}$$

Последнее ограничение данной формулировки гарантирует, что среди непереходных вершин нет подуровней, поэтому возвращаемое значение – это одиночный обход, а не объединение меньших обходов. Поскольку это приводит к экспоненциальному числу возможных ограничений, на практике это решается с помощью отложенной генерации столбцов.

1.2 Параллельные алгоритмы

1.2.1 Параллелизм

В настоящее время стратегия увеличения мощности вычислительных устройств методом наращивания частоты, преобладавшая в компьютерных технологиях до последних лет, достигла технологического предела, и производители были вынуждены перейти на путь увеличения числа вычислительных блоков. Здесь можно отметить как увеличение числа ядер, расположенных на одном кристалле, так и значительный рост числа процессоров, входящих в кластерные системы и суперкомпьютеры. При этом однопроцессорные машины, а вместе с ними и программы, основанные на последовательном исполнении единственного потока команд, постепенно отходят в прошлое. Сейчас эффективная реализация практически любого

алгоритма должна обеспечивать по возможности наиболее равномерную загрузку всех вычислительных блоков системы, что выдвигает новые требования, как при разработке новых алгоритмов, так и при переложении существующих на современные аппаратные средства. Задача параллелизации вычислений стала особенно актуальной не только для исследователей, решающих сверхсложные задачи на мощных вычислительных системах, но также и для разработчиков многих других приложений, которые выполняются на персональных компьютерах.

Параллелизм – это процесс обработки нескольких наборов инструкций одновременно. Этот процесс уменьшает общее время вычислений. Параллелизм может быть реализован с использованием параллельных компьютеров, то есть компьютеров с несколькими процессорами. Для реализации параллелизма требуются алгоритм, который может быть распараллелен, язык программирования, компилятор и операционная система, поддерживающие многозадачность.

Одна из проблем параллелизма – разделение задачи на подзадачи. Подзадачи могут иметь между собой зависимые данные. Поэтому процессоры должны общаться друг с другом, чтобы решить эту проблему. Часто время, необходимое процессорам для связи друг с другом, составляет значительную часть от общего времени обработки. Таким образом, при разработке параллельного алгоритма необходимо учитывать правильную загрузку процессоров, чтобы получить эффективный алгоритм.

1.2.2 Потоки и процессы

Параллельная обработка – это выполнение нескольких задач одновременно: либо выполнение кода одновременно на разных процессорах, либо выполнение кода на одном и том же процессоре и получение ускорения за счет использования «простаивающих» циклов процессора, пока программа ожидает получения внешних ресурсов – загрузка файлов, вызовы API.

Процесс является экземпляром программы (например, блокнот Jupyter, интерпретатор Python). Процессы порождают потоки (подпроцессы) для обработки подзадач, таких как чтение нажатий клавиш, загрузка HTML-страниц, сохранение файлов. Потоки живут внутри процессов и совместно используют одно и то же пространство памяти.

Процесс порождает потоки: один для чтения нажатий клавиш, другой для отображения текста, третий для автоматического сохранения файла и еще один для выделения орфографических ошибок. Создавая несколько потоков, используется свободное время процессора (ожидание нажатия клавиш или файлы для загрузки) и повышает производительность.

Потоки используют память, выделенную под процесс, а процессы требуют себе отдельное место в памяти. Поэтому потоки создаются и завершаются быстрее: системе не нужно каждый раз выделять им новое адресное пространство, а потом высвобождать его.

Процессы работают каждый со своими данными — обмениваться чем-то они могут только через механизм межпроцессного взаимодействия. Потоки обращаются к данным и ресурсам друг друга напрямую: что изменил один — сразу доступно всем. Поток может контролировать соседние потоки в одном процессе, в то время как процесс контролирует исключительно дочерние процессы и потоки. Поэтому переключаться между потоками получается быстрее и коммуникация между ними организована проще.

В стандартной реализации Python (CPython) имеется GIL (Global Interpreter Lock) – способ синхронизации потоков. GIL является самым простым способом избежать конфликтов при одновременном обращении разных потоков к одним и тем же участкам памяти. Когда один поток захватывает его, GIL, работая по принципу мьютекса, блокирует остальные. Нет параллельных потоков — нет конфликтов при обращении к разделяемым объектам. Очерёдность выполнения потоков определяет интерпретатор в зависимости от реализации. Главный недостаток подхода обеспечения потокобезопасности при помощи GIL — это ограничение параллельности вычислений. GIL не позволяет

достигать наибольшей эффективности вычислений при работе на многоядерных и мультипроцессорных системах. Также использование нескольких потоков накладывает издержки на их переключение из-за эффекта конкуренции, когда потоки пытаются перехватить GIL. То есть многопоточное выполнение может занять большее время, чем последовательное выполнение тех же задач.

Перечислим особенности, присущие процессам:

- Создаются операционной системой для запуска программ;
- Процессы могут иметь несколько потоков;
- Два процесса могут выполнять код одновременно в одной и той же программе;
- Процессы имеют больше накладных расходов, чем потоки, так как процессы открытия и закрытия занимают больше времени;
- Обмен информацией между процессами происходит медленнее, чем обмен между потоками, поскольку процессы не разделяют пространство памяти.
- Процессы ускоряют операции Python, которые требуют значительных ресурсов процессора, поскольку они выигрывают от использования нескольких ядер и избегают использования GIL.

Особенности потоков:

- Потоки похожи на мини-процессы, которые живут внутри процесса;
- Они разделяют пространство памяти и эффективно читают и записывают одни и те же переменные;
- Два потока не могут выполнять код одновременно в одной и той же программе на Python.
- Потоки лучше всего подходят для задач ввода-вывода или задач, связанных с внешними системами, поскольку потоки могут комбинировать свою работу более эффективно. Процессы должны травить свои результаты, чтобы объединить их, что требует времени.

• Потоки не дают никаких преимуществ в Python для задач с интенсивным использованием процессора из-за GIL.

1.2.3 Синхронизация в многопоточных приложениях

Для организации одновременной работы потоков с одной и той же областью и решения проблем, связанных с очерёдностью принятия изменений, используют механизмы синхронизации. Потоки обмениваются информацией с помощью сигналов. Каждый поток сообщает другим, что он сейчас делает и каких изменений следует ждать. Так данные всех потоков о текущем состоянии ресурсов синхронизируются. В категориях объектно-ориентированного программирования сигналы — это объекты синхронизации. У каждого из них есть своя роль во взаимодействии. Перечислим основные средства синхронизации.

Взаимоисключение (мьютекс) — флаг, переходящий к потоку, который в данный момент имеет право работать с общими ресурсами. Исключает доступ остальных потоков к занятому участку памяти. Мьютексов в приложении может быть несколько, и они могут разделяться между процессами. К недостаткам мьютекса можно отнести то, что он заставляет приложение каждый раз обращаться к ядру операционной системы, что накладно.

Семафор — позволяет ограничить число потоков, имеющих доступ к ресурсу в конкретный момент. Так снижается нагрузка на процессор при выполнении кода, где есть узкие места. Проблема в том, что оптимальное число потоков зависит от машины пользователя.

Событие — определяется условие, при наступлении которого управление передаётся нужному потоку. Данными о событиях потоки обмениваются, чтобы развивать и логически продолжать действия друг друга. Один получил данные, другой проверил их корректность, третий — сохранил на жёсткий диск. События различаются по способу отмены сигнала о них. Если нужно уведомить о событии несколько потоков, для остановки сигнала

придётся вручную ставить функцию отмены. Если же целевой поток только один, можно создать событие с автоматическим сбросом. Оно само остановит сигнал, после того как он дойдёт до потока. Для гибкого управления потоками события можно выстраивать в очередь.

Критическая секция — более сложный механизм, который объединяет в себе счётчик цикла и семафор. Счётчик позволяет отложить запуск семафора на нужное время. Преимущество в том, что ядро задействуется лишь в случае, если секция занята и нужно включать семафор. В остальное время поток выполняется в пользовательском режиме. К недостаткам можно отнести то, что секцию можно использовать только внутри одного процесса.

1.3 Метод ветвей и границ и его улучшения для решения задачи коммивояжера

1.3.1 Общая схема метода ветвей и границ для оптимизационных задач

Рассмотрим теорию алгоритмов ветвей и границ. Понятия ветвей и границ не относятся к одному единственному алгоритму. Это структура, которая используется в разработке алгоритмов для задач комбинаторной оптимизации. Мы рассмотрим метод ветвей и границ в общем виде, но не будем забывать о задаче коммивояжера, чтобы рассмотреть конкретные примеры алгоритма.

Предположим, у нас есть множество возможных решений определенной проблемы комбинаторной оптимизации. Каждое решение имеет объективную ценность, и мы хотим найти решение, которое минимизирует эту объективную ценность. Для задачи коммивояжера это объективное значение – длина маршрута. Идея состоит в том, чтобы разбить множество всех решений на два или более непересекающихся подмножества, каждое из этих подмножеств также можно разделить на подмножества и так далее. Расщепление основано на ограничении, которому решения в подмножестве должны удовлетворять. Этот процесс называется ветвлением. Таким образом, мы создаем дерево, в котором

каждый узел содержит подмножество всех решений. Корневой узел не имеет ограничений и содержит все решения. Каждый дочерний узел наследует ограничения своего родителя, а также имеет дополнительное ограничение. В контексте задачи коммивояжера, ограничением может быть наличие определенного преимущества в маршруте. В этом случае узел имеет два дочерних элемента: левый потомок представляет все маршруты, которые содержат определённое ребро и правый потомок представляют все маршруты, которые не содержат это ребро. На определенной точке в дереве ограничения определяют единственное решение. Они являются листьями дерева: никакие дальнейшие ограничения не могут быть наложены.

Для каждого узла мы можем, в зависимости от задачи, рассчитать нижнюю границу на всех решениях, представленных в узле. Когда нижняя граница выше значения какого-то известного решения, нам не нужно искать дальше в этой части дерева. Мы можем быть уверены, что оптимальное решение не будет найдено в этом узле, и этот узел можно удалить. Когда на листе дерева найдено новое решение, мы сравниваем его с нашим лучшим решением и сохраняем лучшее из них. Таким образом, мы продолжаем находить лучшие решения, и многие другие узлы могут быть сокращены.

Процесс заканчивается, когда больше нет узлов для исследования. Мы можем быть уверены, что наше текущее лучшее решение является оптимальным решением. Это можно показать следующим образом. Рассмотрим часть дерева, которая была сгенерирована, когда алгоритм был завершён. Предположим, что где-то в дереве есть лучшее решение, чем наше текущее лучшее решение. Это решение должно быть в любом из узлов в конце ветви исследуемой части дерева. Эти узлы являются либо листовыми, либо обрезанными узлами. В том случае, если решения являлись листовыми узлами, они в какой-то момент сравнивались с лучшим решением в ходе решения, но были отброшены, поэтому они не могут быть меньше текущего лучшего решения. В том случае если это обрезанный узел, нижняя граница для этого узла должна быть выше, чем наше текущее лучшее решение. Это означает, что

мы можем быть уверены, что в дереве нет лучшего решения.

Алгоритм ветвей и границ состоит из трёх основных компонентов: ветвление, ограничение и выбор следующего узла. В этих компонентах должны быть приняты важные решения, которые повлияют на общую производительность алгоритма. Рассмотрим эти три компонента.

Ветвление определяет, как мы решаем, какие ограничения нам следует накладывать на потомков узла.

Ограничение позволяет нам выбрать, каким образом нам следует определять нижнюю границу для всех решений в узле. Качество этой нижней границы имеет большое значение для работы алгоритма. С одной стороны, мы хотим, чтобы нижняя граница была как можно выше. В идеальной ситуации ограничение будет равно лучшему решению в узле. В этом случае, будет удалено больше узлов, и дерево может остаться маленьким. С другой стороны, вычислительные затраты для нахождения жесткой нижней границы могут быть слишком высокими.

Выбор следующего узла для обработки. После создания потомков и вычисления их нижней границы, мы должны выбрать, какой из этих потомков является следующим узлом для обработки. Это определяет, в каком порядке мы обходим дерево. Этот выбор может зависеть от нижней границы потомков.

1.3.2 Метод ветвей и границ, применительно к решению задачи коммивояжера

Теперь мы применим наш метод ветвей и границ к точным решениям примеров для задачи коммивояжера. Ограничения, которые мы налагаем на маршруты, заключаются в том, чтобы ребро, которое должно быть представлено в маршруте, было в нём представлено. А ребро, которого в нём быть не должно, отсутствовало. Мы будем называть их включенными и исключенными ребрами соответственно. После того как были сгенерированы два дочерних элемента, следующим обрабатываемым узлом будет дочерний элемент с наименьшей

нижней границей, потому что более вероятно найти лучшие туры в этом узле, что приведет к большему количестве узлов, которые будут сокращены. Как для ветвления, так и для ограничивающего шага мы будем рассматривать две разные стратегии. Их можно объединить в четыре разных алгоритма. Мы применим эти алгоритмы к небольшим экземплярам задачи коммивояжера и сравним производительность четырех алгоритмов. Блок-схема алгоритма представлена на рисунке 1.4.

Рассмотрим детально стратегии ветвления в алгоритме ветвей и границ.

Простейшая стратегия ветвления заключается в наложении ограничений в лексикографическом порядке. Это означает, что мы начинаем с ребра (1, 2), затем (1, 3), (1, 4) и так далее. Мы в дальнейшем будем именовать эту стратегию как LG.

Другой способ ветвления заключается в порядке увеличения длины ребер. Это означает, что следующим ограничением всегда является наименьшее ребро, которое еще не включено или не исключено. Идея заключается в том, что более мелкие рёбра, скорее всего, будут присутствовать в лучшем маршруте. Таким образом, мы ранее найдем короткие маршруты, с помощью которых будет обрезано больше ветвей. Мы будем называть эту стратегию ветвления как IL.

После наложения нового ограничения мы должны проверить, должны ли быть включены или исключены другие ребра. Мы делаем это в соответствии со следующими правилами, описанными Винером:

• Если исключены все ребра, кроме двух, смежных с вершиной, эти два ребра должны быть включенным, иначе тур был бы невозможен.

• Если включены два ребра, смежные с вершиной, все остальные ребра, смежные с этой вершиной, должны быть исключены.

• Если при включении ребра будет образована петля в маршруте с другими включенными ребрами, это ребро должно быть исключено.

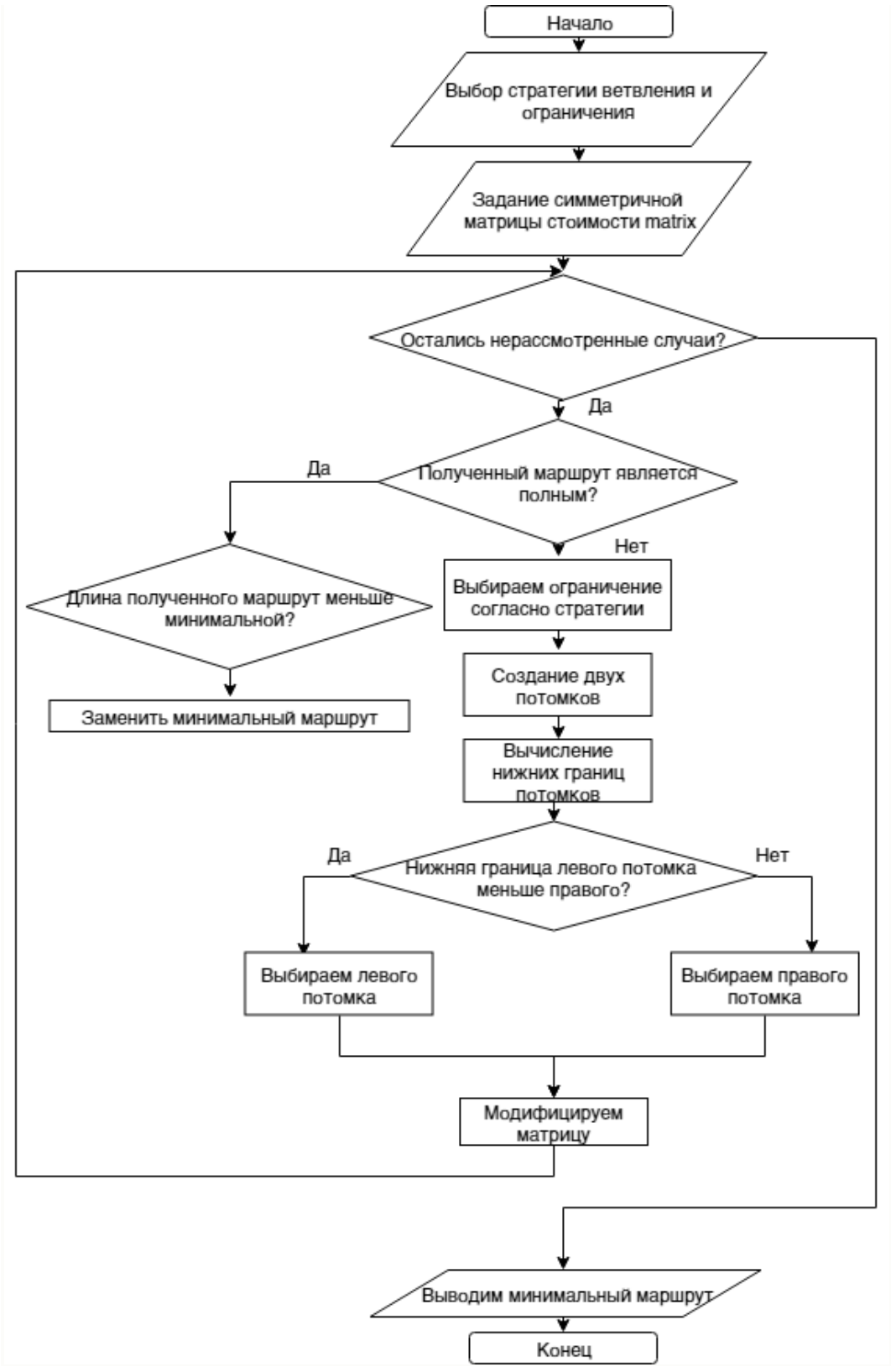

Рисунок 1.4 – Блок-схема метода ветвей и границ

Теперь рассмотрим варианты ограничения в методе ветвей и границ.

Простая нижняя граница – это один из способов определения нижней

границы. Предположим, у нас есть тур, который удовлетворяет определенным ограничениям. Для каждой вершины i, пусть (a_i, i) и (b_i, i) будут кратчайшими рёбрами рядом с той вершиной, которая должна присутствовать в маршруте с учетом ограничений. Пусть (k_i, i) и (l_i, i) будут рёбрами, которые представлены в туре. Тогда длина L тура равна

$$L = \frac{1}{2} \sum_{i=1}^{n} (w_{k,i} + w_{l,i}) \tag{18}$$

и это не меньше, чем суммирование по наименьшим возможным краям:

$$\frac{1}{2} \sum_{i=1}^{n} w_{a_i i} + w_{b_i i} \leq \frac{1}{2} \sum_{i=1}^{n} w_{k_i i} + w_{l_i i} = L \tag{19}$$

Это даёт нам более низкую оценку в любом туре. Мы будем называть эту нижнюю границу SB.

Другой способ вычисления нижней границы – использование алгоритма 1-дерева. Этот способ был впервые описан Хельдом и Карпом. Алгоритм 1-дерево – это вариант с использование минимального остовного дерева.

Пусть G(V, E) – связный неориентированный взвешенный граф. Тогда остовное дерево является подмножеством E’ ⊂ E таким, что для любого s, t ∈ V существует путь из s в t, используя ребра только из E’ и такие, что ребра в E’ не образуют циклов. Минимальное остовное дерево (MST) является таким остовным деревом, что сумма весов ребер в E’ сводится к минимуму.

Когда веса всех ребер различны, минимальное остовное дерево уникально. Если учесть, что n – это число вершин в графе, то число ребер в минимальном остовном дереве равно n - 1. Найти минимальное связующее дерево, мы можем использовать алгоритм Крускала, который описан ниже.

Пусть G(V, E) – граф с |V| = n. Включать рёбра в порядке увеличения длины пока очередное ребро не образует цикл с уже включенными ребрами. Остановиться, когда будет охвачено всё дерево. Это произойдёт после добавления n - 1 ребер.

Теперь можем определить метод 1-дерева.

Пусть G (V, E) – связный неориентированный взвешенный граф с V = {1, 2, ..., n}. Тогда 1-дерево является остовным деревом вершин {2, ..., n} вместе с двумя ребрами, смежными с вершиной 1. Минимальное 1-дерево – это 1-дерево минимальной длины.

Минимальное 1-дерево можно найти, складывая два самых коротких ребра, смежных с вершиной 1 до минимального остовного дерева с ребрами {2, ..., n}. Пример минимального 1-дерева показан на рисунке 1.5.

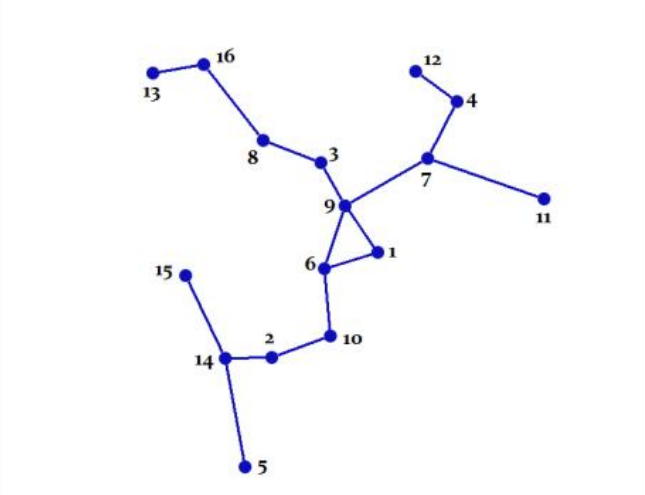

Рисунок 1.5 – Пример минимального 1-дерева

Минимальное 1-дерево может быть использовано в качестве нижней границы для TSP следующим образом. Пусть G (V, E) будет графом, как в определении метода 1-дерева, и пусть T ⊂ E будет маршрутом в G. Удалим ребра, смежные с вершиной 1. Поскольку остальные ребра образуют остовное дерево вершин {2, ..., n}, отсюда следует, что T является 1-деревом. Это означает, что набор всех туров является подмножеством набора из 1-деревьев. Тогда длина минимального 1-дерева является нижней границей в любом туре. Мы будем ссылаться на это определение нижней границы как OT.

1.3.3 Параллельные алгоритмы

Большинство параллельных алгоритмов ветвей и границ реализуют ту или иную форму параллельного исследования дерева. Фундаментальная идея

заключается в том, что в большинстве областей применения алгоритма размер дерева метода ветвей и границ растет очень быстро. Таким образом, если исследование дерева поиска будет выполняться несколькими процессами, быстрое получение знаний во время поиска позволит сократить количество узлов или устранить больше ветвей дерева.

Последовательный алгоритм ветвей и границ по сути является рекурсивной процедурой, которая извлекает узел из пула, тем самым удаляя его из структуры данных, выполняет серию операций (вычисление границы, ветвление) и завершает цикл, вставляя одну или несколько новых подзадач, полученных в результате операции ветвления в один и тот же пул.

К узлам в пуле обычно обращаются и получают доступ в соответствии с их приоритетом на основе различных атрибутов узла значений нижней и верхней границы и стратегии исследования дерева. Таким образом, приоритеты узла определяют порядок на узлах пула, а также последовательное планирование поиска. Очевидное свойство последовательного поиска ветвей и границ состоит в том, что при каждом планировании узла принимается решение с полным знанием информации о состоянии поиска, которая является глобальным представлением всех ожидающих обработки узлов, созданных до сих пор.

В параллельной среде могут распространяться как поисковые решения, включая решения по планированию узлов, так и поисковая информация. В частности, не вся соответствующая информация может быть доступна вовремя и в том процессоре, где принимается решение. При изучении поисковой информации в параллельном контексте необходимо решить две проблемы: как информация хранится и какая информация доступна во время принятия решения.

Большая часть информации о поиске состоит из пула узлов, и стратегии управления пулами решают первую проблему, касающуюся ее: как разложить пул узлов. Централизованная стратегия сохраняет все узлы в центральном пуле. Это означает, что этот уникальный пул в той или иной форме обслуживает все

процессоры, участвующие в параллельных вычислениях. Альтернативно, в распределенной стратегии пул разделен, причем каждое подмножество сохраняется одним процессором. Другой вопрос заключается не в том, распространяется ли глобальная информация о лучшем узле или нет, а в том, доступна ли она в обновленной форме при принятии решений.

С точки зрения управления поиском мы различаем два основных, но фундаментальных типа процессов: ведущие или управляющие и подчиненные процессы. Главные процессы выполняют весь спектр задач. Они определяют работу, которую должны выполнять подчиненные процессы, и полностью участвуют в обмене данными с другими процессами, чтобы реализовать параллельное планирование узлов, управлять обменом информацией и определением прекращения поиска. Подчиненные процессы обмениваются данными исключительно с назначенным им главным процессом и выполняют заранее заданные задачи в подзадаче, которую они получают. Подчиненные процессы не участвуют в планировании деятельности.

Классическая стратегия централизованного управления использует эти два типа в двухслойной процессорной архитектуре, одном мастер-процессе и нескольких рабочих процессах. Данная стратегия представлена на рисунке 1.6. Эта стратегия обычно объединяется со стратегией централизованного управления пулом. Таким образом, главный процесс поддерживает глобальные знания и контролирует весь поиск, в то время как подчиненные процессы выполняют поиск в узлах, полученных от мастер-процесса и возвращают результат мастеру.

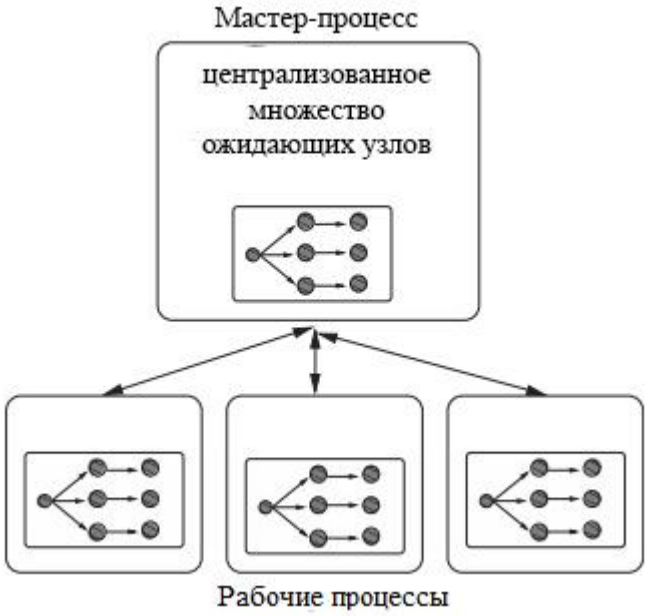

Рисунок 1.6 – Схема централизованного управления

Есть и другая стратегия распределенного управления, объединенная с подходом управления распределенным пулом. В этом случае, иногда называемым коллегиальным и показанным на рисунке 1.7, несколько основных процессов совместно управляют поиском и связанной информацией. В этом базовом подходе нет подчиненных процессов. Каждый главный процесс связан с одним из локальных пулов, в котором хранятся подмножества существующих в данный момент узлов (там нет разделения локальных пулов между несколькими мастерами). Затем он выполняет операции метода ветвей и границ в своем локальном пуле на основе этой частичной информации об узле, а также информации, передаваемой другими процессами. Таким образом, комбинированные действия всех мастеров составляют параллельное планирование с частичным знанием.

Распределение пула и, следовательно, распределение поисковой информации часто приводит к неравномерным рабочим нагрузкам для различных процессов во время поиска. Должна быть реализована стратегия балансировки нагрузки, чтобы указать, как циркулирует информация относительно рабочих нагрузок процессора и как принимаются соответствующие решения по балансировке нагрузки. Также необходимо

обмениваться информацией, обновляющей глобальные переменные состояния.

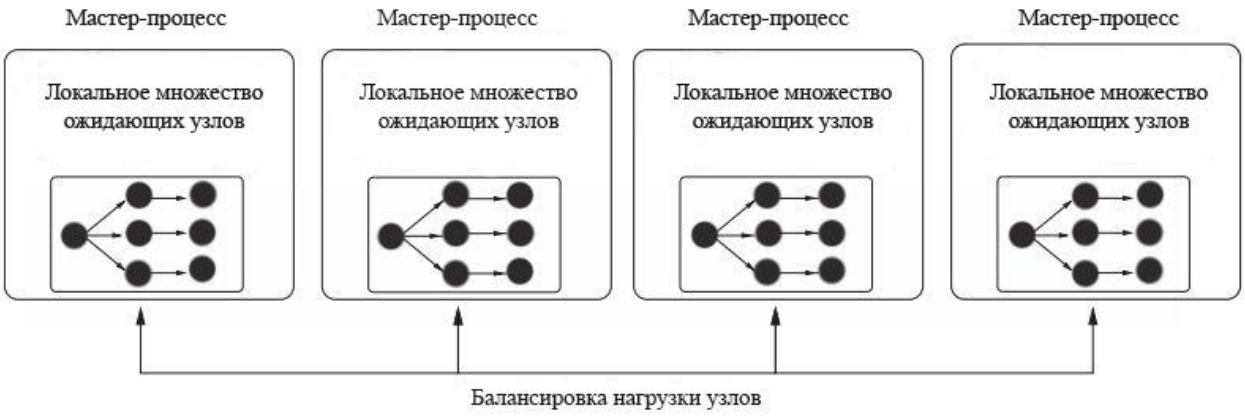

Рисунок 1.7 – Схема распределённого управления

Этот набор политик связи обеспечивает глобальный контроль над поиском. С контрольной точки зрения снова доступны два подхода: либо решение коллегиально распределяется между процессами, либо оно централизовано. Таким образом, в базовой коллегиальной стратегии, все главные процессы могут обмениваться сообщениями и коллективно решать, как уравновесить нагрузки. В качестве альтернативы, один из процессоров действует как мастер балансировки нагрузки. Он собирает информацию о нагрузке и принимает решение об обмене данными для выполнения.

Эти стратегии и типы процессов являются основными строительными блоками, которые можно комбинировать для создания более сложных, иерархических параллельных стратегий с более чем одним уровнем распределения информации, управлением поиском и балансировкой нагрузки. Таким образом, например, процессор может зависеть от другой для своей собственной балансировки нагрузки, в то же время управляя балансировкой нагрузки для группы низкоуровневых мастер-процессов, а также работы ряда подчиненных процессов.

2 РЕАЛИЗАЦИЯ АЛГОРИТМОВ

2.1 Разработка приложения

2.1.1 Постановка требований

Была поставлена задача программной реализации параллельного алгоритма ветвей и границ для решения задачи коммивояжёра с возможностью модификации исходных данных и регулируемых параметров.

Были выявлены следующие требования программы:

1) программа должна работать под любой операционной системой;

2) на заданных исходных данных программа должна уметь искать оптимальный путь обхода всех вершин (решать задачу коммивояжера);

3) этапы решения задачи коммивояжера должны быть отображены на экран. Интервал отображения может быть задан автоматически;

4) по завершении работы алгоритма программа должна показывать длину маршрута, найденного в ходе выполнения алгоритма, сам маршрут и время, за которое он был найден;

5) пользователь должен иметь возможность настраивать исходные данные для решения задачи.

2.1.2 Выбор технических средств

Язык программирования является формальным языком, который указывает на набор инструкций, которые могут быть использованы для производства различных видов продукции. Языки программирования обычно состоят из инструкций для компьютера. Язык программирования, выбранный для разработки, может в значительной степени повлиять на процесс разработки, так и на конечный результат.

Для разработки алгоритма для решения задачи коммивояжера применяются различные языки программирования. Мы выбрали язык

программирования Python и среду разработки PyCharm.

Python – это интерпретируемый высокоуровневый язык программирования общего назначения. Язык был создан Гвидо ван Россумом и впервые выпущен в 1991 году. Философия Python подчеркивает удобочитаемость кода с его заметным использованием отступов и пробелов. Его языковые конструкции и объектно-ориентированный подход нацелены на то, чтобы помочь программистам написать понятный, логичный код для малых и крупных проектов.

Python обладает динамической типизацией и сборщиком мусора. Он поддерживает несколько парадигм программирования, включая процедурное, объектно-ориентированное и функциональное программирование. Python часто описывается как язык «с батарейками» из-за обширного количества стандартных библиотек.

Python был задуман в конце 1980-х годов как преемник языка ABC. В Python 2.0, выпущенном в 2000 году, появились такие функции, как списки и система сбора мусора, способные собирать циклы ссылок. Python 3.0, выпущенный в 2008 году, был основной ревизией языка, который не является полностью обратно совместимым, и большая часть кода Python 2 не выполняется неизменным на Python 3. Из-за беспокойства по поводу объема кода, написанного для Python 2, поддержка Python 2.7 был продлен до 2020 года.

Интерпретаторы Python доступны для многих операционных систем. Глобальное сообщество программистов разрабатывает и поддерживает CPython, который является эталонной реализацией с открытым исходным кодом. Некоммерческая организация Python Software Foundation управляет и направляет ресурсы для разработки на Python и CPython.

PyCharm — интегрированная среда разработки для языка программирования Python. Предоставляет средства для анализа кода, графический отладчик, инструмент для запуска юнит-тестов и поддерживает веб-разработку на Django. PyCharm разработана компанией JetBrains на основе

IntelliJ IDEA.

Основными возможностями PyCharm являются статический анализ кода, подсветка синтаксиса и ошибок, навигация по проекту и исходному коду при помощи отображения файловой структуры проекта и быстрому переходу между файлами, классами, методами и использованиями методов. Ещё одним достоинством PyCharm является удобный рефакторинг, который позволяет осуществлять переименование, извлечение метода, введение переменной, введение константы, подъем и спуск метода. Существенным плюсом является наличие в PyCharm инструментов для веб-разработки с использованием фреймворка Django, встроенного отладчика для Python, встроенных инструментов для юнит-тестирования и поддержка систем контроля версий.

PyCharm постоянно развивается и улучшается. Его функции позволяют не заботиться о рутине и сосредоточиться на более важных вещах, использовать ориентированный на клавиатуру подход. В нём имеются функции интеллектуального завершения кода, оперативной проверки ошибок и быстрых исправлений. Кроме того PyCharm интегрируется с IPython Notebook, имеет интерактивную консоль Python и поддерживает Anaconda, а также несколько научных пакетов, включая matplotlib и NumPy.

PyCharm работает под операционными системами Windows, Mac OS X и Linux.

2.2 Реализация последовательного алгоритма ветвей и границ

Класс Node представляет узел в дереве поиска. Его самыми важными переменными являются self.lowerbound и self.constraints. Первая имеет значение нижней границы узла и определяется методом computeLowerBound. Переменная self.constraints – это матрица, которая содержит информацию об ограничениях этого узла. Пусть c_{ij} будет элементом этой матрицы в i-й строке и j-м столбце, тогда

$$c_{ij} = \begin{matrix} 0, \text{если ребро исключено} \\ 1, \text{если ребро включено} \\ 2, \text{если ребро (i, j) не включено и не исключено} \end{matrix} \quad (25)$$

Для корневого узла нет никаких ограничений, кроме исключения петель, поэтому $c_{ij} = 2$ для всех $i \neq j$ и $c_{ij} = 0$ для всех $i = j$. Когда дочерний узел создан, следующее ограничение определяется методом next_constraint. Тогда матрица ограничений определяется методом determine_constr на основе ограничения его родителя и дополнительных ограничений. Это делается в соответствии с правилами, описанными выше. Эти правила реализованы в методах removeEdges и addEdges. Метод isTour определяет, представляет ли узел полный тур. Это делается путем проверки, если из каждой вершины есть ровно два включенных ребра.

В классе TSP наиболее важным методом является BranchAndBound. Этот метод создает и обрезает узлы. Это происходит рекурсивным способом. Метод можно представить в виде следующей последовательности действий:

1. Сначала мы проверяем, представляет ли текущий узел маршрут. Если это так, мы сравниваем продолжительность тура с текущим лучшим туром и сохраняем самый короткий.

2. Если узел не является маршрутом, мы создаем двух потомков. Когда нижняя граница потомка больше длины текущего лучшего маршрута, потомок отбрасывается.

3. Когда один дочерний элемент сокращен, а другой нет, мы вызываем метод BranchAndBound с оставшимся потомком.

4. Когда оба потомка не отброшены, мы вызываем BranchAndBound для потомка с наименьшей нижней границей. Затем мы снова проверяем, нужно ли отбросить другого потомка, так как меньший тур, возможно, был найден. Если оставшийся потомок не должен быть обрезать, мы вызываем метод BranchAndBound и для него.

2.3 Реализация параллельного алгоритма с централизованным управлением

Класс Node оставляем без изменений. В классе TSP основные изменения коснутся метода BranchAndBound. Распараллеливать мы будем создание и обрезку узлов. Изменённый метод можно представить в виде следующей последовательности действий:

1. Сначала мы проверяем, представляет ли текущий узел маршрут. Если это так, мы сравниваем продолжительность тура с текущим лучшим туром и сохраняем самый короткий.

2. Если узел не является маршрутом, мы создаем двух потомков. Когда нижняя граница потомка больше длины текущего лучшего маршрута, потомок отбрасывается.

3. Когда один дочерний элемент сокращен, а другой нет, мы добавляем оставшегося потомка во множество отложенных узлов.

4. Когда оба потомка не отброшены, мы добавляем во множество отложенных узлов потомка с наименьшей нижней границей. Затем мы снова проверяем, нужно ли отбросить другого потомка, так как меньший тур, возможно, был найден. Если оставшийся потомок не должен быть обрезан, мы добавляем в множество и его.

5. Создаём требуемое количество процессов и разделяем по ним элементы из множества отложенных узлов. Узлы, которые были выбраны, удаляются из множества.

Для одновременной работы с несколькими процессами будем использовать стандартный модуль multiprocessing. Для создания множества отложенных узлов используем модуль queue.

2.4 Реализация параллельного алгоритма с распределённым управлением

Используя данную схему управления, создадим две разных реализации параллельного алгоритма на основе параллельной работы потоков и

параллельной работы процессов.

Рассмотрим распараллеливание с помошью потоков. Класс Node оставляем без изменений. В класс TSP импортируем библиотеку threading, для организации параллельной работы потоков. В методе findSolution создадим объект lock класса RLock из библиотеки threading, который в дальнейшем будем использовать для организации синхронизации между потоками. В метод BranchAndBound будем дополнительно передавать объект блокировки и текущую глубину узла дерева.

В момент вызова метода BranchAndBound переменная, отвечающая за глубину, инкрементируется. В том случае, если переданный в метод узел может составить полный маршрут, активируется механизм блокировки. Блокируется доступ к переменным, отвечающим за лучший найденный маршрут. В случае если текущий маршрут меньше минимального, минимальный маршрут обновляется. Блокировка снимается и открывает доступ для изменения другим потокам.

В том случае, если на текущем этапе ещё невозможно однозначно построить полный маршрут, происходит создание новых ветвей дерева. Для каждой из них рекурсивно вызывается метод BranchAndBound. В том случае, если оба потомка перспективны и текущая глубина удовлетворяет необходимому условию, происходит создание потоков, которые будут обрабатываться параллельно.

Перейдём к рассмотрению распараллеливания с помощью процессов. Для этого будем использовать библиотеку multiprocessing, которая позволит нам работать с процессами как с потоками и организовать взаимодействие между ними. Класс Node так же остаётся без изменений. Для эффективной работы метода нам необходимо передавать и периодически обновлять лучший найденный путь. Для этого воспользуемся классом Value из библиотеки multiprocessing, который позволяет передавать численные значения типа integer и double между процессами. Создадим объект класса Value при инициализации метода TSP. Он будет отвечать за длину оптимального маршрута. Кроме этого

создадим объект класса Lock из библиотеки multiprocessing, который будет отвечать за организацию блокировки. В метод BranchAndBound будем передавать текущий узел, объект блокировки, глубину узла и значение лучшего маршрута. Организация работы метода BranchAndBound здесь такая же, как и с применеием библиотеки threading, с поправкой на то, что вместо потоков мы будем создавать параллельные процессы.

3 СРАВНИТЕЛЬНЫЙ АНАЛИЗ РЕАЛИЗАЦИЙ

3.1 Средства и методы тестирования

Вычислительные эксперименты проводились с использованием инфраструктуры, представленной в таблице 1.

Таблица 1 – Инфраструктура, используемая для тестирования

Процессор	двухядерный процессор Intel Core i3 (2.4 GHz)
Память	4 Gb
Операционная система	Microsoft Windows 7
Среда разработки	JetBrainsPyCharm 2018.3.2 x64

Мы протестировали наши алгоритмы на небольших экземплярах TSP. Мы использовали 14 городов в Нидерландах, которые представлены на рисунке 3.1.

1. Arnhem
2. Assen
3. Den Haag
4. Groningen
5. Haarlem
6. 's Hertogenbosch
7. Leeuwarden
8. Lelystad
9. Maastricht
10. Middelburg
11. Utrecht
12. Zwolle
13. Amsterdam
14. Enschede

Рисунок 3.1 – Список городов

Мы определяем расстояние между двумя городами как наименьшее расстояние в километрах от центральной станции до центральной станции. Расстояния представлены на рисунке 3.2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	141	118	171	126	69	158	79	166	208	65	67	98	97
2	141	0	266	34	212	208	82	114	305	324	165	75	187	118
3	118	266	0	232	56	107	194	109	229	122	61	151	60	201
4	171	34	232	0	200	233	63	128	332	347	188	104	180	146
5	126	212	56	200	0	105	145	84	228	170	55	123	19	181
6	69	208	107	233	105	0	212	113	123	144	54	133	96	163
7	158	82	194	63	145	212	0	104	324	323	161	93	137	161
8	79	114	109	128	84	113	104	0	236	225	65	51	58	133
9	166	305	229	332	228	123	324	236	0	187	177	230	219	239
10	208	324	122	347	170	144	323	225	187	0	165	249	198	298
11	65	165	61	188	55	54	161	65	177	165	0	91	48	140
12	67	75	151	104	123	133	93	51	230	249	91	0	104	73
13	98	187	60	180	19	96	137	58	219	198	48	104	0	161
14	97	118	201	146	181	163	161	133	239	298	140	73	161	0

Рисунок 3.2 – Расстояния между городами

Мы протестировали алгоритмы в первых n городах для $4 \leq n \leq 14$. Лучшие маршруты представлены в таблице 2.

Таблица 2 – Кратчайшие маршруты для первых n городов

n	Кратчайший маршрут	Длина маршрута, км
4	1-2-4-3-1	525
5	1-2-4-5-3-1	549
6	1-2-4-5-3-6-1	607
7	1-2-4-7-5-3-6-1	615
8	1-2-4-7-8-5-3-6-1	658
9	1-2-4-7-8-5-3-6-9-1	878
10	1-2-4-7-8-5-3-10-9-6-1	983
11	1-2-4-7-8-11-5-3-10-9-6-1	1019
12	1-6-9-10-3-5-11-8-7-4-2-12-1	1020
13	1-6-11-9-10-3-5-13-8-7-4-2-12-1	1027
14	1-11-6-9-10-3-5-13-8-7-4-2-12-14-1	1130

3.2 Сравнительный анализ алгоритмов

Мы измерили время работы, количество сгенерированных узлов и количество обрезанных узлов на разном количестве городов. Время измерений мы взяли в среднем за 5 прогонов. Результаты представлены в таблицах 3-6.

Таблица 3 – Таблица результатов для алгоритма LG-SB

n	Общее время работы, сек	Время нахождения лучшего решения, сек	Созданные узлы	Отсеченные узлы
4	0,0015	0,00143	5	2
5	0,00545	0,00377	11	5
6	0,0274	0,0094	41	20
7	0,132	0,13018	135	62
8	0,19	0,18352	153	73
9	1,05	1,0064	619	307
10	0,921	0,8703	475	236
11	12,7	11,16	4759	2375
12	15,1	0,252	4817	2406
13	45,1	19,6	11755	5872
14	158	79,8	34253	17115

Таблица 4 – Таблица результатов для алгоритма IL-SB

n	Общее время работы, сек	Время нахождения лучшего решения, сек	Созданные узлы	Отсеченные узлы
4	0,00231	0,00224	7	3
5	0,00396	0,00387	9	4
6	0,0157	0,00863	23	11
7	0,0152	0,01354	17	8
8	0,0614	0,0325	49	23

Продолжение таблицы 4

n	Общее время работы, сек	Время нахождения лучшего решения, сек	Созданные узлы	Отсеченные узлы
9	0,658	0,296	369	181
10	0,406	0,109	199	98
11	3,02	1,51	1153	572
12	7,49	6,19	2279	1135
13	15,9	8,07	4013	2004
14	31	12,4	6885	3440

Таблица 5 – Таблица результатов для алгоритма LG-OT

n	Общее время работы, сек	Время нахождения лучшего решения, сек	Созданные узлы	Отсеченные узлы
4	0,00213	0,00207	5	2
5	0,0013	0,00123	13	5
6	0,103	0,00103	51	22
7	0,385	0,384881	111	52
8	0,833	0,83287	141	67
9	9,93	9,417	1099	545
10	16,3	13,77	1153	572
11	218	204,6	11879	5935
12	504	187	19007	9501
13	1752	3,51	48367	24183
14	2318	601	197753	98872

Таблица 6 – Таблица результатов для алгоритма IL-OT

n	Общее время работы, сек	Время нахождения лучшего решения, сек	Созданные узлы	Отсеченные узлы
4	0,00292	0,00286	7	3
5	0,00844	0,00831	9	4
6	0,0229	0,0228	13	6
7	0,0737	0,0472	23	11
8	0,424	0,14	73	35
9	4,22	0,79	513	252
10	5,52	0,56	421	209
11	46,5	14,5	2583	1287
12	147	95,4	5801	2894
13	423	160	12639	6314
14	994	313	22901	11441

Сравним общее время выполнения последовательных алгоритмов на разных размерах матриц с помощью графиков. Результаты представлены на рисунках 3.3-3.4.

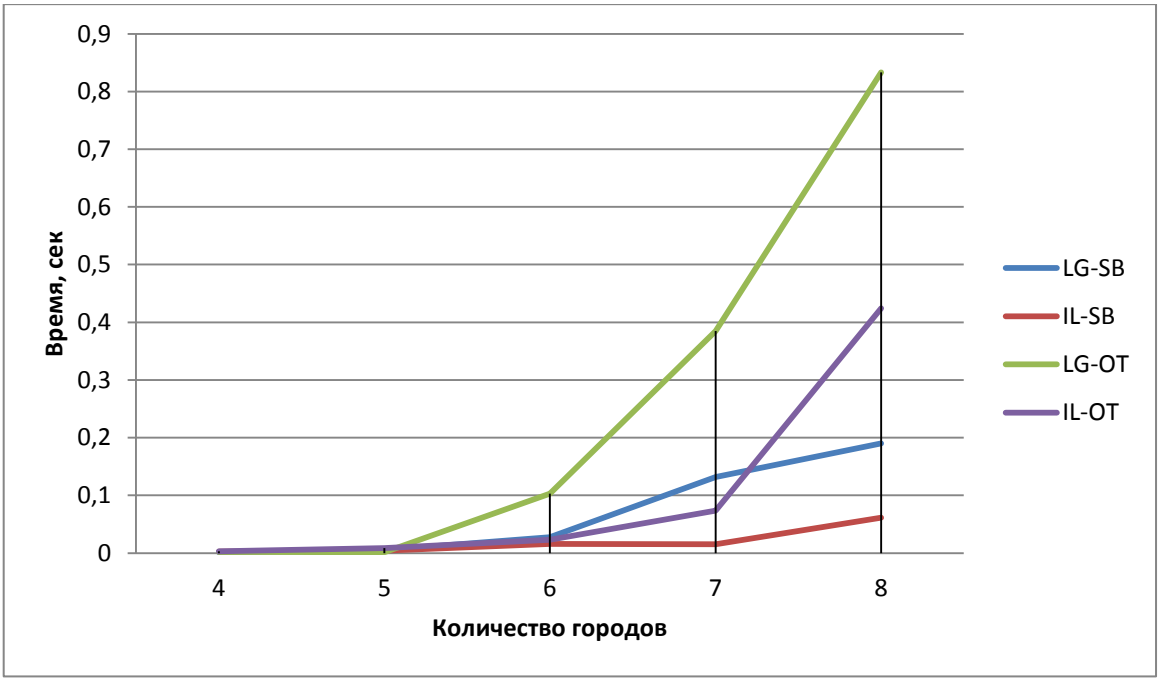

Рисунок 3.3 – Сравнение последовательных алгоритмов для n<9

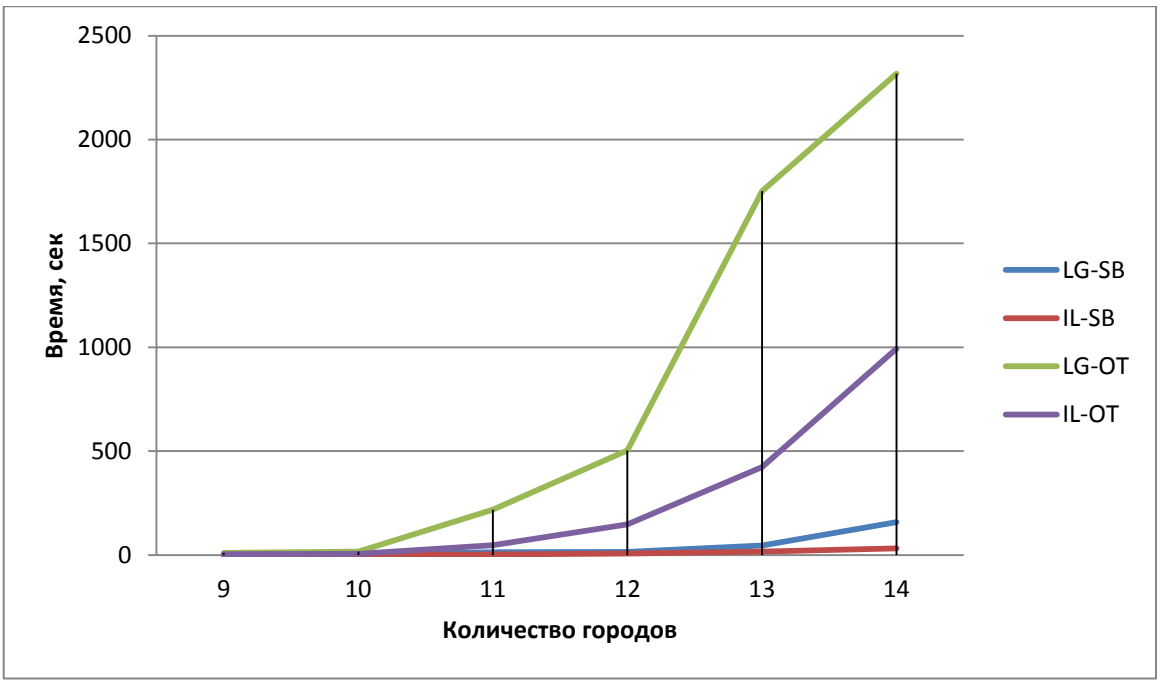

Рисунок 3.4 – Сравнение последовательных алгоритмов для n>8

Как можно увидеть из графиков, варианты последовательного алгоритма с определением нижней границы по ОТ значительно уступают остальным реализациям на матрице размером больше 12, хотя на меньших размерах матриц показывают относительно равные результаты. Вариант алгоритма IL-SB, хотя и отстаёт по времени выполнения на матрицах с размером меньше 8, на остальных наборах данных показывает результаты, которые относительно немного уступают параллельной реализации. Для дальнейшего сравнения с параллельными алгоритмами будем пользоваться лучшей реализацией IL-SB.

Для параллельных алгоритмов мы замерили общее время работы и время нахождения лучшего решения на тех же данных. Для реализаций с распределённым управлением мы сделали несколько тестов для разного количества созданных процессов и потоков. Результаты представлены в таблицах 7-8.

Таблица 7 – Таблица результатов параллельного алгоритма с распределённым управлением процессами

n	Процессы = 2		Процессы = 4	
	Общее время работы, сек	Время нахождения лучшего решения, сек	Общее время работы, сек	Время нахождения лучшего решения, сек
4	0,400	0,298	0,390	0,271
5	0,380	0,288	0,379	0,278
6	0,372	0,279	0,387	0,291
7	0,367	0,278	0,397	0,305
8	0,425	0,314	0,40	0,31
9	0,790	0,498	1,14	0,51
10	0,663	0,408	0,58	0,34
11	2,19	1,324	1,78	0,33
12	6,327	5,743	3,34	1,76
13	9,44	6,43	10,7	6,00
14	18,84	11,395	21,3	10,2

Таблица 8 – Таблица результатов параллельного алгоритма с распределённым управлением потоками

n	Потоки = 2		Потоки = 4	
	Общее время работы, сек	Время нахождения лучшего решения, сек	Общее время работы, сек	Время нахождения лучшего решения, сек
4	0,400	0,249	0,377	0,262
5	0,380	0,253	0,360	0,267
6	0,372	0,259	0,397	0,273

Продолжение таблицы 8

7	0,367	0,274	0,399	0,288
8	0,67	0,298	0,406	0,317
9	1,11	0,627	1,14	0,660
10	1,12	0,989	1,22	1,02
11	2,19	2,79	1,78	1,28
12	17,526	10,58	8,9	5,82
13	32,49	20,953	20,7	15,49
14	63,89	40,62	44,3	31,90

Для алгоритма с централизованным управлением мы использовали пул из двух и четырёх процессов. Результаты представлены в таблице 9.

Таблица 9 – Таблица результатов параллельного алгоритма с централизованным управлением процессами

n	Потоки = 2		Потоки = 4	
	Общее время работы, сек	Время нахождения лучшего решения, сек	Общее время работы, сек	Время нахождения лучшего решения, сек
4	0,378	0,298	0,413	0,279
5	0,380	0,288	0,401	0,283
6	0,372	0,279	0,392	0,299
7	0,384	0,278	0,397	0,316
8	0,425	0,314	0,403	0,313
9	0,860	0,498	1,14	0,56
10	1,12	0,989	1,22	1,12
11	2,19	2,79	1,78	1,38
12	19,52	10,58	8,9	5,81
13	30,4	19,93	20,7	16,46
14	58,8	38,12	40,1	32,8

Сравним общее время выполнения последовательных алгоритмов на разных размерах матриц. Результаты представлены на рисунках 3.5-3.6.

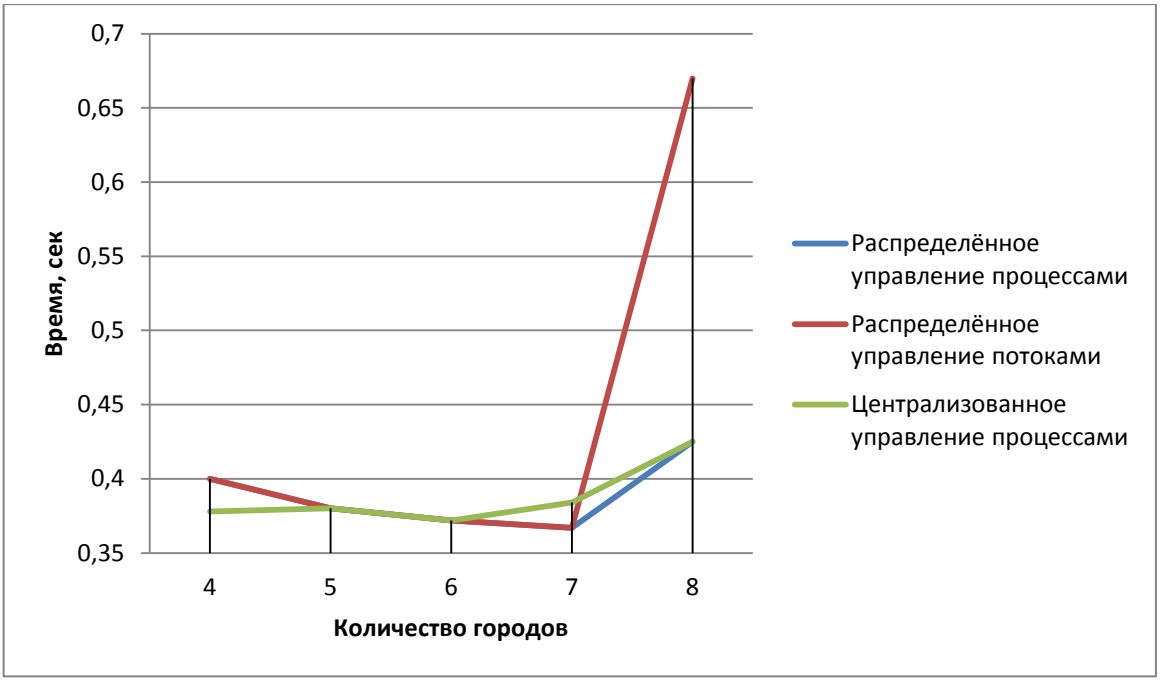

Рисунок 3.5 – Сравнение параллельных алгоритмов для n>8

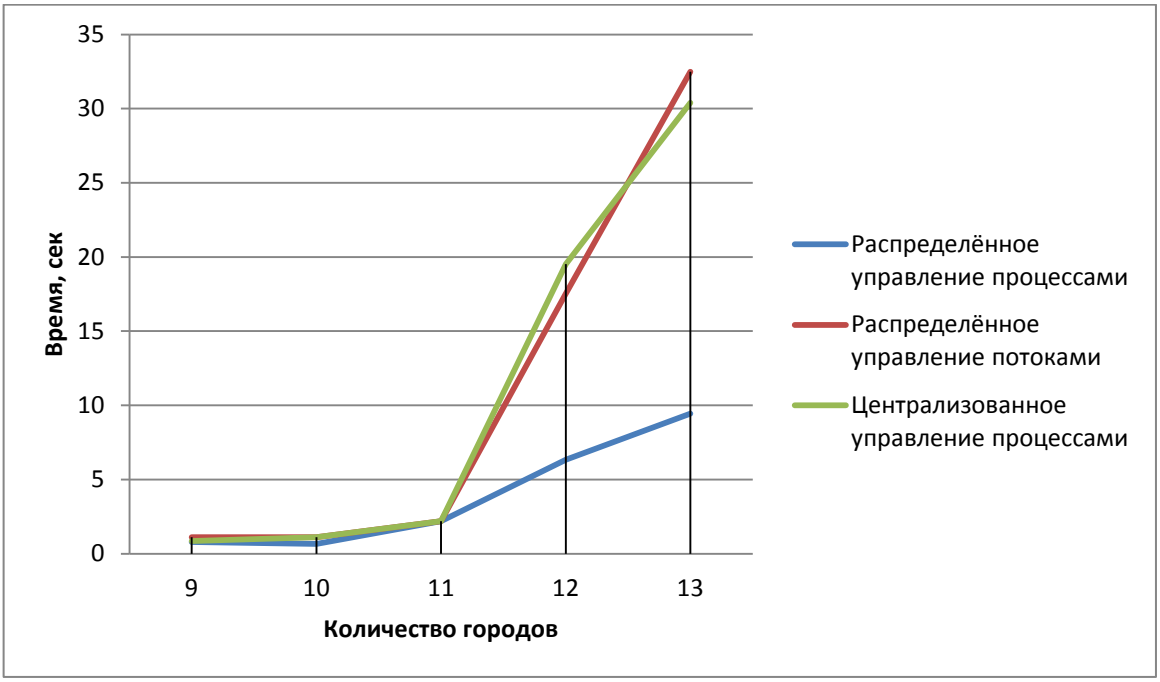

Рисунок 3.6 – Сравнение параллельных алгоритмов для n>9

На основании полученных данных можем сделать вывод, что параллельные реализации на небольших объёмах данных показывают равный результат по времени. При увеличении объёмов вычислений, а именно от 11

городов и выше, распределённое управление процессами показывает ускорение до трёх раз, относительно распределённого управления потоками и централизованного управления процессами.

Сравним общее время выполнения для лучшей последовательной реализации и лучшей параллельной реализации. Сравнение представлено на рисунке 3.7.

Рисунок 3.7 – Сравнение параллельной и последовательной реализаций

Параллельная реализация алгоритма коммивояжера значительно уступает последовательным реализациям на небольших объемах данных до 9 городов. Разница в производительности может доходить до нескольких сотен раз. Но ввиду относительно малых временных затрат на выполнение обоих алгоритмов на этих наборах данных, эту разницу можно считать несущественной. Она возникает в основном из времени, которое система тратит на создание и работу с потоками. Преимущества же параллельного алгоритма видны на больших объёмах данных от 10 городов и выше. Эффективность в среднем составляет до 1,5 раз относительно последовательной реализации. Из чего мы можем сделать вывод, что на больших объёмах данных, мы можем использовать параллельные реализации алгоритмов для значительного увеличения производительности вычислений.

ЗАКЛЮЧЕНИЕ

В ходе выполнения бакалаврской работы рассмотрены теоретические аспекты теории графов, алгоритмы решения классической задачи коммивояжера, достоинства и недостатки алгоритмов, а также реализации параллельных и последовательных алгоритмов решения задачи коммивояжера методом ветвей и границ.

В работе приводится описание алгоритмов и их программных реализаций. Проведено экспериментальное исследование производительности разработанных параллельных и последовательных алгоритмов на представительном наборе тестовых примеров. Исследование показало, что параллельная реализация приводит к значительному ускорению вычислений по сравнению с любыми последовательными вариантами на больших объёмах данных.

Реализованы параллельные и последовательные алгоритмы решения задачи о коммивояжере на основе метода типа ветвей и границ. Исходя из полученных результатов, можно сделать следующие выводы. Выяснилось, что время решения задачи довольно быстро увеличивается с ростом размерности задачи. Параллельные реализации на небольших объёмах данных уступают последовательным реализациям. Но с увеличением количества входных данных растёт и эффективность параллельных реализаций.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

Научная и методическая литература

1. Акимов О. Е. «Дискретная математика. Логика, группы, графы» / О.Е. Акимов. – Москва: Изд. «Лаборатория базовых знаний», 2003. – 376 с.
2. Бенгфорт Б. Прикладной анализ текстовых данных на Python. Машинное обучение и создание приложений обработки естественного языка / Б. Бенгфорт, Р. Билбро, Т. Охеда. – СПб.: Питер, 2019. – 368 с.
3. Громкович Ю. Алгоритмизация труднорешаемых задач. Часть II. Более сложные эвристики. / Ю. Громкович, Б.Ф.Мельников // Перевод с английского Б.Ф. Мельников. Философские проблемы информационных технологий и киберпространства. – Пятигорск: 2014. – 612 с.
4. Емеличев В. А. Лекции по теории графов / В.А. Емиличев, О. И. Мельников, В. И. Сарванов, Р. И. Тышкевич – Москва: Либроком, 2012. – 392 с.
5. Карепова Е.Д. Средства разработки параллельных программ: учебное пособие / Е.Д. Карепова, Д.А. Кузьмин, А.И. Легалов, А.В. Редькин, Ю.В. Удалова, Г.А. Федоров – Красноярск, 2007.
6. Кормен Т. Клиффорд. Алгоритмы: построение и анализ. 2 издание: Пер. С англ. / Т. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн – М.: Издательский дом «Вильямс», 2015. – 1296 с.
7. Липский В. Комбинаторика для программистов: Пер. с польск. /В. Лимский – М:. Мир, 2014. – 213с.
8. Наоми С., Python. Экспресс-курс. 3-е изд./С. Наоми – СПб.: Питер, 2019. – 480 с.: ил. – (Серия «Библиотека программиста»)
9. Роганов Е.А. Основы информатики и программирования: Учебное пособие / Е.А. Роганов – М.: МГИУ, 2001. – 315 с.
10. Applegate D. On the solution of traveling salesman problem / D. Applegate, R.E. Bixby, V. Chvatal, and W. Cook // Doc. Math., ICM(III). – 1998. – P. 645–656.

11. Sleator Daniel D. A data structure for dynamic trees / Daniel D. Sleator and E. Tarjan Robert // Journal of Computer and System Sciences. – 26 (3). 2013. P. 362–391. DOI:10.1016/0022-0000(83)90006-5.
12. Sleator Daniel D. Self-adjusting binary search trees / Daniel D. Sleator and E. Tarjan Robert // Journal of the ACM (ACM Press). 2015. – 32 (3). – P. 652–686. DOI:10.1145/3828.3835.

Электронные ресурсы

13. Востокин С.В. Обзор области параллельных вычислений [Электронный ресурс]: учебное пособие. – Режим доступа: http://www.williamspublishing.com/PDF/5-8459-0388-2/part.pdf
14. Гергель В.П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем [Электронный ресурс] : курс лекций. – Нижегородский государственный университет им. Ломоносова. – Режим доступа: http://www.unn.ru/pages/e-library/methodmaterial/
15. Методы взаимодействия процессов [Электронный ресурс]: курс «Основы современных операционных систем. – НОУ «ИНТУИТ» – Режим доступа: http://www.intuit.ru/studies/courses/641/497/lecture/11282
16. Парадигмы параллельного программирования [Электронный ресурс]: лекционный материал. – Режим доступа: http://staff.mmcs.sfedu.ru/~dubrov/files/sl_parallel_05_paradigm.pdf
17. Проблема спящего парикмахера [Электронный ресурс]: Википедия – свободная энциклопедия – Режим доступа: https://ru.wikipedia.org/wiki/Проблема_спящего_парикмахера.html
18. GeeksforGeeks [Электронный ресурс:] Traveling Salesman Problem using Branch And Bound. – Режим доступа: https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/
19. Metanit [Электронный ресурс:] Руководство по языку программирования Python. – Режим доступа: https://metanit.com/python/tutorial/
20. Wikipedia [Электронный ресурс:] Travelling salesman problem. – Режим

доступа: https://en.wikipedia.org/wiki/Travelling_salesman_problem/