Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

ПНИПУ

УДК 004

**Отчет по лабораторной работе**

**«Решение задачи коммивояжёра и его визуализация»**

Выполнил

Студент группы РИС-23-1б

Хасанишин А.Р

Проверила доц. кафедры ИТАС

Полякова О.А.

г. Пермь, 2024 г.

# **СОДЕРЖАНИЕ**

[**СОДЕРЖАНИЕ** 2](#_Toc165548351)

[**ВВЕДЕНИЕ** 3](#_Toc165548352)

[**1 АЛГОРИТМ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЁРА** 5](#_Toc165548353)

[**2 РАЗРАБОТКА КОДА НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ С++** 6](#_Toc165548354)

[**2.1 АЛГОРИТМ РЕШЕНИЯ ЗАДАЧИ** 6](#_Toc165548355)

[**2.2 НАПИСАНИЕ КОДА ДЛЯ РЕШЕНИЯ ЗАДАЧИ** 7](#_Toc165548357)

[**3. ВИЗУАЛИЗАЦИИ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЁРА** 8](#_Toc165548359)

[**4. ПРИМЕР РАБОТЫ ВИЗУАЛИЗАЦИИ…………………………………………………………...** 9](#_Toc165548359)

[**ЗАКЛЮЧЕНИЕ** 10](#_Toc165548360)

[**СПИСОК ИСТОЧНИКОВ** 11](#_Toc165548361)

# **ВВЕДЕНИЕ**

Задача коммивояжёра (Traveling Salesman Problem, TSP) считается одной из основных проблем в вычислительной математике. Она может быть описана следующим образом: дан набор городов, которые должен посетить коммивояжёр, только один раз, прибывая и уезжая один раз, и начиная и заканчивая в том же городе. Эта задача имеет широкий спектр применений, включая планирование маршрутов, логистику, проектирование микрочипов и даже в области ДНК-секвенирования.[1]

Решение задачи коммивояжёра также является сложной задачей, требующей применения различных оптимизационных и алгоритмических подходов. Существует множество методов для решения TSP, включая методы полного перебора, приближенные алгоритмы, генетические алгоритмы, муравьиные алгоритмы и другие.

Уильям Роуэн Гамильтон, известный ирландский математик, физик и механик XIX века, был одним из первых, кто предложил раннюю формулировку этой задачи. Он проводил исследования в области симметрии икосаэдра (двадцатигранника) и в 1857 году создал игру под названием «Икосиан» (англ. «Icosian Game»). Цель игры состояла в том, чтобы пройти через все вершины додекаэдра (двенадцатигранника) ровно один раз, двигаясь только по его ребрам, и затем вернуться в исходную точку. В других словах, было необходимо найти так называемый гамильтонов цикл на графе с 20 узлами, который в данном случае являлся решением задачи и содержал 20 ребер («двадцать» на древнегреческом «icosa», откуда и произошло название игры).[2]

Изучение задачи коммивояжера не только позволяет находить оптимальные или приближенные решения для конкретных сценариев, но и изучать общие принципы оптимизации пути и маршрутов. Решение TSP играет важную роль в разработке алгоритмов для других комбинаторных задач и служит основой для изучения сложных оптимизационных проблем в области искусственного интеллекта и логистики. Данная лабораторная работа заключается в анализе методов решения задачи коммивояжёра, исследовании их эффективности, изучении возможных оптимизаций, а также она включает в себя поиск наиболее эффективной визуализации решения. Исследование эффективности алгоритмов коммивояжёра требует изучения их скорости работы, точности результата, способности к решению больших задач и т.д.

Таким образом, выполнение данной лабораторной работы способствует развитию не только теоретических знаний, но и практических навыков в области разработки и оптимизации алгоритмов.

Объект исследования: вычислительный процесс нахождения оптимального пути методом ветвей и границ.

Предмет исследования: программа-приложение, определяющая кратчайший путь с помощью метода ветвей и границ для решения задачи коммивояжера.

Цель: реализация программы на языке программирования С++ для решения задачи коммивояжёра, выполнение визуализации решения.

Задачи:

1. Провести анализ алгоритма решения;
2. Разработать код на языке программирования С++;
3. Разработать визуализацию решения задачи коммивояжёра.
4. Провести эксперимент

# 

# **Алгоритм решения задачи коммивояжера**

Перед тем как решать задачу коммивояжёра, мы должны проанализировать метод ветвей и границ, который нужно использовать для данной лабораторной работы.

**Метод ветвей и границ**:

Метод ветвей и границ, предложенный в 1963 году группой ученых, является специфическим случаем метода поиска с ограничениями. Он основан на предположении о наличии функции цены на множестве возможных решений, и целью является поиск решения с минимальной ценой. Метод включает в себя последовательное разбиение множества допустимых решений на подмножества и проверку каждого подмножества на наличие оптимального решения. Если все элементы разбиения отбрасываются, то текущий рекорд является оптимальным решением. В противном случае, выбирается наиболее перспективное подмножество для дальнейшего разбиения и проверки. Успех метода объясняется его гибкостью и способностью адаптироваться к специфике задачи.[3]

**Анализ:**

**Сильные стороны:**

1. Гарантированное нахождение оптимального решения (при правильной реализации): Метод гарантирует нахождение оптимального решения, если оно существует, при условии правильного определения нижней оценки.
2. Эффективность: Метод отсекает целые ветви пространства поиска, которые не могут содержать оптимальное решение, что существенно сокращает количество рассматриваемых вариантов.
3. Гибкость: Метод можно адаптировать к различным задачам оптимизации, используя соответствующие функции оценки.

**Слабые стороны:**

1. Сложность реализации: Метод ветвей и границ может быть сложным в реализации, так как требует тщательного выбора функции оценки и стратегии ветвления.
2. Зависимость от функции оценки: Эффективность метода сильно зависит от качества функции оценки. Неточная или слишком слабая оценка может привести к рассмотрению большого количества лишних вариантов.
3. Потенциально большой объем памяти: Метод может требовать значительного объема памяти для хранения информации о ветвях и оценках.

Таким образом, для задач с небольшим пространством поиска, где требуется гарантированно найти наилучшее решение, полный перебор может быть вполне приемлемым. Если же пространство поиска обширно и допустимо довольствоваться "хорошим" решением, то случайный перебор, особенно в сочетании с эффективными эвристиками, станет предпочтительным выбором. Для сложных задач оптимизации, требующих оптимального решения и допускающих хорошую функцию оценки, метод ветвей и границ представляет собой наиболее эффективный подход, несмотря на сложность его реализации.

# **2 Разработка кода на языке программирования с++**

# **2.1 Алгоритм решения задачи**

**Алгоритм решения:**

1. Составление матрицы смежности;
2. Нахождение минимума по строкам;
3. Редукция строк;
4. Нахождение минимума по столбцам;
5. Редукция столбцов;
6. Нахождение оценок для нулевых элементов;
7. Редукция матрицы;
8. Выбор: Если мы еще не нашли все отрезки пути, которые позволяют вернуться Коммивояжеру в исходный город, то возвращаемся к шагу Если все отрезки пути найдены или оставшаяся часть очевидна – переходим к заключительному шагу – соединение путей. В реалиях данной задачи необходимо перейти к шагу 2.;
9. Построение маршрута;

10)Вычисление длины пути.

Таким образом, чтобы правильно решить задачу коммивояжёра методом ветвей и границ, нужно чётко следовать алгоритму. Написание алгоритма для решения задачи помогает структурировать последующую работу, что облегчает выполнение лабораторной работы.

**2.2 Написание кода для решения задачи**

Для того, чтобы написать код следуя выбранному алгоритму, нужно выбрать платформу, на которой будет происходить реализация. Для выполнения данной работы был использован Фреймворк QT

Qt — это мощный фреймворк для разработки приложений. Он обладает кроссплатформенностью, что дает возможность без изменения кода запускать приложение в различных операционных системах. Так же Qt обеспечивает стабильную и быструю работу приложений, он хорошо оптимизирован и минимизирует задержки. Программисту в фраемворке доступен механизм сигналов и слотов - это ключевая особенность Qt. Сигналы генерируются объектами при определенных событиях, а слоты — это методы, которые реагируют на эти сигналы. Это обеспечивает эффективное взаимодействие между компонентами приложения. В дополнение ко всему этому данный фреймворк обладает встроенной IDE для сборки и отладки приложений и в него встроена графическая библиотека OpenGl, что упрощает процесс написания визуализации, потому что не нужно дополнительно устанавливать библиотеку из сторонних мест.[4]

Таким образом, выбор Qt в качестве платформы для лабораторной работы, дает доступ к мощному инструменту с широким набором функций, упрощающих процесс разработки программного обеспечения.

Код решения задачи коммивояжёра рассмотрен в приложении А. UML таблица рассмотрена в приложении Б.

**3 Визуализация решения задачи коммивояжера**

Визуализация играет ключевую роль, преобразуя сложные структуры данных и алгоритмы в интуитивно понятные графические представления. Это позволяет разработчикам анализировать эффективность алгоритмов, выявлять узкие места, оптимизировать производительность и эффективно доносить сложные технические концепции до коллег и пользователей. Для визуализации решения задачи коммивояжёра была использована графическая библиотека OpenGL.

OpenGL является на данный момент одним из самых популярных программных интерфейсов (API) для разработки приложений в области двумерной и трехмерной графики. Стандарт OpenGL был разработан и утвержден в 1992 году ведущими фирмами в области разработки программного обеспечения, а его основой стала библиотека IRIS GL, разработанная Silicon Graphics. [5]

В целом, использование OpenGL для визуализации решения задачи коммивояжёра способствует более глубокому пониманию алгоритма, эффективной коммуникации результатов и принятию взвешенных решений.

**4 Пример работы визуализации**

Примером для визуализации послужит неориентированный граф с 6 вершинами(рисунок. 1)

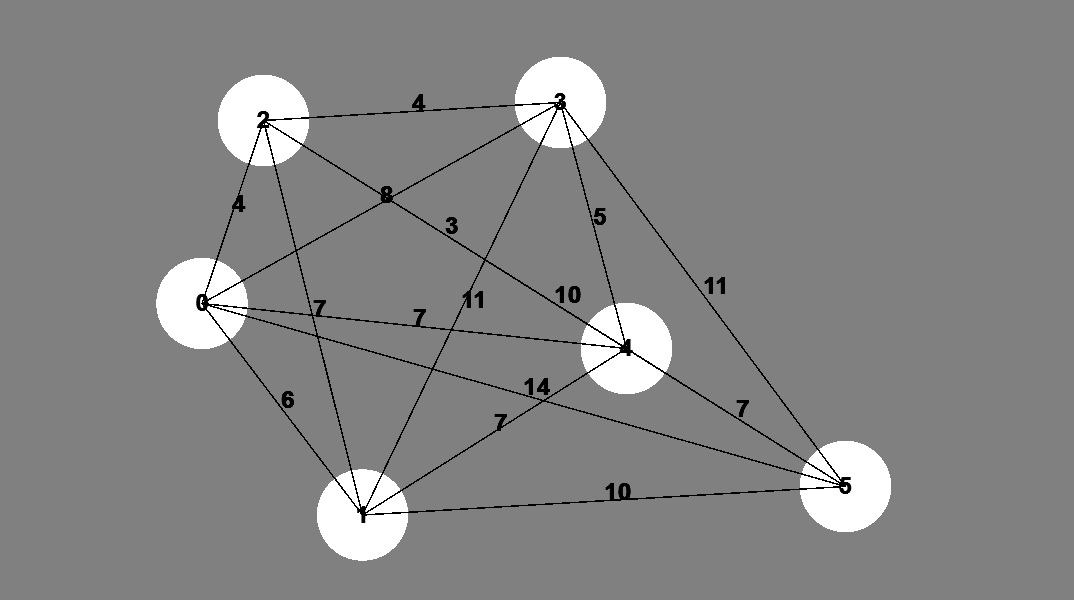


Рисунок 1 – неориентированный граф с 6 вершинами

После вызова метода, решающего задачу коммивояжера программа выводит матрицу смежности и пройденные вершины и расстояние.(рисунок 2)

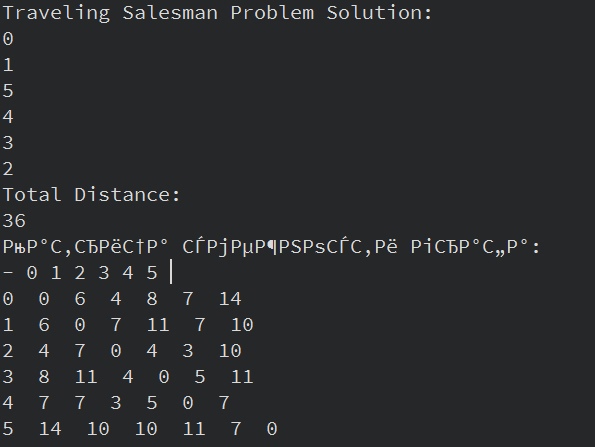


Рисунок 2 – матрица смежности и задача коммивояжера для графа

Исходя из полученных данных, можно сказать, что программа корректно добавляет вершины, грани и веса, а так же выводит вершины и пройдённый путь для задачи коммивояжера

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения лабораторной работы был разработан программный код, реализующий алгоритм решения задачи коммивояжера. Код успешно прошел тестирование и продемонстрировал корректную работу, находя оптимальный маршрут минимальной длины. Для наглядного представления решения была реализована визуализация с использованием графической библиотеки OpenGL, что позволило детально изучить найденный маршрут, проанализировать его эффективность и выявить возможные оптимизации.

Разработанное программное обеспечение с успехом решает поставленную задачу и может быть использовано для анализа и оптимизации логистических цепочек и транспортных сетей.

**СПИСОК ИСТОЧНИКОВ**

1. Методы решения задачи коммивояжёра и ее история. -URL: [https://www.intechopen.com/chapters/7400 (дата](https://www.intechopen.com/chapters/7400%20(дата) обращения 12.05.24)
2. Graph Theory 1736-1936 // Norman L. Biggs - 2013. - №3. – С.102.
3. Метод ветвей и границ. – URL: <https://habr.com/ru/articles/560468/(дата> обращения 12.05.24)
4. Фреймворк QT. – URL: https://www.qt.io
5. Кроссплатформенная библиотека для визуализации 2d и 3d графики. – URL: <https://www.rsdn.org/article/opengl/ogltutor.xml>(дата обращения 12.05.24)

**ПРИЛОЖЕНИЕ А**

**(код для задачи коммивояжёра)**

#ifndef SALE\_MAN\_H

#define SALE\_MAN\_H

#include "graph.h"

using namespace std;

template <class T>

class sale\_man

{

private:

Graph<T>& graph;

public:

int totalDistance;

sale\_man(Graph<T>& graph) : graph(graph) {}

// Функция для выделения нового узла поиска

// 'i' - текущий вершинный индекс

// 'j' - следующий вершинный индекс

//Вычисление нижней границы стоимости маршрута

int calculateLowerBound(const Graph<int>& graph, const vector<int>& tour) {

int minCost = 0;

int n = graph.GetAmountVerts();

// Учитываем минимальную стоимость связующего дерева для остальных городов

vector<bool> visited(n, false);

for (int i = 0; i < tour.size(); ++i) {

visited[tour[i]] = true;

}

// Используем алгоритм Прима, чтобы найти минимальную стоимость связующего дерева

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

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

if (!visited[i]) {

pq.push({ 0, i });

break;

}

}

while (!pq.empty()) {

int weight = pq.top().first;

int u = pq.top().second;

pq.pop();

if (visited[u]) {

continue;

}

minCost += weight;

visited[u] = true;

for (int v = 0; v < n; ++v) {

if (!visited[v] && graph.GetWeight(u, v) > 0) {

pq.push({ graph.GetWeight(u, v), v });

}

}

}

// Добавляем стоимость возврата в начальный город из последнего города тура

if (tour.size() > 1) {

minCost += graph.GetWeight(tour[tour.size() - 1], tour[0]);

}

return minCost;

}

// Рекурсивная функция ветвей и границ

void tspBranchAndBound(const Graph<int>& graph, vector<int>& tour, int currPos, int bound, vector<int>& bestTour, int& bestCost) {

// Проверка посещения всех точек

if (currPos == graph.GetAmountVerts()) {

int tourCost = 0;

for (int i = 0; i < tour.size() - 1; ++i) {

tourCost += graph.GetWeight(tour[i], tour[i + 1]);

}

tourCost += graph.GetWeight(tour[tour.size() - 1], tour[0]);

// Обновляем лучший маршрут и пройденный путь, если найдено лучшее решение

if (tourCost < bestCost) {

bestCost = tourCost;

bestTour = tour;

totalDistance = tourCost; // Обновление пройденного пути

}

return;

}

// Проход непосещенных точек

for (int i = 0; i < graph.GetAmountVerts(); ++i) {

if (!count(tour.begin(), tour.end(), i)) {

// Добавитъ точку в маршрут

tour.push\_back(i);

// Расчет нижней границы маршрута

int newBound = calculateLowerBound(graph, tour);

// Убираем ветвь, если нижняя граница больше текущего лучшего пути

if (newBound < bestCost) {

tspBranchAndBound(graph, tour, currPos + 1, newBound, bestTour, bestCost);

}

// Удаление точки

tour.pop\_back();

}

}

}

// Функция нахождения пути

vector<int> solveTSP(const Graph<int>& graph) {

int n = graph.GetAmountVerts();

vector<int> tour;

tour.push\_back(0); // Старт от 1 города

int bound = calculateLowerBound(graph, tour);

int bestCost = INT\_MAX;

vector<int> bestTour;

tspBranchAndBound(graph, tour, 1, bound, bestTour, bestCost);

// Расчет пройденного пути, после нахождения лучшего маршрута

totalDistance = 0;

for (int i = 0; i < bestTour.size() - 1; ++i) {

totalDistance += graph.GetWeight(bestTour[i], bestTour[i + 1]);

}

totalDistance += graph.GetWeight(bestTour[bestTour.size() - 1], bestTour[0]);

return bestTour;

}

};

#endif // SALE\_MAN\_H

Полный код с визуализацией:

https://github.com/RedGnom/graph

**ПРИЛОЖЕНИЕ Б**

**(UML)**

