МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: «Задача коммивояжера»

Студент гр. 3343 _____ Бондаренко Ф. А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Изучить два алгоритма для решения задачи коммивояжера: точный алгоритм с методом ветвей и границ, приближенный алгоритм модификации решения.

Задание.

В волшебной стране Алгоритмии великий маг, Гамильтон, задумал невероятное путешествие, чтобы связать все города страны заклятием процветания. Для этого ему необходимо посетить каждый город ровно один раз, создавая тропу благополучия, и вернуться обратно в столицу, используя минимум своих чародейских сил. Вашей задачей является помощь в прокладывании маршрута с помощью древнего и могущественного алгоритма ветвей и границ.

Карта дорог Алгоритмии перед Гамильтоном представляет собой полный граф, где каждый город соединён магическими порталами с каждым другим. Стоимость использования портала из города в город занимает определённое количество маны, и Гамильтон стремится минимизировать общее потребление магической энергии для закрепления проклятия.

Входные данные:

Первая строка содержит одно целое число N (N — количество городов). Города нумеруются последовательными числами от 0 до $N-1$.

Следующие N строк содержат по N чисел каждая, разделённых пробелами, формируя таким образом матрицу стоимостей MM. Каждый элемент Mi,j этой матрицы представляет собой затраты маны на перемещение из города i в город j.

Выходные данные:

Первая строка: Список из N целых чисел, разделённых пробелами, обозначающих оптимальный порядок городов в магическом маршруте

Гамильтона. В начале идёт город 0, с которого начинается маршрут, затем последующие города до тех пор, пока все они не будут посещены.

Вторая строка: Число, указывающее на суммарное количество израсходованной маны для завершения пути.

Sample Input 1:

3

-1 1 3

3 -1 1

1 2 -1

Sample Output 1:

0 1 2

3.0

Sample Input 2:

4

-1 3 4 1

1 -1 3 4

9 2 -1 4

8 9 2 -1

Sample Output 2:

0 3 2 1

6.0

Индивидуальный вариант.

Вариант 3: последовательный рост пути + использование для отсечения двух нижних оценок веса оставшегося пути: 1) полусуммы весов двух легчайших рёбер по всем кускам; 2) веса МОД. Эвристика выбора дуги — поиск в глубину с учётом веса добавляемой дуги и нижней оценки веса остатка пути. Приближённый алгоритм: АМР. Замечание к варианту 3 Начинать МВиГ со стартовой вершины.

Выполнение работы.

Описание реализованных функций для метода ветвей и границ:

- *namespace tsp* — пространство имен, отвечающее за функции для метода ветвей и границ:
  - *std::pair<std::vector<int>, double> bnbSearch(const std::vector<std::vector<double>>& weights, bool record)* — функция, отвечающая за инициализацию начальных данных для правильного поиска решения задачи коммивояжера. Внутри запускает функцию поиска *search*, что ищет оптимальный маршрут.
  - *void search(const std::vector<std::vector<double>>& weights, std::vector<int>& path, std::unordered_set<int>& remaining, double currentWeight, double& bestWeight, std::vector<int>& bestPath, bool record)* — функция поиска оптимального маршрута. Работает на основе алгоритма backtracking (поиск с возвратом) с отсечением неоптимальных ветвей. Обходит граф состояний через DFS, выбирая при этом самые "подходящие" ребра (допустимые, с минимальной общей оценкой).
  - *double getLowerBound(const std::vector<std::vector<double>> weights, const std::vector<piece_t>& pieces, bool record)* — функция для получения нижней оценки оставшегося пути. Работает на основе двух оценок: через полусумму веса минимальных дуг для каждого "куска" и через вес минимального остовного дерева, составленного из "кусков". Возвращает максимум из оценок.
  - *double getHalfSumBound(const std::vector<std::vector<double>>& weights, const std::vector<piece_t>& pieces, bool record)* — функция для получения нижней оценки через полусумму веса минимальных дуг для каждого "куска". Работает по следующей логике: каждая вершина (и составленные из вершин "куски")

должны дать вклад — вес хотя бы двух минимальных смежных с ней (с ним) ребер.

- *double getMSTBound(const std::vector<std::vector<double>>& weights, const std::vector<piece_t>& pieces, bool record)* — функция для получения нижней оценки через вес минимального остовного дерева. Работает на следующей логике: для решения задачи коммивояжера должен быть составлен Гамильтонов цикл (цикл, проходящий через все вершины по одному разу (кроме начальной)), следовательно, убрав одно любое ребро, можно получить остовное дерево. Т.к. решение должно быть оптимальным (минимальным по стоимости), то в качестве нижней оценки можно взять минимальное остовное дерево.
- *std::vector<piece_t> getPieces(const std::vector<int>& path, const std::unordered_set<int>& remaining, bool record)* — функция для получения "кусков". "Куском" является, во-первых, каждая вершина, ещё не включённая в цепочку, а во-вторых, текущая цепочка. Возвращает вектор кусков: пары вершин — начало и конец "куска".
- *std::vector<std::tuple<double, int, int>> getArcs(const std::vector<std::vector<double>>& weights, const std::vector<piece_t>& pieces, bool record)* — функция для получения "допустимых" дуг. Дуга является "допустимой", если исходит из конца куска и входит в начало другого куска. Возвращает вектор из кортежей — вес дуги, индекс вершины начала, индекс вершины конца.

Описание реализованных функций для приближенного алгоритма AMP:

- *namespace tsp* — пространство имен, отвечающее за функции для метода ветвей и границ:

- *double getCost(const std::vector<std::vector<double>>& weights, const std::vector<int>& path)* — функция для получения стоимости итогового (замкнутого) маршрута.
- *std::pair<std::vector<int>, double> amrSearch(const std::vector<std::vector<double>>& weights, bool record)* — функция для получения пути на основе приближенного алгоритма модификации решения. Включает в себя две эвристики: эвристика выбора города для перемещения, эвристика выбора новой позиции.

Описание вспомогательных функций:

- *namespace deserialize* — пространство имен для дессериализации матрицы весов:
  - *std::vector<std::vector<double>> from_json(const nlohmann::json& matrix_json)* — функция для дессериализации матрицы весов.
- *namespace serialize* — пространство имен для сериализации матрицы весов:
  - *nlohmann::json to_json(const std::vector<std::vector<double>>& matrix)* — функция для сериализации матрицы весов.
- *class JsonHandler* — класс для работы с файлами, удовлетворяющей идиоме RAII.

Описание метода ветвей и границ:

Основная задача алгоритма — последовательное построение маршрутов с использованием отсечения: на каждом шаге для кандидатов вычисляется нижняя оценка оставшейся части пути с использованием:

- Полусуммы веса минимальных дуг для каждого "куска";
- Веса минимального остовного дерева.

Данные оценки работают по следующим причинам: для получения оптимального маршрута необходимо пройти по всем вершинам графа

(образовать цикл), следовательно, вес оптимального маршрута должен быть не менее:

- Полусуммы двух минимальных по весу смежных дуг для каждой вершины. Берутся две дуги, т. к. нужно в вершину "зайти" и "выйти" из нее, полусумма, т. к. дуги будут повторятся у вершин-соседей.
- Веса минимального остовного дерева: убрав из оптимального цикла одну дугу, получается остовное дерево, следовательно, вес решения не менее веса минимального остовного дерева.

Т.к. точный алгоритм основан на алгоритме поиска с возвратом, то были использованы дополнительные оптимизации:

- При вычислении нижних оценок учитывается наличие только таких дуг, которые потенциально могут быть добавлены к решению. Назовём такие дуги допустимыми. Дуга является допустимой, если исходит из конца куска и входит в начало другого куска. Пусть, например, есть куски AB, CDE, F, G. Тогда множество допустимых дуг: BC, EA, FG, GF, FA, BF, GA, BG, FC, EF, GC, EG.
- Эвристика выбора дуги: алгоритм через поиск в глубину: каждый раз выбирается такая дуга, исходящая из текущей вершины (конца цепочки), что сумма s+L минимальна. s — вес новой дуги, L — нижняя оценка остатка пути после добавления этой дуги в цепочку.

Описание приближенного алгоритма АМР:

Основная идея алгоритма — проводить единичные улучшения по уже полученному решению (например, циклу 1-2-3-...-N). Единичные модификации — улучшения, при которых некоторый город перемещается в другое место в цепочке.

В соответствии с данной идеей можно получить псевдокод алгоритма:

```
i=0; //счётчик выполненных модификаций
m=true; //флаг обнаружения успешной модификации
Пока (m==true && i<F) { //не более F модификаций, взять F=N
   m=false;
   По множеству вариантов модификации {
      Если вес модифицированного решения получился меньше рекорда
{
         m=true;
         i++;
         Сохранить модификацию в качестве текущего решения;
         break;
      }
   }
}
```

Для поиска модификации использовались следующие эвристика: оценка потенциала города.

Оценка потенциала города.

Введем обозначения:

- *Текущий вклад города*: стоимость пути через текущий (рассматриваемый) город — сумма стоимостей входящего и исходящего ребер.
- *Прямая стоимость*: стоимость ребра, соединяющего соседей текущего города напрямую, минуя город.
- *Потенциал*: разница между *Текущим вкладом города* и *Прямой стоимостью*.

Данная эвристика оценивает, насколько выгодно удалить город из его текущей позиции. Города с наибольшим потенциалом рассматриваются первыми.

Оценка сложности алгоритма.

Точного алгоритма с методом ветвей и границ:

- По времени: в худшем случае: $O(n!)$, где n — число вершин в графе: при полном переборе (полный граф с почти одинаковыми весами рёбер).
- По памяти: $O(n^2)$, где n — число вершин в графе: хранение матрицы весов.

Приближенного алгоритма АМР:

- По времени: в худшем случае: $O(n^3)$, где n — число вершин в графе:
  - Число итераций внешнего цикла (не более n модификаций): $O(n)$.
  - Число итераций внутреннего цикла (для нахождения места вставки): $O(n^2)$.
- По памяти: $O(n^2)$, где n — число вершин в графе: хранение матрицы весов.

Тестирование.

Входные данные	Выходные данные	Комментарий
2 -1 76 65 -1	0 1 141.0	Успех для точного алгоритма (случайная матрица)
4 -1 29 25 59 29 -1 57 23 25 57 -1 22 59 23 22 -1	0 1 3 2 99.0	Успех для точного алгоритма (симметричная матрица)
-1 94 8 61 82 93 17 75 -1 91 18 69 86 23 87 86 -1 58 40 8 57 89 99 53 -1 8 8 100 27 78 64 96 -1 76 42 2 72 81 100 78 -1 39 43 90 38 58 68 87 -1	Для точного алгоритма: 0 6 2 5 1 3 4 188.0 Для приближенного: 0 6 2 4 1 3 5 201.0	Работа приближенного алгоритма (случайная матрица)
-1 29 25 59 29 -1 57 23 25 57 -1 22 59 23 22 -1	Для точного алгоритма: 0 1 3 2 99.0 Для приближенного: 0 2 3 1 99.0	Успех (загрузка матрицы из файла)

Вывод.

Изучен принцип работы метода ветвей и границ, алгоритма АМР. Реализована программа для решения задачи коммивояжера двумя способами: точно и приближенно.

ПРИЛОЖЕНИЕ А

Файл *main.cpp*:

```
#include <iostream>
#include <random>
#include <string>
#include <utility>
#include <vector>

#include "Deserializer.hpp"
#include "JsonHandler.hpp"
#include "Serializer.hpp"
#include "tsp.hpp"

#define JSON_FILE_ "json/matrix.json"

std::vector<std::vector<double>> generateRandomMatrix(int n) {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> dis(1, 100);

  std::vector<std::vector<double>> matrix(n, std::vector<double>(n));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      matrix[i][j] = i == j ? -1 : dis(gen);
    }
  }
  return matrix;
}

std::vector<std::vector<double>> generateSymmetricMatrix(int n) {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> dis(1, 100);

  std::vector<std::vector<double>> matrix(n, std::vector<double>(n));
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      int value = (i == j) ? -1 : dis(gen);
      matrix[i][j] = value;
      matrix[j][i] = value;
    }
  }
  return matrix;
}

std::vector<std::vector<double>> inputMatrix(int n) {
  std::vector<std::vector<double>> matrix(n, std::vector<double>(n));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      std::cin >> matrix[i][j];
    }
  }
  return matrix;
}

void saveMatrixToFile(const std::vector<std::vector<double>>& matrix,
                      const std::string& filename) {
```

```
  try {
    JsonHandler handler(filename);
    nlohmann::json json = serialize::to_json(matrix);
    handler.write(json);
    std::cout << "Matrix saved in: " << filename << std::endl;

  } catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
  }
}

std::vector<std::vector<double>> loadMatrixFromFile(
    const std::string& filename) {
  try {
    JsonHandler handler(filename);
    nlohmann::json json = handler.read();
    return deserialize::from_json(json);

  } catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
    throw;
  }
}

void printMatrix(const std::vector<std::vector<double>>& matrix) {
  if (matrix.empty()) return;

  size_t rows = matrix.size();
  size_t cols = matrix[0].size();

  int maxVal = 0;
  for (auto& row : matrix) {
    for (int x : row) {
      maxVal = std::max(maxVal, x);
    }
  }

  size_t numberWidth = std::to_string(maxVal).size();
  size_t columnWidth = std::to_string(cols - 1).size();
  size_t rowWidth = std::to_string(rows - 1).size();

  size_t width = std::max({numberWidth, columnWidth, rowWidth}) + 1;

  std::cout << std::setw(width) << ' ';
  for (size_t j = 0; j < cols; j++) {
    std::cout << std::setw(width) << j;
  }
  std::cout << std::endl;

  for (size_t i = 0; i < rows; i++) {
    std::cout << std::setw(width) << i;
    for (size_t j = 0; j < cols; j++) {
      std::cout << std::setw(width) << matrix[i][j];
    }
    std::cout << std::endl;
  }
}

void runAlgorithm(const std::vector<std::vector<double>>& weights,
```

```
                  bool stepMode) {
  auto [bnbPath, bnbWeight] = tsp::bnbSearch(weights, stepMode);

  std::cout << std::endl;
  std::cout << "================================" << std::endl;
  std::cout << "Branch and boundary solutuion: [ ";

  for (size_t i = 0; i < bnbPath.size(); i++) {
    std::cout << bnbPath[i] << ' ';
  }

  std::cout << ']' << std::endl;

  std::cout << std::fixed << std::setprecision(1) << "bnb weight: ["
            << bnbWeight << ']' << std::endl;

  auto [amrPath, amrWeight] = tsp::amrSearch(weights, stepMode);

  std::cout << "================================" << std::endl;
  std::cout << "AMR solution: [ ";

  for (size_t i = 0; i < amrPath.size(); i++) {
    std::cout << amrPath[i] << ' ';
  }
  std::cout << ']' << std::endl;

  std::cout << std::fixed << std::setprecision(1) << "amr weight: ["
            << amrWeight << ']' << std::endl;
   std::cout << "accuracy: [" << amrWeight / bnbWeight << ']' <<
std::endl;
}

int main() {
  std::cout << std::endl;
  std::cout << "Choose matrix generate mode:\n";
  std::cout << "1 - manual" << std::endl;
  std::cout << "2 - random" << std::endl;
  std::cout << "3 - random symmetric" << std::endl;
  std::cout << "4 - load from file" << std::endl;

  int choice;
  std::cin >> choice;

  std::vector<std::vector<double>> matrix;

  try {
    if (choice == 4) {
      matrix = loadMatrixFromFile(JSON_FILE_);
      printMatrix(matrix);
    } else {
      int n;
      std::cout << "Enter matrix size (n): ";
      std::cin >> n;

      if (n <= 0) {
        std::cerr << "Incorrect matrix size!" << std::endl;
        return 1;
      }
```

```
      switch (choice) {
        case 1:
          matrix = inputMatrix(n);
          break;
        case 2:
          matrix = generateRandomMatrix(n);
          saveMatrixToFile(matrix, JSON_FILE_);
          printMatrix(matrix);
          break;
        case 3:
          matrix = generateSymmetricMatrix(n);
          saveMatrixToFile(matrix, JSON_FILE_);
          printMatrix(matrix);
          break;
        default:
          std::cerr << "Unknown choice!\n";
          return 1;
      }
    }

    std::cout << std::endl;
    std::cout << "Choose mode:" << std::endl;
    std::cout << "1 - default" << std::endl;
    std::cout << "2 - step" << std::endl;

    int mode;
    std::cin >> mode;

    switch (mode) {
      case 1:
        runAlgorithm(matrix, false);
        break;
      case 2:
        runAlgorithm(matrix, true);
        break;
      default:
        std::cerr << "Unknown choice!\n";
        return 1;
    }

  } catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
    return 1;
  }

  return 0;
}
```

Файл *tsp.hpp*:

```
#ifndef TSP_HPP_
#define TSP_HPP_

#include <utility>
#include <vector>

namespace tsp {

using piece_t = std::pair<int, int>;
```

```
std::pair<std::vector<int>, double> bnbSearch(
    const std::vector<std::vector<double>>& matrix, bool record);

std::pair<std::vector<int>, double> amrSearch(
    const std::vector<std::vector<double>>& matrix, bool record);

}  // namespace tsp

#endif  // TSP_HPP_
```

Файл *Deserializer.hpp*:

```
#ifndef DESERIALIZER_HPP_
#define DESERIALIZER_HPP_

#include <nlohmann/json.hpp>
#include <vector>

namespace deserialize {

std::vector<std::vector<double>>    from_json(const    nlohmann::json&
matrix_json);

}  // namespace deserialize

#endif  // DESERIALIZER_HPP_
```

Файл *Serializer.hpp*:

```
#ifndef SERIALIZER_HPP_
#define SERIALIZER_HPP_

#include <nlohmann/json.hpp>
#include <vector>

namespace serialize {

nlohmann::json     to_json(const     std::vector<std::vector<double>>&
matrix);

}  // namespace serialize

#endif  // SERIALIZER_HPP_
```

Файл *JsonHandler.hpp*:

```
#ifndef JSONHANDLER_HPP_
#define JSONHANDLER_HPP_

#include <fstream>
#include <nlohmann/json.hpp>
#include <string>

class JsonHandler {
 private:
  std::fstream file;

 public:
  JsonHandler(const std::string& filename);

  void write(nlohmann::json& json);
  nlohmann::json read();
```

```
  ~JsonHandler();
};

#endif  // JSONHANDLER_HPP_
```

Файл *tsp.cpp*:

```
#include "tsp.hpp"

#include <algorithm>
#include <cfloat>
#include <climits>
#include <cstddef>
#include <iostream>
#include <unordered_set>

namespace tsp {

namespace {

double getCost(const std::vector<std::vector<double>>& weights,
               const std::vector<int>& path) {
  double summ = 0;
  for (size_t i = 0; i < path.size(); i++) {
    summ += weights[path[i]][path[(i + 1) % path.size()]];
  }
  return summ;
}

std::vector<std::tuple<double, int, int>> getArcs(
    const std::vector<std::vector<double>>& weights,
    const std::vector<piece_t>& pieces, bool record) {
  std::vector<std::tuple<double, int, int>> arcs;

  for (size_t i = 0; i < pieces.size(); i++) {
    for (size_t j = 0; j < pieces.size(); j++) {
      if (i == j) continue;

      int from = pieces[i].second;
      int to = pieces[j].first;
      double weight = weights[from][to];

      arcs.push_back({weight, from, to});
    }
  }

  std::sort(arcs.begin(), arcs.end());

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getArcs]" << std::endl;
       std::cout << "pieces count: [" << pieces.size() << "]" <<
std::endl;
    std::cout << "arcs found: [" << arcs.size() << "]" << std::endl;
    std::cout << "sorted arcs (weight, from, to): [" << std::endl;
    for (size_t i = 0; i < arcs.size(); i++) {
      auto [w, from, to] = arcs[i];
```

```
      std::cout << "  (" << w << ", " << from << ", " << to << ")" <<
std::endl;
    }
    std::cout << "]" << std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  return arcs;
}

std::vector<piece_t> getPieces(const std::vector<int>& path,
                                        const std::unordered_set<int>&
remaining,
                               bool record) {
  std::vector<piece_t> pieces;
  if (!path.empty()) {
    pieces.push_back({path.front(), path.back()});
  }

  for (int v : remaining) {
    pieces.push_back({v, v});
  }

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getPieces]" << std::endl;
    std::cout << "pieces (first, second): [" << std::endl;
    for (const auto& p : pieces) {
        std::cout << "  (" << p.first << ", " << p.second << ")" <<
std::endl;
    }
    std::cout << "]" << std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  return pieces;
}

double     getHalfSumBound(const     std::vector<std::vector<double>>&
weights,
                           const std::vector<piece_t>& pieces, bool
record) {
  if (pieces.size() <= 1) return 0;

  double summ = 0;

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getHalfSumBound] enter" << std::endl;
      std::cout << "pieces count: [" << pieces.size() << "]" <<
std::endl;
```

```
    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  for (size_t i = 0; i < pieces.size(); i++) {
    double enter = DBL_MAX;
    double exit = DBL_MAX;

    for (size_t j = 0; j < pieces.size(); j++) {
      if (i == j) continue;

      double arcIn = weights[pieces[j].second][pieces[i].first];
      double arcOut = weights[pieces[i].second][pieces[j].first];

      enter = std::min(enter, arcIn);
      exit = std::min(exit, arcOut);
    }

    summ += enter + exit;

    if (record) {
      std::cout << std::endl;
      std::cout << "--------------------------------" << std::endl;

      std::cout << "[getHalfSumBound] iteration" << std::endl;
           std::cout << "piece (" << pieces[i].first << ", " <<
pieces[i].second
                << ')' << std::endl;
      std::cout << "min_enter = [" << enter << ']' << std::endl;
      std::cout << "min_exit = [" << exit << ']' << std::endl;

      std::cout << "--------------------------------" << std::endl;
      std::cout << std::endl;
    }
  }

  double result = summ / 2;

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getHalfSumBound] exit" << std::endl;
    std::cout << "result: [" << result << "]" << std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  return result;
}

double getMSTBound(const std::vector<std::vector<double>>& weights,
                   const std::vector<piece_t>& pieces, bool record) {
  if (pieces.size() <= 1) return 0;

  size_t verticesCount = pieces.size();
  if (pieces.front().first != pieces.front().second) {
```

```
    verticesCount++;
  }

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getMSTBound] enter" << std::endl;
       std::cout << "pieces count: [" << pieces.size() << "]" <<
std::endl;
      std::cout << "vertices count: [" << verticesCount << "]" <<
std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  std::unordered_set<int> vertices;
  std::vector<std::tuple<double, int, int>> arcs =
      getArcs(weights, pieces, record);

  double summ = 0;
  for (const auto& arc : arcs) {
    if (vertices.size() >= verticesCount) break;

    auto [weight, from, to] = arc;
    if ((vertices.count(from) != vertices.count(to))) {
      vertices.insert(from);
      vertices.insert(to);
      summ += weight;

      if (record) {
        std::cout << std::endl;
        std::cout << "--------------------------------" << std::endl;

        std::cout << "[getMSTBound] iteration" << std::endl;
        std::cout << "Adding arc: (" << weight << ", " << from << ", "
<< to
                  << ")" << std::endl;
        std::cout << "summ: [" << summ << ']' << std::endl;

        std::cout << "--------------------------------" << std::endl;
        std::cout << std::endl;
      }
    }
  }

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getMSTBound] exit" << std::endl;
    std::cout << "result: [" << summ << "]" << std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  return summ;
```

```
}

double getLowerBound(const std::vector<std::vector<double>> weights,
                     const std::vector<piece_t>& pieces, bool record)
{
  double halfSumBound = getHalfSumBound(weights, pieces, record);
  double mstBound = getMSTBound(weights, pieces, record);
  double result = std::max(halfSumBound, mstBound);

  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[getLowerBound]" << std::endl;
       std::cout << "HalfSumBound: [" << halfSumBound << "]" <<
std::endl;
    std::cout << "MSTBound: [" << mstBound << "]" << std::endl;
    std::cout << "LowerBound: [" << result << "]" << std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  return result;
}

void search(const std::vector<std::vector<double>>& weights,
                    std::vector<int>& path, std::unordered_set<int>&
remaining,
            double currentWeight, double& bestWeight,
            std::vector<int>& bestPath, bool record) {
  if (record) {
    std::cout << std::endl;
    std::cout << "--------------------------------" << std::endl;

    std::cout << "[search] enter" << std::endl;
    std::cout << "path: [ ";
    for (auto it = path.begin(); it != path.end(); it++) {
      std::cout << *it << ' ';
    }
    std::cout << ']' << std::endl;
       std::cout << "currentWeight: [" << currentWeight << "]" <<
std::endl;
    std::cout << "remaining vertices: [ ";
    for (auto it = remaining.begin(); it != remaining.end(); it++) {
      std::cout << *it << ' ';
    }
    std::cout << ']' << std::endl;

    std::cout << "--------------------------------" << std::endl;
    std::cout << std::endl;
  }

  if (remaining.empty()) {
    double total = currentWeight + weights[path.back()][path.front()];

    if (record) {
      std::cout << std::endl;
      std::cout << "--------------------------------" << std::endl;
```

```
      std::cout << "[search] iteration" << std::endl;
       std::cout << "Complete path found! Total cost (with return): ["
<< total
                 << "]" << std::endl;
      std::cout << "path: [ ";
      for (auto it = path.begin(); it != path.end(); it++) {
        std::cout << *it << ' ';
      }
      std::cout << ']' << std::endl;
    }

       if (total < bestWeight || (total == bestWeight && path <
bestPath)) {
      bestPath = path;
      bestWeight = total;

      if (record) {
        std::cout << "New best path found!" << std::endl;
        std::cout << "cost: [" << total << "]" << std::endl;
        std::cout << "bestPath: [";
        for (auto it = bestPath.begin(); it != bestPath.end(); it++) {
          std::cout << *it << ' ';
        }
        std::cout << ']' << std::endl;
      }
    }

    if (record) {
      std::cout << "--------------------------------" << std::endl;
      std::cout << std::endl;
    }

    return;
  }

  std::vector<std::pair<int, int>> candidates;

  std::vector<int> rem{remaining.begin(), remaining.end()};
  for (int v : rem) {
    path.push_back(v);
    remaining.erase(v);

    std::vector<piece_t> pieces = getPieces(path, remaining, record);
    double boundary = getLowerBound(weights, pieces, record);

    remaining.insert(v);
    path.pop_back();

    candidates.push_back({weights[path.back()][v] + boundary, v});
  }

  std::sort(candidates.begin(), candidates.end());

  for (auto& c : candidates) {
    if (currentWeight + c.first > bestWeight) {
      if (record) {
        std::cout << std::endl;
        std::cout << "--------------------------------" << std::endl;
```

```
        std::cout << "[search] Pruning" << std::endl;
          std::cout << "Skipping vertex [" << c.second << "] - lower
bound: ["
                    << currentWeight + c.first << "] > best: [" <<
bestWeight
                << "]" << std::endl;

        std::cout << "--------------------------------" << std::endl;
        std::cout << std::endl;
      }

      continue;
    }

    int v = c.second;
    double edgeCost = weights[path.back()][v];

    if (record) {
      std::cout << std::endl;
      std::cout << "--------------------------------" << std::endl;

      std::cout << "[search] Branching" << std::endl;
       std::cout << "Adding vertex [" << v << "] with edge cost: [" <<
edgeCost
                << "]" << std::endl;

      std::cout << "--------------------------------" << std::endl;
      std::cout << std::endl;
    }

    path.push_back(v);
    remaining.erase(v);

        search(weights, path, remaining, currentWeight + edgeCost,
bestWeight,
           bestPath, record);

    path.pop_back();
    remaining.insert(v);
  }
}

}  // namespace

std::pair<std::vector<int>, double> bnbSearch(
    const std::vector<std::vector<double>>& weights, bool record) {
  int n = weights.size();
  double bestWeight = DBL_MAX;
  std::vector<int> bestPath;

  std::vector<int> path = {0};
  std::unordered_set<int> remaining;
  for (int i = 1; i < n; i++) remaining.insert(i);

  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;
```

```
    std::cout << "[bnbSearch]: enter" << std::endl;
    std::cout << "n (weights.size): [" << n << "]" << std::endl;
    std::cout << "remaining vertices: [ ";
    for (auto it = remaining.begin(); it != remaining.end(); it++) {
      std::cout << *it << ' ';
    }
    std::cout << ']' << std::endl;

    std::cout << std::endl;
  }

  search(weights, path, remaining, 0.0, bestWeight, bestPath, record);

  if (record) {
    std::cout << std::endl;

    std::cout << "[bnbSearch]: exit" << std::endl;
    std::cout << "bestWeight: [" << bestWeight << ']' << std::endl;
    std::cout << "bestPath: [ ";
    for (auto it = bestPath.begin(); it != bestPath.end(); it++) {
      std::cout << *it << ' ';
    }
    std::cout << ']' << std::endl;

    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }

  return {bestPath, bestWeight};
}

std::pair<std::vector<int>, double> amrSearch(
    const std::vector<std::vector<double>>& weights, bool record) {
  size_t N = weights.size();

  std::vector<int> bestPath;
  for (size_t i = 0; i < N; i++) bestPath.push_back(i);
  double bestCost = getCost(weights, bestPath);

  if (record) {
    std::cout << std::endl;
    std::cout << "================================" << std::endl;

    std::cout << "[amrSearch]: enter" << std::endl;
    std::cout << "N (weights.size): [" << N << "]" << std::endl;
    std::cout << "initialPath: [ ";
    for (auto it = bestPath.begin(); it != bestPath.end(); it++) {
      std::cout << *it << ' ';
    }
    std::cout << ']' << std::endl;
    std::cout << "initialCost: [" << bestCost << "]" << std::endl;

    std::cout << std::endl;
  }

  int count = 0;
  bool m = true;
  int F = N;
```

```
  while (m && count < F) {
    m = false;

    std::vector<std::pair<double, size_t>> potentials;
    for (size_t i = 1; i < N; i++) {
      int prev = bestPath[i - 1];
      int curr = bestPath[i];
      int next = bestPath[(i + 1) % N];

       double currentContribution = weights[prev][curr] + weights[curr]
[next];
      double directCost = weights[prev][next];
      double potential = currentContribution - directCost;

      potentials.push_back({potential, i});
    }

    std::sort(potentials.begin(), potentials.end(),
                  [](const auto& a, const auto& b) { return a.first >
b.first; });

    for (const auto& [potential, from] : potentials) {
      int cityToMove = bestPath[from];

      double bestImp = 0;
      size_t transition = from;

      for (size_t to = 1; to < N; to++) {
        if (to == from || to == (from + 1) % N) continue;

        std::vector<int> path = bestPath;
        path.erase(path.begin() + from);

        size_t insertPos = to;
        if (to > from) insertPos--;

        path.insert(path.begin() + insertPos, cityToMove);

        double newCost = getCost(weights, path);
        double improvement = bestCost - newCost;

        if (improvement > 0 && improvement > bestImp) {
          bestImp = improvement;
          transition = to;
        }
      }

      if (record) {
        std::cout << std::endl;
        std::cout << "--------------------------------" << std::endl;

           std::cout << "[amrSearch]: iteration " << count + 1 << "
checking city "
                  << cityToMove << std::endl;
             std::cout << "Current position: [" << from << "]" <<
std::endl;
         std::cout << "Best improvement found: [" << bestImp << "]" <<
std::endl;
        if (bestImp > 0) {
```

```
          std::cout << "Will move to position: ["
                            << (transition > from ? transition - 1 :
transition) << "]"
                    << std::endl;
        } else {
             std::cout << "No improvement possible for this city" <<
std::endl;
        }
        std::cout << "Current path: [ ";
        for (auto it = bestPath.begin(); it != bestPath.end(); it++) {
          std::cout << *it << ' ';
        }
        std::cout << ']' << std::endl;
             std::cout << "Current cost: [" << bestCost << "]" <<
std::endl;

        std::cout << "--------------------------------" << std::endl;
        std::cout << std::endl;
      }

      if (bestImp > 0) {
        bestPath.erase(bestPath.begin() + from);

        size_t insertPos = transition;
        if (transition > from) insertPos--;

        bestPath.insert(bestPath.begin() + insertPos, cityToMove);
        bestCost -= bestImp;

        m = true;
        count++;
        break;
      }
    }
  }

  if (record) {
    std::cout << std::endl;

    std::cout << "[amrSearch]: exit" << std::endl;
    std::cout << "iterations: [" << count << "]" << std::endl;
    std::cout << "bestCost: [" << bestCost << "]" << std::endl;
    std::cout << "bestPath: [ ";
    for (auto it = bestPath.begin(); it != bestPath.end(); it++) {
      std::cout << *it << ' ';
    }
    std::cout << ']' << std::endl;

    std::cout << "================================" << std::endl;
    std::cout << std::endl;
  }

  return {bestPath, bestCost};
}

}  // namespace tsp
```

Файл *Deserializer.cpp*:

```
#include "../include/Deserializer.hpp"
```

```
namespace deserialize {

std::vector<std::vector<double>>    from_json(const    nlohmann::json&
matrix_json) {
  std::vector<std::vector<double>> matrix =
      matrix_json["weights"].get<std::vector<std::vector<double>>>();
  return matrix;
}

}  // namespace deserialize
```

Файл *Serializer.cpp*:

```
#include "../include/Serializer.hpp"

namespace serialize {

nlohmann::json to_json(const std::vector<std::vector<double>>& matrix)
{
  nlohmann::json matrix_json = nlohmann::json::object();
  nlohmann::json weights = nlohmann::json::array();

  int n = matrix.size();
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      weights[i][j] = matrix[i][j];
    }
  }

  matrix_json["size"] = {n, n};
  matrix_json["weights"] = weights;

  return matrix_json;
}

}  // namespace serialize
```

Файл *JsonHandler.cpp*:

```
#include "JsonHandler.hpp"

JsonHandler::JsonHandler(const std::string& filename) {
  file.open(filename);
  if (!file.is_open()) {
    throw std::runtime_error("Couldn't open a json-file!");
  }
}

void    JsonHandler::write(nlohmann::json&    json)    {    file    <<
json.dump(2); }

nlohmann::json JsonHandler::read() {
  nlohmann::json json = nlohmann::json::object();
  file >> json;
  return json;
}

JsonHandler::~JsonHandler() {
  if (file.is_open()) {
    file.close();
```

```
  }
}
```

Файл *Makefile*:

```
CXX=g++
CXXFLAGS=-std=c++20 -Wall -Wextra -pedantic
EXT=cpp

SRC_DIR=src
OBJ_DIR=obj
JSON_DIR=json

SRC_FILES=$(wildcard $(SRC_DIR)/*.$(EXT))
OBJ_FILES=$(patsubst $(SRC_DIR)/%.$(EXT),$(OBJ_DIR)/%.o,$(SRC_FILES))

EXEC=$(OBJ_DIR)/exec

INCLUDES=-Iinclude

$(shell mkdir -p $(OBJ_DIR) $(JSON_DIR))

all: $(EXEC)

$(EXEC) : $(OBJ_FILES)
	$(CXX) $(CXXFLAGS) $(INCLUDES) -o $@ $^

$(OBJ_DIR)/%.o : $(SRC_DIR)/%.$(EXT)
	$(CXX) $(CXXFLAGS) $(INCLUDES) -c $< -o $@

run: $(EXEC)
	./$(EXEC)

valgrind: $(EXEC)
	@valgrind   --leak-check=full   --show-leak-kinds=all   --track-
origins=yes --error-exitcode=1 ./$(EXEC)

clean:
	rm -rf $(OBJ_DIR) $(JSON_DIR)

.PHONY: all run valgrind clean
```