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 _____ Пименов П.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Реализовать алгоритм, решающий задачу коммивояжера. Вариант 7.

Задание.

Индивидуализация: Вариант 7

Точный метод: динамическое программирование (не МВиГ), рекурсивная реализация.

Приближённый алгоритм: АЛШ-1.

Требование перед сдачей: прохождение кода в задании 3.1 на Stepik.

Замечание к варианту 7 АЛШ-1 начинать со стартовой вершины.

Условие задания:

Напишите программу, решающую задачу коммивояжера. Нужно найти кратчайший маршрут, который проходит через все заданные города ровно один раз и возвращается в исходный город. Не все города могут быть напрямую связаны друг с другом.

Входные данные:

n - количество городов ($5 \leq n \leq 15$).

Матрица расстояний между городами размером n×n, где graph[i][j] обозначает расстояние от города i до города j. Если graph[i][j]=0 (и $i \neq j$), это означает, что прямого пути между городами нет.

Выходные данные:

Минимальная стоимость маршрута, проходящего через все города и возвращающегося в начальный город.

Оптимальный путь в виде последовательности посещаемых городов, начинающейся и заканчивающейся в начальном городе.

Если такого пути не существует, вывести "no path".

Sample Input 1:

5

0 1 13 23 7

12 0 15 18 28

21 29 0 33 28

23 19 34 0 38

5 40 7 39 0

Sample Output 1:

78

0 4 2 3 1 0

Sample Input 2:

3

0 1 0

1 0 1

0 1 0

Sample Output 2:

no path

Выполнение работы.

Алгоритм реализован. Применен метод динамического программирования с мемоизацией. Использована рекурсивная реализация.

Описание полей и методов класса-решения:

- vector<vector<int>> dp – двумерный вектор, в который производится кэширование промежуточных результатов, полученных алгоритмом. dp[mask][pos] = минимальная стоимость пути, который 1) начинается в городе 0 2) прошел все города, указанные в mask 3) закончился в городе pos

- vector<vector<int>> parent – двумерный вектор, содержащий данные для восстановления пути. parent[mask][pos] = город, из которого мы пришли в pos в состоянии mask.
- int tsp(int mask, int pos) – рекурсивная функция для решения задачи точным методом. Ищет минимальный путь, который пройдет через все города, не указанные в mask, при условии, что сейчас находимся в pos. Перебирает все возможные города от 0 до n, и вызывает tsp с обновленной маской (добавленным городом) и следующим городом. Значением пути считается вес ребра + результат tsp. Кэширует результаты в dp. Также сохраняет пометки для восстановления пути в parent («куда пошли из данного города»). Условие выхода – посещение всех городов (двоичное представление маски заполнено единицами).
- vector<int> reconstructPath() – восстанавливает путь, полученный tsp, по пометкам из parent. Берет значение из parent[mask][pos] – получает следующий город, добавляет его в путь, обновляет маску (добавляет полученный город), запускает следующую итерацию цикла, пока маска не будет полной.
- pair<int, vector<int>> als(int start) – приближенный алгоритм, запускает цикл, берет вершину, в которую минимальный путь, добавляет ее в путь, переходит в нее, оттуда в следующую (если она не посещена) с минимальным путем.

Сложность алгоритма:

- Точный метод:
  - По времени: O(n^2 * 2^n) – всего n * 2^n, в каждом из которых можем проверить до n соседей (включительно).
  - По памяти: O(n * 2^n) – количество состояний (2^n масок, на каждую по n позиций).
- Приближенный метод:
  - По времени: O(n^2) – на каждом шаге берет минимальный доступный путь
  - По памяти: O(n) – хранение пути и посещенных вершин

Тестирование:

Входные данные	Выходные данные	Комментарий
5 0 1 13 23 7 12 0 15 18 28 21 29 0 33 28 23 19 34 0 38 5 40 7 39 0	78 0 4 2 3 1 0	Точный метод, успех
3 0 1 0 1 0 1 0 1 0	no path	Точный метод, успех
5 0 58 33 48 38 99 0 44 52 87 5 36 0 56 80 51 9 53 0 94 56 58 1 61 0	150 0 3 1 4 2 0	Точный метод, успех
5 0 1 13 23 7 12 0 15 18 28 21 29 0 33 28 23 19 34 0 38 5 40 7 39 0	106 0 1 2 4 3 0	Приближенный метод, неудача
3 0 1 0 1 0 1 0 1 0	no path	Приближенный метод, успех
5 0 58 33 48 38 99 0 44 52 87 5 36 0 56 80 51 9 53 0 94 56 58 1 61 0	271 0 2 1 3 4 0	Приближенный метод, неудача

Выводы.

Реализован алгоритм, решающий задачу коммивояжера.

ПРИЛОЖЕНИЕ А

Файл main.cpp:

```
#include <algorithm>
#include <iostream>
#include <limits>
#include <vector>

using namespace std;

const int INF = numeric_limits<int>::max();

class TSP {
private:
    int n;
    const vector<vector<int>>& graph;
    vector<vector<int>> dp;
    vector<vector<int>> parent;

public:
    TSP(int n, const vector<vector<int>>& graph) : n(n), graph(graph) {}

    pair<int, vector<int>> solveExact() {
        dp.assign(1 << n, vector<int>(n, -1));
        parent.assign(1 << n, vector<int>(n, -1));

        int minCost = tsp(1, 0);
        if (minCost >= INF) {
            return make_pair(-1, vector<int>{});
        }
        return make_pair(minCost, reconstructPath());
    }

    pair<int, vector<int>> solveApproximate(int start) { return als(start); }

private:
    int tsp(int mask, int pos) {
        if (mask == (1 << n) - 1) {
            cout << "[tsp] Trying to return to start from " << pos << endl;
            if (graph[pos][0] == 0) {
                cout << "[tsp] No path to start, cycle not found" << endl;
                return INF;
            } else {
                cout << "[tsp] Cycle found, returning to start" << endl;
                return graph[pos][0];
            }
        }

        if (dp[mask][pos] != -1) {
            cout << "[tsp] Using memoized value for mask=" << mask
                 << ", pos=" << pos << " is " << dp[mask][pos] << endl;
            return dp[mask][pos];
        }

        int ans = INF;
        for (int city = 0; city < n; ++city) {
            if (!(mask & (1 << city)) && graph[pos][city] > 0) {
                int newMask = mask | (1 << city);
                int nextCost = tsp(newMask, city);

                if (nextCost != INF) {
                    int newCost = graph[pos][city] + nextCost;
                    cout << "[tsp] From " << pos << " to " << city
```

```
                    << " | cost: " << graph[pos][city]
                    << ", total cost: " << newCost << endl;
                if (newCost < ans) {
                    ans = newCost;
                    parent[mask][pos] = city;
                    cout << "   Updating best next city from " << pos
                         << " with mask " << mask << " to " << city
                         << " (new cost: " << ans << ")\n";
                }
            } else {
                cout << "[tsp] No path from " << pos << " to " << city
                     << ", skipping..." << endl;
            }
        }
    }

    dp[mask][pos] = ans;
    return ans;
}

vector<int> reconstructPath() {
    vector<int> path = {0};
    int mask = 1, pos = 0;

    cout << "[reconstructPath] Reconstructing path:" << endl;
    while (mask != (1 << n) - 1) {
        int next = parent[mask][pos];
        if (next == -1) {
            cout << "   Incomplete path: parent[" << mask << "][" << pos
                 << "] = -1" << endl;
            return {};
        }
        cout << "   At mask=" << mask << ", pos=" << pos
             << " to next=" << next << endl;
        path.push_back(next);
        mask |= (1 << next);
        pos = next;
    }

    cout << "[reconstructPath] Returning to start (0)" << endl;
    path.push_back(0);
    return path;
}

pair<int, vector<int>> als(int start) {
    cout << "[als] Starting approximation from city " << start << endl;
    vector<bool> visited(n, false);
    vector<int> path;
    int cost = 0;
    int current = start;
    visited[current] = true;
    path.push_back(current);

    for (int step = 1; step < n; ++step) {
        int nextCity = -1;
        int minDist = INF;

        cout << "   Step " << step << ": from city " << current
             << ", checking neighbors..." << endl;

        for (int i = 0; i < n; ++i) {
            if (!visited[i] && graph[current][i] > 0 &&
                graph[current][i] < minDist) {
```

```
                    minDist = graph[current][i];
                    nextCity = i;
                }
            }

            if (nextCity == -1) {
                cout << "   No unvisited neighbors found. Approximate solution "
                        "failed."
                     << endl;
                return make_pair(-1, vector<int>{});
            }
            cout << "   Next city " << nextCity << " with cost " << minDist
                 << endl;
            cost += minDist;
            visited[nextCity] = true;
            current = nextCity;
            path.push_back(current);
        }

        if (graph[current][start] > 0) {
            cost += graph[current][start];
            path.push_back(start);
            cout << "   Returning to start city " << start << " with cost "
                 << graph[current][start] << endl;
            cout << "   Final cost: " << cost << endl;
            return make_pair(cost, path);
        } else {
            cout << "   Cannot return to start city. Edge from " << current
                 << " to " << start << " is missing." << endl;
            return make_pair(-1, vector<int>{});
        }
    }
};

int main() {
    int n;
    cin >> n;

    vector<vector<int>> graph(n, vector<int>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            cin >> graph[i][j];

    TSP tsp(n, graph);

    pair<int, vector<int>> exactResult = tsp.solveExact();
    cout << "Exact solution:" << endl;
    if (exactResult.first == -1) {
        cout << "no path" << endl;
    } else {
        cout << exactResult.first << endl;
        for (int city : exactResult.second)
            cout << city << " ";
        cout << endl;
    }

    pair<int, vector<int>> approximateResult = tsp.solveApproximate(0);
    cout << "Approximate solution:" << endl;
    if (approximateResult.first == -1) {
        cout << "no path" << endl;
    } else {
        cout << approximateResult.first << endl;
        for (int city : approximateResult.second)
```

```
            cout << city << " ";
        cout << endl;
    }

    return 0;
}
```