МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЁТ ПО ПРАКТИЧЕСКОМУ ЗАДАНИЮ №6

по дисциплине «Структуры и алгоритмы обработки данных»

Тема: «Алгоритмические стратегии. Перебор и методы его сокращения.»

Отчет представлен к рассмотрению:
Студент группы ИНБО-01-20 «20» ноября 2021 г. _____ Салов В.Д.
(подпись)

Преподаватель «20» ноября 2021 г. _____ Сорокин А.В.
(подпись)

Москва, 2021 г.

СОДЕРЖАНИЕ

Цель работы

Разработка и программная реализация задач с применением метода сокращения числа переборов.

Постановка задачи

Разработать алгоритм решения задачи варианта с применением метода, указанного в варианте; реализовать программу; оценить количество переборов при решении задачи стратегией «в лоб» – методом «грубой силы»; привести анализ снижения числа переборов при применении метода, указанного в варианте.

Вариант 6.

Задача	Метод
Дано прямоугольное поле размером m * n клеток. Можно совершать шаги длиной в одну клетку вправо, вниз или по диагонали вправо-вниз. В каждой клетке записано некоторое натуральное число. Необходимо попасть из верхней левой клетки в правую нижнюю. Вес маршрута вычисляется как сумма чисел со всех посещенных клеток. Необходимо найти маршрут с минимальным весом.	Динамическое программирование

Дано:

Размер поля; натуральные числа, записанные в клетки поля.

Результат.

Кратчайший маршрут из верхней левой клетки поля до нижней правой.

Подход к решению.

1) Было решено представить поле движения в виде ориентированного графа. Для этого был разработан класс графа, реализующий следующие методы: создание графа посредством применения операций вставки ребра в граф, нахождение величины кратчайшего пути и восстановление

кратчайшего пути от заданной вершины к другой заданной вершине с помощью метода «грубой силы», а также с помощью метода «Дейкстры» как одного из методов динамического программирования для снижения числа переборов.

2) Разработан консольный пользовательский интерфейс для тестирования работоспособности программы.
3) Разработаны методы обработки графа:

1. Метод вставки ребра в граф – добавление ребра с заданным весом между двумя заданными вершинами в список смежных вершин графа и в матрицу смежности графа;
2. Метод перебора вершин («грубой силы») – поиск величин кратчайших путей от каждой вершины графа к другим вершинам графа путём перебора и возврат матрицы кратчайших путей из найденных величин;
3. Метод «Дейкстры» – поиск величин кратчайших путей до каждой вершины для заданной вершины и возврат массива кратчайших путей из найденных величин;
4. Метод величины кратчайшего пути для алгоритма перебора вершин ("грубой силы") – вызов метода «грубой силы» и возврат величины кратчайшего пути из одной заданной вершины в другую заданную вершину по матрице кратчайших путей;
5. Метод величины кратчайшего пути для алгоритма Дейкстры – вызов метода «Дейкстры» и возврат величины кратчайшего пути между двумя заданными вершинами графа по массиву кратчайших путей.
6. Метод восстановления кратчайшего пути для алгоритма перебора вершин ("грубой силы") – вызов метода «грубой силы» и возврат списка из массивов индексов (i, j) элементов матрицы (клеток поля), последовательно входящих в кратчайший путь.
7. Метод восстановления кратчайшего пути для алгоритма Дейкстры – вызов метода «Дейкстры» и возврат списка из массивов индексов (i, j)

элементов матрицы (клеток поля), последовательно входящих в кратчайший путь.

4) Разработан метод приложения для тестирования:

1. Метод для тестирования нахождения кратчайшего пути – организация нахождения кратчайшего пути от одной заданной вершины к другой заданной вершине с помощью алгоритма перебора вершин и с помощью алгоритма Дейкстры, а также вывода результатов работы алгоритмов в консоль.

Алгоритмы операций на псевдокоде.

Метод вставки ребра в граф:

```
процедура connect(вершина1, вершина2, вес):
  список_смежных_вершин[вершина1].добавить_элемент_в_конец
  ([вершина2, вес]);
  матрица_смежности[вершина1][вершина2] := вес.
```

Метод перебора вершин («грубой силы»):

```
функция brute_force():
  количество_вершин := длина(матрица_смежности);
  min_dist := матрица_смежности;
  количество_сравнений := количество_сравнений + 1;
  для k от 0 до количество_вершин – 1 выполнять:
    количество_сравнений := количество_сравнений + 1;
    для i от 0 до количество_вершин – 1 выполнять:
      количество_сравнений := количество_сравнений + 1;
      для j от 0 до количество_вершин – 1 выполнять:
        количество_сравнений := количество_сравнений + 1;
        если min_dist[i][j] > min_dist[i][k] + min_dist[k][j]:
          min_dist[i][j] := min_dist[i][k] + min_dist[k][j];
        количество_сравнений := количество_сравнений + 1;
      количество_сравнений := количество_сравнений + 1;
    количество_сравнений := количество_сравнений + 1;
  возврат min_dist.
```

Метод величины кратчайшего пути для алгоритма перебора вершин:

```
функция shortest_path_brute_force(вершина1, вершина2):
  возврат brute_force()[вершина1][вершина2].
```

Метод восстановления кратчайшего пути для алгоритма перебора вершин ("грубой силы"):

```
функция path_restoring_brute_force(вершина1, вершина2):
    visited := список();
    количество_сравнений := количество_сравнений + 1;
    для i от 0 до длина(матрица_смежности) – 1 выполнять:
        количество_сравнений := количество_сравнений + 1;
        visited.добавить_элемент_в_конец((None, None));
    visited[0] := (вершина2 // кол-во_столбцов + 1, вершина2 % кол-во_столбцов + 1);
    пред_индекс := 1;
    вес_кр_пути := shortest_path_brute_force(вершина1, вершина2);
    количество_сравнений := количество_сравнений + 1;
    пока вершина2 ≠ вершина1, выполнять:
        количество_сравнений := количество_сравнений + 1;
        для i от 0 до длина(матрица_смежности) – 1 выполнять:
            количество_сравнений := количество_сравнений + 2;
            если матрица_смежности[i][вершина2] < ∞ и матрица_смежности[i][вершина2] ≠ 0:
                temp := вес_кр_пути – матрица_смежности[i][вершина2];
                количество_сравнений := количество_сравнений + 1;
                если temp = self.shortest_path_brute_force(вершина1, i):
                    вес_кр_пути := temp;
                    вершина2 := i;
                    visited[пред_индекс] := (i // кол-во_столбцов + 1, i % кол-во_столбцов + 1);
                    пред_индекс := пред_индекс + 1;
            количество_сравнений := количество_сравнений + 1;
        количество_сравнений := количество_сравнений + 1;
    количество_сравнений := количество_сравнений + длина(visited);
    пока (None, None) в visited, выполнять:
        количество_сравнений := количество_сравнений + visited.индекс_элемента((None, None));
        visited.удалить_элемент((None, None));
    возврат visited.развернуть_список().
```

Метод «Дейкстры»:

```
функция dijkstra(вершина1):
    nodes_to_visit := список();
    nodes_to_visit.добавить_элемент_в_конец((0, вершина1));
    visited := множество();
    количество_сравнений := количество_сравнений + 1;
    min_dist := {i: ∞ для каждого i от 0 до длина(список_смежных_вершин) – 1};
```

```
    количество_сравнений := количество_сравнений +
    длина(список_смежных_вершин);
    min_dist[вершина1] := 0;
    количество_сравнений := количество_сравнений + 1;
    пока длина(nodes_to_visit) > 0, выполнять:
        количество_сравнений := количество_сравнений +
        длина(nodes_to_visit);
        вес, текущая_вершина := минимальный_элемент(nodes_to_visit);
        nodes_to_visit.удалить_элемент((вес, текущая_вершина));
        количество_сравнений := количество_сравнений + 1;
        если текущая_вершина в visited:
            количество_сравнений := количество_сравнений +
            список(visited).индекс(текущая_вершина);
            принудительный_запуск_следующего_прохода_цикла;
        количество_сравнений := количество_сравнений +
        длина(список(visited));
        visited.добавить_элемент(текущая_вершина);
        количество_сравнений := количество_сравнений + 1;
        для след_вес, след_вершина в
        список_смежных_вершин[текущая_вершина] выполнять:
            если след_вершина в visited:
                количество_сравнений := количество_сравнений +
                список(visited).индекс(след_вершина);
            количество_сравнений := количество_сравнений + 1;
            если вес + след_вес < min_dist[след_вершина] и след_вершина не в
            visited:
                количество_сравнений := количество_сравнений +
                длина(список(visited));
                min_dist[след_вершина] := вес + след_вес;
                nodes_to_visit.добавить_элемент_в_конец((вес + след_вес,
                след_вершина));
            количество_сравнений := количество_сравнений + 1;
        количество_сравнений := количество_сравнений + 1;
    возврат min_dist.
```

Метод величины кратчайшего пути для алгоритма Дейкстры:

```
функция shortest_path_dijkstra(вершина1, вершина2):
    возврат dijkstra(вершина1)[вершина2].
```

Метод восстановления кратчайшего пути для алгоритма перебора вершин ("грубой силы"):

```
функция path_restoring_dijkstra (вершина1, вершина2):
    visited := список();
    количество_сравнений := количество_сравнений + 1;
    для i от 0 до длина(матрица_смежности) – 1 выполнять:
        количество_сравнений := количество_сравнений + 1;
        visited.добавить_элемент_в_конец((None, None));
    visited[0] := (вершина2 // кол-во_столбцов + 1, вершина2 % кол-во_столбцов + 1);
    пред_индекс := 1;
    вес_кр_пути := shortest_path_ dijkstra(вершина1, вершина2);
    количество_сравнений := количество_сравнений + 1;
    пока вершина2 ≠ вершина1, выполнять:
        количество_сравнений := количество_сравнений + 1;
        для i от 0 до длина(матрица_смежности) – 1 выполнять:
            количество_сравнений := количество_сравнений + 2;
            если матрица_смежности[i][вершина2] < ∞ и матрица_смежности[i][вершина2] ≠ 0:
                temp := вес_кр_пути – матрица_смежности[i][вершина2];
                количество_сравнений := количество_сравнений + 1;
                если temp = self.shortest_path_ dijkstra(вершина1, i):
                    вес_кр_пути := temp;
                    вершина2 := i;
                    visited[пред_индекс] := (i // кол-во_столбцов + 1, i % кол-во_столбцов + 1);
                    пред_индекс := пред_индекс + 1;
            количество_сравнений := количество_сравнений + 1;
        количество_сравнений := количество_сравнений + 1;
    количество_сравнений := количество_сравнений + длина(visited);
    пока (None, None) в visited, выполнять:
        количество_сравнений := количество_сравнений + visited.индекс_элемента((None, None));
        visited.удалить_элемент((None, None));
    возврат visited.развернуть_список().
```

Код программы.

Класс графа:

```
# Класс, реализующий граф и некоторые операции с ним.
class Graph:
    # Конструктор класса.
    def __init__(self, m, n):
        self.compares_bf = 0  # Количество сравнений при работе метода "грубой силы".
        self.compares_dp = 0  # Количество сравнений при работе метода динамического программирования.
        self.vert_vertex = m  # Количество вершин графа по вертикали.
        self.hor_vertex = n  # Количество вершин графа по горизонтали.
        self.vertexes = self.hor_vertex * self.vert_vertex  # Общее количество вершин графа.
        # Установка списка смежных вершин.
        self.adj_list = list()
        for i in range(self.vertexes):
            self.adj_list.append([])
        # Установка матрицы смежности.
        self.adj_mat = [[float('inf')] * self.vertexes for _ in range(self.vertexes)]
        for i in range(len(self.adj_mat)):
            for j in range(len(self.adj_mat)):
                if i == j:
                    self.adj_mat[i][j] = 0

    # Процедура вставки взвешенного ребра в граф.
    def connect(self, node1, node2, weight):
        self.adj_list[node1].append([node2, weight])  # Вставка ребра в список смежных вершин.
        self.adj_mat[node1][node2] = weight  # Вставка ребра в матрицу смежности для алгоритма перебора вершин.

    # Функция копирования матрицы смежности для дальнейшего перезаполнения под матрицу кратчайших путей.
    def adj_mat_copy(self):
        adj_matrix = [[None] * self.vertexes for _ in range(self.vertexes)]
        for i in range(len(self.adj_mat)):
            for j in range(len(self.adj_mat)):
                adj_matrix[i][j] = self.adj_mat[i][j]
        return adj_matrix

    # Функция алгоритма перебора вершин (метода "грубой силы").
    def brute_force(self):
        v = len(self.adj_mat)  # Количество узлов графа как длина матрицы смежности.
        min_dist = self.adj_mat_copy()  # Матрица кратчайших путей, изначально являющаяся копией матрицы смежности.
        self.compares_bf += 1  # Подсчёт количества сравнений: первое сравнение перед входом в цикл for.
        for k in range(v):
            self.compares_bf += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
            for i in range(v):
                self.compares_bf += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
                for j in range(v):
                    self.compares_bf += 1  # Увеличение количества сравнений на 1: перед условием if.
                    if min_dist[i][j] > min_dist[i][k] + min_dist[k][j]:
                        min_dist[i][j] = min_dist[i][k] + min_dist[k][j]  # Величина текущего кратчайшего пути из i в j.
                    self.compares_bf += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
                self.compares_bf += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            self.compares_bf += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
        return min_dist  # Возврат матрицы кратчайших путей.

    # Функция величины кратчайшего пути, методом перебора вершин ("грубой силы").
    def shortest_path_brute_force(self, node1, node2):  # Для заданного узла node1 и другого заданного узла node2:
        return self.brute_force()[node1][node2]  # Возврат величины кратчайшего пути между node1 и node2.

    # Функция восстановления кратчайшего пути между двумя заданными вершинами, методом перебора вершин ("грубой силы").
    def path_restoring_brute_force(self, node1, node2):
        visited = list()
        self.compares_bf += 1  # Подсчёт количества сравнений: первое сравнение перед входом в цикл for.
        for i in range(len(self.adj_mat)):
            self.compares_bf += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            visited.append((None, None))
        # Начальный элемент - конечная вершина. Добавление в список номера элемента матрицы.
        visited[0] = (node2 // self.hor_vertex + 1, node2 % self.hor_vertex + 1)
        pre = 1  # Индекс предыдущей вершины.
        weight = self.shortest_path_brute_force(node1, node2)  # Вес пути до конечной вершины.
        self.compares_bf += 1  # Увеличение количества сравнений на 1: перед входом в цикл while.
        while node2 != node1:  # Пока не дошли до начальной вершины:
            self.compares_bf += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
            for i in range(len(self.adj_mat)):  # Проход по всем вершинам.
                self.compares_bf += 2  # Увеличение количества сравнений на 2: перед условием if.
                if self.adj_mat[i][node2] < float('inf') and self.adj_mat[i][node2] != 0:  # При наличии связи:
                    temp = weight - self.adj_mat[i][node2]  # Определение веса пути из предыдущей вершины.
                    self.compares_bf += 1  # Увеличение количества сравнений на 1: перед условием if.
                    # Если вес совпал с рассчитанным, то из этой вершины был переход.
                    if temp == self.shortest_path_brute_force(node1, i):
                        weight = temp
                        node2 = i
                        visited[pre] = (i // self.hor_vertex + 1, i % self.hor_vertex + 1)
                        pre += 1
                self.compares_bf += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            self.compares_bf += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла while.
        self.compares_bf += len(visited)  # Увеличение кол-ва сравнений на кол-во пройденных элементов списка visited.
        while (None, None) in visited:
            # Увеличение кол-ва сравнений на кол-во пройденных элементов списка visited.
            self.compares_bf += visited.index((None, None))
            visited.remove((None, None))
        return visited[::-1]
```

Рисунок 1 – Класс графа.

```
    # Алгоритм Дейкстры.
    def dijkstra(self, node):
        nodes_to_visit = list()  # Инициализация списка вершин для посещения.
        nodes_to_visit.append((0, node))  # Добавление стартовой вершины в список как первой вершины для посещения.
        visited = set()  # Множество для хранения посещённых вершин.
        self.compares_dp += 1  # Подсчёт количества сравнений: первое сравнение перед входом в цикл for.
        min_dist = {i: float('inf') for i in range(len(self.adj_list))}  # Заполнение расстояний до вершин.
        self.compares_dp += len(self.adj_list)  # Увеличение кол-ва сравнений на кол-во итераций предыдущего цикла for.
        min_dist[node] = 0  # Заполнение расстояния до стартовой вершины.
        self.compares_dp += 1  # Увеличение количества сравнений на 1: перед входом в цикл while.
        while len(nodes_to_visit):  # Пока nodes_to_visit не пустой:
            self.compares_dp += len(nodes_to_visit)  # Увеличение кол-ва сравнений на кол-во итераций для функции min().
            weight, current_node = min(nodes_to_visit)  # Выбор ближней вершины.
            nodes_to_visit.remove((weight, current_node))  # Удаление этой вершины из списка вершин для посещения.
            if current_node in visited:  # Если выбранная вершина уже посещена:
                # Увеличение количества сравнений на кол-во пройденных элементов списка visited.
                self.compares_dp += list(visited).index(current_node)
                continue  # Запуск следующего прохода цикла без выполнения оставшегося тела цикла.
            # Увеличение кол-ва сравнений на на кол-во пройденных элементов списка visited.
            self.compares_dp += len(list(visited))
            visited.add(current_node)  # Добавление выбранной вершины в список посещённых.
            # next_weight - вес из текущей вершины, next_node - прикреплённая вершина, в которую необходимо попасть.
            self.compares_dp += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
            for next_node, next_weight in self.adj_list[current_node]:  # Проход по всем соединённым вершинам.
                # Проверка на оптимальность пути.
                if next_node in visited:
                    # Увеличение количества сравнений на кол-во пройденных элементов списка visited в следующем "if".
                    self.compares_dp += list(visited).index(next_node)
                self.compares_dp += 1  # Увеличение количества сравнений на 1: перед условием if.
                if weight + next_weight < min_dist[next_node] and next_node not in visited:
                    # Увеличение количества сравнений на кол-во пройденных элементов списка visited.
                    self.compares_dp += len(list(visited))
                    min_dist[next_node] = weight + next_weight  # Обновление расстояния.
                    nodes_to_visit.append((weight + next_weight, next_node))  # Добавление в список для посещения.
                self.compares_dp += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            self.compares_dp += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла while.
        return min_dist  # Возврат множества из словарей {номер_узла: кратчайший путь до него от заданного узла}.

    # Функция величины кратчайшего пути, методом "Дейкстры".
    def shortest_path_dijkstra(self, node1, node2):  # Для заданного узла node1 и другого заданного узла node2:
        return self.dijkstra(node1)[node2]  # Возврат величины кратчайшего пути между node1 и node2.

    # Функция восстановления кратчайшего пути между двумя заданными вершинами, методом "Дейкстры".
    def path_restoring_dijkstra(self, node1, node2):
        visited = list()
        self.compares_dp += 1  # Подсчёт количества сравнений: первое сравнение перед входом в цикл for.
        for i in range(len(self.adj_mat)):
            self.compares_dp += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            visited.append((None, None))
        # Начальный элемент - конечная вершина. Добавление в список номера элемента матрицы.
        visited[0] = (node2 // self.hor_vertex + 1, node2 % self.hor_vertex + 1)
        pre = 1  # Индекс предыдущей вершины.
        weight = self.shortest_path_dijkstra(node1, node2)  # Вес пути до конечной вершины.
        self.compares_dp += 1  # Увеличение количества сравнений на 1: перед входом в цикл while.
        while node2 != node1:  # Пока не дошли до начальной вершины:
            self.compares_dp += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
            for i in range(len(self.adj_mat)):  # Проход по всем вершинам.
                self.compares_dp += 2  # Увеличение количества сравнений на 2: перед условием if.
                if self.adj_mat[i][node2] < float('inf') and self.adj_mat[i][node2] != 0:  # При наличии связи:
                    temp = weight - self.adj_mat[i][node2]  # Определение веса пути из предыдущей вершины.
                    self.compares_dp += 1  # Увеличение количества сравнений на 1: перед условием if.
                    # Если вес совпал с рассчитанным, то из этой вершины был переход.
                    if temp == self.shortest_path_dijkstra(node1, i):
                        weight = temp
                        node2 = i
                        visited[pre] = (i // self.hor_vertex + 1, i % self.hor_vertex + 1)
                        pre += 1
                self.compares_dp += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            self.compares_dp += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла while.
        self.compares_dp += len(visited)  # Увеличение кол-ва сравнений на кол-во пройденных элементов списка visited.
        while (None, None) in visited:
            # Увеличение кол-ва сравнений на кол-во пройденных элементов списка visited.
            self.compares_dp += visited.index((None, None))
            visited.remove((None, None))
        return visited[::-1]
```

Рисунок 2 – Класс графа (продолжение).

```
    # Приложение для нахождения величины кратчайшего пути для черепашки в поле, размером m * n.
    def app_shortest_path(self, matrix_elements):  # Для заданного списка элементов матрицы:
        for node in range(self.vertexes):  # Для каждой вершины:
            if node < self.vertexes - self.hor_vertex:  # Если вершина не на нижней границе поля:
                # Соединение текущей вершины с соседней снизу.
                self.connect(node, node + self.hor_vertex, matrix_elements[node + self.hor_vertex])
                if (node + 1) % self.hor_vertex != 0 or node == 0:  # Если вершина не на правой границе поля:
                    # Соединение текущей вершины с вершиной снизу справа по диагонали.
                    self.connect(node, node + self.hor_vertex + 1, matrix_elements[node + self.hor_vertex + 1])
                    # Соединение текущей вершины с соседней справа.
                    self.connect(node, node + 1, matrix_elements[node + 1])
            elif node < self.vertexes - 1:  # Иначе, если вершина не последняя (но на нижней границе поля):
                # Соединение текущей вершины только с соседней справа.
                self.connect(node, node + 1, matrix_elements[node + 1])
        '''# Вывод графа в виде списка смежных вершин.
        for row in self.adj_list:  # Для каждой строки в списке смежных вершин:
            print(row)  # Вывод текущей строки списка смежных вершин.
        # Вывод графа в виде матрицы смежности.
        for row in self.adj_mat:  # Для каждой строки в матрице смежности:
            print(row)  # Вывод текущей строки списка матрицы смежности.'''
        from time import perf_counter  # Импорт perf_counter из библиотеки time.
        print("_____")
        print('Метод "грубой силы" для поиска кратчайшего пути.')
        time_a = perf_counter()  # Время перед работой алгоритма.
        print(f'Величина кратчайшего пути: {self.shortest_path_brute_force(0, self.vertexes - 1)+ matrix_elements[0]}.')
        print(f'Кратчайший маршрут: {self.path_restoring_brute_force(0, self.vertexes - 1)}.')
        time_b = perf_counter()  # Время после работы алгоритма.
        print(f"Затраченное время на работу алгоритма: {time_b - time_a:0.7f} с.")
        print(f'Количество выполненных сравнений: {self.compares_bf}.')
        print("_____")
        print('Метод "Дейкстры" как один из методов динамического программирования для поиска кратчайшего пути.')
        time_a = perf_counter()  # Время перед работой алгоритма.
        print(f'Величина кратчайшего пути: {self.shortest_path_dijkstra(0, self.vertexes - 1) + matrix_elements[0]}.')
        print(f'Кратчайший маршрут: {self.path_restoring_dijkstra(0, self.vertexes - 1)}.')
        time_b = perf_counter()  # Время после работы алгоритма.
        print(f"Затраченное время на работу алгоритма: {time_b - time_a:0.7f} с.")
        print(f'Количество выполненных сравнений: {self.compares_dp}.')
        print("_____")
```

Рисунок 3 – Класс графа (продолжение).

Основная функция для тестирования:

```
# Главная функция.
if __name__ == '__main__':
    matrix_elements = list()  # Список элементов матрицы (весов клеток поля).
    size = list(map(int, input('Укажите размер матрицы m * n в виде "m n": ').split()))  # Задание размера матрицы.
    m = size[0]  # Количество строк матрицы.
    n = size[1]  # Количество столбцов матрицы.
    print("Введите число для выбора типа ввода данных:\n"
          "0 - установка веса для каждой клетки вручную;\n"
          "1 - автоматическая установка случайного веса для каждой клетки.")
    input_type = bool(int(input("Ваше число: ")))
    if input_type is True:  # При вводе '1' данные будут введены автоматически (сгенерированы случайным образом):
        import random  # Импорт библиотеки random.
        # Заполнение матрицы случайными числами от 1 до 10.
        for element in range(m * n):
            matrix_elements.append(random.randint(1, 10))
    else:  # При вводе '0' данные будут введены вручную:
        print("Построчно вводите элементы матрицы слева направо.")
        for i in range(m):  # Для каждой строки матрицы:
            matrix_row_elements = list(map(int, input(f"Введите элементы {i + 1}-й строки матрицы: ").split()))
            # Добавление введённых элементов в матрицу.
            for element in range(n):
                matrix_elements.append(matrix_row_elements[element])
    print("Данные получены. Определяется кратчайший путь от первого элемента матрицы к последнему.")
    w_graph = Graph(m, n)  # Создание пустого графа заданного размера.
    w_graph.app_shortest_path(matrix_elements)  # Нахождение величины кратчайшего пути от первого элемента к последнему.
    print()
    print("Тестирование завершено.")
```

Рисунок 4 – Основная функция.

Тестирование программы.

Тестирование программы на поле с заданными числами.

Для тестирования было выбрано поле размером 3 * 4 (см. Рисунок 5). Стрелками выделен кратчайший путь из верхней левой клетки поля в правую нижнюю; несложно найти его величину: 3 + 1 + 2 + 4 + 5 = 15.

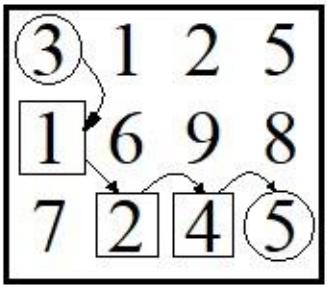

Рисунок 5 – Поле для тестирования.

Вообще говоря, для данного задания можно считать любое заданное поле размером m * n матрицей из m строк и n столбцов, а числа в клетках поля – элементами матрицы. Далее будем считать эти понятия равнозначными.

Было запущено тестирование программы. Сначала был указан размер матрицы, затем построчно вводились элементы каждой строки матрицы (см. Рисунок 6) после выбора формата ввода вручную.

```
Укажите размер матрицы m * n в виде "m n": 3 4
Введите число для выбора типа ввода данных:
0 - установка веса для каждой клетки вручную;
1 - автоматическая установка случайного веса для каждой клетки.
Ваше число: 0
Построчно вводите элементы матрицы слева направо.
Введите элементы 1-й строки матрицы: 3 1 2 5
Введите элементы 2-й строки матрицы: 1 6 9 8
Введите элементы 3-й строки матрицы: 7 2 4 5
Данные получены. Определяется кратчайший путь от первого элемента матрицы к последнему.
```

Рисунок 6 – Ввод данных.

После завершения ввода данных программа осуществляет поиск величины кратчайшего пути от первого элемента матрицы к последнему, а также восстановление кратчайшего пути двумя способами: методом «грубой силы» (алгоритмом перебора вершин) и методом динамического программирования (алгоритмом Дейкстры) (см. Рисунок 7). Чтобы сделать

анализ результатов более удобным, в процессе выполнения алгоритмов также подсчитывалось количество сравнений и замерялось затраченное время.

```
Метод "грубой силы" для поиска кратчайшего пути.
Величина кратчайшего пути: 15.
Кратчайший маршрут: [(1, 1), (2, 1), (3, 2), (3, 3), (3, 4)].
Затраченное время на работу алгоритма: 0.0093615 с.
Количество выполненных сравнений: 37911.
--------------------------------------------------------------------------------
Метод "Дейкстры" как один из методов динамического программирования для поиска кратчайшего пути.
Величина кратчайшего пути: 15.
Кратчайший маршрут: [(1, 1), (2, 1), (3, 2), (3, 3), (3, 4)].
Затраченное время на работу алгоритма: 0.0009599 с.
Количество выполненных сравнений: 2711.
--------------------------------------------------------------------------------

Тестирование завершено.
```

Рисунок 7 – Полученные результаты.

В результате для данного примера и тот, и другой методы выдали правильный ответ к поставленной задаче: величина кратчайшего пути, действительно, равна 15; сам кратчайший путь от первого элемента матрицы (с индексами $i = 1$, $j = 1$) до последнего элемента матрицы (с индексами $i = m = 3$, $j = n = 4$) также выведен верно: $a_{11} \rightarrow a_{21} \rightarrow a_{32} \rightarrow a_{33} \rightarrow a_{34}$.

Из результатов тестирования видно, что для данного случая алгоритм Дейкстры способен уменьшить затраченное время в ~10 раз и количество сравнений в ~14 раз, в сравнении с алгоритмом перебора (методом «грубой силы»). Достигается это благодаря использованию принципа динамического программирования: разбиения сложной задачи на более простые подзадачи – следуя методу «Дейкстры», нет необходимости осуществлять проход по каждой вершине более одного раза; ещё данный алгоритм постоянно осуществляет движение к наиболее «выгодной» вершине, на каждом шаге выбирая кратчайший путь из возможных. Таким образом сложность алгоритма значительно снижается, как и время его выполнения с количеством совершаемых операций сравнения.

Тестирование программы на поле со сгенерированными случайным образом числами.

Теперь для тестирования была выбрана матрица размером 12 * 10. Для упрощения ввода элементы матрицы были сгенерированы случайным образом со значениями в пределах от 1 до 10 включительно. После запуска программы достаточно было задать размер матрицы и ввести «1» для выбора необходимого формата ввода (см. Рисунок 8).

```
Укажите размер матрицы m * n в виде "m n": 12 10
Введите число для выбора типа ввода данных:
0 – установка веса для каждой клетки вручную;
1 – автоматическая установка случайного веса для каждой клетки.
Ваше число: 1
Данные получены. Определяется кратчайший путь от первого элемента матрицы к последнему.
```

Рисунок 8 – Ввод данных.

Далее программа выполнила для созданной матрицы поиск величины кратчайшего пути от первого элемента к последнему, а также восстановление кратчайшего пути двумя способами: методом «грубой силы» (алгоритмом перебора вершин) и методом динамического программирования (алгоритмом Дейкстры) (см. Рисунок 9).

```
_____
Метод "грубой силы" для поиска кратчайшего пути.
Величина кратчайшего пути: 44.
Кратчайший маршрут: [(1, 1), (2, 2), (3, 2), (4, 3), (5, 4), (6, 5), (7, 6), (8, 6), (9, 6), (10, 6), (11, 7), (11, 8), (12, 9), (12, 10)].
Затраченное время на работу алгоритма: 18.4543437 с.
Количество выполненных сравнений: 80162396.
_____
Метод "Дейкстры" как один из методов динамического программирования для поиска кратчайшего пути.
Величина кратчайшего пути: 44.
Кратчайший маршрут: [(1, 1), (2, 2), (3, 2), (4, 3), (5, 4), (6, 5), (7, 6), (8, 6), (9, 6), (10, 6), (11, 7), (11, 8), (12, 9), (12, 10)].
Затраченное время на работу алгоритма: 0.0257396 с.
Количество выполненных сравнений: 476251.
_____

Тестирование завершено.
```

Рисунок 9 – Полученные результаты.

По результатам видно, что и тот, и другой методы так же вывели одинаковый ответ. Однако теперь метод «грубой силы» уступает выбранному методу динамического программирования (алгоритму Дейкстры) по скорости в ~717 раз, а по количеству сравнений – в ~168 раз, что делает его неэффективным как по затратам времени, так и по использованию памяти. С применением методов динамического программирования (в данном случае – алгоритма Дейкстры) поставленная задача была решена за ~0.3 с (что

достаточно мало по сравнению с ~18 с), количество произведённых сравнений – ~$0.476*10^6$ (что тоже достаточно мало в сравнении с ~$8*10^6$).

Таким образом, методы динамического программирования значительно уменьшают сложность алгоритма и ускоряют процесс решения поставленной задачи, разбивая её на более простые подзадачи – благодаря снижению количества переборов уменьшается затрачиваемое время на выполнения алгоритма и расход памяти.

Вывод

В ходе работы был разработан алгоритм решения задачи поиска кратчайшего пути с применением метода динамического программирования; реализована программа для решения задания варианта; приведена оценка количества переборов при решении задачи стратегией «в лоб» – методом «грубой силы»; произведён анализ снижения числа переборов при применении одного из методов динамического программирования.

Список информационных источников

1. Лекции по дисциплине «Структуры и алгоритмы обработки данных» / Л. А. Скворцова, МИРЭА – Российский технологический университет, 2021.