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:
Студентка группы ИНБО-01-20 «10» ноября 2021 г. _____ Тульцова А.Д.
(подпись)

Преподаватель «10» 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 8.

Задача	Метод
Черепашке нужно попасть из пункта A в пункт B. Поле движения разбито на квадраты. Известно время движения вверх и вправо в каждой клетке (улицы). На каждом углу она может поворачивать только на север или только на восток. Найти минимальное время, за которое черепашка может попасть из A в B.	Динамическое программирование

Дано:

Размер поля, время движения от одного угла к другому.

Результат.

Минимальное время, за которое черепашка может попасть из пункта A в пункт B.

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 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 вызова алгоритма перебора вершин – вызов метода «грубой силы» и возврат величины кратчайшего пути из одной заданной вершины в другую заданную вершину по матрице кратчайших путей;
5. Метод вызова алгоритма Дейкстры – вызов метода «Дейкстры» и возврат величины кратчайшего пути между двумя заданными вершинами графа по массиву кратчайших путей.

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 := 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;
   вывод(количество_сравнений);
   возврат min_dist.
```

Метод вызова алгоритма перебора вершин:

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

Метод «Дейкстры»:

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

Метод вызова алгоритма Дейкстры:

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

Код программы.

Класс графа:

```
# Класс, реализующий граф и некоторые операции с ним.
class Graph:
    # Конструктор класса.
    def __init__(self, m, n):
        self.hor_vertex = m + 1  # Количество вершин поля по горизонтали.
        self.vert_vertex = n + 1  # Количество вершин поля по вертикали.
        self.vertexes = self.hor_vertex * self.vert_vertex  # Общее количество вершин поля (узлов графа).
        # Установка спика 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

    # Вставка 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.

    # Алгоритм перебора вершин (метод "грубой силы").
    def brute_force(self):
        from time import perf_counter  # Импорт perf_counter из библиотеки time.
        time_a = perf_counter()  # Время перед работой алгоритма.
        v = len(self.adj_mat)  # Количество узлов графа как длина матрицы смежности.
        min_dist = self.adj_mat  # Копия матрицы смежности для дальнейшего перезаполнения под матрицу кратчайших путей.
        compares = 1  # Подсчёт количества сравнений: первое сравнение перед входом в цикл for.
        for k in range(v):
            compares += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
            for i in range(v):
                compares += 1  # Увеличение количества сравнений на 1: перед входом в цикл for.
                for j in range(v):
                    compares += 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.
                    compares += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
                compares += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            compares += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
        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 сравнений: {compares}.")  # Вывод количества сравнений для данного метода.
        return min_dist  # Возврат матрицы кратчайших путей.

    # Вызов алгоритма перебора вершин.
    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.

    # Алгоритм Дейкстры.
    def dijkstra(self, node):
        from time import perf_counter  # Импорт perf_counter из библиотеки time.
        time_a = perf_counter()  # Время перед работой алгоритма.
        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.
        compares = 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.
        compares += len(self.adj_list)  # Увеличение количества сравнений на количество итераций предыдущего цикла for.
        min_dist[node] = 0  # Заполнение расстояния до стартовой вершины.
        compares += 1  # Увеличение количества сравнений на 1: перед входом в цикл while.
        while len(nodes_to_visit):  # Пока nodes_to_visit не пустой:
            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.
            compares += 1  # Увеличение количества сравнений на 1: перед условием if.
            if current_node in visited:  # Если выбранная вершина уже посещена:
                # Увеличение количества сравнений на кол-во пройденных элементов списка visited.
                compares += list(visited).index(current_node)
                continue  # Запуск следующего прохода цикла без выполнения оставшегося тела цикла.
            visited.add(current_node)  # Добавление выбранной вершины в список посещённых.
            # next_weight - вес из текущей вершины, next_node - прикреплённая вершина, в которую необходимо попасть.
            compares += 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.
                compares += 1  # Увеличение количества сравнений на 1: перед условием if.
                if weight + next_weight < min_dist[next_node] and next_node not in visited:
                    min_dist[next_node] = weight + next_weight  # Обновление расстояния.
                    nodes_to_visit.append((weight + next_weight, next_node))  # Добавление в список для посещения.
                compares += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла for.
            compares += 1  # Увеличение количества сравнений на 1: с каждой итерацией цикла while.
        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 сравнений: {compares}.")
        return min_dist  # Возврат множества из словарей {номер_узла: кратчайший путь до него от заданного узла}.
```

Рисунок 1 – Класс графа.

```
    # Вызов алгоритма 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.

    # Приложение для нахождения величины кратчайшего пути для черепашки в поле, размером m * n.
    def app_shortest_path(self, hor_edges, vert_edges):  # Для заданных списков горизонтальных и вертикальных рёбер:
        for node in range(self.vertexes):  # Для каждой вершины:
            if node < self.vertexes - self.hor_vertex:  # Если вершина не на нижней границе поля:
                if (node + 1) % self.hor_vertex != 0 or node == 0:  # Если вершина не на правой границе поля:
                    # Соединение текущей вершины с соседней справа.
                    self.connect(node, node + 1, hor_edges[node - (node + 1) // self.hor_vertex])
                    # Соединение соседней снизу вершины с текущей.
                    self.connect(node + self.hor_vertex, node, vert_edges[node])
                else:  # Иначе (если вершина на правой границе поля):
                    # Соединение только соседней снизу вершины с текущей.
                    self.connect(node + self.hor_vertex, node, vert_edges[node])
            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, hor_edges[node - (node + 1) // self.hor_vertex])
        start_node = self.vertexes - 1 - n  # Стартовая вершина для черепашки.
        end_node = n  # Конечная вершина для черепашки.
        '''# 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.'''
        print('Рассмотрим метод "грубой силы" для поиска кратчайшего пути.')
        print(f'Величина кратчайшего пути: {self.shortest_path_brute_force(start_node, end_node)}.')
        print('Рассмотрим один из методов динамического программирования для поиска кратчайшего пути.')
        print(f'Величина кратчайшего пути: {self.shortest_path_dijkstra(start_node, end_node)}.')
```

Рисунок 2 – Класс графа (продолжение).

Основная функция для тестирования:

```
# Главная функция.
if __name__ == '__main__':
    size = list(map(int, input("Введите размер поля m * n в формате 'm n': ").split()))  # Задание размера поля.
    m = size[0]  # Количество строк (количество рёбер в строке = количество вершин в строке - 1).
    n = size[1]  # Количество столбцов (количество столбцов в строке = количество вершин в столбце - 1).
    w_graph = Graph(m, n)  # Создание пустого графа заданного размера.
    input_type = bool(int(input("Введите '1', если хотите ввести данные вручную, или '0', "
                                "чтобы создать случайное поле заданного размера: ")))
    if input_type is True:  # При вводе '1' данные будут введены вручную:
        print("Построчно вводите веса рёбер слева направо.")
        hor_edges = list()  # Список горизонтальных рёбер.
        vert_edges = list()  # Список вертикальных рёбер.
        for i in range(m):  # Для каждой строки поля:
            row_hor_edges = list(map(int, input(f"Введите верхние горизонтальные рёбра {i + 1}-й строки: ").split()))
            row_vert_edges = list(map(int, input(f"Введите вертикальные рёбра {i + 1}-й строки: ").split()))
            # Добавление полученных верхних горизонтальных рёбер в общий список горизонтальных рёбер.
            for element in range(n):
                hor_edges.append(row_hor_edges[element])
            # Добавление полученных вертикальных рёбер в общий список вертикальных рёбер.
            for element in range(n + 1):
                vert_edges.append(row_vert_edges[element])
        row_hor_edges = list(map(int, input(f"Введите нижние горизонтальные рёбра {m}-й строки: ").split()))
        # Добавление полученных нижних горизонтальных рёбер в общий список горизонтальных рёбер.
        for element in range(n):
            hor_edges.append(row_hor_edges[element])
    else:  # При вводе '0' данные будут сгенерированы случайным образом:
        import random  # Импорт библиотеки random.
        hor_edges = list()  # Список горизонтальных рёбер.
        vert_edges = list()  # Список вертикальных рёбер.
        for i in range(m):  # Для каждой строки поля:
            # Добавление верхних горизонтальных рёбер в общий список горизонтальных рёбер.
            for element in range(n):
                hor_edges.append(random.randint(1, 10))  # Добавление ребра со случайным весом от 1 до 10 включительно.
            # Добавление вертикальных рёбер в общий список вертикальных рёбер.
            for element in range(n + 1):
                vert_edges.append(random.randint(1, 10))  # Добавление ребра со случайным весом от 1 до 10 включительно.
        # Добавление нижних горизонтальных рёбер в общий список горизонтальных рёбер.
        for element in range(n):
            hor_edges.append(random.randint(1, 10))  # Добавление ребра со случайным весом от 1 до 10 включительно.
    print("Данные получены. Определяется кратчайший путь для черепашки из точки A в точку B.")
    w_graph.app_shortest_path(hor_edges, vert_edges)  # Нахождение величины кратчайшего пути для черепашки.
    print()
    print("Тестирование завершено.")
```

Рисунок 3 – Основная функция.

Тестирование программы.

Тестирование программы на заданном поле движения.

Для тестирования было выбрано поле движения черепашки размером 3*3 (см. Рисунок 4). Стрелками выделен кратчайший путь (минимальное затрачиваемое время) для черепашки из точки A в точку B; его величина – 21.

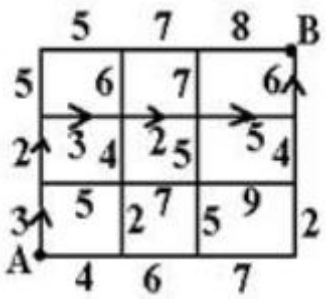

Рисунок 4 – Поле движения черепашки.

Итак, было запущено тестирование программы. Сначала был указан размер поля движения черепашки, затем построчно вводились рёбра поля движения (см. Рисунок 5) после выбора формата ввода вручную.

```
Введите размер поля m * n в формате 'm n': 3 3
Введите '1', если хотите ввести данные вручную, или '0', чтобы создать случайное поле заданного размера: 1
Построчно вводите веса рёбер слева направо.
Введите верхние горизонтальные рёбра 1-й строки: 5 7 8
Введите вертикальные рёбра 1-й строки: 5 6 7 6
Введите верхние горизонтальные рёбра 2-й строки: 3 2 5
Введите вертикальные рёбра 2-й строки: 2 4 5 4
Введите верхние горизонтальные рёбра 3-й строки: 5 7 9
Введите вертикальные рёбра 3-й строки: 3 2 5 2
Введите нижние горизонтальные рёбра 3-й строки: 4 6 7
Данные получены. Определяется кратчайший путь для черепашки из точки A в точку B.
```

Рисунок 5 – Ввод данных.

После завершения ввода данных программа осуществляет поиск величины кратчайшего пути двумя способами: методом «грубой силы» (алгоритмом перебора вершин) и методом динамического программирования (алгоритмом Дейкстры) (см. Рисунок 6). Чтобы сделать анализ результатов более удобным, в процессе выполнения алгоритмов также подсчитывалось количество сравнений и замерялось затраченное время.

```
Рассмотрим метод "грубой силы" для поиска кратчайшего пути.
Затраченное время на работу алгоритма: 0.0015760 с.
Количество сравнений: 8737.
Величина кратчайшего пути: 21.
Рассмотрим один из методов динамического программирования для поиска кратчайшего пути.
Затраченное время на работу алгоритма: 0.0000514 с.
Количество сравнений: 126.
Величина кратчайшего пути: 21.

Тестирование завершено.
```

Рисунок 6 – Полученные результаты.

В результате для данного примера метод «грубой силы» уступает выбранному методу динамического программирования по скорости примерно в 3 раза, а количество сравнений для метода «грубой силы» – 8737. Алгоритм Дейкстры способен уменьшить затраченное время на поиск величины кратчайшего пути и количество сравнений (в ~69 раз) путём снижения общего количества переборов за счёт использования принципа динамического программирования – разбиения общей задачи на более простые подзадачи, а именно – следуя методу «Дейкстры», нет необходимости осуществлять проход по каждой вершине более одного раза; также данный алгоритм постоянно осуществляет движение к наиболее «выгодной» вершине, на каждом шаге выбирая наиболее короткий путь из возможных. Таким образом сложность алгоритма значительно снижается, как и время его выполнения с количеством выполняемых операций сравнения.

Тестирование программы на сгенерированном случайным образом поле движения.

Теперь для тестирования было выбрано поле движения черепашки размером 14*16. Для упрощения ввода оно было заполнено автоматически: веса рёбер были сгенерированы случайным образом в пределах от 1 до 10 включительно. Благодаря реализованному интерфейсу, после запуска программы достаточно было задать размер поля и указать в качестве формата ввода заполнение поля случайным образом (см. Рисунок 7).

```
Введите размер поля m * n в формате 'm n': 14 16
Введите '1', если хотите ввести данные вручную, или '0', чтобы создать случайное поле заданного размера: 0
Данные получены. Определяется кратчайший путь для черепашки из точки A в точку B.
```

Рисунок 7 – Ввод данных.

Далее программа выполнила для созданного поля движения поиск величины кратчайшего пути двумя способами: методом «грубой силы» (алгоритмом перебора вершин) и методом динамического программирования (алгоритмом Дейкстры) (см. Рисунок 8).

```
Рассмотрим метод "грубой силы" для поиска кратчайшего пути.
Затраченное время на работу алгоритма: 5.1884621 с.
Количество сравнений: 33293311.
Величина кратчайшего пути: 103.
Рассмотрим один из методов динамического программирования для поиска кратчайшего пути.
Затраченное время на работу алгоритма: 0.0010637 с.
Количество сравнений: 3675.
Величина кратчайшего пути: 103.

Тестирование завершено.
```

Рисунок 8 – Полученные результаты.

Как видно по результатам – и тот, и другой методы так же вывели одинаковый ответ. Однако теперь метод «грубой силы» уступает выбранному методу динамического программирования (алгоритму Дейкстры) по скорости примерно в 4878 раз, а количество сравнений для метода «грубой силы» уже более 33 млн., что делает его очень неэффективным и по памяти, и по времени. При этом с применением методов динамического программирования (в данном случае – алгоритма Дейкстры) поставленная задача решается примерно за тысячную долю секунды, а количество сравнений относительно небольшое (~ в 10000 раз меньше в сравнении с решением методом «грубой силы»).

Таким образом, методы динамического программирования позволяют значительно упростить решение поставленной задачи, разбивая её на более простые подзадачи; посредством снижения количества переборов уменьшается затрачиваемое время на выполнения алгоритма, а также количество выполняемых операций сравнения.

Вывод

В ходе работы был разработан алгоритм решения задачи поиска кратчайшего пути с применением метода динамического программирования; реализована программа; 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.