

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.

CОДЕРЖАНИЕ

[Цель работы 3](#_Toc86702501)

[Постановка задачи 3](#_Toc86702502)

[Подход к решению. 3](#_Toc86702503)

[Алгоритмы операций на псевдокоде. 5](#_Toc86702504)

[Код программы. 9](#_Toc86702505)

[Тестирование программы. 12](#_Toc86702506)

[Вывод 16](#_Toc86702507)

[Список информационных источников 17](#_Toc86702508)

# 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:



Рисунок 1 – Класс графа.



Рисунок 2 – Класс графа (продолжение).

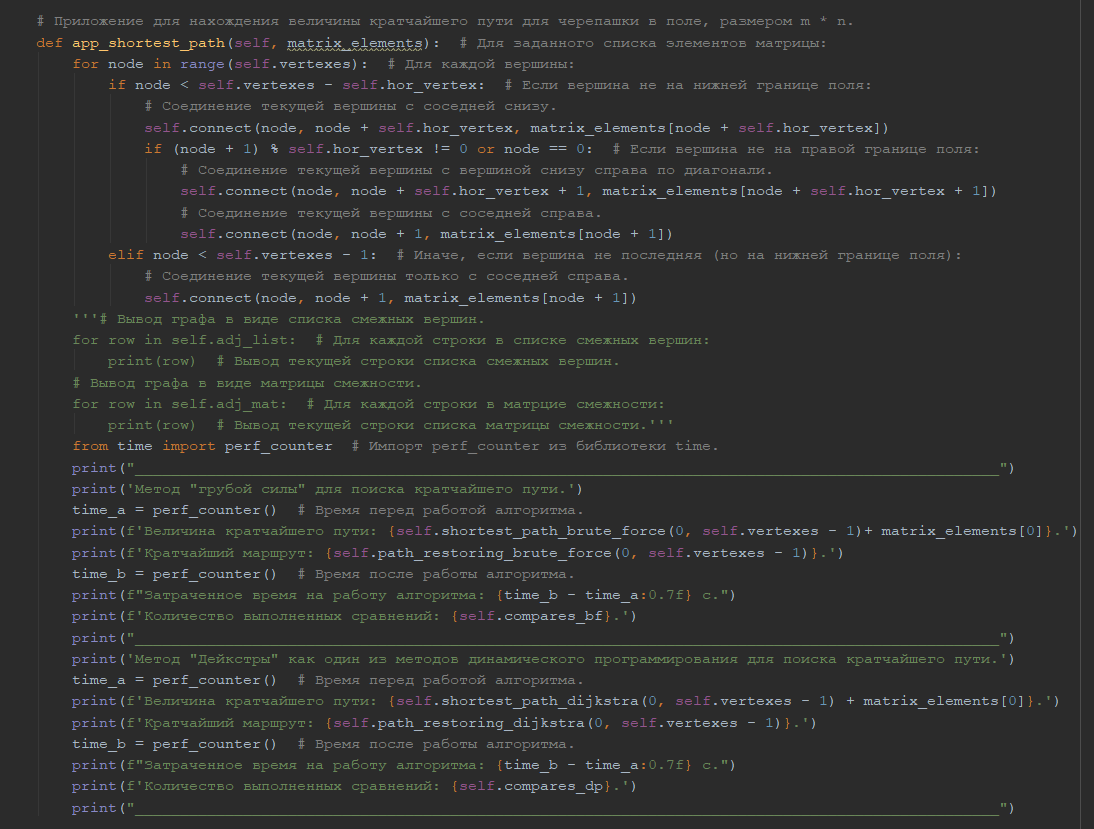


Рисунок 3 – Класс графа (продолжение).

Основная функция для тестирования:

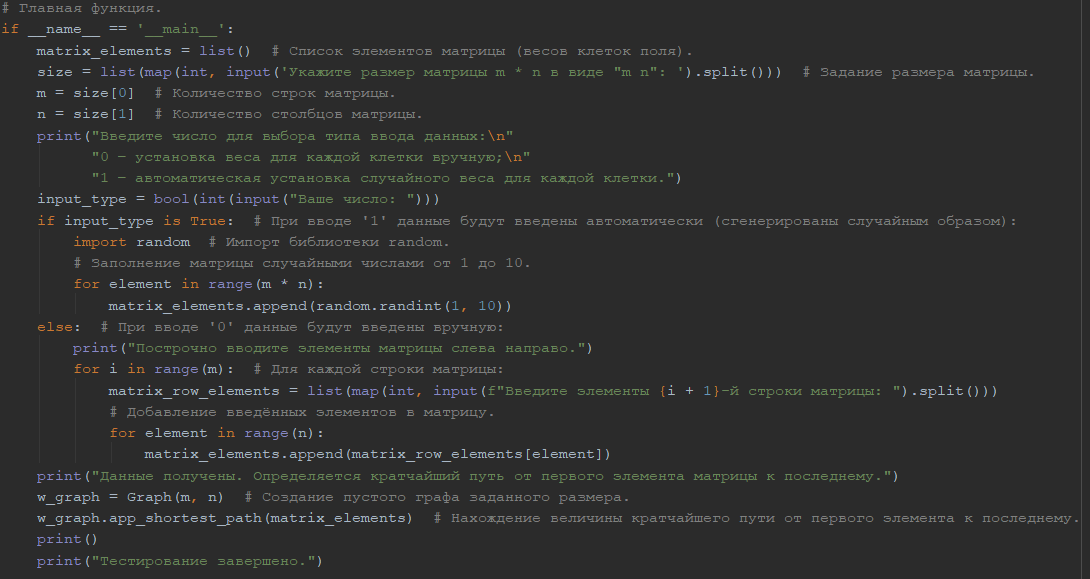


Рисунок 4 – Основная функция.

## 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.

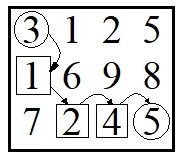


Рисунок 5 – Поле для тестирования.

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.

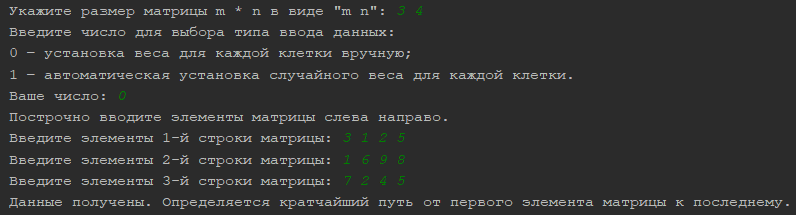


Рисунок 6 – Ввод данных.

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.

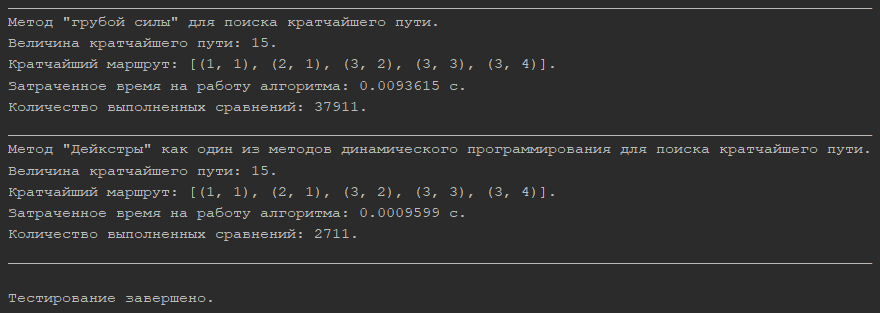


Рисунок 7 – Полученные результаты.

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 (c индексами *i* = 1, *j* = 1) до последнего элемента матрицы (с индексами *i* = *m* = 3, *j* = *n* = 4) также выведен верно: *a*11 → *a*21 → *a*32 → *a*33 → *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).

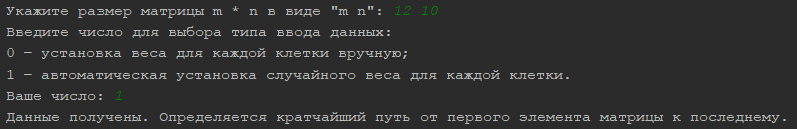


Рисунок 8 – Ввод данных.

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).

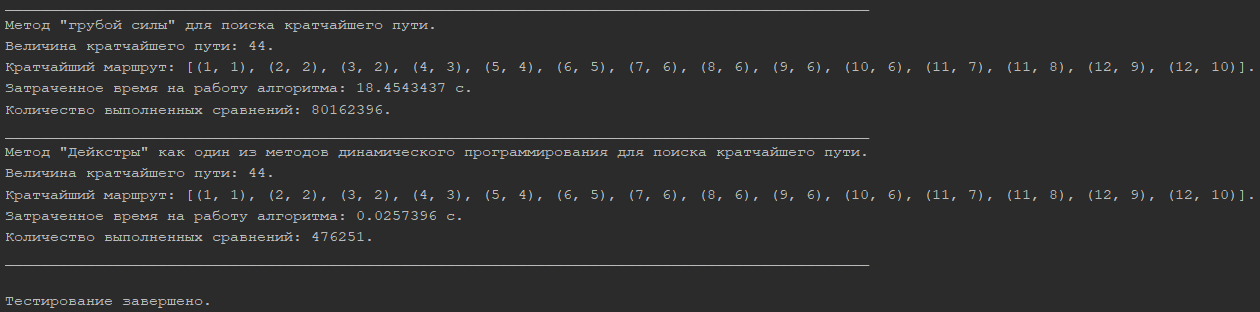


Рисунок 9 – Полученные результаты.

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\*106 (что тоже достаточно мало в сравнении с ~8\*106).

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.