МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»
(СПбГЭУ)

Факультет информатики и прикладной математики

Кафедра прикладной математики и экономико-математических методов

КУРСОВАЯ РАБОТА

по дисциплине:
«Исследование операций»

Тема: «Моделирование оптимизационных задач средствами динамического программирования»

Направление: 01.03.02 Прикладная математика и информатика

Направленность: Прикладная математика и информатика в экономике и управлении

Обучающийся: Бронников Егор Игоревич

Группа: ПМ-1901 Подпись: _____

Проверил: Чернов Виктор Петрович

Должность: профессор

Оценка: _____ Дата: _____

Подпись: _____

Санкт-Петербург
2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Большое количество производственных задач можно сформулировать в виде математической задачи на экстремум. К некоторым из этих задач применимы классические методы оптимизации и математического анализа. Но для значительной части задач производства и управления подобные методы не применимы или малоэффективны. Нахождение экстремума классическими методами зачастую приводит к тому, что на следующем этапе решения приходится решать новые задачи, ещё более сложные, чем поставленные первоначально. Даже в тех случаях, когда задача формулируется, как задача математического программирования, применение необходимых условий для построения оптимального решения очень часто не приводит к нужному результату. В производственных задачах оптимизации к максимуму целевой функции легче бывает подойти поэтапно, чем аналитически решить систему уравнений, определяемую возможными ограничениями поставленной задачи. И даже, в том случае, когда задача математического программирования может быть решена, проверка найденного решения может оказаться довольно сложной и тем сложнее, чем больше аргументов у функции.

Курсовая работа посвящена моделированию оптимизационных задач средствами динамического программирования.

Целью курсовой работы является обоснование метода динамического программирования при моделировании оптимизационных задач, решение задач аналитически и с помощью программных средств.

Объект исследования – оптимизационные модели, решаемые методом динамического программирования.

Предмет исследования – алгоритм метода динамического программирования.

Задачи работы:

1) изучить общий подход динамического программирования;

2) выявить оптимизационные модели, решаемые методом динамического программирования;

3) продемонстрировать применение метода динамического программирования при решение оптимизационных задач аналитически;

4) выполнить программную реализацию некоторых моделей.

Актуальность, выполненной работы можно обосновать тем, что решение ряда оптимизационных задач, например, производственного управления, можно упростить, если процесс управления осуществляется поэтапно, заменив нахождение точек экстремума целевой функции многих переменных многократным нахождением точек экстремума функции одного или небольшого числа переменных, что возможно, если воспользоваться методом динамического программирования.

Результатом работы является программы решения нескольких оптимизационных задач, таких как задача о рюкзаке, задачи о замене оборудования, задача о распределении инвестиций.

1. МЕТОД ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

1.1 Общая постановка задачи динамического программирования

Основоположником динамического программирования можно считать русского математика А. А. Маркова. Его исследовательские работы были продолжены в 1940-х годах американским математиком А. Вальдом, одним из основателей исследовательского анализа.

Однако наиболее полно и систематизировано сформулировать основные принципы оптимального управления многошаговыми процессами впервые удалось американскому математику Р. Беллману. Задачи управления запасами были первыми задачами, которые решались этим методом.

Динамическое программирование является разделом математической науки, в основе которой лежит изучение экстремальных задач управления (методы вычислительной математики, которые применяются для поиска экстремумов функций), планирование и разработка методов их решения, в котором процесс принятия решения разбивается на отдельные этапы. Также можно сказать, что динамическое программирование – это способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной.

В общей постановке задачу динамического программирования можно сформулировать следующим образом. Управляемая физическая система S, характеризуется определённым набором параметров. Требуется построить оптимальное решение u^*, на множестве допустимых решений, переводящее систему из начального состояния x_0 в конечное состояние x_n, обеспечив целевой функции нужный экстремум.

Алгоритмы динамического программирования используются для поиска решения не сразу для всей сложной задачи, а для поиска оптимального решения для нескольких более простых задач аналогичного содержания, на которые распадается исходная задача. Также данные алгоритмы могут применяться для подсчёта количества этих решений.

То есть в динамическом программировании задача разделяется на подзадачи, и решения этих подзадач объединяются вместе для достижения общего решения основной задачи.

Подзадачи могут быть рекурсивно вложены в более крупные задачи, тогда методы динамического программирования применимы, и существует связь между значением более крупной проблемы и значениями подзадач. В литературе такое соотношение называется уравнением Беллмана.

При использовании алгоритмического подхода «разделяй и властвуй», подзадача может быть решена несколько раз. Динамическое программирование решает каждую из этих подзадач только один раз, тем самым уменьшив количество вычислений, а затем сохраняет результат, избегая повторного вычисления на более позднем этапе, когда требуется решение для этой подзадачи.

Динамическое программирование используется для решения задач оптимизации (например, поиск кратчайшего пути), где может существовать множество решений, но интерес представляет только поиск оптимального решения.

Задачи должны иметь свойства перекрывающихся подзадач. Иными словами, решаемая задача может быть разбита на подзадачи, которые многократно используется, причём рекурсивный алгоритм решает одну и ту же подзадачу много раз, а не создаёт новую подзадачу. Например, числа Фибоначчи. В конечном итоге, оптимальное решение может быть построено из оптимальных решений подзадач.

Следует отметить, что во многих случаях алгоритмы перебора могут работать быстрее, чем алгоритмы динамического программирования, но они не гарантируют оптимальное решение задачи.

1.2 Уравнение Беллмана

Согласно Беллману, основной принцип оптимальности управления многошаговыми процессами может быть словесно выражен следующим образом: «Оптимальное поведение обладает тем свойством, что, каковы бы ни

были исходное состояние и первоначальное решение, последующие решения должны составлять оптимальное поведение относительно состояния, получающегося в результате первоначального решения». Иными словами, любой участок оптимальной траектории, в том числе и завершающий, также являются оптимальным, а ошибки в управлении, приводящие к отклонениям от оптимальной траектории, впоследствии не могут быть исправлены.

Рассмотрим функции $B_0(x_0), B_1(x_1), \dots, B_n(x_n)$. Функции $B_i(x_i), i = 0,1, \dots, n$ представляют собой максимальные значения сумм частных целевых функций $z_{i+1}(x_i, u_{i+1}) + \cdots + z_n(x_{n-1}, u_n)$, вычисляемые по всем допустимым «укороченным» наборам управлений $(u_{i+1}, \dots, u_n)$. Иными словами, $B_i(x_i)$ – условно-оптимальное значение целевой функции при переводе системы из состояния x_i после шага с номером i в конечное состояние x_n; условность оптимального значения состоит в том, что оно относится не ко всему процессу, а к его заключительной части, и зависит от выбора состояния x_i, являющегося начальным для «укороченного» процесса. Тем самым функции $B_i(x_i)$, называются функциями Беллмана, характеризуют экстремальные свойства управляемой системы S на последних шагах процесса. При этом имеет место соотношение:

$$B_n(x_n) = 0 \qquad (1)$$

Формула 1. Соотношение функции Беллмана.

Это выражение справедливо, так как состояние x_n уже является конечным, дальнейших изменений состояний системы не происходит, и соответствующий экономический эффект равен 0.

Принцип оптимальности Беллмана, лежащий в основе метода динамического программирования решений рассматриваемых задач, выражается следующим основным функциональным уравнением:

$$B_{i-1}(x_{i-1}) = \max_{u_i}\{z_i(x_{i-1}, u_i) + B_i(x_i) \mid x_i = f_i(x_{i-1}, u_i)\} \qquad (2)$$

Формула 2. Функциональное уравнение Беллмана.

в котором индекс i изменяется по номерам всех шагов процесса в обратном порядке: $i = n, n-1, \dots, 2, 1$;

$z_i(x_{i-1}, u_i)$ – значение целевой функции на k-м шаге.

По своей структуре функциональное уравнение Беллмана является рекуррентным. Это означается, что в последовательности функций $B_0(x_0), B_1(x_1), \dots, B_n(x_n)$ каждая предшествующая выражается через последующую.

Уравнение Беллмана было впервые применено к теории управления техническими системами и другим темам прикладной математики, а затем стало важным инструментом экономической теории.

Чтобы понять функциональное уравнение Беллмана, необходимо понять несколько основных понятий. Во-первых, любая задача оптимизации имеет какую-то цель: минимизация времени в пути, минимизация затрат, максимизация прибыли и т. д. Математическая функция, которая описывает эту задачу, называется целевой функцией.

Динамическое программирование разбивает задачу многопериодного планирования на более простые этапы в разные моменты времени. Следовательно, необходимо отслеживать, как ситуация с решениями меняется со временем. Информация о текущей ситуации, которая необходима для принятия правильного решения, называется «состоянием». Например, чтобы решить, сколько потреблять и тратить в каждый момент времени, люди должны знать (среди прочего) своё первоначальное богатство. Следовательно, богатство будет одной из их переменного состояния, но, вероятно, будут и другие.

Переменные, выбранные в любой данный момент времени, часто называю контрольными переменными. Например, учитывая их текущее состояние, люди могут решить, сколько потреблять сейчас. Выбор управляющих переменных теперь может быть эквивалентен выбору следующего состояния; в более общем случае на следующее состояние влияют другие факторы, помимо текущего контроля. Например, в простейшем случае сегодняшнее богатство и потребление

могут точно определять завтрашнее богатство хотя, как правило, другие факторы также влияют на завтрашнее богатство.

Подход динамического программирования описывает оптимальный план, находя правило, которое сообщает, какими должны быть элементы управления, учитывая любое возможное значение состояния. Например, потребление зависит только от богатства, мы бы искали правило потребление как функцию богатства. Такое правило, определяющее элементы управления как функцию, называется функцией политики.

Наконец, по определению, оптимальным правилом принятия решения является то, которое достигает наилучшего возможного значения цели. Наилучшее возможное значение цели, записанное как функция состояния, называется функцией значения.

Ричард Беллман показал, что задача динамической оптимизации в дискретном времени может быть сформулирована в рекурсивной пошаговой форме, известной как обратная индукция, записав взаимосвязь между функцией значения в одном периоде и функцией значения в следующем периоде.

Существуют условия, которым должна удовлетворять общая задача оптимизации, чтобы её можно было описать методом динамического программирования:

- задача оптимизации формулируется как итоговый многошаговый процесс управления;
- целевая функция, является аддитивной и равна сумме целевых функций каждого шага оптимизации;
- выбор управления на каждом шаге зависит только от состояния самой системы на этом шаге и не влияет на предшествующие шаги (отсутствие обратной связи);
- состояние системы S_k после каждого шага управления зависит исключительно от предшествующего состояния системы и этого управляющего

воздействия x_k(нет последействия), причём оно может быть записано в виде уравнения состояния системы;

- на каждом шаге управление x_k зависит от конечного числа управляющих переменных, а состояния системы S_k от конечного числа параметров;
- оптимальное управление представляет собой вектор X, который определяется как последовательность оптимальных пошаговых управлений, число которых и определяет количество шагов задачи.

Основные свойства задач, в которых можно применять метод динамического программирования:

- задача должна допускать интерпретацию как многошаговый процесс принятия решения;
- задача должна быть определена для любого числа шагов и иметь структуру, не зависящую от их числа;
- при рассмотрении задачи на каждом шаге должно быть задано множество параметров, описывающих состояние системы;
- выбор решения (управления) на каждом шаге не должен оказывать влияния на предыдущие решения.

2. ЗАДАЧИ, РЕШАЕМЫЕ МЕТОДОМ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Особенность производственных задач, решаемых методом динамического программирования, заключается в том, что процесс, протекающий в системе, зависит либо от времени (от этапов), либо имеет многоступенчатую структуру. Метод решения задач динамического программирования состоит в том, что оптимальное управление строится поэтапно шаг за шагом. На каждом этапе оптимизируется только один шаг, но при этом учитывается изменение всего процесса, так как управление, оптимизирующее целевую функцию только для данного шага, может привести к неоптимальному эффекту всего процесса. Управление на каждом шаге должно быть оптимальным с точки зрения процесса в целом.

В основе вычислительных алгоритмов динамического программирования лежит принцип оптимальности Р. Беллмана. (Формула 2) Данный принцип включается в себя три основных этапа:

1) предварительный этап;
2) этап условной оптимизации;
3) этап безусловной оптимизации.

Предварительный этап проводится с целью уменьшения вычислительной работы на последующем этапе решения и, по существу, заключается в нахождении всех допустимых значений управлений u_i и фазовых переменных x_i, то есть фактически область определения функций $B_i(x_i)$ или, в более сложных случаях, множеств, содержащих эти области определения. Иными словами, на данном этапе отбрасываются все заведомо неподходящие, нереализуемые значения фазовых и управляющих переменных. Проводится предварительный этап в естественном порядке от первого шага к последнему: $i = 1,2, \dots, n$, а опираются соответствующие расчёты на уравнение процесса $x_i = f_i(x_{i-1}, u_i)$. Данный этап особенно удобен при табличном способе задания функций, фигурирующих в условии задачи.

Условная оптимизация осуществляется поэтапно при движении из конечного состояния x_n системы S в первоначальное состояние x_0 путём построения на каждом этапе условно-оптимального управления и нахождения условно-оптимального значения функции цели для каждого шага.

Безусловная оптимизация осуществляется в обратном направлении: от первого шага к последнему, в результате чего находятся уже оптимальные управления на каждом шаге с точки зрения всего процесса. На втором этапе отрабатываются рекомендации, полученные на этапе условной оптимизации.

Много интересных многошаговых процессов принятия решений возникает при управлении производственными процессами. Рассмотрим некоторые из них.

Задача о замене оборудования. В настоящее время промышленные предприятия производят замену оборудования в сроки, диктуемые не на основе интуиции, а на основе математических расчётов. Управленцы промышленных предприятий должны владеть методами составления плана замены машинного оборудования, в целях оптимизации его использования. Задачу по замене оборудования можно рассматривать как многоэтапный процесс, к которому применимы методы динамического программирования. Применение методов динамического программирования позволяет максимизировать прибыль или минимизировать затраты.

Задача оптимального распределения инвестиций (ресурсов). В производственной практике очень часто возникают задачи на оптимальное распределение ресурсов между предприятиями или внутри предприятия. Кроме того, к задача оптимального распределения инвестиций можно отнести ещё ряд задач. Например, задачу о размещение по торговым и складским помещениям какого-либо товара, задачу о распределении средств между различными отраслями промышленности и т. п.

Задача о рюкзаке (Задача о загрузке транспортного средства). В рюкзак требуется погрузить несколько видом предметов, так, чтобы ценность рюкзака была максимальной. Также можно переформулировать данную задачу в задачу о загрузке транспортного средства. В транспортное средство требуется

погрузить несколько видом груза, так, чтобы результат загрузки был эффективным. Например, максимизировать стоимость груза, размещённого в транспортном средстве, известна грузоподъёмность транспортного средства, вес единицы груза и соответствующая эффективность.

Вышеперечисленные задачи не составляют полный список производственных задач, решаемых динамическим программированием. Аппарат динамического программирования используется и в численных решениях классических функциональных уравнений, обыкновенных дифференциальных уравнений и дифференциальных уравнений с частными производными.

В разделе рассматриваются решения ряда производственных задач методом динамического программирования аналитически.

2.1 Задача о замене оборудования

Постановка задачи:

Разработать оптимальную стратегию замены оборудования возраста k лет в плановом периоде продолжительностью N лет, если известны:

$r(t)$ – стоимость продукции, производимой в течение года на оборудовании возраста t лет ($t = \overline{0, N}$);

$u(t)$ – ежегодные расходы, связанные с эксплуатацией оборудования возраста t лет ($t = \overline{0, N}$);

$s(t)$ – остаточная стоимость оборудования возраста t лет ($t = \overline{0, N}$);

P – стоимость нового оборудования и расходы, связанные с установкой, наладкой и запуском.

В начале каждого года имеется две возможности: сохранить оборудование и получить прибыль $r(t) - u(t)$ или заменить его и получить прибыль $s(t) - P + r(0) - u(0)$. Прибыль от использования оборудования в последнем N-м году планового периода запишется в следующем виде:

$$F_N(t) = max \begin{cases} r(t) - u(t) & - \text{сохранение} \\ s(t) - P + r(0) - u(0) & - \text{замена} \end{cases} \quad (3)$$

Формула 3. Прибыль от использования оборудования в N-м году.

А прибыль от использования оборудования в период с n-го по N-й год:

$$F_n(t) = max \begin{cases} r(t) - u(t) + F_{n+1}(t+1) & - \text{сохранение} \\ s(t) - P + r(0) - u(0) + F_{n+1}(1) & - \text{замена} \end{cases} \quad (4)$$

Формула 4. Прибыль от использования оборудования в период с n-го по N-й год.

где $F_{n+1}(t+1)$ – прибыль от использования оборудования в период с $(n+1)$-го по N-й год.

В случае, если оба управления («сохранить» и «заменить») приводят к одной и той же прибыли, то целесообразно выбрать управление «сохранить».

Рассмотрим следующий пример. Следует найти оптимальную стратегию замены оборудования возраста 3 года на период продолжительностью 10 лет, если для каждого года планового периода известны стоимость $r(t)$ продукции, производимой с использованием этого оборудования, и эксплуатационные расходы $u(t)$. Известны также остаточная стоимость, не зависящая от возраста оборудования и составляющая 4 денежных ед., и стоимость нового оборудования, равная 18 денежных ед., не меняющаяся в плановом периоде.

t	0	1	2	3	4	5	6	7	8	9	10
$r(t)$	31	30	28	28	27	26	26	25	24	24	23
$u(t)$	8	9	9	10	10	10	11	12	14	16	18

Таблица 1. Данные для задачи о замене оборудования.

Решение:

1 этап. Условная оптимизация.

1 шаг. $n = N = 10$. Начнём процедуру условной оптимизации с последнего, десятого года планового периода. Для этого шага состояние системы: $t = 0,1,2,\dots,9,10$. Функциональное уравнение с учётом числовых данных примера принимает следующий вид:

$$F_{10}(t) = max \begin{cases} r(t) - u(t) \\ 4 - 18 + 31 - 8 \end{cases} = max \begin{cases} r(t) - u(t) & - \text{сохранение} \\ 9 & - \text{замена} \end{cases}$$

Полученные результаты для каждого $t = 0,1,2 \dots,9,10$ занесём в таблицу. (Рисунок 1)

2 шаг. $n = 9$. Проанализируем девятый год планового периода. Для второго шага возможны состояния системы $t = 0,1,2 \dots ,9,10$. Функциональное уравнение с учётом числовых данных примера принимает вид:

$$F_9(t) = max \begin{cases} r(t) - u(t) + F_{10}(t+1) \\ 4 - 18 + 31 - 8 + F_{10}(1) \end{cases}$$

$$= max \begin{cases} r(t) - u(t) + F_{10}(t+1) - \text{сохранение} \\ 9 + F_{10}(1) \qquad\qquad\quad - \text{замена} \end{cases}$$

Полученные результаты для каждого $t = 0,1,2 \dots ,9,10$ занесём в таблицу. (Рисунок 1)

Продолжая вычисления описанным способ, постепенно заполняем всю матрицу функции Беллмана:

Матрица функции Беллмана

F_n \ t	0	1	2	3	4	5	6	7	8	9	10	
10	23	21	19	**18**	17	16	15	13	10	9	9	9
9	44	40	**37**	35	33	31	30	30	30	30	30	30
8	63	**58**	54	51	49	49	49	49	49	49	49	49
7	**81**	75	70	67	67	67	67	67	67	67	67	67
6	98	91	86	**85**	84	84	84	84	84	84	84	84
5	114	107	**104**	102	101	100	100	100	100	100	100	100
4	130	**125**	121	119	117	116	116	116	116	116	116	116
3	**148**	142	138	135	134	134	134	134	134	134	134	134
2	165	159	154	152	**151**	151	151	151	151	151	151	151
1	182	175	171	**169**	168	168	168	168	168	168	168	168

Рисунок 1. Матрица функции Беллмана.

2 этап. Безусловная оптимизация.

В начале исследуемого десятилетнего периода возраст оборудования составляет 3 года. Находим в таблице на пересечении строки $F_1(t)$ и столбца $t = 3$ значение максимальной прибыли – $F_1(3) = 169$. Найдём теперь оптимальную политику, обеспечивающую эту прибыль. Значение 169 принимает значение «сохранения». Это означает, что в начале первого года принимается решение о сохранении оборудования. К началу второго года возраст оборудования $3 + 1 = 4$ года. Расположенная на пересечении строки $F_2(t)$ и столбца $t = 4$ клетка принимает значения «сохранения», следовательно, и второй год нужно работать на имеющемся оборудовании. К началу третьего года возраст оборудования $4 +$

$1 = 5$ лет. Расположенная на пересечении строки $F_3(t)$ и столбца $t = 5$ клетка принимает значение «заменить», следовательно, в начале третьего года следует заменить оборудование. К началу четвёртого года возраст оборудования составит один год. Расположенная на пересечении строки $F_4(t)$ и столбца $t = 1$ клетка принимает значение «сохранить», следовательно, четвёртый год следует работать на имеющемся оборудовании. Продолжая рассуждать таким образом, последовательно, находим: $F_5(2) = 104, F_6(3) = 85, F_7(4) = 67, F_8(1) = 58, F_9(2) = 37, F_{10}(3) = 18$.

Выводы:

Итак, на оборудовании возраста 3 года следует работать 2 года, затем произвести замену оборудования, на новом оборудовании работать 3-й, 4-й, 5-й и 6-й годы, после чего произвести замену оборудования и на следующем оборудовании работать 7-й, 8-й, 9-й и 10-й годы планового периода. При этом прибыль будет максимальной и составит $F_1(3) = 169$ денежных ед.

2.2 Задача о рюкзаке

Постановка задачи:

Дано N предметов, i-предмет имеет массу $w_i > 0$ и стоимость $p_i > 0$. Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины W (вместимость рюкзака), а суммарная стоимость была максимальной.

Можно сформулировать данную задачу следующим образом. Пусть дано N предметов, W – вместимость рюкзака, $w = (w_1, w_2, \dots, w_N)$ – набор положительных целых весов, $p = (p_1, p_2, \dots, p_N)$ – соответствующий ему набор положительных целых стоимостей. Нужно найти набор бинарных величин $B = (b_1, b_2, \dots b_N)$, который обеспечит максимальную стоимость рюкзака, где $b_i = 1$, если предмет i включён в набор, $b_i = 0$, если предмет i не включён.

$$\sum_{i=1}^{N} b_i p_i \to max \tag{4}$$

Формула 4. Целевая функция задачи о рюкзаке.

$$\sum_{i=1}^{N} b_i w_i \leq W$$
$$b_i \in \{0,1\}, \quad \forall\, i \in \{1, \dots, N\} \tag{5}$$

Формула 5. Ограничения задачи о рюкзаке.

Решение задачи о рюкзаке. Пусть $F(k, s)$ есть максимальная стоимость предметов, которые можно уложить в рюкзак вместимости s, если можно использовать только первые k предметов, то есть $(n_1, n_2, \dots, n_k)$ назовём этот набор допустимых предметов для $F(k, s)$. Исходя из этого соображения получается, что $F(k, 0) = 0$ и $F(0, s) = 0$.

Найдём $F(k, s)$. Возможны два варианта:

1) Если предмет k не попал в рюкзак. Тогда $F(k, s)$ равно максимальной стоимости рюкзака с такой же вместимостью и набором допустимых предметов $(n_1, n_2, \dots, n_{k-1})$, то есть $F(k, s) = F(k - 1, s)$.

2) Если k попал в рюкзак. Тогда $F(k, s)$ равно максимальной стоимости рюкзака, где вес s уменьшаем на вес k-ого предмета и набор допустимых предметов $(n_1, n_2, \dots, n_{k-1})$ плюс стоимость k, то есть $F(k - 1, s - w_k) + p_k$.

То есть это можно представить в следующем виде:

$$F(k, s) = \begin{cases} F(k-1, s), & b_k = 0 \\ F(k-1, s - w_k) + p_k, & b_k = 1 \end{cases} \tag{6}$$

Формула 6. Возможные варианты для $F(k, s)$.

Получается, что: $F(k, s) = \max(F(k - 1, s), F(k - 1, s - w_k) + p_k)$. Стоимость искомого набора равна $F(N, W)$, так как нужно найти максимальную стоимость рюкзака, где все предметы допустимы и вместимость рюкзака W.

После того как мы определили максимальную стоимость рюкзака нужно восстановить предметы, которые будут входить в искомый рюкзак. Будем

определять входит ли предмет i в искомый набор. Начинаем с предмета $F(i, w)$, где $i = N, w = W$. Для этого сравниваем $F(i, w)$ со следующими значениями:

1) Максимальная стоимость рюкзака с такой же вместимостью и набором допустимых предметов $(n_1, n_2, \dots, n_{i-1})$, то есть $F(i-1, w)$

2) Максимальная стоимость рюкзака с вместимостью на w_i меньше и набором допустимых предметов $(n_1, n_2, \dots, n_{i-1})$ плюс стоимость p_i, то есть $F(i-1, w-w_i) + p_i$.

Можно заметить, что при построении F мы выбирали максимум их этих значений и записывали в $F(i, w)$. Тогда будем сравнивать $F(i, w)$ с $F(i-1, w)$, если равны, тогда i-предмет не входит в искомый набор, иначе входит.

Рассмотрим следующий пример.

Дано $N = 5$ предметов, вместимость рюкзака составляет $W = 13$ и имеется следующая таблица соотношения предмета, его веса и стоимости:

Номер предмета i	Вес предмета w_i	Стоимость предмета p_i
1	3	1
2	4	6
3	5	4
4	8	7
5	9	6

Таблица 2. Данные для задачи о рюкзаке.

Решение:

1 этап. Условная оптимизация.

Для начала составим матрицу функции Беллмана, она будет выглядеть следующим образом:

Матрица функции Беллмана

k \ s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	1	6	6	6	7	7	7	7	7	7	7
3	0	0	0	1	6	6	6	7	7	10	10	10	11	11
4	0	0	0	1	6	6	6	7	7	10	10	10	13	13
5	0	0	0	1	6	6	6	7	7	10	10	10	13	**13**

Рисунок 2. Матрица функции Беллмана.

Числа от 0 до 13 в первой строчке обозначают вместимость рюкзака. Во второй строке, как только вместимость рюкзака $s \geq 3$, в рюкзак добавляется 1 предмет. Далее мы просто по формуле функции Беллмана рассчитываем оставшиеся элементы и в конечном итоге получается, что стоимость рюкзака $F(5,13) = 13$.

2 этап. Безусловная оптимизация.

Начинаем с клетки $F(5,13) = 13$, сравниваем $F(5,13)$ и $F(4,13)$, получается $F(5,13) = F(4,13)$, следовательно переходим к клетке $F(4,13)$. Сравниваем $F(4,13)$ и $F(3,13)$, $F(4,13) > F(3,13)$, значит берём в набор 4 предмет и перемещаемся к ячейке $F(3,5)$. Продолжаем данный процесс до тех пор, пока значение функции не будет равно нулю. Пройдя данный этап, получилось, что в рюкзак пойдут 2 и 4 предметы.

Матрица функции Беллмана

k \ s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	**0**	0	1	1	1	1	1	1	1	1	1	1	1	
2	0	0	0	1	6	**6**	6	7	7	7	7	7	7	7	1
3	0	0	0	1	6	**6**	6	7	7	10	10	10	11	11	
4	0	0	0	1	6	6	6	7	7	10	10	10	13	**13**	5
5	0	0	0	1	6	6	6	7	7	10	10	10	13	**13**	

Таблица 3. Нахождение предметов, которые обеспечивают оптимум.

Таким образом, в набор входит 2 и 4 предмет, стоимость рюкзака составит 13, а вес рюкзака составит 12.

Так же хочется добавить, что данную задачу можно модифицировать и получить задачу о загрузке транспортного средства или задачу о целочисленном рюкзаке. Она будет формулироваться следующим образом.

Предположим, что имеется транспортное средство загружаемое N различными типами предметов с весом w_i и стоимостью p_i. Максимальная грузоподъёмность равна W. Определить максимальную стоимость груза, вес которого не более W.

Математическая постановка задачи формулируется следующим образом.

$$\sum_{i=1}^{N} x_i c_i \to max \tag{6}$$

Формула 6. Целевая функция задачи о загрузке транспортного средства.

$$\sum_{i=1}^{N} x_i w_i \leq W$$
$$x_i \in Z, \quad \forall\, i \in \{1, \dots, N\} \tag{7}$$

Формула 7. Ограничения задачи о загрузке транспортного средства.

2.3 Задача о распределении инвестиций

Постановка задачи:

В производственное объединение входят N предприятий $\Pi_1, \Pi_2, \dots, \Pi_N$. Руководство объединения решило инвестировать в свои предприятия M условных единиц в общей сумме. Проведённые маркетинговые исследования прогнозируют величину ожидаемой прибыли каждого из предприятий в зависимости от объёма инвестированных средств. Требуется найти такое распределение инвестиций между предприятиями, которое обеспечило бы максимум суммарной ожидаемой прибыли.

Рассмотрим следующий пример.

Инвестор выделяет средства в размере 5 условных единиц, которые должны быть распределены между тремя предприятиями. Известны значения

прироста прибыли в каждом из трёх предприятий. Требуется составить план распределения инвестиций, максимизирующий общий прирост прибыли. Считается, что при нулевых инвестициях ожидается нулевая прибыль.

Инвестируемые средства (усл. ед.)	Приросты прибыли (усл. ед)		
x	$\Pi_1(x)$	$\Pi_2(x)$	$\Pi_3(x)$
1	3,22	3,33	4,27
2	3,57	4,87	7,64
3	4	5,26	10,25
4	4,12	7,34	15,93
5	4,85	9,49	16,12

Таблица 4. Данные для задачи о распределении инвестиций.

Решение:

Число шагов в данной задаче равно 3. Пусть S – количество средств, имеющихся в наличии перед данным шагом, и характеризующих состояние системы на каждом шаге. Управление на i-ом шаге выберем x – количество средств, инвестируемых в i-ое предприятие. Выигрыш $\Pi_i(x_i)$ на i-ом шаге – это прибыль, которую приносит i-ое предприятие при инвестировании в него средств x_i. Если через выигрыш в целом обозначить общую прибыль W, то он будет выглядеть следующим образом.

$$W = \Pi_1(x_1) + \Pi_2(x_2) + \Pi_3(x_3) \quad (8)$$

Формула 8. Общая прибыль в задаче о распределении инвестиций.

Если в наличии имеются средства в количестве S условных единиц и в i-ое предприятие инвестируется x условных единиц, то для дальнейшего инвестирования остаётся $(S - x)$ условных единиц. Таким образом, если на i-ом шаге система находилась в состоянии Sи выбрано управление x, то на $(i + 1)$-шаге система будет находиться в состоянии $(S - x)$, и, следовательно, функция перехода в новое состояние имеет вид: $F_i(S - x) = S - x$.

На последнем i-ом шаге оптимальное управление соответствует количеству средств, имеющихся в наличии, а выигрыш равен доходу, приносимым последним предприятием.

Согласно принципу оптимальности Беллмана, управления на каждом шаге нужно выбирать так, чтобы оптимальной была сумма выигрышей на всех оставшихся до конца процесса шагах, включая выигрыш на данном шаге. Тогда функциональное уравнение Беллмана примет вид:

$$W_i(S) = \max_{x \le S}\{\Pi_i(x) + W_{i+1}(S - x)\} \quad (9)$$

Формула 9. Общая прибыль в задаче о распределении инвестиций

Проведём пошаговую оптимизацию, по результатам которой заполним таблицу.

Итоговые данные						
S	i = 3		i = 2		i = 1	
	$x_3(S)$	$W_3(S)$	$x_2(S)$	$W_2(S)$	$x_1(S)$	$W_1(S)$
1	1	4,27	0	4,27	-	-
2	2	7,64	0	7,64	-	-
3	3	10,25	1	10,97	-	-
4	4	15,93	0	15,93	-	-
5	5	16,12	1	19,26	0	19,26

Рисунок 3. Итоговые данные.

В первой колонке таблицы записываются возможные состояния системы, в верхней строке – номера шагов с оптимальным управлением и выигрышем на каждом шаге, начиная с последнего. Так как для последнего шага $i = 3$ функциональное уравнение Беллмана примет вид: $x_3(S) = S, W_3(S) = \Pi_3(S)$, то две колонки таблицы, соответствующие $i = 3$, заполняются автоматически по таблице исходных данных.

На шаге $i = 2$ основное функциональное уравнение примет вид: $W_2(S) = \max_{x \le S}\{\Pi_2(x) + W_3(S - x)\}$. Поэтому для проведения оптимизации на этом шаге заполним соответствующую таблицу для различных состояний S при шаге $i = 2$.

Таблица состояний при i = 2						
S	X	S-X	$П_2(x)$	$W_3(S-x)$	$П_2(x)+W_3(S-x)$	$W_2(S)$
1	**0**	1	0	4,27	4,27	4,27
	1	0	3,33	0	3,33	
2	**0**	2	0	7,64	7,64	7,64
	1	1	3,33	4,27	7,6	
	2	0	4,87	0	4,87	
3	0	3	0	10,25	10,25	10,97
	1	2	3,33	7,64	10,97	
	2	1	4,87	4,27	9,14	
	3	0	5,26	0	5,26	
4	**0**	4	0	15,93	15,93	15,93
	1	3	3,33	10,25	13,58	
	2	2	4,87	7,64	12,51	
	3	1	5,26	4,27	9,53	
	4	0	7,34	0	7,34	
5	0	5	0	16,12	16,12	19,26
	1	4	3,33	15,93	19,26	
	2	3	4,87	10,25	15,12	
	3	2	5,26	7,64	12,9	
	4	1	7,34	4,27	11,61	
	5	0	9,49	0	9,49	

Рисунок 4. Таблица состояний на 2 шаге.

На шаге $i = 1$ основное функциональное уравнение примет вид: $W_1(S) = \max\limits_{x \le S}\{П_1(x) + W_2(S-x)\}$, а состояние системы перед первым шагом $S = 5$, поэтому для проведения оптимизации на этом шаге заполним соответствующую таблицу.

Таблица состояний при i = 1						
S	X	S-X	$П_1(x)$	$W_2(S-x)$	$П_2(x)+W_2(S-x)$	$W_1(S)$
5	**0**	5	0	19,26	19,26	19,26
	1	4	3,22	15,93	19,15	
	2	3	3,57	10,97	14,54	
	3	2	4	7,64	11,64	
	4	1	4,12	4,27	8,39	
	5	0	4,85	0	4,85	

Рисунок 5. Таблица состояний на 1 шаге.

Видно, что наибольшее значение выигрыша составляет 19,26. При этом оптимальное управление на первом шаге составляет $x_1(S_1) = 0$ при этом $S_1 = 5$, на втором шаге $x_2(S_2) = 1$ при этом $S_2 = S_1 - x_1 = 5$ и на третьем шаге $x_3(S_3) = 4$ при этом $S_3 = S_2 - x_3 = 4$. Это означает, что $(0,1,4)$ – оптимальный план распределения инвестиций между предприятиями.

Таким образом, для получения наибольшей общей прибыли в размере 19,26 условных единиц, необходимо вложить 0 условных единиц в первое

предприятие, 1 условную единицу во второе предприятие и 4 условные единицы в третье предприятие.

В результате было рассмотрено три оптимизационных производственных задачи, также они были решены аналитически методом динамического программирования.

3. РЕАЛИЗАЦИЯ АЛГОРИТМА РЕШЕНИЯ ОПТИМИЗАЦИОННЫХ ЗАДАЧ

3.1 Задача о замене оборудования

Перед разработкой программной реализации решения задачи о замене оборудования было приведено аналитическое решение в среде MS Excel и там был автоматизирован процесс нахождения матрицы Беллмана в соответствии с его функциональным уравнением.

Было принято решение реализовать данный алгоритм на языке программирования Python. По описанной ранее математической модели было реализовано программное обеспечение для решения производственных задач на основе метода динамического программирования.

Для подсказки сопоставления типов был использован модуль $typing$, а для тестирования был использован модуль $unittest$.

```
# Import
from typing import List, Tuple
import unittest
```

Рисунок 6. Модули в реализации задачи о замене оборудования.

Была создана функция $replacing\_equipment$, которая решает поставленную задачу. На вход поступает n – продолжительность работы оборудования (лет), $year$ – возраст оборудования на момент анализа, s – постоянная остаточная стоимость оборудования, P – стоимость нового оборудования, r – список стоимостей продукции, произведённой в течение каждого года планового периода с помощью оборудования, u – список ежегодных затрат, связанных с эксплуатацией оборудования.

На выходе функция выводит кортеж, первый элемент которого представляет собой оптимальный план, а второй элемент – значение целевой функции, то есть максимальная прибыль.

```
def replacing_equipment(n: int, year: int, s: int, P: int, r: List[int], u: List[int]) -> Tuple[List[str], int]:
    """
        @Synopsis
        def replacing_equipment(n: int, year: int, s: int, P: int, r: List[int], u: List[int]) -> Tuple[List[str], int]: ...

        @Description
        Solves the equipment replacement problem by the dynamic programming method.

        @param n: Number of years of planning
        @type n: int
        @param year: Equipment age
        @type year: int
        @param s: Residual value of equipment
        @type s: int
        @param P: New equipment costs
        @type P: int
        @param r: The value of the products produced during the year
        @type r: List[int]
        @param u: Annual costs associated with the operation of equipment
        @type u: List[int]

        @return: Equipment replacement plan for a given period of time `n` and the maximum profit (the target function value).
        @rtype: Tuple[List[str], int]
    """
```

Рисунок 7. Объявление функции.

Далее применяя ранее полученные формулы получаем следующую реализацию условной оптимизации.

```
# Prepare data
matrix = [[0 for _ in range(n+1)] for _ in range(n)]    # Matrix for Bellman function
replacement = [0 for _ in range(n)]                     # Replacement rate

matrix[n-1] = [max(r[t] - u[t], s - P + r[0] - u[0]) for t in range(n+1)]
replacement[n-1] = s - P + r[0] - u[0]

# Conditional optimization
# Filling in the Bellman function matrix
for i in range(n-1, 0, -1):
    for t in range(n):
        matrix[i-1][t] = max(r[t] - u[t] + matrix[i][t+1], s - P + r[0] - u[0] + matrix[i][1])
    replacement[i-1] = s - P + r[0] - u[0] + matrix[i][1]
    matrix[i-1][n] = matrix[i-1][n-1]
```

Рисунок 8. Условная оптимизация.

Аналогично поступаем с безусловной оптимизацией.

```
# Unconditional optimization
column = [row[year] for row in matrix]
value = column[0]

pos = year
plan = []

for i in range(n):
    if pos == matrix[i].index(replacement[i]):  # Save at this time and replace on the future
        plan.append(f"{i+1}")
        plan.append("R")
        pos = 0
    elif pos > matrix[i].index(replacement[i]): # Replace
        plan.append("R")
        plan.append(f"{i+1}")
        pos = 0
    else:                                       # Save
        plan.append(f"{i+1}")
        pos += 1
```

Рисунок 9. Безусловная оптимизация.

Далее были разработаны два теста, для проверки работоспособности программы.

```
class ReplacingEquipmentTests(unittest.TestCase):

    def test1(self):
        n = 10
        y = 3
        s = 4
        P = 18
        r = [31, 30, 28, 28, 27, 26, 26, 25, 24, 24, 23]
        u = [8, 9, 9, 10, 10, 10, 11, 12, 14, 17, 18]
        self.assertEqual((["1", "2", "R", "3", "4", "5", "6", "R", "7", "8", "9", "10"], 169),
                         replacing_equipment(n, y, s, P, r, u))

    def test2(self):
        n = 8
        y = 3
        s = 3
        P = 13
        r = [16, 15, 15, 15, 13, 11, 10, 8, 7]
        u = [4, 6, 6, 6, 7, 8, 8, 10, 12]
        self.assertEqual((["1", "R", "2", "3", "4", "R", "5", "6", "7", "8"], 58),
                         replacing_equipment(n, y, s, P, r, u))
```

Рисунок 10. Тесты.

Результатом выполнения программы является проверка всех тестов и вывод сообщения об успешном или провальном прохождении этих тестов.

Полный кода листинг программы указан в приложении. (Приложение А)

3.2 Задача о рюкзаке

Перед разработкой программной реализации решения задачи о замене оборудования было приведено аналитическое решение в среде MS Excel и там был автоматизирован процесс нахождения матрицы Беллмана в соответствии с его функциональным уравнением.

Сперва рассмотрим решение, которое было получено в среде MS Excel с помощью надстройки «Поиск решения».

Имеется следующий набор исходных данных.

Исходные данные

Максимальный вес	165

Предметы	Вес	Ценность
Предмет 1	23	92
Предмет 2	31	57
Предмет 3	29	49
Предмет 4	44	68
Предмет 5	53	60
Предмет 6	38	43
Предмет 7	63	67
Предмет 8	85	84
Предмет 9	89	87
Предмет 10	82	72

Рисунок 11. Исходные данные.

Для решения данной задачи построим вспомогательную таблицу.

Решение

Предметы	Кол-во	Вес	Ценность
Предмет 1	1	23	92
Предмет 2	1	31	57
Предмет 3	1	29	49
Предмет 4	1	44	68
Предмет 5	0	0	0
Предмет 6	1	38	43
Предмет 7	0	0	0
Предмет 8	0	0	0
Предмет 9	0	0	0
Предмет 10	0	0	0
Итого		165	*309*

Рисунок 12. Таблица решения.

Столбец «Кол-во» является столбцом, в котором будут содержаться изменяемые переменные. Столбец «Вес» является произведением соответствующей ему количеству, на значения веса предмета в исходных данных, «Ценность» будет определяться аналогично. В нижних ячейках указана суммарная ценность, которая будет являться целевой функцией и суммарный вес, который будет являться ограничением.

Исходя из этого мы можем применить «Поиск решения».

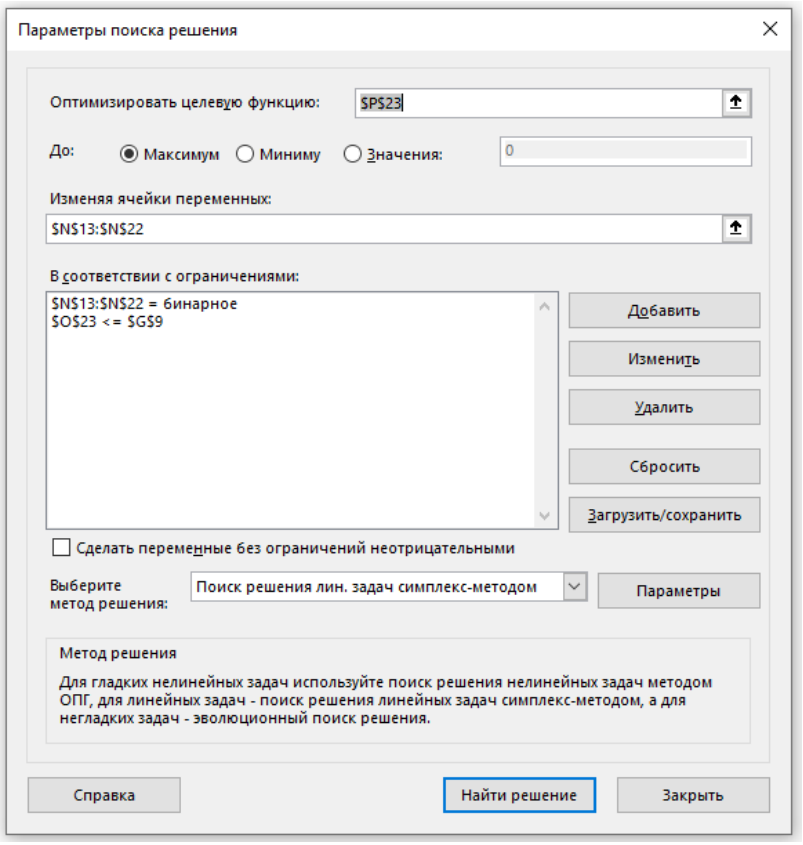

Рисунок 13. Окно «Поиска решения».

После этого можно увидеть получившееся решение и определить какие предметы будут входить в итоговый рюкзак, а какие нет.

Также данная задача была решена на языке программирования Rust, в соответствии с теми формулами, которые были рассмотрены при аналитическом решении задачи.

Для определения максимума между числами была использована функция из стандартной библиотеки max из модуля cmp.

```
use std::cmp::max;
```

Рисунок 14. Модули в реализации задачи о рюкзаке.

Далее в соответствии с формулами условной оптимизации была создана функция $knapsack\_table$, которая принимает на вход: w – вес рюкзака, $weights$ – набор весов соответствующих предметов, $values$ – набор стоимостей предметов. На выходе функция возвращает матрицу функции Беллмана.

```rust
/// knapsack_table(w, weights, values) returns the knapsack table (`n`, `m`) with maximum values, where `n` is number of items
///
/// Arguments:
///     * `w` - knapsack capacity
///     * `weights` - set of weights for each item
///     * `values` - set of values for each item
fn knapsack_table(w: &usize, weights: &[usize], values: &[usize]) -> Vec<Vec<usize>> {
    // Initialize `n` - number of items
    let n: usize = weights.len();
    // Initialize `m`
    // m[i, w] - the maximum value that can be attained with weight less that or equal to `w` using items up to `i`
    let mut m: Vec<Vec<usize>> = vec![vec![0; w + 1]; n + 1];

    for i in 0..=n {
        for j in 0..=*w {
            // m[i, j] compiled according to the following rule:
            if i == 0 || j == 0 {
                m[i][j] = 0;
            } else if weights[i - 1] <= j {
                // If `i` is in the knapsack
                // Then m[i, j] is equal to the maximum value of the knapsack,
                // where the weight `j` is reduced by the weight of the `i-th` item and the set of admissible items plus the value `k`
                m[i][j] = max(values[i - 1] + m[i - 1][j - weights[i - 1]], m[i - 1][j]);
            } else {
                // If the item `i` did not get into the knapsack
                // Then m[i, j] is equal to the maximum cost of a knapsack with the same capacity and a set of admissible items
                m[i][j] = m[i - 1][j]
            }
        }
    }
    m
}
```

Рисунок 15. Функция $knapsack\_table$.

Далее в соответствии с формулами безусловной оптимизации была создана функция $knapsack\_items$, которая принимает на вход: $weights$ – набор весов соответствующих предметов, m – матрица функции Беллмана, i – рассматриваемые предметы (для начального значения $i = N$), j – стоимость рюкзака на соответствующем этапе (для начального значения $j = W$). На выходе функция возвращает список элементов, которые вошли в итоговый рюкзак.

```rust
/// knapsack_items(weights, m, i, j) returns the indices of the items of the optimal knapsack (from 1 to `n`)
///
/// Arguments:
///     * `weights` - set of weights for each item
///     * `m` - knapsack table with maximum values
///     * `i` - include items 1 through `i` in knapsack (for the initial value, use `n`)
///     * `j` - maximum weight of the knapsack
fn knapsack_items(weights: &[usize], m: &[Vec<usize>], i: usize, j: usize) -> Vec<usize> {
    if i == 0 {
        return vec![];
    }
    if m[i][j] > m[i - 1][j] {
        let mut knap: Vec<usize> = knapsack_items(weights, m, i - 1, j - weights[i - 1]);
        knap.push(i);
        knap
    } else {
        knapsack_items(weights, m, i - 1, j)
    }
}
```

Рисунок 16. Функция $knapsack\_items$.

Также была разработана функция $knapsack$, которая объединяет условную и безусловную оптимизацию. Данная функция возвращает кортеж из трёх элементов: первый элемент – стоимость рюкзака, второй элемент – вес рюкзака, третий элемент – набор предметов, вошедших в рюкзак.

```rust
/// knapsack(w, weights, values) returns the tuple where first value is `optimal profit`,
/// second value is `knapsack optimal weight` and the last value is `indices of items`, that we got (from 1 to `n`)
///
/// Arguments:
///     * `w` - knapsack capacity
///     * `weights` - set of weights for each item
///     * `values` - set of values for each item
///
/// Complexity
///     - time complexity: O(nw),
///     - space complexity: O(nw),
///
/// where `n` and `w` are `number of items` and `knapsack capacity`
pub fn knapsack(w: usize, weights: Vec<usize>, values: Vec<usize>) -> (usize, usize, Vec<usize>) {
    // Checks if the number of items in the list of weights is the same as the number of items in the list of values
    assert_eq!(weights.len(), values.len(), "Number of items in the list of weights doesn't match the number of items in the list of values!");
    // Initialize `n` - number of items
    let n: usize = weights.len();
    // Find the knapsack table
    let m: Vec<Vec<usize>> = knapsack_table(&w, &weights, &values);
    // Find the indices of the items
    let items: Vec<usize> = knapsack_items(&weights, &m, n, w);
    // Find the total weight of optimal knapsack
    let mut total_weight: usize = 0;
    for i in items.iter() {
        total_weight += weights[i - 1];
    }
    // Return result
    (m[n][w], total_weight, items)
}
```

Рисунок 17. Функция $knapsack$.

Далее были взяты тесты с бенчмарка, для проверки работоспособности программы.

```rust
#[test]
fn test_p02() {
    assert_eq!(
        (51, 26, vec![2, 3, 4]),
        knapsack(26, vec![12, 7, 11, 8, 9], vec![24, 13, 23, 15, 16])
    );
}

#[test]
fn test_p04() {
    assert_eq!(
        (150, 190, vec![1, 2, 5]),
        knapsack(
            190,
            vec![56, 59, 80, 64, 75, 17],
            vec![50, 50, 64, 46, 50, 5]
        )
    );
}

#[test]
fn test_p01() {
    assert_eq!(
        (309, 165, vec![1, 2, 3, 4, 6]),
        knapsack(
            165,
            vec![23, 31, 29, 44, 53, 38, 63, 85, 89, 82],
            vec![92, 57, 49, 68, 60, 43, 67, 84, 87, 72]
        )
    );
}
```

Рисунок 18. Первый набор тестов.

```rust
#[test]
fn test_p06() {
    assert_eq!(
        (1735, 169, vec![2, 4, 7]),
        knapsack(
            170,
            vec![41, 50, 49, 59, 55, 57, 60],
            vec![442, 525, 511, 593, 546, 564, 617]
        )
    );
}

#[test]
fn test_p07() {
    assert_eq!(
        (1458, 749, vec![1, 3, 5, 7, 8, 9, 14, 15]),
        knapsack(
            750,
            vec![70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110, 113, 115, 118, 120],
            vec![135, 139, 149, 150, 156, 163, 173, 184, 192, 201, 210, 214, 221, 229, 240]
        )
    );
}
```

Рисунок 19. Второй набор тестов.

Результатом выполнения программы является проверка всех тестов и вывод сообщения об успешном или провальном прохождении этих тестов.

Полный кода листинг программы указан в приложении. (Приложение Б)

3.3 Задача о распределении инвестиций

Перед разработкой программной реализации решения задачи о замене оборудования было приведено аналитическое решение в среде MS Excel и там был автоматизирован процесс оптимизации в соответствие с математическими формулами.

Рассмотрим решение, которое было получено в среде MS Excel с помощью надстройки «Поиск решения».

Имеется следующий набор исходных данных.

Исходные данные

Объём капитала	$П_1$	$П_2$	$П_3$	$П_4$
0	0	0	0	0
100	30	50	40	25
200	50	80	50	60
300	90	90	110	100
400	110	150	120	130
500	170	190	180	160
600	180	210	220	200
700	210	220	240	230

Капитал	700

Рисунок 20. Исходные данные.

Для решения данной задачи построим вспомогательную таблицу.

Решение

Объём капитала	$П_1$	$П_2$	$П_3$	$П_4$	
0	0	0	0	0	0
100	0	1	0	0	100
200	0	0	0	0	0
300	0	0	0	0	0
400	0	0	0	0	0
500	0	0	0	0	0
600	0	0	1	0	600
700	0	0	0	0	0
	0	1	1	0	
	<=	<=	<=	<=	
	1	1	1	1	

Ограничение на объём инвестирования		
700	<=	700

Итого	*270*

Рисунок 21. Таблица решения.

Жёлтым цветом в таблице выделены ячейки, которые являются бинарными переменными в данной задаче и показывают будет ли вложение некоторого объёма капитала в данное предприятие. Ниже указано ограничение на то, что мы может только один раз проинвестировать в предприятие. Справа от таблицы с переменными указан столбец, который показывает объём капиталовложения в соответствующее предприятие. Снизу от таблицы указано ограничение на суммарное количество инвестиций, которое не должно превышать объёма капитала. Также под данным ограничением указано значение целевой функции (максимальная прибыль), которое находится как сумма произведения таблицы с переменными задачи и таблицы с исходными данными.

Исходя из этого мы можем применить «Поиск решения».

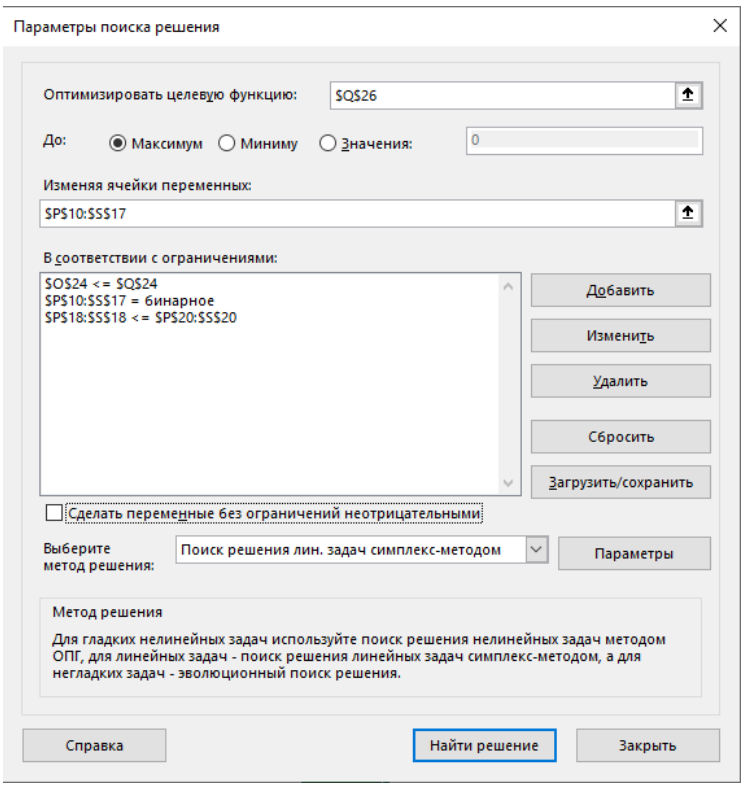

Рисунок 22. Окно «Поиска решения».

После этого можно увидеть получившееся решение и определить сколько стоит инвестировать в какое предприятие.

ЗАКЛЮЧЕНИЕ

Тема курсовой работы посвящена актуальной проблеме реализации алгоритмов решения производственных оптимизационных задач на основе метода динамического программирования.

В ходе выполнения курсовой работы достигнуты следующие результаты:

1) Проанализирован общий подход динамического программирования к решению некоторых типов производственных оптимизационных задач.

2) Приводится аналитическое решение представленных задач с помощью метода динамического программирования.

3) Выполнена реализация алгоритма представленных оптимизационных задач.

Основным результатом выполненной курсовой работы является программная реализация алгоритма решения производственных задач методом динамического программирования.

Результаты работы можно применить для решения практических задач управления ресурсами экономических и производственных систем.

СПИСОК ЛИТЕРАТУРЫ

1. Гладких Б.А. Методы оптимизации и исследования операций. Часть 1. Введение в исследование операций. Линейное программирование. / Гладких Б.А; Томск: НТЛ, 2009. – 200 с.
2. Окулов С.М. Динамическое программирование / Окулов С.М., Пестов О.А.; М.:Бином. Лаборатория знаний, 2017. – 296 с.
3. Лежнёв, А.В. Динамическое программирование в экономических задачах: учеб. пособие / Лежнёв, А.В. – Москва: Бином, 2010. – 176 с.
4. Ширяв В.И. Исследование операций и численные методы оптимизации: учебное пособие / В.И. Ширяв. Москва: Ленанд, 2017. – 224 с.
5. Некрасова М. Г. Методы оптимизации и теория управления: учебное пособие / М. Г. Некрасова. Комсомольск-на-Амуре: КНАГТУ, 2007 – 132 с.
6. Беллман Р. Динамическое программирование – М.: Репринт оригинально издания иностранной литературы, 1960 год, 2012.
7. Колодная Е.М. Математическое программирование / Колодная Е.М. М.:Бином, 2015. – 200 с.
8. Вентцель Е.С. Исследование операций. Задачи, принципы, методологии / Е.С. Вентцель, 1990. – 84 с.
9. Романовская А.М. Динамическое программирование: учебное пособие / А.М. Романовская, М.В. Мендзив – Омск: Издатель Омский институт (филиал) РГТЭУ, 2010. – 58 с.
10. Bertsekas, D.P. Dynamic Programming and Optimal Control (4th ed.), Athena Scientific, 2017. – 10 с.
11. Исследование операций. Курс лекций (слайды). [Электронный ресурс] режим доступа: http://www.math.nsc.ru/LBRT/k5/or_mmf.html свободный (дата обращения: 16.05.2022)
12. Задача о рюкзаке. Университет ИТМО. [Электронный ресурс] режим доступа:https://neerc.ifmo.ru/wiki/index.php?title=%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D1%80%D1%8E%D0%BA%D0%B7%D0%B0%D0%BA%D0%B5 (дата обращения: 19.05.2022)

13. The Rust Programming Language [Электронный ресурс] режим доступа: https://doc.rust-lang.org/book/ свободный (дата обращения: 16.05.2022)

14. Mathsemestr [Электронный ресурс] режим доступа: https://math.semestr.ru/dinam/dinam_manual.php свободный (дата обращения: 20.05.2022)

15. Data for the Bin Packing Problem [Электронный ресурс] режим доступа: https://people.sc.fsu.edu/~jburkardt/datasets/bin_packing/bin_packing.html свободный (дата обращения: 16.05.2022)

ПРИЛОЖЕНИЕ А

Код программы, которая реализует решение задачи о замене оборудования.

```
#! Name:    Solves the problem of replacement of equipment
#! Author:  Egor Bronnikov


# Import
from typing import List, Tuple
import unittest


def replacing_equipment(n: int, year: int, s: int, P: int, r: List[int], u:
List[int]) -> Tuple[List[str], int]:
    """
        @Synopsis
        def replacing_equipment(n: int, year: int, s: int, P: int, r: List[int], u:
List[int]) -> Tuple[List[str], int]: ...

        @Description
        Solves the equipment replacement problem by the dynamic programming method.

        @param n: Number of years of planning
        @type n: int
        @param year: Equipment age
        @type year: int
        @param s: Residual value of equipment
        @type s: int
        @param P: New equipment costs
        @type P: int
        @param r: The value of the products produced during the year
        @type r: List[int]
        @param u: Annual costs associated with the operation of equipment
        @type u: List[int]

        @return: Equipment replacement plan for a given period of time `n` and the
maximum profit (the target function value).
        @rtype: Tuple[List[str], int]
    """

    # Prepare data
    matrix = [[0 for _ in range(n+1)] for _ in range(n)]    # Matrix for Bellman
function
    replacement = [0 for _ in range(n)]                    # Replacement rate

    matrix[n-1] = [max(r[t] - u[t], s - P + r[0] - u[0]) for t in range(n+1)]
    replacement[n-1] = s - P + r[0] - u[0]
```

```
    # Conditional optimization
    # Filling in the Bellman function matrix
    for i in range(n-1, 0, -1):
        for t in range(n):
            matrix[i-1][t] = max(r[t] - u[t] + matrix[i][t+1], s - P + r[0] - u[0]
+ matrix[i][1])
        replacement[i-1] = s - P + r[0] - u[0] + matrix[i][1]
        matrix[i-1][n] = matrix[i-1][n-1]

    # Unconditional optimization
    column = [row[year] for row in matrix]
    value = column[0]

    pos = year
    plan = []

    for i in range(n):
        if pos == matrix[i].index(replacement[i]):  # Save at this time and replace
on the future
            plan.append(f"{i+1}")
            plan.append("R")
            pos = 0
        elif pos > matrix[i].index(replacement[i]): # Replace
            plan.append("R")
            plan.append(f"{i+1}")
            pos = 0
        else:                                       # Save
            plan.append(f"{i+1}")
            pos += 1

    return plan, value


class ReplacingEquipmentTests(unittest.TestCase):

    def test1(self):
        n = 10
        y = 3
        s = 4
        P = 18
        r = [31, 30, 28, 28, 27, 26, 26, 25, 24, 24, 23]
        u = [8, 9, 9, 10, 10, 10, 11, 12, 14, 17, 18]
        self.assertEqual((["1", "2", "R", "3", "4", "5", "6", "R", "7", "8", "9",
"10"], 169),
                         replacing_equipment(n, y, s, P, r, u))

    def test2(self):
        n = 8
        y = 3
        s = 3
```

```
        P = 13
        r = [16, 15, 15, 15, 13, 11, 10, 8, 7]
        u = [4, 6, 6, 6, 7, 8, 8, 10, 12]
        self.assertEqual((["1", "R", "2", "3", "4", "R", "5", "6", "7", "8"], 58),
                         replacing_equipment(n, y, s, P, r, u))


if __name__ == "__main__":
    unittest.main()
```

ПРИЛОЖЕНИЕ Б

Код программы, которая реализует решение задачи о рюкзаке.

```
//! Name:   Solves the knapsack problem
//! Author: Egor Bronnikov
use std::cmp::max;


/// knapsack_table(w, weights, values) returns the knapsack table (`n`, `m`) with
maximum values, where `n` is number of items
///
/// Arguments:
///     * `w` - knapsack capacity
///     * `weights` - set of weights for each item
///     * `values` - set of values for each item
fn knapsack_table(w: &usize, weights: &[usize], values: &[usize]) ->
Vec<Vec<usize>> {
    // Initialize `n` - number of items
    let n: usize = weights.len();
    // Initialize `m`
    // m[i, w] - the maximum value that can be attained with weight less that or
equal to `w` using items up to `i`
    let mut m: Vec<Vec<usize>> = vec![vec![0; w + 1]; n + 1];

    for i in 0..=n {
        for j in 0..=*w {
            // m[i, j] compiled according to the following rule:
            if i == 0 || j == 0 {
                m[i][j] = 0;
            } else if weights[i - 1] <= j {
                // If `i` is in the knapsack
                // Then m[i, j] is equal to the maximum value of the knapsack,
                // where the weight `j` is reduced by the weight of the `i-th` item
and the set of admissible items plus the value `k`
                m[i][j] = max(values[i - 1] + m[i - 1][j - weights[i - 1]], m[i -
1][j]);
            } else {
                // If the item `i` did not get into the knapsack
                // Then m[i, j] is equal to the maximum cost of a knapsack with the
same capacity and a set of admissible items
                m[i][j] = m[i - 1][j]
            }
        }
    }
    m
}


/// knapsack_items(weights, m, i, j) returns the indices of the items of the
optimal knapsack (from 1 to `n`)
```

```rust
///
/// Arguments:
///     * `weights` - set of weights for each item
///     * `m` - knapsack table with maximum values
///     * `i` - include items 1 through `i` in knapsack (for the initial value, use
`n`)
///     * `j` - maximum weight of the knapsack
fn knapsack_items(weights: &[usize], m: &[Vec<usize>], i: usize, j: usize) ->
Vec<usize> {
    if i == 0 {
        return vec![];
    }
    if m[i][j] > m[i - 1][j] {
        let mut knap: Vec<usize> = knapsack_items(weights, m, i - 1, j - weights[i
- 1]);
        knap.push(i);
        knap
    } else {
        knapsack_items(weights, m, i - 1, j)
    }
}


/// knapsack(w, weights, values) returns the tuple where first value is `optimal
profit`,
/// second value is `knapsack optimal weight` and the last value is `indices of
items`, that we got (from 1 to `n`)
///
/// Arguments:
///     * `w` - knapsack capacity
///     * `weights` - set of weights for each item
///     * `values` - set of values for each item
///
/// Complexity
///     - time complexity: O(nw),
///     - space complexity: O(nw),
///
/// where `n` and `w` are `number of items` and `knapsack capacity`
pub fn knapsack(w: usize, weights: Vec<usize>, values: Vec<usize>) -> (usize,
usize, Vec<usize>) {
    // Checks if the number of items in the list of weights is the same as the
number of items in the list of values
    assert_eq!(weights.len(), values.len(), "Number of items in the list of weights
doesn't match the number of items in the list of values!");
    // Initialize `n` - number of items
    let n: usize = weights.len();
    // Find the knapsack table
    let m: Vec<Vec<usize>> = knapsack_table(&w, &weights, &values);
    // Find the indices of the items
    let items: Vec<usize> = knapsack_items(&weights, &m, n, w);
```

```
    // Find the total weight of optimal knapsack
    let mut total_weight: usize = 0;
    for i in items.iter() {
        total_weight += weights[i - 1];
    }
    // Return result
    (m[n][w], total_weight, items)
}


#[cfg(test)]
mod tests {
    // Took test datasets from
https://people.sc.fsu.edu/~jburkardt/datasets/bin_packing/bin_packing.html
    use super::*;

    #[test]
    fn test_p02() {
        assert_eq!(
            (51, 26, vec![2, 3, 4]),
            knapsack(26, vec![12, 7, 11, 8, 9], vec![24, 13, 23, 15, 16])
        );
    }

    #[test]
    fn test_p04() {
        assert_eq!(
            (150, 190, vec![1, 2, 5]),
            knapsack(
                190,
                vec![56, 59, 80, 64, 75, 17],
                vec![50, 50, 64, 46, 50, 5]
            )
        );
    }

    #[test]
    fn test_p01() {
        assert_eq!(
            (309, 165, vec![1, 2, 3, 4, 6]),
            knapsack(
                165,
                vec![23, 31, 29, 44, 53, 38, 63, 85, 89, 82],
                vec![92, 57, 49, 68, 60, 43, 67, 84, 87, 72]
            )
        );
    }

    #[test]
    fn test_p06() {
```

```
        assert_eq!(
            (1735, 169, vec![2, 4, 7]),
            knapsack(
                170,
                vec![41, 50, 49, 59, 55, 57, 60],
                vec![442, 525, 511, 593, 546, 564, 617]
            )
        );
    }

    #[test]
    fn test_p07() {
        assert_eq!(
            (1458, 749, vec![1, 3, 5, 7, 8, 9, 14, 15]),
            knapsack(
                750,
                vec![70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110, 113, 115, 118,
120],
                vec![135, 139, 149, 150, 156, 163, 173, 184, 192, 201, 210, 214,
221, 229, 240]
            )
        );
    }
}
```