Содержание

Аннотация

Инвесторы приобретают ценные бумаги, что дает им право получать часть прибыли от активов (дивиденды). Выплаты дивидендов могут быть реализованы в децентрализованных сетях посредством исполнения смарт контрактов. Данная работа нацелена на разработку алгоритмов эффективной выплаты дивидендов, минимизирующих количество потребляемого газа для различных автоматизированных сценариев. Реализация алгоритмов ведется на языке программирования смарт контрактов Solidity. В данной работе предложено несколько алгоритмов, а также показано преимущество каждого из них для различных сценариев.

Ключевые слова

Блокчейн, смарт контракты, Ethereum, выплаты дивидендов, минимизация газа, Solidity

1 Введение

1.1 Описание предметной области

Термин блокчейн впервые появился как название распределенной базы данных транзакций, используемый и реализованный для криптовалюты Bitcoin [11, 4]. Однако область применения данной технологии оказалась гораздо шире – сейчас блокчейн используется для обеспечения прозрачности в биомедицине [9], биржах, цепочках поставок [15] и т.д. Особую роль играет блокчейн в финансовом секторе, например, для реализации смарт контрактов.

Смарт контракты являются компьютерными программами, исполняемыми в децентрализованных сетях, например на Ethereum-подобных платформах [3]. Разработка смарт контрактов ведется на языках программирования, код которых транслируется в байт-код виртуальной машины Ethereum, например: Solidity [6]. Для обеспечения надежности исполнение независимо дублируется участниками, а поэтому является дорогостоящим. Так, по состоянию на ноябрь 2022, участники сети Ethereum ежедневно оплачивают 500 000 USD комиссии за свои транзакции, а стоимость развертывания новых смарт контрактов может достигать 5000 USD. Поэтому крайне важно при реализации смарт контрактов минимизировать стоимость их развертывания и исполнения [1, 2]. Эта стоимость, в случае с виртуальной машиной Ethereum, традиционно измеряется в «газе».

Газ — условная единица измерения вычислительного времени работы, необходимого для выполнения определенных операций в сети Ethereum. Минимизация газа является важнейшей задачей, т.к. напрямую влияет на стоимость развертывания и исполнения смарт контракта.

Децентрализованные финансы (DeFi) широко представлены в блокчейн экосистеме. В частности, они предлагают ряд механизмов инвестирования, и выплата дивидендов — их неотъемлемая часть. В рамках проекта предлагается построить смарт контракт для выплаты дивидендов, обладающий минимальной стоимостью для фиксированных тестовых сценариев.

1.2 Постановка задачи

Целью курсовой работы является разработка и реализация алгоритмов, которые позволяли бы решать задачу выплаты дивидендов эффективно. Заранее известна матрица выплат, столбцы которой отвечают за ценные бумаги, строки за финансовые активы, а значения соответствующих ячеек за количество актива в ценной бумаге. После произвольной обработки заданной матрицы выплат, а также ее загружения в смарт контракт, необходимо уметь,

принимая на вход массив стоимостей соответствующих активов, вычислять количество денег, которое положено каждому инвестору в соответствии с заранее известной матрицей выплат.

Говоря более формальным языком, постановку задачи выплат дивидендов можно описать следующим образом:

- Есть матрица $A \in \mathbb{R}^{m \times n}$, где m — количество различных финансовых активов, а n — число ценных бумаг. Значение ячейки $A_{i,j}$ содержит в себе количество i-го актива в j-ой ценной бумаге.
- Имеется вектор-столбец $C \in \mathbb{R}^m$ стоимостей финансовых активов $C = (C_1, \ldots, C_m)^T$, где C_i — стоимость i-го актива.
- Есть вектор $I \in \mathbb{R}^n$ соответствий ценных бумаг и инвесторов, т.е. I_i — номер инвестора, вложившийся i-ую в ценную бумагу.

Требуется произвести выплату дивидендов. А именно, инвестор с номером i должен получить на свой счет сумму денег, соответствующую всем ценным бумагам, которые он приобрел. Если обозначить через $\text{cost}[s]$ — «прибыль» s-ой ценной бумаги, а через $\text{balances}[i]$ — прибыль i-го инвестора, то:

$$\text{balances}[i] = \sum_{s=1}^{n} \text{cost}[s] \cdot [i = I_s], \text{ где } [a = b] = \begin{cases} 1 & \text{если } a = b \\ 0 & \text{если } a \neq b \end{cases}$$

$$\text{cost}[s] = \sum_{i=1}^{m} C_i A_{i,s} \Longleftrightarrow \text{cost} = C^T A = (A^T C)^T \tag{1}$$

Целью является вычисления массива $\text{balances}[i]$ алгоритмически по входным данным A, C, I таким образом, чтобы количество потребляемого газа в смарт контракте было минимально возможным.

1.3 Структура работы

Работа организована следующим образом. В разделе 1 дано введение в предметную область и формальная постановка задачи. Обзор литературы проведен в разделе 2. Подробное описание методов решения исходной задачи, псевдокод, а также их теоретический анализ приведены в разделе 3. Постановка и результаты экспериментов подробно описаны в разделе 4. Выводы и результаты работы разъяснены в разделе 5.

2 Обзор литературы

В поставленной задаче принципиально важным является не минимизация асимптотического времени работы алгоритмов при фиксированных автоматизированных сценариях, а именно минимизация количества потребляемого газа в итоговом смарт контракте. Количество потребляемого газа зависит не только от числа выполненных операций, но и таких факторов, как используемые типы данных, способы работы с памятью, а также виды выполняемых операций. О соответствии количества газа и видов операций (арифметических, а также работы с памятью) более подробно описано в [12].

В книге [1] подробно описаны основные концепции и способы анализа количества потребляемого газа, в частности, с помощью специального инструмента газового анализа GASOL, позволяющему не только определять потребление газа, связанного с определенными типами инструкций на EVM, но также выявлять в программном коде недостаточно оптимизированные режимы хранения данных. Описаны также применения инструмента GASOL для оптимизации количества потребляемого газа в смарт контрактах для языка Solidity.

В работе [10] предложен набор шаблонов проектирования смарт контрактов на языке Solidity, которые позволяют достичь минимального потребления газа. Предлагаемые шаблоны разделены на 5 категорий, в зависимости от специфики поставленной задачи.

3 Описание метода

В данном разделе приведено описание автоматизированных сценариев, методов предобработки исходной матрицы выплат, описание и анализ алгоритмов для выплаты дивидендов, а также способы генерации данных для каждого сценария. Все используемые обозначения можно найти в подразделе 1.2.

3.1 Описание автоматизированных сценариев

В реальности, зачастую, поступаемые на вход данные не являются произвольными, а обладают некоторыми характерными особенностями или закономерностями. Именно они и стали мотивацией для рассмотрения задачи выплат дивидендов в условиях различных автоматизированных сценариев. Опишем их постановку:

1 **Случай плотной матрицы.** В этом сценарии предполагается, что на матрицу A не накладывается никакой дополнительной структуры, поэтому в этом случае можно считать, что мы имеем дело с произвольной матрицей размера $m \times n$.

2 **Случай разреженной матрицы**. В этом сценарии, напротив, мы считаем матрицу A сильно разреженной, то есть в ней много нулевых значений. Этот сценарий хорошо согласуется с представлениями о реальных данных — ведь далеко не во все активы инвесторы вкладываются.

3 **Случай матрицы с повторяющимися столбцами.** В этом сценарии мы предполагаем, что у матрицы есть много одинаковых столбов. Более формально, будем считать, что есть некоторое фиксированное множество векторов-столбцов $c_1, \ldots, c_k$, $c_i \in \mathbb{R}^m$, где $k \ll n$, а матрица A имеет вид $A = \left(c_{i_1} \middle| c_{i_2} \middle| \ldots \middle| c_{i_n}\right)$ для некоторых $1 \leq i_j \leq k$. Данный сценарий также хорошо согласуется с тем, что происходит в реальных данных — действительно, зачастую в матрице выплат встречается большое количество одинаковых пакетов (соответствующих столбцам матрицы выплат).

4 **Случай малоранговой матрицы.** В этом сценарии мы предполагаем, что матрица A имеет сравнительно небольшой ранг по сравнению со своими размерами (параметрами m и n) или же она «близка» к матрице малого ранга. Более формально, $\operatorname{rank} A = r \ll n$ или же $\exists B \in \mathbb{R}^{m \times n} : \operatorname{rank} B = r$ и $|A - B| < \epsilon_{\text{machine}}$. Этот сценарий является естественным обобщением сценария 3.

3.2 Алгоритмы выплат для разных сценариев

В рамках выполнения работы было исследовано и реализовано [14] 4 различных алгоритма выплат дивидендов. Ниже, в этом разделе приведено подробное описание этих алгоритмов. Все необходимые обозначения введены в подразделе 1.2.

Любой алгоритм, реализующий выплату дивидендов можно разделить на две независимые части. Первая часть — вычисление массива cost, где $\text{cost}[s]$ — суммарная прибыль, соответствующая s-ой ценной бумаге. Вторая часть — осуществление перевода инвесторам, что можно описать как вычисление массива balances по формуле из подраздела 1.2, где $\text{balances}[i]$ — прибыль i-го инвестора. Как правило, вторая часть алгоритма не представляет особого исследовательского интереса и реализуется одинаково во всех алгоритмах и сценариях, поэтому в дальнейшем содержательно будет описана только первая часть.

1 Алгоритм **payoutNaive**.

В данном алгоритме на вход поступают матрица A в исходном виде без какой-либо предобработки, а также массив C стоимостей финансовых активов. Данный алгоритм

реализует подсчет массива cost наивным алгоритмом по формуле:

$$\text{cost}[s] = \sum_{i=1}^{m} C_i A_{i,s}$$

Асимптотическое время работы алгоритма: $O(mn)$

2 Алгоритм **payoutSparse**.

В данном алгоритме необходима предварительная обработка для компактной записи матрицы. Воспользуемся стандартным способом хранения разреженных матриц — форматом COO [13], который хранит разреженную матрицу с помощью трех массивов: rows (номера строк ненулевых элементов), columns (номера столбцов ненулевых элементов) и values (значения ненулевых элементов). В таком формате можно эффективно производить подсчет массива cost итерируясь только по ненулевым ячейкам.

Сложность алгоритма: $O(n_{\text{nz}} + n + m)$, где n_{nz} — количество ненулевых элементов.

3 Алгоритм **payoutRepeatedColumns**.

В случае, когда матрица A содержит большое количество одинаковых столбцов, то имеет смысл вместо хранения матрицы на блокчейне «целиком», хранить только уникальные столбцы $(c_1, \ldots, c_k)$ и их номера в матрице $i_1, \ldots, i_n$ (см. обозначения в 3.1). В этом случае можно вычислить прибыль, соответствующую каждому из уникальных столбцов $c_1, \ldots, c_k$ за время $O(mk)$, а после линейным проходом за время $O(n)$ присвоить $\text{cost}[s]$ стоимость соответствующего столбца i_s.

Время работы алгоритма: $O(mk + n)$, где k — число различных столбцов.

4 Алгоритм **payoutLowRank**

В данном случае предполагается, что матрица A имеет малый ранг, т.е. $\operatorname{rank} A = r$, где $r \ll n$. Из теоремы о существовании скелетного разложение матрицы [8] следует, что $\exists U \in \mathbb{R}^{m\times r}, V \in \mathbb{R}^{n\times r}$, такие что $A = UV^T$. Так как вычисление cost, в силу равенства 1, эквивалентно вычислению $C^T A = C^T U V^T$, то можно производить умножения соответствующих матриц в следующем порядке: $(C^T U)V^T$. Умножение C^T на U выполняется за $O(mr)$, а умножение $(C^T U)$ на V^T выполняется за $O(nr)$, тем самым итоговая сложность получается $O((n+m)r)$. Для осуществления данного алгоритма, необходимо иметь на вход матрицы U и V из скелетного разложения. Способ их получения подробно описан в подразделе 3.3, посвященном предобработке данных.

Время работы алгоритма: $O((n+m)r)$, где $r = \operatorname{rank} A$.

3.3 Предобработка входных данных

Перед отправкой матрицы выплат A в блокчейн, мы можем совершить некоторую предобработку исходной матрицы, чтобы получить ее более компактное или удобное представление и использовать это представление для алгоритмов выплат внутри смарт контракта, при меняющихся стоимостях финансовых активов. Именно благодаря этой предобработке и удается добиться оптимизации объема потребляемой памяти и количества затраченного газа. Рассмотрим, каким образом устроены алгоритмы предобработки для различных алгоритмов.

1 Алгоритм **payoutNaive**. В наивном алгоритме не требуется какое-либо особенное представление данных, поэтому матрица A поступает на вход функции payuotNaive в исходном виде.

2 Алгоритм **payoutSparse**. В этом алгоритме нужно представить матрицу A в формате, характерным для формата COO [13], то есть нужно по исходной матрице получить три массива:

- rows — массив номеров строк для каждого ненулевого элемента.
- columns — массив номеров столбцов для каждого ненулевого элемента
- values — массив значений для каждого ненулевого элемента.

Таким образом, что A[rows[k], columns[k]] = values[k] $\forall k = 1 \dots n_{\text{nz}}(A)$, где $n_{\text{nz}}(A)$ — количество ненулевых элементов матрицы A. Получить массивы rows, columns, values можно с помощью полного обхода матрицы за $O(mn)$.

3 Алгоритм **payoutRepeatedColumns**. В этом случае нам нужно сформировать множество уникальных столбцов $(c_1, \dots, c_k)$, а также восстановить набор индексов $(i_1, \dots, i_n)$, т.ч. j-й столбец матрицы A совпадает с c_{i_j}. Наивным алгоритмом, это можно сделать за время $O(mnk)$: проходимся по всем n столбцам, для каждого за $O(mk)$ проверяем, не было ли такого же столбца ранее, и если не было, то добавляем текущий столбец в множество c. Индексы $i_1, \dots, i_k$ восстанавливаются тривиально.

4 Алгоритм **payoutLowRank**. Здесь нужно построить «скелетное» разложение матрицы A ранга $\operatorname{rank} A = r$, т.е. найти две такие матрицы $U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{n \times r}$, т.ч. $A = UV^T$.

Для этого предлагается использовать сингулярное разложение [7] матрицы A, то есть разложение матрицы A в виде $A = U\Sigma V^T$, где матрицы $U \in \mathbb{R}^{m\times m}, V \in \mathbb{R}^{n\times n}$ являются унитарными, а Σ — диагональная матрица, имеющая следующий вид:

$$\Sigma = \overbrace{\begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & \ddots & \\ & & & & 0 \end{pmatrix}}^{n}$$

где $\sigma_1 \geq \ldots \geq \sigma_r > 0$. Обозначим через U_r, V_r — матрицы, состоящие из первых r столбцов матрицы U и V соответственно, а Σ_r — матрицу, состоящую из первых r строк и столбцов матрицы Σ. Тогда в качестве скелетного разложения можно взять матрицы $(U_r\Sigma_r)$ и V_r^T.

3.4 Генерация данных для автоматизированных сценариев

Для того, чтобы можно было проводить эксперименты и тестировать различные компоненты смарт контракта (а именно функции выплат, для автоматизированных сценариев) необходимо иметь параметризованный генератор данных для соответствующих сценариев. Опишем способ их работы в каждом из случаев:

1 **Случай плотной матрицы.**

В данном случае генератор устроен крайне просто – необходимо заполнить матрицу размера $m \times n$ произвольными числами. Это можно сделать с помощью стандартного генератора случайных чисел. Важно отметить, что с вероятностью, близкой к 1 матрица, полученная таким образом, не будет иметь дополнительной структуры, такой как малоранговость или разреженность.

2 **Случай разреженной матрицы**.

В данном случае генерация состоит из двух этапов: генерация ненулевых ячеек матрицы и заполнения этих ячеек случайными значениями. Первый этап делается следующим образом: фиксируется параметр density (данный на вход генератору), а затем выбирается $k = \text{density} \cdot mn$ ненулевых ячеек произвольным образом. На втором этапе

эти ячейки заполняются случайными ненулевыми значениями, с помощью генератора случайных чисел.

3 **Случай матрицы с повторяющимися столбцами.**

В данном случае, в первую очередь, фиксируется параметр density — процент уникальных столбцов среди всех. Отсюда, для каждого m, тривиальным образом выражается число уникальных столбцов $k = \text{density} \cdot m$. Следующим шагом является генерация k уникальных столбцов, что делается с помощью генератора случайных чисел (уникальность будет выполнена с вероятностью, близкой к 1). Далее, нужно произвольным образом заполнить матрицу столбцами, для этого можно для каждого $1 \leq i \leq m$ независимо выбирать номер уникального столбца.

4 **Случай малоранговой матрицы.**

Первым шагом нужно сгенерировать две плотные матрицы $U \in \mathbb{R}^{m \times r}$ и $V \in \mathbb{R}^{n \times r}$ способов, описанным для генерации плотных матриц. Так как с вероятностью, близкой к 1, матрицы U и V будут иметь полный ранг, в качестве матрицы исходной A можно положить $A = UV^T$.

4 Эксперименты

4.1 Архитектура эксперимента

После реализации алгоритмов для выплаты дивидендов на языке Solidity, была проведена серия экспериментов, для того чтобы исследовать эффективность реализованных алгоритмов на практике.

Развертывание, исполнение и тестирование смарт контракта было произведено с помощью локальной сети Ethereum, развернутой на моем персональном компьютере с использованием инструмента HardHat [5]. Логика смарт контракта была реализована в файле "contracts/Payout.sol"и представляет собой смарт контракт с описанными в разделе 3 алгоритмами, обернутыми в функции.

Каждый эксперимент проводился по следующей схеме: генерация данных → предобработка данных → запуск на блокчейне (с учетом предобработки) → результат.

4.2 Замер производительности и результаты

Замер производительности (количества потребляемого газа) осуществлялся посредством запуска смарт контракта в локальной сети HardHat Network, предназначенной для тестирования смарт контрактов.

Посмотрим на результаты экспериментов на тех случаях, где матрицы выплат являются квадратными размера $n \times n$, для различных автоматизированных сценариев:

1 **Случай плотной матрицы**

На графике 4.1 видно, что в данном случае выигрывает наивный алгоритм **payoutNaive**, который хранит матрицу выплат полностью и выполняет вычисление массива cost прямолинейным образом. Другие алгоритмы не дают выигрыша, т.к. в случае с плотной матрицей предобработка данных не дает никакого преимущества из-за отсутствия необходимой структуры.

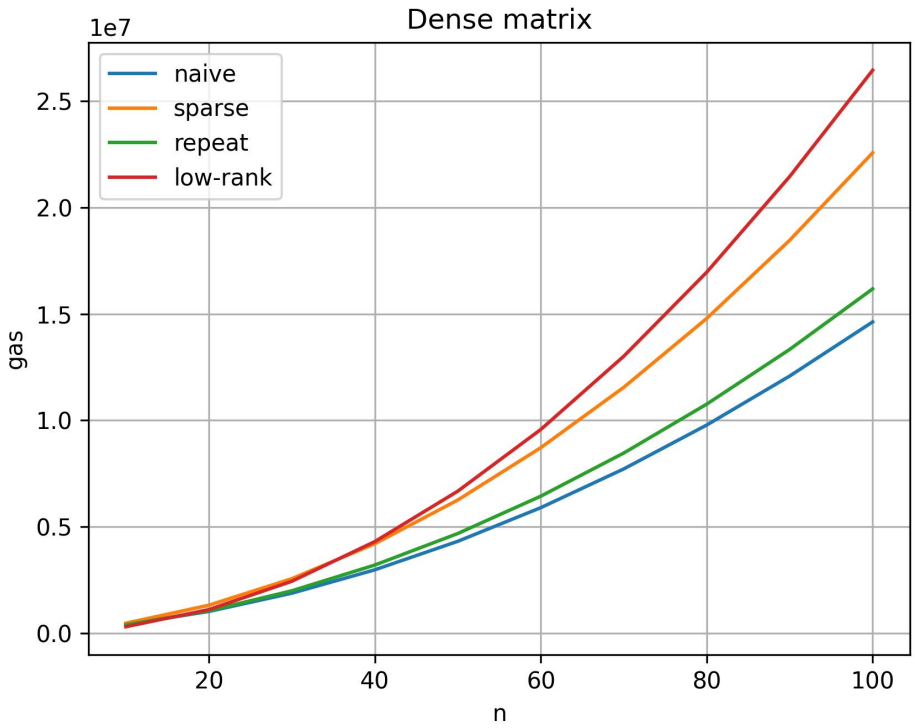

Рис. 4.1: График для плотной матрицы

2 **Случай разреженной матрицы**

График 4.2 показывает, что в случае, когда матрица выплат A является разреженной, то явным лидером является алгоритм **payoutSparse**, т.к. разреженная структура дает ему значительное преимущество по времени и памяти.

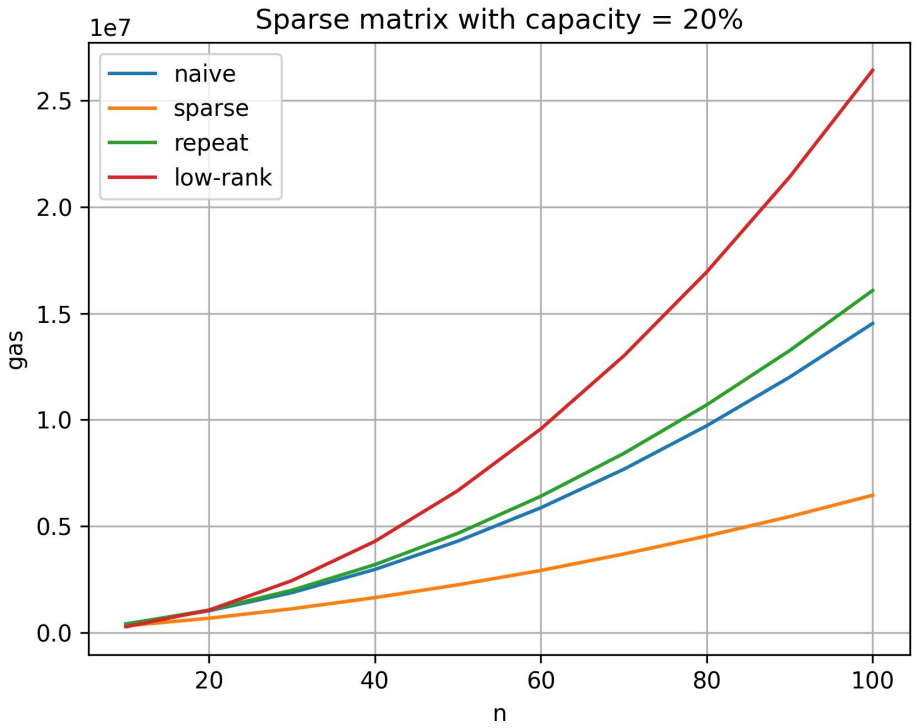

Рис. 4.2: График для разреженной матрицы

3 **Случай матрицы с повторяющимися столбцами**

В данном случае мы видим, что лидерами по количеству затрачиваемого газа являются **payoutRepeatedColumns** и **payoutLowRank** (Рис. 4.3). Ожидаемо, что алгоритм **payoutLowRank** значительно превзошел алгоритмы **payoutSparse** и **payoutNaive**, т.к. случай матрицы с повторяющимися столбцами — это частный случай малоранговой матрицы.

4 **Случай матрицы малого ранга**

В случае, когда исходная матрица A имеет малый ранг, то, исходя из Рис. 4.4, лучше всего себя ведет алгоритм **payoutLowRank**, в то время **payoutSparse** дает худший результат из представленных.

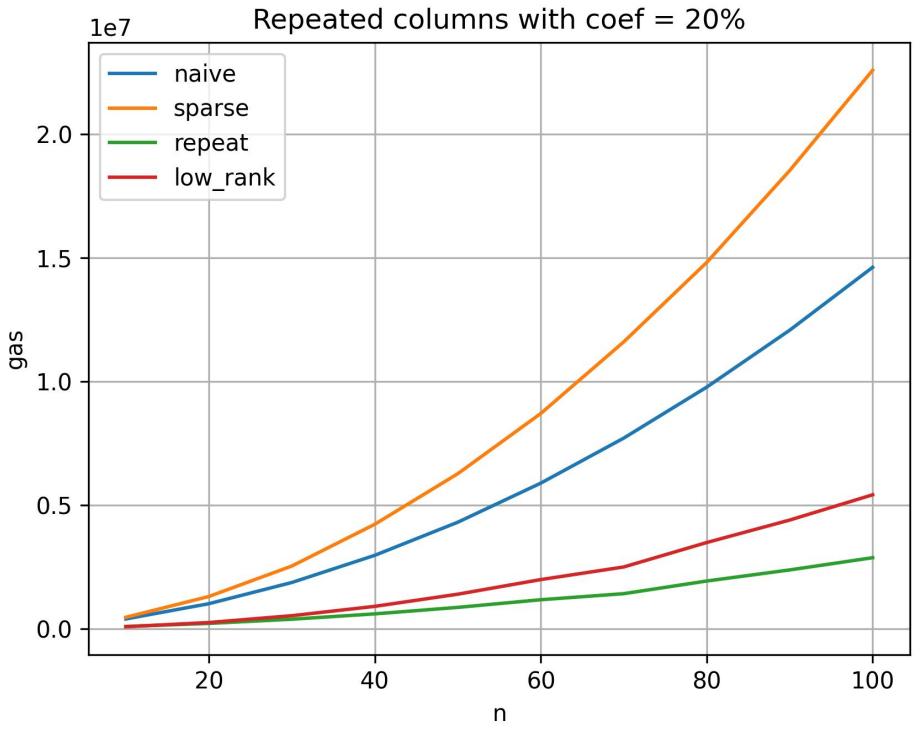

Рис. 4.3: График для матрицы с повторяющимися столбцами

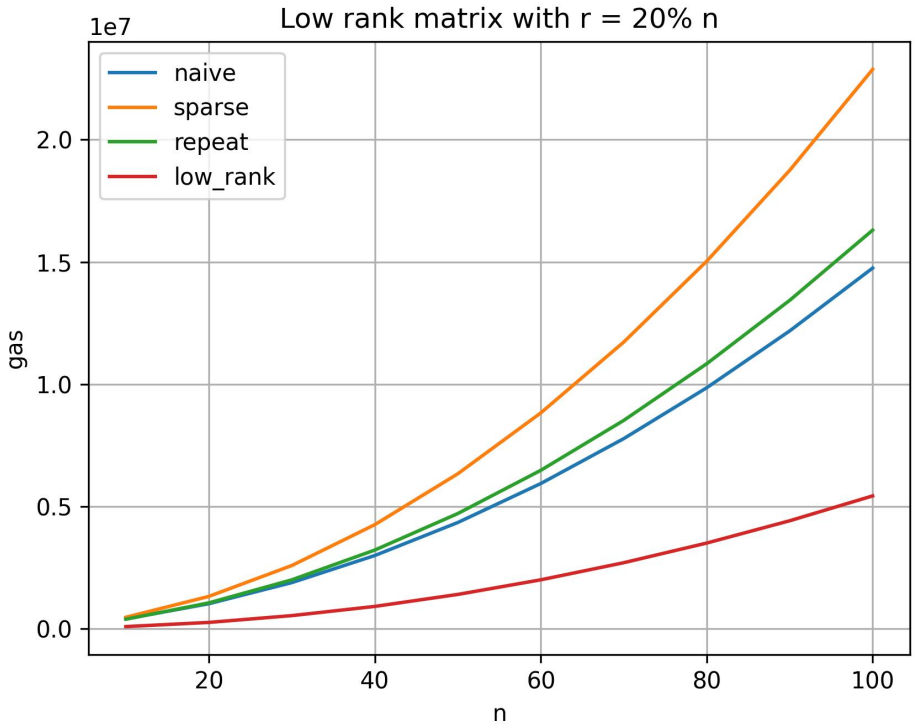

Рис. 4.4: График для матрицы малого ранга

5 Выводы и результаты

В рамках курсовой работы было реализовано 4 алгоритма для выплаты дивидендов, которые были протестированы в рамках различных автоматизированных сценариев. На каждом из сценариев, в котором предполагалась нетривиальная структура матрицы, хотя бы одному из алгоритмов удалось добиться значительного преимущества по сравнению с наивным подходом. В связи с чем цели курсовой работы были достигнуты.

Список литературы

[1] E. Albert, J. Correas, P. Gordillo, G. Roman-Dıez и A. Rubio. *GASOL: gas analysis and optimization for ethereum smart contracts, in: Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II, 2020, pp. 118–125.*

[2] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio и Ilya Sergey. "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode". В: *CoRR* abs/1805.07208 (2018). arXiv: 1805.07208. URL: http://arxiv.org/abs/1805.07208.

[3] Vitalik Buterin. "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform". В: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[4] Vitalik Buterin. "On public and private blockchains". В: *Ethereum blog* 7.1 (2015). URL: https://blog.ethereum.org/2015/08/07/on-public-and-privateblockchains/.

[5] *Ethereum: HardHat*. URL: https://hardhat.org/docs.

[6] *Ethereum: Solidity (2018)*. URL: https://solidity.readthedocs.ios.

[7] V. Klema и A. Laub. "The singular value decomposition: Its computation and some applications". В: *IEEE Transactions on Automatic Control* 25.2 (1980), с. 164—176. DOI: 10.1109/TAC.1980.1102314.

[8] Donald E. Knuth. *The Art of Computer Programming*. Four volumes. Seven volumes planned. Addison-Wesley, 1968.

[9] Polina Mamoshina, Lucy Ojomoko, Yury Yanovich, Alexei Ostrovski, Alexandru Botezatu, Pavel V. Prikhodko, Eugene Izumchenko, Alexander Aliper, Konstantin Romantsov, Alexander Zhebrak, Iraneus Obioma Ogu и Alex Zhavoronkov. "Converging blockchain and next-generation artificial intelligence technologies to decentralize and accelerate biomedical research and healthcare". В: *Oncotarget* 9 (2015), с. 5665—5690.

[10] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino и Danilo Tigano. "Design Patterns for Gas Optimization in Ethereum". В: (2020), с. 9—15. DOI: 10.1109/IWBOSE50093.2020.9050163.

[11] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. B: *Cryptography Mailing list at https://metzdowd.com* (март 2009). URL: https://bitcoin.org/bitcoin.pdf.

[12] *Opcodes for the EVM*. URL: https://ethereum.org/en/developers/docs/evm/opcodes/.

[13] *Scipy Documentation, coo matrix*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html.

[14] Maksim Shuklin. *Source repository for coursework*. URL: https://github.com/maximshuklin/coursework.

[15] Y. Yanovich, I. Shiyanov, T. Myaldzin, I. Prokhorov, D. Korepanova и S. Vorobyov. *“Blockchain-Based Supply Chain for Postage Stamps”, Informatics, vol. 5, No. 4, p. 42, 11 2018.*