Министерство науки и высшего образования РФ
ФГБОУ ВО «Тверской государственный университет»
Математический факультет
Направление 02.04.01 Математика и компьютерные науки
Профиль «Математическое и компьютерное моделирование»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

Вариационный квантовый алгоритм с оптимизацией методом отжига

Автор:
Алешин Дмитрий Алексеевич
Подпись:

Научный руководитель:
д. ф.-м. н. Цирулёв Александр Николаевич
Подпись:

Допущен к защите:
Руководитель ООП: Цветков В.П.

_____
(подпись, дата)

Тверь 2025

Оглавление

Введение

В последние годы вариационные квантовые алгоритмы приобретают всё большее значение в современных исследованиях по математическому моделированию и квантовым вычислениям. Особый интерес представляют гибридные квантово-классические методы, в которых оптимизация параметров квантовых схем сочетается с классическими алгоритмами поиска минимума. Такие подходы позволяют эффективно моделировать и исследовать малоразмерные квантовые системы, актуальные для описания новых состояний вещества, включая топологические материалы, а также задач машинного обучения [1, 2, 3, 4].

Вариационными квантовыми алгоритмами называют семейство гибридных квантово-классических алгоритмов для решения квантовых задач оптимизации посредством квантовых вычислений или их классической имитации. Параметрически управляемое квантовое устройство, обычно представленное квантовой цепью, реализует унитарное преобразование стандартного начального состояния $|0\rangle^{\otimes n}$ или, как вариант, предудущего полученного состояния. На каждом шаге регулирующие параметры подбираются так, чтобы минимизировать целевую функцию. Обычно это выполняется путём измерения энергии состояний, предоставляемых вариационной схемой, и обновления параметров для минимизации целевой функции [5, 6, 7].

Важной прикладной задачей в области вариационных квантовых алгоритмов является поиск основного состояния гамильтониана, что эквивалентно задаче минимизации средней энергии в пространстве квантовых состояний, генерируемых параметризованным анзацем. Сложность алгоритмов прямого вычисления собственных значений гамильтониана растет экспоненциально с ростом числа кубитов, поэтому для больших систем используются квантовые вариационные методы решения зада-

чи оптимизации [8, 9]. На практике решение такой задачи требует эффективных классических методов оптимизации, устойчивых к наличию большого числа локальных минимумов и не требующих вычисления производных целевой функции. Среди таких методов особое место занимает метод имитации отжига [10, 11], который широко используется в задачах глобальной оптимизации и хорошо адаптируется к вариационным квантовым схемам.

Целью работы является построение вариационного квантового алгоритма с оптимизацией методом имитации отжига для поиска основного состояния гамильтониана квантовой системы и исследование с использованием имитационного моделирования на классическом компьютере.

В рамках исследования были поставлены следующие задачи:

1. Изучить теоретические основы вариационных квантовых алгоритмов, включая формализм базиса Паули, и построение анзаца для многокубитных систем;
2. Разработать и реализовать алгоритм вариационной квантовой оптимизации с использованием метода имитации отжига;
3. Реализовать алгоритм в виде программного модуля на языке Python для практического моделирования процесса вариационной оптимизации;
4. Провести сравнительный анализ эффективности различных параметров отжига и схем построения анзаца.

Объектом исследования является задача минимизации энергии гамильтониана в пространстве состояний, порождённых вариационным анзацем в базисе Паули. Предметом исследования выступает разработка и программная реализация классической компьютерной модели гибридного вариационного квантового алгоритма с классическим оптимизатором по методу имитации отжига.

Диссертационная работа включает две главы, заключение, список литературы и приложение с листингом программы на языке Python для основного алгоритма. В первой главе изложены теоретические основы вариационных квантовых алгоритмов, формализм базиса Паули и вопросы построения параметризованного анзаца. Вторая глава посвящена

описанию метода имитации отжига, его адаптации к задачам квантовой оптимизации и программной реализации алгоритма, а также анализу результатов тестирования. В заключении приведены основные выводы по проделанной работе. Список литературы включает современные публикации российских и зарубежных авторов, посвящённые тематике квантовых вычислений и оптимизации.

В работе везде используется система единиц, в которой постоянная Планка и скорость света равны единице, т.е. $\hbar = 1$ и $c = 1$, что соответствует общепринятому подходу в математическом моделировании и вычислительной физике.

Глава 1

Общая схема квантовых вариационных алгоритмов

В квантовой информатике, которая имеет дело с конечномерными гильбертовыми пространствами, применение базиса Паули является общим и эффективным вычислительным подходом [12, 13]

1.1 Базис Паули

1.1.1 Связь стандартного базиса и базиса Паули

Рассмотрим квантовую систему из n кубитов, где каждый кубит связан с двумерным гильбертовым пространством $\mathcal{H}$ и его эрмитово сопряжённым пространством $\mathcal{H}^\dagger$. Обозначим через $\mathcal{H}_n = \mathcal{H}^{\otimes n}$ и $\mathcal{H}_n^\dagger = (\mathcal{H}^\dagger)^{\otimes n}$ гильбертово пространство системы и его эрмитово сопряжение соответственно. Пространство линейных операторов, действующих на $\mathcal{H}$ и $\mathcal{H}^\dagger$ левым и правым умножением, задаётся как $L(\mathcal{H}_n) = \mathcal{H}_n \otimes \mathcal{H}_n^\dagger$. Тогда

$$\dim_{\mathbb{C}} \mathcal{H}_n = \dim_{\mathbb{C}} \mathcal{H}_n^\dagger = 2^n, \quad \dim_{\mathbb{C}} L(\mathcal{H}_n) = 2^{2n}.$$

Пространство $L(\mathcal{H}_n)$ наделено скалярным произведением Гильберта-Шмидта:

$$\langle \hat{A}, \hat{B} \rangle = \mathrm{tr}(\hat{A}^\dagger \hat{B}), \quad \hat{A}, \hat{B} \in L(\mathcal{H}_n), \tag{1.1}$$

которое естественно продолжает скалярное произведение в $\mathcal{H}_n$. Вещественное линейное пространство эрмитовых операторов далее обозначим как $H(\mathcal{H}_n)$.

Пусть $\{|0\rangle, |1\rangle\}$ образуют ортонормированный базис в однокубитном пространстве $\mathcal{H}$. Единичная матрица и матрицы Паули задаются как:

$$\sigma_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},\ \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix},\ \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

а соответствующие операторы Паули представляются в виде:

$$\hat{\sigma}_0 = |0\rangle\langle 0| + |1\rangle\langle 1|,\ \hat{\sigma}_1 = |0\rangle\langle 1| + |1\rangle\langle 0|,$$
$$\hat{\sigma}_2 = -i|0\rangle\langle 1| + i|1\rangle\langle 0|,\ \hat{\sigma}_3 = |0\rangle\langle 0| - |1\rangle\langle 1|.$$

Эти операторы одновременно эрмитовы и унитарны, а также образуют базис в $\mathcal{H}$. Обратное преобразование выражается следующим образом:

$$|0\rangle\langle 0| = \frac{\hat{\sigma}_0 + \hat{\sigma}_3}{2},\ |0\rangle\langle 1| = \frac{\hat{\sigma}_1 + i\hat{\sigma}_2}{2},\ |1\rangle\langle 0| = \frac{\hat{\sigma}_1 - i\hat{\sigma}_2}{2},\ |1\rangle\langle 1| = \frac{\hat{\sigma}_0 - \hat{\sigma}_3}{2}.$$

Для $k, l, m \in \{1, 2, 3\}$ выполняются свойства: $\mathrm{tr}\hat{\sigma}_k = 0$, $\hat{\sigma}_k^2 = \hat{\sigma}_0$, а также

$$\hat{\sigma}_k\hat{\sigma}_l = -\hat{\sigma}_l\hat{\sigma}_k, \quad \hat{\sigma}_k\hat{\sigma}_l = i\,\mathrm{sign}(\pi)\hat{\sigma}_m, \quad (klm) = \pi(123), \tag{1.2}$$

где $\pi(123)$ — произвольная перестановка множества $\{1, 2, 3\}$.

Рассмотрим стандартный[1] бинарный базис в $\mathcal{H}_n$, образованный ортонормированными базисами $\{|0\rangle, |1\rangle\}$ в однокубитных пространствах. Позиция в тензорном произведении позволяет различать кубиты. Для фиксированного n элементы этого базиса и соответствующие им элементы двумерного базиса удобно записывать как:

$$|k\rangle = |k_1 \dots k_n\rangle = |k_1\rangle \otimes \dots \otimes |k_n\rangle, \quad \langle k| = \langle k_1 \dots k_n| = \langle k_1| \otimes \dots \otimes \langle k_n|,$$

[1]Мы избегаем термина «вычислительный», так как он может приводить к неоднозначности. И базис Паули, и стандартный базис являются вычислительными в одинаковом контексте.

где строки $k_1 \dots k_n$ $(k_1, \dots, k_n \in \{0, 1\})$ интерпретируются как двоичные числа с десятичным представлением k. Например, $|101\rangle = |5\rangle$ и $|00110\rangle = |6\rangle$.

В стандартном базисе:

$$|u\rangle = \sum_{k=0}^{2^n-1} u_k |k\rangle, \quad \hat{A} = \sum_{k,l=0}^{2^n-1} a_{kl} |k\rangle\langle l|,$$

где $|u\rangle \in \mathcal{H}_n$ и $\hat{A} \in L(\mathcal{H}_n)$.

Базис Паули $P(\mathcal{H}_n)$ в $L(\mathcal{H}_n)$ определяется как:

$$\{\hat{\sigma}_{k_1 \dots k_n}\}_{k_1,\dots,k_n \in \{0,1,2,3\}}, \quad \hat{\sigma}_{k_1 \dots k_n} = \hat{\sigma}_{k_1} \otimes \dots \otimes \hat{\sigma}_{k_n}, \tag{1.3}$$

где $\hat{\sigma}_{0\dots0}$ — тождественный оператор. Базис $P(\mathcal{H}_n)$ содержит 4^n элементов. Для краткости будем использовать обозначение

$$\hat{\sigma}_K = \hat{\sigma}_{k_1 \dots k_n} \tag{1.4}$$

где строка Паули $k_1 \dots k_n$ $(k_1, \dots, k_n \in \{0, 1, 2, 3\})$ соответствует числу K в десятичной системе $(0 \le K \le 4^n - 1)$. Строка Паули K и элемент $\hat{\sigma}_K$ взаимно однозначно соответствуют друг другу.

Сравним $P(\mathcal{H}_n)$ со стандартным базисом. Для элементов базиса Паули выполняются:

$$\hat{\sigma}_{k_1 \dots k_n} \hat{\sigma}_{k_1 \dots k_n} = \hat{\sigma}_{0\dots0}, \quad \operatorname{tr} \hat{\sigma}_{0\dots0} = 2^n, \quad \operatorname{tr} \hat{\sigma}_{k_1 \dots k_n}\Big|_{k_1 \dots k_n \neq 0\dots0} = 0. \tag{1.5}$$

Базис Паули является эрмитовым, унитарным и ортогональным относительно скалярного произведения (1.1). Отметим, что оператор $|k\rangle\langle l|$ из стандартного базиса не является унитарным или эрмитовым при $k \neq l$. Стандартный базис не включает тождественный оператор, который в этом базисе записывается как:

$$\sum_{k=0}^{2^n-1} |k\rangle\langle k|.$$

В базисе Паули любой оператор $\hat{U}$ из унитарной группы $U(\mathcal{H}_n)$ (где $\hat{U}^\dagger \hat{U} = \hat{\sigma}_{0\dots0}$) раскладывается в виде:

$$\hat{U} = \sum_{i_1,\dots,i_n \in \{0,1,2,3\}} U_{i_1 \dots i_n} \hat{\sigma}_{i_1 \dots i_n}, \quad \hat{U}^\dagger = \sum_{i_1,\dots,i_n \in \{0,1,2,3\}} \overline{U}_{i_1 \dots i_n} \hat{\sigma}_{i_1 \dots i_n},$$

где коэффициенты удовлетворяют условиям:

$$\sum_{i_1,\ldots,i_n\in\{0,1,2,3\}} \overline{U}_{i_1\ldots i_n} U_{i_1\ldots i_n} = 1, \qquad \sum_{\substack{i_1,\ldots,i_n,j_1,\ldots,j_n\in\{0,1,2,3\} \\ (i_1,\ldots,i_n)\neq(j_1,\ldots,j_n)}} \overline{U}_{i_1\ldots i_n} U_{j_1\ldots j_n} = 0.$$

Последнее условие эквивалентно $2^{2n-1}(2^n-1)$ независимым соотношениям.

В базисе Паули эрмитовы операторы представляются разложениями с вещественными коэффициентами [12, 13].

1.1.2 Коммутационные и антикоммутационные соотношения

Коммутатор определяет взаимодействие операторов при их перестановке. Для операторов A и B он задаётся как:

$$[A, B] = AB - BA.$$

Если $[A, B] = 0$, операторы коммутируют; в противном случае — нет. В базисе Паули коммутаторы выражаются через символ Леви-Чивиты ε_{ijk}:

$$[\sigma_i, \sigma_j] = 2i\varepsilon_{ijk}\sigma_k,$$

где ε_{ijk} равен 1 при чётной перестановке индексов (i, j, k), -1 при нечётной и 0 в остальных случаях. Например:

$$[\sigma_1, \sigma_2] = 2i\sigma_3, \quad [\sigma_2, \sigma_3] = 2i\sigma_1.$$

Антикоммутатор характеризует симметричное произведение операторов:

$$\{A, B\} = AB + BA.$$

Если $\{A, B\} = 0$, операторы антикоммутируют. Для операторов Паули:

$$\{\sigma_i, \sigma_j\} = 2\delta_{ij},$$

где δ_{ij} — символ Кронекера (1 при $i = j$, 0 иначе). Примеры:

$$\{\sigma_1, \sigma_2\} = 0, \quad \{\sigma_2, \sigma_3\} = 0.$$

Коммутаторы и антикоммутаторы применяются в квантовой механике для анализа свойств систем (спин электрона, кубиты), в квантовой теории поля (взаимодействия частиц) и квантовых вычислениях (алгоритмы, коррекция ошибок). Коммутаторы помогают определить совместную измеримость наблюдаемых, а антикоммутаторы — описать фермионные системы.

1.1.3 Коммутирующие элементы в алгебре Ли группы $SU(2^n)$

Рассмотрим базис Паули и множество строк Паули длины n:

$$\mathrm{Str}_n = \{K = k_1 \dots k_n\}_{k_1,\dots,k_n \in \{0,1,2,3\}}.$$

1. Множество $\mathbb{F}_4 = \{0, 1, 2, 3\}$ образует квадгруппу Клейна с умножением:

$$0 \cdot k = k, \quad k \cdot k = 0, \quad k \cdot l = m,$$

где $k, l, m \in \{1, 2, 3\}$ и (klm) — произвольная перестановка $(1\,2\,3)$.

2. Функция $s : \mathbb{F}_4 \times \mathbb{F}_4 \to \{1, i, -i\}$ задаётся значениями:

$$\begin{aligned} &s(0,0) = s(0,k) = s(k,0) = s(k,k) = 1, \quad k = 1,2,3, \\ &s(1,2) = s(2,3) = s(3,1) = i, \qquad\qquad s(2,1) = s(3,2) = s(1,3) = -i. \end{aligned}$$

3. Функция $S : \mathrm{Str}_n \times \mathrm{Str}_n \to \{1, -1, i, -i\}$ определяется как:

$$S_{KL} = s(k_1, l_1) \cdot s(k_2, l_2) \cdot \ldots \cdot s(k_n, l_n),$$

где $K = k_1 k_2 \dots k_n$ и $L = l_1 l_2 \dots l_n$.

Симметрия функции S зависит от числа пар (k_r, l_r) (на позициях r в строках K и L), где $k_r, l_r \in \{1, 2, 3\}$ и $k_r \neq l_r$, а также от их взаимного порядка. Пусть ω^+_{KL} и ω^-_{KL} — количество пар вида $(1,2), (2,3), (3,1)$ и $(2,1), (3,2), (1,3)$ соответственно, и $\omega_{KL} = \omega^+_{KL} + \omega^-_{KL}$. Тогда

$$S_{(KL)} = \frac{S_{KL}}{2}\left(1 + (-1)^{\omega_{KL}}\right), \quad S_{[KL]} = \frac{S_{KL}}{2}\left(1 - (-1)^{\omega_{KL}}\right), \tag{1.6}$$

где

$$S_{KL} = i^{\omega_{KL}} (-1)^{\omega^-_{KL}}.$$

$\omega_{KL}\mathrm{mod}4$	0	2	0	2	1	3	1	3
$\omega^{-}_{KL}\mathrm{mod}4$	0	1	1	0	0	1	1	0
S_{KL}		1	-1	-1	i	i	$-i$	$-i$
$S_{(KL)}$	1	1	-1	-1	0	0	0	0
$S_{[KL]}$	0	0	0	0	i	i	$-i$	$-i$

Таблица 1: Множитель до $\hat{\sigma}_M$ в (1.7) для $\hat{\sigma}_K\hat{\sigma}_L, \{\hat{\sigma}_K, \hat{\sigma}_L\}$, и $[i\hat{\sigma}_K, i\hat{\sigma}_L]$.

Здесь $S_{(KL)}$ и $S_{[KL]}$ — симметричная и антисимметричная части S_{KL}. Значения S_{KL}, $S_{(KL)}$ и $S_{[KL]}$ приведены в таблице 2.

Композицию элементов базиса Паули, их антикоммутаторов и коммутаторов можно компактно выразить в виде, удобном для программной реализации:

$$\hat{\sigma}_K\hat{\sigma}_L = S_{KL}\hat{\sigma}_M, \quad \{\hat{\sigma}_K, \hat{\sigma}_L\} = S_{(KL)}\hat{\sigma}_M, \quad [i\hat{\sigma}_K, i\hat{\sigma}_L] = -S_{[KL]}\hat{\sigma}_M, \quad (1.7)$$

где

$$\hat{\sigma}_M = \hat{\sigma}_{m_1\ldots m_n}, \quad m_r = k_r \cdot l_r \quad (r = 1, \ldots, n). \quad (1.8)$$

Две строки Паули длины n могут коммутировать, даже имея ненулевые элементы в одних и тех же позициях. Например, операторы $\hat{\sigma}_{11}$, $\hat{\sigma}_{22}$ и $\hat{\sigma}_{33}$ коммутируют друг с другом. Унитарная матрица перехода из стандартного базиса $\{|i_1 \ldots i_n\rangle\langle j_1 \ldots j_n|\}$ в базис Паули содержит только элементы 0, ± 1 и $\pm i$. Например:

$$|00\ldots0\rangle\langle 00\ldots0| \to \frac{1}{2^n} \sum_{i_1,\ldots,i_n \in \{0,3\}} \hat{\sigma}_{i_1\ldots i_n}.$$

Общее выражение для стандартных ортогональных проекторов имеет вид:

$$|i_1 \ldots i_n\rangle\langle i_1 \ldots i_n| = \frac{1}{2^n} \sum_{k_1,\ldots,k_n \in \{0,3\}} \mathcal{X}^{i_1}_{k_1} \cdots \mathcal{X}^{i_n}_{k_n} \hat{\sigma}_{k_1\ldots k_n},$$

где

$$\mathcal{X}^0_0 = \mathcal{X}^0_3 = \mathcal{X}^1_0 = 1, \quad \mathcal{X}^1_3 = -1.$$

Из выражения (1.7) следует, что: 1. Множество $\{i\hat{\sigma}_K\}_{K=0}^{4^n-1}$ образует ортонормированный базис в $\mathfrak{su}(2^n)$. 2. Множество

$$\widetilde{P}(\mathcal{H}_n) = \{\epsilon\hat{\sigma}_K \mid K \in \mathrm{Str}_n,\ \epsilon \in \{\pm 1, \pm i\}\},$$

содержащее 4^{n+1} элементов, образует группу — т.н. n-кубитную группу Паули.

Нормализатор группы Паули в унитарной группе:

$$\mathcal{C}(\mathcal{H}_n) = \left\{ \hat{U} \in U(\mathcal{H}_n) \mid \hat{U}\hat{\sigma}_K\hat{U}^\dagger \in \widetilde{P}(\mathcal{H}_n), \ \ \forall \hat{\sigma}_K \in \widetilde{P}(\mathcal{H}_n) \right\},$$

называется группой Клиффорда. Исходя из (1.2), (1.5) и (1.8) получаем следующее утверждение

Утверждение 1. *Взаимные унитарные преобразования базисных операторов Паули подчиняются соотношениям* $\hat{\sigma}_{i_1\ldots i_n}\hat{\sigma}_{k_1\ldots k_n}\hat{\sigma}_{i_1\ldots i_n} = \pm\hat{\sigma}_{i_1\ldots i_n}$, *где знак плюс стоит тогда и только тогда, когда количество троек* $(i_m k_m i_m)_{m\in\{1,\ldots,n\}}$, *удовлетворяют условиям* $i_m \neq k_m$, $i_m \neq 0$, *и* $k_m \neq 0$ *четности.*

l	0	1	2	3	4	5	6	7
$l_2l_1l_0$	000	001	010	011	100	101	110	111
$k_2k_1k_0$	011	011	011	011	011	011	011	011
$\bar{l} \wedge k$	011	010	001	000	011	010	001	000
$l \wedge k$	000	001	010	011	000	001	010	011
$l \wedge \overline{k}$	000	000	000	000	100	100	100	100
$\hat{\sigma}_I$	$\hat{\sigma}_{011}$	$\hat{\sigma}_{012}$	$\hat{\sigma}_{021}$	$\hat{\sigma}_{022}$	$\hat{\sigma}_{311}$	$\hat{\sigma}_{312}$	$\hat{\sigma}_{121}$	$\hat{\sigma}_{322}$

Таблица 2: Элементы базиса Паули, возникающие для $k = 011$.

1.2 Вариационная квантовая оптимизация

Пусть $\mathcal{H}_n$ — гильбертово пространство квантовой системы, состоящей из n кубитов, $\mathcal{S} \subset \mathcal{H}_n$ — пространство векторов состояния (т.е. векторов, нормированных на единицу), $L(\mathcal{H}_n)$ — алгебра операторов на $\mathcal{H}_n$ и $\hat{H} \in L(\mathcal{H}_n)$ — эрмитов оператор. Определим функцию

$$E(|u\rangle) = \langle u|\hat{H}|u\rangle, \quad |u\rangle \in \mathcal{S}. \tag{1.9}$$

В простейшей постановке ***квантовой задачи оптимизации*** *требуется найти вектор состояния, на котором целевая функция (функция стоимости)* E *принимает минимальное значение*, т.е., в формальной

записи, решить задачу

$$E(|u\rangle) \xrightarrow{|u\rangle\in\mathcal{S}} \min\,. \tag{1.10}$$

Ниже, для определенности и краткости, будем называть $\hat{H}$ гамильтонианом системы, а целевую функцию E — энергией.

Сложность алгоритмов прямого вычисления собственных значений гамильтониана $\hat{H}$ растет экспоненциально с ростом числа кубитов, поэтому для больших систем используются вариационные методы решения задачи оптимизации (1.10).

Вариационными квантовыми алгоритмами обычно называют такие гибридные квантово-классические алгоритмы, нацеленные на решение квантовых задач оптимизации посредством квантовых вычислений или их классической имитации, которые проводят вариационную настройку параметров квантовой схемы. Параметрически управляемое квантовое устройство, обычно представленное квантовой цепью, реализует анзац, т.е. унитарное преобразование стандартного начального состояния $|0\rangle^{\otimes n}$ или, как вариант, предудущего полученного состояния. На каждом шаге регулирующие параметры подбираются так, чтобы минимизировать энергию (целевую функцию). Обычно это выполняется путём измерения энергии состояний, предоставляемых вариационной схемой, и обновления параметров для минимизации целевой функции.

В точной математической формулировке сказанное означает, что в функции (1.9) вектор состояния $|u\rangle$ зависит от набора m параметров $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$, которые принимают значения в некоторой связной и односвязной области $\Omega \in \mathbb{R}^m$. Вариационная формулировка квантовой задачи оптимизации (1.10) имеет вид

$$E\big(\,|u(\boldsymbol{\theta})\rangle\,\big) \xrightarrow{\boldsymbol{\theta}\in\Omega} \min, \qquad E\big(\,|u(\boldsymbol{\theta})\rangle\,\big) = \langle u(\boldsymbol{\theta})|\,\hat{H}\,|u(\boldsymbol{\theta})\rangle\,. \tag{1.11}$$

Итак, цель вариационного квантового алгоритма — найти такой набор параметров, на котором энергия достигает минимума. Число параметров m в наборе $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$ зависит от конкретной задачи, в частности, от числа кубитов в квантовом устройстве. Для n кубитов размерность пространства состояний, $N = 2^n$, растет экспоненциально с ростом числа кубитов. Поэтому вариационный квантовый алгоритм должен быть

организован и выполнен так, чтобы выполнялось условие $m \ll N$, поскольку в противном случае высокий класс сложности алгоритма сделает его неэффективным с практической точки зрения.

Но наиболее важным вопросом является выбор зависимости вектора состояния $|u(\boldsymbol{\theta})\rangle$ от параметров. В вариационных квантовых алгоритмах используется *анзац* (унитарное преобразование) вида

$$|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})\,|u_0\rangle\,. \tag{1.12}$$

В общем случае форма анзаца определяет, какими будут параметры $\boldsymbol{\theta}$ и как их можно настроить для минимизации энергии (целевой функции). Структура анзаца, как правило, будет зависеть от поставленной задачи, так как во многих случаях можно использовать информацию о проблеме, чтобы подобрать анзац: это "анзац, подсказнный задачей". Однако можно построить анзацы достаточно общего вида, которые пригодны для использования в некоторых классах задач даже тогда, когда интуиция и известная информация о задаче не позволят его уточнить. Две наиболее распространенных формы анзаца рассмотрены в следующем разделе.

1.3 Общая схема алгоритма

1.3.1 Анзац

В вариационных квантовых алгоритмах анзац (1.12) стандартно выбирается в виде композиции m последовательно примененных унитарных преобразований

$$\hat{U}(\boldsymbol{\theta}) = \hat{U}_m(\theta_m)\cdots\hat{U}_1(\theta_1). \tag{1.13}$$

В композиции (1.13) выбор операторов определяется типом задачи и технической возможностью их реализации на конкретном квантовом устройстве. Например, можно выбрать

$$\hat{U}_K(\theta_K) = \hat{W}_K \exp\bigl(i\theta_K\hat{\sigma}_K\bigr) = \hat{W}_K\bigl(\cos\theta_K\hat{\sigma}_{0\ldots0} + i\sin\theta_K\hat{\sigma}_K\bigr), \tag{1.14}$$

где $1 \leqslant K \leqslant m$, $\hat{\sigma}_K = \hat{\sigma}_{k_1}\otimes\ldots\otimes\hat{\sigma}_{k_n}$, $k_1,\ldots,k_n \in \{0,1,2,3\}$, $K = k_1\ldots k_n$ (т.е. K — десятичное представление строки $k_1\ldots k_n$, рассматриваемой как число по основанию 4), n — число кубитов, а $\hat{W}_K$ — независящий от

параметров унитарный оператор. Как правило, в строке $k_1 \dots k_n$ только отдельные числа отличны от нуля, так что в тензорном произведении $\hat{\sigma}_{k_1} \otimes \dots \otimes \hat{\sigma}_{k_n}$ часть операторов являются тождественными.

В другом распространенном варианте операторы в композиции (1.13) имеют вид

$$\hat{U}_K(\theta_K) = \hat{W}_K\big(\mathrm{e}^{i\theta_{k_1}\hat{\sigma}_{k_1}} \otimes \dots \otimes \mathrm{e}^{i\theta_{k_n}\hat{\sigma}_{k_n}}\big), \tag{1.15}$$

где по-прежнему $1 \leqslant K \leqslant m$ и $K = k_1 \dots k_n$. Если в (1.14) все операторы $\hat{W}_K$ могут быть тождественными, то в (1.15), по крайней мере некоторые операторы $\hat{W}_K$ должны быть запутывающими и, следовательно, как минимум двухкубитными.

Таким образом, анзацы (1.13), (1.14) и (1.15) конкретизируют вариационную квантовую задачу оптимизации (1.11) и (1.12) в отношении параметрической зависимости вектора состояния,

$$|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})\,|0\dots0\rangle\,,$$

где начальное состояние имеет вид $|0\dots0\rangle = |0\rangle^{\otimes n}$. Если предположить далее, что мы в состоянии уверенно приготовить начальное состояние, реализовать анзац на физическом устройстве и вычислить значение энергии $E\big(|u(\boldsymbol{\theta})\rangle\big) = \langle u(\boldsymbol{\theta})|\hat{H}\,|u(\boldsymbol{\theta})\rangle$ посредством измерений (с привлечением классического компьютера), то следующий — основной — вопрос можно сформулировать так: как искать параметры, которые обеспечивают глобальный минимум энергии. Этот этап выполняется с помощью классического компьютера, так что вариационный квантовый алгоритм — гибридный квантово-классический алгоритм: параметризованная квантовая схема и измерительный прибор представляют квантовую часть, а алгоритм настройки параметров — классическую.

Опишем кратко наиболее общую схему вариационного квантового алгоритма посредством псевдокода в свободной форме (см. также Рис. 1.1).

1.3.2 Алгоритм в псевдокодах

1. Ввод начального вектора параметров $\boldsymbol{\theta}$. Полагаем $\boldsymbol{\theta}_0 = \boldsymbol{\theta}$.
2. Генерация состояния $|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})\,|0\rangle^{\otimes n}$ и случайного вектора $\delta\boldsymbol{\theta}$.
3. Вычисление энергии $E(\boldsymbol{\theta}) = \langle u(\boldsymbol{\theta})|\hat{H}\,|u(\boldsymbol{\theta})\rangle$ с текущим вектором $\boldsymbol{\theta}$.

4. Алгоритмический переход к вектору параметров $\boldsymbol{\theta}_1 = \boldsymbol{\theta} + \delta\boldsymbol{\theta}$. Если $E(\boldsymbol{\theta}_1) < E(\boldsymbol{\theta})$, то положим $\boldsymbol{\theta} = \boldsymbol{\theta}_1$ и запомним $\boldsymbol{\theta}_0 = \boldsymbol{\theta}_1$.

5. Условие завершения работы. Если условие выполнено, то выходим с результатом $|u(\boldsymbol{\theta}_0)\rangle$ и $E(\boldsymbol{\theta}_0)$. Иначе переходим к метке 2.

Переход к новому вектору параметров $\boldsymbol{\theta}_1$ является вторым ключевым пунктом (после выбора анзаца), который характеризует различные типы вариационных квантовых алгоритмов. Условие завершения работы может быть выбрано очень многими способами. Простейшим и одновременно универсальным является условие отсутствия понижения энергии в течение определенного числа циклов работы алгоритма.

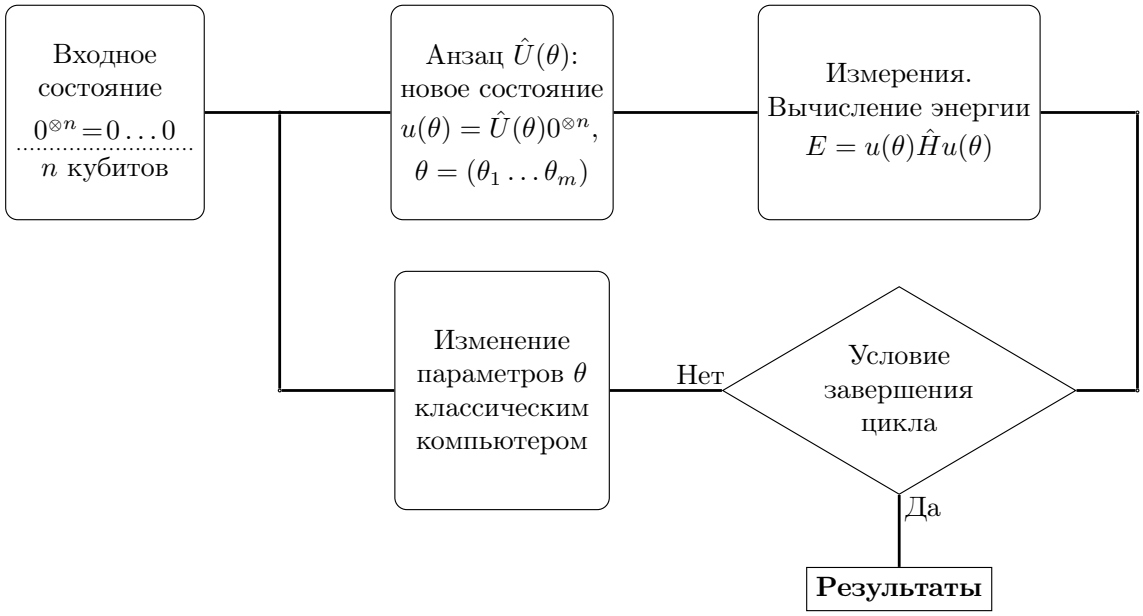

Рис. 1.1: Общая схема квантового вариационного алгоритма. Более подробная блок-схема алгоритма представлена на Рис. 2.1

1.4 Пример, иллюстрирующий особенности алгоритма

Для иллюстрации алгоритма рассмотрим гамильтониан

$$\hat{H} = 2\,\hat{\sigma}_{03} + \hat{\sigma}_{30} - 4\,\hat{\sigma}_{11}, \tag{1.16}$$

который в стандартном базисе $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ имеет матрицу

$$H = \begin{pmatrix} 3 & 0 & 0 & -4 \\ 0 & -1 & -4 & 0 \\ 0 & -4 & 1 & 0 \\ -4 & 0 & 0 & -3 \end{pmatrix}.$$

Используя систему Maple находим собственные значения и собственные состояния в порядке возрастания собственных значений, начиная с основного состояния $|u_0\rangle$ с собственным значением E_0:

$$E_0 = -5, \qquad |u_0\rangle = \frac{1}{\sqrt{5}}|00\rangle + \frac{2}{\sqrt{5}}|11\rangle, \tag{1.17}$$

$$E_1 = -\sqrt{17}, \quad |u_1\rangle = \frac{\sqrt{17}+1}{\sqrt{34+2\sqrt{17}}}|01\rangle + \frac{\sqrt{8}}{\sqrt{17+\sqrt{17}}}|10\rangle, \tag{1.18}$$

$$E_2 = \sqrt{17}, \quad |u_1\rangle = -\frac{\sqrt{17}-1}{\sqrt{34-2\sqrt{17}}}|01\rangle + \frac{\sqrt{8}}{\sqrt{17-\sqrt{17}}}|10\rangle, \tag{1.19}$$

$$E_3 = 5, \qquad |u_3\rangle = -\frac{2}{\sqrt{5}}|00\rangle + \frac{1}{\sqrt{5}}|11\rangle. \tag{1.20}$$

Рассмотрим далее пошаговое выполнение вариационного квантового алгоритма, который позволяет найти состояние, близкое к основному.

Первый шаг — выбор анзаца, т.е. унитарного преобразования $\hat{U}(\boldsymbol{\theta})$. В гамильтониан (1.16) не входят операторы вида $\hat{\sigma}_{k2}$ и $\hat{\sigma}_{k2}$ с $k \neq 2$, поэтому имеет смысл сразу выбирать анзац так, чтобы при действии на $k \neq 2$ он давал вектор состояния с вещественными коэффициентами. Других наводящих соображений относительно формы анзаца не видно, поэтому следует рассмотреть разные варианты. В общем случае вектор параметров $\boldsymbol{\theta}$ четырехмерен. В простейшем варианте анзац с четырехмерным вектором параметров $\boldsymbol{\theta} = (\xi, \lambda, \mu, \nu)$ можно выбирать как композицию экспонент

$$\hat{U}(\boldsymbol{\theta}) = \mathrm{e}^{i\xi\hat{\sigma}_{02}}\mathrm{e}^{i\lambda\hat{\sigma}_{03}}\mathrm{e}^{i\mu\hat{\sigma}_{30}}\mathrm{e}^{i\nu\hat{\sigma}_{11}} \tag{1.21}$$

операторов Паули, присутствующих в гамильтониане (1.16). Вычислим

вначале

$$\mathrm{e}^{i\mu\hat{\sigma}_{30}}\mathrm{e}^{i\nu\hat{\sigma}_{11}}\,|00\rangle = \big(\cos\mu\,\hat{\sigma}_{00} + i\sin\mu\,\hat{\sigma}_{30}\big)\big(\cos\nu\,|00\rangle + i\sin\nu\,|11\rangle\big)$$

$$= \cos\mu\cos\nu\,|00\rangle - \sin\mu\sin\nu\,|11\rangle + i\sin\mu\cos\nu\,|00\rangle + i\cos\mu\sin\nu\,|11\rangle$$

$$= \mathrm{e}^{i\mu}\cos\nu\,|00\rangle + i\mathrm{e}^{i\mu}\sin\nu\,|11\rangle = \mathrm{e}^{i\mu}\cos\nu\,|00\rangle + \mathrm{e}^{i(\mu+\pi/2)}\sin\nu\,|11\rangle\,.$$

Действуя на результат оператором $\mathrm{e}^{i\lambda\hat{\sigma}_{03}}$, получим следующий промежуточный вектор состояния:

$$\mathrm{e}^{i\lambda\hat{\sigma}_{03}}\mathrm{e}^{i\mu\hat{\sigma}_{30}}\mathrm{e}^{i\nu\hat{\sigma}_{11}}\,|00\rangle = \mathrm{e}^{i\mu}\cos\nu\big(\cos\lambda\,\hat{\sigma}_{00} + i\sin\lambda\hat{\sigma}_{03}\big)\,|00\rangle$$
$$+ \mathrm{e}^{i(\mu+\pi/2)}\sin\nu\big(\cos\lambda\,\hat{\sigma}_{00} + i\sin\lambda\hat{\sigma}_{03}\big)\,|11\rangle$$

$$= \mathrm{e}^{i\mu}\cos\nu\big(\cos\lambda + i\sin\lambda\big)\,|00\rangle + \mathrm{e}^{i(\mu+\pi/2)}\sin\nu\big(\cos\lambda + i\sin\lambda\big)\,|11\rangle$$
$$= \mathrm{e}^{i(\mu+\lambda)}\cos\nu\,|00\rangle + \mathrm{e}^{i(\mu+\pi/2-\lambda)}\sin\nu\,|11\rangle\,. \qquad (1.22)$$

Очевидно, что этот анзац не является универсальным.

Варьируя параметры λ, μ, ν, можно получить основное состояние (1.17) с точностью до несущественного множителя $\mathrm{e}^{i(\mu+\pi/4)}$, например, при

$$\lambda = \pi/4, \quad \mu \in \mathbb{R}, \quad \cos\nu = 1/\sqrt{5}, \quad \sin\nu = 2/\sqrt{5}. \qquad (1.23)$$

Здесь μ — любое, поэтому к нужному результату приводит более простой анзац (при $\mu = 0$) $\hat{U}(\boldsymbol{\theta}) = \mathrm{e}^{i\lambda\hat{\sigma}_{03}}\mathrm{e}^{i\nu\hat{\sigma}_{11}}$, однако заранее это нам не известно. Более того основное состояние (1.17) можно достигнуть (что заранее также неизвестно и неочевидно) даже однопараметрическим анзацем

$$\mathrm{e}^{i\nu\hat{\sigma}_{12}}\,|00\rangle = \big(\cos\nu\,\hat{\sigma}_{00} + i\sin\nu\,\hat{\sigma}_{12}\big)\,|00\rangle = \cos\nu\,|00\rangle + \sin\nu\,|11\rangle\,,$$

с теми же значениями $\cos\nu$ и $\sin\nu$, что и в (1.23).

Действуя на (1.22) оператором $\mathrm{e}^{i\xi\hat{\sigma}_{02}}$, получим вектор состояния

$$|\Phi\rangle = \hat{U}(\boldsymbol{\theta})\,|00\rangle$$

$$= \big(\cos\xi\,\hat{\sigma}_{00} + i\sin\xi\,\hat{\sigma}_{02}\big)\big(\mathrm{e}^{i(\mu+\lambda)}\cos\nu\,|00\rangle + \mathrm{e}^{i(\mu+\pi/2-\lambda)}\sin\nu\,|11\rangle\big)$$

$$= \mathrm{e}^{i(\mu+\lambda)}\sin\xi\cos\nu\,|01\rangle - \mathrm{e}^{i(\mu+\pi/2-\lambda)}\sin\xi\sin\nu\,|10\rangle$$
$$+ \mathrm{e}^{i(\mu+\lambda)}\cos\xi\cos\nu\,|00\rangle + \mathrm{e}^{i(\mu+\pi/2-\lambda)}\cos\xi\sin\nu\,|11\rangle\,, \qquad (1.24)$$

который зависит от четырех параметров. Из формы данного вектора видно, что анзац (1.21) универсален (с учетом замечания о вещественности коэффициентов, сделанного выше). Основное состояние достигается при произвольном $\mu \in \mathbb{R}$ и

$$\xi = 0,\ \lambda = \{\pi/4, 7\pi/4\},\ \cos\nu = 1/\sqrt{5},\ \sin\nu = \{2/\sqrt{5}, -2/\sqrt{5}\} \quad (1.25)$$

или

$$\xi = \pi,\ \lambda = \{\pi/4, 7\pi/4\},\ \cos\nu = -1/\sqrt{5},\ \sin\nu = \{-2/\sqrt{5}, 2/\sqrt{5}\}. \quad (1.26)$$

Для сокращения записи имеет смысл освободиться в (1.24) от фазового множителя и записать вектор состояния в виде

$$|\Phi\rangle = \sin\xi\big(\cos\nu\,|01\rangle - \mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\,|10\rangle\big) + \cos\xi\big(\cos\nu\,|00\rangle + \mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\,|11\rangle\big) \quad (1.27)$$

Второй шаг — вычисление энергии состояния, т.е. среднего значения $\langle\Phi|\hat{H}|\Phi\rangle$. Заметим, что первый и второй шаги должны выполняться на квантовых устройствах, а при классической симуляции алгоритма необходимо проводить явные вычисления. Из (1.16) и (1.27) находим

$$\begin{aligned}\hat{H}|\Phi\rangle &= \sin\xi\big(2\,\hat{\sigma}_{03} + \hat{\sigma}_{30} - 4\hat{\sigma}_{11}\big)\big(\cos\nu\,|01\rangle - \mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\,|10\rangle\big) \\ &\quad + \cos\xi\big(2\,\hat{\sigma}_{03} + \hat{\sigma}_{30} - 4\hat{\sigma}_{11}\big)\big(\cos\nu\,|00\rangle + \mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\,|11\rangle\big) \\ &= \sin\xi\Big\{\big(4\mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu - \cos\nu\big)\,|01\rangle \\ &\qquad - \big(\mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu + 4\cos\nu\big)\,|10\rangle\Big\} \\ &\quad + \cos\xi\Big\{\big(3\cos\nu - 4\mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\big)\,|00\rangle \\ &\qquad - \big(4\cos\nu + 3\mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\big)\,|11\rangle\Big\}.\end{aligned}$$

Поскольку

$$\langle\Phi| = \sin\xi\big(\cos\nu\,\langle 01| - \mathrm{e}^{-i(\pi/2-2\lambda)}\sin\nu\,\langle 10|\big) + \cos\xi\big(\cos\nu\,\langle 00| + \mathrm{e}^{-i(\pi/2-2\lambda)}\sin\nu\,\langle 11|\big),$$

то

$$
\begin{aligned}
E_\Phi &= \langle\Phi|\,\hat{H}\,|\Phi\rangle \\
&= \sin^2\xi\Big\{\cos\nu\big(4\mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu-\cos\nu\big) \\
&\qquad\qquad +\sin\nu\big(\sin\nu+4\mathrm{e}^{-i(\pi/2-2\lambda)}\cos\nu\big)\Big\} \\
&\quad+\cos^2\xi\Big\{\cos\nu\big(3\cos\nu-4\mathrm{e}^{i(\pi/2-2\lambda)}\sin\nu\big) \\
&\qquad\qquad -\sin\nu\big(4\mathrm{e}^{-i(\pi/2-2\lambda)}\cos\nu+3\sin\nu\big)\Big\} \\
&= \sin^2\xi\big(4\sin2\lambda\sin2\nu-\cos2\nu\big)+\cos^2\xi\big(3\cos2\nu-4\sin2\lambda\sin2\nu\big). \quad (1.28)
\end{aligned}
$$

Разумеется, если взять значения ξ, λ, $\cos\nu$, $\sin\nu$ как в (1.25) или в (1.26), то мы получим энергию основного состояния (1.17), т.е. $E_\Phi=-5$.

Третий шаг — изменение значений параметров λ,μ,ν (с целью минимизации E_Φ) и возвращение к первому шагу; предполагается, что в начале выполнения алгоритма начальные значения параметров заданы. Из (1.28) видно, что на значение E_Φ параметр μ не влияет, а параметры λ,ν должны варьироваться в области $[0,\pi]\times[0,\pi]$. Однако изначально это неизвестно, поэтому все четыре параметра должны варьироваться в области $[0,2\pi]\times[0,2\pi]\times[0,2\pi]\times[0,2\pi]$. Имеет смысл установить независимость энергии от параметра μ (и ее зависимость от остальных параметров) в начале работы алгоритма.

Мы уже знаем, что глобальный минимум энергии достигается для четырех наборов параметров (1.25) и (1.26). Соответствующие собственные векторы, вычисленные по выражению (1.27) отличаются от (1.17) только фазовыми множителями. Теперь необходимо выяснить, имеются ли у функции (трех переменных) (1.28) другие локальные минимумы.

Используя систему Maple, вычислим производные

$$
\partial_\xi E_\Phi,\ \ \partial_\lambda E_\Phi,\ \ \partial_\nu E_\Phi,
$$

$$
A=\partial^2_\xi E_\Phi,\ \ B=\partial^2_\lambda E_\Phi,\ \ C=\partial^2_\nu E_\Phi,
$$

$$
K=\partial_{\xi\lambda} E_\Phi,\ \ L=\partial_{\xi\nu} E_\Phi,\ \ M=\partial_{\lambda\nu} E_\Phi.
$$

Находим

$$\begin{aligned}
\partial_\xi E_\Phi &= 4\sin 2\xi\big(2\sin 2\lambda\sin 2\nu - \cos 2\nu\big),\\
\partial_\lambda E_\Phi &= -8\cos 2\xi\cos 2\lambda\sin 2\nu,\\
\partial_\nu E_\Phi &= \sin^2\xi\big(8\sin 2\lambda\cos 2\nu + 2\sin 2\nu\big) - \cos^2\xi\big(6\sin 2\nu + 8\sin 2\lambda\cos 2\nu\big),\\
A &= 8\cos 2\xi\big(2\sin 2\lambda\sin 2\nu - \cos 2\nu\big),\\
B &= 16\cos 2\xi\sin 2\lambda\sin 2\nu,\\
C &= 4\cos^2\xi\big(4\sin 2\lambda\sin 2\nu - 3\cos 2\nu\big) - 4\sin^2\xi\big(4\sin 2\lambda\sin 2\nu - \cos 2\nu\big),\\
K &= 16\sin 2\xi\cos 2\lambda\sin 2\nu,\\
L &= 4\sin 2\xi\big(4\sin 2\lambda\cos 2\nu + 2\sin 2\nu\big),\\
M &= -16\cos 2\xi\cos 2\lambda\cos 2\nu.
\end{aligned}$$

Необходимые и достаточные условия минимума имеют вид

$$\partial_\xi E_\Phi = 0, \quad \partial_\lambda E_\Phi = 0, \quad \partial_\nu E_\Phi = 0,$$

$$A > 0, \qquad \det\begin{pmatrix} A & K \\ K & B \end{pmatrix} > 0, \qquad \det\begin{pmatrix} A & K & L \\ K & B & M \\ L & M & C \end{pmatrix} > 0.$$

Снова проводя вычисления с помощью системы Maple, обнаруживаем четыре точки локального минимума с энергией $E_\Phi = -\sqrt{17}$:

$$\begin{aligned}
&\xi = \frac{\pi}{2}, \qquad \lambda = \frac{\pi}{4}, \qquad \nu = \pi - \frac{1}{2}\arctan(4),\\
&\xi = \frac{\pi}{2}, \qquad \lambda = \frac{\pi}{4}, \qquad \nu = 2\pi - \frac{1}{2}\arctan(4),\\
&\xi = \frac{\pi}{2}, \qquad \lambda = \frac{3\pi}{4}, \qquad \nu = \frac{1}{2}\arctan(4),\\
&\xi = \frac{\pi}{2}, \qquad \lambda = \frac{3\pi}{4}, \qquad \nu = \pi + \frac{1}{2}\arctan(4).
\end{aligned}$$

Таким образом, в процессе оптимизации целевой функции должны использоваться методы, которые позволяют избежать попадания в точку локального минимума, например, метод отжига.

Глава 2

Вариационный квантовый алгоритм и метод отжига

2.1 Метод отжига

Метод отжига или, подробнее, метод имитации отжига относится к семейству методов Монте-Карло, разработанных для поиска *глобальˆного* минимума целевой функции. Этот метод обладает одним существенным преимуществом по сравнению с аналитическими методами оптимизации: он применим к целевым функциям произвольной природы. Особенно важно то, что метод отжига полностью сохраняет свою эффективность в задачах, где целевая функция не дифференцируема и при этом имеет большое число локальных минимумов. По этой причине, метод отжига, как фундаментальная концепция в теории глобальной оптимизации, широко используется в квантовых вычислениях, особенно в вариационных квантовых алгоритмах (в их квантовой реализации, а не классической эмуляции), где вычисление целевой функции не обладает достаточной точностью. Основная идея метода заключается в постепенном снижении "температуры" системы, чтобы достичь состояния с минимальным значением целевой функции (далее, для краткости, энергии) [10, 11]. В этом разделе подробно рассматривается классический вариант отжига и детали его практического применения.

Классический метод отжига основывается на аналогии с физическим процессом термического отжига, при котором материал медленно охла-

ждается, чтобы избежать образования дефектов и достичь состояния минимальной энергии. Математическое обоснование метода связано с распределением Больцмана, которое описывает вероятность состояния системы при заданной температуре T формулой

$$p(x) = \frac{1}{Z(T)} \exp\left(-\frac{E(x)}{k_B T}\right),$$

в которой $E(x)$ — энергия состояния x, k_B — постоянная Больцмана, а $Z(T)$ — статистическая сумма. Процесс отжига является дискретным и моделирует систему, которая на каждом шаге может переходить между состояниями x и y с вероятностью, которая зависит от разности энергий $\Delta E = E(y) - E(x)$:

$$p(x \to y) = \min\left(1, \exp\left(-\frac{\Delta E}{k_B T}\right)\right). \tag{2.1}$$

С вероятностью $1 - p(x \to y)$ состояние системы не изменяется.

2.1.1 Список данных

В самом общем случае в абстрактную математическую постановку задачи оптимизации методом отжига входят следующие данные.

1. Множество состояний C.
2. Энергия (целевая функция) $E : C \to \mathbf{R}$.
3. Семейство $Rnd = \left\{R_s\right\}_{s \in J}$ отображений $R_s \colon C \to C$, где множество индексов J, вообще говоря, не счетно.
4. Начальная температура T, минимальная температура T_{min} и закон ее понижения $T_k \to T_{k+1} = F(T_k, k)$, где k — шаг процесса отжига.

Практическая реализация имитации отжига обычно осуществляется посредством последовательности конечного числа шагов — случайных испытаний. Каждое испытание состоит, во-первых, из случайного выбора некоторого отображения $R_s \in Rnd$, задающего переход $x \to y = R_s(x)$, т.е. этот переход выбирается случайным образом из некоторого семейства

возможностей. В зависимости от типа задачи вероятностное распределение (вероятностная мера) на Rnd выбирается в достаточно широком диапазоне (нормальное распределение, распределение Коши, равномерное распределение и т.д.). Такое распределение, в свою очередь, может зависеть от температуры.

Во-вторых, с помощью генератора псевдослучайных чисел выбирается значение ε равномерно распределенной на интервале $(0, 1)$ случайной величины. Если выполнено условие

$$\varepsilon < \exp \frac{E(x) - E(y)}{k_B T},$$

то переход $x \to y$ принимается даже если энергия нового состояния, $E(y)$, больше, чем энергия $E(x)$ предыдущего состояния. В противном случае, когда $E(y) \leqslant E(x)$, данное условие выполняется. Такой способ перехода предохраняет от попадания в локальный минимум, в особенности, если процедуру отжига повторять несколько раз. Наконец, в третьих, на каждом шаге производится понижение температуры в соответствии с некоторым правилом. Постепенное уменьшение температуры приводит к уменьшению вероятности перехода в состояния с более высокой энергией, в то время как система стремится к состоянию глобального минимума энергии.

2.2 Алгоритм

2.2.1 Краткое описание алгоритма

В задачах квантовой оптимизации в системе из n кубитов множество состояний C, входящее в *Список данных* из предыдущего раздела 2.1, является подмножеством в пространстве состояний системы $\mathcal{H}_n$. Оно возникает в результате применения параметризованного анзаца $\hat{U}(\boldsymbol{\theta})$ к начальному состоянию $|0 \dots 0\rangle = |0\rangle^{\otimes n}$, так что $|u(\boldsymbol{\theta})\rangle = \hat{U}(\boldsymbol{\theta})\,|0\rangle^{\otimes n}$, где

$$\boldsymbol{\theta} = (\theta_1, \dots, \theta_m) \in \Omega = [0, 2\pi]^m.$$

Таким образом, $|u(\boldsymbol{\theta})\rangle \in C$ — это функция на пространстве параметров Ω со значениями в $\mathcal{H}_n$; при этом C не является подпространством в $\mathcal{H}_n$.

Энергия (целевая функция на Ω) из *Списка данных, п. 2*, определяется выражением $E(\boldsymbol{\theta}) = \langle u(\boldsymbol{\theta})|\hat{H}\,|u(\boldsymbol{\theta})\rangle$.

В предлагаемом алгоритме выбор отображения из семейства *Rnd* (*Список данных, п. 3*) равновероятен в отношении любого отображения в следующем смысле. С помощью генератора псевдослучайных чисел выбирается m значений $\varepsilon_i \in (0,\ 1)$, $i = 1, \ldots, m$ случайной величины, равномерно распределенной на данном интервале, а затем формируется случайный вектор

$$\delta\boldsymbol{\theta} = (\delta\theta_1, \ldots, \delta\theta_m) \in \Omega, \quad \delta\theta_i = \varepsilon_i\pi.$$

Отображение $R_{\delta\boldsymbol{\theta}} \colon C \to C$ определяется формулой $|u(\boldsymbol{\theta})\rangle \mapsto |u(\boldsymbol{\theta}+\delta\boldsymbol{\theta})\rangle$, где вектор $\boldsymbol{\theta} + \delta\boldsymbol{\theta}$ берется по модулю 2π. Далее производим понижение температуры (*Список данных, п. 4*) по простейшей схеме равномерного уменьшения ее значения на каждом шаге процесса отжига $T_{k+1} = T_k - \tau$, где выбор значения τ производится на основе численных экспериментов с конкретной задачей.

2.2.2 Квантовая оптимизация методом отжига

Краткая формулировка алгоритма на уровне псевдокода выглядит следующим образом [10, 11].

1. Вводим начальный вектор $\boldsymbol{\theta} \in \Omega$, минимальную температуру T_{min}, текущую температуру $T > T_{min}$ и значение τ. Положим $\boldsymbol{\theta}_0 = \boldsymbol{\theta}$.
2. Генерируем состояние $|u(\boldsymbol{\theta})\rangle$ и вычисляем энергию $E(\boldsymbol{\theta})$ при текущей температуре T.
3. Генерируем случайный вектор $\delta\boldsymbol{\theta} \in \Omega$, $\boldsymbol{\theta}_1 = \boldsymbol{\theta} + \delta\boldsymbol{\theta} \pmod{2\pi}$. Вычисляем энергию $E(\boldsymbol{\theta}_1)$.
4. Если $E(\boldsymbol{\theta}_1) \leqslant E(\boldsymbol{\theta})$, то полагаем $\boldsymbol{\theta} = \boldsymbol{\theta}_1$ и $\boldsymbol{\theta}_0 = \boldsymbol{\theta}_1$. Иначе генерируем $\varepsilon \in (0, 1)$ и если $\varepsilon < \exp\{[E(\boldsymbol{\theta}) - E(\boldsymbol{\theta}_1)]/T\}$, то полагаем $\boldsymbol{\theta} = \boldsymbol{\theta}_1$.
5. Понижаем температуру: $T \mapsto T - \tau$.
6. Если $T > T_{min}$, то переходим к метке 2. Иначе выходим с результатом $|u(\boldsymbol{\theta}_0)\rangle$ и $E(\boldsymbol{\theta}_0)$.

Теперь рассмотрим программную реализацию вариационного квантового алгоритма с оптимизацией методом отжига.

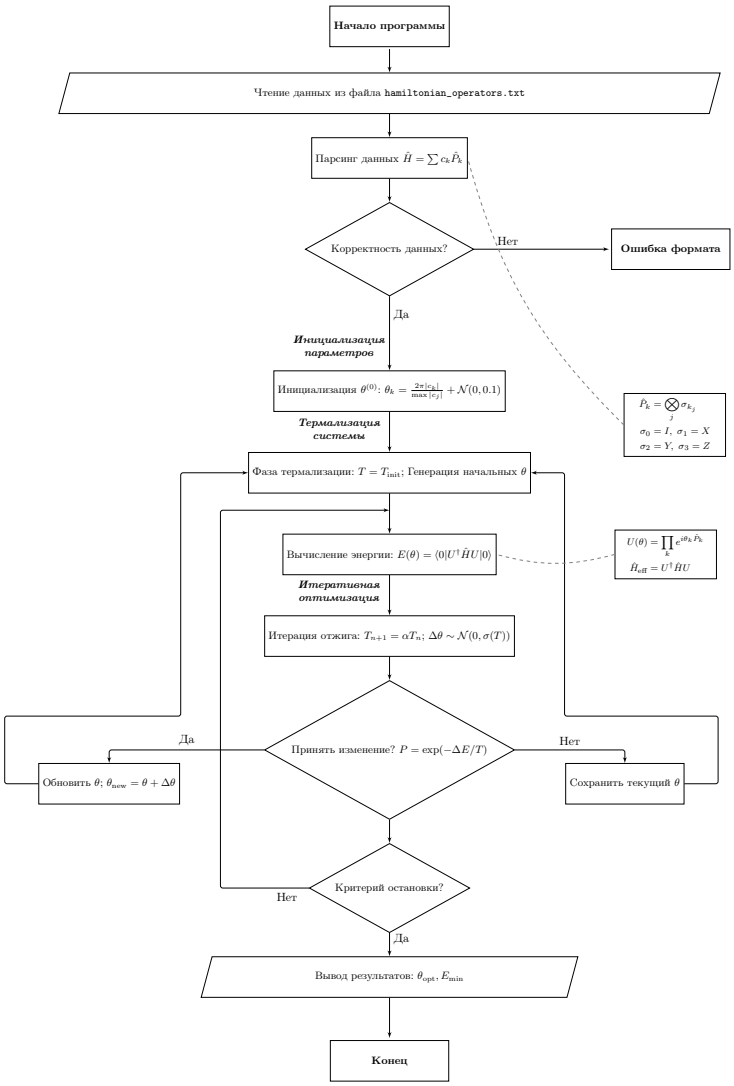

Рис. 2.1: Подробная блок-схема квантового вариационного алгоритма

2.2.3 Общая структура программного кода

Разработанная программа реализует вариационный квантовый алгоритм с классическим оптимизатором на основе метода имитации отжига (simulated annealing). На вход подаётся гамильтониан в базисе Паули, после чего осуществляется поиск минимума энергии в пространстве параметризованных квантовых состояний, генерируемых вариационным анзацем. Программа построена модульно и позволяет последовательно:

1. Прочитать описание гамильтониана из текстового файла;

2. Построить вариационный анзац в виде произведения экспонент Паули-операторов;

3. Численно вычислить энергию состояния $E(\boldsymbol{\theta})$;

4. Найти минимум энергии методом отжига;

5. Вывести подробную информацию о найденном состоянии и параметрах анзаца.

2.2.4 Описание гамильтониана

Гамильтониан $\hat{H}$ вводится в виде линейной комбинации операторов Паули

$$\hat{H} = \sum_{K \in \mathcal{T}} h_K \hat{\sigma}_K, \tag{2.2}$$

где $K = k_1 \dots k_n$ — строка Паули (см. определения (1.3) и (1.4)), набор индексов $\mathcal{T}$ — множество строк Паули (или, что равносильно, их десятичных представлений), а h_K — действительные числа.

В файле `hamiltonian_operators.txt` каждая строка имеет вид

`[действительная часть] [мнимая часть] [строка Паули]`

Например, строка `2.0 0.0 03` задаёт слагаемое $2.0 \cdot \sigma_0 \otimes \sigma_3$.

2.2.5 Загрузка гамильтониана

Для загрузки и парсинга гамильтониана используется функция

```python
def read_hamiltonian_data(file_path):
    lines = read_file_lines(file_path, ignore_comments=False)
    pauli_operators = []
    pauli_strings = []
    for line in lines:
        parts = line.strip().split()
        if len(parts) == 3:
            real_part, imag_part, index_str = (
                float(parts[0]),
                float(parts[1]),
                str(parts[2]),
            )
            coefficient = np.complex128(real_part + imag_part * 1j)
            index_list = [int(c) for c in index_str]
            if coefficient != 0:
                pauli_operators.append((coefficient, index_list))
```

```
            pauli_strings.append(index_list)
    return pauli_operators, pauli_strings
```

2.2.6 Вариационный анзац в базисе Паули

Анзац реализован в виде произведения операторов Паули, параметризованных углами $\boldsymbol{\theta}$:

$$\hat{U}(\boldsymbol{\theta}) = \prod_{j=1}^{m} \exp\left(i\theta_j \cdot \|c_j\| \cdot \hat{\sigma}_{K_j}\right) \tag{2.3}$$

где $\hat{\sigma}_{K_j}$ — базисные операторы Паули, входящие в гамильтониан, а c_j — соответствующие коэффициенты. Для разложения экспоненты используется формула Эйлера

$$\exp(i\alpha\hat{\sigma}) = \cos\alpha \cdot I + i\sin\alpha \cdot \hat{\sigma}. \tag{2.4}$$

Реализация разложения анзаца по базису Паули:

```
def calculate_ansatz(
    theta: np.ndarray, pauli_operators: List[Tuple[complex, List[int]]]
) -> Tuple[Dict[Tuple[int, ...], complex], str, str]:
    operator_length = len(pauli_operators[0][1])
    result = {tuple([0] * operator_length): 1.0}
    for t, (coeff, op) in zip(theta, pauli_operators):
        angle = t * abs(coeff)
        cos_t = np.cos(angle)
        sin_t = np.sin(angle)
        new_result = {}
        op_tuple = tuple(op)
        for existing_op, existing_coeff in result.items():
            new_result[existing_op] = (
                new_result.get(existing_op, 0) + existing_coeff * cos_t
            )
            compose_coeff, compose_op = pauli_compose(existing_op,
               op_tuple)
            final_coeff = existing_coeff * 1j * sin_t * compose_coeff
            new_result[compose_op] = new_result.get(compose_op, 0) +
               final_coeff
        result = new_result
    symbolic_str, numeric_str = format_ansatz(pauli_operators, result)
    return result, symbolic_str, numeric_str
```

2.2.7 Алгебра Паули и композиция операторов

Композиция двух операторов Паули $\hat{\sigma}_K$, $\hat{\sigma}_L$ реализуется покубитно согласно формулам (1.2), которые можно представить таблицей умножения

$$\hat{\sigma}_i\hat{\sigma}_j = \begin{cases} \hat{\sigma}_0 & \text{если } \ i = j \\ \hat{\sigma}_j & \text{если } \ i = 0 \\ \hat{\sigma}_i & \text{если } \ j = 0 \\ i\,\mathrm{sign}(\pi)\hat{\sigma}_k & \text{если } \ (ijk) = \pi(123), \end{cases} \tag{2.5}$$

где $i, j, k \in \{0, 1, 2, 3\}$, π — перестановка.

Программная реализация:

```python
def multiply_pauli(i: int, j: int) -> Tuple[complex, int]:
    if i == j:
        return (1, 0)
    if i == 0:
        return (1, j)
    if j == 0:
        return (1, i)
    return PAULI_MAP.get((i, j), (1, 0))

def pauli_compose(s1: tuple, s2: tuple) -> Tuple[complex, tuple]:
    coefficient = 1.0
    result = []
    for a, b in zip(s1, s2):
        coeff, idx = multiply_pauli(a, b)
        coefficient *= coeff
        result.append(idx)
    return coefficient, tuple(result)
```

2.2.8 Вычисление энергии состояния

Энергия состояния для текущего набора параметров $\boldsymbol{\theta}$ вычисляется как математическое ожидание гамильтониана в состоянии $|u(\boldsymbol{\theta})\rangle$:

$$E(\boldsymbol{\theta}) = \langle 0|\hat{U}^\dagger(\boldsymbol{\theta})\,\hat{H}\,\hat{U}(\boldsymbol{\theta})\,|0\rangle \tag{2.6}$$

Реализация последовательной композиции операторов $U^\dagger HU$ и вычисления среднего значения:

```python
def compute_uhu(
    u_dict: Dict[Tuple[int, ...], complex], h_terms:
       List[Tuple[complex, List[int]]]
) -> Dict[Tuple[int, ...], complex]:
    uhu_dict = {}
    for coeff_h, op_h in h_terms:
        op_h_tuple = tuple(op_h)
        for j_op, j_coeff in u_dict.items():
            conj_j_coeff = np.conj(j_coeff)
            c1, op_uh = pauli_compose(j_op, op_h_tuple)
            for k_op, k_coeff in u_dict.items():
                c2, op_uhu = pauli_compose(op_uh, k_op)
                total_coeff = conj_j_coeff * k_coeff * coeff_h * c1 * c2
                uhu_dict[op_uhu] = uhu_dict.get(op_uhu, 0) + total_coeff
    return uhu_dict

def calculate_expectation(uhu_dict: Dict[Tuple[int, ...], complex]) ->
   float:
    expectation = 0.0
    for op, coeff in uhu_dict.items():
        if all(p in (0, 3) for p in op):
            expectation += coeff.real
    return expectation
```

Вклад в среднее значение дают только те операторы Паули, которые не изменяют состояние $|0\ldots0\rangle$, то есть состоящие только из I и Z на каждом кубите.

2.2.9 Метод имитации отжига (Simulated Annealing)

Поиск глобального минимума осуществляется методом имитации отжига, который повторяет физический процесс медленного охлаждения системы. На каждом шаге генерируется новое состояние $\boldsymbol{\theta}' = \boldsymbol{\theta} + \delta\boldsymbol{\theta}$ с малым случайным возмущением, и принимается по правилу Метрополиса

$$P_{\text{accept}} = \begin{cases} 1, & \text{если } E(\boldsymbol{\theta}') < E(\boldsymbol{\theta}) \\ \exp\left(-\dfrac{E(\boldsymbol{\theta}') - E(\boldsymbol{\theta})}{T}\right), & \text{иначе,} \end{cases} \tag{2.7}$$

с тем отличием от алгоритма Метрополиса, что температура также понижается на каждом шаге. В данной работе температура T постепенно понижается по закону $T_{k+1} = \alpha T_k$, где $0 < \alpha < 1$. Оптимальный выбор

максимальной температуры и коэффициента α производится на основе численных экспериментов.

Реализация основного цикла отжига:

```
def simulated_annealing(
    initial_theta: np.ndarray,
    pauli_operators: List[Any],
    progress: Any,
    task: Any,
    initial_temp: float = 1000.0,
    cooling_rate: float = 0.99,
    min_temp: float = 1e-5,
    num_iterations_per_temp: int = 500,
    step_size: float = 0.5,
) -> Tuple[np.ndarray, float]:
    current_theta = initial_theta.copy()
    best_theta = current_theta.copy()
    best_energy = float("inf")
    rng = np.random.default_rng()
    temp = initial_temp
    thermalization_steps = int(num_iterations_per_temp * 0.2)
    while temp > min_temp:
        # Термализация
        for _ in range(thermalization_steps):
            neighbor_theta = generate_neighbor_theta(current_theta,
               step_size)
            ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
               pauli_operators)
            uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
            current_energy = calculate_expectation(uhu_dict)
            if current_energy < best_energy:
                best_theta = neighbor_theta.copy()
                best_energy = current_energy
            current_theta = neighbor_theta.copy()
            if progress is not None:
                progress.update(task, advance=1)
        # Основной цикл
        for _ in range(num_iterations_per_temp):
            perturbation = rng.normal(0, step_size * (temp /
               initial_temp), current_theta.shape)
            neighbor_theta = (current_theta + perturbation) % (2 *
               np.pi)
            ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
               pauli_operators)
            uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
            current_energy = calculate_expectation(uhu_dict)
            energy_diff = current_energy - best_energy
            if energy_diff < 0 or rng.random() < np.exp(-energy_diff /
               temp):
                current_theta = neighbor_theta.copy()
                if current_energy < best_energy:
```

```
                    best_theta = current_theta.copy()
                    best_energy = current_energy
            if progress is not None:
                progress.update(task, advance=1)
        temp *= cooling_rate
    return best_theta, best_energy
```

2.2.10 Визуализация и логирование результата

Для удобства анализа и отладки программа реализует вывод промежуточных и итоговых результатов в консоль в виде таблиц, панелей и прогресс-баров с помощью библиотеки `rich`, а также ведёт лог-файл всего вывода. В частности, реализованы функции

```
def print_hamiltonian(console, pauli_operators): ...
def print_pauli_table(console, pauli_operators): ...
def print_composition_table(console, pauli_compose, pauli_strings): ...
```

Итоговое значение энергии и параметры оптимального анзаца выводятся в символьном и численном виде.

2.3 Сравнительные результаты тестирования

Вариационные квантовые алгоритмы оказались наиболее эффективными в задачах квантовой химии. В частности, при решении этих задач разработаны методы построения анзацев для многокубитных гамильтонианов [14, 15, 16, 17], а для классической оптимизации целевой функции обычно применяются варианты метода градиентного спуска.

В данной работе для тестирования вариационного квантового алгоритма с оптимизацией по методу отжига в системе из четырех кубитов используется гамильтониан

$$\hat{H}_{OH} = 0.501\,\hat{\sigma}_{1230} - 0.501\,\hat{\sigma}_{2103} - 1.252\,\hat{\sigma}_{0330} - 1.453\,\hat{\sigma}_{2323} + 1.700\,\hat{\sigma}_{1010} + 0.223\,\hat{\sigma}_{1313}, \quad (2.8)$$

взятый из работы [18], в которой изучается спектроскопия основного состояния гидроксильной группы OH^-; гамильтониан переписан в базисе

Паули и редуцирован по нулевому уровню энергии (т.е. учтена только бесследовая часть оператора). Сравнительное тестирование включает в себя две задачи: во-первых, сравнительное тестирование двух методов Монте-Карло выбора нового набора параметров, а именно, метода отжига и метода случайного поиска; во-вторых, сравнительное тестирование анзацев — универсального анзаца, построенного на основе базисных операторов Паули из $\hat{H}_{OH}$, и нескольких "угаданных" пробных анзацев.

Универсальный анзац имеет вид

$$\hat{U}(\boldsymbol{\theta}) = \mathrm{e}^{i\theta_4\hat{\sigma}_{0330}}\mathrm{e}^{i\theta_3\hat{\sigma}_{1313}}\mathrm{e}^{i\theta_2\hat{\sigma}_{2103}}\mathrm{e}^{i\theta_1\hat{\sigma}_{1230}}, \tag{2.9}$$

где $\theta_k \in [0, 2\pi)$, $k = 1, 2, 3, 4$. Он записан на основе операторов Паули из гамильтониана (2.8) за исключением операторов $\hat{\sigma}_{1100}$ $\hat{\sigma}_{2323}$, которые дублируются оператором $\hat{\sigma}_{1313}$. Последний выбран только потому, что он входит в гамильтониан с минимальным коэффициентом.

Испытания пробных одно- и двухпараметрических анзацев требуют очень большого числа вычислительных экспериментов, в которых эти анзацы генерируются случайным образом. В конце концов испытания показывают, что унитарное преобразование из $SU\big(2^4\big)$, переводящее $|0000\rangle$ в основное состояние, принадлежит однопараметрической подгруппе оператора $\hat{\sigma}_{1020} \in \mathfrak{su}\big(2^4\big)$ (или оператора $\hat{\sigma}_{2010}$). А именно,

$$\hat{U}(\theta) = \mathrm{e}^{i\theta\hat{\sigma}_{1020}} = \big(\cos\theta\,\hat{\sigma}_{0000} + i\sin\theta\hat{\sigma}_{1020}\big),$$

так что основное состояние $|\psi\rangle_0 = \hat{U}(\theta)\,|0000\rangle$ достигается при

$$\cos\theta = 0.8209, \quad \sin\theta = 0.5711, \quad |\psi\rangle_0 = 0.8209\,|0000\rangle - 0.5711\,|1010\rangle\,.$$

Для гамильтониана (2.8) легко проверить прямым вычислением, что $\hat{H}_{OH}\,|\psi\rangle_0 = -3.601\,|\psi\rangle_0$, а используя матричное представление данного гамильтониана можно убедиться (с использованием системы Maple), что $E_0 = -3.601$ является энергией основного состояния $|\psi\rangle_0$. Кроме того, также нетрудно найти, что функция энергии $E(\boldsymbol{\theta}) = \langle\psi(\boldsymbol{\theta})|\hat{H}\,|\psi(\boldsymbol{\theta})\rangle$ для анзаца (2.9) имеет двенадцать локальных минимумов.

2.3.1 Сравнительный анализ параметров отжига

Для оценки эффективности вариационного квантового алгоритма с оптимизацией методом имитации отжига были проведены серии тестов с различными наборами параметров отжига. В качестве эталонного значения использовалась энергия основного состояния, равная -3.601. В каждом тесте варьировались следующие параметры: начальная температура (`initial_temp`), множитель охлаждения (`cooling_rate`), минимальная температура (`min_temp`), число итераций на каждой температуре (`num_iterations_per_temp`), а также стандартное отклонение для шума параметров θ (`step_size`).

Полученные результаты позволяют выделить основные закономерности:

- При фиксированных начальной температуре и числе итераций на температуре уменьшение шага $step\_size$ приводит к ухудшению результата: алгоритм хуже исследует пространство параметров и чаще застревает в локальных минимумах. Наилучшие значения энергии наблюдались при $step\_size = 0.1$.

- Увеличение числа итераций на каждой температуре не приводит к принципиальному улучшению результата, хотя небольшое положительное влияние имеет место.

- Повышение начальной температуры (например, до 1000) также не даёт существенного выигрыша, указывая на то, что роль этого параметра в пределах выбранных диапазонов невелика.

- Во всех тестах достигнутое значение энергии оставалось выше эталонного примерно на $0.20 - 0.24$ единицы, что может быть связано с ограничениями самого метода или особенностями выбора анзаца.

Таким образом, можно сделать вывод, что параметры $step\_size$ и $num\_iterations\_per\_temp$ оказывают наибольшее влияние на итоговую энергию, однако даже при их оптимизации достичь эталонного значения не удалось. Это свидетельствует о необходимости дальнейшей доработки схемы оптимизации: возможно, стоит рассмотреть более сложные схемы

охлаждения, адаптивное изменение шага или комбинированные алгоритмы поиска. Полученные результаты подчеркивают важность тонкой настройки параметров имитации отжига для задач вариационной квантовой оптимизации.

Заключение

В работе получены следующие основные результаты:

1. Предложена новая схема вариационного квантового алгоритма, отличающаяся использованием метода имитации отжига в качестве метода оптимизации в вариационной квантовой схеме.

2. Разработан алгоритм классического компьютерного моделирования предложенной схемы вариационной квантовой оптимизации для тестирования эффективности метода отжига для конкретных гамильтонианов.

3. Алгоритм классического моделирования реализован в виде программного модуля на языке Python. Проведён сравнительный анализ различных параметров отжига и анзацев квантовой схемы.

Литература

[1] J. Preskill. *Quantum Computing in the NISQ era and beyond*. Quantum, vol. 2, p. 79, 2018.
(*quantum-journal:q-2018-08-06-79*)

[2] M. Schuld, I. Sinayskiy, F. Petruccione. *An introduction to quantum machine learning*. Contemporary Physics, vol. 56, no. 2, pp. 172-185, 2015.
(*tandfonline:00107514.2014.964942*)

[3] J. Biamonte, et al. *Quantum machine learning*. Nature, vol. 549, pp. 195-202, 2017.
(*nature:nature23474*)

[4] A. W. Harrow, A. Hassidim, and S. Lloyd. *Quantum algorithm for linear systems of equations*. Physical Review Letters, vol. 103, no. 15, p. 150502, 2009.
(*aps:PhysRevLett.103.150502*)

[5] N. Moll, P. Barkoutsos, L. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn, et al. *Quantum optimization using variational algorithms on near-term quantum devices*. Quantum Science and Technology, vol. 3, no. 3, p. 030503, 2018.
(*iopscience:2058-9565/aab822*)

[6] V. Havlicek, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, J. M. Gambetta. *Supervised learning with quantum-enhanced feature spaces*. Nature, vol. 567, pp. 209-212, 2019.
(*nature:s41586-019-0980-2*)

[7] M. Cerezo, et al. *Variational Quantum Algorithms*. Nature Reviews Physics, vol. 3, pp. 625-644, 2021.
(*nature:42254-021-00348-9*)

[8] A. Peruzzo, et al. *A variational eigenvalue solver on a photonic quantum processor*. Nature Communications, vol. 5, p. 4213, 2014.
(*nature:ncomms5213*)

[9] E. Farhi, J. Goldstone, and S. Gutmann. *A Quantum Approximate Optimization Algorithm*. arXiv preprint arXiv:1411.4028, 2014.
(*arXiv:1411.4028*)

[10] P. Salamon, P. Sibani, R. Frost. *Facts, Conjectures, and Improvements for Simulated Annealing*. SIAM: Society for Industrial and Applied Mathematics, Philadelphia, 2002.
(*epubs.siam.org/doi/epdf/10.1137/1.9780898718300*)

[11] A. A. Lopatin. *Метод отжига*. Санкт-Петербургский Государственный Университет, 2005.
(*math.spbu:user/gran/sb1/lopatin*)

[12] Нильсен М., Чанг И. *Квантовые вычисления и квантовая информация*. Пер. с англ - М: Мир, 2006г. - 824 с.
(*djvu.online/file/cO8qDRGAwOIe2*)

[13] V. V. Nikonov, A. N. Tsirulev. *Pauli basis formalism in quantum computations*. Volume 8, No 3, pp. 1 – 14, 2020.
(*doi:10.26456/mmg/2020-831*)

[14] A. Kandala, et al. *Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets*. Nature, vol. 549, pp. 242-246, 2017.
(*nature:nature23879*)

[15] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, M. Head-Gordon. *Simulated Quantum Computation of Molecular Energies*. Science, vol. 309, no. 5741, pp. 1704-1707, 2005.
(*science:1113479*)

[16] A. Daskin, S. Kais. *Decomposition of unitary matrices for finding quantum circuits: Application to molecular Hamiltonians*. The Journal of Chemical Physics, vol. 141, no. 23, p. 234115, 2014.
(*aip:1.4904315*)

[17] J. Romero, R. Babbush, J. R. McClean, C. Hempel, P. J. Love, A. Aspuru-Guzik. *Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz*. Quantum Science and Technology, vol. 4, no. 1, p. 014008, 2018.
(*iopscience:2058-9565/aad3e4*)

[18] N. Cawley, Z. Howard, M. Kleinert et al. *Analytical study of level crossings in the Stark-Zeeman spectrum of ground state OH*. Eur. Phys. J. D, vol. 67, 233, 2013. – M. Bhattacharya. S. Marin, M. Kleinert. *Coherent cancellation of geometric phase for the OH molecule in external fields*, 2014.
(*arXiv.1404.6285*)

Приложение Python

```python
import sys
import io
import numpy as np
from pathlib import Path
from rich.console import Console
from rich.table import Table
from rich.panel import Panel
from rich.progress import Progress, BarColumn, TextColumn, SpinnerColumn
from rich import box
from sympy import re as sp_re, im as sp_im
from functools import lru_cache
from typing import Any, List, Dict, Tuple, Union, Callable


def get_base_path() -> Path:
    """
    Определяет базовую директорию проекта.

    Returns:
        Path: Абсолютный путь к папке с .exe-файлом (если приложение зам
           орожено)
              или к корню проекта (в режиме разработки).

    Примечания:
        - sys.frozen (атрибут пакета PyInstaller) используется для опред
           еления,
          был ли код скомпилирован в исполняемый файл.
        - Это позволяет корректно определять относительные пути при запу
           ске
          как из исходников, так и из собранного .exe.
    """
    if getattr(sys, "frozen", False):
        # Для скомпилированного EXE возвращаем директорию исполняемого ф
           айла.
        return Path(sys.argv[0]).parent
    else:
        # Для разработки возвращаем корень проекта (на уровень выше
           constants/).
        return Path(__file__).parent.parent
```

```python

# Абсолютный путь к файлу с гамильтонианом (описание операторов Паули и их коэффициентов).
HAMILTONIAN_FILE_PATH: Path = get_base_path() / "params" / "hamiltonian_operators.txt"

# Абсолютный путь к файлу для логирования вывода (очищается при запуске).
OUTPUT_FILE_PATH: Path = get_base_path() / "output.log"
# Карта произведения для базисных операторов Паули:
# (i, j) -> (мнимая единица с правильным знаком, индекс результата)
# Индексы: 0=I, 1=X, 2=Y, 3=Z
PAULI_MAP = {
    (1, 2): (1j, 3),  # X*Y = iZ
    (2, 1): (-1j, 3),  # Y*X = -iZ
    (3, 1): (1j, 2),  # Z*X = iY
    (1, 3): (-1j, 2),  # X*Z = -iY
    (2, 3): (1j, 1),  # Y*Z = iX
    (3, 2): (-1j, 1),  # Z*Y = -iX
}


def calculate_temp_steps(
    initial_temp: float, cooling_rate: float, min_temp: float
) -> int:
    """
    Вычисляет количество температурных шагов для метода отжига.

    Алгоритм: на каждом шаге температура уменьшается по формуле:
        T_{n+1} = T_n * cooling_rate
    Шаги считаются, пока температура не станет меньше min_temp.

    Args:
        initial_temp (float): Начальная температура.
        cooling_rate (float): Множитель охлаждения (0 < cooling_rate < 1).
        min_temp (float): Минимально допустимая температура.

    Returns:
        int: Число температурных шагов до достижения min_temp.
    """
    steps = 0
    current_temp = initial_temp
    while current_temp > min_temp:
        current_temp *= cooling_rate
        steps += 1
    return steps


def console_and_print(console: Console, message: Any) -> None:
```

```
    """
    Выводит сообщение в консоль и дублирует его в лог-файл.

    Args:
        console (Console): объект rich.Console для форматированного вывода.
        message (Any): строка, rich.Panel или другой объект, печатаемый в консоль.

    Примечания:
        - Используется rich для красивого форматирования в консоли и логах.
        - Лог хранит весь вывод, включая цветовые коды (если export_text это поддерживает).
    """
    console.print(message)
    with open(OUTPUT_FILE_PATH, "a", encoding="utf-8") as file:
        file.write(console.export_text() + "\n")


def create_table(
    columns: List[Dict[str, str]],
    data: List[List[Any]],
    title: str,
    border_style: str = "yellow",
) -> Panel:
    """
    Создает таблицу rich.Table и оборачивает ее в rich.Panel для консольного вывода.

    Args:
        columns (List[Dict[str, str]]): Описание столбцов (ключи: name, style, justify).
        data (List[List[Any]]): Массив данных для строк таблицы.
        title (str): Заголовок панели.
        border_style (str): Цвет рамки панели.

    Returns:
        Panel: Панель с таблицей для печати в консоль.
    """
    table = Table(box=box.ROUNDED, border_style=border_style)
    for col in columns:
        table.add_column(
            col["name"],
            justify=col.get("justify", "default"),
            style=col.get("style", ""),
        )
    for row in data:
        table.add_row(*row)
    return Panel(table, title=title, border_style=border_style)

```

```python

def format_ansatz(
    pauli_operators: List[Tuple[complex, List[int]]],
    result: Dict[Tuple[int, ...], complex],
) -> Tuple[str, str]:
    """
    Форматирует вариационный анзац в символьное и численное представлени
        е.

    Args:
        pauli_operators (List[Tuple[complex, List[int]]]): Список операт
            оров Паули для анзаца,
            где каждый оператор представлен кортежем из коэффициента и в
                ектора индексов.
        result (Dict[Tuple[int, ...], complex]): Разложение анзаца после
            экспоненцирования
            по базису Паули (ключ: вектор индексов, значение: коэффициен
                т).

    Returns:
        Tuple[str, str]: Строки символьного (произведение экспонент) и ч
            исленного (разложение) представления.

    Символьное представление важно для понимания структуры унитарного оп
        ератора:
        $U(\theta) = \exp(i \cdot \theta_1 \cdot c_1 \cdot \sigma_1) \cdot \exp(i \cdot \theta_2 \cdot c_2 \cdot \sigma_2) \cdot \cdots$

    Численное разложение полезно для анализа конечного состояния операто
        ра:
        $U = \alpha_0 \cdot I + \alpha_1 \cdot \sigma_1 + \alpha_2 \cdot \sigma_2 + \cdots$
    """
    ansatz_symbolic = "U(θ) = " + " * ".join(
        [
            f"exp(i·θ_{i+1}·{format_complex_number(coeff)}·
            σ_{''.join(map(str, op))})"
            for i, (coeff, op) in enumerate(pauli_operators)
        ]
    )
    ansatz_numeric = "U = " + " + ".join(
        [
            f"{format_complex_number(c)}·σ_{''.join(map(str, op))}"
            for op, c in result.items()
            if abs(c) > 1e-12
        ]
    )
    return ansatz_symbolic, ansatz_numeric


def format_complex_number(c) -> str:
    """
```

```
    Форматирует комплексное число (или выражение SymPy) в строку с подавлением артефактов.

    Args:
        c (complex|sympy.Expr): Комплексное число или выражение.

    Returns:
        str: Отформатированное комплексное число (пример: '1.2-3i').

    Особенности:
        - Если число очень близко к нулю (<1e-12), оно не выводится.
        - Мнимая часть ±1 форматируется как ±i.
        - Корректно обрабатывает как стандартные числа, так и объекты SymPy.
    """
    real = float(sp_re(c))
    imag = float(sp_im(c))

    real_str = format_number(real) if abs(real) > 1e-12 else ""
    imag_str = ""

    if abs(imag) > 1e-12:
        abs_imag = abs(imag)
        imag_value = format_number(abs_imag)
        if imag_value == "1":
            imag_str = "i" if imag > 0 else "-i"
        else:
            imag_sign = "" if imag > 0 else "-"
            imag_str = f"{imag_sign}{imag_value}i"

    parts = []
    if real_str:
        parts.append(real_str)
    if imag_str:
        parts.append(imag_str)

    if not parts:
        return "0"

    result = parts[0]
    for part in parts[1:]:
        if part.startswith("-"):
            result += f"-{part[1:]}"
        else:
            result += f"+{part}"

    return result.replace("+ -", "- ").replace("1i", "i").replace(".0i", "i")


def format_number(num: float | int) -> str:
```

```
    """
    Форматирует число для вывода, корректно подавляя артефакты округлени
        я.

    Args:
        num (float|int): Число для форматирования.

    Returns:
        str: Число в виде строки, без лишних нулей и ошибок округления.
    """
    if abs(num - round(num)) < 1e-15:
        return str(int(round(num)))
    s = f"{num:.14f}".rstrip("0").rstrip(".")
    if s.startswith("."):
        s = "0" + s
    elif s.startswith("-."):
        s = s.replace("-.", "-0.")
    if "." in s:
        int_part, dec_part = s.split(".")
        dec_part = dec_part[:4].ljust(4, "0").rstrip("0")
        s = f"{int_part}.{dec_part}" if dec_part else int_part
    return s


def get_operator_for_console(c: Union[complex, float, int], i: str) ->
    str:
    """
    Формирует строку для красивого вывода оператора Паули.

    Args:
        c (complex|float|int): Коэффициент перед оператором.
        i (str): Индексная строка оператора (например, '03' для $I \otimes Z$).

    Returns:
        str: Строка вида '$\sigma_i$' или'$c \cdot \sigma_i$'
        (если коэффициент не равен 1).
    """
    if c == 1:
        return f"$\sigma_{i}$"
    else:
        return f"{format_complex_number(c)}*$\sigma_{i}$"


def initialize_environment() -> Console:
    """
    Инициализирует окружение для запуска программы:
    - Очищает лог-файл с прошлых запусков.
    - Возвращает объект rich.Console для форматированного вывода.

    Returns:
        Console: Готовый к использованию rich.Console.
```

```
    """
    if OUTPUT_FILE_PATH.exists():
        OUTPUT_FILE_PATH.unlink()
    return Console(force_terminal=True, color_system="truecolor",
       record=True)


def print_composition_table(
    console: Console,
    pauli_compose: Callable[[tuple, tuple], Tuple[complex, tuple]],
    pauli_strings: List[List[int]],
) -> None:
    """
    Выводит таблицу композиции операторов Паули для всех их пар.

    Args:
        console (Console): Объект rich.Console.
        pauli_compose (Callable): Функция для композиции двух операторов
            Паули.
        pauli_strings (List[List[int]]): Список векторных индексов опера
           торов Паули.
    """
    results = []
    # Перебор всех пар операторов Паули
    for s1 in pauli_strings:
        for s2 in pauli_strings:
            coeff, product = pauli_compose(tuple(s1), tuple(s2))
            results.append((s1, s2, format_complex_number(coeff),
               product))

    table_data = [
        [str(s1), str(s2), str(h).lower(), str(p)] for s1, s2, h, p in
           results
    ]
    console_and_print(
        console,
        create_table(
            columns=[
                {"name": "Оператор 1", "style": "cyan", "justify":
                   "center"},
                {"name": "Оператор 2", "style": "magenta", "justify":
                   "center"},
                {"name": "Коэффициент", "style": "green", "justify":
                   "center"},
                {"name": "Результат", "style": "red", "justify":
                   "center"},
            ],
            data=table_data,
            title="Композиции операторов Паули",
            border_style="green",
        ),
```

```
    )


def print_hamiltonian(
    console: Console, pauli_operators: List[Tuple[complex, List[int]]]
) -> None:
    """
    Выводит гамильтониан в удобочитаемом виде.

    Args:
        console (Console): rich.Console для вывода.
        pauli_operators (List[Tuple[complex, List[int]]]): Список операторов Паули с коэффициентами.
    """
    hamiltonian_str = "H = " + " + ".join(
        [get_operator_for_console(c, "".join(map(str, i))) for c, i in pauli_operators]
    )
    console_and_print(
        console,
        Panel(
            hamiltonian_str,
            title="[bold]Введенный гамильтониан[/bold]",
            border_style="green",
        ),
    )


def print_pauli_table(
    console: Console, pauli_operators: List[Tuple[complex, List[int]]]
) -> None:
    """
    Выводит таблицу всех операторов Паули из гамильтониана.

    Args:
        console (Console): rich.Console для вывода.
        pauli_operators (List[Tuple[complex, List[int]]]): Список операторов Паули с коэффициентами.
    """
    table_data = [[format_complex_number(c), str(i)] for c, i in pauli_operators]
    console_and_print(
        console,
        create_table(
            columns=[
                {"name": "Коэффициент", "style": "cyan"},
                {"name": "Индекс", "style": "magenta", "justify": "center"},
            ],
            data=table_data,
            title="Операторы Паули",
```

```
            border_style="purple",
        ),
    )


from pathlib import Path
from typing import List, Union


def read_file_lines(file_path: Union[str, Path], ignore_comments: bool) -> List[str]:
    """
    Считывает строки из файла, игнорируя комментарии (начинающиеся с '#').

    Args:
        file_path (str|Path): Путь к файлу.
        ignore_comments (bool): Если True, строки, начинающиеся с '#', игнорируются.

    Returns:
        List[str]: Список строк без лишних пробелов и пустых строк.

    Raises:
        FileNotFoundError: Если файл не существует.
    """
    file_path = Path(file_path) if not isinstance(file_path, Path) else file_path
    if not file_path.exists():
        raise FileNotFoundError(f"Файл {file_path} не найден.")
    with open(file_path, "r") as file:
        return [
            line.strip()
            for line in file
            if not (ignore_comments and line.strip().startswith("#"))
        ]


def read_hamiltonian_data(
    file_path,
) -> Tuple[List[Tuple[complex, List[int]]], List[List[int]]]:
    """
    Читает список операторов Паули из текстового файла.

    Формат файла:
        <действительная часть> <мнимая часть> <строка Паули>

    Returns:
        Tuple[List[Tuple[complex, List[int]]], List[List[int]]]:
            - Список операторов (коэффициент, индексы Паули)
            - Список только индексов (без коэффициентов)
```

```
    """
    lines = read_file_lines(file_path, ignore_comments=False)
    pauli_operators: List[Tuple[complex, List[int]]] = []
    pauli_strings: List[List[int]] = []
    for line in lines:
        parts = line.strip().split()
        if len(parts) == 3:
            real_part, imag_part, index_str = (
                float(parts[0]),
                float(parts[1]),
                str(parts[2]),
            )
            coefficient = np.complex128(real_part + imag_part * 1j)
            index_list = [int(c) for c in index_str]
            if coefficient != 0:
                pauli_operators.append((coefficient, index_list))
            pauli_strings.append(index_list)
    return pauli_operators, pauli_strings


def calculate_ansatz(
    theta: np.ndarray, pauli_operators: List[Tuple[complex, List[int]]]
) -> Tuple[Dict[Tuple[int, ...], complex], str, str]:
    """
    Вычисляет вариационный анзац в виде произведения экспонент операторов Паули.

    Args:
        theta (np.ndarray): Вектор параметров (обычно одного размера с числом операторов).
        pauli_operators (List[Tuple[complex, List[int]]]): Операторы Паули с коэффициентами.

    Returns:
        Tuple[
            Dict[Tuple[int, ...], complex],   # Разложение анзаца по Паули-операторам
            str,                              # Символьное представление (произведение экспонент)
            str                               # Численное разложение
        ]

    Алгоритм:
        U(θ) = prod_j exp(i · θ_j · |c_j| · σ_j)
        Реализуется по принципу покомпонентного разложения через формулу Эйлера:
            exp(i · α · σ) = cos(α) · I + i · sin(α) · σ
        С каждым новым оператором результат рекурсивно обновляется через pauli_compose.
    """
    operator_length = len(pauli_operators[0][1])
```

```
    result: Dict[Tuple[int, ...], complex] = {tuple([0] *
        operator_length): 1.0}

    for t, (coeff, op) in zip(theta, pauli_operators):
        angle = t * abs(coeff)  # Используем абсолютное значение коэффиц
            иента!
        cos_t = np.cos(angle)
        sin_t = np.sin(angle)
        new_result: Dict[Tuple[int, ...], complex] = {}
        op_tuple = tuple(op)
        for existing_op, existing_coeff in result.items():
            # cos(angle) * I (сохраняем индекс базисного оператора)
            new_result[existing_op] = (
                new_result.get(existing_op, 0) + existing_coeff * cos_t
            )
            # i*sin(angle)*σ (композиция Паули)
            compose_coeff, compose_op = pauli_compose(existing_op,
                op_tuple)
            final_coeff = existing_coeff * 1j * sin_t * compose_coeff
            new_result[compose_op] = new_result.get(compose_op, 0) +
                final_coeff
        result = new_result

    symbolic_str, numeric_str = format_ansatz(pauli_operators, result)
    return result, symbolic_str, numeric_str


from typing import Tuple, Dict


def calculate_expectation(uhu_dict: Dict[Tuple[int, ...], complex]) ->
    float:
    """
    Вычисляет среднее значение энергии ⟨0|U†HU|0⟩|0⋯0⟩.

    Args:
        uhu_dict (Dict[Tuple[int, ...], complex]): Разложение оператора
            U†HU
            по базису Паули (ключ: индекс, значение: коэффициент).

    Returns:
        float: Ожидаемое значение (энергия).

    Примечание:
        - Только те операторы, которые не изменяют |0⋯0⟩
          (то есть содержащие только I и Z),
          могут дать нетривиальный вклад в среднее значение.
    """
    expectation = 0.0
    for op, coeff in uhu_dict.items():
```

```
489         if all(p in (0, 3) for p in op):  # Только I или Z на каждом куб
                ите
490             expectation += coeff.real
491     return expectation
492
493
494 def compute_uhu(
495     u_dict: Dict[Tuple[int, ...], complex], h_terms:
            List[Tuple[complex, List[int]]]
496 ) -> Dict[Tuple[int, ...], complex]:
497     """
498     Вычисляет оператор U†HU в базисе Паули.
499
500     Args:
501         u_dict (Dict[Tuple[int, ...], complex]): Разложение оператора U.
502         h_terms (List[Tuple[complex, List[int]]]): Операторы гамильтониа
                на с коэффициентами.
503
504     Returns:
505         Dict[Tuple[int, ...], complex]: Разложение U†HU
506           по Паули операторам.
507
508     Алгоритм:
509         (U†HU)_{kl} = sum_{i,j} conj(U_{ik}) * H_{ij} * U_{jl}
510         Реализовано через перебор всех Паули-операторов.
511     """
512     uhu_dict: Dict[Tuple[int, ...], complex] = {}
513     u_items = list(u_dict.items())
514     for coeff_h, op_h in h_terms:
515         op_h_tuple = tuple(op_h)
516         for j_op, j_coeff in u_items:
517             conj_j_coeff = np.conj(j_coeff)
518             c1, op_uh = pauli_compose(j_op, op_h_tuple)
519             for k_op, k_coeff in u_items:
520                 c2, op_uhu = pauli_compose(op_uh, k_op)
521                 total_coeff = conj_j_coeff * k_coeff * coeff_h * c1 * c2
522                 uhu_dict[op_uhu] = uhu_dict.get(op_uhu, 0) + total_coeff
523     return uhu_dict
524
525
526 def generate_neighbor_theta(
527     current_theta: np.ndarray, step_size: float = 0.1
528 ) -> np.ndarray:
529     """
530     Генерирует новое состояние θ, добавляя нормальный шум
531     с заданной дисперсией, и приводит все значения к диапазону [0, 2π).
532
533     Args:
534         current_theta (np.ndarray): Текущий вектор параметров.
535         step_size (float): Стандартное отклонение для гауссового шума.
536
```

```
    Returns:
        np.ndarray: Новый вектор параметров.
    """
    perturbation = np.random.normal(scale=step_size,
       size=current_theta.shape)
    return (current_theta + perturbation) % (2 * np.pi)


def generate_shifted_theta(
    pauli_operators: List[Tuple[complex, List[int]]]
) -> np.ndarray:
    """
    Генерирует начальный вектор θ для анзаца, масштабируя его пропорцион
       ально
    модулям коэффициентов операторов Паули.

    Args:
        pauli_operators (List[Tuple[complex, List[int]]]): Операторы Пау
           ли.

    Returns:
        np.ndarray: Начальный вектор θ (размер соответствует числу опера
           торов).
    """
    if not pauli_operators:
        return np.array([], dtype=np.float64)
    # Используем модуль коэффициента (абсолютная величина, чтобы избежат
       ь ошибок для комплексных коэффициентов)
    coeffs = np.array([abs(op[0]) for op in pauli_operators],
       dtype=np.float64)
    norm = np.linalg.norm(coeffs)
    if norm < 1e-12:
        return np.zeros(len(coeffs))
    # Масштабируем на диапазон [0, 2π) и добавляем небольшой случайный ш
       ум
    scaled = (coeffs / norm) * 2 * np.pi
    return scaled + np.random.normal(0, 0.1, len(scaled))


def multiply_pauli(i: int, j: int) -> Tuple[complex, int]:
    """
    Перемножает два базисных оператора Паули.

    Args:
        i (int): Первый индекс (0=I, 1=X, 2=Y, 3=Z).
        j (int): Второй индекс (0=I, 1=X, 2=Y, 3=Z).

    Returns:
        Tuple[complex, int]: Коэффициент и индекс результата.
    """
    if i == j:
```

```
        return (1, 0)
    if i == 0:
        return (1, j)
    if j == 0:
        return (1, i)
    return PAULI_MAP.get((i, j), (1, 0))


@lru_cache(maxsize=4096)
def pauli_compose(s1: tuple, s2: tuple) -> Tuple[complex, tuple]:
    """
    Перемножает два оператора Паули, заданных покубитно.

    Args:
        s1 (tuple): Индексы первого оператора (например, (0,3)
            для I⊗Z).
        s2 (tuple): Индексы второго оператора.

    Returns:
        Tuple[complex, tuple]: Коэффициент и индексы результата.
    """
    coefficient = 1.0
    result = []
    for a, b in zip(s1, s2):
        coeff, idx = multiply_pauli(a, b)
        coefficient *= coeff
        result.append(idx)
    return coefficient, tuple(result)


def simulated_annealing(
    initial_theta: np.ndarray,
    pauli_operators: List[Any],
    progress: Any,
    task: Any,
    initial_temp: float = 1000.0,
    cooling_rate: float = 0.99,
    min_temp: float = 1e-5,
    num_iterations_per_temp: int = 500,
    step_size: float = 0.5,
) -> Tuple[np.ndarray, float]:
    """
    Алгоритм отжига (simulated annealing) для оптимизации параметров θ в
       ариационного анзаца.

    Args:
        initial_theta (np.ndarray): Начальный вектор θ.
        pauli_operators (List[Tuple[complex, List[int]]]): Операторы Пау
           ли.
        progress (Any): Индикатор прогресса (может быть None).
        task (Any): Задача для прогресс-бара.
```

```
        initial_temp (float): Начальная температура (чем выше - тем вероятнее принять ухудшающее решение).
        cooling_rate (float): Множитель охлаждения (0 < cooling_rate < 1).
        min_temp (float): Минимально допустимая температура.
        num_iterations_per_temp (int): Количество шагов на каждой температуре.
        step_size (float): Стандартное отклонение для шума θ.

    Returns:
        Tuple[np.ndarray, float]: Оптимальный найденный θ и соответствующая энергия.

    Принцип работы:
        - На каждом шаге генерируется новое состояние θ (случайным образом).
        - Если энергия уменьшилась -- принимаем новое состояние.
        - Если энергия увеличилась -- принимаем с вероятностью
          exp(−△E/T).
        - Температура постепенно понижается (охлаждение).

    Важно:
        - В термализации используется локальный случайный шаг (как и в основном цикле).
        - Это обеспечивает более 'физичное' поведение отжига.
    """
    current_theta = initial_theta.copy()
    best_theta = current_theta.copy()
    best_energy = float("inf")
    rng = np.random.default_rng()
        temp = initial_temp
    thermalization_steps = int(num_iterations_per_temp * 0.2)

        while temp > min_temp:
        # Этап термализации: локальные случайные шаги для прогрева цепочки
        for _ in range(thermalization_steps):
            neighbor_theta = generate_neighbor_theta(current_theta, step_size)
            ansatz_dict, _, _ = calculate_ansatz(neighbor_theta, pauli_operators)
            uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
            current_energy = calculate_expectation(uhu_dict)
            if current_energy < best_energy:
                best_theta = neighbor_theta.copy()
                best_energy = current_energy
            current_theta = neighbor_theta.copy()
            if progress is not None:
                progress.update(task, advance=1)

        # Основной цикл отжига с возможностью принимать ухудшения
```

```
671             for _ in range(num_iterations_per_temp):
672             perturbation = rng.normal(
673                 0, step_size * (temp / initial_temp),
                        current_theta.shape
674             )
675             neighbor_theta = (current_theta + perturbation) % (2 *
                    np.pi)
676                 ansatz_dict, _, _ = calculate_ansatz(neighbor_theta,
                        pauli_operators)
677                 uhu_dict = compute_uhu(ansatz_dict, pauli_operators)
678                 current_energy = calculate_expectation(uhu_dict)
679                 energy_diff = current_energy - best_energy
680             # Классическое правило Метрополиса
681                 if energy_diff < 0 or rng.random() <
                        np.exp(-energy_diff / temp):
682                     current_theta = neighbor_theta.copy()
683                     if current_energy < best_energy:
684                         best_theta = current_theta.copy()
685                         best_energy = current_energy
686             if progress is not None:
687                     progress.update(task, advance=1)
688             temp *= cooling_rate
689
690     return best_theta, best_energy
691
692
693 # Установка корректной кодировки стандартных потоков для поддержки
        Unicode
694 sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding="utf-8")
695 sys.stderr = io.TextIOWrapper(sys.stderr.buffer, encoding="utf-8")
696
697
698 def main() -> None:
699     """
700     Основная функция программы. Реализует следующий цикл:
701       1. Чтение гамильтониана из файла.
702       2. Вывод операторов Паули и их композиции.
703       3. Оптимизация анзаца методом отжига для последовательных подмноже
              ств операторов.
704       4. Поиск минимальной энергии и вывод оптимального анзаца.
705
706     Программа построена как демонстрация вариационного квантового алгори
          тма
707     с классическим оптимизатором (simulated annealing).
708     """
709     console = initialize_environment()
710
711     # Проверка наличия файла с гамильтонианом
712     if not HAMILTONIAN_FILE_PATH.exists():
713         msg = (
714             f"Файл [bold]{HAMILTONIAN_FILE_PATH}[/] не найден!\n"
```

```
715             "Убедитесь, что рядом с EXE есть папка [bold]params[/] с файлом [bold]hamiltonian_operators.txt[/]."
716         )
717         console_and_print(console, Panel(msg, border_style="red"))
718         return
719 
720     try:
721         pauli_operators, pauli_strings = read_hamiltonian_data(HAMILTONIAN_FILE_PATH)
722         print_hamiltonian(console, pauli_operators)
723         print_pauli_table(console, pauli_operators)
724         print_composition_table(console, pauli_compose, pauli_strings)
725     except FileNotFoundError:
726         console_and_print(
727             console,
728             Panel(
729                 f"[red]Файл {HAMILTONIAN_FILE_PATH} не найден[/red]", border_style="red"
730             ),
731         )
732         return
733 
734     if len(pauli_operators) < 2:
735         console_and_print(
736             console,
737             Panel("[red]Требуется минимум 2 оператора Паули[/red]", border_style="red"),
738         )
739         return
740 
741     # Параметры отжига
742     SA_PARAMS = {
743         "initial_temp": 100.0,
744         "cooling_rate": 0.95,
745         "min_temp": 1e-3,
746         "num_iterations_per_temp": 100,
747         "step_size": 0.1,
748     }
749 
750     # Оценка общего количества шагов для прогресс-бара
751     thermalization_steps = int(SA_PARAMS["num_iterations_per_temp"] * 0.2)
752     temp_steps = calculate_temp_steps(
753         SA_PARAMS["initial_temp"], SA_PARAMS["cooling_rate"], SA_PARAMS["min_temp"]
754     )
755     steps_per_m = temp_steps * (
756         thermalization_steps + SA_PARAMS["num_iterations_per_temp"]
757     )
758     total_steps = steps_per_m * (len(pauli_operators) - 1)
759 
```

```
    best_energy = float("inf")
    best_result = None
    all_results = []

    # Запуск прогресс-бара с симпатичным оформлением
    with Progress(
        SpinnerColumn(),
        TextColumn("[progress.description]{task.description}"),
        BarColumn(bar_width=None),
        TextColumn("[progress.percentage]{task.percentage:>3.0f}%"),
    ) as progress:
        task = progress.add_task("[cyan]Отжиг...", total=total_steps)

        # Последовательно увеличиваем число операторов в анзаце
    for m in range(2, len(pauli_operators) + 1):
            current_ops = pauli_operators[:m]
            initial_theta = generate_shifted_theta(current_ops)

            # Оптимизация параметров для текущего поднабора операторов
        optimized_theta, energy = simulated_annealing(
            initial_theta=initial_theta,
                pauli_operators=current_ops,
                progress=progress,
                task=task,
                **SA_PARAMS,
        )

        all_results.append(
            {
                "m": m,
                "theta": optimized_theta,
                "energy": energy,
                "operators": current_ops,
            }
        )

        if energy < best_energy:
            best_energy = energy
            best_result = all_results[-1]

    # Выводим результаты оптимизации
    if best_result is None:
        console_and_print(
            console, Panel("[red]Не удалось найти решение[/red]",
               border_style="red")
        )
        return

    _, ansatz_symbolic, ansatz_numeric = calculate_ansatz(
        best_result["theta"], best_result["operators"]
    )
```

```
810
811     console_and_print(
812         console,
813         Panel(
814             ansatz_symbolic,
815             title="[bold]Символьное представление анзаца[/]",
816             border_style="green",
817         ),
818     )
819
820     console_and_print(
821         console,
822         Panel(
823             ansatz_numeric,
824             title="[bold]Численное представление анзаца[/]",
825             border_style="purple",
826         ),
827     )
828
829     console_and_print(
830         console,
831         Panel(
832             f"{best_result['energy']:.6f}",
833             title="[bold]Энергия (<0|U†HU|0> для состояния |0...0>)[/]",
834             border_style="green",
835         ),
836     )
837
838     input("Нажмите Enter для выхода...")
839
840
841 if __name__ == "__main__":
842     main()
```