АННОТАЦИЯ

Цель работы — разработать вариационный квантовый алгоритм для задачи коммивояжёра. Полученные результаты ??? Рекомендации на основе данной работы ??

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Квантовый компьютер манипулирует кубитами. Состоящие из кубитов компьютерные регистры могут находиться в суперпозиции нескольких состояний. Когда квантовый компьютер производит вычисление над суперпозицией, он производит вычисление над всеми состояниями одновременно. Это делает квантовые компьютеры привлекательными для задач, в которых требуется перебор большого числа вариантов.

В этой работе мы попытаемся решить задачу коммивояжёра.

Будем искать решение в классе вариационных квантовых алгоритмов. В них квантовый компьютер запускается снова и снова с небольшими изменениями в параметрах квантовой цепи. Процессом варьирования управляет классический компьютер. Его задача — подобрать параметры цепи такие, чтобы на выходе получалось оптимальное по некоторому заданному критерию состояние.

Уже разработаны вариационные квантовые алгоритмы для задач из нескольких разных областей. Среди них — алгоритмы поиска собственных значений матрицы, максимального разреза графа и другие. Они позволяют решать задачи на существующих квантовых компьютерах — подверженных шумам и оперирующим малым числом кубитов.

Для задачи коммивояжёра, которой посвящена данная работа, эффективного квантового алгоритма пока не существует.

Особенностью этой задачи является наличие ограничений: например, коммивояжёр не может посетить один город дважды. Традиционно ограничения в вариационных квантовых алгоритмах реализуются через дополнительные штрафные слагаемые к оптимизируемой величине. Но мы вместо этого попробуем добиться того, чтобы состояние, не удовлетворяющее ограничениям, в принципе невозможно было измерить.

Такая реализация ограничений задачи непосредственно через саму квантовую цепь — главное направление этой работы, в этом заключается её новизна и возможная научная значимость.

Настоящие квантовые компьютеры всё ещё не широко доступны, поэтому мы тестируем алгоритм с помощью компьютерной симуляции. Для этого мы используем `cirq` — библиотеку для Python от Google Quantum AI.

1 Классические подходы

1.1 Постановка задачи коммивояжёра

В задаче коммивояжёра есть n городов, соединённых друг с другом, и требуется найти кратчайший маршрут, проходящий через все города по одному разу.

Существует несколько вариаций задачи. Маршрут может быть замкнутым или незамкнутым, расстояния между городами могут подчиняться или не подчиняться неравенству треугольника; граф городов может быть полным или неполным, ориентированным или неориентированным. Все эти разновидности можно свести друг к другу, поэтому мы можем выбрать ту формулировку, которая кажется наиболее удобной.

Итак, пусть каждый город соединён с каждым (т.е. граф городов полный), пусть мы ищем незамкнутый маршрут и пусть граф городов ориентированный. Ориентированность означает, что условная длина (или стоимость) пути из города A в город B необязательно равна стоимости пути из B в A.

Введём обозначение

$$l_{ij} - \text{ длина пути из } i\text{-того города в } j\text{-тый}$$

Ответом на задачу служит последовательность городов. Есть n способов выбрать первый город, $n-1$ способ выбрать второй и так далее — всего $n!$ возможных ответов. Если бы мы выбрали другую формулировку задачи, то пришлось бы, например, учитывать, что в замкнутом пути начальную точку можно выбрать несколькими способами.

Простейший алгоритм — перебрать все возможные перестановки — будет иметь сложность $O(n!)$ по времени и $O(1)$ по памяти.

1.2 Алгоритм Хелда-Карпа

Альтернативой является алгоритм Хелда-Карпа, который имеет сложность $O(n^2 2^n)$ по времени и $O(n2^n)$ по памяти. Это лучший классический алгоритм из тех, которые гарантированно приходят к точному решению.[1] Рассмотрим его подробнее, чтобы лучше понять, с чем мы соревнуемся.

[1] Ещё есть эвристические приближённые алгоритмы

Для нашей постановки задачи алгоритм выглядит следующим образом:

1) Создаётся мнимый нулевой город, связанный со всеми остальными городами дорогами длины 0.

2) Определяется рекурсивная функция $g(S, e)$, возвращающая длину наименьшего пути, начинающегося в городе 0, проходящего через каждый город некоторого подмножества городов S и заканчивающегося в городе $e \notin S$.

3) Эта функция вычисляется следующим образом:

$$g(S, e) = \min_{e' \in S} \left\{ g(S \backslash e',\ e') + l_{e'e} \right\}$$

(через перебор всех возможных вариантов для предпоследнего города e')

4) Существует 2^n возможных множеств S. Если последовательно вычислять $g(S, e)$ от меньших множеств к большим, перебирая все $e \notin S$, то мы сможем использовать уже вычисленные значения g, и «входить в рекурсию» не потребуется. Функция g будет вызвана $O(n2^n)$ раз, и на хранение результатов потребуется $O(n2^n)$ памяти.

5) При каждом вызове функции $g(S, e)$ происходит перебор $O(n)$ возможных предпоследних вершин e', поэтому сложность функции g — $O(n)$, а сложность алгоритма в целом — $O(n^2 2^n)$.

6) В конце вычисляется $g(S_{\text{all}}, 0)$, где S_{all} — множество всех городов. Это и будет длиной искомого минимального пути.

7) Можно восстановить и сам оптимальный путь, а не только его длину, если в ходе решения помимо промежуточных значений функции $g(S, e)$ сохранять номера оптимальных предпоследних вершин e'. Это не повлияет на асимптотику алгоритма.

Алгоритм Хелда-Карпа работает быстрее, чем простой перебор, но требует экспоненциально много памяти.

2 Описание алгоритма

На рисунке 2.1 приведена общая схема нашего алгоритма. В начале некоторыми практически произвольными параметрическими гейтами создаётся суперпозиция нескольких возможных маршрутов. Затем кубиты измеряются. Это происходит несколько раз, и на классическом компьютере вычисляется средняя стоимость маршрута.

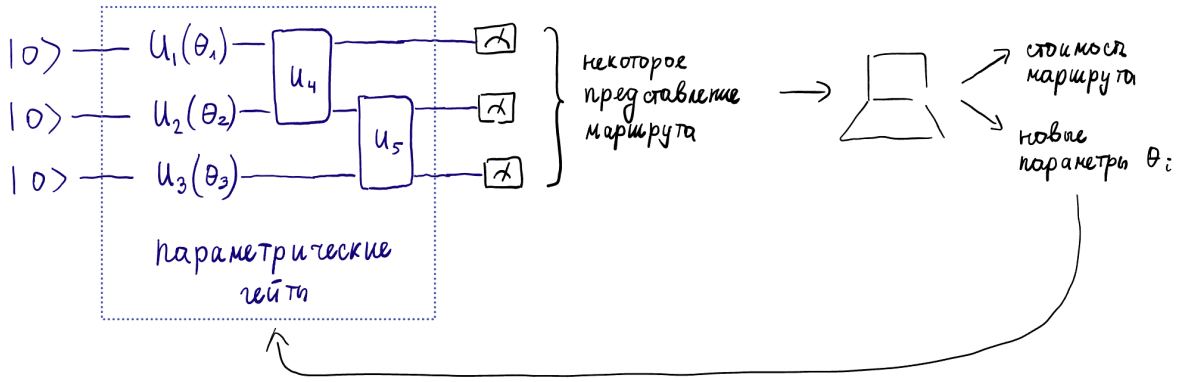

Рисунок 2.1 — Общая схема нашего алгоритма

Основываясь на средней стоимости, мы обновляем параметрические гейты и запускаем цепь заново в надежде, что средняя стоимость уменьшится. Здесь мы можем следовать любому численному алгоритму нахождения минимума функции. Лучше всего себя показал алгоритм Rotosolve (Ostaszewski et al, **rotosolve**), который опирается на то, что все квантовые гейты — унитарные, и, следовательно, функция $\langle \text{cost} \rangle = f(\theta_1, \theta_2 \dots)$ (зависимость средней стоимости от параметров гейтов) не совсем произвольная.

Квантовый компьютер нужен для того, чтобы преобразовать пространство поиска. Исходная задача была дискретной, но, добавив «прослойку» в виде квантового компьютера, мы преобразовали её к непрерывной. Это позволяет применять всё многообразие градиентных и неградиентных методов оптимизации, в том числе выбранный нами алгоритм Rotosolve.

Рассмотрим теперь детали нашего решения.

2.1 Представление маршрутов на квантовом компьютере

Коротко: все возможные маршруты можно занумеровать в лексикографическом порядке. Этот номер кодируется на квантовом компьютере, для чего требуется $\lceil \log_2 n! \rceil$ кубитов. В конце схемы кубиты измеряются, и на

классическом компьютере номер маршрута переводится в факториальную систему счисления, затем вычисляется сам маршрут и его стоимость.

Подробно:

Мы выбрали вариант задачи коммивояжёра, в котором требуется найти незамкнутый маршрут в полном ориентированном графе. Корректным маршрутом является любая последовательность посещённых городов, в которой каждый город встречается по одному разу — то есть любая из $n!$ перестановок городов. В нашей формулировке задачи между маршрутами и перестановками есть взаимно однозначное соответствие; мы можем использовать эти слова как синонимы.

Присвоим городам номера от 0 до $n - 1$.

Перестановки можно строить с помощью дерева (рис. 2.2): в начале мы выбираем первый город из n вариантов, затем с каждым следующим городом количество ветвей уменьшается на 1. Ветви у каждого узла расположены по возрастанию. Перестановки, соответствующие листьям дерева, будут расположены тогда в лексикографическом порядке.

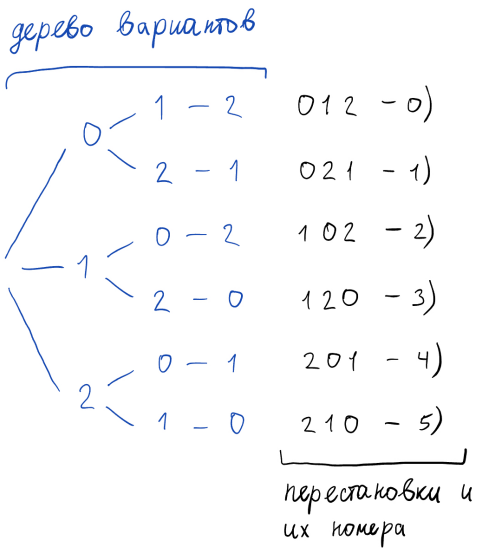

Рисунок 2.2 — Все возможные маршруты для случая трёх городов в лексикографическом (т.е. «алфавитном») порядке.

Поскольку перестановки теперь упорядочены, мы можем их пронумеровать. Номер перестановки удобно записывать в факториальной системе счисления, поскольку эта система отражает структуру дерева вариантов.

Обсудим факториальную систему счисления. В этой системе последняя

цифра должна быть из множества $\{0, 1\}$, вторая с конца — из $\{0, 1, 2\}$, третья — из $\{0, 1, 2, 3\}$ и так далее. Если в десятичной системе цифры умножаются на числа 1, 10, 10^2 и так далее, то в факториальной — на числа 1!, 2!, 3!. . . :

$$\overline{\ldots cba}_! = a \cdot 1! + b \cdot 2! + c \cdot 3! \ldots$$

Максимальное число из $n - 1$ цифр равно $n! - 1$.

Если записать номер перестановки в факториальной системе счисления, то последовательность его цифр можно будет интерпретировать как последовательность выборов в дереве вариантов с рисунка 2.2. В самом деле, первая цифра в $n-1-$значном числе соответствует выбору из n вариантов, вторая — выбору из $n - 1$ варианта, последняя — выбору из двух вариантов. Взяв число

$$a = \overline{a_1 a_2 \ldots a_{n-1}}_!$$

двигаясь по дереву и выбирая на k-том шаге ветвь номер a_k, мы придём в конце к перестановке номер a.

Таким образом, имея номер перестановки в факториальной системе счисления, мы можем цифра за цифрой восстановить по нему саму перестановку. Хранить факториально большое дерево для этого не нужно: доступные ветви на каждом шаге определяются ещё не использованными городами.

Вышесказанное позволяет в качестве представления маршрута использовать его номер $c \in [0, n!)$. Любое целое число, попадающее в этот промежуток, является валидным маршрутом.

Такое представление выгодно отличается от представления маршрута через его рёбра, которое использовалось в других работах (см. раздел «Обзор литературы»). Наше представление автоматически учитывает нетривиальные ограничения задачи, касающиеся того, чтобы маршрут проходил через все города и не распадался на несвязанные друг с другом петли.

На квантовом компьютере для хранения номера маршрута мы будем использовать двоичную систему счисления, что потребует $\lceil \log_2 n! \rceil = O(n \log n)$ кубитов.

Хотелось бы построить такую схему, которая генерирует только валидные номера перестановок — то есть из промежутка $[0, n!)$. Однако мы не будем этим

себя утруждать и позволим схеме генерировать все номера $b \in [0,\ 2^{\lceil \log_2 n! \rceil})$, которые могут быть записаны на имеющемся наборе кубитов.

Слишком большие номера перестановок мы реинтерпретируем на классическом компьютере как номера из допустимого множества по формуле

$$c = b \ \% \ n!$$

После измерения мы переводим номер перестановки из двоичной системы в факториальную, используя обычный алгоритм перевода между системами счисления, основанный на делении с остатком.

Затем на классическом компьютере мы вычисляем сам маршрут, а потом, зная маршрут, вычисляем его стоимость. Этот процесс повторяется достаточное количество раз, чтобы вычислить среднюю стоимость состояния, которое в общем случае представляет собой суперпозицию разных маршрутов.

2.2 Квантовая цепь

На рис. 2.3 изображена использованная нами квантовая схема. Она состоит из слоя гейтов Rx, за которым идёт слой гейтов CNOT. В схему можно добавить ещё несколько таких пар слоёв, но мы не стали этого делать. Гейты поворота Rx имеют параметры: углы θ_i. Обсудим, почему мы выбрали эту схему.

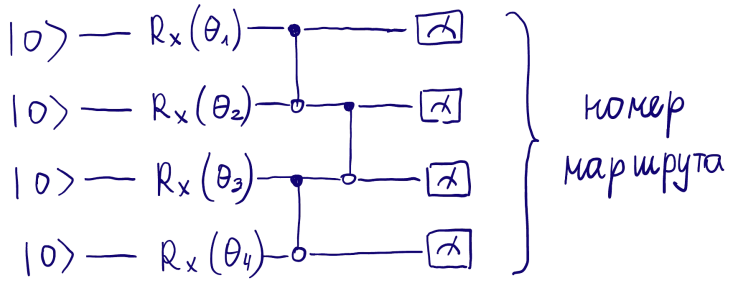

Рисунок 2.3 — Квантовая схема, использованная в разработанном алгоритме

Нам было нужно параметрически создавать квантовое состояние, которое соответствует какому-то маршруту или их суперпозиции. Исходя из этого, мы выдвинули три требования к квантовой цепи:

1) все состояния-маршруты должны быть «достижимыми»: для каждого маршрута должен существовать набор параметров такой, что на выходе цепи

должно получиться состояние, соответствующее этому маршруту

2) все достижимые состояния должны быть «корректными»: при любом наборе параметров на выходе цепи должна получаться суперпозиция допустимых маршрутов

3) параметрические гейты должны позволять использовать алгоритм Rotosolve, который мы обсудим в следующем разделе. Он применим к широкому классу гейтов, но не ко всем.

Ранее мы выбрали способ представления маршрутов (через их номер), в котором все состояния, составляющие вычислительный базис, соответствуют какому-то допустимому маршруту. А совершенно произвольное состояние, таким образом, соответствует некоторой суперпозиции допустимых маршрутов, и требование (2) выполнено автоматически.

Требование достижимости (1) тоже выполнено. В этом несложно убедиться, если сначала заметить, что для создания произвольного маршрута (двоичного числа, символизирующего его номер) было бы достаточно одного слоя гейтов Rx. Углы в остальных слоях (если они есть) можно положить равными 0, тогда останутся только гейты CNOT, которые легко обращаются.

Чтобы выполнить все ограничения, гейты CNOT не понадобились. Однако они устанавливают связь между кубитами, поэтому мы их добавили. Это улучшило результат в симуляции.

Обсудим также то, как кубиты должны быть соединены между собой на физическом устройстве. Эта тема остаётся «больным местом» физических реализаций квантовых компьютеров. Никому до сих пор не удалось создать квантовый компьютер, в котором каждый кубит был бы связан с каждым, и это вряд ли изменится когда-либо в обозримом будущем.

К счастью, в нашей схеме двухкубитные гейты производятся только над соседними кубитами, следовательно, кубитам достаточно просто быть соединёнными в линию (рис. 2.4).

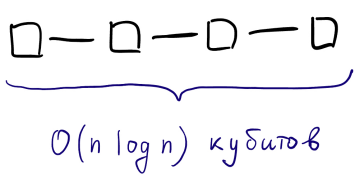

Рисунок 2.4 — Required qubit connectivity

Квантовые компьютеры с такой простой структурой можно ожидать в ближайшем будущем.

2.3 Алгоритм Rotosolve для подбора параметров

Наша схема содержит гейты поворота $\mathrm{Rx}(\theta)$, и в ходе алгоритма мы пытаемся подобрать оптимальные углы θ, минимизирующие среднюю стоимость состояния, получающегося на выходе. Для этого мы могли бы использовать любой алгоритм поиска минимума функции, например метод градиентного спуска.

Но в **rotosolve** году Ostaszewski et al. [**rotosolve**] предложили более эффективный алгоритм, названный Rotosolve. Они заметили, что большинство доступных квантовому компьютеру параметрических гейтов может быть представлено в виде

$$\hat{U}(\theta) = \exp\left(-\frac{i\theta}{2}\hat{H}\right) = \cos\left(\frac{\theta}{2}\right)\hat{1} - i\sin\left(\frac{\theta}{2}\right)\hat{H}, \tag{2.1}$$

где $\hat{H}$ — это некоторая унитарная эрмитова матрица — такая, что $\hat{H}^2 = \hat{1}$.

В частности, при $\hat{H} = \hat{\sigma}_x,\ \hat{\sigma}_y,\ \hat{\sigma}_z$ мы получаем знакомые гейты поворота Rx, Ry, Rz. Физический смысл параметра θ заключается во времени воздействия на кубит.

Из формулы (2.1) видно, что если зафиксировать параметры всех гейтов, кроме одного, то математическое ожидание произвольной наблюдаемой C будет иметь синусоидальную форму:

$$\langle C\rangle_\theta = a\sin(\theta + b) + c$$

Если мы сможем оценить коэффициенты a, b и c, то мы сможем охарактеризовать синусоиду и найти минимум. Авторы показали, что он с точностью до $2\pi k$ даётся выражением в замкнутой форме

$$\begin{aligned}\theta^* &= \operatorname*{argmin}_\theta \langle C\rangle_\theta = \\ &= \theta_0 - \frac{\pi}{2} - \operatorname{arctan2}\left(2\langle C\rangle_{\theta_0} - \langle C\rangle_{\theta_0+\frac{\pi}{2}} - \langle C\rangle_{\theta_0-\frac{\pi}{2}},\quad \langle C\rangle_{\theta_0+\frac{\pi}{2}} - \langle C\rangle_{\theta_0+\frac{\pi}{2}}\right),\end{aligned}$$

где θ_0 — произвольная начальная точка.

Алгоритм Rotosolve оптимизирует углы для всех гейтов по очереди, затем цикл повторяется, пока не выполнится критерий остановки. В качестве критерия остановки можно использовать:

1) максимальное число итераций
2)

3 Оценки эффективности алгоритма

Мы протестировали наш алгоритм на случайных графах для случая 4, 6, 8, 10 городов (24, 120, 40 320, 3 628 800 возможных решений). Стоимости путей между всеми парами городов генерировались равномерно на отрезке [0, 1]. Мы рассмотрели самую общую задачу, когда стоимость пути может быть разной в зависимости от направления.

3.1 Качество решений

Для начала рассмотрим случай 4 городов. Мы построили распределения стоимостей на входе и на выходе алгоритма (рис. 3.1). Вообще говоря, алгоритм Rotosolve детерминистический: каждая следующая точка в пространстве параметров однозначно определяется предыдущей. Однако мы, во-первых, запускали его из различных случайных начальных точек, и во-вторых, оценивали матожидание стоимости с помощью сэмплирования, что вносит случайную ошибку. Из-за этого, запуская алгоритм несколько раз на одном графе, мы получали разные результаты. Их распределение и изображено на рис. 3.1.

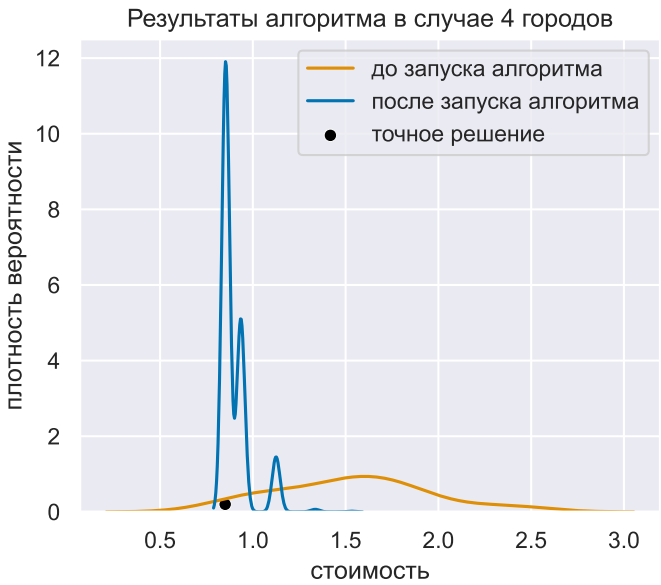

Рисунок 3.1 — Оранжевое: распределение стоимостей всех возможных $n!$ ($n = 4$) маршрутов. Синее: распределение стоимостей решений, к которым приходит алгоритм при старте из различных случайных начальных точек. Чёрной точкой на оси X отмечена стоимость точного решения. В большинстве случаев решение совпадало с точным, это соответствует высокому левому пику. Он имеет ненулевую ширину из-за того, что при построении графика использовался метод ядерной оценки плотности.

Из рисунка 3.1 видно, что в случае 4 городов алгоритм практически всегда выдаёт точное решение. К сожалению, при большем числе городов такого чуда больше не происходит. Однако алгоритм продолжает стабильно выдавать решения с маленькой стоимостью.

Чтобы убедиться в этом, мы построили аналогичные распределения при другом числе городов и объединили их в одну скрипичную диаграмму (рис. 3.2). Для удобства сопоставления мы отнормировали стоимость на среднее по всем перестановкам. Это среднее приблизительно равно $\frac{n-1}{2}$, поскольку стоимость каждого ребра равномерно распределена на [0, 1], а таких рёбер в незамкнутом маршруте ровно $n - 1$.

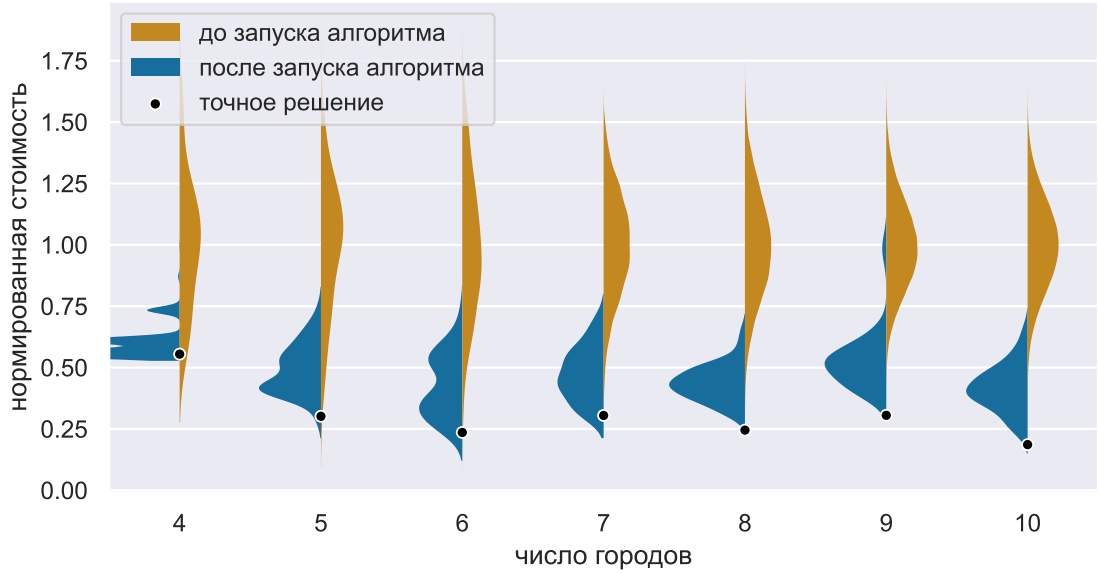

Рисунок 3.2 — Скрипичная диаграмма стоимостей до и после оптимизации. По оси Y отложена стоимость, нормированная на среднее.
Синее: распределение стоимостей решений, к которым приходит алгоритм при старте из различных случайных начальных точек.
Оранжевое: распределение стоимостей всех $n!$ возможных маршрутов.
Толщина символизирует сглаженную плотность вероятности. Чёрными точками отмечены точные решения.

Из рисунка 3.2, во-первых, видно, что оранжевое распределение при росте числа городов в силу центральной предельной теоремы стремится к нормальному. Во-вторых, синее распределение не смещается вверх, это означает, что алгоритм продолжает выдавать решения одного качества при любом числе городов, их стоимость составляет около 0.5 от средней.

3.2 Сложность

3.3 Сравнение с простым перебором

Предположим, мы запустили наш алгоритм на некоторых входных данных, и после M итераций получили решение со стоимостью c. Хорошо это или плохо?

Введём величину $p(c)$ — долю решений со стоимостью, меньшей или равной c (процентильный ранг). Заметим, что в случае простого перебора

$$M \cdot p(c) \approx 1,$$

где $M = nfev$ — количество вычислений целевой функции. Например, чтобы получить решение из лучших 50%, в среднем достаточно двух попыток, а чтобы получить наилучшее решение $\big(p(c) = 1/n!\big)$, требуется перебрать $n!$ кандидатов.

Мы можем вычислить величину $M \cdot p(c)$ для нашего алгоритма и сравнить с единицей (рис. 3.3). Если она меньше единицы, то алгоритм лучше простого перебора, если больше, то хуже.

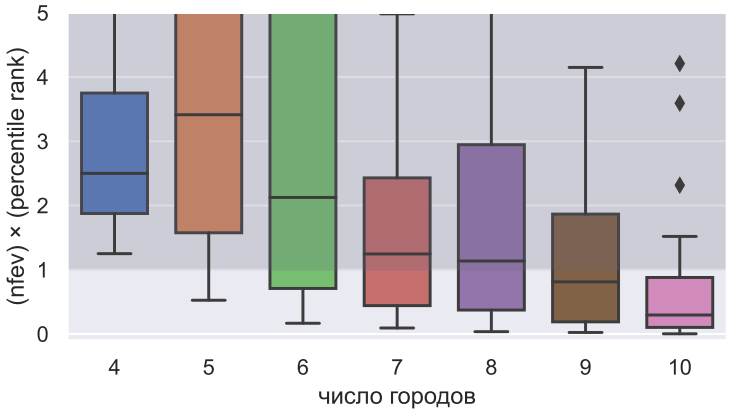

Рисунок 3.3 — Распределение величины $M \cdot p(c)$, где M — количество вычислений функции $\langle \text{cost} \rangle = f(\vec{\theta})$, $\vec{\theta}$ — вариационные параметры, $p(c)$ — процентильный ранг решения, к которому пришёл алгоритм. Незатенённая область соответствует случаю, когда $M \cdot p(c) < 1$ и разработанный алгоритм оказывается производительнее простого перебора.

Рисунок 3.3 показывает, что с ростом числа городов величина $M \cdot p(c)$ в среднем уменьшается, и при 10 городах она оказывается меньше единицы более чем в половине запусков.

Но нужно отметить, что при построении графика мы в качестве *nfev* брали количество вычислений функции $\langle \text{cost} \rangle$ — средней стоимости по суперпозиции. Чтобы её вычислить, нужно измерить квантовое состояние несколько раз (в нашей симуляции *n_samples* = 100 раз).

Более честная версия графика 3.3 выглядела бы растянутой по оси Y в *n_samples* раз. Следовательно, при 10 городах квантовое преимущество всё ещё не будет достигнуто. Будет ли оно достигнуто при большем числе городов — открытый вопрос. Величина на графике 3.3 быстро уменьшается, что позволяет надеяться, что она достигнет 1 даже при умножении на *n_samples*, однако мы не можем просимулировать цепи с таким большим количеством кубитов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ