Министерство науки и высшего образования РФ
ФГБОУ ВО «Тверской государственный университет»
Математический факультет
Направление 02.03.01 Математика и компьютерные науки
Профиль «Математическое и компьютерное моделирование»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программирование квантовых случайных блужданий

Автор:
Соловьев Илья Олегович

Научный руководитель:
д. ф.-м. н. Цирулёв А.Н.

Допущен к защите:
Руководитель ООП:

____________________ В.П. Цветков

Тверь 2022

Оглавление

Введение

В данной работе изучаются квантовые случайные блуждания на прямой с дискретным временем. Впервые понятие «Случайное блуждание» было введено английским математиком Карл Пирсон в 1905 году. Теория случайных блужданий используется в разных областях. Но большую популярность эта теория приобрела в естественных науках, однако встречается и в других сферах.

С появлением у компьютоторов больших вычислительных мощностей стало актуальным использование стохастически= методов для решения практических задач - это объясняет актуальность данной работы. Целью работы является изучение теории классических и квантовызх случайных блужданий, а также их моделирование для .... Для достижения этих целей были поствлены задачи: изучение учебной и научной литературы, разбор принципа классических и квантовых случайных блужданий, составление программ на языке программирования RUST.

Работа состоит из двух глав, заключения и списка литературы. Первая глава носит вводный и обзорный характер. В ней рассматриваются основные принципы классических случайных блужданий на решеткаъ, связь случайных блужданий с цепями маркова и основные принципы квантовых случайных блужданий . Во второй главе исследуются квантовые случайные блуждания на примерах. В заключении сформулированы основные результаты работы. Список литературы состоит из 5 источников.

Глава 1

Квантовые блуждания на решетках

1.1 Случайные блуждания на прямой

Популярной моделью случайного блуждания является модель случайного блуждания по решетке, где на каждом шаге местоположение переходит к другому в соответствии с некоторым распределением вероятностей. При простом случайном блуждании местоположение может переходить только к соседним узлам решетки. В классическом симметричном случайном блуждании по конечной решетке вероятности перехода местоположения к каждому из его непосредственных соседей одинаковы. Если пространство состояний ограничено конечными размерами, модель случайного блуждания называется простым симметричным случайным блужданием с границами, а вероятности перехода зависят от местоположения состояния, поскольку в краевых и угловых состояниях движение ограничено.

Рассмотрим простейшую математическую модель случайного блуждания. пусть в дискретные моменты времени $t_0, t_1, ...$ частица может совершить только один скачок вдоль прямой так, что в момент времени t_{n+1}, она отстает от точки t_n влево или вправо на единичное расстояние. Пусть координата частицы в любой момент времени есть число. Введем на прямой начало отсчета (для удобства возьмем 0, как начало отсчета)

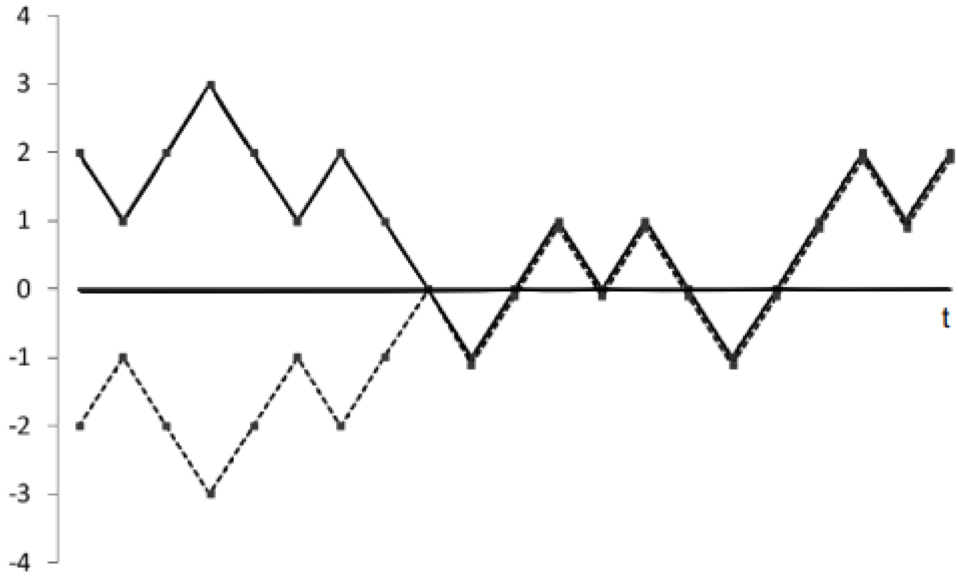

Рис. 1.1: Ломаная из n звеньев

Блуждание имеет случайный характер: с вероятностью p частица может совершить прыжок в права, тогда с вероятностью $q = 1 - p$ частица будет выполнять прыжок влево. В данной случае любые другие перемещения невозможны, так как $p + q = 1$. Отметим, что состояние частицы в момент времени t_{n+1} зависит только от состояния частицы в момент времени t_n.

Для анализа случайных блужданий удобно пользоваться понятием "случайной"траектории на n шагов.Это набор точек $(j, \xi_j), j = 0, 1...n$ на двумерной плоскости, где первая координата - момент времени $t = j$, а вторая - координата частицы в момент времени $t = j$. Для удобного визуального восприятия удобно соединить точки траектории отрезками прямых, тогда на графики мы увидим непрерывную ломаную из n звеньев. Нам интересно узнать распределения попаданий точек в каждую из координат.

1.2 Метод цепей Маркова в моделировании случайных блужданий

Случайные блуждания можно сформулировать по разному. Обычно мы говорим, что случайные блуждания - это последовательность движений, сгенерированных стохастическим (случайным) образом внутри заданного состояния. Если стохастические движения коррелируют во времени, мы говорим о немарковских блужданиях (блуждания с памятью), однако дальше мы будем говорить только о марковских блужданиях - случайные движения частицы не зависят от времени. Стоит отметить, что блуждания могут зависеть от положения. Последовательности таких ходов приводят к так называемые цепям Маркова.

Рассмотрим марковский процесс с дискретным временем, в котором вероятности не зависят от номера шага. Общий вид:

$$p^{(k+1)} = p^{(k)}P, \tag{1.1}$$

где $p^{(k)}$ - распределение вероятностей на k-ом шаге, а P - матрица переходных вероятностей.

$$(...p_{n_1}, p_{n_2}, p_{n_3}, ...)^{(k+1)} = (...p_{n_1}, p_{n_2}, p_{n_3}, ...)^{(k)} \begin{pmatrix} ... & ... & ... & ... & ... \\ ... & p_{n_1n_1} & p_{n_1n_2} & p_{n_1n_3} & ... \\ ... & p_{n_2n_1} & p_{n_2n_2} & p_{n_2n_3} & ... \\ ... & p_{n_3n_1} & p_{n_3n_2} & p_{n_3n_3} & ... \\ ... & ... & ... & ... & ... \end{pmatrix}, \tag{1.2}$$

где $...p_{n_1}, p_{n_2}, p_{n_3}, ...$ - вероятности нахождения частицы в точках $..., n_1, n_2, n_3, ...$ на k-ом шаге. Для любой строки матрицы P элементы $p_{ij} > 0$, сумма элементов в строке должна равняться 1. p_{ij} - вероятность перейти в позицию j при условии, что на k шаге мы находились в позиции i.

Имея матрицу P и начальное состояние $(...p_{n_1}, p_{n_2}, p_{n_3}, ...)^{(0)}$, распределение после m шагов описывается уравнением:

$$(...p_{n_1}, p_{n_2}, p_{n_3}, ...)^{(m)} = (...p_{n_1}, p_{n_2}, p_{n_3}, ...)^{(0)}P^m, \tag{1.3}$$

Для примера возьмем случайное блуждание на целочисленном отрезке $[-2, 2] \subset \mathbb{Z}$, тогда получим:

$$(p_{-2}, p_{-1}, p_0, p_1, p_2)^{(k+1)} = (p_{-2}, p_{-1}, p_0, p_1, p_2)^{(k)} \begin{pmatrix} q & p & 0 & 0 & 0 \\ q & 0 & p & 0 & 0 \\ 0 & q & 0 & p & 0 \\ 0 & 0 & q & 0 & p \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad (1.4)$$

1.3 Квантовые случайные блуждания

Квантовые блуждания — это квантовые аналоги классических случайных блужданий. Они также как и в случае с классическими случайными блужданиями делятся на два типа: непрерывные и дискретные(далее будут рассматриваться дискретниые). В отличие от классического случайного блуждания, где частица принимает определенные состояния, а случайность возникает за счет стохастических переходов между состояниями, в квантовых блужданиях случайность возникает за счет: квантовой суперпозиции состояний, неслучайной обратимой унитарной эволюции. Характерным свойством квантовых случайных блужданий является локализация, которая определяется тем, что вероятность того, что частица окажется в точке, не сходится к нулю даже в пределе больших времен.

Квантовые блуждания мотивированы широким использованием классических случайных блужданий при разработке рандомизированных алгоритмов и являются частью нескольких квантовых алгоритмов. Для некоторых задач оракула [1] квантовые блуждания обеспечивают экспоненциальное ускорение по сравнению с любым классическим алгоритмом. Квантовые блуждания также дают полиномиальное ускорение по сравнению с классическими алгоритмами для многих практических задач, таких как проблема уникальности элементов, проблема поиска треугольников. Известный алгоритм поиска Гровера также можно рассматривать как алгоритм квантового блуждания.

[1] В теории вычислимости машина-оракул — это абстрактная машина, используемая для изучения проблем принятия решений. Его можно представить как машину Тьюринга с черным ящиком, называемым оракулом, который способен решать определенные задачи за одну операцию. Задача может быть любого класса сложности. Можно использовать даже неразрешимые проблемы, такие как проблема остановки.

Далее мы сосредоточимся на пространственно-неоднородных квантовых блужданий в одномерном пространстве. В качестве простых случаев мы рассматривать квантовые блуждания, вызванные блужданием Адамара. Кубит — наименьшая единица информации в квантовом компьютере (аналог бита в обычном компьютере), использующаяся для квантовых вычислений. Как и бит, кубит допускает два собственных состояния, обозначаемых $|0\rangle$ и $|1\rangle$, но при этом может находиться и в их суперпозиции. В общем случае его волновая функция имеет вид $A|0\rangle + B|1\rangle$ где A и B амплитудами вероятностей и являются комплексными числами, удовлетворяющими условию $|A|^2 - |B|^2 = 1$. При измерении состояния кубита можно получить лишь одно из его собственных состояний. Вероятности получить каждое из них равны $|A|^2$ и $|B|^2$ соответственно.

Квантовый вентиль (квантовый логический элемент) — это базовый элемент квантового компьютера, преобразующий входные состояния кубитов на выходные по определённому закону. Отличается от обычных логических вентилей тем, что работает с кубитами. Квантовые вентили в отличие от многих классических всегда являются обратимыми.

Так как кубит можно представить вектором в двумерном пространстве, то действие вентиля можно описать унитарной матрицей, на которую умножается соответствующий вектор состояния входного кубита. Однокубитные вентили описываются матрицами размера $2 * 2$, двухкубитные — $4 * 4$, а n-кубитные — $2^n * 2^n$.

Примеры простейших однокубитовых вентилей:

- Тождественное преобразование

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- Отрицание

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- Преобразование Адамара

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Глава 2

Моделирование квантовых случайных блужданий

2.1 Правила перехода

В классических случайных блужданиях на прямой мы начинали с позиции 0. На каждом шаге мы имели возможность переместиться влево или вправо с одинаковой вероятностью. В квантовых случайных блужданиях аналогом будет квантовый процесс с базисными состояниями $|n\rangle, n \in \mathbb{Z}$. На каждом временном шаге он будет выполнять преобразование

$$|n\rangle \to a|n-1\rangle + b|n\rangle + c|n+1\rangle \tag{2.1}$$

что означает движение влево с амплитудой a сохранение позиции с амплитудой b и движение вправо с амплитудой c.Также мы хотели бы, чтобы движение осуществлялось в каждую из сторон. Т.е. a, b, c не должны зависеть от шага(так же, как и в классических блужданиях перемещение влево/вправо не зависят от шага n). Но в данном случае так не выйдет.

Преобразование U, определяемое уравнением (2.3) является унитарным тогда и только тогда, когда выполняется одно из следующих условий:

1. $|\mathrm{a}| = 1$, $\mathrm{b} = \mathrm{c} = 0$;
2. $|\mathrm{b}| = 1$, $\mathrm{a} = \mathrm{c} = 0$;

3. |c| = 1, a = b = 0;

Из этого мы можем сделать вывод, что единственно возможными преобразованиями является тривиальные (те, при которых движение возможно только в одну из сторон, либо остановка на месте).

Эту проблему можно решить введя дополнительное "монетное"состояние. Мы будем рассматривать пространство из состояний $|n, 0\rangle$ и $|n, 1\rangle$ при $n \in \mathbb{Z}$. На каждом шаге мы будем выполнять две операции:

1. $C|n, 0\rangle = a|n, 0\rangle + b|n, 1\rangle$
2. $C|n, 1\rangle = c|n, 0\rangle + d|n, 1\rangle$

Сдвиг S:

$$S|n, 0\rangle = |n - 1, 0\rangle, S|n, 1\rangle = |n + 1, 1\rangle \tag{2.2}$$

Шаг квантового блуждания - это SC. Для C часто выбирают преобразование Адамара:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \tag{2.3}$$

Можно использовать любое другое двумерное унитарное преобразование.

Можно думать о C как о квантовом аналоге подбрасывания монетки, в котором мы решаем, в каком из направлений мы будем двигаться. Чтобы разобраться с тем, как это работает, мы рассмотрим пример, в котором между C и S мы измерим состояние. Если состояние до С было равно $|n, 0\rangle$, то после состояние будет равно $\frac{1}{\sqrt{2}}|n, 0\rangle + \frac{1}{\sqrt{2}}|n, 1\rangle$ и измерение этого выражения дает нам $|n, 0\rangle$ и $|n, 1\rangle$ с вероятностью 0.5 для каждого.Таким оразом мы видим, что C эквивалентно выбору одного кубита из $|n, 0\rangle$ и $|n, 1\rangle$ с вероятностями 0.5 для каждого.

Рассмотрим три первых шага квантового блуждания с преобразованием Адамара с начальным состоянием $|0, 0\rangle$:

1. $|0, 0\rangle \rightarrow$

2. $\frac{1}{\sqrt{2}}|0,0\rangle + \frac{1}{\sqrt{2}}|0,1\rangle \to \frac{1}{\sqrt{2}}|-1,0\rangle + \frac{1}{\sqrt{2}}|1,1\rangle \to$

3. $\frac{1}{2}|-1,0\rangle + \frac{1}{2}|-1,1\rangle + \frac{1}{2}|1,0\rangle - \frac{1}{2}|1,1\rangle \to \frac{1}{2}|-2,0\rangle + \frac{1}{2}|0,1\rangle + \frac{1}{2}|0,0\rangle - \frac{1}{2}|2,1\rangle \to$

4. $\frac{1}{2\sqrt{2}}|-2,0\rangle + \frac{1}{2\sqrt{2}}|-2,1\rangle + \frac{1}{\sqrt{2}}|0,0\rangle - \frac{1}{2\sqrt{2}}|2,0\rangle + \frac{1}{2\sqrt{2}}|2,1\rangle \to \frac{1}{2\sqrt{2}}|-3,0\rangle + \frac{1}{2\sqrt{2}}|-1,1\rangle + \frac{1}{\sqrt{2}}|-1,0\rangle - \frac{1}{2\sqrt{2}}|1,0\rangle + \frac{1}{2\sqrt{2}}|3,1\rangle$

2.2 Случайные возмущения процесса

Нужна помощь с написанием вступления в этой главе(для чего это делается)

Квантовым вентелем может быть любая матрица A определяемая как:

$$A = \begin{pmatrix} a_0 + ia_3 & -a_1 + ia_2 \\ a_1 + ia_2 & a_0 - ia_3 \end{pmatrix} \tag{2.4}$$

При условии, что:

1. $a_0^2 + a_1^2 + a_2^2 + a_3^2 = 1$
2. $A^* * A = E$

Для дальнейщего моделирования квантового блуждания введем слежущее возмущение процесса: пусть $p \in [0,1)$ - число, которое случайным образом генирируется перед каждым преобразованием. В случае когда $p > 0.9$ мы будем выполнять следущее преобразование:

$$\begin{pmatrix} \sqrt{p} & -\sqrt{1-p} \\ \sqrt{1-p} & \sqrt{p} \end{pmatrix} \tag{2.5}$$

2.3 Математические модели и алгоритмы простейших квантовых случайных блужданий

Алгоритм реализован на языке программирования Rust. Он был выбран из-за скорости его работы, довольно удобного синтаксиса (позволяет разрабатывать как в функциональном так и объектно-ориентированном стиле) и компилятора.

Для начала нужно реализовать дополнительные функции для того, чтобы код был самодокументируемым и простым в расширении. Начнем с создания структуры Qbit.

```rust
struct Qbit {
    amplitude: f64,
    position: i32,
    coin_state: bool,
}
```

- амплитуда
- точка в которой может находиться наша частица
- монетное состояние

Добавим "вспомогательные"функции:

- `hadamar(initial_qbit: &Qbit) -> (Qbit, Qbit)`

  \- преобразование Адамара

- `coin_transformation(initial_qbit: Qbit) -> Qbit`

  \- трансформация подбрасывания монеты, которая ранее была названа, как S

- `get_qbit_key(position: i32, coin_state: bool) -> String`

  \- получение уникального ключа для каждого кубита

- `unit_transformation(initial_qbit: &Qbit, p: f64) -> (Qbit, Qbit)`

  \- реализация (2.5)

Рассмотри подробнее каждую из этих функций с комментариями.

```
fn hadamar(initial_qbit: &Qbit) -> (Qbit, Qbit) {
  return if initial_qbit.coin_state == false {
    (
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()),
        position: initial_qbit.position,
        coin_state: false,
      },
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()),
        position: initial_qbit.position,
        coin_state: true,
      })
  } else {
    (
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()),
        position: initial_qbit.position,
        coin_state: false,
      },
      Qbit {
        amplitude: initial_qbit.amplitude*(1.0 / 2_f64.sqrt()) * -1.0,
        position: initial_qbit.position,
        coin_state: true,
      })
  };
}
```

Реализация (2.1) с выбранным преобразованием Адамара. На основе исходного кубита, мы получаем два новых с амплитудами, которые зависят от монетного состояния и граничных условий.

```
fn coin_transformation(initial_qbit: Qbit, borders: (i32, i32)) -> Qbit
   {
  return if initial_qbit.coin_state {
    if initial_qbit.position == borders.1 {
      Qbit {
        amplitude: initial_qbit.amplitude,
        coin_state: initial_qbit.coin_state,
        position: initial_qbit.position - 1,
      }
    } else {
      Qbit {
        amplitude: initial_qbit.amplitude,
        coin_state: initial_qbit.coin_state,
        position: initial_qbit.position + 1,
      }
    }
  } else {
    if initial_qbit.position == borders.0 {
```

```
18       Qbit {
19         amplitude: initial_qbit.amplitude,
20         coin_state: initial_qbit.coin_state,
21         position: initial_qbit.position + 1,
22       }
23     } else {
24       Qbit {
25         amplitude: initial_qbit.amplitude,
26         coin_state: initial_qbit.coin_state,
27         position: initial_qbit.position - 1,
28       }
29     }
30   };
31 }
```

Релизация (2.2). В случае, когда монетное состояние равно 1, мы возвращаем новый кубит у которого позиция изменилась на $n+1$, в обратном кубит с позицией частицы $n-1$. Но также стоит сотметить, что в случаае, когда частица находиться на границах, то мы будем перемещать частиуц в противоположную от границы сторону на 1 единицу. За это отвечают строчки 7, 13, 21, 27.

```
    fn get_qbit_key(position: i32, coin_state: bool) -> String {
        coin_state.to_string() + ";" + &*position.to_string()
    }
```

Кубит идентифицируют его два состояния - точка в которой находиться частица, монетное состояние. Эта функция нужна, чтобы во время блуждания найти в хеш-таблице кубит с такими же состояниями и пересчитать его амплитуду основываясь на только что полученых новых значения амплитуды.

Перейдем к основной программе.

```
  let mut states: HashMap<String, Qbit> = HashMap::new();
  let borders = (-15, 15);
  let mut rng = rand::thread_rng();
  let mut entropy: f64 = 0.0;
  let mut file = File::create("quantum-walks2.txt").unwrap();
```

- *states* - хеш-таблица с ключом, созданным с помощью *getQbitKey* и значением - структурой *Qbit*
- *entropy* - зачение энтропии
- *borders* - кортеж из двух чисел, которые является граничными условиями

- *rng* - инстанс структуры rand, которые дает возможность получать случайное число от 0 до 1
- *file* - структура с методами записи информации в файл "quantum-walks2.txt"

Теперь нужно задать начальные условия $|0, 0\rangle$. Для этого вставим в *states* первое значение.

```
states.insert(get_qbit_key(0, false), Qbit {
   amplitude: 1.0,
   position: 0,
   coin_state: false,
 });
```

```
 for n in (10..=100).step_by(10) {
   for _ in 1..=n {
     let mut temp_states: HashMap<String, Qbit> = HashMap::new();

     let mut insert_qbit_to_hash_map = |key: String, qbit: Qbit| {
       match temp_states.get(&key) {
         None => {
           temp_states.insert(key, qbit);
         }
         Some(currentQbit) => {
           let amplitude: f64 = currentQbit.amplitude + qbit.amplitude;
           if amplitude == 0.0 {
             temp_states.remove(&*key);
           } else {
             temp_states.insert(key, Qbit {
               amplitude: currentQbit.amplitude + qbit.amplitude,
               coin_state: qbit.coin_state,
               position: qbit.position,
             });
           }
         }
       }
     };
     let p = rng.gen::<f32>();

     for state in states.values() {

       let qbits: (Qbit, Qbit);

       if p > 0.9 {
         qbits = unit_transformation(state, p as f64);
       } else {
         qbits = hadamar(state);
       }

```

```
36       let shiftedFirstQbit = coin_transformation(qbits.0, borders);
37       let shiftedSecondQbit = coin_transformation(qbits.1, borders);
38
39       let first_qbit_key = get_qbit_key(shiftedFirstQbit.position,
 shiftedFirstQbit.coin_state);
40       let second_qbit_key = get_qbit_key(shiftedSecondQbit.position,
 shiftedSecondQbit.coin_state);
41
42       insert_qbit_to_hash_map(first_qbit_key, shiftedFirstQbit);
43       insert_qbit_to_hash_map(second_qbit_key, shiftedSecondQbit);
44     }
45
46     states = temp_states;
47   }
48
49
50   for state in states.values() {
51     let probability = f64::powf(state.amplitude, 2.0);
52
53     if probability != 0.0 {
54       entropy -= probability * probability.ln();
55     }
56   }
57   writeln!(&mut file, "n: {} s: {}", n, entropy);
58
59   entropy = 0.0;
60 }
```

В строках 1-2 мы создаем цикл for, с шагом n в 10 единиц. Количество выполенных преобразований перед вычислением энтропии будут зависеть от n. В 3 строке создадаем временную хеш таблицу в которую мы будем записывать результаты нашего преобразования на каждом шаге. Строки 5-23 отвечают за замыкание, которое будет добавлять в нашу временнцю хеш таблицу только что полученные кубиты. Давайте детально рассмотрим, как эта функция работает. Если в нашей таблице еще не существует значения с переданным ключем, то мы просто вставляем это значение в таблциу. Но в случае, когда это значение есть мы вчислаяем амплитуду по формуле $currentQbit.amplitude + qbit.amplitude$. В случае, если это значение равняется нулю, то мы удаляет из таблицы кубит по переданному ключу, но если значение амплитуды не равно нулю, то мы записываем по ключу новы кубит с тоько что вычисленной амплитудой.

На строке 24 создаем переменную p, которая принимает случайным образом значение от 0 до 1. Она отвечает за возмещение процесса, кеоторое было описано в (2.5)

В строках 26-47 происходит вычисление состояний кубитов. В 26 строке мы создаем итератор по значения которого мы будем пробегать и на их основе вычислать новые значения кубитов. На строке 28 создаем новую переменную *qbits*, в которой будет находиться результат унитарного преобразования. Строки 30-34 отвечают за само унитрное преобразование, в которых с вероятностью p будет выполнение преобразование Адамара, и с вероятностью $1-p$ преобразование (2.5). В строках 36-37 мы создадим две перенных *shiftedFirstQbit* и *shiftedSecondQbit*, в которых будут находиться результат "монетного преобразования"первого и второго кубита соответственно. В строках 42-43 мы вставляем результаты в нашу временную хеш таблицу.

На 46 строке мы записываем значниея, которые хранились во временной хеш таблице в основную. Временная таблица была создана для того, чтобы во вроемя итерации по основной не мутировать ее. В случае, когда бы мы ее мутировали, то у нас этот цикл выполнялся бесконечно, а компилятор Rust даже не дал бы запустить программу из-за утечки памяти.

Строки 50-59 отвечают за вычисление энтропии и записи результата в файл с соотетствующим ей количеством шагов.

Заключение

В работе получены следующие основные результаты

Литература

[1] Х. Гулд, Я. Тобочник. *Компьютерное моделирование в физике. Том 2.* М.: Мир, 1990

[2] D. Reitzner, D. Nagaj, V. Buzek. *Quantum walks.* Acta Physica Slovaca, **61**, No. 6, 2011, pp. 603 – 725 (*arXiv:* 1207.7283)

[3] A. Ambainis. *Quantum walks and their algorithmic applications.* 2004 (*arXiv:* quant-ph/0403120)

[4] J. Kempe. *Quantum random walks — an introductory overview.* Contemporary Physics, **44**, Issue 4, 2003, pp. 307 – 327 (*arXiv:* quant-ph/0303081)

[5] G. D. Molfetta. *Quantum walks, limits and transport equations, Sec. 3.* 2021 (*arXiv:* 2112.11828)

Приложение Rust

1. Программа

```rust
use rand::Rng;
use std::fs::File;
use std::io::prelude::*;
use std::collections::HashMap;

struct Qbit {
  amplitude: f64,
  position: i32,
  coin_state: bool,
}

fn main() {
  quantum_walks();
}


fn quantum_walks() {
  let mut file = File::create("quantum-walks2.txt").unwrap();
  let mut states: HashMap<String, Qbit> = HashMap::new();
  let borders = (-15, 15);
  let mut rng = rand::thread_rng();

  let mut entropy: f64 = 0.0;

  states.insert(get_qbit_key(0, false), Qbit {
    amplitude: 1.0,
    position: 0,
    coin_state: false,
  });
  for n in (10..=100).step_by(10) {
    for _ in 1..=n {
      let mut temp_states: HashMap<String, Qbit> = HashMap::new();

      let mut insert_qbit_to_hash_map = |key: String, qbit: Qbit| {
        match temp_states.get(&key) {
          None => {
```

```
          temp_states.insert(key, qbit);
        }
        Some(currentQbit) => {
          let amplitude: f64 = currentQbit.amplitude + qbit.amplitude;

          if amplitude == 0.0 {
            temp_states.remove(&*key);
          } else {
            temp_states.insert(key, Qbit {
              amplitude: currentQbit.amplitude + qbit.amplitude,
              coin_state: qbit.coin_state,
              position: qbit.position,
            });
          }
        }
      }
    };
    let p = rng.gen::<f32>();

    for state in states.values() {

      let qbits: (Qbit, Qbit);

      if p > 0.9 {
        qbits = unit_transformation(state, p as f64);
      } else {
        qbits = hadamar(state);
      }

      let shiftedFirstQbit = coin_transformation(qbits.0, borders);
      let shiftedSecondQbit = coin_transformation(qbits.1, borders);

      let first_qbit_key = get_qbit_key(shiftedFirstQbit.position, shiftedFirstQbit.coin_state);
      let second_qbit_key = get_qbit_key(shiftedSecondQbit.position, shiftedSecondQbit.coin_state);

      insert_qbit_to_hash_map(first_qbit_key, shiftedFirstQbit);
      insert_qbit_to_hash_map(second_qbit_key, shiftedSecondQbit);
    }

    states = temp_states;
  }


  for state in states.values() {
    let probability = f64::powf(state.amplitude, 2.0);

    if probability != 0.0 {
      entropy -= probability * probability.ln();
    }
```

```
    }
    writeln!(&mut file, "n: {} s: {}", n, entropy);

    entropy = 0.0;
  }
}

fn get_qbit_key(position: i32, coin_state: bool) -> String {
  coin_state.to_string() + ";" + &*position.to_string()
}

fn hadamar(initial_qbit: &Qbit) -> (Qbit, Qbit) {
  return if initial_qbit.coin_state == false {
    (
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()),
        position: initial_qbit.position,
        coin_state: false,
      },
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()),
        position: initial_qbit.position,
        coin_state: true,
      })
  } else {
    (
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()),
        position: initial_qbit.position,
        coin_state: false,
      },
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 / 2_f64.sqrt()) * -1.0,
        position: initial_qbit.position,
        coin_state: true,
      })
  };
}

fn unit_transformation(initial_qbit: &Qbit, p: f64) -> (Qbit, Qbit) {
  return if initial_qbit.coin_state == false {
    (
      Qbit {
        amplitude: initial_qbit.amplitude * p.sqrt(),
        position: initial_qbit.position,
        coin_state: false,
      },
      Qbit {
        amplitude: initial_qbit.amplitude * (-1.0 *(1.0 - p).sqrt()),
        position: initial_qbit.position,
        coin_state: true,
```

```
      })
  } else {
    (
      Qbit {
        amplitude: initial_qbit.amplitude * (1.0 *(1.0 - p).sqrt()),
        position: initial_qbit.position,
        coin_state: false,
      },
      Qbit {
        amplitude: initial_qbit.amplitude * p.sqrt(),
        position: initial_qbit.position,
        coin_state: true,
      })
  };
}

fn coin_transformation(initial_qbit: Qbit, borders: (i32, i32)) -> Qbit {
  return if initial_qbit.coin_state {
    if initial_qbit.position == borders.1 {
      Qbit {
        amplitude: initial_qbit.amplitude,
        coin_state: initial_qbit.coin_state,
        position: initial_qbit.position - 1,
      }
    } else {
      Qbit {
        amplitude: initial_qbit.amplitude,
        coin_state: initial_qbit.coin_state,
        position: initial_qbit.position + 1,
      }
    }
  } else {
    if initial_qbit.position == borders.0 {
      Qbit {
        amplitude: initial_qbit.amplitude,
        coin_state: initial_qbit.coin_state,
        position: initial_qbit.position + 1,
      }
    } else {
      Qbit {
        amplitude: initial_qbit.amplitude,
        coin_state: initial_qbit.coin_state,
        position: initial_qbit.position - 1,
      }
    }
  };
}
```