Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра математической кибернетики

Соколов Михаил Александрович

Исследование и реализация некоторых протоколов общения двух игроков в процессе вычисления значения универсального отношения

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
д.ф.-м.н., профессор
С.А.Ложкин

Москва, 2024

Содержание

1. Введение

Протоколы общения двух игроков в процессе вычисления значения универсального отношения изучались в ряде работ ([1, 2]). В работе [1] была доказана связь между глубиной функции и ее т.н. коммуникативной сложностью, то есть количеством бит информации, которыми обмениваются игроки, чтобы определить индекс значения, отличающегося в двух наборах с разными значениями этой функции. В работе [2] были приведены некоторые протоколы, вычисляющие т.н. универсальное отношение, которые близки к оптимальным по количеству раундов или пересланной информации, обоснована их корректность. В работе [6] была рассмотрена связь между задачей строгого универсального отношения и глубиной т.н. мультиплексорной функции μ_n – функции, которая имеет $n + 2^n$ переменных, первые n из которых являются ее адресными переменными, остальные 2^n – информационными, а значение функции равно той ее информационной переменной, номер которой получается из адресных переменных. Также в этой работе была написана программа, которая по введенному протоколу строит схему для мультиплексорной функции, получающейся после обмена двух игроков своими наборами. В работах [3, 4] были найдены точные значения глубины мультиплексорной функции от n адресных булевых переменных (БП) в стандартном базисе из конъюнкции, дизъюнкции и отрицания, где глубина элементов конъюнкции и дизъюнкции равна 1, а отрицания – нулю. При $n = 1$ глубина равна 2, при $2 \leq n \leq 5$ и $n \geq 10$ равна $n + 2$. А для $6 \leq n \leq 9$ глубина равна либо $n + 2$, либо $n + 3$. В работе [7] были установлены оценки для стандартного базиса: при $n > 109$ глубина мультиплексорной функции равна $n + 2$, а при $3 < n \leq 109$ равна либо $n + 2$, либо $n + 3$.

В данной работе изучается соотношение между строгим универсальным отношением и глубиной мультиплексорной функции, а также протоколы общения двух игроков из статьи [2] и их коммуникативная сложность, затем они реализуются в машинном коде с целью проиллюстрировать их работу.

2. Основные понятия

В данной работе рассматривается модель общения двух игроков - Алисы и Боба. Они получают два набора длины n, состоящих из нулей и единиц: x_1, x_2, …, x_n и y_1, y_2, …, y_n соответственно, причем вначале каждый игрок знает только свой набор. За одно сообщение им можно передать один бит другому игроку, при этом другой игрок его примет и запомнит.

Протокол – это некоторое заранее определенное соглашение, или правило, которому следуют игроки при пересылке информации. Протокол может быть задан письменно в виде алгоритма (псевдокода), по которому можно свои входные данные (полученный набор значений) преобразовать в выходные данные (в ту информацию, которая будет отправлена другому игроку), чтобы в итоге получить некоторое необходимое знание, в нашем случае - различающийся бит в двух наборах. Под сложностью протокола будем понимать общее число пересланных игроками бит в худшем случае. Также интерес будет представлять наибольшее число раундов для каждого протокола.

Раунд – это процесс отправки заданного протоколом на данном этапе числа бит одним игроком другому. Таким образом, весь протокол можно разбить на раунды, в течение каждого из которых один игрок пересылает другому по одному несколько бит, которые объединены общим правилом, а второй игрок принимает эти данные и начинает следующий раунд.

Задача универсального отношения – это задача поиска индекса различающегося значения в двух наборах при условии, что наборы не совпадают и состоят из n нулей и единиц.

Задача строгого универсального отношения – это задача универсального отношения, но с условием, что изначально наборы могут совпасть, а оба игрока в этом случае должны будут это распознать.

Протоколы в данной работе решают задачу универсального отношения. В статье [2] было показано, что любой протокол для задачи универсального отношения можно преобразовать в протокол, решающий задачу строгого универсального отношения, добавив при этом не более одного раунда и двух бит пересланной информации.

Будет использована библиотека asyncio, которая позволяет реализовать асинхронное общение между двумя функциями в классе протоколов – async def Alice и async def Bob. Обмен одним битом информации будет проходить через асинхронную очередь – self.queue = asyncio.Queue(1). Один игрок может положить в эту очередь некоторое значение x с помощью библиотечной функции await self.queue.put(x), а другой может принять это значение в некоторую переменную x = await self.queue.get(). Ключевое слово await используется для того, чтобы можно было ожидать значение, которое еще пока не отправили. Также после каждой отправки используется await

asyncio.sleep(0), чтобы другой игрок мог принять отправленное сообщение.

Протокол Хэмминга основан на том, что среди всевозможных 2^n наборов длины $n = 2^r$ можно выделить 2^{n-r} наборов, которые будут составлять некоторое множество C_n, такое что для любого набора x длины n найдется набор c из C_n, такой что x и c будут отличаться лишь в одной позиции. И если каждому c из C_n присвоить порядковый номер из $n - r$ бит, то для набора x номер набора c будет его сферой – sph(x), а индекс одной отличающейся позиции будет его индексом – ind(x). Теперь мы сможем у наборов длины n брать некоторые части длины 2^r и сравнивать между собой лишь их сферы длины r, потому что при равенстве этих частей наборов сферы будут равны, а при различии будут разными, так как может быть только одна позиция, в которой наборы отличаются.

Некоторые основные понятия, которые не были объяснены здесь, но могут встретиться в данной работе, можно найти, например, в [5].

3. Постановка задачи

В рамках выпускной квалификационной работы требовалось написать программу, которая реализует процесс общения двух игроков на наборах из таблицы истинности функции, введенной пользователем. Также требовалось проверить ввод на корректность, затем присвоить игрокам два разных набора, таких что на наборе Алисы функция принимает значение 1, а на наборе Боба – 0. В программе должны быть реализованы три протокола: Simple, Trivial и Ham_3 с помощью асинхронных функций, которые работают по очереди, ожидая значения наборов, отправленных через асинхронную очередь, выводя эти значения на экран, таким образом симулируя реальное общение между двумя игроками. Программа должна быть протестирована и работать корректно на всех наборах. Также изучается соотношение между строгим универсальным отношением и глубиной мультиплексорной функции, ставится вопрос, достигается ли глубина мультиплексорной функции на формулах специального вида, которые можно построить по данным протоколам, а именно таких, что формула не содержит отрицаний информационных переменных и любая информационная переменная входит в формулу только один раз.

4. Основная часть

В данной работе была написана программа на языке Python и выложена на удаленный репозиторий Github [8]. Программа принимает функцию, заданную столбцом ее значений. Она проверяет ввод на корректность, а именно на то, что введенная пользователем последовательность символов состоит только из нулей и единиц, и ее длина является некоторой степенью двойки. Сообщает пользователю, если он ошибся, или ввел функцию, тождественно равную нулю или единице, и просит повторить ввод, пока, наконец, пользователь не введет корректный столбец значений функции. Далее программа строит таблицу истинности заданной функции, разделяет наборы на два множества – B0 и B1 – прообразы нуля и единицы функции соответственно. Далее выбирается один набор из B0 и один из B1, набор из B1 дается Алисе, а набор из B0 – Бобу, затем для каждого из протоколов Simple, Trivial и Ham_3 запускаются функции из соответствующего класса, получая эти наборы в качестве входных значений, и начинают работать асинхронно с помощью библиотеки asyncio: обмениваются битами своих наборов через асинхронную очередь с одним значением и ждут друг друга. Таким образом, в конце они получают первый индекс той позиции, в которой их наборы отличаются друг от друга.

Протокол Trivial является очевидной верхней оценкой сложности данной игры. Он состоит из двух раундов, и игроки пересылают в общей сложности $n + \lceil \log_2 n \rceil$ бит информации. В первом раунде Алиса пересылает свое слово целиком Бобу (это n бит информации), затем Боб находит индекс отличающегося бита и пересылает его во втором раунде за $\lceil \log_2 n \rceil$ пересылок (числа от 0 до $n - 1$ можно перевести в двоичные, состоящие не более чем из $\lceil \log_2 n \rceil$ бит). Таким образом, оба игрока находят индекс отличающегося значения.

Протокол Simple состоит из n раундов и в нем игроки пересылают друг другу $n + 2$ бита информации. По очереди два игрока пересылают один бит из своего набора и принимают один бит из набора другого игрока, причем Алиса пересылает биты с нечетными номерами в своем наборе, а Боб – с четными. Если в каком-то раунде игрок видит несовпадение присланного бита со своим, то он возводит специальную переменную lock_A (lock_B) в 1, отсылает 1, а дальше начинает пересылать лишь одни нули. В конце игроки обмениваются своими переменными lock_A и lock_B, и, если у другого игрока получился lock = 1, то значит есть различие в том индексе, перед которым в последний раз была отправлена 1, соответственно, другой игрок сможет узнать, где именно было несовпадение в наборах.

Протокол Ham_3 основан на коде Хэмминга. Он использует 3 раунда, и игроки пересылают $n + 2$ бита информации. Длина набора n представляется в виде $s + 2^r$, где $0 \leq s < 2^r$, а сам набор

разбивается на три поднабора X_0, X_1 и X_2 у Алисы и на Y_0, Y_1 и Y_2 у Боба. X_0 и Y_0 имеют длину s, а X_1, Y_1 и X_2, Y_2 - 2^{r-1}. Вначале Алиса посылает Бобу весь набор X_0 и sph(X_1). Боб сравнивает X_0 и Y_0, и, если они не равны, то пересылает два бита 1 и 0, а затем пересылает $\lceil \log_2 s \rceil$ бит индекса отличающегося значения, соответственно Алиса в этом случае понимает, что их наборы отличаются в позиции с этим индексом. Сложность в этом случае составит $s + (2^{(r-1)} - (r - 1)) + 2 + \lceil \log_2 s \rceil$, что меньше, чем $n + 2$. Теперь если X_0 и Y_0 совпали, но отличаются sph(X_1) и sph(Y_1), то тогда Боб пересылает два бита 1 и 1, а затем весь набор Y_1. Алиса сравнивает его с X_1 и пересылает $r - 1$ бит индекса отличающегося значения Бобу. Таким образом, он теперь знает, что наборы отличаются в позиции с этим индексом. Сложность в этом случае составит $s + (2^{(r-1)} - (r - 1)) + 2 + 2^{(r-1)} + r - 1$, что равно $n + 2$. Если же совпали X_0 и Y_0, и также равны sph(X_1) и sph(Y_1), то тогда Боб отсылает 0, затем целиком набор Y_2. Алиса находит отличающийся от Y_2 бит в X_2 и отправляет Бобу 0 и $r - 1$ бит индекса этого значения. Таким образом, и Боб узнает, где отличаются их наборы. Сложность в этом случае составит $s + (2^{(r-1)} - (r - 1)) + 1 + 2^{(r-1)} + 1 + r - 1$, что также равно $n + 2$. То есть с помощью этого протокола оба игрока в конце могут понять, в каком индексе есть различие между их наборами, переслав при этом не более $n + 2$ бит информации.

Для проверки всех протоколов на корректность была написана функция, которая берет каждый набор из таблицы истинности, для него ищет все наборы, отличающиеся ровно в одной позиции, и затем запускает каждый из трех протоколов на этих двух наборах и сравнивает ответы Алисы и Боба с индексом той позиции, в которой действительно есть различие между наборами. Если ответы участников совпали между собой и с этим индексом на всех наборах для всех протоколов, то в конце выводится сообщение об успешном прохождении тестов, а иначе выводится тот набор и протокол, в котором обнаружилось несоответствие. Программа была протестирована при n = 9 и работает корректно.

5. Полученные результаты

В данной выпускной квалификационной работе была реализована программа на языке Python и выложена на удаленный репозиторий [8]. Программа принимает на вход столбец значений некоторой функции, проверяет его на корректность, затем строит таблицу истинности и два множества, которые являются прообразами нуля и единицы введенной функции, потом программа присваивает игрокам наборы из двух этих множеств, которые отличаются ровно в одной позиции, и запускает три протокола, с помощью которых игроки могут найти эту позицию, при этом для иллюстрации работы протоколов каждая пересылка битов информации выводится на экран. В конце программы проводится тестирование этих протоколов для всех наборов, чтобы убедиться, что все работает корректно.

Список литературы

[1] Karchmer M., Wigderson A. Monotone circuits for connectivity require super-logarithmic depth // SIAM Journal on Discrete Mathematics. — 1990. — V. 3. — № 2. — P. 255—265.

[2] Tardos G., Zwick U. The communication complexity of the universal relation // Proceedings of Computational Complexity. Twelfth Annual IEEE Conference. — IEEE, 1997. — P. 247—259

[3] Ложкин С. А., Власов Н. В. О глубине мультиплексорной функции // Вестн. Моск. ун-та. Сер. 15. Вычисл. матем. и киберн. — 2011. — Т. 2. — С. 40-46.

[4] Ложкин С. А. О глубине мультиплексорной функции от “небольшого” числа адресных переменных // Математические заметки — Том 115 — 05.05.2024 — С. 741-748

[5] Ложкин С. А. Лекции по основам кибернетики. — М.: Издательский отдел факультета ВМиК МГУ им. М.В. Ломоносова, 2004.

[6] Белашкин И.А. О глубине и коммуникативной сложности некоторых булевых функций и отношений: Выпускная квалификационная работа, 2022.

[7] Титов В. А. О глубине мультиплексорной функции и поведении функции Шеннона для глубины в некоторых базисах: Магистерская диссертация, 2019.

[8] Программа, реализующая протоколы: репозиторий на Github Соколова Михаила. [Электронный ресурс]. 2024. Дата обновления: 07.05.2024. URL: https://github.com/ISokomi/diploma (дата обращения: 07.05.2024)

6. Приложение

```
import asyncio, math
from itertools import product


class Trivial:
    def __init__(self, n):
        self.n = n
        self.queue = asyncio.Queue(1)

    async def Alice(self, alpha):
        for i in range(1, self.n + 1):
            x = alpha[i]
            print("Alice sends:   ", x)
            await self.queue.put(x)
            await asyncio.sleep(0)

        index = ""
        for i in range(int(math.log(self.n - 1, 2)) + 1):
            bit_of_index = await self.queue.get()
            print("Alice receives index:", bit_of_index)
            index += bit_of_index

        index = int(index, 2) + 1
        print("\nAlice thinks that index is:", index)
        return index

    async def Bob(self, beta):
        for i in range(1, self.n + 1):
            a = await self.queue.get()
            print("Bob receives:  ", a)

            if beta[i] != a:
                index = i

        print()
        index_bit_len = int(math.log(self.n - 1, 2)) + 1
        d = bin(index - 1)[2:].zfill(index_bit_len)
        for i in range(index_bit_len):
            print("Bob sends index:     ", d[i])
            await self.queue.put(d[i])
            await asyncio.sleep(0)

        print("Bob thinks that index is:  ", index)
        return index


class Simple:
    def __init__(self, n):
        self.n = n
        self.queue = asyncio.Queue(1)

    async def Alice(self, alpha):
        lock_A = 0
        last_1_a = 0
        last_1_b = 0

        x = alpha[1]
        print("Alice sends:   ", x)
        await self.queue.put(x)
        await asyncio.sleep(0)

        for i in range(2, self.n, 2):
```

```
            b = await self.queue.get()
            print("Alice receives:", b)
            if b == 1:
                last_1_b = i

            if lock_A == 1:
                x = 0
            else:
                if alpha[i] == b:
                    x = alpha[i + 1]
                else:
                    lock_A = 1
                    x = 1
                    last_1_a = i + 1

            print("Alice sends:   ", x)
            await self.queue.put(x)
            await asyncio.sleep(0)

        if self.n % 2 == 0:
            b = await self.queue.get()
            print("Alice receives:", b, "\n")
            if b == 1:
                last_1_b = self.n

        print("Alice sends lock_A:   ", lock_A)
        await self.queue.put(lock_A)
        await asyncio.sleep(0)

        lock_B = await self.queue.get()
        print("Alice receives lock_B:", lock_B)

        i_A = last_1_a - 1 if lock_A == 1 else self.n
        i_B = last_1_b - 1 if lock_B == 1 else self.n

        index = min(i_A, i_B)
        print("\nAlice thinks that index is:", index)
               return index

           async def Bob(self, beta):
               lock_B = 0
               last_1_a = 0
               last_1_b = 0

               for i in range(1, self.n, 2):
                   a = await self.queue.get()
                   print("Bob receives:  ", a)
                   if a == 1:
                       last_1_a = i

                   if lock_B == 1:
                       x = 0
                   else:
                       if beta[i] == a:
                           x = beta[i + 1]
                       else:
                           lock_B = 1
                           x = 1
                           last_1_b = i + 1

                   print("Bob sends:     ", x)
                   await self.queue.put(x)
                   await asyncio.sleep(0)

               if self.n % 2 == 1:
                   a = await self.queue.get()
```

```
                    print("Bob receives:  ", a, "\n")
                    if a == 1:
                        last_1_a = self.n

                lock_A = await self.queue.get()
                print("Bob receives lock_A:  ", lock_A)

                print("Bob sends lock_B:     ", lock_B)
                await self.queue.put(lock_B)
                await asyncio.sleep(0)

                i_A = last_1_a - 1 if lock_A == 1 else self.n
                i_B = last_1_b - 1 if lock_B == 1 else self.n

                index = min(i_A, i_B)
                print("Bob thinks that index is:  ", index)
                return index


        class Ham3:
            def __init__(self, n):
                self.n = n
                self.queue = asyncio.Queue(1)

                self.r = int(math.log(self.n, 2))
                self.s = self.n - 2**self.r

                self.C_n = []

                table1 = [*([*map(lambda input: [*input], product({0, 1},
                                                    repeat=2**(self.r-1)))])]
                table2 = [*([*map(lambda input: [*input], product({0, 1},
                                                    repeat=self.r-1))])]
                table3 = [*([*map(lambda input: [*input], product({0, 1},
                                                    repeat=2**(self.r - 1) - (self.r - 1)))])]
                self.label = [''.join(map(str, t)) for t in table3]

                for i in range(len(table1)):
                    flag = 1
                    for j in range(self.r - 1):
                        sum = 0
                        for k in range(2**(self.r - 1)):
                            sum += table1[i][k] * table2[k][j]
                        if sum % 2:
                            flag = 0
                            break
                    if flag:
                        self.C_n.append(table1[i])

            def sph(self, x):
                for i in range(len(self.C_n)):
                    c = self.C_n[i]
                    diff = sum([abs(x[k] - c[k]) for k in range(2**(self.r - 1))])
                    if diff == 1:
                        return self.label[i]

            def ind(self, x):
                c = self.C_n[int(self.sph(x), 2)]
                for i in range(2**(self.r - 1)):
                    if x[i] != c[i]:
                        return i + 1

            async def Alice(self, alpha):
                X0 = alpha[1 : self.s + 1]
                X1 = alpha[self.s + 1 : self.s + 2**(self.r - 1) + 1]
                X2 = alpha[self.s + 2**(self.r - 1) + 1 :]
```

```
195.
196.                 print("Sending X0:")
197.                 for i in range(self.s):
198.                     x = X0[i]
199.                     print("Alice sends:    ", x)
200.                     await self.queue.put(x)
201.                     await asyncio.sleep(0)
202.
203.                 print("\nSending sph(X1):")
204.                 sphX1 = self.sph(X1)
205.                 for i in range(2**(self.r - 1) - (self.r - 1)):
206.                     x = sphX1[i]
207.                     print("Alice sends:    ", x)
208.                     await self.queue.put(x)
209.                     await asyncio.sleep(0)
210.
211.                 bit1 = await self.queue.get()
212.                 print("Alice receives bit1:  ", bit1)
213.
214.                 if bit1 == 0:
215.                     index = 0
216.                     for i in range(2**(self.r - 1)):
217.                         b = await self.queue.get()
218.                         print("Alice receives: ", b)
219.
220.                         if X2[i] != b:
221.                             index = i + 1
222.                     print()
223.
224.                     if index != 0:
225.                         bit2 = 0
226.                         print("Alice sends bit2:   ", bit2)
227.                         await self.queue.put(bit2)
228.                         await asyncio.sleep(0)
229.
230.                         d2 = bin(index - 1)[2:].zfill(self.r - 1)
231.                         for i in range(self.r - 1):
232.                             print("Alice sends index:  ", d2[i])
233.                             await self.queue.put(d2[i])
234.                             await asyncio.sleep(0)
235.                         res = self.s + 2**(self.r - 1) + index
236.                     elif index == 0:
237.                         bit2 = 1
238.                         print("Alice sends bit2:   ", bit2)
239.                         await self.queue.put(bit2)
240.                         await asyncio.sleep(0)
241.
242.                         index = self.ind(X1)
243.                         d1 = bin(index - 1)[2:].zfill(self.r - 1)
244.                         for i in range(self.r - 1):
245.                             print("Alice sends index:  ", d1[i])
246.                             await self.queue.put(d1[i])
247.                             await asyncio.sleep(0)
248.                         res = self.s + index
249.                 elif bit1 == 1:
250.                     bit3 = await self.queue.get()
251.                     print("Alice receives bit3:  ", bit3)
252.
253.                     if bit3 == 0:
254.                         d0 = ""
255.                         for i in range(int(math.log(self.s, 2)) + 1):
256.                             bit_of_index = await self.queue.get()
257.                             print("Alice receives index: ", bit_of_index)
258.                             d0 += bit_of_index
259.                         d0 = int(d0, 2) + 1
260.                         res = d0
```

```
                        print()
                    elif bit3 == 1:
                        index = 0
                        for i in range(2**(self.r - 1)):
                            b = await self.queue.get()
                            print("Alice receives: ", b)

                            if X1[i] != b:
                                index = i + 1
                        print()

                        d1 = bin(index - 1)[2:].zfill(self.r - 1)
                        for i in range(self.r - 1):
                            print("Alice sends index: ", d1[i])
                            await self.queue.put(d1[i])
                            await asyncio.sleep(0)
                        res = self.s + index

                print("Alice thinks that index is:", res)
                return res

            async def Bob(self, beta):
                Y0 = beta[1 : self.s + 1]
                Y1 = beta[self.s + 1 : self.s + 2**(self.r - 1) + 1]
                Y2 = beta[self.s + 2**(self.r - 1) + 1 :]

                index = 0
                for i in range(self.s):
                    a = await self.queue.get()
                    print("Bob receives:   ", a)

                    if Y0[i] != a:
                        index = i + 1

                sphY1 = self.sph(Y1)
                sphX1 = ""
                for i in range(2**(self.r - 1) - (self.r - 1)):
                    a = await self.queue.get()
                    print("Bob receives:   ", a)
                    sphX1 += a
                print()

                if index == 0 and sphX1 == sphY1:
                    bit1 = 0
                    print("Bob sends bit1:       ", bit1)
                    await self.queue.put(bit1)
                    await asyncio.sleep(0)

                    print("\nSending Y2:")
                    for i in range(2**(self.r - 1)):
                        x = Y2[i]
                        print("Bob sends:      ", x)
                        await self.queue.put(x)
                        await asyncio.sleep(0)

                    bit2 = await self.queue.get()
                    print("Bob receives bit2:  ", bit2)

                    if bit2 == 0:
                        d2 = ""
                        for i in range(self.r - 1):
                            bit_of_index = await self.queue.get()
                            print("Bob receives index: ", bit_of_index)
                            d2 += bit_of_index
                        d2 = int(d2, 2) + 1
                        res = self.s + 2**(self.r - 1) + d2
```

```
                elif bit2 == 1:
                    d1 = ""
                    for i in range(self.r - 1):
                        bit_of_index = await self.queue.get()
                        print("Bob receives index: ", bit_of_index)
                        d1 += bit_of_index
                    d1 = int(d1, 2) + 1
                    res = self.s + d1
                print()
            elif index != 0:
                bit1 = 1
                print("Bob sends bit1:       ", bit1)
                await self.queue.put(bit1)
                bit3 = 0
                print("Bob sends bit3:       ", bit3)
                await self.queue.put(bit3)
                await asyncio.sleep(0)

                index_bit_len = int(math.log(self.s, 2)) + 1
                d0 = bin(index - 1)[2:].zfill(index_bit_len)
                for i in range(index_bit_len):
                    print("Bob sends index:      ", d0[i])
                    await self.queue.put(d0[i])
                    await asyncio.sleep(0)
                res = index
            elif index == 0 and sphX1 != sphY1:
                bit1 = 1
                print("Bob sends bit1:       ", bit1)
                await self.queue.put(bit1)
                bit3 = 1
                print("Bob sends bit3:       ", bit3)
                await self.queue.put(bit3)
                await asyncio.sleep(0)

                print("\nSending Y1:")
                for i in range(2**(self.r - 1)):
                    x = Y1[i]
                    print("Bob sends:      ", x)
                    await self.queue.put(x)
                    await asyncio.sleep(0)

                d1 = ""
                for i in range(self.r - 1):
                    bit_of_index = await self.queue.get()
                    print("Bob receives index:", bit_of_index)
                    d1 += bit_of_index
                d1 = int(d1, 2) + 1
                res = self.s + d1
                print()

            print("Bob thinks that index is:  ", res)
            return res


    def error_detect(s):
        if (len(s) & (len(s) - 1)) or len(s) < 2 or not all(c in "01" for c in s):
            return True
        else:
            if all(c == "0" for c in s):
                print("Your function is identically equal to 0")
                return True

            if all(c == "1" for c in s):
                print("Your function is identically equal to 1")
                return True

```

```
393.                 return False
394.
395.
396.         def get_input():
397.             s = input("\nEnter a column of function values: ")
398.             while error_detect(s):
399.                 s = input("Enter exactly 2^n zeros and ones:  ")
400.
401.             return s
402.
403.
404.         async def launch():
405.             values = [int(c) for c in get_input()]
406.             n = int(math.log(len(values), 2))
407.             table = [*([*map(lambda input: [*input], product({0, 1}, repeat=n))])]
408.
409.             print("\n Truth table:\n")
410.             for i in range(len(table)):
411.                 print("", *table[i], "|", values[i])
412.
413.             B0 = []
414.             B1 = []
415.             for i in range(len(table)):
416.                 if values[i] == 0:
417.                     B0.append(table[i])
418.                 else:
419.                     B1.append(table[i])
420.
421.             print("\nB0:")
422.             for line in B0:
423.                 print(*line)
424.             print("\nB1:")
425.             for line in B1:
426.                 print(*line)
427.             print()
428.
429.             for i in range(len(B0)):
430.                 for j in range(len(B1)):
431.                     diff = sum([abs(B0[i][k] - B1[j][k]) for k in range(n)])
432.                     if diff == 1:
433.                         alpha = B1[j]
434.                         beta = B0[i]
435.                         break
436.                 if diff == 1:
437.                     break
438.
439.             print("Alice:", *alpha)
440.             print("Bob:  ", *beta)
441.
442.             protocols = {"Simple": Simple, "Trivial": Trivial, "Ham3": Ham3}
443.
444.             for protocol in protocols:
445.                 print("\n\nProtocol ", protocol, ":\n", sep="")
446.                 protocol = protocols[protocol](n)
447.
448.                 f1 = loop.create_task(protocol.Alice(["a"] + alpha))
449.                 f2 = loop.create_task(protocol.Bob(["b"] + beta))
450.                 await asyncio.wait([f1, f2])
451.
452.             # testing
453.             test_mode = 0
454.             if test_mode:
455.                 flag = 0
456.                 for i in range(len(table)):
457.                     for j in range(len(table)):
458.                         diff = 0
```

```
                        index = 0
                        for k in range(n):
                            di = abs(table[i][k] - table[j][k])
                            if di == 1:
                                index = k + 1
                            diff += di
                        if diff == 1:
                            alpha = table[j]
                            beta = table[i]

                            for protocol in protocols:
                                protocol = protocols[protocol](n)

                                f1 = loop.create_task(protocol.Alice(["a"] + alpha))
                                f2 = loop.create_task(protocol.Bob(["b"] + beta))
                                await asyncio.wait([f1, f2])

                                f1_res = f1.result()
                                f2_res = f2.result()
                                if f1_res != f2_res or f1_res != index:
                                    print(alpha, "\n", beta)
                                    print(f1_res, f2_res, index)
                                    flag = 1
            if not flag:
                print("\nAll tests have been passed!")


    if __name__ == "__main__":
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        loop.run_until_complete(launch())
        loop.close()
```