Содержание

1 Введение 3

2 Основные понятия 9

3 Постановка задачи 11

4 Основная часть 12

4.1 Алгоритм минимизации 12

4.2 Генетический алгоритм 16

5 Полученные результаты 23

6 Список литературы 24

7 Приложение А 26

1 Введение

Задача минимизации полиномиального представления булевой функции является важной при проектировании Сверхбольших Интегральных Схем (СБИС), чем меньше сложность полинома - тем он предпочтительнее, так как меньшая сложность дает меньшие размеры и большую скорость работы электронных схем, построенных с использованием данного полинома. Свойства сложения по модулю два делают полиномиальные представления пригодными для задач информационной безопасности [1,2] и квантовых вычислений [3].

Обобщённым полиномом назовём сумму по модулю два элементов некоторого множества мономов, где мономом является множество элементарных конъюнкций переменных и их отрицаний или константа 1. Будем говорить, что полином P_f представляет булеву функцию $f(x_1, \ldots, x_n)$, если для любого двоичного набора $\sigma_1, \ldots, \sigma_n$ выполняется равенство $f(\sigma_1, \ldots, \sigma_n) = P_f(\sigma_1, \ldots, \sigma_n)$. Множество мономов полинома P обозначим через $M(P)$. В один моном каждая переменная входит не более одного раза. Задачей минимизиации назовём задачу поиска полинома минимальной сложности, где сложность полинома определяется как число его слагаемых $L(P) : L(P) = |M(P)|$.

Каждая булева функция от n переменных имеет $2^{3^n-2^n}$ различных полиномиальных представлений. Минимальных полиномов может быть как один, так и несколько.

Сложность булевой функции определим как наименьшую из сложностей полиномов, представляющих данную функцию: $L(f) = \min\limits_{P_f} |M(P_f)|$.

Получение оценок сложности представления булевых функций в различных классах полиномиальных форм - одна из главных задач на сегодняшний день. Обозначим $L(n)$, как наибольшую из сложностей множества всех булевых функций P_2^n от n переменных $L(n) = \max\limits_{f \in F^n}(L(f))$. Для получения верхних оценок используют алгоритмы минимизации булевых функций, а также библиотеки предварительно вычисленных функций. Нижняя оценка сложности для обобщённых полиномиальных представлений была найдена ещё в 1967 [13] $L(n) \geq \frac{2^n}{n} \log_2 3$. Точные значения данной функции были найдены для небольшого числа переменных, например, для $n = 6$ был разработан алгоритм [11], вычисляющий точное значение $L(6) = 15$, для $n = 7$ с помощью алгоритмов минимизации и вычисления самых сложных функций для $n = 6$, были получены оценки $24 \leq L(7) \leq 26$ [19, 20]. Верхняя оценка для $n <= 70$ найдена в [20]: $L(n) \leq \frac{26}{128} \cdot 2^n$, а для $n > 70$ наилучшей оценкой является [21] $L(n) \leq \frac{2^n(2\log_2 n + 2)}{n}$.

Существуют различные подклассы полиномиальных форм булевых функций, которые были единообразно описаны в работе [4] с использованием операторных форм.

Наиболее простым является полиномиальное представление булевой функции в виде полинома Жегалкина. Основным свойством такого представления является, то что для каждой функции из P_2 существует единственный реализующий её полином Жегалкина,

соответственно по вектору значений функции можно определить длину реализующего её полинома Жегалкина.

Большой интерес имеет представление булевых функций с помощью поляризованных полиномов. Основным свойством такого представления является возможность получить различные полиномиальные представления функции, меняя вектор поляризации.

Для обобщённых полиномиальных форм задача нахождения минимального полинома булевой функции является открытой, и сводится к двум направлениям исследований: точным методам, в которых производится поиск минимального полинома для данной функции и приближённым или эвристическим методам, основной целью которых является нахождение небольшого, но не всегда минимального полинома. Основное преимущество эвристических алгоритмов - это скорость их работы.

Предложено множество как эвристических [5, 6, 9, 10, 17, 18], так и точных алгоритмов и подходов для поиска минимального или близкого к минимальному полинома функции [7, 8, 16], но на текущий момент точные методы позволяют работать только с функциями от небольшого числа переменных, чаще всего не более 6-8 и требуют больших вычислительных и временных ресурсов.

Например, в точном алгоритме [7], использующим формализацию задачи минимизации полинома из булевой алгебры в классическую, с дальнейшим её решением при помощи метода нелинейного программирования, для вычисления минимального полинома функции в некоторых случаях потребуются сотни часов. Подход формализации так же используется и в методе [8]. В данном случае задача поиска минимального полинома формулируется в терминах задачи выполнимости булевой функции: существует ли один и более полином фиксированной длины k для булевой функции f от n переменных. Для этого вводятся две матрицы P и Q размера k на n, элементы которых могут принимать значения 1 или 0, причём $P_{j,l}$ или $Q_{j,l} = 1$, если переменная x_l или $\bar{x}_l$ соответственно входит в мономом j. Для каждого значения входной функции $f(x_1, \ldots, x_n) = b$ и соответствующего набора $\tilde{\sigma}$ вводятся $k * n + k + 1$ условий.

Далее эти условия преобразуются в Конъюнктивную Нормальную Форму (КНФ) и подаются на вход SAT-решателю. Данный алгоритм особенно эффективен для функций, у которых минимальный полином имеет небольшую длину.

В отличие от точных, некоторые разработанные эвристические алгоритмы позволяют работать с полиномами и функциями более чем от 16 переменных [6].

Существуют эвристические методы основанные на генетических алгоритмах [9, 10, 17, 18]. Булевы функции имеют естественный вид для генетических алгоритмов, и сама задача нахождения приближённо минимального полинома вариативно формализуется в терминах данного оптимизационного метода.

Основной целью генетический алгоритмов является нахождение таких значений, на которых целевая функция достигает своего максимального (минимального) значения.

Целевой функцией в данном типе алгоритмов является сложная функция, представляющая исходную задачу. С помощью данного метода можно найти приближённый максимум (минимум) целевой функции, но не всегда можно быть уверенным, что получено значение глобального экстремума.

Генетические методы относятся к стохастическим алгоритмам и основаны на идеях теории эволюции и принципах естественного отбора. Поскольку в генетических алгоритмах исходная задача моделируется в виде биологического процесса, то при их описании обычно используются соответствующие биологические термины.

Основной принцип работы генетических алгоритмов заключается в следующей схеме:

1. Генерация начальной популяции
2. Отбор пары особей-родителей
3. Скрещивание пары особей-родителей с некоторой вероятностью, производя потомство
4. Проведение мутации потомков с некоторой вероятностью
5. Отбор особей в новую популяцию
6. Повторение шагов 2-5, пока не будет достигнут критерий окончания алгоритма

Хромосомой называют вектор состоящий из каких-либо элементов (чаще всего чисел). Каждая позиция хромосомы называется геном. Самым простым представлением хромосомы является бинарный вектор, состоящий из нулей и единиц, например, 10011111. Особью или индивидуумом называют непустой набор хромосом, который является потенциальным решением исходной задачи. Популяцией является множество особей, которые представляют множество решений. С помощью оператора приспособленности или пригодности определяется жизнеспособность особи в текущей популяции, чаще всего это критерий или функция, экстремум которой нужно найти, и затем происходит отбор особей в новую популяцию с учётом данной оценки. Скрещивание или рекомбинация - операция при которой две особи обмениваются своими хромосомами. Мутацией же является случайное изменение одного или нескольких генов в хромосоме.

Существует несколько основных подходов к выбору родительской пары для последующего скрещивания.

Самый простой - панмиксия, при таком подходе каждой особи в рамках одной популяции присваивается уникальный номер. Далее выбираются каких-то два случайных неповторяющихся номера, которые и определят пару особей, которые будут участвовать в создании новых особей. Какие-то члены популяции могут принять участие в процессе скрещивания несколько раз, какие-то ни разу. Несмотря на тривиальность подхода, он

является гибким, хотя и эффективность алгоритма, достаточно часто сильно зависит от мощности популяции.

Ещё один подход к оператору выбора родителей заключается в использовании понятия схожести особей. Такой критерий определяется для каждой задачи свой, но самым простым примером может являться расстояние Хемминга между бинарными векторами. При таком методе первый родитель выбирается случайно, а вторым родителем является наиболее близкая или наоборот, максимально далёкая особь из популяции. Такие подходы называются инбридинг и аутбридинг соответственно. Инбридинг приводит к разбиению популяции на отдельные локальные подмножества популяции вокруг потенциальных локальных экстремумов, аутбридинг же позволяет предотвращать сходимость алгоритма к уже найденным решениям, позволяя исследовать новые подпространства решений целевой функции.

Оператор скрещивания применяется сразу же после оператора отбора родителей для получения новых особей, которые являются потомками. Цель скрещивания заключается в том, что созданные потомки должны наследовать генную информацию от обоих родителей.

Скрещивание двух дискретных особей принято называть кроссинговером. Одноточечный кроссинговер происходит следующим образом: пусть имеются две родительские особи, которые были получены оператором отбора родителей, они имеют набор хромосом $X = x_i, i \in [0; l]$ и $Y = y_i, i \in [0; l]$ соотвественно. Случайным образом определяется точка k разрыва внутри хромосомы, в которой обе хромосомы делятся на две части и обмениваются ими $(x_1, \ldots, x_{k-1}, x_k, \ldots, x_l)$, $(y_1, \ldots, y_{k-1}, y_k, \ldots, y_l) \rightarrow (x_1, \ldots, x_{k-1}, y_k, \ldots, y_l)$, $(y_1, \ldots, y_{k-1}, x_k, \ldots, x_l)$, где последние два вектора и являются потомками. Точка разрыва может быть не одна, а на её выбор можно влиять различными критериями, например, как в кроссинговере с уменьшением замены, в которому оператор уменьшения ограничивает кроссинговер, чтобы по возможности всегда создавать новые особи, это осуществляется за счёт выбора точки разрыва, которая должна появиться только там, где гены различаются.

Пространство новых решений или особей в результате работы оператора скрещивания, которое может покрыть генетический алгоритм, жёстко зависит от генофонда популяции, чем более разнообразны особи, тем больше пространство покрытия. При появлении локального экстремума соответствующий генотип будет стремиться вытеснить все остальные особи популяции, и алгоритм может сойтись к ложному оптимуму. Чтобы такого не происходило, в генетических алгоритмах используют оператор мутации.

Оператор мутации происходит после этапа скрещиваний и напрямую влияет на эффективность генетического алгоритма и препятствует преждевременной сходимости. Вероятность мутации может быть как и случайным числом, так и функцией от некоторой характеристики особи или популяции. Непосредственно оператор мутации определяется исходной задачей, но существует нескольно базовых видов, а именно мутация

присоединением, когда к хромосоме особи присоединяется случайный ген из множества всевозможных генов, мутация вставкой, при которой на случайно выбранную позицию хромосомы вставляется случайный ген, мутация удалением при которой убирается из хромосомы случайный ген и мутация обменом в которой два случайных гена обмениваются местами.

Финальным этапом одной итерации генетического алгоритма, является алгоритм отбора особей в новую популяцию для следующего цикла. На данном этапе обычно отталкиваются от численности популяции, она может быть фиксированной, зависеть от генефонда популяции, номера итерации алгоритма или от каких-либо других критериев. Основная цель оператора отбора - ограничить численность популяции при этом сохраняя генофонд и не допуская потери наилучших решений. При оценке пригодности особи для новой популяции чаще всего недостаточно оценивания особи только с точки зрения целевой функции, иначе мутации будут малополезными, и популяция сойдётся в локальном экстремуме.

Примером использования генетического алгоритма в задаче нахождения приближённо минимального полинома булевой функции, может служить алгоритм, описанный и реализованный в статье [5]. В данной работе используется разложение булевой функции в виде суммы: $f(x_1, \ldots, x_n) = \bar{x}_n \wedge f_1 \oplus f_2 \oplus x_n \wedge f_3$, в которой участвуют функции от меньшего числа переменных. f_1 является особью (функцией-параметром) $f_1(x_1, \ldots, x_{n-1})$, остальные функции определяются как $f_2(x_1, \ldots, x_{n-1}) =$
$= f(x_1, \ldots, x_{n-1}, 0) \oplus f_1$, $f_3(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, 1) \oplus f_2$. Оператор приспособленности особи определяется как сложность полинома для минимизируемой функции, полученной подстановкой f_1 в разложение. Целевой функцией задачи является нахождение минимума $L(f_1) + L(f_2) + L(f_3)$. Оператор мутации реализован в виде изменения нескольких (не обязательно идущих подряд) бит в векторе особи, а оператор кроссовера — в виде обмена как произвольными частями векторов, так и в виде обмена остаточных по одной из переменных.

Основной идеей данного алгоритма является предположение о том, что задача минимизиации функции-параметра значительно проще, чем задача минимизации исходной функции f.

Данная идея далее была развита в алгоритме [20] и изменена для систем булевых функций состоящих из 3 и более целевых функций [10]. Особью уже является непосредственно набор из k множеств мономов $\{M_1, \ldots, M_k\}$, где k - количество функций в исходной системе. Функция приспособленности определяется как:

$$J(\{M_1, \ldots, M_k\}) = \left| \bigcup_{i=1}^{k} M_i \right| + \sum_{i=1}^{k} L(f_i \oplus P(M_i)),$$

где $P(M_i)$ - полином, равный сумме всех мономов из множества M_i.

Задача алгоритма сводится к нахождению минимуму функции J по всевозможным наборам мономов. Оператор мутации реализуется в виде добавления или удаления монома в любое из множств. Оператор кроссовера реализуется в виде обмена мономами между соответствующими множествами мономов для двух особей.

В данной ВКР предпринята попытка применить и улучшить некоторые существующие подходы минимизации обобщённых полиномиальных представлений булевых функций.

2 Основные понятия

Пусть задано множество $E_2 = \{0, 1\}$. $E^n = \{\tilde{\sigma_1}, \ldots, \tilde{\sigma_{2^n}}\}, \tilde{\sigma_j} = (\sigma_1, \ldots, \sigma_n), \sigma_k \in E_2$ - множество всех упорядоченных наборов длины n.

Булевой функцией от n переменных (или функцией алгебры логики), назовём отображение $f : E^n \to E_2$. Множество всех булевых функций от n переменных обозначим $P_2(n)$.

Будем полагать, что любая функция из $P_2(n)$ зависит от n переменных из множества $\{x_1, \ldots, x_n\}$. Мономом или элементарной конъюнкцией (ЭК) M_i обозначим выражение вида $M_i = x_{i_1}^{\sigma_{i_1}} \wedge \cdots \wedge x_{i_r}^{\sigma_{i_r}}, \sigma_{i_j} \in E_2, x^0 = \bar{x}, x^1 = x$, причём x_{i_j} - различные булевы переменные, а r является рангом ЭК. Константа 1 - является мономом ранга 0. Будем считать тождественными элементарный конъюнкции, в которые входят одни и те же литералы (возможно, в разном порядке).

Пусть $M_1, \ldots, M_l$ — элементарные конъюнкции. Обобщённой полиномиальной формой булевой функции f, или полиномом назовём выражение вида $P(f) = M_1 \oplus \cdots \oplus M_l$. Монотонной ЭК называется ЭК, в которую все переменные входят без отрицаний. Будем считать, что 1 - это монотонная ЭК ранга 0. Полином, в который входят только монотонные ЭК, называется полиномом Жегалкина.

Будем говорить, что полином P_f представляет булеву функцию $f(x_1, \ldots, x_n)$, если для любого набора $\tilde{\sigma} \in E_2^n$ выполняется равенство $f(\tilde{\sigma}) = P_f(\tilde{\sigma})$. Через $|P_f|$ будем обозначать сложность полинома, которая равняется количеству в нём слагаемых. С помощью $L(f)$ обозначим сложность полинома, представляющего булеву функцию и имеющего минимальное число слагаемых: $L(f) = \min\limits_{P_f} |M(P_f)|$. Под сложностью $L(n)$ класса всех n-местных функций будем понимать сложность самой сложной функции: $L(n) = \max\limits_{f \in P_2(n)} L(f)$.

Длину набора $\tilde{\sigma} = (\sigma_1, \ldots, \sigma_n), \sigma_j \in E_2, 1 \leq j \leq n$ определим как $|\tilde{\sigma}| = n$. Весом набора $\|\sigma\|$ назовём количество ненулевых элементов в наборе: $\|\sigma\| = \sum_{i=1}^{n} \sigma_i$. Множество всех наборов длины n и веса k обозначим E_k^n. На множестве E^n установим частичный порядок, а именно, $(\sigma_1, \ldots, \sigma_n) \leq (\tau_1, \ldots, \tau_n)$ тогда и только тогда, когда $\sigma_i \leq \tau_i$ для всех $i \in \{1, \ldots, n\}$ и обозначим, что набор $\tilde{\tau}$ покрывает набор $\tilde{\sigma}$. Множество всех наборов $S(\tilde{\sigma})$, которые покрыты набором $\tilde{\sigma}$, назовём его тенью.

Пусть Q - некоторое множество наборов, тогда:

$$S(Q) = \bigcup_{\tilde{\sigma} \in Q} S(\tilde{\sigma}).$$

Множество наборов T^n будем называть затеняющим множеством длины n, если

$$\bigcup_{\tilde{\sigma} \in T^n} S(\tilde{\sigma}) = E^n \setminus (1 \ldots 1).$$

Мощность минимального из затеняющих множеств длины n обозначим как R^n.

К.Д.Кириченко был предложен [21] алгоритм построения полиномов для булевой функции от n числа переменных, позволяющий строить полиномы сложности не более чем $R^n + 1$.

3 Постановка задачи

1. Изучить подходы к построению эвристических алгоритмов.
2. Реализовать метод построения обобщённых полиномов на основе статьи К.Д.Кириченко [21].
3. Разработать генетический алгоритм для задачи минимизации обобщённой полиномиальной формы булевой функции.
4. На основе разработанных алгоритмов написать и протестировать программу, реализующую минимизацию представления булевой функции в виде обобщённого полинома.

4 Основная часть

4.1 Алгоритм минимизации

В данной работе был предложен и реализован на языке Python 3.7 алгоритм построения полиномов, на основе алгоритма описанного в статье [21].

Описанный в статье [21] алгоритм позволяет по заданному минимальному затеняющему множеству построить полином сложности не более, чем $R^n + 1$ для любой булевой функции, зависящей от n переменных.

Опишем алгоритм, предложенный в данной ВКР.

Для построения затеняющего множества, используется градиентный алгоритм. Опишем его.

Так как в любом затеняющем множестве $T^n = \{\tilde{\sigma}^1, \ldots, \tilde{\sigma}^l\}$, где l - мощность множества, набор ранга r затеняет только наборы ранга $r-1$, то чтобы построить затеняющее множество достаточно найти подзатеняющее множество каждого ранга $1 \leq r \leq n$. Опишем алгоритм для ранга r, затеняющее множество T^r в начале является пустым:

1. Добавляем в множество T^r набор $T^r \cup \tilde{\sigma}$, где $\tilde{\sigma}$ - набор ранга r, затеняющий наибольшее число наборов из множества $E^{r-1} \setminus S(T^r)$, если таких наборов несколько, то берётся случайный.

2. Проверяем, затенено ли всё подмножество наборов ранга $r-1$

На вход алгоритма построения обобщённого полинома подаётся затеняющее множество $T^n = \{\tilde{\sigma}^1, \ldots, \tilde{\sigma}^l\}$, построенное описанным градиентным алгоритмом, и булева функция f, зависящая от n переменных. На первоначальном этапе по затеняющему множеству T^n строится полином $\Phi = \bigoplus_{\tilde{\sigma} \in T^n} \prod_{\sigma_i=1} x_i$. Для входной функции f строится полином Жегалкина F. Далее строится полином $\Psi = \Phi \oplus F$. Вводится результирующий пустой полином Δ. Если слагаемое $x_1 \ldots x_n$ присутствует в полиноме Ψ, удаляем его из Ψ и добавляем в пустой полином Δ. Упорядочим наборы в затеняющем множестве T^n по убыванию весов. Будем говорить, что элементарная конъюнкция $\prod_{\sigma_i=1}$ соответствует набору $\tilde{\sigma}$.

Далее алгоритм имеет l циклов из двух шагов:

- В полином Δ добавляется импликанта K', $\Delta = \Delta \oplus K'$, которая строится из переменных импликанты K, соответствующей набору $\tilde{\sigma}$, но в которой каждая переменная x_i будет входить с отрицанием, если импликанта соответствующая набору $\{\sigma_1, \ldots, \sigma_{i-1}, 0, \sigma_{i+1}, \ldots, \sigma_n\}$ присутствует в Ψ.

- В полином Ψ добавляется полином Жегалкина от $K \oplus K'$.

Результирующий полином Δ будет эквивалентен исходной функции f.

Данный алгоритм работает за полиномиальное время относительно длины вектора значений исходной функции и позволяет построить полином длины не более $l+1$.

Предложим алгоритм минимизации полинома с помощью тождеств, уменьшающих длину полинома. Введём эти формулы:

1. $(1 \oplus x_{i_1}^{\sigma_{i_1}}) \cdot K = x_{i_1}^{\bar{\sigma}_{i_1}} \cdot K$, (1)
2. $(1 \oplus x_{i_1}^{\sigma_{i_1}} x_{i_2}^{\sigma_{i_2}} \oplus x_{i_1}^{\bar{\sigma}_{i_1}} x_{i_2}^{\bar{\sigma}_{i_2}}) \cdot K = (x_{i_1}^{\bar{\sigma}_{i_1}} x_{i_2}^{\sigma_{i_2}} \oplus x_{i_1}^{\sigma_{i_1}} x_{i_2}^{\bar{\sigma}_{i_2}}) \cdot K$, (2)
3. $(1 \oplus x_{i_1}^{\bar{\sigma}_{i_1}} x_{i_2}^{\sigma_{i_2}} \oplus x_{i_1}^{\sigma_{i_1}} x_{i_2}^{\bar{\sigma}_{i_2}}) \cdot K = (x_{i_1}^{\sigma_{i_1}} x_{i_2}^{\sigma_{i_2}} \oplus x_{i_1}^{\bar{\sigma}_{i_1}} x_{i_2}^{\bar{\sigma}_{i_2}}) \cdot K$, (3)

где $1 \le m \le n$, $1 \le i_j \le n$, K - элементарная конъюнкция, а элементы набора $\tilde{\sigma}$ принимают значения {0,1}, причём $\bar{\sigma}_i = \begin{cases} 0, & \text{если } \sigma_i = 1 \\ 1, & \text{если } \sigma_i = 0 \end{cases}$

Первое (1) тождество является тривиальным, докажем третье (3), используя формулы де Моргана и преобразование $A \oplus B = A \cdot \bar{B} \vee \bar{A} \cdot B$:

$1 \oplus x\bar{y} \oplus \bar{x}y = \overline{x\bar{y}} \oplus \bar{x}y = \overline{x\bar{y}} \cdot \overline{\bar{x}y} \vee x\bar{y} \cdot \bar{x}y = (\bar{x} \vee y) \cdot (x \vee \bar{y}) = \bar{x}x \vee yx \vee \bar{x}\bar{y} \vee y\bar{y} = xyxy \vee \bar{x}\bar{y}\bar{x}\bar{y} = xy \oplus \bar{x}\bar{y}$

Второе (2) тождество доказывается аналогично.

Минимизация заключается в последовательном применении приведённых тождеств к заданному полиному P, пока это возможно.

Приведём пример использования данной минимизации для полинома, построенного описанным алгоритм построения полиномов.

Пример 1. Исходной функцией от $n = 4$ переменных, является булева функция с вектором значений $f = (1100000111100000)$. С помощью описанного градиентного алгоритма было построено затеняющее множество $T = \{(1111), (1110), (1101), (0111), (1100), (0011), (0100)\}$.

1. По затеняющему множеству T строим полином:

$$\Phi = x_1x_2x_3x_4 \oplus x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_2x_3x_4 \oplus x_1x_2 \oplus x_3x_4 \oplus x_2$$

2. Построим полином Жегалкина для входной функции f:

$$F = x_1x_2x_3 \oplus x_1x_3x_4 \oplus x_2x_3x_4 \oplus x_1x_3 \oplus x_2x_3 \oplus x_2 \oplus x_3 \oplus 1$$

3. Построим $\Psi = F \oplus \Phi$:

$$\Psi = x_1x_2x_3x_4 \oplus x_1x_2x_4 \oplus x_1x_3x_4 \oplus x_1x_2 \oplus x_3x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_3 \oplus 1$$

4. Так как в Ψ присутствует слагаемое $x_1x_2x_3x_4$, то

$$\Delta = x_1x_2x_3x_4$$

5. Удаляем слагаемое $x_1x_2x_3x_4$ из Ψ:

$$\Psi = x_1x_2x_4 \oplus x_1x_3x_4 \oplus x_1x_2 \oplus x_3x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_3 \oplus 1$$

6. Наборы в множестве T упорядочены по невозрастанию ранга, T остаётся без изменений

7. По набору (1111) строим $K = x_1x_2x_3x_4$, и получаем $K' = x_1\bar{x}_2\bar{x}_3x_4$, так как $x_1x_2x_4$ и $x_1x_3x_4$ присутствуют в Ψ. Соответственно:

   7.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4$

   7.2 $\Psi = x_1x_2 \oplus x_3x_4 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_4 \oplus x_3 \oplus 1$

8. По набору (1110) строим $K = x_1x_2x_3$, и получаем $K' = \bar{x}_1\bar{x}_2\bar{x}_3$, так как x_1x_2, x_1x_3 и x_2x_3 присутствуют в Ψ. Соответственно:

   8.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3$

   8.2 $\Psi = x_3x_4 \oplus x_1x_4 \oplus x_1 \oplus x_2$

9. По набору (1101) строим $K = x_1x_2x_4$, и получаем $K' = x_1\bar{x}_2x_4$, так как x_1x_4 присутствует в Ψ. Соответственно:

   9.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus x_1\bar{x}_2x_4$

   9.2 $\Psi = x_3x_4 \oplus x_1 \oplus x_2$

10. По набору (0111) строим $K = x_2x_3x_4$, и получаем $K' = \bar{x}_2x_3x_4$, так как x_3x_4 присутствует в Ψ. Соответственно:

    10.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus x_1\bar{x}_2x_4 \oplus \bar{x}_2x_3x_4$

    10.2 $\Psi = x_1 \oplus x_2$

11. По набору (1100) строим $K = x_1x_2$, и получаем $K' = \bar{x}_1\bar{x}_2$, так как x_1 и x_2 присутствуют в Ψ. Соответственно:

    11.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus x_1\bar{x}_2x_4 \oplus \bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2$

    11.2 $\Psi = 1$

12. По набору (0011) строим $K = x_3x_4$, и получаем $K' = x_3x_4$, так как x_3 и x_4 отсутствуют в Ψ. Соответственно:

    12.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus x_1\bar{x}_2x_4 \oplus \bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2 \oplus x_3x_4$

    12.2 $\Psi = 1$

13. По набору (0100) строим $K = x_2$, и получаем $K' = \bar{x}_2$, так как 1 присутствует в Ψ. Соответственно:

13.1 $\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus x_1\bar{x}_2x_4 \oplus \bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2 \oplus x_3x_4 \oplus \bar{x}_2$

13.2 $\Psi = 0$

Нетрудно проверить, что полученный полином Δ длины 8 реализует исходную функцию f.

Используем минимизацию с помощью тождеств (1) и (3):

$\Delta = x_1x_2x_3x_4 \oplus x_1\bar{x}_2\bar{x}_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus x_1\bar{x}_2x_4 \oplus \bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2 \oplus x_3x_4 \oplus \bar{x}_2 =$
$=$ (используя тождество (1): $x_1\bar{x}_2\bar{x}_3x_4 \oplus x_1\bar{x}_2x_4 = x_1\bar{x}_2x_3x_4$, в свою очередь $x_1\bar{x}_2x_3x_4 \oplus$
$\oplus\, \bar{x}_2x_3x_4 = \bar{x}_1\bar{x}_2x_3x_4$) $= x_1x_2x_3x_4 \oplus \bar{x}_1\bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus \bar{x}_1\bar{x}_2 \oplus x_3x_4 \oplus \bar{x}_2 =$
$=$ (используя тождество (3): $x_1x_2x_3x_4 \oplus \bar{x}_1\bar{x}_2x_3x_4 \oplus x_3x_4 = \bar{x}_1x_2x_3x_4 \oplus x_1\bar{x}_2x_3x_4$) $=$
$= \bar{x}_1x_2x_3x_4 \oplus x_1\bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2\bar{x}_3 \oplus \bar{x}_1\bar{x}_2 \oplus \bar{x}_2 =$ (используя тождество (1): $\bar{x}_1\bar{x}_2\bar{x}_3 \oplus \bar{x}_1\bar{x}_2 =$
$= \bar{x}_1\bar{x}_2x_3$) $= \bar{x}_1x_2x_3x_4 \oplus x_1\bar{x}_2x_3x_4 \oplus \bar{x}_1\bar{x}_2x_3 \oplus \bar{x}_2$

С помощью такой минимизации длина полинома сократилась с 8 до 4.

Алгоритм построения с минимизацией была протестирован для n от 3 до 9 на 100 случайных функциях [22]. Для $n \geq 6$ минимизация уменьшила длину полинома на всех входных функциях, как видно из Таблицы 1:

Таблица 1: Результаты тестирования алгоритма минимизации полинома

n	Количество протестированных функций	Средняя длина полинома без минимизации	Средняяя длина полинома после минимизации	Количество функций для которых минимизация не привела к улучшению	Наибольшая разница длин полиномов до и после минимизации
3	100	4,52	2,85	13	5
4	100	7,48	5,61	11	6
5	100	13,46	10,25	1	6
6	100	24,51	19,56	0	9
7	100	42,38	36,51	0	14
8	100	73,49	66,98	0	12
9	100	136,49	123,98	0	20

Так как для разных затеняющих множеств описанный алгоритм построения полинома для одной и той же функции может построить разные полиномы, минимизация таких полиномов на разных затеняющих множествах для одной и той же функции может дать разный результат (4).

Для проверки гипотезы (4), был программно реализован алгоритм, который для входной функции от n переменных строит полиномы с использованием k различных затеняющих множеств, а затем к каждому построенному полиному применяется алгоритм минимизации и выбирается наикратчайший полином из полученных. Алгоритм был протестирован для 100 функций [22] с использованием 100 различных затеняющих множеств построенных градиентным алгоритмом:

Таблица 2: Результаты тестирования алгоритма минимизации полинома с использованием различных затеняющих множеств

n	Количество протестированных функций	Средняя длина полинома без минимизации	Средняяя длина полинома после минимизации	Количество функций для которых минимизация не привела к улучшению	Наибольшая разница длин полиномов до и после минимизации
5	100	13,46	7,58	0	8
6	100	24,66	15,48	0	14
7	100	42,49	31,33	0	17
8	100	75,59	61,52	0	21

Из приведённых в Таблице 2 результатов видно, что алгоритм минимизации на различных затеняющих множествах действительно уменьшает длину полинома.

В результате, была предложена минимизация и реализован метод построения обобщённого полинома булевой функции на основе статьи [21].

4.2 Генетический алгоритм

Для задачи нахождения приближённо минимального обобщённого полиномиального представления булевой функции был разработан генетический алгоритм на языке Python 3.7.

На вход алгоритма подаётся вектор значений функции f для которой нужно найти полином.

Особью в данном алгоритме характеризуется полиномом P, который реализует исходную функцию. У особи есть гены, множество которых представляет собой множество элементарных конъюнкций полинома особи. Аллелью назовём несколько неповторяющихся конъюнкций (генов). Приведём пример особи.

Пример 2.

Особь P: $P = x_1x_2\bar{x}_3x_5 \oplus \bar{x}_1x_2x_5 \oplus x_1\bar{x}_2\bar{x}_4 \oplus x_2 \oplus 1$

Один из генов особи P: $\bar{x}_1x_2x_5$

Множество генов особи P: $\{x_1x_2\bar{x}_3x_5, \bar{x}_1x_2x_5, x_1\bar{x}_2\bar{x}_4, x_2, 1\}$

Одна из аллелей особи P: $\{x_1x_2\bar{x}_3x_5, 1\}$

Оператор скрещивания двух особей представляет собой поиск пары аллелей, которые имеют одинаковый вектор значений. Затем происходит обмен аллелями и результирующие полиномы представляют два новых потомка. Если таких пар несколько, то берётся случайная пара. Приведём пример скрещивания особей.

Пример 3.

Пусть $f = (11100010110011111101110100001010)$ - исходная функция.

Пусть есть две особи P и M представляющие f.

$P = x_1x_2x_3\bar{x}_5 \oplus x_1\bar{x}_2 \oplus \bar{x}_1\bar{x}_3\bar{x}_4 \oplus \bar{x}_2x_4\bar{x}_5 \oplus \bar{x}_1x_2x_3$.

$M = x_1x_3 \oplus x_1x_2 \oplus x_2x_4x_5 \oplus x_3 \oplus \bar{x}_2x_3\bar{x}_4 \oplus \bar{x}_3\bar{x}_4 \oplus x_4x_5 \oplus x_1x_3x_4 \oplus x_1x_2x_3x_5 \oplus \bar{x}_2\bar{x}_3x_4 \oplus x_1x_4$.

В результате поиска, были найдены аллели P_1 особи P и M_1 особи M реализующие одну и ту же функцию f_1:

$P_1 = x_1\bar{x}_2 \oplus \bar{x}_1\bar{x}_3\bar{x}_4$.

$M_1 = x_1x_3 \oplus x_1x_2 \oplus \bar{x}_3\bar{x}_4 \oplus x_1x_3x_4 \oplus x_1x_4$.

$f_1 = (11000000110000001111111100000000)$

В результате обмена аллелями получаем два новых потомка H и G, реализующих исходную функцию f:

$H = x_1x_3 \oplus x_1x_2 \oplus \bar{x}_3\bar{x}_4 \oplus x_1x_3x_4 \oplus x_1x_4 \oplus x_1x_2x_3\bar{x}_5 \oplus \bar{x}_2x_4\bar{x}_5 \oplus \bar{x}_1x_2x_3$

$G = x_1\bar{x}_2 \oplus \bar{x}_1\bar{x}_3\bar{x}_4 \oplus x_2x_4x_5 \oplus x_3 \oplus \bar{x}_2x_3\bar{x}_4 \oplus x_4x_5 \oplus x_1x_2x_3x_5 \oplus \bar{x}_2\bar{x}_3x_4$

Для больших популяций дуальные скрещивания могут занимать достаточно продолжительное время, поэтому был введён второй оператор скрещивания "турнирным" образом. В таком скрещивании участвует 3 и более особей. Поиск пары аллелей, представляющих одну и ту же функции уже у всех особей пока не найдётся хотя бы одно пересечение у двух особей. Далее они обмениваются аллелями и пораждают потомство, турнир на этом завершается.

Оператором выбора родителей является панмиксия, в соответствии с которым каждому члену популяции сопоставляется случайное целое число на отрезке $[1; n]$, где n - количество особей в популяции. Эти числа рассматриваются как номера особей, которые примут участие в скрещивании. Далее выбираются два или более (при турнирном способе) случайных номера, и особи с соответствующими номерами скрещиваются.

Так как поиск пар аллелей может занимать большое число итераций, то он ограничивается. Если за заданное количество итераций пара была не найдена, то скрещивание

оканчивается без потомства.

Также для ускорения скрещиваний, у каждой особи уже посчитанные аллели запоминаются, причём запоминаются как и векторы функций, так и набор генов особи реализующий их. В результате такой оптимизации, для одних и тех же пар особей участвующих в скрещиваниях на разных итерациях алгоритма новые аллели могут быть не посчитаны, так как на начальном этапе уже имеется пересечение, и в его результате могут появиться уже существующие в популяции потомки. Для предотвращения таких случаев, при скрещивании, после нахождения пересечения аллелей, происходит проверка, существуют ли уже особи с таким же набором генов, как и у потенциальных потомков. Если уже существуют - поиск пар продолжается.

Для получения большего генетического разообразия, реализовано несколько операторов мутации.

Первый оператор мутации выбирает случайная аллель особи, которая впоследствии будет мутирована. Вектор значений полинома, который образует аллель, подаётся на вход алгоритму частичного перебора с использований разложений Шеннона. Опишем его:

Для функции от n переменных, возможно составить 3^n различных конъюнкций. Охарактеризуем каждую конъюнкцию её вектором значений и составим словарь отсортированных конъюнкций, в котором реализуется взаимо-однозначное соответствие вектор $\Leftrightarrow$ конъюнкция.

<конъюнкция> $= [i_1, \ldots, i_n | i_k = \{0, 1, 2\}, k = 1 \ldots n, i_k = 0$ соответствует отсутствию переменной x_k, $i_k = 1$ соответствует x_k, $i_k = 2$ соответствует $\bar{x}_k]$

Например <конъюнкция> = [1, 0, 2, 2] будет соответствовать записи $x_1 \wedge \bar{x}_3 \wedge \bar{x}_4$.

<вектор конъюнкций> = вектор значений конъюнкции, как функции от n переменных.

Соответственно словарь, например, для $n = 2$ будет иметь следующий вид:

[{'con': 0, 'value': (0, 0, 0, 0)}, {'con': 1, 'value': (1, 1, 1, 1)}, {'con': [1, 0], 'value': (0, 0, 1, 1)}, {'con': [2, 0], 'value': (1, 1, 0, 0)}, {'con': [0, 1], 'value': (0, 1, 0, 1)}, {'con': [0, 2], 'value': (1, 0, 1, 0)}, {'con': [1, 1], 'value': (0, 0, 0, 1)}, {'con': [2, 1], 'value': (0, 1, 0, 0)}, {'con': [1, 2], 'value': (0, 0, 1, 0)}, {'con': [2, 2], 'value': (1, 0, 0, 0)}]

Каждой конъюнкции присвоем номер [1,k], где k - размер словаря.

Составим дерево поиска. В вершинах дерева у нас будут различные векторы конъюнкций, а также значение полинома в данной вершине.

Корнем дерева всегда будет являться тождественный 0, так как он имеет длину 0 и $0 + P(f) = P(f)$ и в итоговый полином входить не будет.

Каждая ветвь характеризуется полиномом, составленным из конъюнкций в её вершинах. Построим дерево так, чтобы не было никаких повторяющихся ветвей (полиномов), учитывая, что конъюнкции с одинаковыми литералами в разном (не обязательно) порядке мы считаем тождественными:

1. Указываем корневую вершину 0.

2. От неё строим ветви в вершинах которых будут всевозможные конъюнкции.

3. Далее от этих вершин таким же образом строим ветви, причём у дочерних вершин номер конъюнкции не должны быть ниже, чем у родительской вершины. На каждом следующем уровне поддерева соответственно строится на 1 конъюнкцию меньше.

Каждой вершине дадим ещё одну характеристику - её глубина в дереве. Эта длина будет соответствовать длине полинома построенного из этой вершины. Таким образом самая правая граничная ветвь будет иметь глубину 1.

Ограничим глубину дерева числом k - длина полинома Жегалкина для данной функции. Соответсвенно, достигнув глубины $k-1$, перестаём дальше строить подветви.

Чтобы не хранить всё дерево целиком, оптимизируем построение дерева. Обходить дерево будем слева-направо в глубину. На основе словаря и зная номер родительской конъюнкции, можно определить, какую вершину нужно построить и проверить дальше, а предыдущую удалить.

Если текущий вектор полинома оказался равен вектору значений исходной функции, то полином запоминается и его длина становится новой границей глубины дерева.

Для $n \geq 4$ данный алгоритм полного перебора уже работает долго (более 10 минут), поэтому для функций $n \geq 4$ будем использовать разложения Шеннона.

Зададим три разложения любой булевой функции f по Шеннону:

1. $f(x_1, \ldots, x_n) = x_i \wedge f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \oplus \bar{x}_i \wedge f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$

2. $f(x_1, \ldots, x_n) = x_i \wedge (f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \oplus f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)) \oplus$
$\oplus f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$

3. $f(x_1, \ldots, x_n) = \bar{x}_i \wedge (f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \oplus f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)) \oplus$
$\oplus f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$

Рекурсивно раскладываем функцию по всем трём разложениями Шеннона, по каждой переменной и ищем минимальное. Когда подфункции в разложениях становятся зависимы от 3 переменных, тогда для этих подфункций будем искать полином по описанному выше алгоритму перебора.

Данная мутация была написана на языке C# и поточно распараллелена для первого уровня разложений Шеннона.

Приведём пример данной мутации на потомке H из примера 3.

Пример 4.

Пусть H - потомок в результате скрещивания двух особей, $H = x_1x_3 \oplus x_1x_2 \oplus \bar{x}_3\bar{x}_4 \oplus x_1x_3x_4 \oplus x_1x_4 \oplus x_1x_2x_3\bar{x}_5 \oplus \bar{x}_2x_4\bar{x}_5 \oplus \bar{x}_1x_2x_3$.

Случайным образом выбрана аллель $H_1 = x_1x_3 \oplus x_1x_2 \oplus \bar{x}_3\bar{x}_4 \oplus x_1x_3x_4 \oplus x_1x_4 \oplus x_1x_2x_3\bar{x}_5 \oplus \bar{x}_1x_2x_3$ для мутации, которая реализует функцию f_2 с вектором значений $f_2 = (11000000110011111111111100001010)$. Он подаётся на вход описанному алгоритму мутации.

В результате работы алгоритма, получен полином $H_2 = x_1x_2x_3x_5 \oplus \bar{x}_1\bar{x}_3\bar{x}_4 \oplus x_2x_3 \oplus x_1\bar{x}_2$, реализующий функцию f_2.

Заменяя в особи H аллель H_1 на мутированную аллель H_2, получаем обновлённую особь $H = x_1x_2x_3x_5 \oplus \bar{x}_1\bar{x}_3\bar{x}_4 \oplus x_2x_3 \oplus x_1\bar{x}_2 \oplus \bar{x}_2x_4\bar{x}_5$.

Второй мутацией является алгоритм минимизации описанный в первой части с использованием формул сокращения. Мутация применяется целиком или частично к полиному особи, и преимуществом данной мутации является то, что она может уменьшить длину полинома особи.

Третьей мутацией является алгоритм построения полинома описанный в первой части с использованием минимизации по разным затеняющим множествам. Выбирается случайная аллель особи и её вектор значений подаётся на вход алгоритму.

Оператором приспособленности, в данном алгоритме выступает длина полинома особи $l(P)$, чем меньше длина полинома особи - тем она более приспособлена. К примеру, приспособленность особи H из примера 4 будет равняться 5.

Вымиранием назовём такой шаг алгоритма, на котором, используя оператор приспособленности, некоторые особи исключаются из популяции с некоторой вероятностью, зависящей от фазы алгоритма.

Назовём фазой – определённое состояние генетического алгоритма после некоторого количества итераций.

Алгоритм делится на три фазы:

1. Фаза активного роста популяции.

   Основная задача алгоритма на данной фазе - как можно быстрее увеличить размер популяции и нарастить количество посчитанных аллелей у особей. Мутации почти не происходят, количество скрещиваний велико и пропорционально размеру популяции, причём скрещивания преимущественно дуальные. Вымирания особей редки. Фаза заканчивается при достижении заданного для этой фазы размера популяции.

2. Фаза стабилизации

   Цель этой фазы - генетически разнообразить популяцию и остановить её рост. Количество скрещиваний за один цикл алгоритма в данной фазе уже значительно

меньше и преимущественно турнирные. Операторы мутаций и вымирания применяются чаще. Фаза ограничивается заданных для этой фазы числом итераций алгоритма.

3. Фаза вымирания

   В завершающей фазе алгоритма популяция должна постепенно уменьшаться пока не останется последняя особь с самым коротким полиномом, который и будет результатом работы алгоритма.

   Описанное выше разбиение по фазам позволяет контролировать численность популяции. Ниже приведён график (Рис. 1) изменения численности популяции для одной случайной входной функции от $n = 7$ переменных, с ограничением первой фазы до размера популяции в 250 особей, а второй фазы в 15 итераций.

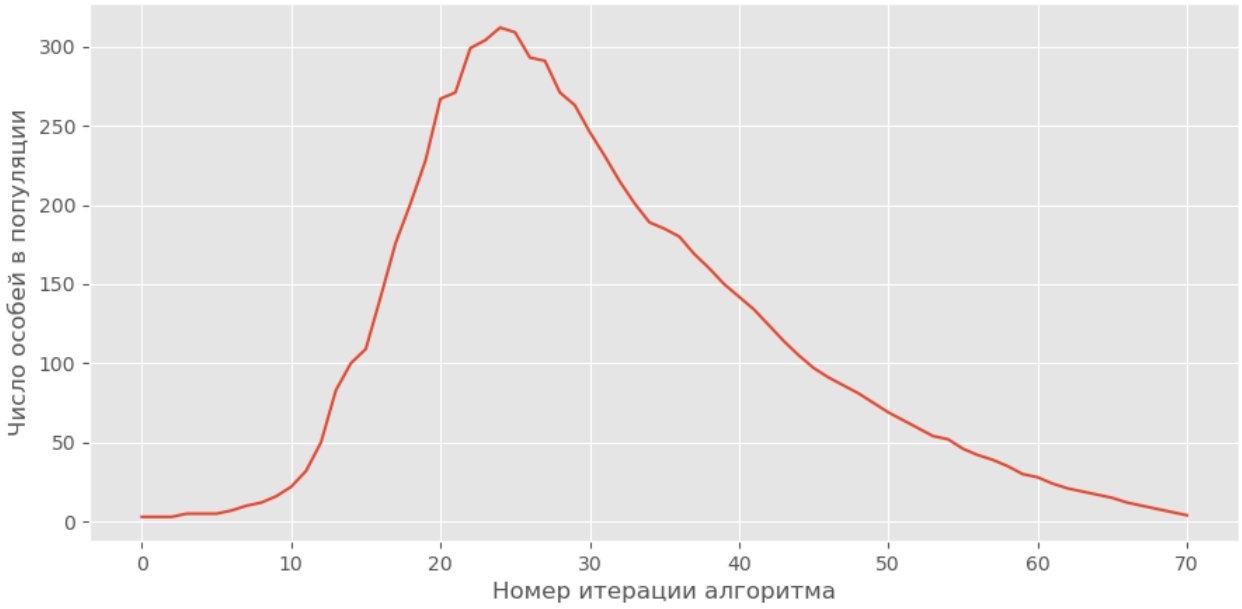

Рис. 1: Пример изменения численности популяции

   На старте алгоритма генерируются 3 особи, первые две с помощью мутаций 1 и 3, и полином Жегалкина, на вход которым подаётся вектор значений входной функции.

В общем виде одну итерацию алгоритма можно описать следующим образом:

1. Определяется число скрещиваний в зависимости от фазы и текущего размера популяции
2. Из популяции выбираются особи для скрещиваний
3. Создаётся потомство
4. Случайные особи из популяции мутируют

5. Оператор вымирания, используя оператор приспособленности, чистит популяцию

6. Проверяется условие окончания алгоритма

Данный генетический алгоритм был протестирован на 10 случайных функциях [22] от $n = 6$ и $n = 7$ переменных и были получены следующие результаты:

- Для n = 6, длина результирующего полинома в среднем 12,8
- Для n = 7, длина результирующего полинома в среднем 22,6

Слабым местом алгоритма является тот факт, что с ростом числа переменных, возрастает количество итераций требуемых для поиска аллелей реализующих одну ту же функцию при скрещивании особей. Для небольших $n \leq 7$ алгоритм работает за приемлемое время для небольших предельных популяций. Преимуществом алгоритма является гибкость выбора операторов мутаций, к использованным в данном алгоритме мутациям можно добавить любой другой алгоритм минимизации или построения обобщённого полинома.

5 Полученные результаты

1. Изучены подходы построения эвристических алгоритмов для задачи минимизации обобщённого полиномиального представления булевой функции.

2. Программно реализован алгоритм построения обобщённого полиномиального представления булевой функции.

3. Предложено и реализовано уменьшение сложности полиномиального представления полученного посредством формул сокращения и использования различных затеняющих множеств в алгоритме из пункта 2.

4. Разработан генетический алгоритм для задачи минимизации обобщённого полиномиального представления булевой функции.

5. Проведено тестирование реализованных алгоритмов.

6 Список литературы

[1] Mizuki, T., Otagiri, T., Sone, H.: An application of ESOP expressions to secure computations. Journal of Circuits, Systems, and Computers 16(2), 191–198 (2007).

[2] Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings. pp. 486–498

[3] Fazel, K., Thornton, M.A., Rice, J.E.: ESOP-based Toffoli gate cascade generation. In: Pacific Rim Conference on Communications, Computers and Signal Processing (2007)

[4] Избранные вопросы теории булевых функций / под ред. С. Ф. Винокурова, Н. А. Перязева. М. : Физматлит, 2001, 67-159

[5] Казимиров А. С. Параллельные генетические алгоритмы в задачах минимизации булевых функций / А. С. Казимиров // Вестн. ТГУ. Приложение. – 2006. – № 17. – С. 226–230.

[6] Soeken, M., Roetteler, M., Wiebe, N., De Micheli, G.: Design automation and design space exploration for quantum computers. In: Design, Automation and Test in Europe. pp. 470–475 (2017).

[7] Papakonstantinou, K.G., Papakonstantinou, G.: A nonlinear integer programming approach for the minimization of Boolean expressions. Journal of Circuits, Systems, and Computers 29(10) (2018)

[8] Heinz Riener, Rüdiger Ehlers, Bruno de O. Schmitt, Giovanni De Micheli, Exact Synthesis of ESOP Forms, In Advanced Boolean Techniques, Springer, pp. 177-194, 2019.

[9] А. С. Казимиров, С. Ю. Реймеров, “Генетический алгоритм синтеза дискретных управляющих систем на базе ПЛМ”, Интеллектуальные системы. Теория и приложения, 20:3 (2016), 151–154

[10] А. С. Казимиров, С. Ю. Реймеров, “Генетический алгоритм поиска минимальных полиномиальных представлений систем булевых функций”, Известия Иркутского государственного университета. Серия Математика, 4:4 (2011), 82–86

[11] Gaidukov A. Algorithm to derive minimum ESOP for 6-variable function // 5th International Workshop on Boolean Problems. — Freiberg, Germany, 2002. — P. 141–148.

[12] Перязев Н. А. Сложность булевых функций в классе полиномиальных поляризованных форм. Алгебра и логика, 34, вып. 3, 1995, с. 323-326.

[13] Even S., Kohavi I., Paz A., On minimal modulo 2 sums of products for switching functions. IEEE Trans. Elect. Comput. (1967), 671-674

[14] А. С. Казимиров, С. Ю. Реймеров, “Вычислительная оценка сложности полиномиальных представлений булевых функций”, Известия Иркутского государственного университета. Серия Математика, 3:4 (2010), 33–43

[15] Кириченко К. Д. Верхняя оценка сложности полиномиальных нормальных форм булевых функций. Дискретная математика, т, 17, вып. 3, 2005, с. 80-88.

[16] Stergiou, S., Papakonstantinou, G.K.: Exact minimization of ESOP expressions with less than eight product terms. Journal of Circuits, Systems, and Computers 13(1), 1–15 (2004).

[17] Stergiou, S., Daskalakis, K., Papakonstantinou, G.: A fast and efficient heuristic ESOP minimization algorithm. In: Proceedings of the 14th ACM Great Lakes symposium on VLSI. pp. 78–81. ACM (2004)

[18] Mishchenko, A., Perkowski, M.: Fast heuristic minimization of exclusive-sums-of-products. Proc. Int. Workshop Appl. ReedMuller Expansion Circuit Des. pp. 242 – 250 (2001)

[19] С. Ф. Винокуров, А. С. Казимиров, О сложности одного класса булевых функций, Известия Иркутского государственного университета. Серия Математика, 2010, том 3, выпуск 4, 2–6

[20] А. С. Казимиров, С. Ю. Реймеров, Вычислительная оценка сложности полиномиальных представлений булевых функций, Известия Иркутского государственного университета. Серия Математика, 2010, том 3, выпуск 4, 33–43

[21] К. Д. Кириченко, “Верхняя оценка сложности полиномиальных нормальных форм булевых функций”, Дискрет. матем., 17:3 (2005), 80–88; Discrete Math. Appl., 15:4 (2005), 351–360.

[22] https://github.com/S-Samoylov/diploma

7 Приложение А

Весь исходный код, функции на которых проводилось тестирование и результаты тестов, можно найти по ссылке [22]. Ниже приведены ключевые фрагменты реализованных программ на языке Python.

Алгоритм построения и минимизации полинома

```
import math
import collections as col
import copy
import numpy as np
import random

NUMBER_OF_SHADOW_COVERAGES = 100

def gen_min_shadow(n):
    shadow = []
    rank = n
    s = set()
    allvec = []

   #generate all rank collections

    for i in range(2**n - 1, -1, -1):
        allvec.append(decToBin(i, n))
    #sort by rank
    all_sorted = sorted(allvec, key = con_rank, reverse = True)
    #cycle
    sum = 0
    for k in range(n):
        one_rank_vec = all_sorted[sum:sum+math.comb(n,k)]
        for its in range(len(one_rank_vec)):
            #now check whole set and create dict {len(locla_s):{shadow_vec}}
            d_vec = col.defaultdict(list)
            for vec in one_rank_vec:
                sh = vec_shadow(vec)
                len_delta = len(s | sh) - len(s)
                d_vec[len_delta].append(vec)
            #find the best
```

```
            max_key = max(d_vec.keys())
            if (max_key == 0): break
            shadow.append(d_vec[max_key][0])
            for x in vec_shadow(d_vec[max_key][0]):
                s.add(tuple(x))
        sum += math.comb(n,k)
        if (check_shadow_full(shadow)): break
    return shadow

def logic_minimize(s): #accept only list with tuples, returns same
    poli = copy.deepcopy(s)
    d = col.defaultdict(list)
    #generate dict {rank:list_of_cons}
    for ik in range(len(poli)):
        if (poli[ik] != 1): poli[ik] = list(poli[ik])
    for x in poli:
        if (x == 1): d[con_rank(x)].append(x)
        else: d[con_rank(x)].append(x)
    z = 0
    while (z != len(poli)):
        #print('start', z, poli)
        x = poli[z]
    #for x in poli:
        if (x == 1):
            if (1 in d.keys()):
                #print(True)
                #optimization has found
                #1. change dict, delete 0-key
                del d[0]
                #2. delete con from poli
                poli.remove(x)
                #3. change poli
                ind = poli.index(d[1][0])
                not_zero_ind = find_not_zero_index(d[1][0])
                if (poli[ind][not_zero_ind] == 1): \\
                    poli[ind][not_zero_ind] = 2
                elif (poli[ind][not_zero_ind] == 2): \\
                    poli[ind][not_zero_ind] = 1
                else: print('error1')
```

```
            #4. delete if replications
            deleted_before = 0
            repl_count = poli.count(poli[ind])
            if (repl_count > 1):
                while(repl_count > 1):
                    if (poli.index(poli[ind]) <= z): deleted_before += 1
                    poli.remove(poli[ind])
                    if (poli.index(poli[ind]) <= z): deleted_before += 1
                    poli.remove(poli[ind])
                    repl_count-=2
            #here is needed to restart cycle with updated x
            z = z - 1 - deleted_before
    else:
        for i in range(len(x)):
            if (x[i] == 0): continue
            new = x[:i] + [0] + x[i+1:]
            if (new in d[con_rank(x) - 1]):
                #optimization has found
                #1. change poli
                ind = poli.index(x)
                if (poli[ind][i] == 1): poli[ind][i] = 2
                elif (poli[ind][i] == 2): poli[ind][i] = 1
                else: print('error1')
                #2. change dict
                d[con_rank(new)].remove(new) #delete subcon
                #d[con_rank(x)][d[con_rank(x)].index(x)] = poli[ind]
                #3. delete con from poli
                poli.remove(new)
                x = copy.deepcopy(poli[ind])
                #4. delete if replications
                deleted_before = 0
                repl_count = poli.count(x)
                if (repl_count > 1):
                    while(repl_count > 1):
                        if (poli.index(x) <= z): deleted_before += 1
                        poli.remove(x)
                        if (poli.index(x) <= z): deleted_before += 1
                        poli.remove(x)
                        d[con_rank(x)].remove(x)
```

```
                            d[con_rank(x)].remove(x)
                            repl_count-=2
                    #here is needed to restart cycle with updated x
                    z = z - 1 - deleted_before
                    break
        #print('end', z, poli)
        z += 1
        if (z < 0 ): z = 0
        if (len(poli) == 0): break
    for ik in range(len(poli)):
        if (poli[ik] != 1): poli[ik] = tuple(poli[ik])
    #delete duplicates
    s = set()
    if (len(set(poli)) != len(poli)):
        for x in set(poli):
            l = poli.count(x)
            if (l == 1 or (l > 1 and l % 2 == 1)): s.add(x)
        poli = list(s)
    return poli

def kir_with_min(vector, cover = 0, param = 1):
    vec = copy.deepcopy(vector)
    #StartKir
    n = int(math.log(len(vec), 2))
    if (cover == 0):
        T_min_set = gen_min_shadow(n)
    else:
        T_min_set = []
        for vect in cover:
            T_min_set.append(list(vect))
    input_vec = [int(x) for x in vec]
    zheg_input_vec = parseZhigalkin(findZhegalkin(input_vec))
    F_Tshadow_poli = T_min_set #format: x1*x2*x3 + x1*x2 + x1*x4 + x2 #Phi
    #step 3
    #Psi
    control_set = set([tuple(x) for x in F_Tshadow_poli if x != 1]) \\
    ^ set([tuple(x) for x in zheg_input_vec if x != 1])
    if ((1 in F_Tshadow_poli) ^ (1 in zheg_input_vec)): control_set.add(1)
    control_set = list(control_set)
```

```
    #step 4-5
    result_poly = [] #delta
    if (tuple([1]*n) in control_set):
        result_poly.append(tuple([1]*n))
        control_set.remove(tuple([1]*n))
    #step 6 always done
    #step 7-8
    for shadow in T_min_set:
        K = copy.deepcopy(shadow)
        K1 = copy.deepcopy(shadow)
        low_rank_list = [x for x in control_set if (con_rank(x) \\
            == con_rank(shadow)-1)]
        if (low_rank_list == [1]):
            K1[shadow.index(1)] = 2
        else:
            for x in low_rank_list:
                sub = (np.array(shadow) - np.array(x)).tolist()
                if (con_rank(sub) == 1 and 1 in sub):
                    K1[sub.index(1)] = 2
        #step 8
        result_poly.append(tuple(K1))
        control_set.append(tuple(K))
        control_set.append(tuple(K1))
        control_set = \\
        = parseZhigalkin(findZhegalkin(value_by_poly(control_set)))
    if (param == 1):
        return logic_minimize(result_poly)
    elif (param == 2):
        polo = logic_minimize(result_poly)
        while (True):
            s = dop_formuls(polo)
            if (s == polo):
                break
            else:
                polo = s
        return logic_minimize(polo)
    else: return result_poly

def dop_formuls(pol):
```

```
s = copy.deepcopy(pol)
one = False
if (1 in s):
    one = True
    s.remove(1)
if (s == []): return [1]
n = len(s[0])
pairs = []#[[type,(vec),(vec), ind1_pol, ind2_pol],...]
for i in range(len(s)-1):
    for j in range(i+1,len(s)):
        typ = checktype(s[i],s[j])
        if (typ != 0):
            pairs.append([typ,s[i],s[j],i,j])
used_ind = set()
for_del = set()
for l in pairs:
    if (l[3] in used_ind or l[4] in used_ind):
        continue
    v1 = l[1]
    v2 = l[2]
    new_con = []
    for i in range(len(v1)):
        if (v1[i] == v2[i]):
            new_con.append(v1[i])
        else:
            new_con.append(0)
    if (tuple(new_con) in s or new_con == [0]*n and one):
        if (new_con == [0]*n and one):
            one = False
        #third condition true
        if (l[0] == 1): #type = 1
            if (new_con != [0]*n):
                if (s.index(tuple(new_con)) in used_ind):
                    continue
                used_ind.add(s.index(tuple(new_con)))
                for_del.add(s.index(tuple(new_con)))
            used_ind.add(l[4])
            used_ind.add(l[3])
            med = tuple(np.array(s[l[3]]) - np.array(s[l[4]]))
```

```
        for i in range(len(med)):
            if (med[i] != 0):
                pr = list(s[l[3]])
                if (pr[i] == 1):
                    pr[i] = 2
                elif (pr[i] == 2):
                    pr[i] = 1
                s[l[3]] = tuple(pr)
                pr = list(s[l[4]])
                if (pr[i] == 1):
                    pr[i] = 2
                elif (pr[i] == 2):
                    pr[i] = 1
                s[l[4]] = tuple(pr)
                break
    elif(l[0] == 2): #type = 2
        if (new_con != [0]*n):
            if (s.index(tuple(new_con)) in used_ind):
                continue
            used_ind.add(s.index(tuple(new_con)))
            for_del.add(s.index(tuple(new_con)))
        used_ind.add(l[4])
        used_ind.add(l[3])
        med = tuple(np.array(s[l[3]]) - np.array(s[l[4]]))
        for i in range(len(med)):
            if (med[i] != 0):
                pr = list(s[l[3]])
                if (pr[i] == 1):
                    pr[i] = 2
                elif (pr[i] == 2):
                    pr[i] = 1
                s[l[3]] = tuple(pr)
                pr = list(s[l[4]])
                if (pr[i] == 1):
                    pr[i] = 2
                elif (pr[i] == 2):
                    pr[i] = 1
                s[l[4]] = tuple(pr)
                break
```

```
    new_s = []
    for i in range(len(s)):
        if (i not in for_del):
            new_s.append(s[i])
    if (one): new_s.append(1)
    return new_s

def gen_some_shadows(n, colect): #returns \\
list of sets with colect number of shadow coverages
    rank = n
    allvec = []

   #generate all rank collections
    shadow_coverages = []
    col_coverages = 0

    for i in range(2**n - 1, -1, -1):
        allvec.append(decToBin(i, n))
    #sort by rank
    all_sorted = sorted(allvec, key = con_rank, reverse = True)

    while (col_coverages < colect):
        #start cycle
        sum = 0
        shadow = []
        s = set()
        for k in range(n):
            one_rank_vec = all_sorted[sum:sum+math.comb(n,k)]
            for its in range(len(one_rank_vec)):
                #now check whole set and create dict
                d_vec = col.defaultdict(list)
                for vec in one_rank_vec:
                    sh = vec_shadow(vec)
                    len_delta = len(s | sh) - len(s)
                    d_vec[len_delta].append(vec)
                #find the best
                max_key = max(d_vec.keys())
                if (max_key == 0): break
                num = random.randint(0,len(d_vec[max_key])-1)
```

```
                shadow.append(d_vec[max_key][num])
                for x in vec_shadow(d_vec[max_key][num]):
                    s.add(tuple(x))
            sum += math.comb(n,k)
            if (check_shadow_full(shadow)):
                break
        set_s = set()
        for vec in shadow:
            set_s.add(tuple(vec))

        if (set_s not in shadow_coverages):
            shadow_coverages.append(set_s)
            col_coverages += 1

    return shadow_coverages
```

Генетический алгоритм

```
import collections as col
import numpy as np
import scipy.stats as sts
import random
import sys
import time
import subprocess
from kir_one_shadow_performed_for_some_shadows import *


MAX_INDIVID_NUM = 100
MAX_ITER_FAZE_1 = 15
RESULT_INDIVID_NUM = 5
START_NUM = 3 #minimum = 2
CROSS_ITERS_START = 100

class population:
popul_count = 0
iter_global = 0
iter_faze = 0
faze = 0
```

```
#bad mutations
bad_kir = 0
bad_shannon = 0
bad_minimize = 0
#good mutations
good_kir = 0
good_shannon = 0
good_minimize = 0

'''

'''

def crossing(self):
cr_num = self.crossing_num()
print('Number of crosses: ', cr_num)
for i in range(cr_num):
print("Crossing ", i)
p_dual, num_tourn = self.crossing_params()
if (p_dual >= 0.1):
bernoulli_rv = sts.bernoulli(p_dual)
tourn_or_dual = bernoulli_rv.rvs(1)[0]
else:
tourn_or_dual = 1
print("Tourn or dual = ", tourn_or_dual)
if (tourn_or_dual == 1):
if (self.crossing_dual(*random.sample(self.individs, k = 2)) == -1):
return -1
else:
if (num_tourn > self.popul_count):
num_tourn = self.popul_count
if (self.crossing_tourn(random.sample(self.individs, k = num_tourn)) == -1):
return -1

def crossing_dual(self, ind1, ind2):
print("Crossing_dual_started")
print("id1 = ", ind1.get_id(), "id2 = ", ind2.get_id())
for k in range(self.crossing_iters):
```

```
for i in range(100):
ind1.add_gen_chain()
ind2.add_gen_chain()
s1 = ind1.gen_chains_vec_gens.keys()
s2 = ind2.gen_chains_vec_gens.keys()
s_before = s1 & s2

s = set()
#check for indentic pols
for vec in s_before:
if (ind1.gen_chains_vec_gens[vec] != ind2.gen_chains_vec_gens[vec]):
s.add(vec)


if (len(s) != 0):

#create childs with gen_chains
print('Crossing on iter = ', k*100)
#choose one vec if >1 crosses
vec = list(s)[random.randint(0, len(s)-1)]
gens_crossed_1 = ind1.gen_chains_vec_gens[vec]
gens_crossed_2 = ind2.gen_chains_vec_gens[vec]
new_gens_set_1 = gens_crossed_2 #set of new gens for 1
new_gens_set_2 = gens_crossed_1 #set of new gens for 2
new_pol1 = ind1.get_set_of_remaining_gens(gens_crossed_1) \\
    ^ new_gens_set_1
defected_gens1 = ind1.get_set_of_remaining_gens(gens_crossed_1) \\
    & new_gens_set_1
new_pol2 = ind2.get_set_of_remaining_gens(gens_crossed_2) \\
    ^ new_gens_set_2
defected_gens2 = ind2.get_set_of_remaining_gens(gens_crossed_2) \\
    & new_gens_set_2
d1 = col.defaultdict(set)#vec:gens
d2 = col.defaultdict(set)#gen:vecs
#полный проход по словарю в поиске цепочек с этими генfvb
for vec, gens_set in ind1.gen_chains_vec_gens.items():
#если гены не пересекаются с изменёнными то наследуем
if (gens_set & (gens_crossed_1 | defected_gens1) == set()):
d1[vec] |= gens_set
```

```
for gen in gens_set:
d2[gen].add(vec)
inh1 = [d1,d2]

d3 = col.defaultdict(set)#vec:gens
d4 = col.defaultdict(set)#gen:vecs
for vec, gens_set in ind2.gen_chains_vec_gens.items():
#если гены не пересекаются с изменёнными то наследуем
if (gens_set & (gens_crossed_2 | defected_gens2) == set()):
d3[vec] |= gens_set
for gen in gens_set:
d4[gen].add(vec)
inh2 = [d3,d4]

#check
if (''.join([str(x) for x in value_by_poly(list(new_pol1))]) != self.vec \\
or ''.join([str(x) for x in value_by_poly(list(new_pol2))]) != self.vec):
pass
else:
acp1 = False
acp2 = False
for ind in self.individs:
if (new_pol1 == ind.gens):
acp1 = True
if (new_pol2 == ind.gens):
acp2 = True
if (acp1 == False): self.add_individ(new_pol1, inh1)#inharit chains
if (acp2 == False): self.add_individ(new_pol2, inh2)#inharit chains
break
print("Crossing_dual_ended")
print("id1 = ", ind1.get_id(), "id2 = ", ind2.get_id())
return 0

def crossing_tourn(self, ind_list):#ind - list of individuals
print("Crossing_tourn_started")
d_ind = col.defaultdict(list)
for ind in ind_list:
print("id = ", ind.get_id())
for k in range(self.crossing_iters):
```

```
for i in range(100):
for ind in ind_list:
ind.add_gen_chain()
#checking sets
s = set()
for ind in ind_list:
for next_ind in ind_list[ind_list.index(ind)+1:]:
s1 = ind.gen_chains_vec_gens.keys()
s2 = next_ind.gen_chains_vec_gens.keys()
perec = s1 & s2
if (perec != set()):
#checking same subpols
for vec in perec:
if (ind.gen_chains_vec_gens[vec] != next_ind.gen_chains_vec_gens[vec]):
s.add(vec)
d_ind[vec].append(ind)
d_ind[vec].append(next_ind)
#if (len(s) != 0): break


if (len(s) != 0):

#create childs with gen_chains
print('Crossing on iter = ', k*100)
for vec in s:
#choice 2 inds
ind1, ind2 = random.sample(d_ind[vec], 2)

gens_crossed_1 = ind1.gen_chains_vec_gens[vec]
gens_crossed_2 = ind2.gen_chains_vec_gens[vec]
new_gens_set_1 = gens_crossed_2 #set of new gens for 1
new_gens_set_2 = gens_crossed_1 #set of new gens for 2
new_pol1 = ind1.get_set_of_remaining_gens(gens_crossed_1) \\
    ^ new_gens_set_1
defected_gens1 = ind1.get_set_of_remaining_gens(gens_crossed_1) \\
    & new_gens_set_1
new_pol2 = ind2.get_set_of_remaining_gens(gens_crossed_2) \\
    ^ new_gens_set_2
defected_gens2 = ind2.get_set_of_remaining_gens(gens_crossed_2) \\
```

```
    & new_gens_set_2
d1 = col.defaultdict(set)#vec:gens
d2 = col.defaultdict(set)#gen:vecs
#полный проход по словарю в поиске цепочек с этими генами
for vec, gens_set in ind1.gen_chains_vec_gens.items():
#если гены не пересекаются с изменёнными то наследуем
if (gens_set & (gens_crossed_1 | defected_gens1) == set()):
d1[vec] |= gens_set
for gen in gens_set:
d2[gen].add(vec)

inh1 = [d1,d2]

d3 = col.defaultdict(set)#vec:gens
d4 = col.defaultdict(set)#gen:vecs
for vec, gens_set in ind2.gen_chains_vec_gens.items():
#если гены не пересекаются с изменёнными то наследуем
if (gens_set & (gens_crossed_2 | defected_gens2) == set()):
d3[vec] |= gens_set
for gen in gens_set:
d4[gen].add(vec)
inh2 = [d3,d4]

#check
if (''.join([str(x) for x in value_by_poly(list(new_pol1))]) != self.vec \\
or ''.join([str(x) for x in value_by_poly(list(new_pol2))]) != self.vec):
    pass
else:
acp1 = False
acp2 = False
for ind in self.individs:
if (new_pol1 == ind.gens):
acp1 = True
if (new_pol2 == ind.gens):
acp2 = True
if (acp1 == False):
self.add_individ(new_pol1, inh1)#inharit chains
print("New individ by tourn")
if (acp2 == False):
```

```
self.add_individ(new_pol2, inh2)#inharit chains
print("New individ by tourn")
if (acp1 == False or acp2 == False):
break
break
print("Crossing_tourn_ended")
return 0

def mutation(self):
if (self.faze == 0):
self.mutation_faze_0()
elif (self.faze == 1):
self.mutation_faze_1()
elif (self.faze == 2):
self.mutation_faze_2()

'''

'''

def mutagen_kir(self, ind):
gens_before = ind.gens
num = random.randint(30, len(ind.gens)-1)
sample = random.sample(ind.gens, num)
old_gens = set(sample)
new_gens = set(kir_with_min(''.join(str(x) for x in value_by_poly(sample))))
if ("".join(str(x) for x in \\
value_by_poly(list(ind.mini_upd(old_gens, \\
new_gens)))) != self.vec):
population.bad_kir += 1
else:
ind.update(old_gens, new_gens)
population.good_kir += 1

def mutagen_shennon(self, ind):
gens_before = ind.gens
num = random.randint(22, len(ind.gens)-1)
sample = random.sample(ind.gens, num)
old_gens = set(sample)
```

```
res = subprocess.check_output(['poli.exe', '2', ''.join(str(x) \\
for x in value_by_poly(sample))], encoding='utf-8')
sharp_res = parse_sharp_out(res,N)
new_gens = set(sharp_res)
if ("".join(str(x) for x in \\
value_by_poly(list(ind.mini_upd(old_gens, new_gens)))) != self.vec):
population.bad_shannon += 1
else:
ind.update(old_gens, new_gens)
population.good_shannon += 1

def mutagen_logic_minimize(self, ind):
try:
gens_before = ind.gens
s = set(logic_minimize(list(ind.gens)))
if ("".join(str(x) for x in \\
value_by_poly(list(ind.mini_upd(s)))) != self.vec):
population.bad_minimize +=1
else:
ind.update(s)
population.good_minimize +=1
except:
f = open('log.txt','a')
f.writelines('Minimize failed with unexpected error\n')
f.close()

'''

'''

def selection(self):
if (self.faze == 0):
self.selection_faze_0()
elif (self.faze == 1):
self.selection_faze_1()
elif (self.faze == 2):
self.selection_faze_2()

def selection_faze_0(self):
```

```
pass

def selection_faze_1(self):
#probability 0.2 kill of > avg_len num
avg_pop_len = 0
k = 0
for ind in self.individs:
avg_pop_len += len(ind.gens)
k += 1
avg_pop_len = int(avg_pop_len/k)
i = 0
while(i < len(self.individs)):
if (len(self.individs[i].gens) > avg_pop_len):
if (random.randint(1,10) >= 9):
self.individs.remove(self.individs[i])
i -= 1
i += 1
def selection_faze_2(self):
#kill max ~0.5 prob
while (self.popul_count < len(self.individs) + 2):
max_ind = 0
max = 0
for ind in range(len(self.individs)):
if (len(self.individs[ind].gens) > max):
max_ind = ind
max = len(self.individs[ind].gens)
self.individs.remove(self.individs[max_ind])

'''

'''


class individual:

'''


'''
```

```
#main part#
vect = input()
N = int(math.log(len(vect), 2))
pop1 = population(vect)
pop1.gen_start()

while(True):
print(pop1.iter_global)
pop1.crossing()
pop1.mutation()
pop1.selection()
if (pop1.end_of_iter() == -1):
break
```