ВЫЧИСЛИТЕЛЬНЫЕ МЕТОДЫ В ДИСКРЕТНОЙ МАТЕМАТИКЕ

DOI 10.17223/20710410/21/11

УДК 519.6

О ЗАДАЧЕ ОПРЕДЕЛЕНИЯ ЛИНЕЙНОЙ И АФФИННОЙ ЭКВИВАЛЕНТНОСТИ ПОДСТАНОВОК

И. В. Панкратов

Национальный исследовательский Томский государственный университет, г. Томск, Россия

E-mail: ivan.pankratov2010@yandex.ru

Представлены результаты экспериментов над программной реализацией алгоритмов определения линейной и аффинной эквивалентности подстановок. Подробно описаны алгоритмы определения линейной эквивалентности между двумя блоками замены, построения класса линейной эквивалентности и определения аффинной эквивалентности. Произведено сравнение оценки трудоёмкости алгоритмов с реальным временем работы.

Ключевые слова: *линейная эквивалентность, аффинная эквивалентность, блок замены, подстановка.*

В работе рассматривается задача определения линейной и аффинной эквивалентности подстановок, описанная в [1]. Подробно описаны и программно реализованы алгоритм LE определения линейной эквивалентности между двумя блоками замены, алгоритм LEC построения класса линейной эквивалентности и алгоритм AE определения аффинной эквивалентности.

1. Постановка задачи

Введём некоторые обозначения:

- $V_n = \{0,1\}^n$ — пространство булевых векторов размерности n;
- $A \cdot B(x) = A(B(x))$ — композиция отображений A и B;
- пару векторов $x, y \in V_n$, такую, что $A(x) = y$, назовём *точкой преобразования* A и обозначим $(x \to y)$.

Отображение $L : V_n \to V_n$ назовём *линейным преобразованием пространства* V_n, если оно удовлетворяет свойству

$$\forall x, y \in V_n \; L(x + y) = L(x) + L(y).$$

Такие преобразования удобно задавать матрицами размера $n \times n$. Квадратная матрица L размера $n \times n$ задаёт линейное преобразование вектор-строк $L(x) = x \cdot L$. Далее будем отождествлять матрицу и задаваемое ею отображение.

Преобразование называется *аффинным*, если его можно представить в виде $A(x) = L(x) + c$, где $L(x)$ — линейное преобразование с матрицей L, а c — константа из V_n. Матрицу L будем называть матрицей аффинного преобразования. Известно, что аффинное преобразование обратимо, если его матрица невырождена.

Две обратимые подстановки S_1 и S_2 называются *линейно* или *аффинно эквивалентными*, если существуют такие линейные или аффинные соответственно отображения A и B^{-1}, что

$$B^{-1} \cdot S_1 \cdot A = S_2.$$

Очевидно, что отображения A и B^{-1} обратимы, то есть их матрицы невырождены. Можно записать эквивалентное условие: существуют такие обратимые линейные или аффинные соответственно отображения A и B, что

$$S_1 \cdot A = B \cdot S_2. \tag{1}$$

Необходимо для пары заданных подстановок S_1, S_2 найти отображения A и B, удовлетворяющие (1), либо убедиться, что таких отображений нет.

2. Алгоритм определения линейной эквивалентности

Рассмотрим задачу определения линейной эквивалентности подстановок S_1 и S_2 на множестве V_n. Для решения этой задачи разработан алгоритм LE [1]. Данный алгоритм пытается построить невырожденные матрицы A и B, удовлетворяющие условию (1).

В алгоритме используются две основные идеи:

1) «эффект иголки» (needlework effect), суть которого в том, что подбор части точек преобразования A позволяет найти точки преобразования B. В свою очередь, новые точки B позволяют получить новую информацию про точки A;

2) «экспоненциальный рост количества догадок» (exponential amplification of guesses), происходящий благодаря линейности преобразований. Этот эффект состоит в том, что, зная k линейно-независимых точек A, можно легко получить 2^k линейных комбинаций этих точек.

Будем использовать следующие множества:

— C_A и C_B — множества точек, для которых соответствующее преобразование (A или B) известно. По построению, эти множества линейно замкнуты, то есть содержат все линейные комбинации известных точек;
— множества N_A и N_B содержат вновь полученные точки, для которых стали известны преобразования (A и B соответственно), но которые не являются линейными комбинациями точек из C_A и C_B;
— U_A и U_B — множества неизвестных точек.

Множества прообразов точек в C_A, N_A и U_A попарно не пересекаются и их объединение есть все векторное пространство V_n, то есть они представляют разбиение векторного пространства. То же верно и для множеств C_B, N_B и U_B. С учётом этого свойства в программной реализации вместо множеств U_A и U_B использованы объединения $C_A \cup N_A$ и $C_B \cup N_B$ соответственно.

Суть алгоритма такова: пока имеются точки в N_A, можно построить соответствующие им точки преобразования B. Если $(x \to y)$ — точка преобразования A, то $(S_2(x) \to S_1(y))$ должна быть точкой преобразования B, поскольку предполагается, что $S_1 \cdot A = B \cdot S_2$. Помимо этого, поскольку преобразования A и B линейны, можно получить точки преобразования B для всех линейных комбинаций известных точек преобразования A с участием точки из N_A. Часть вновь полученных точек преобразования B может оказаться линейно независимой от его известных точек и попасть в множество N_B. Это происходит благодаря нелинейности преобразований S_1 и S_2. Аналогично по точкам из N_B можно получать точки преобразования A.

Когда множества N_A и N_B пусты, необходимо заниматься подбором, то есть строить предположение о точке преобразования A и добавлять её в N_A. Затем можно пытаться достроить преобразования A и B с помощью описанной выше процедуры. Когда найдено достаточное количество точек, строятся матрицы A и B и проверяется, удовлетворяют ли они свойству (1). Если да, то подстановки линейно эквивалентны и найдены соответствующие матрицы, в противном случае последнее предположение отвергается и строится следующее. Если все предположения исчерпаны, делается вывод, что подстановки не линейно эквивалентны.

Для описания алгоритма введём обозначения: $F(W) = \{F(x) : x \in W\}$, $W + c = \{x + c : x \in W\}$, где $W \subseteq V_n$; $c \in V_n$; $F : V_n \to V_n$. Для простоты изложения операции над точками будем обозначать как операции над прообразами, предполагая соответствующие преобразования над образами. Например, в шаге 3.3.2 под $S_2(x+C_A)$ предполагается $\{(S_2(x + c) \to S_1(y + d)) : (c \to d) \in C_A\}$, а в шаге 3.3.3 под $x + C_A$ понимается $\{((x + c) \to (A(x) + A(c))) : (c \to A(c)) \in C_A\}$.

Алгоритм LE

1. $N_A := N_B := \varnothing$; $C_A := C_B := \{(0 \to 0)\}$.
2. **Если** $S_1(0) \neq 0$ и $S_2(0) \neq 0$, добавляем $(S_2^{-1}(0) \to S_1^{-1}(0))$ в C_A и $(S_2(0) \to S_1(0))$ в C_B;
   иначе проверяем, что $S_1(0) = S_2(0) = 0$, в противном случае подстановки не эквивалентны.
3. **Повторяем** в безусловном цикле:
   3.1. **Если** последнее предположение отвергнуто, восстанавливаем состояния C_A и C_B, какие были до предположения; полагаем $N_A := N_B := \varnothing$.
   3.2. **Если** $N_A = N_B = \varnothing$:
      3.2.1. Делаем предположение о значении $A(x)$ для некоторого $x \in U_A$.
         Если все возможные предположения были отвергнуты, то подстановки не эквивалентны.
      3.2.2. Добавляем в N_A точку $(x \to A(x))$.
   3.3. **Пока** $N_A \neq \varnothing$:
      3.3.1. Выбираем некоторую точку $(x \to y) \in N_A$; полагаем $N_A := N_A \setminus \{(x \to y)\}$.
      3.3.2. Полагаем $N_B := S_2(x + C_A) \setminus C_B$.
      3.3.3. Полагаем $C_A := C_A \cup (x + C_A)$.
      3.3.4. **Если** $|N_B| + \log|C_B| > \text{const} \cdot n$:
         Если B — линейное обратимое отображение, порождаем A и проверяем условие (1) на всех точках, оставшихся в U_A и U_B;
         иначе отвергаем последнее предположение.
   3.4. **Пока** $N_B \neq \varnothing$:
      3.4.1. Выбираем некоторую точку $(y \to z) \in N_B$; полагаем $N_B := N_B \setminus \{(y \to z)\}$.
      3.4.2. $N_A := S_2^{-1}(y + C_B) \setminus C_A$.
      3.4.3. $C_B := C_B \cup (y + C_B)$.
      3.4.4. **Если** $|N_A| + \log|C_A| > \text{const} \cdot n$:
         Если A — линейное обратимое отображение, порождаем B и проверяем условие (1) на всех точках, оставшихся в U_A и U_B;
         иначе отвергаем последнее предположение.

Константа const влияет на быстродействие алгоритма. Очевидно, её значение должно быть меньше единицы, иначе условие может стать невыполнимым.

В случае, когда $S_1(0) \neq 0$ и $S_2(0) \neq 0$, как правило, достаточно подобрать одну точку преобразования A, после чего остальные точки находятся с помощью описанной процедуры. Если же $S_1(0) = S_2(0) = 0$, то необходимо перебирать, как минимум, две точки. Объём перебора составляет соответственно 2^n и 2^{2n}. Самый трудоёмкий шаг проверки подобранных точек — построение матриц — имеет трудоёмкость порядка n^3. Итоговая трудоёмкость получается соответственно $\mathrm{O}(n^3 2^n)$ и $\mathrm{O}(n^3 2^{2n})$.

3. Построение класса линейной эквивалентности

Класс линейной эквивалентности определяется каноническим представителем. От выбора канонического представителя класса зависит трудоёмкость его нахождения. В [1] в качестве представителя класса выбрана минимальная подстановка в смысле лексикографического упорядочивания таблицы значений. Там же даётся алгоритм нахождения этого представителя LEC.

Алгоритм LEC последовательно строит минимальную подстановку, эквивалентную заданной, то есть для подстановки S строится минимальная подстановка $R_S = B^{-1} \cdot S \cdot A$. Схема алгоритма аналогична алгоритму LE, используются похожие множества точек:

- C_A и C_B — множества точек одного преобразования, для которых известны соответствующие им точки другого преобразования. Например, если $(x \to y)$ — точка из C_A, то в C_B должна быть точка $(x' \to S(y))$ для некоторого x', и наоборот. Таким образом, для точек из этих множеств известны точки самого отображения R_S;
- множества N_A и N_B содержат точки, для которых ещё не известны соответствующие точки другого преобразования;
- U_A и U_B — множества неизвестных прообразов.

Основной шаг алгоритма — достраивание преобразований A и B оптимальным образом с точки зрения минимизации итогового отображения R_S. Для этого, пока есть точки в N_A, то есть прообразы R_S, для которых ещё не определены образы, выбирается наименьший прообраз $x \in N_A$. Затем выбирается наименьший прообраз $y \in U_B$ и определяются точки A и B так, чтобы добиться выполнения условия $R_S(x) = y$, после чего все множества обновляются с использованием свойства линейности отображений A и B. Если же множество N_A пусто, поступаем наоборот, выбирая наименьший образ $y \in N_B$ и наименьший прообраз $x \in U_A$ и вновь определяя точки A и B, добиваясь $R_S(x) = y$. Это продолжается до тех пор, пока не окажутся пустыми оба множества N_A и N_B. При этом либо станут полностью известны преобразования A и B, либо производится подбор какой-либо точки преобразования A. При подборе из всех полученных вариантов выбирается тот, что доставляет минимум отображению R_S.

Как и в случае алгоритма LE, теоретическая трудоёмкость алгоритма LEC составляет $\mathrm{O}(n^3 2^n)$ и $\mathrm{O}(n^3 2^{2n})$ в зависимости от того, отображает ли подстановка S ноль на ноль.

4. Определение аффинной эквивалентности

Простой алгоритм определения аффинной эквивалентности, основанный на алгоритме LE, подразумевает подбор констант аффинных преобразований A и B из условия (1). Перебор двух констант приводит к необходимости 2^{2n} раз выполнить процедуру LE, что даёт итоговую трудоёмкость $\mathrm{O}(n^3 2^{3n})$.

Другой подход к этой задаче основан на алгоритме LEC. Он позволяет решить задачу, выполнив алгоритм LEC $2 \cdot 2^n$ раз: идея состоит в построении классов эквивалентности для подстановки $S_1(x + a)$ для всех $a \in V_n$ и для подстановки $S_2(x + b)$ для

всех $b \in V_n$. После этого достаточно проверить наличие пересечения между классами первой и второй подстановок.

Алгоритм AE

1. Для всех $a \in V_n$ находим $\mathrm{LEC}(S_1(x+a))$ и помещаем в таблицу T_1.
2. Для всех $b \in V_n$ находим $\mathrm{LEC}(S_2(x)+b)$ и помещаем в таблицу T_2.
3. Если $T_1 \cap T_2 \neq \varnothing$, то подстановки аффинно эквивалентны, иначе нет.

Сложность данного алгоритма равна $\mathrm{O}(n^3 2^{2n})$, поскольку среди подстановок с константами найдётся ровно одна такая, которая отображает ноль на ноль.

5. Экспериментальные данные

Для получения статистических данных для каждого n от двух до десяти построено по 10000 подстановок на векторах размерности n. Для каждой размерности проведено по 1000 экспериментов каждого типа.

Для измерения времени работы алгоритмов использовалась функция Windows API `GetThreadTimes`, которая считает время работы только одного потока, для большей объективности измерений и независимости от загруженности компьютера в целом.

В столбцах табл. 1, 2, 4 перечислены размерность векторного пространства V_n, над которым строились подстановки, среднее $T_{\mathrm{ср}}$ и максимальное $T_{\max}$ время работы алгоритма (в секундах), дисперсия времени работы и разброс, то есть отношение максимального времени к среднему. В табл. 1 приведены результаты измерения времени работы алгоритма LE над двумя случайно выбранными подстановками из общего множества. В табл. 2 приведены аналогичные результаты для алгоритма LEC. Полужирным шрифтом выделены ячейки с неожиданно большими значениями.

Таблица 1

Время работы алгоритма LE

n	$T_{\mathrm{ср}}$	$T_{\max}$	Дисперсия	Разброс
2	0,000234	0,0156	3,6E−06	66,66667
3	0,001841	0,0624	2,88E−05	33,89831
4	0,005242	0,093601	6,99E−05	17,85714
5	0,011513	0,343202	0,000171	29,8103
6	0,023915	0,0624	8,35E−05	2,609263
7	0,094552	0,280802	0,000633	2,969807
8	0,198948	0,561604	0,001218	2,822865
9	0,411624	0,483603	0,002031	1,174865
10	0,840861	0,967206	0,004319	1,150257

По табл. 1 и 2 можно заметить, что среднее время работы алгоритма LEC меньше такового для алгоритма LE для всех размерностей, за исключением размерности 10, где среднее время работы алгоритма LEC превышает таковое для алгоритма LE более чем вдвое. На векторах этой размерности самый долгий эксперимент длился 759,4753 с, хотя самый долгий из остальных — только 3,775224 с. Более детальный анализ показал, что в этом случае алгоритму понадобилось находить подбором три точки преобразования A. Дополнительные опыты показали, что при поиске канонического представителя класса эквивалентности подобное случается достаточно редко. В табл. 3 представлены результаты исследования длины подбора для задач различных размерностей; для каждого k от 1 до 4 указано количество экспериментов, в которых пришлось находить подбором k точек. Видно, что для задач размерности 10 бит подбор трёх точек

Таблица 2

Время работы алгоритма LEC

n	$T_{\text{ср}}$	$T_{\max}$	Дисперсия	Разброс
2	0,000156	0,0156	2,41E−06	100
3	0,001123	0,0156	1,63E−05	13,88889
4	0,00404	0,156001	9,06E−05	38,61004
5	0,009641	0,234002	0,000507	24,27184
6	0,022995	0,951606	0,004902	41,38399
7	0,059046	3,650423	0,067703	61,82299
8	0,177467	**19,39092**	1,276042	**109,2651**
9	0,317727	1,107607	0,032235	3,486031
10	1,769223	**759,4753**	**575,5924**	**429,2705**

случается приблизительно в 1,2 % случаев. Правый столбец таблицы содержит математическое ожидание M количества подстановок, сохраняющих ноль, среди десяти тысяч случайных подстановок указанной размерности. Это значение достаточно близко к количеству подстановок, вызвавших максимально длинный перебор.

Таблица 3

Количество точек, получаемых подбором

	k				
n	1	2	3	4	M
2	7481	2519	0	0	2500
3	4753	4287	960	0	1250
4	5380	3953	650	17	625
5	5721	3942	337	0	312,5
6	5968	3905	127	0	156,25
7	5990	3923	87	0	78,125
8	5956	4006	38	0	39,0625
9	6041	3941	18	0	19,53125
10	6020	3968	12	0	9,765625
11	6117	3876	7	0	4,882813
12	6060	3939	1	0	2,441406

Собрана статистика по алгоритму LEC без учёта одного — самого длительного — эксперимента для каждой размерности. Результаты можно видеть в табл. 4. Далее такую статистику будем обозначать LEC*. Можно заметить, что величина разброса в ней значительно меньше.

Сумма величин среднего времени работы алгоритма LE составила 1,58873 с, LEC — 2,36141 с, а LEC* — 1,57799 с. Можно видеть, что значения для LE и LEC* близки. Они были использованы для нормирования значений времени с тем, чтобы проверить соответствие значений реальных измерений времени работы алгоритма их оценке $\mathrm{O}(n^3 2^n)$. В табл. 5 приведены отношения среднего времени работы алгоритма к теоретической оценке, нормированной по сумме значений.

Таблица 4

Время работы алгоритма LEC без учёта самых длительных экспериментов

n	$T_{\text{ср}}$	$T_{\max}$	Дисперсия	Разброс
2	0,000141	0,0156001	2,17E−06	111
3	0,001109	0,0156001	1,61E−05	14,07042254
4	0,003888	0,0468003	6,75E−05	12,03614458
5	0,009416	0,1716011	0,000457	18,2238806
6	0,022065	0,7020045	0,004042	31,81528662
7	0,055451	3,6192232	0,054834	65,26837511
8	0,158234	18,798121	0,907054	118,7994698
9	0,316937	0,936006	0,031641	2,953291289
10	1,010758	3,7752242	0,324316	3,735041018

Таблица 5

Сравнение экспериментальных значений времени работы с теоретической оценкой

n	LE	LEC
2	7,324469	3,285201
3	8,536171	3,504215
4	5,127128	2,65896
5	2,882911	1,624204
6	1,732779	1,120923
7	2,157124	0,906304
8	1,520328	0,912413
9	1,104615	0,573643
10	0,822492	1,164306

Заключение

Практические измерения показали, что для алгоритма LE разброс значений времени работы, то есть максимальное превышение среднего времени работы для одиночного опыта, очень невелик — около 15–17 % для подстановок размерности 9 и 10 битов. Для алгоритма LEC разброс может превышать 40000 %.

ЛИТЕРАТУРА

1. *Biryukov A., De Canniere C., Braeken A., and Preneel B.* A toolbox for cryptanalysis: linear and affine equivalence algorithms // EUROCRYPT 2003. LNCS. 2003. V. 2656. P. 33–50.