УДК 681.3
Э.И. Ватутин, канд. техн. наук, доцент, кафедра вычислительной техники, ЮЗГУ (e-mail: evatutin@rambler.ru)
А.Д. Журавлев, программист, интернет-портал BOINC.ru (e-mail: alexone07@mail.ru)
О.С. Заикин, канд. техн. наук, н.с., ИДСТУ СО РАН, лаборатория дискретного анализа и прикладной логики (e-mail: zaikin.icc@gmail.com)
В.С. Титов, докт. техн. наук, профессор, зав. кафедрой вычислительной техники, ЮЗГУ (e-mail: titov-kstu@rambler.ru)

УЧЕТ АЛГОРИТМИЧЕСКИХ ОСОБЕННОСТЕЙ ЗАДАЧИ ПРИ ГЕНЕРАЦИИ ДИАГОНАЛЬНЫХ ЛАТИНСКИХ КВАДРАТОВ

В статье приведена постановка задачи получения диагональных латинских квадратов (ДЛК) заданного порядка N. Показано, что данная задача достаточно просто формулируется, однако для ее решения могут потребоваться как использование эвристических методов, так и ряд алгоритмических приемов, относящихся к алгоритмической и высокоуровневой оптимизации программных средств, учитывающих особенности решаемой задачи (формируемой комбинаторной структуры) и позволяющих повысить темп генерации диагональных латинских квадратов. К ним относятся: использование диагонального порядка заполнения элементов ДЛК; использование статических структур данных вместо размещения их в динамической памяти; учет числа возможных значений $\left|S_{ij}\right|$ *для еще не заполненных ячеек квадрата в совокупности с ранним внеочередным заполнением ячеек с* $\left|S_{ij}\right|=1$ *и ранним отсечением неперспективных ветвей дерева комбинаторного перебора с* $\left|S_{ij}\right|=0$*; применение вспомогательных структур данных (одномерных массивов) для быстрого заполнения множеств допустимых элементов* S_{ij}*; отключение технологии Hyper-Threading при однопоточной генерации ДЛК в совокупности с ликвидацией фоновой нагрузки на ядра CPU, не используемые в процессе генерации ДЛК; выбор порядка заполнения ячеек квадрата по критерию* $\left|S_{ij}\right|\to\min$*, что уменьшает арность узлов дерева комбинаторного перебора; использование PGO-компиляции. Для каждой из оптимизаций даны конкретные цифры темпа генерации, измеренного для однопоточной программной реализации на современных процессорах. Показано, что наиболее эффективным является подход на базе метода полного перебора с ранним отсечением неперспективных решений, который при специализированной программной реализации с использованием* N^2 *вложенных циклов обеспечивает темп генерации до 340 000 квадратов в секунду.*

Ключевые слова: латинские квадраты, дискретная комбинаторная оптимизация, эвристические методы, метод взвешенного случайного перебора, метод муравьиной колонии, задача о булевой выполнимости, алгоритмическая и высокоуровневая оптимизация.

***

Диагональный латинский квадрат (ДЛК) порядка N – это квадратная матрица размеров $N\times N$, заполненная элементами некоторого множества $U, \left|U\right|=N$ таким образом, что в каждой строке, в каждом столбце, а также в главной и побочной диагоналях таблицы каждый элемент из U встречается в точности один раз [1]. В дальнейшем в настоящей статье в качестве U будет использовано множество $\{0,1,...,N-1\}$. Диагональные латинские квадраты и системы, построенные на их основе, могут применяться, например, в теории

кодирования, криптографии и в других областях. В настоящей статье предлагается ряд алгоритмов генерации ДЛК порядка 10. Разработка быстрых алгоритмов такого рода аргументируется тем, что с помощью генерации ДЛК можно находить новые комбинаторные объекты, основанные на системах ДЛК (как это было сделано, например, в [2]).

В работе [3] были рассмотрены подходы к заполнению элементов ДЛК, основанные на использовании взвешивающих эвристик, соответствующих случайному (англ. Random Search, сокр. RS) [4] и взвешенному случайному (англ. Weighted Random Search, сокр. WRS) [5] методу, а также методу муравьиной колонии (англ. Ant Colony, сокр. AC) [6–8] и методу полного перебора (англ. Brute Force, сокр. BF) [9]. В работе [10] для генерации латинских квадратов был предложен алгоритм полного перебора, основанный на заполнении элементов формируемого квадрата начиная с диагональных как имеющих наибольшее количество ограничений с последующим заполнением остальных (недиагональных) элементов. Указанный порядок заполнения элементов можно использовать совместно с подходами на базе взвешивающих эвристик, следствием чего следует ожидать сокращение затрат вычислительного времени на получение одного ДЛК и, как следствие, повышение темпа их генерации. С целью убедиться в этом был организован вычислительный эксперимент, в котором с использованием программной реализации на языке Delphi производилась оценка темпа генерации ДЛК, результаты которого приведены в табл. 1 (все замеры произведены на компьютере с процессором Intel Core i7 4770 @ 3,4 ГГц, ядро Haswell).

Таблица 1. Темп генерации ДЛК порядка 10 для программной реализации [3].

Метод	Последовательное заполнение	Диагональное заполнение	Выигрыш
RS	≈ 0 ДЛК/с (0,4 ДЛК/с при $C \le 2$)[1]	0,5 ДЛК/с (18 ДЛК/с при $C \le 2$)[1]	45x
WRS	≈ 0 ДЛК/с (0,7 ДЛК/с при $C \le 2$)[1]	0,8 ДЛК/с (16 ДЛК/с при $C \le 2$)[1]	23x
AC	≈ 0 ДЛК/с (0,05 ДЛК/с при $C \le 7$)[1]	0,14 ДЛК/с	–
BF	≈ 0 ДЛК/с	28 ДЛК/с	–

Замечание 1. C обозначает допустимое число нарушений к квадрате. Оценки для $C > 0$ приведены ввиду того, что при $C = 0$ темп генерации получается слишком маленьким.

Полученные данные подтверждают целесообразность использования диагонального заполнения элементов ДЛК, существенно повышающего темп

генерации. При этом метод полного перебора с диагональным заполнением элементов обеспечивает максимальный темп генерации ДЛК.

В статье [3] подробно описан процесс выбора значения очередного элемента квадрата a_{ij}, который заключается в определении множества S_{ij} допустимых значений элементов с последующим выбором одного из них на базе одной из эвристик (для методов RS, WRS и AC) или последовательного перебора (для метода BF). При этом мощность множества S_{ij} варьируется для различных элементов квадрата, что наиболее просто реализуется в программе с использованием динамических массивов (число элементов множества S_{ij} динамически определяет длину массива). Как известно, динамическая память не является быстрой, поэтому вызывает интерес исследование того, как повлияет отказ от ее использования на темп генерации ДЛК. Результаты соответствующего вычислительного эксперимента приведены в табл. 2.

Таблица 2. Темп генерации ДЛК порядка 10 в зависимости от использования динамической памяти.

Метод	С использованием динамической памяти	Без использования динамической памяти	Выигрыш
RS	0,5 ДЛК/с	1,0 ДЛК/с	2x
WRS	0,8 ДЛК/с	3,1 ДЛК/с	4x
AC	0,14 ДЛК/с	0,2 ДЛК/с	1,4x
BF	28 ДЛК/с	363 ДЛК/с	13x

Анализ полученных данных показывает, что отказ от использования динамической памяти повышает темп генерации ДЛК в несколько раз. Наибольший выигрыш наблюдается для метода полного перебора, что, по-видимому, определяется б**о**льшим вкладом данных операций в общее время работы рекурсивно вызываемой подпрограммы в соответствии с законом Амдала.

В настоящей реализации используется следующий порядок заполнения элементов ДЛК. Сначала первая строка заполняется так, чтобы все элементы в ней шли по порядку (т.е. по сути первая строка всегда фиксирована). Отметим, что любой ДЛК элементарными преобразованиями может быть сведен к ДЛК, в котором первая строка будет иметь именно такой вид (по аналогии с процедурой нормализации для латинских квадратов). Далее заполняется главная диагональ, затем – побочная диагональ, а следом за ними – все остальные незаполненные элементы построчно (рис. 1).

1	2	3	4	5
13	6	14	10	15
16	17	7	18	19
20	11	21	8	22
12	23	24	25	9

Рис. 1. Порядок заполнения элементов ДЛК

Данный порядок заполнения учитывает общее число ограничений на значения каждого из элементов квадрата, однако не учитывает самих значений, которые также могут оказывать существенное влияние. Чтобы учесть данную особенность, после заполнения очередного элемента квадрата будем производить оценку мощностей множеств S_{ij} для всех еще не заполненных элементов. Если для одного из элементов окажется, что $\left|S_{ij}\right|=1$, это означает, что данный элемент имеет единственный вариант заполнения. При этом, если не заполнить его именно этим единственно возможным значением, а использовать данное значение для другого элемента квадрата, связанного проверками с *ij*-м (например, расположенного с ним в одной строке или одном столбце), то заполнение квадрата окажется невозможным («квадрат не сложится»). Ввиду указанной особенности заполнение элементов с $\left|S_{ij}\right|=1$ необходимо осуществлять в обход указанного на рис. 1 порядка. Пример подобного заполнения показан на рис. 2.

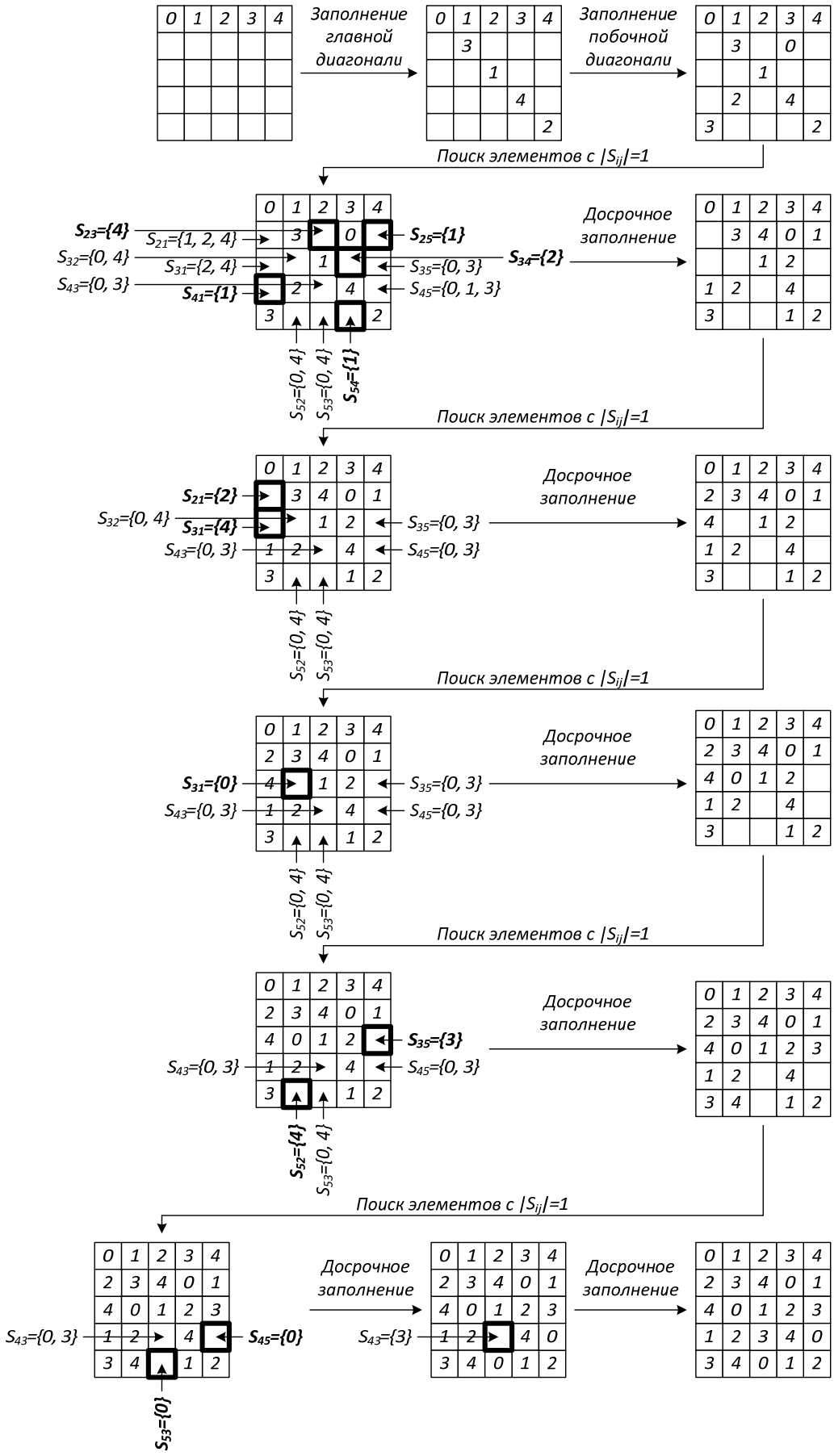

Рис. 2. Иллюстрация, поясняющая принцип досрочного заполнения элементов ДЛК, имеющих единичную мощность множества доступных значений (элементы квадрата с указанной особенностью выделены жирным)

В приведенном на рис. 2 примере указанного заполнения первой строки и двух диагоналей ДЛК порядка 5 достаточно, для того, чтобы сказать, что ему соответствует единственно возможный ДЛК. При этом данное заполнение выполняется итеративно и за полиномиальное время, а не рекуррентно. Если, например, первому элементу второй строки присвоить значение «1» или «4», то «квадрат не сложится» и ни одного корректного ДЛК путем дозаполнения остальных элементов получить не удастся. Следует отметить, что данный принцип не ограничивается только лишь задачей построения ДЛК, он может быть применен в ряде схожих задач, таких как раскраска графов [11], анализ таблиц включений [12] (идентификация «предельных случаев» [13]) при построении разбиений [14, 15] и т.п. Применение данного принципа к задаче генерации ДЛК позволяет существенно увеличить темп (табл. 3).

Таблица 3. Темп генерации ДЛК порядка 10 в зависимости от использования досрочного внеочередного заполнения элементов ДЛК с $\left|S_{ij}\right|=1$.

Метод	Без досрочного заполнения	С досрочным заполнением	Выигрыш
RS	1,0 ДЛК/с	164 ДЛК/с	164x
WRS	3,1 ДЛК/с	203 ДЛК/с	65x
AC	0,2 ДЛК/с	224 ДЛК/с	1120x
BF	363 ДЛК/с	13 000 – 15 000[2] ДЛК/с	36x – 41x

Замечание 2. Темп генерации ДЛК с использованием метода полного перебора зависит от области пространства поиска.

При работе всех рассмотренных выше методов построения ДЛК и приемов их алгоритмической и высокоуровневой оптимизации при заполнении очередного *ij*-го элемента квадрата производится оценка значения множества возможных элементов S_{ij}, а затем один из входящих его состав элементов используется в качестве значения элемента формируемого квадрата. Множество возможных элементов формально можно определить как

$$S_{ij} = U \setminus \bigcup_{k=1}^{N}\{a_{ik}\} \setminus \bigcup_{k=1}^{N}\{a_{kj}\} \setminus \underbrace{\bigcup_{k=1}^{N}\{a_{kk}\}}_{\substack{\text{только для элементов}\\ \text{главной диагонали}\\ \text{с } i=j}} \setminus \underbrace{\bigcup_{k=1}^{N}\{a_{k,\,N-k}\}}_{\substack{\text{только для элементов}\\ \text{побочной диагонали}\\ \text{с } i+j=N}}, \qquad (1)$$

где $U=\{0,1,2,...,N-1\}$ – универсум (множество доступных значений элементов квадрата, определяемое его порядком *N*), « \ » – операция разности множеств, a_{ij} – значение *ij*-го элемента квадрата (для незаполненных

элементов будем полагать $a_{ij} = \varnothing$). Другими словами, из множества всех возможных значений исключаются уже использованные в строке, столбце и на диагоналях, если текущий *ij*-й элемент является диагональным, значения. Чтобы ускорить процесс определения значений множеств S_{ij}, промежуточные значения $s_i = \bigcup_{k=1}^{N} \{a_{ik}\}$, $r_j = \bigcup_{k=1}^{N} \{a_{kj}\}$, $d_1 = \bigcup_{k=1}^{N} \{a_{kk}\}$ и $d_2 = \bigcup_{k=1}^{N} \{a_{k,\,N-k}\}$, $i, j = \overline{0, N-1}$, можно хранить совместно с формируемым ДЛК и изменять одновременно с изменением его элементов. В таком случае расчет по формуле (1) при неизменном заполнении элементов ДЛК значениями фактически оперирует 5 константными значениями и не требует их пересчета. Результаты указанной алгоритмической оптимизации приведены в табл. 4.

Таблица 4. Быстрые проверки множеств значений не использованных элементов с использованием формулы (1).

Метод	Без использования (1)	С использованием (1)	Выигрыш
RS	164 ДЛК/с	662 ДЛК/с	4x
WRS	203 ДЛК/с	740 ДЛК/с	3,6x
AC	224 ДЛК/с	781 ДЛК/с	3,5x
BF	13 000 – 15 000 ДЛК/с	38 000 ДЛК/с	2,5x – 2,9x

Используемая в тестировании машина принимает участие в проектах добровольных распределенных вычислений Gerasim@home [16, 17] и SAT@home [18, 19], которые запускают на всех ядрах код соответствующих расчетных модулей и приоритетом простоя (англ. Idle), что теоретически не должно оказывать сильного влияние за скорость работы приложений пользователя, т.к. они выполняются с более высоким приоритетом Normal. Однако наличие общих для нескольких CPU-ядер кешей и технологии Hyper-Threading [20] способно приводить к деградации темпа формирования ДЛК, что выполняется в одном потоке. Чтобы убедиться в этом, было проведено отдельное тестирование, результаты которого приведены в табл. 5.

Таблица 5. Зависимость темпа генерации ДЛК от наличия фоновой вычислительной нагрузки на CPU.

Метод	С фоновой нагрузкой	Без фоновой нагрузки	Выигрыш
RS	662 ДЛК/с	1 040 ДЛК/с	1,6x
WRS	740 ДЛК/с	1 130 ДЛК/с	1,5x
AC	781 ДЛК/с	1 190 ДЛК/с	1,5x

BF	38 000 ДЛК/с	56 000 ДЛК/с	1,5x

При заполнении элементов квадрата в соответствии с рассмотренным выше порядком кроме раннего внеочередного заполнения элементов с $|S_{ij}|=1$ возможно реализовать также поиск элементов с $|S_{ij}|=0$. Появление указанных элементов на одном из шагов свидетельствует о невозможности выбора значения для *ij*-го элемента квадрата, т.к. $S_{ij}=\varnothing$. Из этого следует, что необходимо либо досрочно прервать формирование ДЛК с использованием одной из эвристических стратегий (RS, WRS или AC), либо досрочно осуществить рекуррентный возврат (в случае стратегии BF). Пример подобной ситуации приведен на рис. 3.

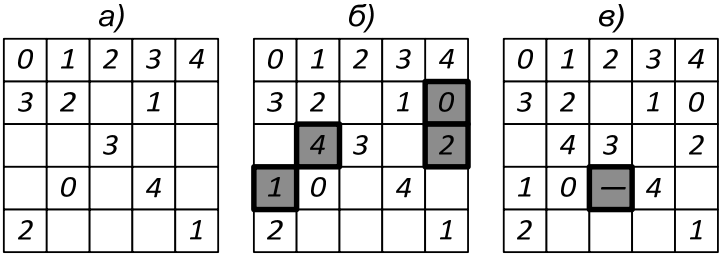

Рис. 3. Начальное «диагональное» заполнение ДЛК (а); досрочное заполнение элементов a_{ij} с $|S_{ij}|=1$ (б); элемент a_{43} не может быть заполнен (в), т.к. $|S_{43}|=0$, следовательно необходимо выполнить комбинаторный возврат и переходить к следующему заполнению, лексикографически идущему за начальным заполнением (а)

Результаты тестирования данного раннего отсечения неперспективных решений приведены в табл. 6.

Таблица 6. Зависимость темпа генерации ДЛК порядка 10 от использования раннего отсечения неперспективных решений

Метод	Без отсечения	С отсечением	Выигрыш
RS	1 040 ДЛК/с	1 040 ДЛК/с	–
WRS	1 130 ДЛК/с	1 170 ДЛК/с	+3,5%
AC	1 190 ДЛК/с	–[3]	
BF	56 000 ДЛК/с	101 000 ДЛК/с	1,8x

Замечание 3. При использовании алгоритма муравьиной колонии подобные отсечения делать нельзя, т.к. это фактически приводит к запрету нахождения муравьями путей в комбинаторном дереве, соответствующих нарушениям в ДЛК, а метод вырождается в метод взвешенного случайного перебора, т.к. движение муравьев продолжается до нахождения первого корректного ДЛК.

Полученные результаты показывают, что наибольший эффект от указанного раннего отсечения достигается для метода полного перебора. Все указанные выше замеры были выполнены для кода, полученного с использованием 32-битного компилятора Delphi 7 со включенной оптимизацией кода и выключенными отладочными проверками. При использовании компилятора Free Pascal 3.0 [21], а также 64-битного компилятора в составе RAD Studio XE 4 существенного увеличения темпа генерации добиться не удалось (для некоторых «удачных» областей пространства поиска он достигал 120 000 ДЛК/с, однако усредненное значение близко к приведенному в табл. 6).

В качестве развития идеи с досрочным заполнением еще не заполненных элементов ДЛК с $\left|S_{ij}\right|=1$ и ранним отсечением неперспективных решений путем нахождения незаполненных элементов ДЛК с $\left|S_{ij}\right|=0$ была опробована следующая стратегия с вариацией порядка заполнения элементов ДЛК в рамках стратегии BF. Прежде всего, как и ранее, рекуррентно производится заполнение первой фиксированной строки квадрата и его диагоналей. Далее для всех еще не заполненных элементов ДЛК происходит нахождение такого элемента a_{ij}, для которого $\left|S_{ij}\right|\to\min$. После этого производится заполнение выбранного элемента значениями из множества возможных значений S_{ij}, для каждого из которых описанные действия рекуррентно повторяются либо до получения полностью заполненного значениями ДЛК, либо до выявления факта невозможности получения корректного ДЛК (при наличии элемента с $\left|S_{ij}\right|=0$) и необходимости комбинаторного возврата. Указанная стратегия автоматически (без каких-либо дополнительных проверок) учитывает рассмотренные выше приемы алгоритмической оптимизации с досрочным заполнением еще не заполненных элементов ДЛК с $\left|S_{ij}\right|=1$ и ранним отсечением неперспективных решений путем нахождения незаполненных элементов ДЛК с $\left|S_{ij}\right|=0$. Схематично данная стратегия с вариацией порядка заполнения элементов ДЛК изображена на рис. 4.

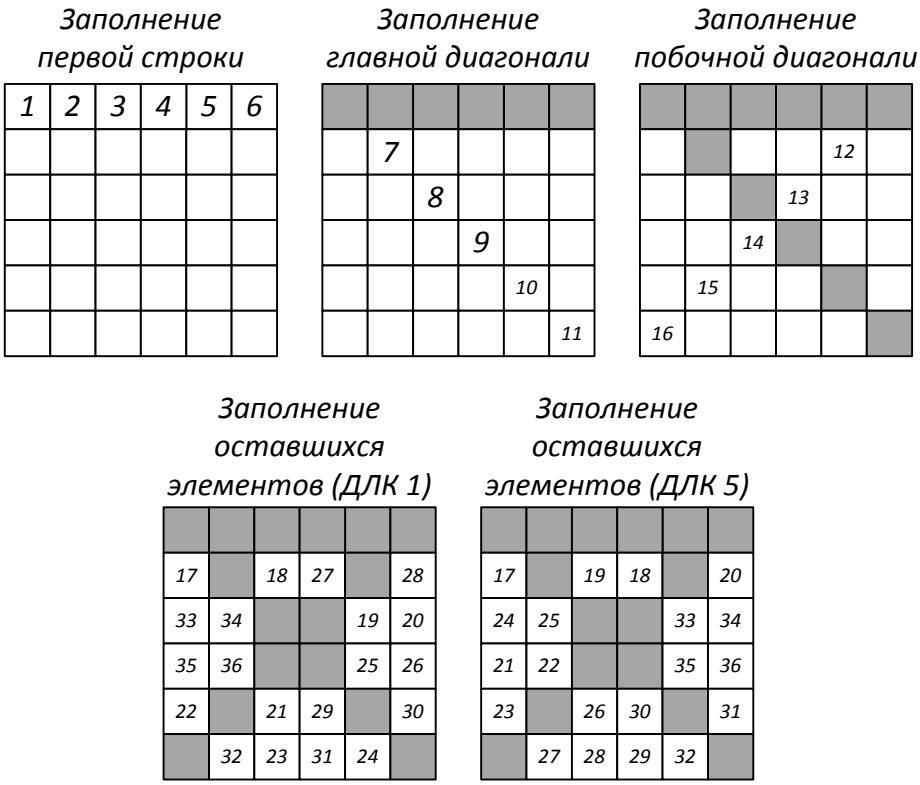

Рис. 4. Примеры вариации порядка заполнения элементов ДЛК в зависимости от начальных значений диагональных элементов (серым выделены заполненные ранее элементы)

В качестве примера взято заполнение двух ДЛК порядка 6, цифрами отмечен порядок заполнения значениями их элементов для различных начальных значений диагональных элементов. Видно, что порядок заполнения зависит от выбранных значений и отличается для различных ДЛК. При использовании данной стратегии совместно с методом полного перебора по сравнению с константным порядком заполнения (например, по строкам, что изображено на рис. 1) темп генерации ДЛК можно дополнительно повысить с 101 000 ДЛК/с (см. табл. 6) до 190 000 – 200 000 ДЛК/с в зависимости от области пространства поиска при использовании реализации на Delphi и до 214 000 – 217 000 ДЛК/с при использовании реализации на Си и компилятора, входящего в состав Microsoft Visual Studio 2012 (с использованием PGO-оптимизации темп можно дополнительно увеличить до 240 000 ДЛК/с).

Разработанная программная реализация метода полного перебора является рекуррентной (рекурсивная подпрограмма явно вызывает сама себя, что в терминах программирования именуется явной рекурсией [22]) и универсальной, т.к. размерность ДЛК и порядок заполнения его элементов могут быть с легкостью изменены. При этом для решения указанной задачи для фиксированного порядка ДЛК N возможно построение специализированной программной реализации с N вложенными циклами, которая теоретически должна работать быстрее за счет отсутствия вызовов подпрограмм (ассемблерная команда CALL), сохранения и последующего

восстановления значений локальных переменных рекурсивной подпрограммы (команды PUSH/POP) и отсутствия косвенных условных переходов и необходимости их предсказания (команда RET). Для этого была взята достаточно простая тестовая задача о ферзях [9, 22], в который были опробованы указанные рекуррентный и итеративный способы ее решения (табл. 7).

Таблица 7. Сравнение времени решения задачи о ферзях для рекуррентного и специализированного (с N циклами) вариантов реализации переборного алгоритма (время в числе тактах CPU).

N	Рекуррентный	Специализированный	Выигрыш
4	$5{,}0\cdot10^3$	$2{,}0\cdot10^3$	2,5x
5	$4{,}0\cdot10^6$	$6{,}3\cdot10^3$	635x
6	$4{,}0\cdot10^6$	$2{,}4\cdot10^5$	17x
7	$3{,}9\cdot10^6$	$1{,}1\cdot10^5$	35x
8	$3{,}3\cdot10^5$	$5{,}0\cdot10^5$	проигрыш 1,5x
9	$3{,}0\cdot10^6$	$2{,}5\cdot10^6$	1,2x
10	$1{,}5\cdot10^7$	$1{,}2\cdot10^7$	1,3x

Полученные результаты показывают, что для случаев малой размерности подход с N вложенными циклами имеет существенное преимущество по-видимому ввиду того, что переменные – счетчики циклов располагаются в регистрах CPU. С ростом размерности все переменные невозможно расположить в регистрах, кроме того растет число выполняемых проверок, негативно влияющее на механизм предсказания условных переходов (англ. Branch Prediction [20]), что приводит к падению выигрыша в скорости решения задачи с десятков раз $(N<8)$ до 20–30% $(9\le N\le10)$, а временами и к проигрышу $(N=8)$ по времени по сравнению с универсальной рекуррентной реализацией.

С целью исследования возможностей данного итеративного подхода была разработана пара специализированных реализаций на языках Delphi и Си, включающие в своем составе 100 вложенных циклов. В них использован ряд описанных выше алгоритмических особенностей, однако порядок заполнения ячеек при их использовании является строго фиксированным, что не позволяет, например, использовать раннее заполнение элементов квадрата с $|S_{ij}|=1$. С их использованием также производилась оценка темпа генерации ДЛК порядка 10, который составил 212 000 ДЛК/с для Delphi-реализации (64-битный компилятор RAD Studio XE 4) и 303 000 ДЛК/с для Си-реализации (64-битный компилятор в составе Microsoft Visual Studio 2012).

При использовании PGO-оптимизации темп дополнительно увеличивается до 340 000 ДЛК/с.

Для генерации ДЛК нами был применен также и SAT-подход. Данный подход состоит в сведении исходной задачи к проблеме булевой выполнимости (SAT) [23]. В рамках данного подхода по исходной задаче строится булева формула (обычно в виде конъюнктивной нормальной формы, далее КНФ), для которой нужно решить SAT-задачу. Под решением SAT-задачи для некоторой КНФ подразумевается поиск выполняющего эту КНФ набора (если он существует), либо констатация отсутствия такового (если он не существует). Под выполняющим набором имеется в виду набор значений входящих в КНФ переменных, который обращает ее в 1. При этом после решения SAT-задачи для построенной КНФ можно эффективно перейти к решению исходной задачи. Ранее (например, в [2]) SAT-подход был успешно применен для поиска новых пар ортогональных ДЛК порядка 10.

Для настоящего исследования нами была сделана новая КНФ, кодирующая задачу поиска ДЛК порядка 10. Для этого использовались принципы т.н. «наивной» кодировки (см., например, [2]), согласно которой для кодирования каждой ячейки ДЛК порядка 10 используются 10 булевых переменных. Таким образом, в полученной КНФ оказалось 1 000 булевых переменных. Общее количество дизъюнктов в этой КНФ составило 14 720 – в них записаны все ограничения на строки, столбцы и диагонали (главную и побочную) в ДЛК. Для поиска выполняющих наборов данной КНФ (а в ней каждый такой набор соответствует некоторому ДЛК) мы использовали SAT-решатель cryptominisat [24] 4-ой версии. Этот решатель был выбран по той причине, что он позволяет находить заданное количество выполняющих наборов КНФ (обычно SAT-решатели ориентированы на поиск ровно одного выполняющего набора). Эксперименты проводились на одном ядре процессора Intel i7 4500U. В таблице 8 приведены результаты по генерации ДЛК с помощью SAT-подхода. Решатель cryptominisat был запущен в трех вариантах – с требованием сгенерировать 1 000, 10 000 и 100 000 ДЛК соответственно.

Таблица 8. Зависимость темпа генерации ДЛК с помощью SAT-решателя cryptominisat от количества сгенерированных ДЛК.

Сгенерированное количество ДЛК	Время генерации	Темп генерации
1 000	0,34 с	2 941 ДЛК/с
10 000	8,528 с	1 173 ДЛК/с
100 000	504,468 с	198 ДЛК/с

Из табл. 8 видно, что изначально довольно невысокая (по сравнению с большинством описанных выше подходов) скорость генерации ДЛК в процессе работы SAT-решателя значительно падает. Это объясняется тем, что после нахождения очередного выполняющего набора (и соответствующего ему очередного ДЛК) cryptominisat добавляет к исходным дизъюнктам КНФ новый дизъюнкт, запрещающий повторное нахождение этого же выполняющего набора. В результате этого действия увеличивается время, необходимой SAT-решателю для нахождения следующих выполняющих наборов. При этом довольно быстро количество добавленных таким образом дизъюнктов начинает превышать количество дизъюнктов в исходной КНФ, поэтому темп генерации ДЛК быстро уменьшается.

Таким образом, в результате сопоставления темпа генерации рассмотренных выше способов можно сделать вывод о том, что наиболее эффективным является метод полного перебора, программная реализация которого является специализированной для конкретной размерности задачи N и базируется на использовании N^2 вложенных циклов и использовании быстрого построения множества допустимых элементов с использованием (1). Другие способы, основанные на универсальной рекуррентной программной реализации, эвристической или SAT-генерации ДЛК, являются существенно более медленными.

Литература

1. Colbourn C.J., Dinitz J.H. Handbook of Combinatorial Designs. Second Edition. Chapman&Hall, 2006. 984 p.
2. Заикин О.С., Кочемазов С.Е. Поиск пар ортогональных диагональных латинских квадратов порядка 10 в проекте добровольных распределенных вычислений SAT@home // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. 2015. Т. 4, № 3. С. 95–108.
3. Ватутин Э.И., Журавлев А.Д., Заикин О.С., Титов В.С. Особенности использования взвешивающих эвристик в задаче поиска диагональных латинских квадратов // Известия Юго-Западного государственного университета. Серия: Управление, вычислительная техника, информатика. Медицинское приборостроение. 2015. № 3 (16). С. 18–30.
4. Ватутин Э.И., Колясников Д.В., Мартынов И.А., Титов В.С. Метод случайного перебора в задаче построения разбиений граф-схем параллельных алгоритмов // Многоядерные процессоры, параллельное программирование, ПЛИС, системы обработки сигналов. Барнаул: Барнаул, 2014. С. 115–125.
5. Ватутин Э.И., Дремов Е.Н., Мартынов И.А., Титов В.С. Метод взвешенного случайного перебора для решения задач дискретной комбинаторной оптимизации // Известия ВолГТУ. Серия: Электроника, измерительная техника, радиотехника и связь. № 10 (137). Вып. 9. 2014. С. 59–64.

6. Dorigo M. Optimization, Learning and Natural Algorithms // PhD thesis. Politecnico di Milano, Italie, 1992.
7. Ватутин Э.И., Титов В.С. Анализ результатов применения алгоритма муравьиной колонии в задаче поиска пути в графе при наличии ограничений // Известия Южного федерального университета. Технические науки. 2014. № 12 (161). С. 111–120.
8. Ватутин Э.И., Титов В.С. Об одном подходе к использованию алгоритма муравьиной колонии при решении задач дискретной комбинаторной оптимизации // Интеллектуальные и информационные системы (Интеллект 2015). Тула, 2015. С. 8–13.
9. Ватутин Э.И., Титов В.С., Емельянов С.Г. Основы дискретной комбинаторной оптимизации. М.: Инфра-М, 2016. 271 с.
10. Заикин О.С., Ватутин Э.И., Журавлев А.Д., Манзюк М.О. Применение высокопроизводительных вычислений для поиска троек взаимно частично ортогональных диагональных латинских квадратов порядка 10 // Параллельные вычислительные технологии (ПаВТ'2016). Принята к опубликованию.
11. https://ru.wikipedia.org/wiki/Раскраска_графов
12. Ватутин Э.И., Зотов И.В. Построение блоков разбиения в задаче декомпозиции параллельных управляющих алгоритмов // Материалы и упрочняющие технологии – 2003. Т. 2. Курск, 2003. С. 38–42.
13. Ватутин Э.И., Волобуев С.В., Зотов И.В. Комплексный сравнительный анализ качества разбиений при синтезе логических мультиконтроллеров в условиях присутствия технологических ограничений // Параллельные вычисления и задачи управления (PACO'08). М.: Институт проблем управления им. В.А. Трапезникова РАН, 2008. С. 643–685.
14. Комбинаторно-логические задачи синтеза разбиений параллельных алгоритмов логического управления при проектировании логических мультиконтроллеров / Э.И. Ватутин, И.В. Зотов, М.Ю. Сохен, В.С. Титов. Курск: изд-во КурскГТУ, 2010. 200 с.
15. Ватутин Э.И. Проектирование логических мультиконтроллеров. Синтез разбиений параллельных граф-схем алгоритмов. Saarbrücken: Lambert Academic Publishing, 2011 г. 292 с.
16. Vatutin E.I., Titov V.S. Voluntary distributed computing for solving discrete combinatorial optimization problems using Gerasim@home project // Distributed computing and grid-technologies in science and education: book of abstracts of the 6th international conference. Dubna: JINR, 2014. PP. 60–61.
17. http://gerasim.boinc.ru
18. Заикин О.С., Посыпкин М.А., Семёнов А.А., Храпов Н.П. Опыт организации добровольных вычислений на примере проектов OPTIMA@home и SAT@home // Вестник Нижегородского университета им. Н.И. Лобачевского. 2012. № 5-2. С. 340–347.
19. http://sat.isa.ru/pdsat/

20. Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. Intel press, order number 325462-052US, 2014. 3439 p.
21. http://www.freepascal.org
22. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi (учебное пособие). М.: Аргамак-Медиа, 2014. 352 с.
23. Biere A., Heule V., van Maaren H., Walsh T. (eds.). Handbook of Satisfiability. IOS Press, 2009. 980 p.
24. Soos M., Nohl K. Extending SAT Solvers to Cryptographic Problems // Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09), 2009. pp. 244–257.