ИНФОРМАТИКА

2005 **октябрь-декабрь** **№ 4**

УДК 519.7

В.И. Романов

ПРОГРАММНЫЕ СРЕДСТВА ДЛЯ РЕШЕНИЯ ЛОГИКО-КОМБИНАТОРНЫХ ЗАДАЧ

Предлагается описание средств для программирования трудоемких алгоритмов логико-комбинаторного характера, основанных на представлении информации булевыми векторами и матрицами. Такие средства определяются классами в языке программирования C++. Приводятся результаты экспериментов по сравнению времен исполнения популярных логических операций, реализованных в разработанных классах.

Введение

Среди всего множества задач, решаемых при помощи вычислительной техники в настоящее время, можно выделить логико-комбинаторные задачи [1, 2]. Очень часто математическая модель, используемая для их формулировки, базируется на представлении информации в виде булевых векторов и матриц. Для удобства программирования при решении такого рода задач в языке C++ были разработаны специальные классы *CBV* и *CBM* [3], реализующие объекты «булев вектор» и «булева матрица» соответственно. По аналогии с ними чуть позже банк программных инструментов пополнился классами *CTB* и *CTM* [4] для поддержки объектов «троичный вектор» и «троичная матрица». Указанные средства в течение нескольких лет активно применялись при решении задач в областях логического проектирования, искусственного интеллекта, криптоанализа [5] и ряде других. Учитывая высокую трудоемкость алгоритмов логико-комбинаторного характера, вопросы оптимизации создаваемых программ остаются актуальными, несмотря на рост вычислительных ресурсов современных компьютеров.

В настоящей работе представлены материалы исследований, связанных с совершенствованием программного инструментария, ориентированного на программирование задач, базирующихся на булевых объектах.

1. Функциональная альтернатива перезагрузки операторов

Одним из декларируемых преимуществ объектно-ориентированного подхода при программировании на языке C++ [6] является возможность перезагрузки операторов языка. Действительно, перезагрузка операторов способна обеспечить высокий уровень понимания текста программы и естественность программного представления используемой математической модели.

Чем поддерживается реализация перезагрузки операторов? Все современные компиляторы обеспечивают трансляцию выражений, в основе которых лежит построение обратной польской записи и обеспечение описываемых ею вычислений. При использовании типовых операндов компилятор строит последовательность машинных команд, вычисляющих заданное выражение. При использовании сложных операндов, задаваемых атрибутами рассматриваемого класса, вычисление выражения строится путем последовательного вызова методов, определенных для перезагрузки операторов в этом классе.

В классе *CBV* булев вектор описывается массивом байтов, каждый бит в которых соответствует отдельному разряду вектора, и тремя дополнительными атрибутами, задающими фактическую длину вектора в битах и байтах, а также длину захваченного участка оперативной памяти (она может быть больше фактической длины в расчете на будущее увеличение вектора).

Набор методов, реализованных в рамках класса *CBV*, включал функции побитового доступа, функции сдвига, функции конкатенации и выделения нового вектора из существующего, защищенные функции перераспределения памяти и ряд других. Кроме того, в данном классе была выполнена перезагрузка оператора присваивания и операторов покомпонентной дизъюнкции (|), конъюнкции (&), сложения по модулю 2 (^) и разности (–), а также операторов

сравнения (==, !=, >=, <=, >, <), основанных на проверке отношений эквивалентности и поглощения между операндами операции.

Анализ машинного кода строящихся программ показал наличие возможностей его оптимизации за счет сокращения накладных расходов, которые возникают при трансляции выражений, содержащих перечисленные выше логические операции.

Предположим, что необходимо произвести вычисление нового булева вектора *b1* по формуле *b1 = b2 & b3 | b4*. Это делается следующим образом:

– захватывается участок оперативной памяти *temp1* по размеру вектора;

– выполняется цикл покомпонентной конъюнкции ($temp1_i = b2_i \,\&\, b3_i$);

– захватывается участок оперативной памяти *temp2* по размеру вектора;

– выполняется цикл покомпонентной дизъюнкции ($temp2_i = temp1_i \mid b4_i$);

– освобождается участок памяти *temp1*;

– выполняется групповая пересылка из *temp2* в *b1* (реализация присваивания);

– освобождается участок памяти *temp2*.

Вместе с тем тот же результат может быть получен путем последовательного выполнения циклов:

– покомпонентной конъюнкции ($b1_i = b2_i \,\&\, b3_i$);

– покомпонентной дизъюнкции ($b1i = b1_i \mid b4_i$).

Однако компилятор так не делает, потому что заложенные в нем алгоритмы трансляции ориентированы на универсальную обработку произвольных выражений и при реализации одного оператора никаких предположений об использовании его результата не делается.

Указанные накладные расходы, связанные с выделением и освобождением памяти, могут быть сокращены следующим образом. Вместе с перезагруженными операторами в класс *CBV* включаются методы-функции, предназначенные для выполнения тех же самых операций:

```
void AssignDiz(int nBitLength, const BYTE* v1, const BYTE* v2);
void AssignCon(int nBitLength, const BYTE* v1, const BYTE* v2);
void AssignXor(int nBitLength, const BYTE* v1, const BYTE* v2);
void AssignDif(int nBitLength, const BYTE* v1, const BYTE* v2);
```

После выполнения такой функции объект получает значение, равное результату соответствующей логической операции над операндами, указываемыми ее вторым и третьим аргументами. Первый аргумент задает длину участвующих в операции булевых векторов.

С точки зрения программного кода изменения не очень значительны. Конструкция

```
b1 = b2 | b3;
```

заменяется

```
b1.AssignDiz(b2.GetBitLength(), (BYTE*) b2, (BYTE*) b3);
```

Сокращение накладных расходов и, как следствие, скорость выполнения фрагмента программы, в котором в вектор записывается результат логической операции, возрастают весьма существенно (рис. 1). При этом выигрыш зависит от длины участвующих в операции векторов.

Кроме указанных функций в состав класса были включены функции реализации многократных логических операций (за исключением разности):

```
void AssignDiz(int nBitLength, int Num, BYTE* v1, ...);
void AssignCon(int nBitLength, int Num, BYTE* v1, ...);
void AssignXor(int nBitLength, int Num, BYTE* v1, ...);
```

Данные функции теоретически не ограничены по количеству аргументов и позволяют за одно обращение выполнить покомпонентную логическую операцию для произвольного набора операндов. Первый аргумент функции задает длину участвующих в операции булевых векторов, второй – их количество. Оценки эффективности применения такого рода функций в зависимости от длины вектора и числа участвующих в операции операндов показаны на рис. 2.

Откладываемый по оси ординат коэффициент показывает отношение времени выполнения функции ко времени выполнения соответствующего оператора.

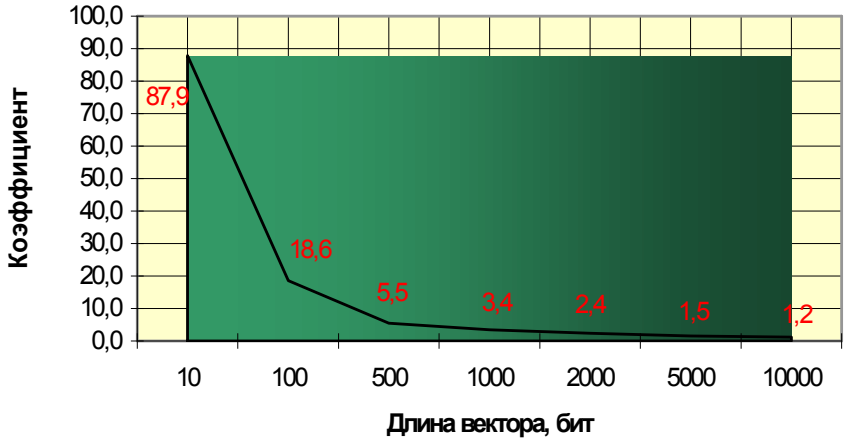
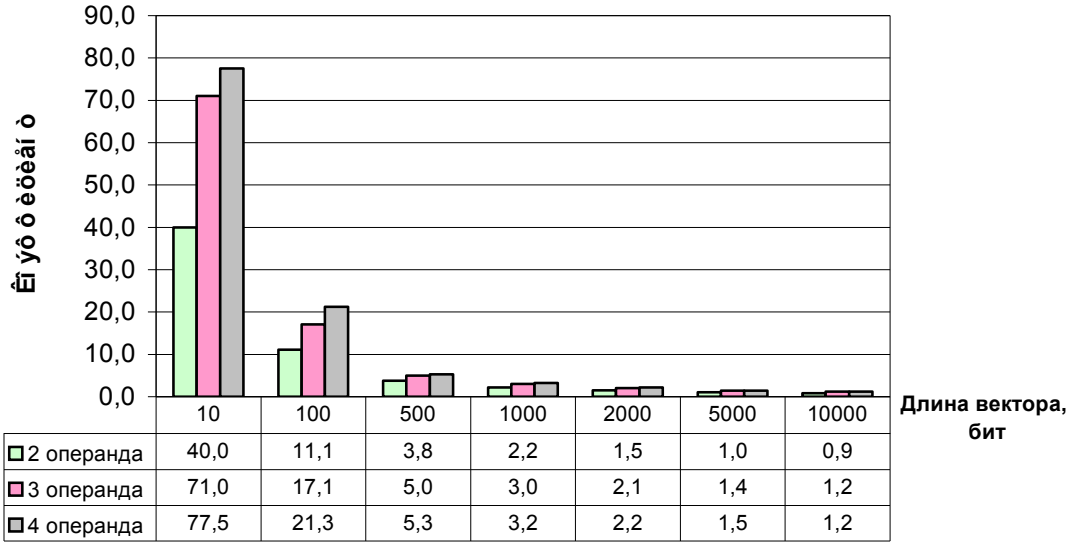


Рис. 1. Повышение быстродействия программы при замене оператора присваивания *b = b1| b2* на функцию *b.AssignDiz(b1,b2)*



	10	100	500	1000	2000	5000	10000
2 операнда	40,0	11,1	3,8	2,2	1,5	1,0	0,9
3 операнда	71,0	17,1	5,0	3,0	2,1	1,4	1,2
4 операнда	77,5	21,3	5,3	3,2	2,2	1,5	1,2

Рис. 2. Повышение быстродействия программы при замене операторов многократной дизъюнкции на соответствующую функцию b.AssignDiz (len, num (BYTE*)b1, …)

Подобная методика повышения быстродействия программ была реализована и в классе булевых матриц *CBM*, основной особенностью которого является то, что для этого класса перезагрузка операторов логических операций осуществляется на уровне отдельных строк, а не матрицы в целом. В связи с этим в данный класс были добавлены функции записи строки, являющейся результатом вычисления некоторой логической операции SetRowDiz, SetRowCon, SetRowXor, SetRowDif.

Следует отметить, что поскольку класс *CBM* часто используется для представления не только булевых матриц, но и их миноров, то он содержит реализацию логических операций над строками такого минора. Практически это выражается в наличии в методах класса дополнительного аргумента-маски (булева вектора), единицы которой отмечают принадлежащие минору столбцы матрицы. Реализация логической операции в этом случае дополняется конъюнкцией с этой маской. Для обеспечения возможности повышения эффективности логических операций и для случая моделирования минора в состав методов класса *CBM* включены варианты новых функций с аргументом-маской. Таким образом, имеются варианты записи результата дизъюнкции:

– двух операндов (b1 = b2 | b3)

```
void SetRowDiz(int nRow, const BYTE* v1, const BYTE* v2);
```

– двух операндов с маской (b1 = b2 | b3 & mask)

```
void SetRowDiz(const BYTE* mask, int nRow, const BYTE* v1, const BYTE* v2);
```

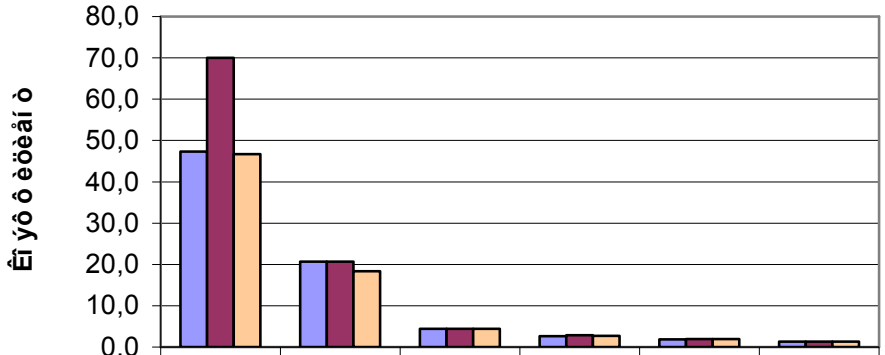
– *Num* операндов (b1 = b2 | b3 | b4)

```
void SetRowDiz(int nRow, int Num, BYTE* v1, ...);
```

– *Num* операндов с маской (b1 = b2 | b3 | b4 & mask)

```
void SetRowDiz(const BYTE* mask, int nRow, int Num, BYTE* v1, ...);
```

Результаты экспериментов по оценке повышения быстродействия при использовании новых функций показаны на рис. 3. На представленном графике ось ординат определяет коэффициент, выражающий отношение времен выполнения новых и старых функций.



	10	100	500	1000	2000	5000
Квадратная	47,3	20,7	4,4	2,7	1,9	1,3
100 строк	70,0	20,7	4,4	2,9	1,9	1,3
1000 строк	46,7	18,4	4,4	2,7	1,9	1,3

Рис. 3. Повышение быстродействия программы при переходе к функции SetRowDiz

2. Реализация операций «взвешивания» и поиска единичных разрядов

Среди множества операций, реализованных в классах *CBV* и *CBM*, особо выделяются специфические для обработки булевой информации операции «взвешивания» (подсчета числа единиц) булева вектора и поиска ближайшего единичного разряда вектора, начиная с заданной позиции.

В базовой версии классов для реализации операции «взвешивания» применялся алгоритмический подход, при котором результат вычисляется по специальному алгоритму, описанному в работе [7]. Для оценки числа единиц в одном байте памяти *E* он выглядит как

```
BYTE count = E;
count =(count & 0x55)+((count>>1) & 0x55);
count =(count & 0x33)+((count>>2) & 0x33);
return((count & 0x0f)+((count>>4) & 0x0f));
```

Альтернативой данному подходу может служить табличный подход, при котором в памяти представляется таблица констант *TabC* размером 256 Б, каждый элемент которой содержит число единичных разрядов в его адресе. В этом случае подсчет числа единиц в некоторой области памяти *Vect* размером *n* Б сводится к линейному циклу вида

```
int j, count = 0;
BYTE* pB;
pB = (BYTE*)Vect;
for (j=0; j < n; j++) count += TabC[pB[j]];
return count;
```

Экспериментально было установлено, что в общем случае табличный подход обеспечивает ускорение выполнения операции «взвешивания» на 18 %.

Проведенные исследования показали, что преобладающим вариантом использования операций поиска единиц *LeftOne* и *RightOne* является организация цикла перебора всех компонент булева вектора, например

```
for (int j = bv.LeftOne(); j>=0; j = bv.LeftOne(j)) {…}
```

В данной ситуации с точки зрения эффективности создаваемых программ оказалась полезной разработка специального вспомогательного класса-итератора. Такой класс в теории объектно-ориентированного программирования используется для перебора элементов некоторой коллекции (в нашем случае в роли коллекции выступает булев вектор), обладающих заданным свойством (единичным значением).

Новый класс *CiBV* обладает следующим набором методов:

```
int Begin(int First = -1);
int End(int Last = -1);
int Next();
int Prev();
```

Организация цикла, аналогичного приведенному в последнем примере, с использованием класса CiBV выглядит так:

```
CiBV it(bv); for (int j = it.Begin(); j>=0; j = it.Next()){…}
```

Оценки эффекта в виде коэффициента, представляющего отношение времен выполнения программы перебора единиц при использовании итератора и без него, показаны на рис. 4.

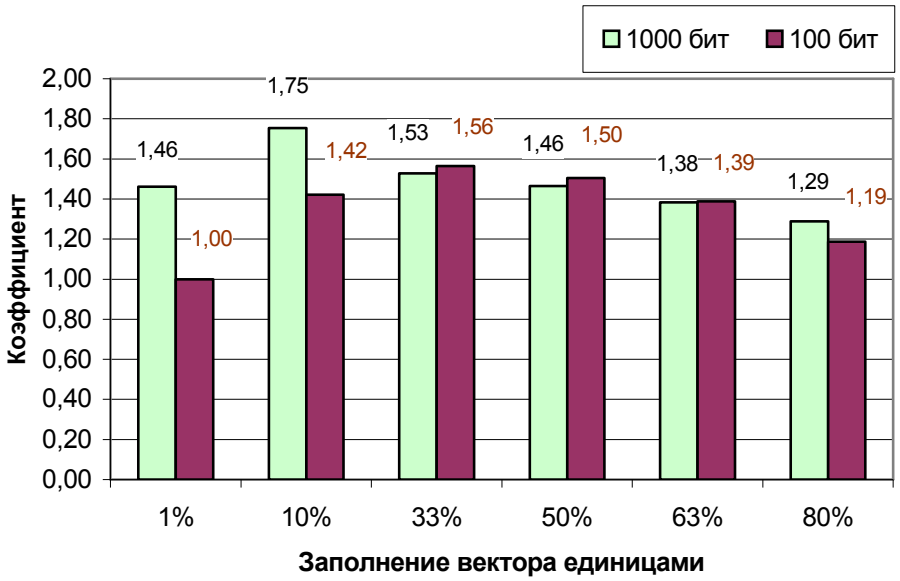


Рис. 4. Эффект от применения методов класса-итератора CiBV

3. Использование пословного представления информации

В разработанных ранее классах булевых объектов использовалось хранение бит в полях памяти, трактуемых как последовательности байтов при помощи описания *BYTE**. Такое представление обеспечивало наиболее экономичный расход оперативной памяти и простоту разработки требуемого инструментария. При этом способе хранения одна логическая операция, традиционная для рассматриваемой проблемной области, выполняется для восьми булевых значений одновременно. Вместе с тем в современных условиях есть возможность повысить степень параллелизма обработки, воспользовавшись пословной организацией памяти, при которой во

время выполнении команды одновременно обрабатываются 16, 32 или даже 64 бита. Однако фактический эффект может быть получен только в условиях полного согласования используемого технического, операционного и инструментального программного обеспечения.

Наиболее распространенные в настоящее время версии операционной системы Windows построены на платформе Win32. Это означает, что на уровне машинных команд обеспечивается только обслуживание 32-разрядных слов.

С целью увеличения параллелизма выполнения покомпонентных логических операций были разработаны новые классы *CuBV* и *CuBM* [8], которые обладают тем же составом методов, что и базовые классы *CBV* и *CBM*, но основаны на пословном представлении информации при помощи описания *ULONG**.

Проведенные эксперименты показали, что, к сожалению, использование этих классов пользы в общем случае не приносит – одни задачи решаются быстрее, а для других время выполнения только увеличивается. Причиной такого эффекта оказался факт незначительного (порядка 3 %), но стабильного, не зависящего от длины обрабатываемого вектора проигрыша операций доступа к отдельному биту информации при ее пословном, а не побайтовом представлении.

В таких условиях оказалась полезной разработка специального класса *ClBV*, который не определяет самостоятельный объект в оперативной памяти, а представляет в некотором роде *синоним* базового объекта класса *CBV*. В рамках класса *ClBV* память, занятая вектором, описывается через указатель последовательности слов ULONG*, что позволяет реализовать побитовые операции, например дизъюнкцию с 32-кратным параллелизмом. В состав методов этого класса включены логические операции и операции сравнения. Что касается программирования, то изменения в текстах программ при проведении оптимизации с использованием синонима минимальны. Так, например, вместо выполнения функции инвертирования, записываемой в виде

```
bv.Invert();
```

следует записать

```
ClBV blv(bv); blv.Invert();
```

Пусть требуется решить задачу: «В заданной булевой матрице инвертировать каждую вторую строку». На рис. 5 показано отношение времени решения этой задачи с использованием синонима и без него для различных размерностей исходных данных.

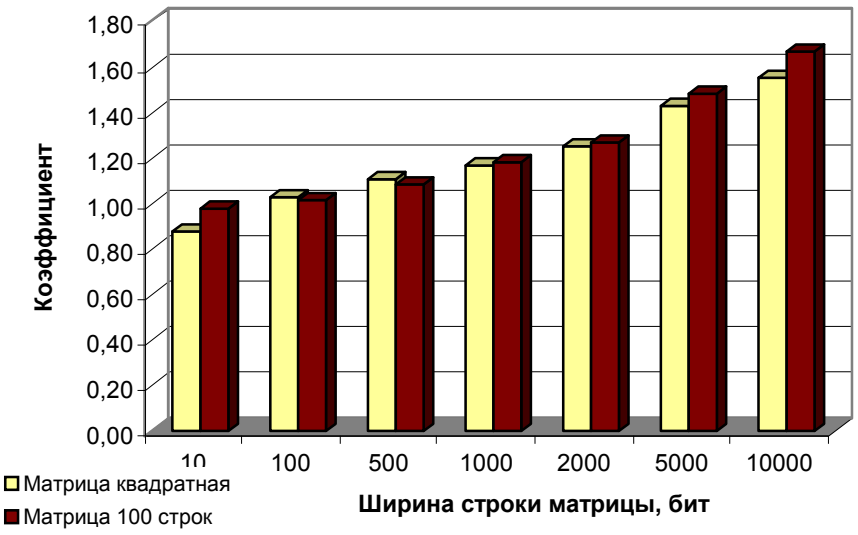


Рис. 5. Эффект от применения методов класса-синонима ClBV

Очевидно, что положительный эффект от применения синонима проявляется только при достаточно длинных векторах, имеющих 500 и более компонент.

4. Булевы объекты ограниченного размера

Рассматривая логико-комбинаторные задачи, обязательно следует учитывать высокую трудоемкость их решения, обусловленную переборным характером используемых алгоритмов. Фактически даже при небольших по размеру исходных данных время реализации алгоритма может быть достаточно велико и при использовании современной техники достигать десятков часов. В связи с этим предлагается применять новые классы *CsBV* и *CsBM* [8, 9], предназначенные для реализации тех же самых булевых объектов, но с наложенными на них ограничениями по размерности обрабатываемых данных: длина булева вектора и строки булевой матрицы не может превосходить 32 разрядов. Указанные ограничения порождаются изменением способа представления информации: вместо произвольной последовательности байтов, динамически запрашиваемой у операционной системы в процессе выполнения программы, для булева вектора и отдельной строки булевой матрицы используется длинное слово (тип данных «*unsigned long*»), определяемое в момент создания объекта.

В табл. 1 и 2 приводятся экспериментальные данные, показывающие повышение эффективности программ, разработанных с применением классов CsBV и CsBM, по сравнению с базовыми классами CBV и CBM соответственно. Рассматривая набор основных логических операций, можно заметить, что для булевых векторов получается 30-кратный выигрыш, а для булевых матриц в зависимости от характера операции выигрыш меняется от 1 (нет выигрыша) до 30.

Таблица 1

Сравнение времен выполнения групп операций над булевыми векторами

Набор выполняемых операций	Время выполнения, с		Коэффициент
	CsBV	CBV	
`CBV bv_1 = bv1 | bv2;`	0,4	13	32,5
`bv = bv1<<7;`	0,4	14,2	35,5
`bv = bv2>>25;`	0,4	13,6	34

`CBV bv_1 = bv1 | bv2;`
`CBV bv_2 = bv1 & bv2;`
`CBV bv_3 = bv1 ^ bv2;`
	1,2	40,5	34

`bv = bv1 ^ ~(bv2>>3);`	1,2	41,3	34,5

Таблица 2

Сравнение времен выполнения групп операций над булевыми матрицами

Набор выполняемых операций	Число строк матрицы	Время выполнения, с		Коэффициент
		CsBM	CBM	
`bm1.GetRowBv(j)&bm2.GetRow(k);`	1000000	0,75	27,95	37,2
	31	0,22	8,02	36,4
`bm.Disjunction()`	1000000	0,98	1,52	1,55
	31	1,03	4,3	4,2
`Bm.ExchangeRow (j, j+1);`	1000000	0,15	0,15	1
	31	0,45	0,45	1
`Bm.Add(TRUE);`	40000	1,5	14,1	9,4
	60000	3,67	31,39	8,7
`bm1.SetRow(j, bm.GetRowBv(i)-` `bm.GetRow(k));`	31	1,8	11,05	6,1

В пределах 32-разрядного ограничения по длине в новых классах динамизм поведения объектов поддерживается точно так же, как и для их предков – классов *CBV* и *CBM*. Идентичность старых и новых классов по составу реализуемых методов позволяет проводить отладку разрабатываемых программ только для одного из них. По завершении отладки достаточно просто заменить способ определенных в ней булевых объектов с одного на другой.

Предлагаемый инструментарий особенно хорошо подходит для алгоритмов комбинаторного поиска. В этих алгоритмах обычно используется дерево поиска решений, причем объем обрабатываемых данных редуцируется по мере удаления рассматриваемой вершины от корня дерева. Тогда программу, реализующую поиск, выгодно организовать таким образом, что сначала используются обычные булевы объекты, а при переходе к вершине нижнего уровня по

возможности осуществляется переход к представлению данных с помощью новых классов коротких векторов. В итоге вся обработка наиболее многочисленных листовых вершин дерева будет выполняться значительно быстрее и, тем самым, будет существенно повышаться общее быстродействие программы поиска в целом.

5. Класс «редких» булевых векторов

Практика логического синтеза дискретных устройств показала, что использование классов *CBV* и *CBM* для работы с реальными объектами, имеющими размерности десятки и сотни тысяч разрядов, фактически неприменимо из-за низкой эффективности получаемых программ. Одним из возможных решений проблемы высокой размерности может быть изменение формы представления исходных объектов. Дело в том, что на практике очень часто приходится иметь дело с «редкими» булевыми векторами, для которых характерно существенное количественное преобладание одних булевых компонент над другими.

Формально, рассматривая булев вектор как универсум $U = M^1 \cup M^0$, «редкий» вектор может быть определен на основании соблюдения условия

$$\left|M^1\right| >> \left|M^0\right| \text{ или } \left|M^1\right| << \left|M^0\right|. \quad (1)$$

Для работы с булевыми векторами, удовлетворяющими условию (1), выполнена разработка отдельного класса данных *CrBV*, описываемого в настоящей работе.

Прежде всего рассмотрим пример. Пусть имеется булев вектор «001000100001». Ясно, что этот же вектор может быть однозначно определен перечислением позиций, занятых единичными значениями (2, 6, 11) или занятых нулевыми значениями (0, 1, 3, 4, 5, 7, 8, 9, 10). Такое представление булева вектора будем называть *списковым*. Для однозначности предположим упорядоченность значений из списка по возрастанию. Если список содержит позиции единичных компонент, будем говорить о прямом списковом представлении, в противном случае – об обратном. Заметим, что прямое и обратное представления, по сути, задают подмножества M^1 и M^0 соответственно.

Таким образом, булев вектор может быть однозначно представлен тройкой

$$\boldsymbol{b} = (s, f, l), \quad (2)$$

где s – упорядоченный список позиций; f – булево значение, определяющее форму представления (*TRUE* – прямое, *FALSE* – обратное); l – количество разрядов (длина) булева вектора.

Именно такое представление булева вектора используется для описания объекта в создаваемом классе «редких» булевых векторов. Следует отметить, что применение типа данных *int* для определения длины вектора и номера позиции из списка обеспечивает представимость булевых векторов длиной до 2 147 483 647 разрядов.

Набор методов класса *CrBV* в целом аналогичен набору методов базового класса *CBV* за исключением специальных операций, связанных со спецификой представления описываемого объекта. Состав методов класса *CrBV* приведен в работе [10].

Логические операции играют ключевую роль во всех булевых вычислениях, поэтому их эффективность в существенной степени определяет полезность создаваемого класса объектов. Рассматривая далее реализацию операций, будем через (s, f, l) обозначать атрибуты результирующего вектора, а через (s_1, f_1, l_1) и (s_2, f_2, l_2) – атрибуты первого и второго операндов соответственно. Для всех логических операций предполагается одинаковая размерность участвующих в них операндов, поэтому $l = l_1 = l_2$.

Отметим, что унарная операция инверсии реализуется в рамках класса *CrBV* элементарно и сводится к инвертированию значения описателя формы представления списка: $s = s_1$, $f = \tilde{f}_1$. Что касается остальных логических операций класса (дизъюнкции, конъюнкции, сложения по модулю 2, разности), то их реализация основывается на выполнении известных операций над множествами (табл. 3).

В соответствии с данными из табл. 3 в классе *CrBV* были определены защищенные методы, реализующие операции объединения (*SetU*), пересечения (*SetI*), разности (*SetD*) и суммы

(*SetA*). В качестве операндов этих методов выступают упорядоченные списки позиций. Алгоритмы указанных методов построены на линейном переборе этих списков и обладают сложностью $O(r) = C^{r}(|s_1| + |s_2|)$, где C^{r} – некоторая константа, определяющая затраты на реализацию отдельного шага алгоритма; $|s_i|$ – длина i-го списка.

Вместе с тем алгоритмы реализации логических операций над булевыми векторами, представленными в традиционном битовом представлении (объекты класса *CBV*), могут быть оценены как $O(b) = C^{b}(l)$, где l – длина булева вектора. Тогда выражение

$$C^{r}(|s_1| + |s_2|) < C^{b}(l) \qquad (3)$$

является определяющим условием применения при программировании класса «редких» булевых векторов. Проведенные программные эксперименты показали, что преимущества класса «редких» векторов становятся безусловными при заполненности вектора не более чем на 0,6 %. Выигрыш по эффективности становится наиболее существенным с ростом размерности обрабатываемых данных и количеством редких значений (рис. 6).

Таблица 3

Реализация логических операций в зависимости от формата представления операндов

Операция		Формат представления операндов			
		f_1=TRUE, f_2=TRUE	f_1=TRUE, f_2=FALSE	f_1= FALSE, f_2= TRUE	f_1= FALSE, f_2=FALSE
Дизъюнкция	s=	$s_1 \cup s_2$	$s_2 \setminus s_1$	$s_1 \setminus s_2$	$s_1 \cap s_2$
	f=	TRUE	FALSE	FALSE	FALSE
Конъюнкция	s=	$s_1 \cap s_2$	$s_1 \setminus s_2$	$s_2 \setminus s_1$	$s_1 \cup s_2$
	f=	TRUE	TRUE	TRUE	FALSE
Сложение по модулю 2	s=	$s_1 + s_2$	$s_1 + s_2$	$s_1 + s_2$	$s_1 + s_2$
	f=	TRUE	FALSE	FALSE	TRUE
Разность	s=	$s_1 \setminus s_2$	$s_1 \cap s_2$	$s_1 \cup s_2$	$S_2 \setminus s_1$
	f=	TRUE	TRUE	FALSE	TRUE

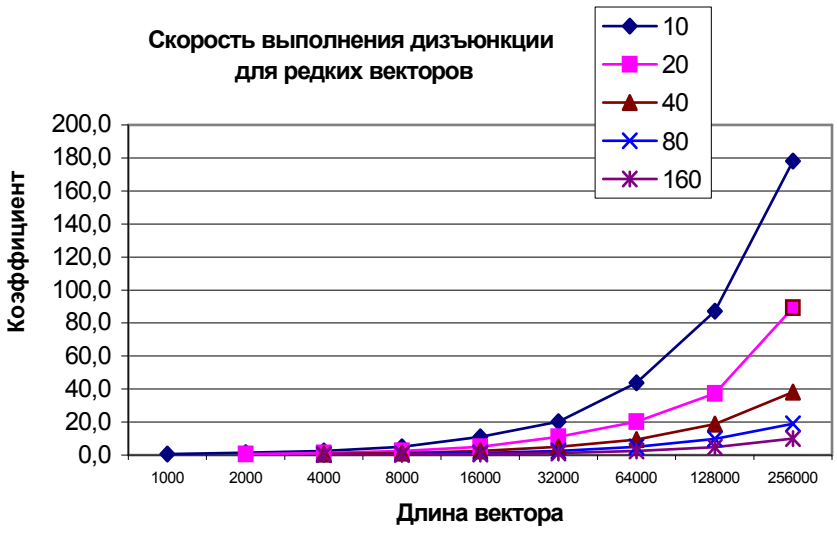


Рис. 6. Эффект от применения методов класса CrBV

Заключение

Предложенные программные средства дают возможность повысить эффективность компьютерных вычислений при решении трудоемких логико-комбинаторных задач. Ввиду особой важности вопроса повышения скорости решения таких задач отдельно рассмотрены различные области исходных данных и для каждой из них предложен свой способ оптимизации: при работе с короткими по длине векторами – использование классов *CsBV* и *CsBM*; при работе с очень

длинными векторами – класса *CrBV*; в остальных случаях – новых классов *CiBV*, *ClBV*, *ClBM* и модифицированных классов *CBV* и *CBM*.

Исследования проведены при частичной поддержке Международного научно-технического центра (проект В-986) и БРФФИ (проект Т05-258).

Список литературы

1. Закревский А.Д. Комбинаторные задачи над логическими матрицами в логическом проектировании и искусственном интеллекте // Успехи современной радиоэлектроники. – 1998. – № 2. – С. 59–67.

2. Романов В.И. Разработка инструментальных средств логического проектирования // Логическое проектирование. Вып. 6. – Мн.: Ин-т техн. кибернетики НАН Беларуси, 2001. – С. 151–170.

3. Василькова И.В., Романов В.И. Булевы векторы и матрицы в С++ // Логическое проектирование. Вып. 2. – Мн.: Ин-т техн. кибернетики НАН Беларуси, 1997. – С. 150–158.

4. Черемисинов Д.И., Черемисинова Л.Д. Троичные векторы и матрицы в С++ // Логическое проектирование. Вып. 3. – Мн.: Ин-т техн. кибернетики НАН Беларуси, 1998. – С.146–156.

5. Закревский А.Д., Василькова И.В. Криптоанализ машины Hagelin – метод решения системы логических уравнений // Комплексная защита информации. Вып 2. – Мн.: Ин-т техн. кибернетики НАН Беларуси, 1999. – С. 129–138.

6. Романов В.И. Перезагрузка операторов в С++ и эффективность программ // Актуальные проблемы радиоэлектроники: научные исследования, подготовка кадров: сб. науч. статей. В 3 ч. Ч. 2. – Мн.: МГВРК, 2005. – С. 163–168.

7. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. – М.: Мир, 1980. – 476 с.

8. Romanov V. Tools for programming boolean calculations // ECCO XVIII «Combinatorics for modern manufacturing, logistics and supply chains»: Abstracts of the XVIII European Conference 26–28 May, Minsk, Belarus. – Minsk: UIIP of the NASB, 2005. – С. 57–58.

9. Романов В.И. Оптимизация булевых вычислений на программном уровне // Танаевские чтения: докл. Второй науч. конф. 28 марта 2005 г., Минск. – Мн.: ОИПИ НАН Беларуси, 2005. – С. 91–93.

10. Романов В.И. Списковое представление булевых векторов // Известия Белорусской инженерной академии. – № 1(19)/2. – 2005. – С. 72–75.

Поступила 30.06.05

Объединенный институт проблем
информатики НАН Беларуси,
Минск, Сурганова, 6
e-mail: rom@newman.bas-net.by

V.I. Romanov

TOOLKIT FOR SOLVING LOGICAL COMBINATORIAL TASKS

A description of the toolkit for programming logical combinatorial algorithms based on the representation of the information by Boolean vectors and matrixes is offered. Such tools are represented by classes in the programming language C++. The efficiency of application of these classes is discussed.