ПРОГРАММИРОВАНИЕ, 2023, № 4, с. 21–26

КОМПЬЮТЕРНАЯ АЛГЕБРА

УДК 004.451.3,519.683.4,519.686.2,004.94

ПРИМЕНЕНИЕ В GINV ДИНАМИЧЕСКОГО ПЕРЕРАСПРЕДЕЛЕНИЯ ПАМЯТИ

© 2023 г. Ю. А. Блинков[a,b,*], Е. Ю. Щетинин[c,]**

[a]*Саратовский национальный исследовательский государственный университет имени Н.Г. Чернышевского*
410012 Саратов, ул. Астраханская, д. 83, Россия
[b]*Российский университет дружбы народов (РУДН)*
117198 Москва, ул. Миклухо-Маклая, д. 6, Россия
[c]*Финансовый университет при Правительстве Российской Федерации*
125167 Москва, Ленинградский пр-т, д. 49, Россия
**E-mail: blinkovua@info.sgu.ru*
***E-mail: riviera-molto@mail.ru*

Поступила в редакцию 01.07.2022 г.
После доработки 29.07.2022 г.
Принята к публикации 21.10.2022 г.

Представлена новая версия GInv (Gröbner Involutive) по вычислению инволютивных базисов Грёбнера в виде библиотеки на языке *C*++11. В GInv для динамических структур данных, таких как списки, красно-черные и бинарные деревья, библиотеки GMP для вычислений с целыми числами с произвольной точностью использовано объектно-ориентированное перераспределение памяти. Интерфейс пакета оформлен в виде модуля языка Python3.

DOI: 10.31857/S0132347423020061, **EDN:** GZFUDG

1. ИНВОЛЮТИВНЫЕ БАЗИСЫ ГРЁБНЕРА

Понятие базиса Грёбнера было введено Бруно Бухбергером в 1965 году [1]. Основой созданного им алгоритма является понятие *S-полинома*. Альтернативный подход к созданию конструктивных алгоритмов с помощью разбиения переменных на *мульти* и *немультикативные* переменные развит в работах [2, 3]. инволютивный подход пришел в компьютерную алгебру из теории линейных уравнений в частных производных, где метод приведения системы в инволюцию использовался уже в начале и середине XX века (работы Жане, Рикье, Томаса, Поммаре).

Во всех версиях алгоритмов построения базисов Грёбнера приходится иметь дело со структурами данных, которые меняются в ходе вычислений. Например, для представления полиномов могут использоваться списки, красно-черные деревья или строка в разряженной матрице [4].

Для вычислений с целыми числами с произвольной точностью также используют списки или массивы переменой длины состоящие из машинных слов, которые играют роль *цифр*, если рассматривать аналогию с обычными десятичными цифрами.

В настоящее время все современные системы компьютерной алгебры имеют встроенные пакеты построения базисов Грёбнера. Поскольку алгоритмы его построения имеют экспоненциальную сложность как по времени, так и требуемой памяти ее фрагментация является серьезной проблемой.

При работе пакета *INVBASE* (`http://www.reduce-algebra.com/manual/manualse151.html`), который инволюционными методами строит базис Грёбнера [2] для систем полиномиальных уравнений, в системе компьютерной алгебры *Reduce* было замечено, что в некоторых задачах процент работы сборки мусора составлял 8–10% от общего времени программы.

В предыдущей версии пакета *GInv* для построения инволютивных базисов Грёбнера [5] написанном на *C*++ при задачах, считающихся длительное время, обнаруживались значительные временные затраты на выделение/освобождение памяти.

Одним из возможных способов борьбы с этой проблемой является перезапуск программы. Другими словами программа в какой-то момент времени, выбор которого очень не тривиальная задача, сохраняет данные на диск в простом виде или используя сериализацию. Затем при перезапуске происходит чтение сохраненных данных, что во первых устраняет фрагментацию, а во вторых *соседние* данные располагаются, как правило, на одной странице памяти, что оптимизируют их для

использования кэша процессора. В результате на некоторых задачах компьютерной алгебры ускорение может возрасти в разы и даже больше, несмотря на затраты для *записи/чтения*.

В связи с вышесказанным возникает необходимость в перераспределении памяти.

2. ПРОБЛЕМА ПЕРЕРАСПРЕДЕЛЕНИЯ ПАМЯТИ

В современных языках программирования все переменные можно разделить на три категории по времени их *жизни* в программе:

• все время *жизни*, так называемая *статическая память*, а переменные, использующие эту память, часто называют *статическими*. Сюда также попадают и *глобальные*;

• время выполнения функции, память выделяется в стеке, а переменные называют *автоматическими*, но в языках программирования используют слово *локальные*;

• время *жизни* определяет программист, если нет *сборки мусора* (garbage collection), а память выделяется в *куче*. Или определяет алгоритм *сборки мусора*, если она есть в выбранном языке программирования. Иногда поддерживаются оба варианта работы.

Сборка мусора была впервые применена Джоном Маккарти в 1959 году в среде программирования на разработанном им функциональном языке программирования *Лисп* (Lisp), а сам термин возник в сноске статьи [6].

Если память в *куче* выделяется разного размера, то несмотря на возможные алгоритмы слияния соседних свободных фрагментов, возникают области памяти, которые не используются. Джоном Маккарти в языке программирования *Лисп* было предложено очень красивое решение. Все лисповские ячейки одного размера, а списками (точнее *списками списков*) можно представить любые структуры данных.

В настоящее время существует несколько базовых алгоритмов *сборки мусора* и их огромное число модификаций. Например, для языка *Java* разработчики обычно поставляют сразу несколько вариантов алгоритмов *сборки мусора* для разных вариантов использования программ написанных на *Java*.

Можно выделить основные критические моменты при использовании *сборки мусора*:

• *сборка мусора* потребляет вычислительные ресурсы для принятия решения о том, какую память освободить, даже если программист, возможно, уже знал эту информацию;

• момент, когда мусор действительно будет собран, может быть непредсказуемым, что приводит к остановкам (паузы для сдвига/освобождения памяти), разбросанных по всему сеансу. Инкрементные, параллельные *сборщики мусора* и *сборщики мусора* в реальном времени решают эти проблемы с различными компромиссами;

• самое главное: *сборка мусора* не учитывает *защищенный режим*, *страничную организацию памяти* и основу современных процессоров – *кэш* (сверхоперативную память).

Рассмотрим для сравнения реализации *malloc/free* для языко *C/C++*:

• В 2020 году компания Google представила новый вариант системы распределения памяти *TCMalloc*, которая используется во многих внутренних проектах Google. Для работы требуется наличие компилятора с поддержкой *C++17* для языка *C++*, и *C11* для языка *C* (gcc 9.2+ или clang 9.0+). Из операционных систем поддерживается только Linux (x86, PPC).

Также, с 2005 года существует ещё один вариант *tcmalloc*, который поставлялся в составе пакета *gperftools* (Google Performance Tools).

• В 2021 году выпущен *mimalloc* бесплатный компактный менеджер памяти общего назначения с открытым исходным кодом, разработанный Microsoft с акцентом на характеристики производительности. Библиотека составляет около 11000 строк кода и работает как замена *malloc* стандартной библиотеки *C* и не требует дополнительных изменений кода.

• *jemalloc* реализация *malloc* общего назначения, для предотвращения фрагментации и поддержки масштабируемого параллелизма. *jemalloc* впервые начал использоваться в качестве распределителя libc FreeBSD в 2005 году.

Об остроте рассматриваемой здесь проблемы, также говорит множество докладов [7–11] по работе с динамическим перераспределением памяти на конференциях ISMM (`http://www.sigplan.org/Conferences/ISMM/`).

3. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПЕРЕРАСПРЕДЕЛЕНИЕ ПАМЯТИ

Перечислим основные концепции предлагаемого подхода

• Предлагаемый подход принципиально отличается от вышеизложенных и основан на реализации ООП (Объектно-ориентированное программирование) в *C++*.

• Ограничить использование указателей на внутреннюю структуру класса *C++* объявив их в закрытой области видимости. При правильной технологии ООП, а это свойство называется *инкапсуляцией*, это и так необходимо делать, пользователь не должен видеть как реализован класс внутри, он должен использовать только его интерфейс.

• В *C++* имеется выделенный *конструктор копирования*, который позволяет построить копию объекта. Осталось для выбранного класса определить функцию *swap*, для подмены объекта на его копию, и можно организовывать динамическое перераспределение памяти на *C++*.

В качестве примера использования рассмотрим простейший класс на версии языка *C++11* с динамическим выделением памяти – линейный однонаправленный список целых чисел.

```
1 #include “allocator.h”
2
3 class List {
4   struct Node {
5     int  mData;
6     Node* mNext;
7
8     Node(int data, Node* next=nullptr):
9        mData(data),
10       mNext(next) {
11    }
12    ~Node() {};
13  };
14
15  Allocator* mAllocator;
16  Node* mHead;
17
```

Основное отличие от обычного кода строчка 15, где содержится объявление переменной указателя на тип *Allocator*. В строках 23 и 27 показано его использование для выделения и освобождения памяти. Освобождение использует шаблонную функцию *destroy*, поскольку перегрузку стандартого *delete* дополнительными аргументами позволяют не все компиляторы *C++*. Для базовых типов и классов, для которых в принципе не нужен вызов деструктора, определена шаблонная функция *dealloc*. *destroy* и *dealloc* могут также дополнительно вызываться с дополнительным целочисленным аргументом для удаления массивов.

```
18 public:
19   List()=delete;
20   List(Allocator *allocator):
21   …
22   List(const List& a)=delete;
23   List(const List& a, Allocator *allocator):
24   …
25     *mLink = new(mAllocator) Node(i,
26                                   *mLink);
27   …
28     auto* tmp=*mLink;
29     *mLink = tmp->mNext;
30     mAllocator->destroy(tmp);
31   …
32   void swap(List &a) {
33     auto *tmp1=mAllocator;
34     mAllocator = a.mAllocator;
35     a.mAllocator = tmp1;
36
37     auto* tmp2=mHead;
38     mHead = a.mHead;
39     a.mHead = tmp2;
40   }
41   …
42 };
```

В строке 19 фактически объявлен запрет на использование конструктора по умолчанию, его роль теперь играет конструктор объявленный в строке 20.

Как и для *конструктора по умолчанию* в строке 22 объявлен запрет на использование стандартного *конструктора копирования*. В строчках 23 и 32 содержатся заголовки нового *конструктора копирования* и функции *swap*, которые необходимы для замены текущего объекта на его копию. Если Вы хотите для своего класса использовать оператор =, то его тоже необходимо реализовать. Он, как известно, не наследуется, но его поддержка в классе шаблона *GC* имеется. Ниже дан пример его использования.

```
43 typedef GC<List> GCList;
44 …
45 GCList a;
46 …
47 a.reallocate();
48 …
```

На строке 43 объявляем новый тип *GCList*. Затем вызываем на строке 45 конструктор по умолчанию. Предположим на строке 46 производим массовые операции по выделению/освобождению памяти. Память выделяется блоками, кратными размеру страницы памяти, затем раздается внутри конкретного объекта класса *List*. Если процент не используемой памяти высок, то вызов *reallocate* на строке 45 приведет к построению копии объекта, с его последующей заменой на первоначальную переменную a с использованием swap внутри тела функции *reallocate*.

Естественно, вызов *reallocate* может происходит неоднократно, согласуясь с логикой программы. Также, за счет показанных в следующем разделе очень простых решений попутно выполняются учет работы времени по перераспределению памяти, выделанная в данный момент времени память, максимальная память и в *debug*-версии программы проверки на утечки памяти. Выделение памяти большими блоками очень выгодно, особенно для большего количества мелких объек-

Таблица 1.

	T	GC	GC/T	M
cyclic7				
Allocator	93.48	3.83	4.1%	172
Allocator (кроме GMP)	80.89	0.28	0.35%	180
malloc/free	119.47	–	–	124
Задача Попова 21 [13]				
Allocator	505.37	2.52	0.50%	1231
Allocator (кроме GMP)	487.59	0.64	0.13%	559
malloc/free	495.81	–	–	609
eco11				
Allocator	871.75	2.31	0.27%	797
Allocator (кроме GMP)	611.21	0.86	0.14%	243
malloc/free	924.41	–	–	186

тов. Стандартные *new*, *delete*, *malloc*, *free* должны хранить на каждое выделение информацию о начале и конце выделенной области памяти. В данном подходе этого не нужно, поскольку конкретный объект может с помощью конструктора-копирования построить свою копию и обратно вернуть используемую память в операционную систему. Замечу, что предлагаемый подход, позволяет собирать объект из *кирпичиков* с общим *Allocator*-ом. Лишь бы в каждом инкапсулированом объекте память выделялась/освобождалась с помощью указателя на общий *Allocator* и были определены соответствующие *конструкторы-копирования* и функция *swap*. Можно также, как показано ниже, определять *Allocator* вначале блока и затем его использовать для работы с автоматическими переменными, использующими *Allocator*.

```
1 {
2   Allocator a [1];
3   …
4   List lst(a);
5   …
6 }
```

После закрытия блока вся занятая ими память будет возвращена в систему. Это хорошо имитирует нестандартную функцию *alloca* из многих Unix подобных систем, которая выделяет память из стека с последующим автоматическим ее возвращением в стек при завершении блока с локальными переменными. При этом в отличии от *alloca* и фактически при тех же затратах по требуемой памяти и времени выполнения можно выделять для сложных объектов, а не только для встроенных типов данных и массивов.

Внутри класса *Allocator* используется набор статических элементов класса для ведения статистики: затраты машинного времени по перераспределению памяти, выделенная в данный момент времени память, максимальная память. Переменные класса хранят информацию по выделанной в данный момент времени и используемой памяти для конкретного объекта класса *Allocator*. При этом размер разовой выделяемой памяти выравнивается на границу кратную кэшу данного компьютера и выставляется вручную при компиляции библиотеки.

При возврате памяти в классе *Allocator* память физически не возвращается в систему, единственно уменьшается переменная используемой памяти. При вызове деструктора класса *Allocator* в *debug* версии программы происходит проверка, что счетчик используемой памяти равен 0.

Сама память представляет собой односвязный список из блоков памяти кратный странице памяти компьютера. Блоки для переносимости библиотеки выделяются стандартной функцией *malloc*. Возврат памяти в систему происходит при вызове деструктора с помощью последовательных вызовов функции *free*. Структура классов и их основные атрибуты и методы показаны на рисунке.

За счет директивы макросов языка *C++* при сборке можно собрать класс *Allocator*, который фактически только вызывает стандартные *malloc/free*. Это бывает удобно для сравнения и отладки.

Размер обоих реализаций составляет в сумме примерно 320 строчек кода со всеми комментариями и пробельными строками и расположен на GitHub по адресу `https://github.com/blinkovua/GInv/tree/master/util`.

4. ИСПОЛЬЗОВАНИЕ В GInv

Объектно-ориентированное перераспределение памяти в новой версии *GInv* (`https://github.com/blinkovua/GInv`) используется для объектов, которым приходится многократно перестраиваться в памяти. Рассмотрим стандартный пример представления полинома в виде упорядоченного списка мономов и их коэффициентов. Для сохранения порядка мономы с их коэффициентами часто меняются местами, при равенстве нулю коэффициента происходит их удаление, а деление полинома на его *содержание* приводит к изменению всех коэффициентов.

В настоящий момент в *GInv* все структуры данных, которые меняются в процессе построения инволютивного базиса Грёбнера имеют динамическое перераспределение памяти. В качестве наиболее характерного примера можно привести бинарные деревья специального вида – деревья Жане [12].

Библиотека *GMP* (`https://gmplib.org/`) для вычислений с целыми числами с произвольной точностью не позволяет на прямую воспользоваться классом *Allocator* пакета *GInv*. Для нее была сделана *обертка*, которая заранее вычисляет

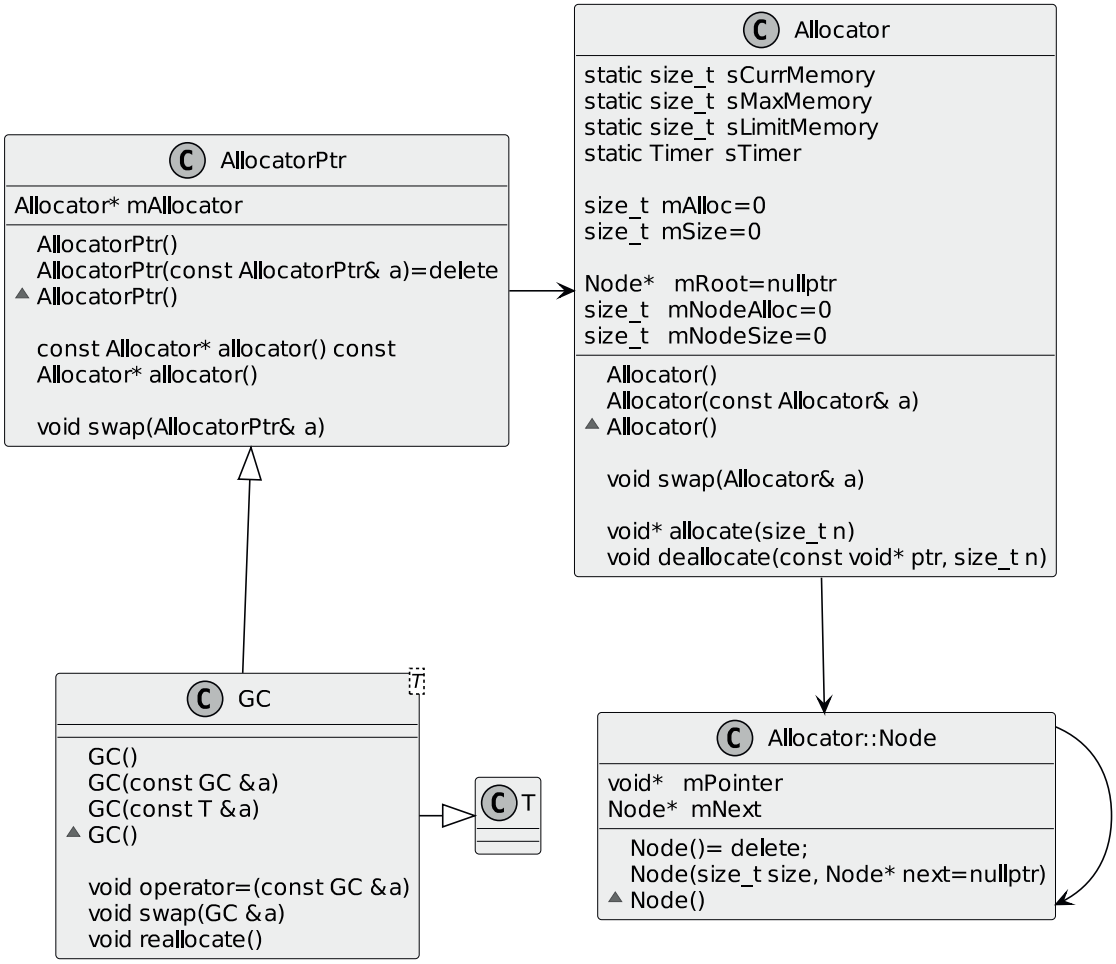

Рис. 1.

требуемый объем памяти и выделяет ее используя класс *Allocator* для проведения операций *сложения/вычитания*, *умножения/деления*, возведения в степень и вычисления наибольшего делителя.

Результаты представлены в таблице 1 при вычислении на процессоре *Intel(R) Xeon(R) CPU L5630 @ 2.13GHz, cache size: 12288 KB* в операционной системе Debian GNU/Linux 11. Столбцы таблицы: T – время в секундах, GC – затраченное время на динамическое перераспределение памяти и составляет часть T, GC/T – процент от общего времени. По середине таблицы условное название примера, M – максимальная память в мегабайтах (MiB), измеренная системной утилитой *time*.

5. ЗАКЛЮЧЕНИЕ

Как видно из результатов вычислений в некоторых случаях заметно значительное ускорение программ, кроме *Задача Попова 21* [13], что связанно с тем, что внутри этой задачи размеры чисел очень велики, а остальные структуры памяти практически не используются, а имеются лишь затраты на лишнее копирование чисел при их динамическом перераспределении памяти.

Предложенный подход практически лишен недостатков стандартных *malloc/free*, так и подхода с использованием *сборки мусора*. Хорошо использует страничную организацию памяти и кэш процессора. Очень прост в реализации по сравнению со стандартными *malloc/free* и позволяет эффективно находить ошибки, связанные с *утечками* памяти.

БЛАГОДАРНОСТИ

Работа выполнена при поддержке Программы стратегического академического лидерства РУДН.

СПИСОК ЛИТЕРАТУРЫ

1. *Buchberger B.* Gröbner bases: an Buchberger algorithmic method in polynomial ideal theory // Recent Trends in Multidimensional System Theory / Ed. by *N.K. Bose*. V. 6. Reidel, Dordrecht, 1985. P. 184–232.

2. *Жарков А.Ю., Блинков Ю.А.* Инволютивные системы алгебраических уравнений // Программирование. 1994. № 1. С. 53–56.
3. *Gerdt V.P., Blinkov Yu.A.* Minimal involutive bases // Mathematics and Computers in Simulation. 1998. V. 45. P. 543–560.
4. *Faugère J.-C.* A new efficient algorithm for computing Gröbner bases (F4) // Journal of Pure and Applied Algebra. V. 139 (1–3). 1999. P. 61–88.
5. *Блинков Ю.А., Гердт В.П.* Специализированная система компьютерной алгебры GINV // Программирование. 2008. № 2. С. 67–80.
6. *McCarthy J.* Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I // Commun. ACM, 1960. № 4. P. 184–195.
7. *Bansal A., Goel S., Shah P., Sanyal A., Kumar P.* Garbage Collection Using a Finite Liveness Domain // Proceedings of the 2020 ACM SIGPLAN ISMM, 2020. P. 1–15.
8. *Yang A.M., Österlund E., Wilhelmsson J., Nyblom H., Wrigstad T.* ThinGC: Complete Isolation with Marginal Overhead //Proceedings of the 2020 ACM SIGPLAN ISMM, 2020. P. 74–86.
9. *Onozawa H., Ugawa T., Iwasaki H.* Fusuma: Double-Ended Threaded Compaction //Proceedings of the 2021 ACM SIGPLAN ISMM, 2021. P. 94–106.
10. *Tripp C., Hyde D., Grossman-Ponemon B.* FRC: A High-Performance Concurrent Parallel Deferred Reference Counter for C++ // Proceedings of the 2018 ACM SIGPLAN ISMM, 2018. P. 14–28.
11. *Seyri A., Pan A., Vamanan B.* MemSweeper: Virtualizing Cluster Memory Management for High Memory Utilization and Isolation //Proceedings of the 2022 ACM SIGPLAN ISMM, 2022. P. 15–28.
12. *Гердт В.П., Янович Д.А., Блинков Ю.А.* Быстрый поиск делителя Жане // Программирование. 2001. № 1. С. 32–36.
13. *Попов А.С.* Кубатурные формулы на сфере, инвариантные относительно группы вращений икосаэдра // Сиб. журн. вычисл. математики / РАН. Сиб. отд-ние. Новосибирск, 2008. Т. 11. № 4. С. 433–440.