Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В.И. Шелехов

РАЗРАБОТКА И ВЕРИФИКАЦИЯ АЛГОРИТМОВ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ В ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Препринт
164

Новосибирск 2012

Описаны три алгоритма пирамидальный сортировки: классический, алгоритм Флойда и улучшенный алгоритм, причем последний является самым быстрым из существующих алгоритмов сортировки. Алгоритмы представлены на языке предикатного программирования P вместе со спецификациями в виде предусловий и постусловий. Дедуктивная верификация алгоритмов в системе автоматического доказательства PVS вместе с повторными релизами спецификаций потребовала около двух месяцев.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

V.I. Shelekhov

DEVELOPMENT AND VERIFICATION OF HEAPSORT ALGORITHMS IN PREDICATE PROGRAMMING PARADIGM

Preprint
164

Novosibirsk 2012

Three heapsort (classic, Floyd's, and improved) algorithms are described. The improved heapsort algorithm is the fastest sorting algorithm. The algorithms are described in the P predicate programming language with their specifications in the form of a precondition and postcondition. Deductive verification of the algorithms in the framework of the PVS prover and repeated development of the specifications were performed over two months.

1. ВВЕДЕНИЕ

Описаны три алгоритма пирамидальный сортировки: классический алгоритм Дж. Вильямса [15], алгоритм Флойда и улучшенный алгоритм [8], причем последний является самым быстрым из существующих алгоритмов сортировки. Алгоритмы представлены на языке предикатного программирования P [24], обеспечивающем адекватное и прозрачное описание логики различных решений задачи сортировки.

Эффективность предикатных программ достигается их оптимизирующей трансформацией на императивное расширение языка P с последующей конвертацией на C++, ФОРТРАН и другие языки. Базовыми трансформациями являются: склеивание переменных, реализующее замену нескольких переменных одной, замена хвостовой рекурсии циклом, подстановка тела программы на место ее вызова и кодирование структурных объектов через массивы и указатели.

Предикатная программа состоит из набора программ (определений предикатов) следующего вида:

<имя программы>(<описания аргументов>**:** <описания результатов>)
pre <предусловие>
{ <оператор> }
post <постусловие>

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они определяют спецификацию программы. Предусловие должно быть истинно перед исполнением программы, а постусловие – после ее завершения. Ниже представлены основные конструкции языка P: оператор присваивания, блок, параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> **else** <оператор2>
<имя программы>(<список аргументов>**:** <список результатов>)
<тип> <пробел> <список имен переменных>

Операторы, образующие параллельный оператор, информационно независимы; иначе используется блок. Фрагмент вида T y; F(... : y) можно записать короче: F(... : T y), где T – тип, y – локальная переменная.

Проведена дедуктивная верификация описанных алгоритмов пирамидальной сортировки с использованием системы автоматического доказательства PVS. Нетривиальной проблемой оказалось построение достаточно точных спецификаций – при неточных спецификациях условия корректности программы становятся недоказуемыми. Условия корректности, теории с дополнительными леммами и все доказательства доступны в формате системы PVS: http://www.iis.nsk.su/persons/vshel/files/heapsort.zip.

Спецификация сортировки массива элементов произвольного типа с произвольным отношением порядка между элементами сформулирована в разделе 2. Понятие двоичной кучи, используемой в пирамидальной сортировке, дано в разделе 3. В разделах 4–6 описываются программы пирамидальной сортировки вместе с их спецификациями. Обзор работ представлен в разделе 7. Верификация программ сортировки описана в разделе 8.

2. СПЕЦИФИКАЦИЯ ЗАДАЧИ СОРТИРОВКИ В ОБЩЕМ ВИДЕ

Исходными данными задачи сортировки является некоторый сортируемый массив a. Элементы массива имеют произвольный тип T с линейным (тотальным) порядком "≤". Предположим, массив a состоит из n элементов, причем $n \neq 0$. В алгоритмах пирамидальной сортировки массив a удобнее нумеровать с 1. Данные определения представим в виде последовательности описаний на языке P.

```
type T;
type nat1 = subtype(nat x: x > 0);
nat1 n;
type nat1n = 1 .. n;
type Arn = array (T, nat1n);
```

Здесь тип Arn есть тип массива a с элементами типа T и индексами от 1 до n. Тип T и число элементов n являются параметрами задачи.

Спецификация задачи сортировки определяется следующим описанием:

```
sort(Arn a: Arn a') post sorted(a') & perm(a, a');
```

Штрих в имени a′ предполагает склеивание переменных a и a′ в реализации; это означает, что результат сортировки должен быть получен в мас-

сиве a. Спецификация постулирует, что массив a′ должен быть сортирован (предикат sorted) и получен перестановкой исходного массива a (предикат perm).

```
formula sorted(Arn a) =  ∀ nat1 i, j. i < j ⇒  a[i] ≤ a[j];
```

Два массива являются *перестановочными*, если второй массив можно получить из первого некоторой последовательностью перестановок пар элементов. Однако для доказательства удобнее классическое определение: существует биективное отображение второго массива в первый.

```
type Func1n = predicate(nat1n : nat1n);   // тип функции из nat1n в nat1n.
formula injective(Func1n f ) = ∀ nat1n x, z. f(x) = f(z) ⇒ x = z;
formula surjective(Func1n f ) = ∀ nat1n y. ∃ nat1n x. f(x) = y;
formula bijective(Func1n f ) = injective(f) & surjective(f);
formula perm(Arn a, b) = ∃ Func1n f. bijective(f) & ∀ nat1 j. a[j] = b[f(j)];
```

3. ДВОИЧНАЯ КУЧА

Существует простой, эффективный и компактный способ представления двоичного дерева внутри массива a. Вершинами дерева являются элементы a[1], a[2], …, a[m], где m ≤ n. На рис.1 дается пример представления дерева для m = 8.

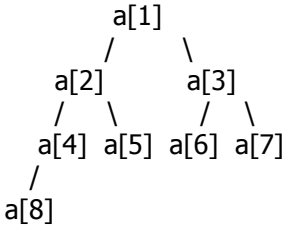

Рис.1. Представление двоичного дерева внутри массива

Родитель вершины с индексом j имеет индекс j **div** 2, а левая и правая дочерние вершины имеют индексы 2 j и 2 j + 1, соответственно. Определения для правого и левого потомка и родителя вершины a[j] даются ниже.

```
formula left(nat1 j : nat) = 2 * j;
formula right(nat1 j : nat) = 2 * j + 1;
type nat2 = subtype(nat x : x > 1);
formula father(nat2 j : nat) =  j div 2;
                  // div – операция деления нацело; j div 2 = floor(j / 2)
```

Двоичная максимальная куча (или пирамида) есть двоичное дерево, представленное внутри массива, в котором значение каждой вершины не меньше значений ее потомков. Формальное определение кучи дано ниже.

```
formula heap(Arn a, nat1n m) =
    ∀ nat j = 1 .. m. (left(j) ≤ m ⇒ a[j] ≥ a[left(j)]) &
                      (right(j) ≤ m ⇒ a[j] ≥ a[right(j)]) .
```

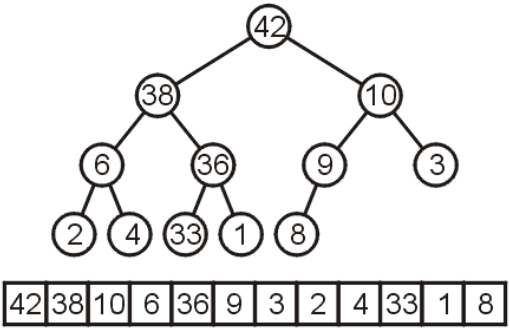

Рис.2. Пример двоичной максимальной кучи при m = 12

4. КЛАССИЧЕСКИЙ АЛГОРИТМ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ

Пирамидальная сортировка реализуется следующей программой:

```
heapsort(Arn a: Arn a')
{  buildHeap(a: Arn b);
   sort(b: a')
} post sorted(a') & perm(a, a');
```

Программа buildHeap строит по массиву a двоичную кучу b, а sort – реализует собственно сортировку.

Программа buildHeap строит кучу с конца массива a. Куча, построенная на части массива a для элементов с индексами от k до m, называется *обобщенной* и определяется формулой:

```
formula heap(Arn a, nat k, nat1n m) =
    ∀ nat j = k .. m. (left(j) ≤ m ⇒ a[j] ≥ a[left(j)]) &
                      (right(j) ≤ m ⇒ a[j] ≥ a[right(j)]) .
```

В *обобщенной двоичной куче* допускается более одной корневой вершины.

На очередном шаге работы программы buildHeap строится обобщенная куча для элементов от k до n. Поскольку на предыдущем шаге обобщенная куча построена для элементов от k+1 до n, остается переместить элемент a[k] в дереве с вершиной a[k] таким образом, чтобы получить кучу от k до n.

В императивном программировании описанная схема реализуется в виде цикла. В предикатном программировании строится рекурсивная программа, однако перед этим исходная задача buildHeap(a: b) *сводится к более общей задаче* buildHeapG(a, k: b), которая строит кучу для всего массива a в предположении, что уже построена обобщенная куча от k+1 до n. Поскольку дерево из одной вершины a[n] является обобщенной кучей, то истинно heap(a, n, n). Это позволяет использовать k = n - 1 для сведения исходной задачи. Однако возможно лучшее решение. При k = father(n) = n **div** 2 массив от k+1 до n является обобщенной кучей, состоящей лишь из корневых вершин. Сведение реализуется в виде следующей программы:

```
buildHeap(Arn a: Arn b)
{       buildHeapG(a, n div 2: Arn b)
} post perm(a, b) & heap(b, 1, n);
```

Программа buildHeapG представлена следующим рекурсивным определением:

```
buildHeapG(Arn a, nat k: Arn b)
    pre heap(a, k + 1, n)
{     if (k = 0) b = a
      else { siftDown(a, k, n: Arn c);
             buildHeapG(c, k - 1: b)
      }
} post perm(a, b) & heap(b, 1, n) measure k;
```

Программа siftDown строит обобщенную кучу для элементов от k до n. Поскольку обобщенная куча построена от k+1 до n, достаточно переместить a[k] на свое место вниз по одному из путей в дереве с корневой вершиной a[k]. Эта операция перемещения элемента a[k] называется «просеивание вниз».

Мера, заданная конструкцией **measure** k, используется при верификации для доказательства того, что аргументы рекурсивного вызова по мере строго меньше аргументов программы – это гарантирует завершение программы.

Вместо оператора присваивания вида a[j] := x в языке P используется конструкция: a **with** [j: x]. Используемая далее операция перестановки элементов с индексами j и k в массиве a представляется определением функции:

```
function swap(Arn a, nat1n  j, k: Arn) = a with [k: a[j], j: a[k]];
```

В предположении, что обобщенная куча построена от k+1 до m ($m \le n$), программа siftDown строит обобщенную кучу от k до m. Элемент a[k] сравнивается с наибольшим из потомков и обменивается с ним операцией swap, если потомок оказался большим. После обмена элементов применяется рекурсивный вызов siftDown для продолжения просеивания вниз исходного элемента a[k], находящегося в позиции потомка.

```
siftDown(Arn a, nat1 k, nat1n  m: Arn c)
    pre heap(a, k + 1, m)
{     if (left(k) > m)  c = a
      else   if (a[left(k)] ≤ a[k])
                        if (right(k) > m or a[right(k)] ≤ a[k])   c = a
                        else    siftDown(swap(a, k, right(k)), right(k), m: c)
             else   if (right(k) > m or a[left(k)] > a[right(k)])
                                 siftDown(swap(a, k, left(k)), left(k), m: c)
                    else   siftDown(swap(a, k, right(k)), right(k), m: c)
} post perm(a, c, m) & heap(c, k, m) measure (k>=n)? 0 : n - k;
```

Эффективность программы можно улучшить введением локальной переменной j = 2*k с заменой всех вхождений left(k) на j, а right(k) – на j+1.

Предикат perm(a, c, m) в постусловии ограничивает перестановочность массивов a и c первыми m элементами; элементы от m+1 до n должны совпадать:

```
formula perm(Arn a, b, nat1n m) =
    ∃ Func1n f. bijective(f) & ∀ nat1 j. a[j] = b[f(j)] & (j > m ⇒ j = f(j));
```

Данное уточнение оказалось необходимым для верификации.

Пирамидальная сортировка относится к классу алгоритмов *сортировки выбором* и реализуется по следующей схеме. Массив a разделен на две части. Правая часть состоит из элементов a[m+1], a[m+2], …, a[n] и уже отсортирована. Левая часть является двоичной кучей. Любой ее элемент не больше любого из элементов правой части. На первом шаге m = n. На очередном шаге сортировки элемент a[1], являющийся максимальным в левой части, обменивается местами с a[m]. Для восстановления кучи в левой час-

ти обновленный элемент a[1] просеивается вниз программой siftDown. При m = 1 массив a оказывается отсортированным.

```
sort(Arn a: Arn a')
pre heap(a, 1, n)
{       sortG(a, n : a')
}  post sorted(a') & perm(a, a');
```

Программа sort сводится к более общей программе sortG(a, m: a') при m = n. Программа sortG реализует сортировку массива a в предположении, что правая часть является двоичной кучей, а левая – отсортирована, причем любой элемент правой части не меньше любого из элементов левой части.

```
formula sorted(Arn a, nat1 m) =  ∀ i, j= m..n. i < j ⇒  a[i] ≤ a[j];
formula twoPart(Arn a, nat1 m) = m ≤ n ⇒ a[1] ≤ a[m];
sortG(Arn a, nat1n m: Arn a')
  pre heap(a, 1, m) & sorted(a, m + 1) & twoPart(a, m + 1)
{       if (m = 1) a' = a
        else { siftDown(swap(a, 1, m), 1, m - 1: Arn c);
               sortG(c, m - 1 : a')
        }
} post sorted(a') & perm(a, a') measure m;
```

Предикат twoPart в предусловии гарантирует, что элементы правой части не меньше элементов левой.

Приведенный набор определений на языке P составляет полную предикатную программу пирамидальной сортировки. К программе применяется следующий набор оптимизирующих трансформаций: склеивание переменных (a', b, c склеиваются с a), замена хвостовой рекурсии циклом, подстановка тел программ на место их вызовов, оформление циклов и упрощения. Итоговая программа на императивном расширении языка P приведена ниже.

```
swap(Arn a, nat1n  k, j) { T x = a[j]; a[j] = a[k]; a[k] = x };
siftDown(Arn a, nat1 k, nat1n  m)
{ for(; ; ){ nat j = k * 2;
             if (j > m) break;
             if (a[j] ≤ a[k])
               if (j + 1 ≤ m & a[j + 1] > a[k]) { swap(a, k, j + 1);  k = j + 1}
               else break
             else  if (j + 1 > m or a[j] > a[j + 1]) { swap(a, k, j);  k = j }
                   else  { swap(a, k, j + 1);  k = j + 1 }
}}
```

```
heapsort(Arn a)
{   for(nat k = n div 2; k ≠ 0 ; k = k – 1)  siftDown(a, k, n);
    for(nat1n  m = n; m ≠ 1; m = m – 1)
                { swap(a, 1, m); siftDown(a, 1, m - 1) }
}
```

5. АЛГОРИТМ ФЛОЙДА

Программа sort имеет сложность O($n \log n$), а buildHeap – O(n), при этом время исполнения buildHeap сравнительно мало. Поэтому эффективность пирамидальной сортировки зависит главным образом от алгоритма восстановления двоичной кучи после обмена элементов a[1] и a[m]. Р. Флойд предложил алгоритм 245 [13], улучшающий классический алгоритм Дж. Вильямса [15], который вместе с последующими модификациями других авторов известен как алгоритм Флойда.

Программа siftDown, используемая в программе sort для восстановления кучи, имеет следующие особенности. Корневой элемент a[1] в большинстве случаев перемещается вниз дерева близко к листьям. Перемещение элемента на следующий уровень вниз по дереву требует два сравнения между элементами. Идея алгоритма 245 в том, что вместо последовательного обмена элементов при просеивании вниз можно последовательно поднять вверх элементы дерева, а затем вставить просеиваемый элемент в нужное место.

Описываемый алгоритм базируется на более радикальной модификации. Сначала реализуется прохождение вниз вдоль пути дерева от корня до листа по максимальным сыновьям. Этот путь называется *специальным*. Достигнутый лист также называется *специальным*. Отметим, что продвижение по пути на следующий уровень требует одно сравнение между сыновьями вместо двух в siftDown. Далее, двигаясь вверх от специального листа, находим нужную позицию для элемента a[1]. Наконец, реализуется поднятие элементов вверх по специальному пути, находящихся выше новой позиции для a[1]. Отметим, что массив, полученный поднятием элементов по специальному пути кучи, также является кучей.

Программа для алгоритма Флойда реализуется модификацией приведенной выше программы классического алгоритма. Заменим вызов siftDown в программе sortG на вызов siftEl.

```
sortG(Arn a, nat1n m: Arn a')
  pre heap(a, 1, m) & sorted(a, m + 1) & twoPart(a, m + 1)
{     if (m = 1) a' = a
      else { siftEl(a with [m: a[1]], m – 1, a[m]: Arn c);
             sortG(c, m - 1 : a')
      }
} post sorted(a') & perm(a, a') measure m;
```

Отличие здесь лишь в том, что элемент a[m] не записывается на место элемента a[1], а помещается в вызове siftEl дополнительным параметром. Вместо обмена элементов a[1] и a[m] здесь реализуется лишь перемещение элемента a[1] в позицию элемента a[m]. В остальном спецификации программ siftEl и siftDown совпадают. В соответствии с алгоритмом Флойда программа siftEl следующая:

```
siftEl(Arn a, nat1n  m, T e: Arn c)
    pre heap(a with [1: e], 2, m)
{       getSpecLeaf(a, 1, m: nat1n  leaf);
        midNode(a, m, leaf, e: nat1n p);
        shiftUp(a, m, p, e: c)
} post perm(a with [1: e], c, m) & heap(c, 1, m);
```

Здесь leaf – индекс элемента, являющегося специальным листом, p – индекс элемента в специальном пути, куда следует записать элемент e, равный старому значению a[m]. Программа getSpecLeaf находит специальный лист, midNode – определяет индекс p, а shiftUp – поднимает на одну позицию вверх элементы специального пути, начиная с позиции p. Отметим, что элемент a[1] считается отсутствующим; его место в спецификации занимает элемент e.

Условие того, что некоторая вершина a[s] находится на *специальном* пути от вершины a[k], представляется следующей рекурсивно определяемой формулой:

```
formula specPath(Arn a, nat1n k, m, s) =
  s = k or left(k) = s & s = m or
    right(k) <= m & specPath(a, (a[left(k)] >= a[right(k)])? left(k): right(k), m, s)
```

Второй дизъюнкт формулы определяет случай, когда имеется только один (левый) потомок вершины a[k]. *Специальный лист* a[leaf] на пути от вершины a[k] определяется следующей формулой:

```
formula specLeaf(Arn a, nat1n k, m, leaf) =
    leaf ≤ m & left(leaf) > m & specPath(a, k, m, leaf)
```

Программа getSpecLeaf определения индекса leaf специального листа реализуется следующим рекурсивным алгоритмом:

```
getSpecLeaf(Arn a, nat1n  k, m: nat1n leaf)
    pre k <= m
{    nat1  j = left(k);
      if (j > m) leaf = k
      else if (j = m) leaf = j
      else  getSpecLeaf(a, ((a[j] >= a[j + 1]) ? j : j + 1), m : leaf)
} post specLeaf(a, k, m, leaf)
  measure (k>=n)? 0 : n - k;
```

Применена оптимизация: right(k) заменено на j+1.

Далее требуется найти позицию p на специальном пути, куда следует поместить элемент e, сдвинув элементы выше позиции p на один элемент вверх. Измененный путь должен удовлетворять свойству кучи. Во-первых, элемент e в позиции p должен удовлетворять свойству кучи, что определяется следующей формулой:

```
formula heapNode(Arn a, nat1n m, p, T e) =
    (left(p) ≤ m ⇒ e ≥ a[left(p)]) & (right(p) ≤ m ⇒ e ≥ a[right(p)]) .
```

Во-вторых, элемент a[p] должен быть не меньше e. Предпочтительно использовать условие a[p] > e для уменьшения числа сдвигаемых вверх элементов специального пути. Однако при p = 1 условие a[p] > e не имеет смысла, поскольку позиция 1 «пуста» – там находится элемент, перемещенный в позицию m. Поэтому более точным является условие a[p] > e ∨ p = 1. Полное условие сохранения свойства кучи для пути, полученного вставкой элемента e в позицию p специального пути от вершины a[1] до вершины a[s], представлено формулой:

```
formula MiddleNode(Arn a, nat1n  m, s, p, T e) =
  (a[p] > e or p = 1) & heapNode(a, m, p, e)
```

На специальном пути от a[1] до a[s] программа midNode определяет такую позицию p для вставки туда элемента e, чтобы модифицированный путь удовлетворял свойству кучи.

```
midNode(Arn a, nat1n  m, s, T e: nat1n p)
    pre specPath(a, 1, m, s) & heapNode(a, m, s, e)
{       if (a[s] > e or s = 1) p = s
        else midNode(a, m, father(s), e: p)
} post specPath(a, p, m, s) & MiddleNode(a, m, s, p, e) measure s;
```

В постусловии первый конъюнкт, постулирующий, что вершина a[s] находится на специальном пути от вершины a[p], вставлен для упрощения доказательства корректности программы sort.

Программа shiftUp реализует вставку элемента e в позицию p. На специальном пути от a[1] до a[p] проводится сдвиг на одну позицию вверх элементов пути, начиная с позиции p.

```
shiftUp(Arn a, nat1n m, nat1n p, T e: Arn a')
    pre heap(a with [1: e], 2, m) & specPath(a, 1, m, p) &
                heapNode(a, m, p, e) & (a[p] ≥ e or p = 1)
{       if (p = 1) a' = a with [1: e]
        else shiftUp(a with [p: e], m, father(p), a[p]: a')
} post perm(a with [1: e], a', m) & heap(a', 1, m) measure p;
```

Подпрограмма sort для алгоритма Флойда приведена ниже на императивном расширении языка P. Подпрограмма получена как результат следующих трансформаций. Переменные a' и c заменены на a; s и leaf – на p. Хвостовая рекурсия замена циклами. Тела программ getSpecLeaf, midNode, shiftUp, siftEl и sortG подставлены в программу sort. Проведены упрощения и оформления заголовков циклов.

```
sort(Arn a)
{   nat1n p;
    for(nat1n m = n; m ≠ 1; ) {
        T e = a[m]; a[m] = a[1]; m = m – 1;
        for(nat1n  k = 1; ; ) { nat1n  j = left(k);
                                if (j > m) { p = k; break }
                                else if (j = m) { p = j; break }
                                else  if (a[j] >= a[j + 1]) k = j
                                else k = j + 1
        }
        for( ; a[p] ≤ e & p ≠ 1; p = father(p)) {};
        for( ; p ≠ 1; p = father(p)) { T t = a[p]; a[p] = e; e =t}
        a[1] = e
    }
}
```

6. УЛУЧШЕННЫЙ АЛГОРИТМ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ

Алгоритм Флойда и его модификации (например, [17, 18, 19]) оптимизируют просеивание вниз элемента a[1] для восстановления двоичной кучи. Однако просеивания вниз можно избежать, если вместо обмена элементов a[1] и a[m] переместить a[1] в конец специального пути на место специального листа a[s], подняв все элементы пути на позицию вверх. Позиция m, куда в конце сортировки должен попасть перемещенный элемент a[1], запоминается в дополнительном массиве рангов: ra[s] = m. При этом позиция s уже не принадлежит куче. Куча организована другим способом: позиция k принадлежит куче, если ra[k] = 0. После завершения процесса сортировки необходимо переместить все элементы в соответствии с их рангами.

Данный алгоритм представлен в работе [8] с оценками по числу сравнений: n log n - 0.788928n в худшем случае и n log n - n – в среднем. Он быстрее других алгоритмов пирамидальной сортировки [16] и оказывается быстрее других видов сортировки (Clever quicksort, mergesort) при n > 900000 [6].

Усложнился переход от вершины к его потомку. Например, проверка наличия левого потомка a[k] реализуется более сложным условием: left(k) <= n & ra[left(k)] = 0. Это условие можно заменить на ra[left(k)] = 0, если доопределить массив ra отличным от нуля за пределами диапазона 1..n. Данное улучшение (отсутствующее в [8]) упрощает и ускоряет алгоритм, удлиняя вдвое массив рангов. Описываемый далее алгоритм имеет те же оценки по числу сравнений, что и [8], но быстрее его. Ниже представлена программа initR инициализации массива ra:

```
type nat2n = 2 .. n;
type natn2 = 1 .. n*2 + 1;
type RANK = array (natn2, natn);
formula Prank0(RANK ra) = ∀ i = 1 .. n.  ra[i] = 0;
formula Prank2(RANK ra) = ∀ i = n+1 .. 2*n+1.  ra[i] ≠ 0
initR( : RANK ra) { ra = for(natn2 j){   case 1..n: 0
                                         case n+1 .. 2*n + 1: 1}
} post  Prank0(ra) & Prank2(ra)
```

Массив ra достаточно длинный: right(k) ∈ natn2 для k=1..n. В диапазоне 1..n он инициализирован нулями, а за его пределами – единицами. Ниже представлена начальная часть программы пирамидальной сортировки.

```
HeapSort(Arn a: Arn a')
{ buildHeap(a: Arn b);
      initR( : RANK ra);
      sort(b, ra: a')
} post sorted(a') & perm(a, a');
sort(Arn a, RANK(n) ra: Arn a')
pre heap(a, n, ra) & Prank0(ra)
{       sortR(a, ra, n : Arn b, RANK(1) rb);
        rearrange(b, rb, 2:  a')
}  post sorted(a') & perm(a, a');
```

Здесь программа sortR сортирует массив a по описанной выше схеме, формируя также массив рангов rb, в соответствии с которым программа rearrange переставляет элементы массива b на окончательные позиции. Далее представлен тип RANK(m) как подмножество типа RANK, в котором однозначно определены ранги со значениями от m+1 до n для отсортированных элементов массива a.

```
formula Prank(RANK ra, nat1n m) =
                        Prank2(ra) & ∀k = m+1..n. ∃! nat2n d. ra[d] = k;
type RANK(nat1n m) = subtype(RANK ra: Prank(ra, m));
```

Программа sortR реализует сортировку массива a в предположении, что отсортированы элементы, которые окончательно будут занимать позиции от m+1 до n, а элементы, оставшиеся в куче, меньше любого из отсортированных элементов.

```
sortR(Arn a, nat1n  m, RANK(m) ra: Arn a', RANK(1) ra')
  pre heap(a, m, ra) & sortedR(a, m, ra) & heapLess(a, m, ra)
{       if (m = 1) {a' = a || ra' = ra}
        else { reheap(a, m, ra, 1, a[1]: Arn b, RANK(m-1) rb);
               sortR(b, m – 1, rb: a', ra')  }
} post Frank(2, ra') & sortedR(a', 1, ra') & perm(a, a') & firstLess(a');
```

Программа reheap поднимает элементы специального пути на позицию вверх, перемещает элемент a[1] на место специального листа, исключая его из кучи и назначая ему ранг m.

Определим предикаты, входящие в предусловие и постусловие. Формула для двоичной кучи аналогична приведенной выше, но здесь она дополняется предикатом pathToRoot, определяющим, что куча представлена единым деревом с корнем a[1].

```
formula pathTo(Arn a, natn2 p, nat1n k, m, RANK(m) ra)  =
  (ra[p] = 0) &
        (k = p or pathTo(a, left(p), k, m, ra) or pathTo(a, right(p), k, m, ra));
formula pathToRoot(Arn a, nat1n k, m, RANK(m) ra) =
        pathTo(a, 1, k, m, ra);
formula heap(Arn a, nat1n m, RANK(m) ra) =
    ∀ nat1n i. ra(i) = 0 ⇒
          (ra[left(i)] = 0 ⇒ a[i] ≥ a[left(i)]) &
          (ra[right(i)] = 0 ⇒ a[i] ≥ a[right(i)]) &
          pathToRoot(a, i, m, ra);
```

Для адекватного определения других предикатов с учетом рангов введем функцию inv, обратную по отношению к рангам, такую что inv(ra[s]) = s. Аккуратное определение обратной функции требует двух дополнительных параметров.

```
type natm1(nat1n m)  = m+1 .. n;
formula inv(nat1n m, RANK(m) ra, natm1(m) k : nat1n) = (nat1n d : ra[d] = k);
```

Выражение (nat1n d : ra[d] = k) определяет единственное значение d, удовлетворяющее свойству ra[d] = k. Наличие такого значения следует из определения типа RANK(m).

Сортированность sortedR и условие heapLess того, что отсортированные элементы не меньше элементов кучи (аналог twoPart), представляются следующими формулами:

```
formula sortedR(Arn a, nat1n m, RANK(m) ra) =
      ∀ i, j= m+1..n. i ≤ j ⇒ a[inv(m, ra, i)] ≤ a[inv(m, ra, j)];
formula heapLess(Arn a, nat1n m, RANK(m) ra) =
      m < n ⇒ a[1] ≤ a[inv(m, ra, m+1)];
```

В дополнении к этим для верификации потребовалось условие, что в конце сортировки единственный элемент кучи a[1] не больше всех других элементов:

```
formula firstLess(Arn a) =  ∀ nat2n j. a[1] ≤ a[j];
```

Программа rearrange реализует перестановку элементов и их рангов таким образом, чтобы в итоге ранги совпали с позициями элементов. В предикате Frank это условие реализуется для элементов с позицией меньше k.

```
formula Frank(nat k, RANK(1) ra) =
                ∀ nat2n t. 1 <= ra[t] <= n & (t < k ⇒ ra[t] = t);
```

При доказательства предиката Frank(2, ra′) в постусловии необходимо будет доказать, что ra′[1] = 0.

Программа reheap поднимает элементы специального пути на позицию вверх в предположении, что элементы от корня до вершины a[k] уже подняты, и таким образом, позиция k пуста. Элемент e, равный исходному элементу a[1], записывается в позицию специального листа. Ему устанавливается ранг m, что приводит к исключению его из кучи.

```
reheap(Arn a, nat2n m, RANK(m) ra, nat1n k, T e: Arn a′, RANK(m-1) ra′)
     pre heap(a, m, ra) & sortedR(a, m, ra) & eLess(a, m, ra, e) & a[1] <= e
{         natn2 j = left(k);
          if (ra[j] = 0)  if (ra[j+1] = 0 & a[j] <= a[j+1])
                                    reheap(a with [k: a[j+1]], m, ra, j+1, e: a′, ra′)
                          else reheap(a with [k: a[j]], m, ra, j, e: a′, ra′)
          else if (ra[j+1] = 0) reheap(a with [k: a[j+1]], m, ra, j+1, e: a′, ra′)
          else { a′ = a with [k: e] || ra′ = ra with [k: m] }
} post heap(a′, m - 1, ra′) & sortedR(a′, m - 1, ra′) & heapLess(a′, m - 1, ra′) &
                    perm(a with [k: e], a′)
     measure (k>=n)? 0 : n - k;
```

Условие того, что максимальный элемент кучи e не больше элементов отсортированной части массива a, представлено формулой:

```
formula eLess(Arn a, nat1n m, RANK(m) ra, T e) =
     m < n ⇒ e ≤ a[inv(m, ra)[m+1]];
```

Программа rearrange переставляет элементы массива a и их ранги так, чтобы в итоге ранги совпали с позициями элементов. Если ранг очередного элемента a[k] отличен от k, то он переставляется на свое место, обмениваясь с элементом a[ra[k]]. Программа работает в предположении, что для элементов с позицией, меньшей k, позиции и их ранги совпадают.

```
rearrange(Arn a, RANK(1) ra, nat k: Arn a′)
    pre k>=2 & Frank(k, ra) & sortedR(a, 1, ra) & firstLess(a)
{       if (k ≥ n) a′ = a
        else if (rank[k] = k) rearrange(a, ra, k+1: a′)
        else rearrange( a with [k: a[ra[k]], ra[k]: a[k]],
                                  ra with [k: ra[ra[k]], ra[k]: ra[k]], k:  a′)
} post sorted(a′) & perm(a, a′)
   measure ((k≥n)? 0 : n – k) + size((nat2n k : ra(k) ≠ k));
```

Функция size вычисляет число элементов множества, подставляемого параметром.

Итоговая программа на императивном расширении языка P приведена ниже.

```
HeapSort(Arn a)
{ buildHeap(a);
  RANK ra;
  for(nat1 j = 1; j <= n; j =j+1) ra[j] = 0;
  for(nat1 j = n+1; j <= 2*n + 1; j =j+1) ra[j] = 1;
  for(nat1n  m = n; m ≠ 1; m = m – 1) {
     T e = a[1];
     for( k = 1; ; ) {
        j = left(k);
        if (ra[j] = 0)  if (ra[j+1] = 0 & a[j] <= a[j+1])
                                          {a[k] = a[j+1]; k = j+1}
                        else {a[k] =a[j]; k = j}
        else if (ra[j+1] = 0) {a[k] = a[j+1]; k = j+1}
        else { a[k] = e; ra[k] = m; break }
     }
  }
  for( k = 2; k < n; )
       if (ra[k] = k) k = k+1
       else {T t = a[k]; a[k] = a[ra[k]; a[ra[k]] = t;
                                 nat s = ra[k]; ra[k] = ra[ra[k]], ra[ra[k]] = s }
}
```

Возможна более простая и эффективная модификация последнего алгоритма, в которой отсортированный массив записывается в другой массив b. При этом массив рангов можно заменить битовым массивом длины 2*n+1.

7. ОБЗОР РАБОТ

Существуют десятки различных модификаций алгоритма пирамидальной сортировки. Значительная их часть базируется на разных модификациях двоичной кучи. Двоичная куча является частным случаем d-кучи – дерева, в котором вершина дерева может иметь d потомков. Доказано, что наилучшая оценка для худшего случая реализуется для троичной кучи [5, 10]. Эдвард Дейкстра разработал алгоритм **smoothsort** [12], являющийся модификацией пирамидальной сортировки. Алгоритм работает быстрее в случаях, когда массив является частично сортированным. Вместо одной кучи

используется несколько двоичных куч, а их размеры соответствуют числам Леонардо. Имеется другой алгоритм [14], использующий несколько куч. После обмена элементов a[1] и a[m] получаются две кучи, корни которых – сыновья прежней вершины a[1]. Далее обмен проводится с корнем одной из двух куч и т.д. Кучи объединяются, когда их становится много. Алгоритм **weak-heapsort** [2] использует *слабую двоичную кучу* с возможными пропусками в позициях листьев на последних двух рядах дерева и отсутствием левого потомка для корневой вершины; дальнейшие улучшения алгоритма описаны в [4]. Алгоритм **ultimate heapsort** [3] использует *двухуровневую кучу* с дополнительным условием: элемент левой части больше любого элемента правой части.

Имеется ряд различных улучшений алгоритма Флойда. Вместо последовательного поиска позиции p, реализуемого в midNode, для перемещения туда элемента a[1] Carlsson предложил использовать бинарный поиск [17]. Отметим, что операция father(s) реализуется сдвигом на один бит вправо, а позиция j-го предка получается сдвигом s на j битов вправо. В работе [18] применяются дополнительные структуры, фиксирующие пути по максимальным сыновьям. В программе shiftUp используется операция обмена элементов для реализации сдвига на одну позицию вверх вершин, начиная от позиции p. Перемещение вершин на отрезке пути от позиции p, реализуемое в shiftUp, Wegener I. предложил проводить от корня движением к позиции p [19], не используя при этом обмена между вершинами. Другая его модификация – поиск позиции p движением начиная от 2/3 высоты дерева [7].

В технологии программирования на базе инвариантов [9, 11] построению каждого цикла предшествует конструирование соответствующего инварианта цикла. Технология иллюстрировалась также на классическом алгоритме пирамидальной сортировки [15] с дедуктивной верификацией на PVS [20] в интеграции с SMT-решателем Yices [21]. В статье [1] описывается программный синтез классического алгоритма [15] в виде программы на языке Haskell, при этом сортируемый массив представляется списком, а двоичная куча моделируется двоичным деревом.

8. ОПЫТ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ АЛГОРИТМОВ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ

Для каждого из трех представленных выше алгоритмов пирамидальной сортировки проведена дедуктивная верификация предикатных программ.

Метод дедуктивной верификации описан в работах [22, 23]. Корректность программы в виде определения предиката

A(x: y) **pre** P(x) { S(x: y) } **post** Q(x, y)

представляется следующими формулами тотальной корректности:

$$P(x) \Rightarrow [\ L(S(x{:}\ y)) \Rightarrow Q(x, y)\ ] \ \&\ \exists y\ L(S(x{:}\ y)) \qquad (1)$$
$$P(x)\ \&\ Q(x, y) \Rightarrow L(S(x{:}\ y)) \qquad (2)$$

где L(S(x: y)) есть логика оператора S(x: y). Формула (1) используется для общего случая, (2) – для однозначной и тотальной спецификации [P(x), Q(x, y)]. Однозначность и тотальность реализуются при истинном предусловии P(x) и определяются как для математических функций. Для языка P построены две системы правил, упрощающих генерацию набора формул корректности программы по формулам (1) и (2). Данный метод верификации проверен примерно на 30 небольших программах. Формулы корректности, сгенерированные для этих программ, доказаны в системе автоматического доказательства PVS [20].

В процессе верификации программ пирамидальной сортировки обнаружено более 15 ошибок. Но не в программах, все ошибки – в спецификациях. Половина из них – ошибки недостаточно точных условий. Например, первоначально постусловие программы siftDown включало предикат perm(a, c). При верификации программы sortG обнаружилась недоказуемость предиката sorted в предусловии рекурсивного вызова sortG. Чтобы обеспечить доказуемость, было уточнено постусловие siftDown: условие perm(a, c) было заменено на perm(a, c, m), ограничивающее перестановочность первыми m элементами. Аналогичное уточнение проведено для постусловия программы siftEl. При верификации улучшенного алгоритма пирамидальной сортировки проведен каскад уточнений. Так, дополнительные предикаты Frank(2, ra') и firstLess(a'), вставленные в постусловии программы sortR, совершенно необходимы для доказательства корректности вызова rearrange в программе sort.

В процессе верификации алгоритма улучшенной сортировки обнаружены такие ошибки, исправление которых потребовало дважды серьезно модифицировать большую часть спецификаций. Серьезные трудности в спецификации и доказательстве обусловлены введением функции inv, обратной по отношению к рангам.

Набор формул корректности, генерируемых на базе формулы (2) для случая однозначной спецификации, проще, чем соответствующий набор, генерируемый по формуле (1). По этой причине ранее верификация прово-

дилась по формуле (2). Наличие неоднозначных спецификаций в программах сортировки вынудило использовать верификацию по формуле (1). Парадоксально, но доказательство формул корректности по формуле (2) оказалось более сложным и трудоемким, чем для формулы (1). Причина в том, что вместе с теоремой существования часто требуется доказать теорему единственности. Например, наряду с доказательством теоремы существования

theorem ∀ Arn a, nat1n p. ∃ Arn b. sorted(b, 1, p) & perm(a, b, p)

требуется доказать следующую теорему единственности:

theorem ∀ Arn a, b. perm(a, b) & sorted(a) & sorted(b) ⇒ a = b

Важное следствие данного факта – классические методы программного синтеза, базирующиеся на извлечении программы из доказательства теоремы существования, сложнее и более затратные в сравнении с дедуктивной верификацией.

Спецификация и доказательство корректности алгоритмов на PVS заняло около двух месяцев, что могло бы быть оправдано, только если бы разработанный алгоритм использовался в приложении с довольно высокой ценой ошибки. Полный набор условий корректности, теорий для перестановок, сортировки и операций с двоичной кучей, дополнительные леммы и все доказательства доступны в формате системы PVS:
http://www.iis.nsk.su/persons/vshel/files/heapsort.zip.

9. ЗАКЛЮЧЕНИЕ

Описаны три алгоритма пирамидальный сортировки: классический, алгоритм Флойда и улучшенный алгоритм, причем последний является самым быстрым из существующих алгоритмов сортировки. Очередной алгоритм является развитием предыдущего и сложнее его. Такой способ изложения от простого алгоритма к более сложному улучшает понимание. Императивная программа каждого алгоритма, получаемая применением трансформаций, короче, но менее понята по сравнению с исходной программой на языке P.

Нетривиальной задачей является написание спецификаций в виде предусловий и постусловий. Написание спецификаций значительно сложнее построения соответствующих программ. Кроме того, в дедуктивной верификации требуются достаточно точные спецификации, иначе условия корректности оказываются недоказуемыми. Отметим, что формальные специ-

фикации применяются также и в других видах верификации программ, например, при автоматической генерации тестов.

СПИСОК ЛИТЕРАТУРЫ

1. Guttmann W.N. Deriving an Applicative Heapsort Algorithm. – Ulm Univ., 2002 – 73 p.
2. Edelkamp S., Wegener I. On the performance of weak-heapsort // Lect. Notes Comput. Sci. – 2000. – Vol. 1770 – P. 254–260.
3. Katajainen J. The ultimate heapsort // Computing: The Australasian Theory Symposium. – 1998. – P. 87–96.
4. Edelkamp S., Stiegeler P. Pushing the Limits in Sequential Sorting // Lect. Notes Comput. Sci. –2001. – Vol. 1982. – P. 39–50.
5. Islam T.M., Kaykobad M. Worst-case analysis of generalized heapsort algorithm revisited // Int. J. Comput. Math. – Vol. 83, No. 1. – 2006. – P. 59–67.
6. Sharma V., Singh S., Kahlon K. S. Performance study of improved Heap Sort algorithm and other sorting algorithms on different platforms // IJCSNS International J.l of Computer Science and Network Security. – 2008. – Vol. 8, No. 4. – P.101–105.
7. Wegener I. A simple modification of Xunrang and Yuzhang's HEAPSORT variant improving its complexity significantly // Computer Journal 36. – Vol. 3. – 1993. – P. 286–288.
8. Wang X.D., Wu Y. J. An improved HEAPSORT algorithm with n logn – 0.788928n comparisons in the worst case // J. Computer Science and Technology. – 2007. – Vol. 22 (6). – P. 898–903.
9. Eriksson J., Back R. Applying PVS background theories and proof strategies in invariant based programming // Lect. Notes Comput. Sci. – 2010. – Vol. 6447. – P. 24–39.
10. A new variant of heapsort: A 3-heap bottom up approach. – Islamic University of Technology, 2003.
11. Eriksson J. Tool-Supported Invariant-Based Programming. – Turku Centre for Computer Science, 2010. – 164 p. (TUCS Dissertations; 127).
12. Dijkstra E.W. Smoothsort – an alternative to sorting in situ (EWD-796a). – E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original; transcription)
13. Floyd R.W. Algorithm 245 – Treesort 3 // Commun. ACM. – 1964. – Vol. 7 (12). – P. 701.
14. Levendeas D., Zaroliagis C. Heapsort using multiple heaps // 2nd Panhellenic Student Conference on Informatics -- EUREKA. – 2008. – P. 93–104.
15. Williams J.W.J., Algorithm 232 // Commun. ACM. – 1964 – P. 347–348**.**
16. Sharma V., Singh S., Kahlon K.S. Comparative performance study of improved heap sort
17. Carlsson S. A variant of HEAPSORT with almost optimal number of comparisons // Inform. Process. Lett. – 1987. – N 24. – P. 247–250.

18. Shahjalal Md., Kaykobad M. A new data structure for heapsort with improved number of comparisons // WALCOM. – 2007. – P. 88–96.
19. Wegener, I., 1990. BOTTOM-UP-HEAPSORT, a new variant of heapsort beating on average QUICKSORT (if n is not very small) // Lect. Notes Comput. Sci. – 1990. – Vol. 452. – P. 516–522.
20. PVS Specification and Verification System. – SRI International. – http://pvs.csl.sri.com/
21. Dutertre B., Moura L. The YICES SMT solver. – 2006. – http://yices.csl.sri.com/tool-paper.pdf
22. Шелехов В.И. Методы доказательства корректности программ с хорошей логикой // Межд. конф. "Современные проблемы математики, информатики и биоинформатики", посвященная 100-летию со дня рождения А.А. Ляпунова. – 2011. – 17с. http://conf.nsc.ru/files/conferences/Lyap-100/fulltext/74974/75473/Shelekhov_prlogic.pdf
23. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия. – 2011. – № 2. — С. 14–21.
24. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. – Новосибирск, 2010. – 42с. – (Препр. / ИСИ СО РАН; № 153).

В.И. Шелехов

РАЗРАБОТКА И ВЕРИФИКАЦИЯ АЛГОРИТМОВ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ В ТЕХНОЛОГИИ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Препринт
164

Рукопись поступила в редакцию 25.11.12
Редактор Т. М. Бульонкова
Рецензент М.А. Бульонков

Подписано в печать 15.12.12
Формат бумаги 60 × 84 1/16 Объем 1.48 уч.-изд.л., 1.6 п.л.
Тираж 50 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42