Efficient representation of integer sets

Marco Almeida Rogério Reis

Technical Report Series: DCC-2006-06
Version 1.0

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Efficient representation of integer sets

Marco Almeida Rogério Reis
{mfa,rvr}@ncc.up.pt
DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal

Abstract

We present a solution that uses bitmaps and bitwise operators to represent non-negative integer sets and implement some common set theoretical operations. We also show how it is possible to extend this representation to set partitions.

1 Introduction

Finite sets are a common and frequently used data structure. Several algorithms make heavy use of these objects, and, in some cases, they are at the very core of the algorithm. This makes it necessary to have a simple and efficient way to represent and manipulate these objects. We present a solution that uses bitmaps and bitwise operators to represent non-negative integer sets and implement some common set theoretical operations. We also show how it is possible to extend this representation to set partitions.

This report is organized as follows. In the next section we describe our set representation using a bitmap. Section 3 explains how to complete the bitmap representation so that we can also deal with set partitons. In Section 4 we present the implementation details of a set library based on this representation and in Section 5 we describe some computational advantages. The complete API is given in Appendix A.

2 Sets with bitmaps

Using a bitmap with n bits we can represent a set of non-negative integers in the range $0, \ldots, n-1$ in the following way. For each bit in the bitmap, if the bit's value is 1, we assume that the integer corresponding to the bit's position in the bitmap is a member of the set. The rest of the bits represent integers that do not belong to the set.

Let us consider the sets $A = \{2, 3, 5\}$ and $B = \{0, 2, 4\}$. With bitmaps of size eight these sets are written as 01110100 and 10101000, respectively. Bitwise operators can now be used to implement some common set theoretical operations. The bitwise `AND` and `OR`, for example, allow us to calculate the intersection and union, of two given sets, respectively. The unary bitwise **not** operator can be used to obtain a set's complement. Some examples are:

$A \cap B = \{2\}$

```
  00110100
AND 10101000
  = 00100000
```

$A \cup B = \{0, 2, 3, 4, 5\}$

```
  00110100
OR  10101000
  = 10111100
```

$\bar{A} = \{0, 1, 4, 6, 7\}$

```
NOT 00110100
  = 11001011
```

Other operations, such as set difference, can be implemented by combining these basic bitwise operators.

3 Set Partitions with bitmaps

Formally, a partition of a given set A is a set $\{A_i\}_{i \in I}$, where I is an index set, such that:

$$\begin{aligned} \forall i \in I,\ \emptyset \neq A_i \subseteq A \\ A = \cup_{i \in I} A_i \\ \forall i, j \in I, i \neq j,\ A_i \cap A_j = \emptyset \end{aligned}$$

We call each A_i a *block* of the partition.

If we tried to apply the former ideas to the representation of partitions, some problems would arise. In general, for any finite set S with $|A| = n \geq 0$, $\mathcal{P}(A) = 2^n$. A bitmap for a set partition of a set with n elements would have 2^n bits. It would also be extremely sparse, with only n bits actually used. A partition of a set with 32 bits would require a bitmap with 4294967296 bits. This is not a viable solution.

However, a partition P of a set S with n elements, can not have more than n blocks. Since each block is a subset of S, it can not hold any more than n elements. If we use bitmaps to represent each of the blocks in P, we will be needing, at most, n bitmaps with n bits each, in a total of n^2 bits. This is a considerably smaller bitmap, and, for the same set of 32 elements of the previous example, only 1024 bits are required. In more pratical terms, we are representing a partition not as a set of sets, but as an array of sets. This way, the space needed to represent a partition grows quadraticly and not exponencialy. The following example illustrates this approach.

Let us consider the set $S = \{0, 2, 3, 4\}$ and a partition $P = \{\{0\}, \{2, 3\}, \{4\}\}$. The bitmap 10111, with five bits, is enough to represent S. The naive approach we first described uses a bitmap B, with 2^5 bits, to represent P,

$$B = 10011000000000000000000000000000.$$

Of the whole thirty two bits, only the first, the fourth and the fifth are active. Instead, if we use the proposed alternative, we need only five bitmaps with five bits each, which will require a total of twenty five bits.

$$\begin{aligned} B_0 &= 10000 \\ B_1 &= 00110 \\ B_2 &= 00001 \\ B_3 &= 00000 \\ B_4 &= 00000 \end{aligned}$$

Each one of these bitmaps B_i represents a block of the partition P and can be written as an array.

$$A = [0000, 00110, 00001, 00000, 00000]$$

This representation allows us to avoid dealing with a huge, sparse, bitmap while making use of all the advantages of the integer set representation we described earlier.

4 Implementation

When efficiency is a concern, representing set partitions with bitmap arrays is a solution that actually solves some problems in a simple and elegant way. There are, however, some set theoretical operations that require special attention. Finding out if some set S is a block of a given partition P, for example, is no longer a matter of performing a simple bitwise `AND` between P and S — except, of course, if S has the same size as P, but this would imply a waste of space that we wish to avoid.

In a more general way, operations that need to search the partition (such as block removal or existence testing) can be quite expensive in computational terms. We propose a solution based on AVL trees. These are binary search trees on which the lookup, insertion and deletion operations take $O(\log n)$ time.

4.1 AVL trees

An *AVL tree* (or height-balanced tree) [Knu98] is a self-balancing binary search tree. The balance factor of a node is the difference between the height of its subtrees. An AVL tree is considered balanced if every node has a balance factor of, at most, one. A node with any other balance factor is considered unbalanced and requires rebalancing the tree.

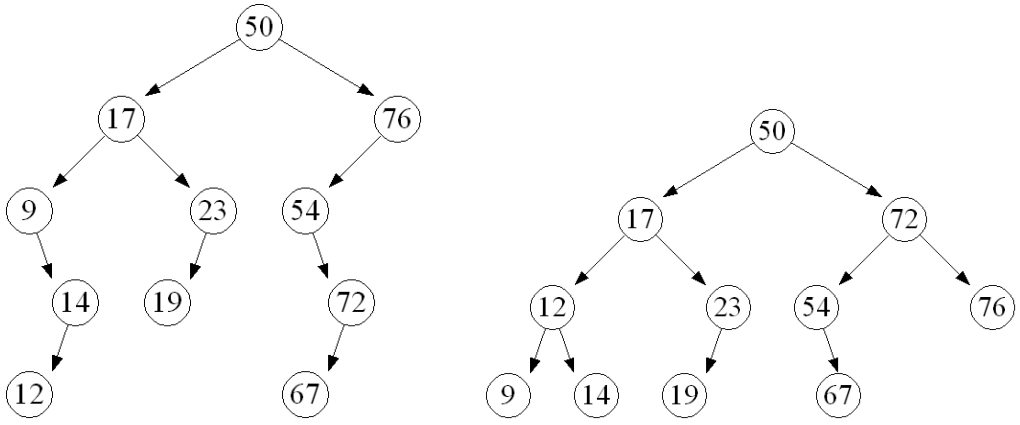

Figure 1: a) example of an unbalanced tree; b) the same tree after rebalancing (now an AVL)

In a balanced AVL tree, searching, inserting and removing nodes are operations that require $O(\log n)$ time (where n is the number of nodes in the tree) on both the average and the worst case. Clearly, the insertion or removal of nodes may force the tree to be rebalanced. In a general way, this is a very efficient data structure that actually outperforms *red-black trees* or *splay trees* in search intensive applications [Pfa04].

4.2 Bitmap arrays with AVL trees

In Section 3 we proposed a representation of set partitions based on bitmap arrays. It is quite easy to have a total order over the elements of such an array. From the definition of set partition, we know that the intersection of any two blocks must result in the empty set and that every block contains at least one element. This allows us to order the blocks, for

example, by comparing the first element of each block. Since we are using sets of non-negative integers, the usual $\leq$ relation can be used.

Having a total order defined, we can now use an AVL tree to represent the array of bitmaps. This way, all the operations necessary to the implementation of the usual set theoretical functions (searching, inserting and removing elements) will take $O(\log n)$ time. In an unordered array (or linked list for that matter) search alone is an $O(n)$ operation.

4.3 The library

Using the methods described in the previous sections, we have written a small library to handle non-negative integer sets. This library, `zset`, is written in the `C` programming language and implements the usual set theoretical operations.

Because of the need to deal with arbitrarily big bitmaps, which may exceed the size of a `C` integer (be it 32 or 64 bits), we used the GMP library. GMP is a free library for arbitrary precision arithmetic. It has a rich set of functions, with a regular interface and is designed to be as fast as possible, both for small and huge operands.

We used GMP's multiple precision integers (`mpz_t` type) to represent bitmaps. The logical and bit manipulation functions were used instead of the bitwise operators to implement all set theoretical functions.

As for the set partitions, and given the complexity of the algorithms involved, mainly the ones required to maintain balance, we used a small implementation of AVL trees for the C programming language [avl]. This library is also freely available under the LGPL license.

Some of the functions are simple enough to be implemented as *macros*. While at first this may look a good idea for efficiency concerns, it is not. Because the preprocessor replaces every macro call with its definition, macros are unknown to the debugger. This "feature" not only makes the debugging process a lot more complicated but it also renders profiling impossible. Instead of macros, we used the *inline* keyword in all function declarations. This is part of the ISO C99 standard and instructs the compiler to integrate the function's code into the code for its callers. It is just as fast as a macro [StGDC05] but allows the debugger to follow the function calls, thus enabeling both debug and profiling.

5 Computational advantages

Boolean algebras can represent basic operations of both set theory and propositional logic. Using bit sequences and propositional logic connectives, we can represent finite non-negative integer sets and implement some basic set theoretical operations such as union, intersection and difference.

In computational terms, there are two main advantages that come directly out of this representation. First, the amount of memory needed by this data structure is very small. Second, any general purpose modern CPU has very efficient implementations of bitwise operations. These instructions require very few clock cycles to complete, even when compared to the most elementary arithmetic instructions.

Bitmaps present significant advantages over equivalent simple set data structures such as arrays or linked lists. These are compact data structures which allow to store n elements in $\lceil n/w \rceil$ memory words. Small bitmaps (those with a size that does not exceed the word size) can be stored and manipulated in registers during the execution of several instructions without the need to access memory. This allows to exploit bit-level parallelism, limit

the memory access and maximize the use of the data cache, making it possible to obtain significant speed gains.

A API

We now present the API for `zset`, the library described in the previous section.

Module **set**

Data types

`typedef unsigned long int zset_elm_t;`
set element data type

`typedef avl_tree_t * partition_t;`
set partition data type

```
typedef struct zset {
    mpz_t s;
    unsigned long int n;
} *zset_t;
```
set data type

Macros

```
#ifdef INLINE
#define INLINE_DECL extern __inline__
#else
#define INLINE_DECL static
#endif
```

Functions

`INLINE_DECL zset_t zsetInit(unsigned long n)`
initialize a set with `n` elements

`INLINE_DECL void zsetDestroy(zset_t s)`
free the memory space allocated for set `s`

`INLINE_DECL void zsetAdd(zset_t s, zset_elm_t e)`
add element `e` to set `s`

`INLINE_DECL void zsetDel(zset_t s, zset_elm_t e)`
remove element `e` from set `s`

`INLINE_DECL void zsetIntersection(zset_t s, zset_t s1, zset_t s2)`
$s = s_1 \cap s_2$

`INLINE_DECL void zsetUnion(zset_t s, zset_t s1, zset_t s2)`
$s = s_1 \cup s_2$

`INLINE_DECL void zsetDifference(zset_t s, zset_t s1, zset_t s2)`
$s = s_1 - s_2$

`INLINE_DECL int zsetHasElm(zset_t s, zset_elm_t e)`
returns 0 if $e \notin s$

`INLINE_DECL int zsetIsSubzset(zset_t s1, zset_t s2)`
returns 0 if $s_1 \not\subseteq s_2$

`INLINE_DECL unsigned long int zsetCardinal(zset_t s)`
returns the number of elements in `s`

`INLINE_DECL partition_t partitionInit(void)`
initialize a partition

`INLINE_DECL void partitionDestroy(partition_t p)`
free the memory space allocated for partition `p`

`INLINE_DECL void partitionAdd(partition_t p, zset_t s)`
add set `s` to partition `p`

`INLINE_DECL void partitionDel(partition_t p, zset_t s)`
remove set `s` from partition `p`

`INLINE_DECL int partitionCardinal(partition_t p)`
returns the number of blocks in partition `p`

References

[avl] Gnu libavl, binary search trees library. http://www.stanford.edu/ blp/avl/.

[Knu98] Donald E. Knuth. *The Art of Computer Programming. Searching and Sorting.*, volume 3. AW, 1998.

[Pfa04] B. Pfaff. Performance analysis of bsts in system software. SIGMETRICS/Performance poster, June 2004.

[StGDC05] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, 2005. Version 4.0.2.