Partition Algorithms

Ralph Freese

March 4, 1997

These are algorithms for partitions on the set $\{0, 1, \ldots, n-1\}$. We represent partitions abstractly as forests, i.e., a collection of trees, one tree for each block of the partition. We only need the parent information about the tree so we represent the partition as a vector V with $\mathrm{V}[i]$ the parent of i unless i has no parent (and so is a root), in which case $\mathrm{V}[i]$ is negative the size of the block with i. In this scheme the least partition would be represented by the vector $\langle -1, -1, \ldots, -1\rangle$ and the greatest partition could be represented in many ways including the vector $\langle -n, 0, \ldots, 0\rangle$. [2] contains an elementary discussion of this type of representation of partitions.

We say that a vector representing a partition is in *normal form* if the root of each block is the least element of that block and the parent of each nonroot is its root. This form is unique, i.e., two vectors represent the same partition if and only if they have the same normal form. The examples above are in normal form. Algorithm 1 gives a simple recursive procedure for finding the root of any element i. Note that i and j are in the same block if and only if $root(i) = root(j)$.

```
procedure root(i, V)
    j ← V[i]
    if j < 0 then return(i)
    else return(root(j)) endif
endprocedure
```

Algorithm 1: Finding the root

The running time for *root* is proportional to the depth of i in its tree, so we would like to keep the depth of the forest small. Algorithm 2 finds the root and at the same time modifies V so that the parent of i is its root without increasing the order of magnitude of the running time.

In many applications you want to build up a partition by starting with the least partition and repeatedly join blocks together. Algorithm 3 does this.

Note that Algorithm 3 always joins the smaller block onto the larger block. This assures us that the resulting partition will have depth at most $\log_2 n$ as the next theorem shows.

```
procedure root(i, V)
    j ← V[i]
    if j < 0 then return(i)
    else V[i] ← root(j); return(V[i]) endif
endprocedure
```

Algorithm 2: Finding the root and compressing V

```
procedure join-blocks(i, j, V)
    r_i ← root(i, V); r_j ← root(j, V);
    if r_i ≠ r_j then
        s_i ← −V[r_i]; s_j ← −V[r_j]
        if s_i < s_j then
            V[i] ← r_j; V[j] ← −(s_i + s_j)
        else
            V[j] ← r_i; V[i] ← −(s_i + s_j)
        endif
    endif
    return(V)
endprocedure
```

Algorithm 3: Join two blocks together

Theorem 1 *If* Algorithm 3 *is applied any number of times starting with the least partition, the depth of the resulting partition will never exceed* $\log_2 n$.

Proof: Let i be a fixed node. Note an application of *join-blocks* increases the depth of i by at most 1 and, if this occurs, the size of the block with i is at least doubled. Thus the depth of i can be increased (by 1) from its original value of 0 at most $\log_2 n$ times.

This result shows that the time required to run the *join-blocks*–procedure m times is $O(m \log_2 n)$. In [1] Tarjan has shown that, if we use the *root* operation given in Algorithm 2, the time required is $O(m\alpha(m))$, where α is the pseudo-inverse of the Ackermann function. The Ackermann function is extremely fast growing and so α grows very slowly; in fact, $\alpha(m) \leq 4$ unless m is at least

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$$

with 65536 2's.

By Theorem 1 we may assume that all our (representations of) partitions have depth at most $\log_2 n$. The *rank* of a partition (in the partition lattice Π_n of an n element set) is $n - k$, where k is the number of blocks. The join of two partitions U and V can be found by executing *join-blocks*$(i, \mathrm{U}[i], \mathrm{V})$ for each i which is not a root of U. This can be done in time $O(\mathrm{rank}(\mathrm{U}) \log_2 n)$

and so in time $O(n \log_2 n)$. (Actually, such an algorithm should make a copy of V so the original V is not modified.)

It is relatively easy to write $O(n \log_2 n)$ time procedures for putting V into normal form and for testing if $\mathrm{V} \leq \mathrm{U}$ in Π_n. Finding an $O(n \log_2 n)$ time algorithm for the meet of two partitions is a little more difficult. Algorithm 4 does this. In this algorithm, HT is a hash table. (In place of a hash table, one could use a balanced tree or some other data structure described in texts on algorithms and data structures.)

```
procedure meet($V_1, V_2$)
    $n \leftarrow$ size($V_1$)
    for $i \in \mathbb{Z}$ with $0 \leq i < n$ do
        $r_1 \leftarrow root(i, V_1)$; $r_2 \leftarrow root(i, V_2)$
        if HT[$r_1, r_2$] is defined then
          $r \leftarrow$ HT[$r_1, r_2$]
          V[$r$] $\leftarrow$ V[$r$] $- 1$
          V[$i$] $\leftarrow r$
        else
          HT[$r_1, r_2$] $\leftarrow i$
          V[$i$] $\leftarrow -1$
        endif
    endfor
    return(V)
endprocedure
```

Algorithm 4: Meet of two partitions

References

[1] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach. **22** (1975), 215–225.

[2] M. A. Weiss, *Data Structures and Algorithm Analysis*, Benjamin Cummings, Redwood City, California, 1994, Second Edition.