A note on data structures for maintaining bipartitions

Gruia Calinescu

Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA
Available online 2 May 2006

Abstract

We consider the following problem: given a ground set $U$ of elements $\{1, 2, \ldots, n\}$, and a set $S$ of bipartitions of $U$, design a data structure to support the following three operations: Report($S$)—report the partition of $U$ induced by $S$, Insert($P$, $S$)—add a new bipartition $P$ to $S$, and Delete($P$, $S$)—delete the existing partition $P$ from $S$, where the partition of $U$ induced by $S$ is given by two elements of $U$ being in the same class if and only if they are in the same class for every bipartition of $S$.

We describe a straightforward deterministic data structure with an amortized bound of $O(n)$ per update, which is optimal.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Data structures

1. Introduction

We consider the following problem: given a ground set $U$ of elements $\{1, 2, \ldots, n\}$, and a set $S$ of bipartitions of $U$ (a bipartition is a partition into two parts), support the following three operations:

- Report($S$): report the partition of $U$ induced by $S$
- Insert($P$, $S$): add a new bipartition $P$ to $S$
- Delete($P$, $S$): delete the existing partition $P$ from $S$

where the partition of $U$ induced by $S$ is given by two elements of $U$ being in the same class if and only if they are in the same class for every bipartition of $S$.

In a paper obtained in parallel by Bender, Sethia, and Skiena [3], five results are presented. First, they provide a deterministic data structure which supports Report and Insert in $O(n)$ time, and Delete in $O(n \log |S|)$, for $S$ being composed of arbitrary partitions (not just bipartitions). Second, they provide a Monte Carlo algorithm which supports Report, Insert, and Delete in $O(n)$ time, for bipartitions only. Third, for bipartitions only, they provide a Las Vegas algorithm which supports Report and Insert in $O(n)$ time, and Delete in amortized $O(n \alpha(n))$ time, with $\alpha(n)$ being the inverse Ackerman function. They also describe data structures for reporting the approximate number of partitions separating two elements, and for reporting, inserting, and deleting a type of geometric bipartitions. The journal version of Bender, Sethia, and Skiena [4] extends the Monte Carlo algorithm to arbitrary partitions.

---

1 E-mail address: calinesc@iit.edu (G. Calinescu).
2 Research performed in part at DIMACS and supported in part by a DIMACS postdoctoral fellowship.

1570-8667/ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2006.03.002
Bender et al. [3] motivate this problem by classification and decision trees [1,5,6,11,12], well studied techniques [2, 8,9,15] with various applications such as optical character recognition [13,14]. Also VLSI design [10] uses bipartition-based placement heuristics (such as Fiducia-Mattheyses), and some bipartitions produced could be discarded later, with a report of the partition induced by the current set of bipartitions being necessary.

From a computational point of view, a bipartition is presented as a 0–1 vector of length \(n\). Our data structure is also based on bit vectors, one for each element of the ground set. The elements are kept sorted lexicographically according to their bit vectors, with positions corresponding to deleted partitions ignored. Inserting bipartitions is straightforward, while deleting bipartitions makes use of an additional pointer kept for each element, and is done by a merging procedure. The running time for Insert and Report operations is \(O(n)\) (and therefore within a constant of optimum), while deletions take amortized time \(O(n)\). We use the potential method [7] to prove the amortized time bound.

Our result is a deterministic counterpart to the Las Vegas algorithm of [3] with a slight improvement in the time complexity of deletions. The Las Vegas algorithm of [3] also has straightforward implementation, and its amortized time complexity is obtained from the involved analysis of the union-find data structure of Tarjan [16].

Another two operations can be easily supported in optimal time: given two elements, check whether two elements are in the same class of the partition induced by \(\mathcal{S}\), and given one element, list the elements in its class.

2. Data structure

Our data structure maintains for each element a 0–1 array (called element-array) of size \(t\), where \(t\) is a parameter related to \(|\mathcal{S}|\). This variable length property of the arrays is needed to overcome the fact that we do not know the total number of bipartitions in advance. This is not a real problem: whenever the element-arrays become half full, we start copying them (without the positions representing the already deleted bipartitions) in arrays twice as big. Precisely, for every Insert we copy two positions of each element-array into the future element-array, and therefore by the time the element-array overflows the future element-array is ready to be used, and is at most half full.

Conceptually, we use only the positions of the element-arrays corresponding to active bipartitions (those still in \(\mathcal{S}\), and we call active-arrays the arrays having only the active positions. We denote by \(m\) total number of bipartitions (active and nonactive) represented at this moment in the element-arrays. We always ignore the nonactive positions of the element-arrays.

An array \(\text{SORT}\) of length \(n\) keeps the elements sorted by lexicographical order of their active-arrays. Another array \(\text{LOC}[i]\) of length \(n\) points to the first active position (as an index from 1 to \(m\)) where the active-array of \(\text{SORT}[i]\) differs from the active-array of \(\text{SORT}[i-1]\), with the conventions that \(\text{LOC}[i] = m + 1\) when the two active-arrays coincide, and \(\text{LOC}[1] = 0\). These arrays (of total size \(O(tn)\)) constitute our data structure. We refer to Fig. 1 for an example.

To answer a Report query, traverse \(\text{SORT}\) and notice that if \(\text{LOC}[i] = m + 1\), then \(\text{SORT}[i]\) and \(\text{SORT}[i-1]\) are in the same class.

Updating the data structure after inserting a bipartition is also simple. We go through \(\text{SORT}\). We add a 0 or 1 in position \(m + 1\) of each element-array—according to the new bipartition. Before the insertion, the elements in the same class are in consecutive positions of \(\text{SORT}\). We need to rearrange them according to the new bipartition and update \(\text{LOC}\), which is straightforward.

2.1. The delete operation

Deleting a partition is more difficult and we will discuss the process in detail. Suppose \(c\)th bipartition is to be deleted. We need to keep sorted the list \(\text{SORT}\) and update \(\text{LOC}\) to take into account the fact that the \(c\)th position of the element-arrays is irrelevant. Our procedure is similar to merging.

We now give an outline of the update procedure. Among the indexes with \(\text{LOC} < c\), consider two consecutive one in \(\text{SORT}\): \(i\) and \(j + 1\). Consider three groups of elements (as they appear in \(\text{SORT}\)): the first \(i - 1\), the elements from \(i\) to \(j\), and the elements from \(j + 1\) to \(n\). Also after we start ignoring the \(c\)th position of the element-arrays, the active arrays of the elements from the first group remain smaller (in the new lexicographical order) than the active arrays of the elements from the second group, which remain smaller than the active arrays of the elements of the third group.
For the elements in the middle group, the bits in all active positions smaller than \( c \) are the same in the consecutive elements-arrays, since the differences start from \( LOC > c \). So we can separate the middle group and rearrange it separately. The elements of this group with 0 in the \( c \)th position of their array all appear before the elements of this group with 1 in the \( c \)th position. We copy \( SORT[i, \ldots, j] \) into two arrays, \( U \) and \( V \), to separate these two subgroups. The lengths of \( U \) and \( V \) are denoted respectively, by \( Umax \) and \( Vmax \). The cases of \( U \) or \( V \) being empty are trivial, so we assume that \( Vmax > 0 \) and \( Umax > 0 \).

For intuition, see Fig. 2. So \( SORT[i] = U_1 \), \( SORT[i + 1] = U_2 \) and so on till \( V_{V_{\text{max}}} = SORT[j] \). We will rearrange these elements, using the new lexicographical order, back in \( SORT \). We copy \( LOC[i, \ldots, j] \) into \( ULOC[1, \ldots, Umax] \) and \( VLOC[1, \ldots, Vmax] \). Notice that \( LOC[i] \) and \( LOC[j + 1] \) are not changed regardless of which element will occupy \( SORT[i] \) and \( SORT[j] \). Also notice that \( VLOC[1] = c \).

Note that the elements of \( V \) are already sorted lexicographically by their active arrays after making position \( c \) nonactive, since on position \( c \) all their element-arrays have 0. The same property holds for \( U \)—here on position \( c \) all the element-arrays have 1. Now what remain to be done is to merge the arrays \( U \) and \( V \). To make this merge fast we use the \( ULOC \) and \( VLOC \) arrays, and we need extra effort to keep \( LOC \) up to date.

We now describe the way \( ULOC \) and \( VLOC \) can be used to compare two elements for the merging. Suppose we are at the element \( l \) and we want to choose a next element from \( \{p, q\} \), where the active-arrays of both \( p \) and \( q \) are greater in lexicographical order than the active array of \( l \). Also suppose we know \( pNext \) and \( qNext \), where \( pNext \) and \( qNext \) are defined similar to \( LOC \): \( pNext \) is the first active index where the active arrays of \( l \) and \( p \) differ and \( qNext \) is the first active index where the active arrays of \( l \) and \( q \) differ. We have three cases: \( pNext < qNext \), \( pNext = qNext \), or \( pNext > qNext \).

If \( pNext < qNext \), then \( q \) is the next element in lexicographical order. Moreover, \( pNext \) is the first active index where the active arrays of \( q \) and \( p \) differ. The case \( pNext > qNext \) is similar. These two cases can be handled by the merging algorithm in constant time.

If \( pNext = qNext \), we only know that the active arrays of \( p \) and \( q \) coincide up to \( pNext \). Then, starting from \( pNext \), we check the active positions one by one (ignoring the nonactive positions) until we determine the first position where the active arrays of \( p \) and \( q \) differ—if they do.

This paragraph gives the intuition that the total time, over all deletions, is not too high, even if the process from the previous paragraph can result in high time consumption for a single deletion. Let \( m \) be the total number of Insert operations. For each comparison performed by any deletion, there is an increment for a certain \( LOC \) variable. This limits the total number of comparisons performed over all deletions to \( O(mn) \). A formal proof appears later.

The pseudocode of the merge procedure appears in Table 1. The index variable \( curr \) denotes the current position of \( SORT \) which we are filling. The variables \( Vnext \) and \( Unext \) are the indexes from \( V \) and \( U \) such that \( V[V_{\text{next}}] \) and
Fig. 2. The horizontal line represents the bipartition about to be deleted. A middle group is identified and further separated into arrays $U$ and $V$. $U$ and $V$ have to be merged.

$U[Unext]$ are the candidates to fill the current position of $SORT$. With the exception of the first iteration, when the while loop starts executing, the following invariants are satisfied. $Uguess$ is the first active position where $U[Unext]$ differs from $SORT[curr - 1]$, and $Vguess$ is the first active position where $V[Vnext]$ differs from $SORT[curr - 1]$. These invariants are easy to check and they imply the correctness of the procedure. Also note that $LOC[i] < c$ and it does not change.

2.2. Running time

We conclude with:

**Theorem 1.** With the above data structure, Report and Insert take $O(n)$ time, and Delete takes amortized $O(n)$ time.

**Proof.** We use potential-based amortized analysis. Define $\Phi(m) = n(m + 1) - \sum_{i=1}^{n} LOC[i]$ as the potential function. A Report operation takes actual time $O(n)$ and the potential function is not modified. For an Insert operation, the actual time is $O(n)$ and the increase in $\Phi(m)$ is also $O(n)$, and therefore the amortized time is $O(n)$. Copying the $SORT$ and $LOC$ arrays after an overflow does not change the potential function, as both $m$ and $LOC[i]$ decrease by the number of inactive positions.

For any Delete operation, the actual time is $O(n)$ plus the number of comparisons done in lines 12–15 of the pseudocode. Let $SORT^{-1}$ be the permutation inverse to the one described in the array $SORT$. We note that for an element $k$, $LOC[SORT^{-1}_k]$ does not change, unless:

- $k$ is being compared in lines 12–15 of the pseudocode with $k'$, and $k$ is found to be larger than $k'$ (by comparing lexicographically the active arrays), or equal to $k'$ and in $V$. In this case $LOC[SORT^{-1}_k]$ increases by the number of comparisons done in lines 12–15.
- $k$ is $U[1]$ or $V[1]$, and $V[1]$ is found to be smaller than $U[1]$. Then the new $LOC[SORT^{-1}_{V[1]}]$ is the old $LOC[SORT^{-1}_{U[1]}]$, while the new $LOC[SORT^{-1}_{U[1]}]$ is $c$ (the partition to be deleted) plus the number of comparisons done in lines 12–15. Note that the old $LOC[SORT^{-1}_{V[1]}] = c$, so this case is the result $U[1]$ or $V[1]$ swapping $LOC[SORT^{-1}]$, after which we are in the previous case.
Table 1

Merge procedure

/*INITIALIZATION*/
1 Vguess ← c
2 Uguess ← c
3 curr ← i
4 Vnext ← 1
5 Unext ← 1
6 /*MAIN LOOP*/
8 while Vnext ≤ Vmax and Unext ≤ Umax do
9 if Vguess < Uguess then set ‘U is next’
10 if Vguess > Uguess then set ‘V is next’
11 if Vguess = Uguess then begin
12 start comparing the active-arrays of V[Vnext] and U[Unext] from
13 position Uguess + 1 upwards, increasing both Uguess and Vguess until
14 a difference is found, this way determining if ‘U is next’ or ‘V is next’,
15 or, if the active-arrays are equal, set ‘U is next’ and Vguess ← m + 1
16 end
17 if ‘U is next’ then begin
18 SORT[curr] ← U[Unext]
19 Unext ← Unext + 1
20 Uguess ← ULOC[Unext]
21 end
22 if ‘V is next’ then begin
23 SORT[curr] ← V[Vnext]
24 Vnext ← Vnext + 1
25 Vguess ← VLOC[Vnext]
26 end
27 curr ← curr + 1
28 LOC[curr] ← max(Vguess, Uguess)
29 endwhile
30 /*COPY THE REMAINING*/
31 if Vnext > Vmax
32 for k ← curr to Umax + Vmax
33 SORT[k] ← U[Unext]
34 LOC[k] ← ULOC[Unext]
35 Unext ← Unext + 1
36 endfor
37 if Unext > Umax
38 for k ← curr to Umax + Vmax
39 SORT[k] ← V[Vnext]
40 LOC[k] ← VLOC[Vnext]
41 Vnext ← Vnext + 1
42 endfor

We see that in both cases, the actual time is at most the decrease in the potential function plus the amortized time of O(n). □

Acknowledgement

We thank Alexander Zelikovsky for suggesting this problem.

References