Improving the Input Operation for the Set-Covering Problem

Joel Phinney, Ryan Knuesel, Qi Yang
Department of Computer Science and Software Engineering
University of Wisconsin at Platteville

Abstract

Our previous solution to the Set Cover problem works correctly, but was determined to be too slow in certain aspects of it's operation. The original implementation used an unmodified binary search tree that can create, in worst case scenarios, linked list performance. In an attempt to increase the performance of the algorithm, we created two optimized tree data structures to replace the original tree. The optimized trees were faster than the unmodified tree, but the performance wasn't consistent in all cases. To make it more consistent, preparing the data into a tree before loading the matrix was removed from the process. Removing the tree process ensured consistency of the algorithm but decreased performance dramatically. The sorting and matrix insertion algorithms are being investigated for ways to improve performance.

Joel Phinney, Ryan Knuesel, Qi Yang
Department of Computer Science and Software Engineering
University of Wisconsin at Platteville
1 University Plaza
Platteville, WI 53818
yangq@uwplatt.edu

Having redundant data in an application can make it consume more than appropriate resources as well as display less unique data to a user at one time. Detecting that redundant data to display only unique data to users simply adds more overhead every time we want to display it. With so many different companies creating and publishing so many different kinds of redundant data, applications that take advantage of more than one service at a time are faced with processing redundant data. Some consumer-targeted applications that fall along these lines are news aggregators and meta-search engines. The need to remove redundant data is an application of the set-cover problem. Using an efficient algorithm that solves the set-covering problem, this redundancy can be minimized. The set cover problem has been proved to be NP hard, which means that with our current technology we'll never implement a practical solution that runs fast enough to give us minimal covers. The set cover algorithm trades off approximate covers for performance. Previous work has been done on a set-covering algorithm to improve it from O(M * N * min(M, N)) to O(M * N). Since then, further optimizations have been made to this redundancy-minimizing process.

The set-covering problem is interesting and has an application in some programs. Given N sets, let X be the union of all the sets. A cover of X is a group of sets from the N sets such that every element of X belongs to a set in the group. The set-covering problem is to find the smallest cover for those N sets. See the following example.

	a	b	c	d	e	f
S1	0	1	1	0	1	0
S2	0	0	1	1	0	0
S3	1	1	0	1	0	1
S4	0	1	0	0	1	1

Table 1: Set Cover example data.

In this example, the number of sets is 4. The number of elements is 6. {S1, S2, S3} is a cover, but {S1, S2, S4} isn't, because it doesn't cover element a. The minimal cover is {S1, S3}, because it covers all elements of the set. This problem is directly related to reducing data redundancy in meta-search engines, to give one example. The set-covering problem is a well-known mathematical problem, but is largely ignored because very few real-world applications of computers require the use of it. Certain programs, however, need a very efficient and effective solution to this problem.

An early solution to the set-covering problem was a greedy algorithm from [1]. The greedy algorithm ran with O(M * N * min(M, N)), where N is the number of sets and M is the number of elements in the union of all N sets. Later, the a check-and-remove

CAR algorithm was unable to find small cover sizes. By using some components of the greedy and CAR algorithms, a list-and-remove (LAR) algorithm was developed and it created small enough covers with O(M * N).

1.1 Analyzing the Input Methods

While testing these algorithms, certain anomalous tests ran slower than expected. Timing certain parts of the covering process revealed that the data-input method was quite slow compared to the rest of the process. Initially, a standard binary search tree (BST) was being used to store each element that was read, and a bit array in each node recorded the ids of sets that element belonged to. The contents of the bit arrays was then used to generate a sparse matrix of sets and elements. This matrix was then used in the LAR and CAR algorithms.

We used a standard BST for organizing input, so performance was between a balanced tree (O(log(N))) and a linked list (O(N)). Because a balanced tree would provide the best performance a tree could yield, self-balancing trees were tested. Two self-balancing BSTs were considered: the red-black and the AVL trees. The red-black tree we implemented was a red-black tree that assumed null nodes were black. The red-black tree outperformed the standard BST greatly. The AVL tree was a custom iterative implementation, and performed better than the red-black tree. Both trees greatly reduced the effect of linked-list data organization situations.

Further analysis of the redundancy-minimizing process revealed that a BST could be eliminated, if some changes were made to the sparse linked-list matrix of sets and elements. By sorting the elements of each set as they're read, the elements could be inserted in the matrix without the need of a BST. This was then implemented, using an efficient heapsort algorithm to sort the elements. Our initial tests with this implementation have performed even slower than the original BST approach.

1.2 Results of Tests

Choosing one input method that was the best was more difficult because each method performed better in different situations. The results for follow:

Elements in Set	Standard BST Time	AVL Tree Time	Red-Black Tree Time	Hybrid-Matrix Time
5,000	0.87	0.99	0.83	1.95
10,000	1.10	1.24	1.28	2.28

40,000	1.78	1.85	2.00	5.59
80,000	2.09	2.22	2.30	10.17
120,000	2.22	2.37	2.48	14.51
160,000	2.43	2.57	2.67	18.32
200,000	2.49	2.74	2.81	21.74

Table 2: Speed results for different input algorithms with normalized input.

Elements in Set	Standard BST Time	AVL Tree Time	Red-Black Tree Time	Hybrid-Matrix Time
50,000	1.88	1.92	1.97	6.37
80,000	1.97	2.19	2.19	8.36
90,000	2.03	2.37	2.25	10.23
120,000	2.28	2.51	2.53	11.51
200,000	2.27	2.62	2.52	15.66
500,000	15.64	9.40	18.08	18.42
800,000	2.07	7.97	2.41	10.64
1,000,000	157.31	28.71	187.14	5.58

Table 3: Results for different input algorithms with unpredictable input.

Our diverse tests have proved that certain solutions create cover sets much more quickly in certain input data cases. For example, the standard BST worked really well with a random distribution of data, but performed incredibly inefficiently when ran with completely sorted data. Using this information, it is clear to see that the red-black tree performs inadequately and should not be used to process data. It can also be seen that the matrix-insertion method only really works when the input data is already completely sorted, as this emulates the process that happens after the trees unravel.

An analysis of the ideal process for inserting into the modified matrix revealed that this process is of a larger order of magnitude than that of a BST insertion. Because of this, we decided that the original implementation with an AVL tree is the best solution to the problem.

[1] T. H. Cormen, C.E. Leiserson, R. L. Rivest. “Introduction to Algorithms”. The MIT Press, 1991.
[2] Q. Yang, J. McPeek, A. Nofsinger. “Efficient and Effective Practical Algorithms for the Set-Covering Problem”.