Disjoint-Set Data Structures

Saketh Are

April 08, 2011

1 Introduction

Disjoint-set data structures, sometimes referred to as Union-Find or Merge-Find sets, are data structures used to store and keep track of a set of elements partitioned into disjoint (nonoverlapping) subsets. We will be interested in implementing three basic operations:

1. **MakeSet(x)** - Create a new set containing the element x.
2. **Find(x)** - Find and return the set of elements containing x.
3. **Merge(A,B)** - Merge the sets A and B.

Generally, each set is assigned a representative element in order to allow these functions to be well defined. Find(x) returns the representative of the set x belongs to, and Merge(A, B) takes the representatives of A and B as its arguments.

2 Linked Lists

A first attempt at implementing a union-find data structure might involve storing each set as a linked list.

1. **MakeSet(x)** - Simply append a new linked list to our array of sets. O(1)
2. **Find(x)** - Perform a linear search through all elements. O(N)
3. **Merge(A,B)** - Append list B to list A. O(1)

Alternatively, we can have each element store a pointer to its representative, reducing the runtime complexity of Find to O(1). However, we would need O(N) operations to perform a Merge, since the pointers associated with every element in B would need to be updated.

An improvement that can be made on the Merge operation is to always append the smaller set to the larger, resulting in fewer updates of representative pointers. For an n-element linked-list structure with O(1) Find and O(N)

Merge, a sequence of m operations performed using this optimization will take O(m+nlogn) time. To achieve faster performance, however, we will have to turn to a different data structure.

3 Disjoint-Set Forests

In a disjoint-set forest, each set is stored as a tree. Every node holds a pointer to its parent node, with the exception of root nodes. The root of a set's tree is used as its representative.

1. **MakeSet(x)** - Create a new node. O(1)
2. **Find(x)** - Starting at x, traverse parent pointers up to the root. O(N)
3. **Merge(A,B)** - Set root B's parent pointer to root A. O(1)

A naive implementation, as outlined above, does not actually produce better results than the linked-list approach. However, we can make huge improvements on the tree-based structure. The biggest problem we face with the naive implementation is the emergence of highly unbalanced trees due to the simplistic nature of the Merge operation. We can employ an optimization called *merge by rank* to address this issue. It is analogous to the appending of the smaller set that we used with linked lists.

3.1 Merge By Rank

We begin by defining the rank of a one-element tree to be 0. Since all larger sets are produced by merging smaller sets, we can now define all higher ranks recursively.

Suppose we are merging two sets, one of rank p and the other of rank q. If $p{=}q$, then the rank of the union of these sets will be $p{+}1$ (or $q{+}1$). Otherwise, assume without loss of generality that $p{>}q$. The lower-ranked set is attached to the root of the higher-ranked set, and the overall rank is p.

Using the technique of *merge by rank* alone to improve disjoint-set forest performance produces an amortized run-time of O(logn) per operation.

3.2 Path Compression

Now that we've improved our Merge operation, let's reexamine Find. We've already seen that lower-ranked, flatter trees produce much faster runtimes due to having less levels to traverse through to reach the representative element (root). Thus, once we know the identity of a certain node's representative element, connecting it directly to the root will drastically reduce runtime for future operations. In fact, we can relink every node visited in our upward traversal of the tree directly to the root.

3.3 Runtime Complexity

The merge by rank and path compression optimizations complement each other incredibly well. With both implemented, the amortized runtime for operations on our data structure becomes $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the extremely quickly-growing Ackermann function. As such, the inverse grows very very slowly and our asymptotic runtime complexity is quite an improvement over the linear time approach we started with.

In fact, $\alpha(n)$ grows so slowly that runtimes are effectively constant for all remotely practical values of n. To give you a sense of just how slowly this function grows, consider the following values:

$$\alpha(1) = 1$$

$$\alpha(6.121 \cdot 10^{35163}) \approx 5$$

4 Usage Examples

1. Keeping track of connectivity on an undirected graph

   Is there a path between these two nodes?

   Would adding an edge between these two nodes produce a cycle?

2. Separate an undirected graph into connected components

3. Kruskal's algorithm for finding minimal spanning trees