Disjoint-set data structure

From Wikipedia, the free encyclopedia

In computer science, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (nonoverlapping) subsets. It supports two useful operations:

- *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.

MakeSet creates 8 singletons.

After some operations of *Union*, some sets are grouped together.

The other important operation, *MakeSet*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved (see the *Applications* section).

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*(x) returns the representative of the set that *x* belongs to, and *Union* takes two set representatives as its arguments.

Contents

Disjoint-set linked lists

A simple disjoint-set data structure uses a linked list for each set. The element at the head of each list is chosen as its representative.

MakeSet creates a list of one element. *Union* appends the two lists, a constant-time operation if the list carries a pointer to its tail. The drawback of this implementation is that *Find* requires O(*n*) or linear time to traverse the list backwards from a given element to the head of the list.

This can be avoided by including in each linked list node a pointer to the head of the list; then *Find* takes constant time, since this pointer refers directly to the set representative. However, *Union* now has to update each element of the list being appended to make it point to the head of the new combined list, requiring O(*n*) time.

When the length of each list is tracked, the required time can be improved by always appending the smaller list to the longer. Using this *weighted-union heuristic*, a sequence of *m MakeSet*, *Union*, and *Find* operations on *n* elements requires O($m + n\log n$) time.[1] For asymptotically faster operations, a different data structure is needed.

Analysis of the naive approach

We now explain the bound $O(n \log(n))$ above.

Suppose you have a collection of lists and each node of each list contains an object, the name of the list to which it belongs, and the number of elements in that list. Also assume that the total number of elements in all lists is n (i.e. there are n elements overall). We wish to be able to merge any two of these lists, and update all of their nodes so that they still contain the name of the list to which they belong. The rule for merging the lists A and B is that if A is larger than B then merge the elements of B into A and update the elements that used to belong to B, and vice versa.

Choose an arbitrary element of list L, say x. We wish to count how many times in the worst case will x need to have the name of the list to which it belongs updated. The element x will only have its name updated when the list it belongs to is merged with another list of the same size or of greater size. Each time that happens, the size of the list to which x belongs at least doubles. So finally, the question is "how many times can a number double before it is the size of n?" (then the list containing x will contain all n elements). The answer is exactly $\log_2(n)$. So for any given element of any given list in the structure described, it will need to be updated $\log_2(n)$ times in the worst case. Therefore, updating a list of n elements stored in this way takes $O(n \log(n))$ time in the worst case. A find operation can be done in $O(1)$ for this structure because each node contains the name of the list to which it belongs.

A similar argument holds for merging the trees in the data structures discussed below. Additionally, it helps explain the time analysis of some operations in the binomial heap and Fibonacci heap data structures.

Disjoint-set forests

Disjoint-set forests are data structures where each set is represented by a tree data structure, in which each node holds a reference to its parent node (see parent pointer tree). They were first described by Bernard A. Galler and Michael J. Fischer in 1964,[2] although their precise analysis took years.

In a disjoint-set forest, the representative of each set is the root of that set's tree. *Find* follows parent nodes until it reaches the root. *Union* combines two trees into one by attaching the root of one to the root of the other.

Implementation

Naive

One way of implementing these might be:

```
function MakeSet(x)
    x.parent := x
```

```
function Find(x)
    if x.parent == x
       return x
    else
       return Find(x.parent)
```

```
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

In this naive form, this approach is no better than the linked-list approach, because the tree it creates can be highly unbalanced.

Union by rank

The previous implementation can be enhanced in two ways.

The first way, called *union by rank*, is to always attach the smaller tree to the root of the larger tree. Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. In the context of this algorithm, the term *rank* is used instead of *depth* since it stops being equal to the depth if path compression (described below) is also used. One-element trees are defined to have a rank of zero, and whenever two trees of the same rank r are united, the rank of the result is $r+1$. Just applying this technique alone yields a worst-case running-time of $O(\log n)$ for the *Union* or *Find* operation. Pseudocode for the improved `MakeSet` and `Union`:

```
function MakeSet(x)
    x.parent := x
    x.rank   := 0
```

```
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    // if x and y are already in the same set (i.e., have the same root or representative)
    if xRoot == yRoot
        return

    // x and y are not in same set, so we merge them
    if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
    else if xRoot.rank > yRoot.rank
        yRoot.parent := xRoot
    else
        yRoot.parent := xRoot
        xRoot.rank := xRoot.rank + 1
```

Path compression

The second improvement, called *path compression*, is a way of flattening the structure of the tree whenever *Find* is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as `Find` recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. Here is the improved `Find`:

```
function Find(x)
    if x.parent != x
       x.parent := Find(x.parent)
    return x.parent
```

These two techniques complement each other; applied together, the amortized time per operation is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the function $n = f(x) = A(x, x)$, and A is the extremely fast-growing Ackermann function. Since $\alpha(n)$ is the inverse of this function, $\alpha(n)$ is less than 5 for all remotely practical values of n. Thus, the amortized running time per operation is effectively a small constant.

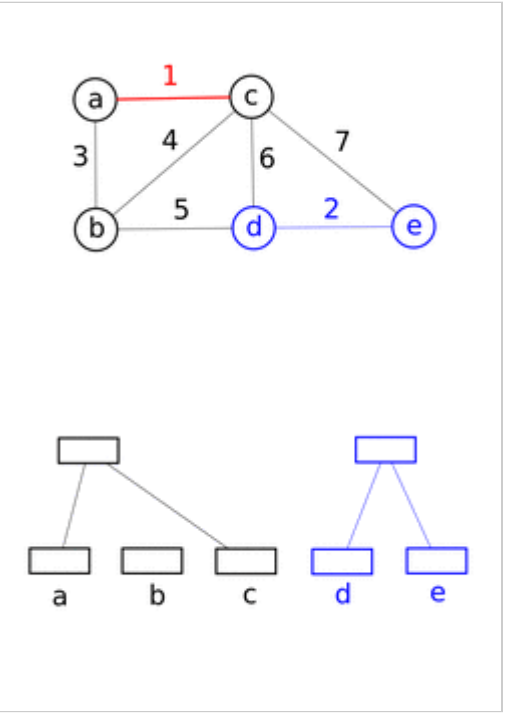
In fact, this is asymptotically optimal: Fredman and Saks showed in 1989 that $\Omega(\alpha(n))$ words must be accessed by *any* disjoint-set data structure per operation on average.[3]

Applications

Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle. The Union–Find algorithm is used in high-performance implementations of unification.[4]

This data structure is used by the Boost Graph Library to implement its Incremental Connected Components (http://www.boost.org/libs/graph/doc/incremental_components.html) functionality. It is also used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

Note that the implementation as disjoint-set forests doesn't allow deletion of edges—even without path compression or the rank heuristic.



A demo for Union-Find when using Kruskal's algorithm to find minimum spanning tree.

History

While the ideas used in disjoint-set forests have long been familiar, Robert Tarjan was the first to prove the upper bound (and a restricted version of the lower bound) in terms of the inverse Ackermann function, in 1975.[5] Until this time the best bound on the time per operation, proven by Hopcroft and Ullman,[6] was $O(\log^* n)$, the iterated logarithm of n, another slowly growing function (but not quite as slow as the inverse Ackermann function).

Tarjan and Van Leeuwen also developed one-pass *Find* algorithms that are more efficient in practice while retaining the same worst-case complexity.[7]

In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a persistent version of the disjoint-set forest data structure, allowing previous versions of the structure to be efficiently retained, and formalized its correctness using the proof assistant Coq.[8] However, the implementation is only asymptotic if used ephemerally or if the same version of the structure is repeatedly used with limited backtracking.

See also

- Partition refinement, a different data structure for maintaining disjoint sets, with updates that split sets apart rather than merging them together
- Dynamic connectivity

References