Disjoint-set data structure

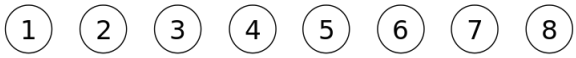
In computer science, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It provides near-constant-time operations (bounded by the inverse Ackermann function) to add new sets, to merge existing sets, and to determine whether elements are in the same set. In addition to many other uses (see the Applications section), disjoint-sets play a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph.

Disjoint-set/Union-find Forest	
Type	multiway tree
Invented	1964
Invented by	Bernard A. Galler and Michael J. Fischer

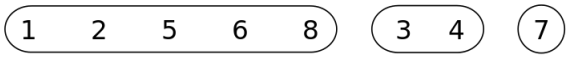
Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$[1]	$O(n)$[1]
Search	$O(\alpha(n))$[1]	$O(\alpha(n))$[1]
Merge	$O(\alpha(n))$[1]	$O(\alpha(n))$[1]

Contents



MakeSet creates 8 singletons.



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

History

Disjoint-set forests were first described by Bernard A. Galler and Michael J. Fischer in 1964.[2] In 1973, their time complexity was bounded to $O(\log^*(n))$, the iterated logarithm of n, by Hopcroft and Ullman.[3] (A proof is available here.) In 1975, Robert Tarjan was the first to prove the $O(\alpha(n))$ (inverse Ackermann function) upper bound on the algorithm's time complexity,[4] and, in 1979, showed that this was the lower bound for a restricted case.[5] In 1989, Fredman and Saks showed that $\Omega(\alpha(n))$ (amortized) words must be accessed by *any* disjoint-set data structure per operation,[6], thereby proving the optimality of the data structure.

In 1991, Galil and Italiano published a survey of data structures for disjoint-sets.[7]

In 1994, Richard J. Anderson and Heather Woll described a parallelized version of Union–Find that never needs to block.[8]

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.[9] However, the implementation is only asymptotic if used ephemerally or if the same version of the structure is repeatedly used with limited backtracking.

Representation

A disjoint-set forest consists of a number of elements each of which stores an id, a parent pointer, and, in efficient algorithms, a value called the "rank".

The parent pointers of elements are arranged to form one or more trees, each representing a set. If an element's parent pointer points to no other element, then the element is the root of a tree and is the representative member of its set. A set may consist of only a single element. However, if the element has a parent, the element is part of whatever set is identified by following the chain of parents upwards until a representative element (one without a parent) is reached at the root of the tree.

Forests can be represented compactly in memory as arrays in which parents are indicated by their array index.

Operations

MakeSet

The *MakeSet* operation makes a new set by creating a new element with a unique id, a rank of 0, and a parent pointer to itself. The parent pointer to itself indicates that the element is the representative member of its own set.

The *MakeSet* operation has $O(1)$ time complexity.

Pseudocode:

```
function MakeSet(x)
  if x is not already present:
    add x to the disjoint-set tree
    x.parent := x
    x.rank   := 0
```

Find

Find(x) follows the chain of parent pointers from *x* upwards through the tree until an element is reached whose parent is itself. This element is the root of the tree and is the representative member of the set to which *x* belongs, and may be *x* itself.

Path compression, is a way of flattening the structure of the tree whenever *Find* is used on it. Since each element visited on the way to a root is part of the same set, all of these visited elements can be reattached directly to the root. The resulting tree is much flatter, speeding up future operations not only on these elements, but also on those referencing them.

Pseudocode:

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

Tarjan and Van Leeuwen also developed one-pass *Find* algorithms that are more efficient in practice while retaining the same worst-case complexity.[4]

Union

Union(x,y) uses *Find* to determine the roots of the trees *x* and *y* belong to. If the roots are distinct, the trees are combined by attaching the root of one to the root of the other. If this is done naively, such as by always making *x* a child of *y*, the height of the trees can grow as $O(n)$. To prevent this *union by rank* is used.

Union by rank always attaches the shorter tree to the root of the taller tree. Thus, the resulting tree is no taller than the originals unless they were of equal height, in which case the resulting tree is taller by one node.

To implement *union by rank*, each element is associated with a rank. Initially a set has one element and a rank of zero. If two sets are unioned and have the same rank, the resulting set's rank is one larger; otherwise, if two sets are unioned and have different ranks, the resulting set's rank is the larger of the two. Ranks are used instead of height or depth because path compression will change the trees' heights over time.

Pseudocode:

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

  // x and y are already in the same set
  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
    //Arbitrarily make one root the new parent
    yRoot.parent := xRoot
    xRoot.rank   := xRoot.rank + 1
```

Time Complexity

There are four scenarios to consider, namely implementations which use: *path compression*, *union by rank*, both, or neither.

If neither heuristic is used, the height of trees can grow unchecked as $O(n)$, which implies that *Find* and *Union* operations will take $O(n)$ time.

Using *path compression* alone gives a worst-case running time of $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$,[10] for a sequence of n *MakeSet* operations (and hence at most $n - 1$ *Union* operations) and f *Find* operations.

Using *union by rank* alone gives a running-time of $O(m \log_2 n)$ (tight bound) for m operations of any sort of which n are *MakeSet* operations.[10]

Using both *path compression* and *union by rank* ensures that the amortized time per operation is only $O(\alpha(n))$,[4][5] which is optimal,[6] where $\alpha(n)$ is the inverse Ackermann function. This function has a value $\alpha(n) < 5$ for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.

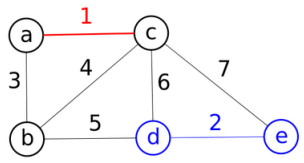
Applications

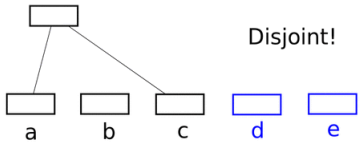
Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle. The Union–Find algorithm is used in high-performance implementations of unification.[11]

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 a key component in implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

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

Sharir and Agarwal report connections between the worst-case behavior of disjoint-sets and the length of Davenport–Schinzel sequences, a combinatorial structure from computational geometry.[12]





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

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

1. Tarjan, Robert Endre (1975). "Efficiency of a Good But Not Linear Set Union Algorithm" (http://portal.acm.org/citation.cfm?id=321884). *Journal of the ACM*. **22** (2): 215–225. doi:10.1145/321879.321884 (https://doi.org/10.1145%2F321879.321884).
2. Galler, Bernard A.; Fischer, Michael J. (May 1964). *An improved equivalence algorithm* (http://portal.acm.org/citation.cfm?doid=364099.364331). *Communications of the ACM*. **7**. pp. 301–303. doi:10.1145/364099.364331 (https://doi.org/10.1145%2F364099.364331).. The paper originating disjoint-set forests.
3. Hopcroft, J. E.; Ullman, J. D. (1973). "Set Merging Algorithms". *SIAM Journal on Computing*. **2** (4): 294–303. doi:10.1137/0202024 (https://doi.org/10.1137%2F0202024).
4. Tarjan, Robert E.; van Leeuwen, Jan (1984). "Worst-case analysis of set union algorithms". *Journal of the ACM*. **31** (2): 245–281. doi:10.1145/62.2160 (https://doi.org/10.1145%2F62.2160).
5. Tarjan, Robert Endre (1979). "A class of algorithms which require non-linear time to maintain disjoint sets". *Journal of Computer and System Sciences*. **18**: 110–127.
6. Fredman, M.; Saks, M. (May 1989). "The cell probe complexity of dynamic data structures". *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*: 345–354. "Theorem 5: Any CPROBE(log *n*) implementation of the set union problem requires Ω(*m* α(*m*, *n*)) time to execute *m* Find's and *n*–1 Union's, beginning with *n* singleton sets."
7. Galil, Z.; Italiano, G. (1991). "Data structures and algorithms for disjoint set union problems". *ACM Computing Surveys*. **23**: 319–344.
8. Anderson, Richard J.; Woll, Heather (1994). *Wait-free Parallel Algorithms for the Union-Find Problem*. 23rd ACM Symposium on Theory of Computing. pp. 370–380.
9. Conchon, Sylvain; Filliâtre, Jean-Christophe (October 2007). "A Persistent Union-Find Data Structure". *ACM SIGPLAN Workshop on ML* (https://www.lri.fr/~filliatr/puf/). Freiburg, Germany.
10. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). "Chapter 21: Data structures for Disjoint Sets". *Introduction to Algorithms* (Third ed.). MIT Press. pp. 571–572. ISBN 978-0-262-03384-8.

11. Knight, Kevin (1989). "Unification: A multidisciplinary survey" (http://portal.acm.org/citation.cfm?id=62030). *ACM Computing Surveys*. **21**: 93–124. doi:10.1145/62029.62030 (https://doi.org/10.1145%2F62029.62030).
12. Sharir, M.; Agarwal, P. (1995). *Davenport-Schinzel sequences and their geometric applications*. Cambridge University Press.

External links

- C++ implementation (http://www.boost.org/libs/disjoint_sets/disjoint_sets.html), part of the Boost C++ libraries
- A Java implementation with an application to color image segmentation, Statistical Region Merging (SRM), IEEE Trans. Pattern Anal. Mach. Intell. 26(11): 1452–1458 (2004) (http://www.lix.polytechnique.fr/~nielsen/Srmjava.java)
- Java applet: A Graphical Union–Find Implementation (http://www.cs.unm.edu/~rlpm/499/uf.html), by Rory L. P. McGuire
- Python implementation (http://code.activestate.com/recipes/215912-union-find-data-structure/)
- Visual explanation and C# code (http://www.mathblog.dk/disjoint-set-data-structure/)