textit There are many "100 things to do before you die" lists on the web. Implementing Union-Find should be one of those things in a computer scientist's list.

Lecture XIII
DISJOINT SETS

Let U be a set of items, and let $\equiv$ be an equivalence relation on U. We want to make two kinds of requests on this equivalence relation: First, given $x, y \in U$, we want to know if x and y are equivalent, is $x \equiv y$? Second we want to modify the equivalence relation by declaring that, from now on, x and y will be equivalent.

This equivalence maintenance problem is called **union-find problem** because the two requests can be reduced the Find and Union operations. Other names for this problem are **set union**, **disjoint sets** or **set equivalence**. Underlying the best solutions to this problem is a data structure called compressed trees. The algorithms are this data structure are simple and easy to implement. Nevertheless, the complexity analysis of these algorithms is highly non-trivial, requiring a sophisticated form of amortization argument. The remarkable fact is that the analysis (Tarjan 1975) leads to the inverse Ackermann function, an extremely slow growing function.

There are many practical applications of union find. We shall look at three: the implementation of Kruskal's algorithm for minimum spanning tree, a problem of reducing numerical expressions, and the problem of computing Betti numbers in Computational Topology.

§1. Union Find Problem

Let U be a set of items, also called the **universe**. A **partition** of U is a collection

$$P = \{S_1, \ldots, S_k\}$$

of pairwise disjoint non-empty sets such that $U = \cup_{i=1}^{k} S_i$. We say $x, y \in [1..n]$ are **equivalent**, written $x \equiv y$, if they belong to the same set in P. Each $S_i \in P$ is also called an **equivalence class**. For any $x \in U$, let $set(x)$ denote the equivalence class of x. The items in U have no particular properties except that two items can be checked for equality. In particular, there is no ordering property on the items.

In order to facilitate computation, we will assume that each S_i has a unique **representative item** $x_i \in S_i$. Let $rep(S_i)$ denote this representative item x_i. The choice of this representative item is arbitrary. Because of this representative item, we can now define the **Find** operation: $Find(x)$ returns the representative item in the equivalence class of x. Hence the

$$x \equiv y \quad \Longleftrightarrow \quad \mathtt{rep}(x) = \mathtt{rep}(y) \quad \Longleftrightarrow \quad \mathtt{set}(x) = \mathtt{set}(y).$$

The problem of checking if $x, y \in U$ are equivalent is now reduced to computing $Find(x)$ and $Find(y)$ and checking whether $Find(x) = Find(y)$.

In the following, it is convenient to assume that

$$U = [1..n] := \{1, 2, \ldots, n\}.$$

The partition P is dynamically changing because of the Union operation. If $x, y \in U$, then the **Union** operation declares that henceforth x and y are equivalent. This reduces to the problem of replacing $set(x)$ and $set(y)$ in P by their union $set(x) \cup set(y)$. Thus, the number of equivalence classes can only decrease, not increase.

The **Union Find problem** is the problem of processing a sequence of Find/Union requests on a set U which is initially given the **trivial equivalence relation** (i.e., each equivalence class is the singleton set $\{x\}$ where $x \in U$).

¶1. Example. Let $n = 4$ and $U = [1..4]$. Initially, the partition is $P = \{\{1\}, \{2\}, \{3\}, \{4\}\}$. Consider the sequence of requests:

$$Union(2,3), Find(2), Union(1,4), Union(2,1), Find(2).$$

After processing this sequence, $P = \{\{1,2,3,4\}\}$. The two Find requests return (respectively) the representatives of the sets $\{2,3\}$ and $\{1,2,3,4\}$ that exist at the moment when the Finds occur. To be specific, suppose we choose the smallest integer in a set to be the representative item. Then these two Find requests return 2 and 1, respectively.

Each Union (resp., Find) requests can be viewed as an **equivalence assertion** (resp., **equivalence query**). The original motivation for this problem is in FORTRAN compilers, where one needs to process the EQUIVALENCE statement in the FORTRAN language. This statement assert the equivalence of two programming variables. Another application is in finding connected components in bigraphs. For more information, see Tarjan [6, 8] and a survey from Galil and Italiano [3]. Note that it is difficult to extend the union-find technique to allow the "undoing" of unions. One step in this direction is to have a persistent version of the union-find data structure where you can can go back to previous versions of the data structure. See Conchon and Filliâtre (2007).

¶2. Representation of Sets. In most solutions to the Union Find Problem, a set $S \subseteq U$ is concretely represented by a rooted unordered tree T whose node set is S. To represent T, each item x has a **parent pointer** $x.p$. The tree T is completely determined by these pointers. The root is the unique node $x \in S$ with the property $x.p = x$, and it serves as **representative** of S. We call T a **compressed tree** representation of the set S. This data structure is from Galler and Fischer (1964). It is important to realize that there are no pointers from a node of T to any of its children, and the set of children are unordered (unlike, say, in binary search trees).

Using compressed trees, we can now represent any collection P of disjoint sets by a forest of compressed trees. For instance, let $P = \{0, \underline{5}, 6, 8, 9\}, \{\underline{3}\}, \{1, 2, 4, \underline{7}\}$. The underlined items (*i.e.*, 5, 3 and 7) are representatives of the respective sets. A possible compressed tree representation of P is shown in Figure 1.

If x, y are roots of compressed trees, then we have the concrete or "physical" operation

as opposed to ADT operations

$$Link(x, y)$$

that **links** x to y. Basically, this amounts to the assignment $x.p \leftarrow y$. Under the assumption that x, y are roots of compressed trees, the result of linking x to y is a new compressed tree rooted at y, representing the union of the original two sets. If $x = y$, then $Link(x, y)$ is a null operation. Now we can physically implement a Union requests by two Finds and a Link:

$$Union(x, y) \equiv Link(Find(x), Find(y)). \quad (1)$$

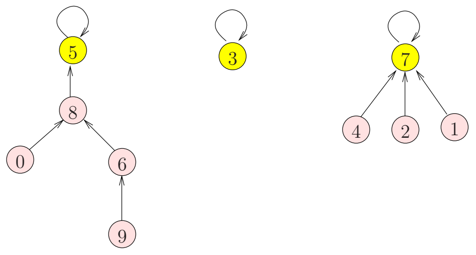


Figure 1: Forest of compressed trees representing $P = \{0, \underline{5}, 6, 8, 9\}, \{\underline{3}\}, \{1, 2, 4, \underline{7}\}$.

¶3. Complexity parameters. The complexity of processing a sequence of Union-Find requests will be given as a function of m and n, where n is the size of the universe $U = [1..n]$ and m the number of requests (i.e., Union or Find operations). But the parameter m in the following discussions will be given a slightly different meaning: recall that the Union operation can be replaced by two finds and a single link as in (1). So we may replace a sequence of m Union/Find requests by an equivalent sequence of $\le 3m$ Link/Find requests. Up to Θ-order, the complexity of a sequence of m Link/Finds and the complexity of a sequence of m Union/Finds are equal. Hence, it suffices to analyze a sequence of m Link/Find operations. In some of our analysis, we use another parameter m' which is the number of Find requests among the m operations; thus $m' \le m$.

Although m and n are arbitrary parameters, for the purposes of lower bounds, we often assume that

$$m \ge n/2. \tag{2}$$

This inequality might be justified by insisting that every item in the universe must be referenced in either a Find or a Union request. But fundamentally, the reason for making this assumption allows us to show that our lower bounds are tight.

¶4. Two Solutions to the Link/Find Problem.

Linked List Solution An obvious solution to the Link/Find problem is to represent each set in P by a singly linked list. We can regard a singly linked list as a special kind of compressed tree T which has only one leaf. The unique leaf of T corresponds to the **head** while the root is the **tail** of this linked list. Following links from any node will lead us to the tail of the list. This is like a traditional singly-linked list where the pointer $x.p$ plays the role of the "next node" pointer. See Figure 2(a). Note that linking takes constant time, *provided* the head of each linked list stores a **tail pointer** that points to its own tail. The tail pointer is easy to maintain during a link operation. On the other hand, a Find operation requires a worst case time proportional to the length of the list. Clearly, the complexity of a sequence of m Link/Find requests is $\mathcal{O}(mn)$.

Traditionally, the tail of the linked list points to `nil`*.*

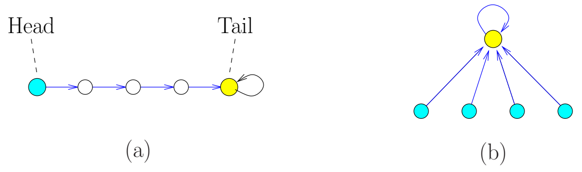


Figure 2: Two special kinds of compressed trees: (a) Linked list, (b) Anti-list.

Anti-list Solution Another solution is to represent each set in P as a compressed tree of height one: thus every non-root is a leaf that points to the root. See Figure 2(b). This data structure is the antithesis of lists: we will call such trees **anti-lists**. Clearly a Find request will now take constant time. On the other hand, linking x to y takes time proportional to the size of $\mathtt{set}(x)$ since we need to make every element in $\mathtt{set}(x)$ point to y. Note that each Find has unit cost, and each Link costs $O(n)$. Since there are at most $n-1$ links, we see that the cost of sequence of m operations costs

$$O(m+n^2). \tag{3}$$

For a lower bound, we can make the ith operation link a set of size i to another set, costing i units. Therefore a sequence of $\min\{m,n\}$ links would cost $(\min\{m,n\})^2$. Thus proves that the worst case complexity of processing m Link/Find requests is

$$\Omega(m+(\min\{m,n\})^2).$$

Assuming (2), this lower bound matches the upper bound in (3).

¶5. Naive Compressed Tree Solution. The constrast between the list and anti-list representations is as sharp as can be: Linking is easy for lists but hard for anti-lists. Conversely, Find is easy using anti-lists, but hard for lists. Here, "easy" means $\Theta(1)$ and "hard" means $\Theta(n)$. Galler and Fischer (1964) shows that the relative advantages of both solutions can be exploited if we use the general compressed tree representation. But this advantage may not be immediately apparent. To see why, consider a straight forward implementation of the Find and Link operations under compressed tree representation. For $Find(x)$, we just follow parent pointers from x until the root, which is returned. For $Link(x,y)$, we simply set $x.p \leftarrow y$ and return y. These "naive" algorithms can lead to a degenerate tree that is a linear list of length $n-1$. Clearly $\Theta(mn)$ is the worst case bound for a sequence of m Link/Find Operations on n items. How can we do better?

Exercises

Exercise 1.1: Show the stated upper bounds for the above data structures are tight by demonstrating matching lower bounds (assume (2)).
(a) When using compressed trees or linked lists, give an $\Omega(mn)$ lower bound.
(b) When using anti-lists, give an $\Omega(m+n^2)$ lower bound. ♢

Exercise 1.2: Assume the anti-list representation, let us link two trees using the following "size heuristic": always append the smaller set to the larger set. To implement this heuristic, we can easily keep track of the sizes of anti-lists. Show that this scheme achieves $\mathcal{O}(m+n\log n)$ complexity. Prove a corresponding lower bound. ♢

Exercise 1.3: We propose another data structure: assume that the total number n of items in all the sets is known in advance. We represent each set (*i.e.*, equivalence class) by a tree of depth exactly 2 where the set of leaves correspond bijectively to items of the set. Each node in the tree and keeps track of its degree (= number of children) and maintains three pointers: parent, child and sibling. Thus there is a singly-linked **sibling list** for the set of children of a node u; the child pointer of u points to the head of this list. The following properties hold:
(i) Each child of the root has degree between 1 and $2\lg n$.

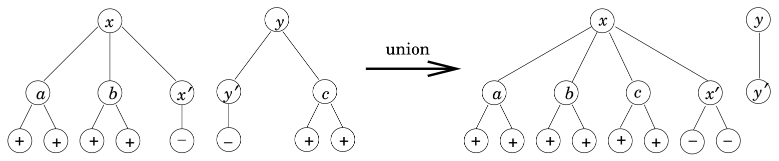


Figure 3: Illustration of Union.

(ii) Say a child of the root is **full** if it has degree at least $\lg n$. At most one child of the root is **not full**.

Using this representation, a Find operation takes constant time. The union of two sets with roots x, y is done as follows: if $degree(x) \geq degree(y)$ then each full child of y becomes a child of x. Say x', y' are the children of x, y (respectively) which are not full and assume that $degree(x') \geq degree(y')$. [We leave it to the reader to consider the cases where x' or y' does not exist, or when $degree(x') < degree(y')$.] Then we make each child of y' into a child of x'. Note that since y and y' do not represent items in the sets, they can be discarded after the union. Prove that the complexity of this solution is

$$\mathcal{O}(m + n \log\log n)$$

where m is the total number of operations.
HINT: Devise a charging scheme so that the work in a Union operation is charged to items at depths 1 and 2, and the overall charges at depths 1 and 2 are (respectively) $\mathcal{O}(n)$ and $\mathcal{O}(n \log\log n)$. ♢

Exercise 1.4: Suppose we add to the Union-Find problem a third operation: **break**. Interpret the union operation as adding an edge to a bigraph that initially has no edges. The break operation is just the reverse of union, as it removes an edge (i.e., breaks a link). The Find(x) operation should again return a representative node of the connected component of the current bigraph that contains x. How efficiently can you solve this problem? ♢

End Exercises

§2. Rank and Path Compaction Heuristics

There are several heuristics for improving the performance of the naive compressed tree algorithms. They fall under two classes, depending on whether they seek to improve the performance of Links or of Finds.

¶6. Size and Rank heuristics. We now improve the overall performance of our data structure by restrictions on the way linking is done. For any node x, let $size(x)$ be the number of items in the subtree rooted at x. Suppose we keep track of the size of each node, and in the union of x and y, we link the root of smaller size to the root of larger size (if the sizes are equal, this is arbitrary). This rule for linking is called the **size heuristic**. Then it is easy to see that our compressed trees have depth at most $\lg n$ under this heuristic. Hence Find operations take $\mathcal{O}(\log n)$ time.

An improvement on the size heuristic was suggested by Tarjan and van Leeuwen: we keep track of a simpler number called $rank(x)$ that can be viewed as a lower bound on $\lg(size(x))$ (see next lemma). The **rank** of x is initialized to 0 and subsequently, whenever we link x to y, we will modify the rank of y as follows:

$$rank(y) \leftarrow \max\{rank(y), rank(x)+1\} \tag{4}$$

This assignment is the only way by which a rank changes, and this change cannot decrease the rank of an item. Thus the rank of an item is non-decreasing over time. If compressed trees are never modified except through linking, then it is easy to see that $rank(x)$ is simply the height of x. But we shall see that, in general, $rank(x)$ is just an upper bound on the height of x. Using the rank information, we specify a linking order for union:

Rank Heuristic: *When forming the union of two trees rooted at x and y, respectively, link x to y if $rank(x) \le rank(y)$; otherwise link y to x.*

Under this heuristic, the assignment (4) increases the rank of y by at most 1.

¶7. Path Compaction heuristics. The size and rank heuristics are applicable to Links. Now consider heuristics to improve the performance of Finds. The first idea was introduced by McIlroy and Morris. When doing a Find on an item x, we will traverse the path from x to the root of its tree. This is called the **find-path** of x. We specify a transformation of the compressed tree to accompany each Find operation:

Path Compression Heuristic: *After performing a Find on x that returns the root u, modify the parent pointer of each node z along the find-path of x to point to u.*

For example, if the find-path of x is (x, w, v, u) as in Figure 4, then after Find(x), the path compression heuristic will make x and w into children of u (note that v remains a child of u).

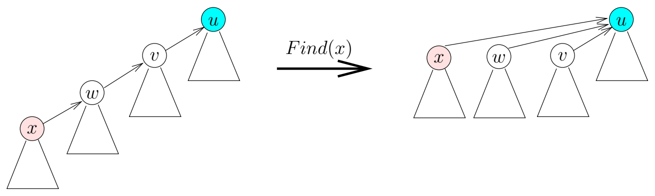


Figure 4: Path compression heuristic.

Path compression requires two passes along the find-path, first to locate u and the second time to change the parent pointer of nodes along the path. Can we do better? Path compression can be seen as a member of the family of **path compaction heuristics**. Such a heuristic specifies a rule to modify the parent pointer of each node z along a find-path to point to some ancestor of z. Note that if z is the root or a child of the root, then the path compaction heuristic has no effect on z (or $z.p$ to be precise).

The "trivial" path compaction heuristic is the one that never changes $z.p$ for any z on the path. Path compression is the other extreme, where each $z.p$ is made to point to the ultimate

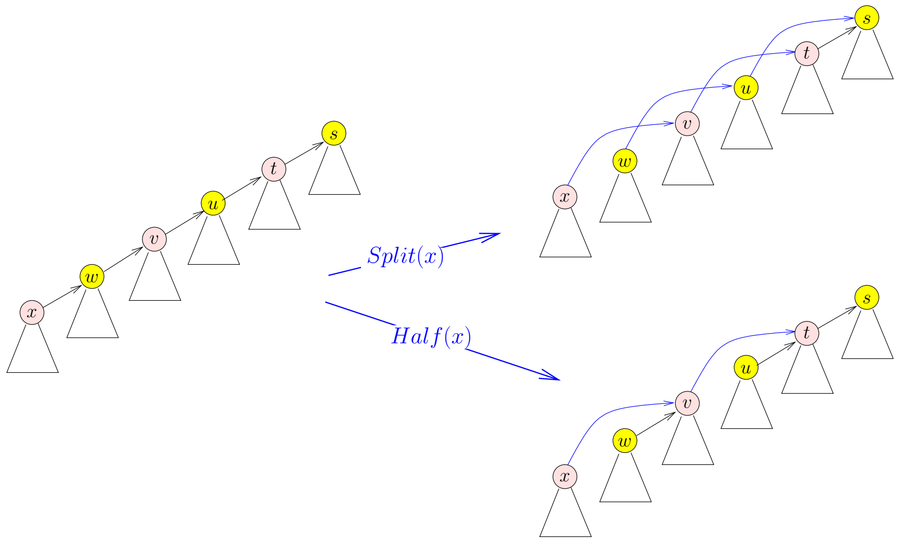


Figure 5: Path Compaction: Splitting and Halving

ancestor, the root. We now consider two intermediate forms of path compaction, from van der Weide and van Leeuwen (1977). The **path splitting heuristic** says that for each node z along the find-path, we should update its parent pointer to its grandparent: $z.p \leftarrow (z.p).p$. Of course, $(z.p).p$ is the grandparent of z, and it may also be written as

$$z.p.p \quad \text{or} \quad z.p^2.$$

The effect of this heuristic is to split the find-path into two paths comprising the odd and even nodes, respectively. Both the odd and even paths are about half the original length. See Figure 5(a) where the find-path $(x\text{—}w\text{—}v\text{—}u\text{—}t\text{—}s)$ is split into two paths $(x\text{—}v\text{—}t\text{—}s)$ and $(w\text{—}u\text{—}s)$. The path splitting heuristic can now be implemented with only one pass over the find-path, and in this sense is an improvement over path compression. A further improvement is the following variant:

Path Halving Heuristic: *After performing a Find on x, modify the parent pointer of every other node z along the find-path of x to point its grandparent.*

In Figure 5(b), we see that the find-path $(x\text{—}w\text{—}v\text{—}u\text{—}t\text{—}s)$ is halved into the path $(x\text{—}v\text{—}t\text{—}s)$. The other node w and u join this half path at intermediate points. This halving heuristic requires only half the number of pointer assignments used in the path splitting heuristic.

Although the splitting and halving heuristics have the advantage of being 1-pass algorithms, they do not instantly reduce the find-path of elements along its path to length $O(1)$ like in path compression. So it is not immediately clear these two heuristics can match the performance of path compression.

¶8. Analysis of the rank heuristic. We state some simple properties of the rank function. Notice that the rank function is defined according to equation (4), whether or not we use the rank heuristic for linking:

Lemma 1.
The rank function (whether or not the rank heuristic is used, and even in the presence of path compression) has these properties:
(i) A node has rank 0 *iff it is a leaf.*
(ii) The rank of a node x *does not change after* x *has been linked to another node.*
(iii) Along any find-path, the rank is strictly increasing.
If the rank heuristic is used, then the rank function has additional properties:
(a) $rank(x) \le degree(x)$ *and* $rank(x) \le \lg(size(x))$*.*
(b) No path has length more than $\lg n$*.*
(c) In any sequence of Links/Find requests, for any $k \ge 1$*, the number of times that the rank of any item gets promoted from* $k-1$ *to* k *is at most* $n2^{-k}$*.*

Proof. Parts(i)-(iii) are immediate. Part(a) is shown by induction: it is true when x is initially a singleton, since $rank(x) = degree(x) = 0$ and $size(x) = 1$. We must look at events that causes the rank, degree or size of any node x to change. When we link y to x, the degree of x increases by one, and the size of x increased. If the rank of x did not change (this happens because $rank(y) < rank(x)$) then the truth of property (a) is preserved by the linking. If the rank of x changes, it must have increased by 1 and this resulted from y having the same rank as x. Let $rank'(x)$, $degree'(x)$ and $size'(x)$ be the new rank, degree and size of x. Then $rank'(x) = 1 + rank(x)$, $degree'(x) = 1 + degree(x)$ and $size'(x) = size(x) + size(y)$. Thus $rank'(x) \le degree'(x)$ follows from $rank(x) \le degree(x)$. We come to the key argument:

$$2^{rank'(x)} = 2^{rank(x)+1} = 2^{rank(x)} + 2^{rank(y)} \le size(x) + size(y) = size'(x).$$

Thus $rank'(x) \le \lg size'(x)$.
(b) Consider a path of length ℓ in a tree rooted at x. By part(iii), the $\ell \le rank(x)$. By part(a), $rank(x) \le \lg n$. Thus $\ell \le \lg n$.
(c) Suppose x and y are two items that were promoted from rank $k-1$ to k at two different times. Let T_x and T_y be the subtrees rooted at x and y immediately after the respective promotions. Clearly T_x and T_y are disjoint. By part(a), $size(x) \ge 2^{rank(x)} = 2^k$, and similarly $size(y) \ge 2^k$. There can be at most $n/2^k$ such trees. **Q.E.D.**

Using the rank heuristic alone, each Find operation takes $\mathcal{O}(\log n)$ time, by property (b). This gives an overall complexity bound of $\mathcal{O}(m \log n)$. It is also easy to see that $\Omega(m \log n)$ is a lower bound if only the rank heuristic is used.

¶9. Analysis of the path compression heuristic. We will show (see Theorem 7 below) that with the path compression heuristic alone, a charge of $\mathcal{O}(\log n)$ for each Find is sufficient. Again this leads to a complexity of $\mathcal{O}(m \log n)$.

So both the path compression heuristic alone and rank heuristic alone lead to the same complexity bound of $\mathcal{O}(m \log n)$. But closer examination shows important differences: unlike the rank heuristic, we cannot guarantee that *each* Find operation takes $\mathcal{O}(\log n)$ time under path compression. Thus, this $\mathcal{O}(m \log n)$ bound is a true amortization bound and not a worst case bound. On the other hand, path compression has the advantage of not requiring extra storage (the rank heuristic requires up to $\lg \lg n$ extra bits per item). Hence there is an interesting tradeoff.

To prove a lower bound on path compression heuristic, we use binomial trees. A **binomial tree** is any tree from an infinite family

$$B_0, B_1, B_2, \ldots,$$

of trees defined recursively as follows: B_0 is trivial tree with one node, and B_{i+1} is obtained from two copies of B_i such that the root of one B_i is a child of the other root (see Figure 6). Clearly, the size of B_i is 2^i. Note that we regard B_i as an non-oriented or un-ordered tree (i.e., the children of a node is not ordered).

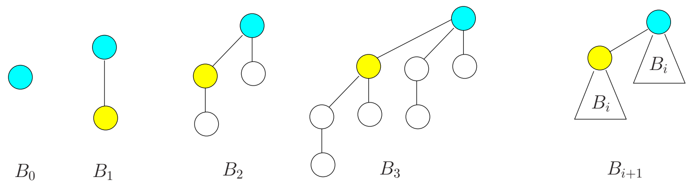


Figure 6: Binomial trees

The next lemma shows some nice properties of binomial trees (the arguments are routine and left as an Exercise).

Lemma 2.
(i) B_i has 2^i nodes.
(ii) For $i \geq 1$, B_i can be decomposed into its root together with a sequence of subtrees of shapes

$$B_0, B_1, \ldots, B_{i-1}.$$

(iii) B_i has depth i. Moreover, level j (for $j = 0, \ldots, i$) has $\binom{i}{j}$ nodes. In particular, B_i has a unique deepest node at level i.

Property (iii) is the reason for the name "binomial trees". To illustrate this lemma, consider B_3 in Figure 6: it has $8 = 2^3$ nodes as claimed by Lemma 2(i). The root has three children, each representing subtrees with shapes B_2, B_1, B_0, as stated in Lemma 2(ii). Finally, levels $0, 1, 2, 3$ (resp.) of B_3 have $1, 3, 3, 1$ nodes. This is equal to $\binom{3}{0}, \binom{3}{1}, \binom{3}{2}, \binom{3}{3}$, as expected by Lemma 2(iii).

¶10. A Self-reproducing Property of Binomial Trees. M.J. Fischer observed an interesting property of binomial trees under path compression. Let $B_{i,k}$ denote any compressed tree which

- has size $k + 2^i$,
- contains a copy of B_i, and
- the root of this B_i coincides with the root of $B_{i,k}$.

Note that $B_{i,0}$ is just B_i. A copy of B_i that satisfies this definition of $B_{i,k}$ is called an "anchored B_i". There may be many such anchored B_i's in $B_{i,k}$. A node in $B_{i,k}$ is **distinguished** if it is the deepest node of some anchored B_i. The right-hand side of Figure 7 shows an instance of $B_{3,1}$.

Also let $B'_{i,k}$ denote the result of linking the root of $B_{i,k}$ to a singleton. Distinguished nodes of $B'_{i,k}$ are inherited from the corresponding $B_{i,k}$. The left-hand side of Figure 7 illustrates $B'_{3,0}$.

Lemma 3. *Suppose we perform a Find on any distinguished node of $B'_{i,k}$. Under the path compression heuristic, the result has shape $B_{i,k+1}$.*

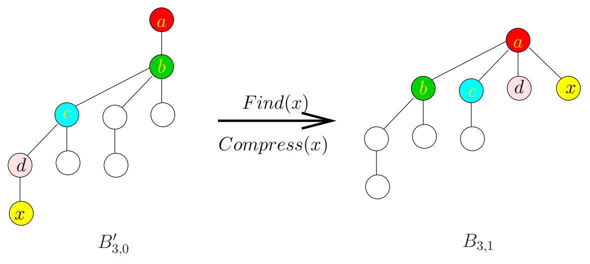


Figure 7: Self-reproducing binomial tree B_3.

This lemma is illustrated in Figure 7 for $B'_{3,0}$ (the node x is the only distinguished node here). We now obtain a lower bound for path compression heuristics as follows. First perform a sequence of links to get B_i such that B_i contains between $n/4$ and $n/2$ nodes. Thus $i \geq \lg n - 2$. Then we perform a sequence of Links and Finds which reproduces B_i in the sense of the above lemma. This sequence of operations has complexity $\Omega(m \log n)$. Our construction assumes $m = \Theta(n)$ but it can be generalized to $m = \mathcal{O}(n^2)$ (Exercise).

Note that if we use path halving heuristic, the resulting tree would be a bit harder to analyze.

At this point, you have the complete algorithmic description of the Union-Find data structure and its algorithms. To understand this data structure really well, we suggest an implementation project in your favorite programming language. The rest of this chapter is concerned with the analysis of its complexity and applications of this data structure.

Exercises

Exercise 2.1: (1) Programming Project: implement the Union-Find data structure in your favorite programming language.
(2) Use it to solve any of the three applications in the the final section of this chapter (Kruskal's algorithm, Optimized Expressions, Betti Numbers). ♢

Exercise 2.2: Suppose we start out with 5 singleton sets $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$, and we perform a sequence of Link/Find operations. Assume the path compression heuristic, but not the rank heuristic. The cost of a Link is 1, and the cost of a Find is the number of nodes in the find-path.
(a) Construct a sequence of 10 Link/Find operations that achieves as large a cost as you can. It is possible to achieve a cost of 22.
(b) Suppose the cost of a Find is *twice* the number of nodes in the find-path (this is more realistic for implementing path compression). This impacts the actual worst case cost, but would your sequence of 12 operations in (a) have changed? ♢

Exercise 2.3: Show that when the size heuristic is used, then height of x is at most $\lg(size(x))$. ♢

Exercise 2.4: We need to allocate $\mathcal{O}(\log\log n)$ extra bits to each node to implement the rank heuristic. Argue that the rank heuristic only uses a **total** of $\mathcal{O}(n)$, not $\mathcal{O}(n\log\log n)$, extra bits at any moment. In what sense is this fact hard to exploited in practice? [Idea: to exploit this fact, we do not pre-allocate $\mathcal{O}(\log\log n)$ bits to each node.] ♢

Exercise 2.5: Prove lemma 2 and lemma 3. ♢

Exercise 2.6: (Chung Yung) Assuming naive linking with the path compression heuristic, construct for every $i \geq 0$, a sequence $(l_1, f_1, l_2, f_2, \ldots, l_{2^i}, f_{2^i})$ of alternating link and find requests such that the final result is the binomial tree B_i. Here l_i links a compressed tree to a singleton element and f_i is a find operation. ♢

Exercise 2.7: In the lower bound proof for the path compression heuristic, we have to make the following basic transformation:

$$B_{i,k} \longrightarrow B'_{i,k} \longrightarrow B_{i,k+1} \tag{5}$$

Beginning with $B_{i,0}$, we repeat the transformation (5) to get an arbitrarily long sequence

$$B_{i,1}, B_{i,2}, B_{i,3}\ldots \tag{6}$$

(a) Can you always choose your transformations so that the trees in this sequence has the form $B_{i+1,k}$ (for some $k \geq 0$)? (b) Characterize those $B_{i,k}$'s that are realizable using the lower bound construction in the text. HINT: first consider the case $i = 1$ and $i = 2$. You may restrict the transformation (5) in the sense that the node for Find is specially chosen from the possible candidates. ♢

Exercise 2.8: Suppose that all the Links requests appear before any Find requests. Show that if both the rank heuristic and path compression heuristic are used, then the total cost is only $\mathcal{O}(m)$. ♢

Exercise 2.9: Let $G = (V, E)$ be a bigraph, and $W : V \to \mathbb{R}$ assign weights to its vertices. The **weight** of a connected component C of G is just the sum of all the weights of the vertices in C.
(a) Give an $\mathcal{O}(m+n)$ algorithm to compute the weight of the heaviest component of G. As usual, $|V| = n, |E| = m$.
(b) Suppose the edges of E are given "on-line", in the order

$$e_1, e_2, e_3, \ldots, e_m.$$

After the appearance of e_i, we must output the weight of the heaviest component of the graph $G_i = (V, E_i)$ where $E_i = \{e_1, \ldots, e_i\}$. Give a data structure to store this information about the heaviest components and which can be updated. Analyze the complexity of your algorithm. ♢

Exercise 2.10: For any $m \geq n$, we want a lower bound for the union-find problem when the path compression heuristic is used alone.
(a) Fix $k \geq 1$. Generalize binomial trees as follows. Let $F_k = \{T_i : i \geq 0\}$ be a family of trees where T_i is a singleton for $i = 0, \ldots, k-1$. For $i \geq k$, T_i is formed by linking a T_{i-k} to a T_{i-1}. What is the degree $d(i)$ and height $h(i)$ of the root of T_i? Show that the size

of T_i is at most $(1+k)^{(i/k)-1}$.
(ii) [Two Decompositions] Show that if the root T_i has degree d then T_i can be constructed by linking a sequence of trees

$$t_1, t_2, \ldots, t_d \tag{7}$$

to a singleton, and each t_i belongs to F_k. Show that T_{i+1} can also be constructed from a sequence

$$s_0, s_1, \ldots, s_p, \quad p = \lfloor (i+1)/k \rfloor \tag{8}$$

of trees by linking s_j to s_{j-1} for $j = 1, \ldots, p$, and each s_j belongs to F_k. The decompositions (7) and (8) are called the **horizontal** and **vertical decompositions** of the respective trees. Also, s_p in (8) is called the **tip** of T_{i+1}.
(iii) [Replication Property] Let T' be obtained by linking T_i to a singleton node R. Show that there are k leaves $x_1, \ldots, x_k$ in T' such that if we do finds on $x_1, \ldots, x_k$ (in any order), path compression would transform T' into a copy of T_i except the root has an extra child. HINT: Consider the trees $r_0, r_1, \ldots, r_{k-1}, r_k$ where $r_j = T_{i-k-j}$ (for $j = 0, \ldots, k-1$) and $r_k = T_{i-k}$. Note that T_i can be obtained by linking each of $r_0, r_1, \ldots, r_{k-1}$ to r_k. Let C be the collection of trees comprising r_k and the trees of the vertical decomposition of r_j for each $j = 0, \ldots, k-1$. What can you say about C?
(iv) Show a lower bound of $\Omega(m \log_{1+(m/n)} n)$ for the union-find problem when the path compression heuristic is used alone. HINT: $k = \lfloor m/n \rfloor$. Note that this bound is trivial when $m = \Omega(n^2)$. ♢

Exercise 2.11: Explore the various compaction heuristics. ♢

End Exercises

§3. Multilevel Partition

The surprising result is that the combination of both rank and path compression heuristics leads to a dramatic improvement on the $\mathcal{O}(m \log n)$ bound of either heuristic. The sharpest analysis of this result is from Tarjan, giving almost linear bounds. We give a simplified version of Tarjan's analysis.

¶11. Partition functions. A function

$$a : \mathbb{N} \to \mathbb{N}$$

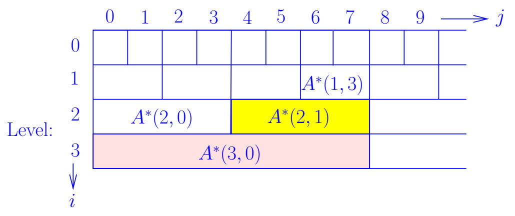
is a **partition function** if $a(0) = 0$ and $a(j) < a(j+1)$ for all $j \geq 0$. Such a function induces a partition of the natural numbers $\mathbb{N}$ where each block of the partition has the form $[a(j)..a(j+1)-1]$. For instance, if a is the identity function, $a(j) = j$ for all j, then it induces the **discrete partition** where each block has exactly one number.

A **multilevel partition function** is $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that:
a) For each $i \geq 0$, $A(i, \cdot)$ is a partition function, which we denote by A_i. The partition on $\mathbb{N}$ induced by A_i is called the **level i partition**.
b) The level 0 partition function A_0 is the identity function, inducing the discrete partition.
c) The level i partition is a coarsening of the level $i-1$ partition.

Let $block(i,j) = [A(i,j)..A(i,j+1)-1]$ denote the jth **block** of the level i partition. For $i \ge 1$, let

$$b(i,j) \ge 1$$

denote the number of $i-1$st level blocks whose union equals $block(i,j)$.



A simple example of a multilevel partition function is A^* defined by

$$A^*(i,j) := j2^i, \qquad (i,j \ge 0). \qquad (9)$$

For reference, call this the **binary partition function**. In this case $b(i,j) = 2$ for all $i \ge 1, j \ge 0$.

Figure 8: Binary Partition Function, $A^*(i,j)$

¶12. Basic Goal. Our goal is to analyze path compaction heuristics (§2), *without necessarily assuming that the rank heuristic is used.* Recall the rank function defined in equation (4); as said, it is meaningful even if we do not use the rank heuristic. Of course, if we do not use the rank heuristic then we need not maintain rank information in our data structure. In this analysis, we assume some fixed sequence

$$r_1, r_2, \ldots, r_m \qquad (10)$$

of m Links/Find request on a universe of n items.

¶13. The eventual rank of an item. The **rank** of an item is monotonically non-decreasing with time. Relative to the sequence (10) of requests, we may speak of the **eventual rank** of an item x. E.g., if x is eventually linked to some other item, its eventual rank is its rank just before it was linked. To distinguish the two notions of rank, we will write $\texttt{erank}(x)$ to refer to its eventual rank. For emphasis, we say "current rank of x" to refer to the time-dependent concept of $rank(x)$.

¶14. The level of an item. The notion of "level of an item" is defined relative to some choice of multilevel partition function $A(i,j)$. The function $A(i,j)$ is used in the analysis only, and does not figure in the algorithms. We say x has **level** i (at any given moment) if i is the smallest integer such that the eventual ranks of x and of its current parent $x.p$ are in the same ith level block. Note the juxtaposition of *current* parent with *eventual* rank in this definition.

For instance, assuming the binary partition function A^* of (9), if $\texttt{erank}(x) = 2$ and $\texttt{erank}(x.p) = 5$ then $level(x) = 3$ (since $2, 5$ both belong in the level 3 block $[0,7]$ but are in different level 2 blocks). Although $\texttt{erank}(x)$ is fixed, $x.p$ can change and consequently the level of x may change with time. Notice that x is a root if and only $\texttt{erank}(x.p) = \texttt{erank}(x)$. Hence x is a root iff it is at level 0.

We leave the following as an easy exercise:

Lemma 4. *Assume that some form of compaction heuristic is used. The following holds, whether or not we use the rank compression heuristics:*
(i) The erank *of nodes strictly increases along any find-path.*
(ii) For a fixed x, $\texttt{erank}(x.p)$ *is non-decreasing with time.*
(iii) The level of node x is 0 iff x is a root.

(iv) The level of any node is non-decreasing over time.

(v) The levels of nodes along a find-path need not be monotonic (non-decreasing or non-increasing), but it must finally be at level 0*.*

¶15. Bound on the maximum level. Let $\alpha \ge 0$ denote an upper bound on the level of any item. Of course, the definition of a level depends on the choice of multilevel partition function $A(i,j)$. Trivially, all ranks lie in the range $[0..n]$. If we use the binary partition function A^* in (9) then we may choose $\alpha = \lg(n+1)$. In general, we could choose α to be the smallest i such that $A(i,1) > n$. If the rank heuristic is used, all ranks lies in the range $[0..\lg n]$. Hence if we again choose the binary partition function A^*, we will obtain

$$\alpha = \lg\lg(2n). \tag{11}$$

¶16. Charging scheme. We charge one unit for each Link operation; this is sufficient to pay for the cost of Linking. The cost of a Find operation is proportional to the length of the corresponding find-path. To charge this operation, we introduce the idea of a **level account** for each item x: for each $i \ge 1$, we keep a *charge account for x at level i*. We **charge** the Find operation in two ways:

- **Upfront Charge**: we simply charge this Find $1+\alpha$ units.
- **Level Charges**: For each node y along the find-path, if $y.p \ne root$ and $level(y.p) \ge level(y)$

  then we **charge one unit to y's account at level** i for each $i \in [level(y)..level(y.p)]$. Thus y incurs a total debit of $level(y.p) - level(y) + 1$ units over all the levels.

¶17. Justification for charging scheme.

Lemma 5. *The charge for each Find operation covers the cost of the operation.*

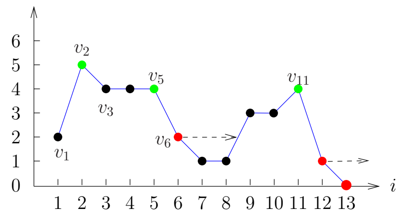


Figure 9: Plot of $level(x_i)$ against i.

This is best shown visually as follows. Let the find-path be $(x_1, x_2, \ldots, x_h)$ where x_h is the root. We must show that $Find(x_1)$ is charged at least h units by the above charging scheme. Plot a graph of $level(x_i)$ against i. As illustrated in Figure 9, this is a discrete set $v_1, v_2, \ldots, v_h$ of points where $v_i = (i, level(x_i))$. Note that by lemma 4(v), the level of v_h is 0. The graph is a polygonal path $(v_1 - v_2 - \cdots - v_h)$ with $h-1$ edges. An edge $(v_i - v_{i+1})$ is called **positive, zero** or **negative**, according to the sign of its slope. Thus Figure 9 has 3 positive edges, 5 negative edges and 4 zero edges. Our level charges amount to charging $k+1$ units to a non-negative edge whose slope is $k \ge 0$; negative edges have no charges. In Figure 9, edge $(v_8 - v_9)$ has slope 2 and so has a level charge of 3. In the figure, we use the following color code for the nodes in a path:

- v_i is **black** if the slope of $(v_i - v_{i+1})$ is non-negative. E.g., $v_1, v_3, v_4, v_7, v_8, v_9, v_{10}$ in Figure 9.
- v_i is **green** if the slope of $(v_{i-1} - v_i)$ is non-negative. E.g., v_2, v_5, v_{11} in Figure 9.
- v_i is **red** otherwise. This means that the slopes of $(v_{i-1} - v_i)$ (if $i > 1$) and $(v_i - v_{i+1})$ (if $i < h$) are both negative. E.g., v_6, v_{12}, v_{13} in Figure 9.

The total cost of a Find operation can be distributed to the vertices, so each vertex has unit cost. The cost of a black vertex v_i is covered by the level charges to $(v_i - v_{i+1})$; likewise, the cost of a green vertex v_i is covered by the level charges to $(v_{i-1} - v_i)$. But how do we cover the cost of a red vertex v_i? This is the key geometrical insight: for each red vertex v_i, we cast a horizontal ray from v_i to the right. There are two possibilities.
(1) This ray may never hit any edge, as in the case of the rays from v_{12} and v_{13}. Note that there are most $1 + \alpha$ of these rays since $0 \le level(x_i) \le \alpha$, and there is at most one ray to infinity at each level.
(2) This ray may hit a point on a positive edge, as in the case of the ray from v_6 hitting the edge $(v_8 - v_9)$. We will cover the cost of v_i using the level charges associated with the positive edge. To see that this is a valid method, note that a positive edge with slope $k \ge 1$ may be hit by no more than k rays. Since it is endowed with $k + 1$ level charges, and one of these charges is to pay for the associated black vertex, we are left with k unused charges. One subtlety is that the ray may hit a zero edge as well. For instance, if the vertex v_6 in Figure 9 were raised one unit higher, it would hit the zero edge $(v_9 - v_{10})$. Nevertheless, it still hits a positive edge, and so we can avoid choosing the zero edge for this accounting.

¶18. The number of level charges. Let x be an item. For each $i \ge 1$, $\texttt{erank}(x)$ belongs to $block(i, j(i))$ for some $j(i)$.

Lemma 6. *The charges to x at level $i \ge 1$ is at most $b(i, j(i))$.*

To show this, note that whenever x is charged at level i, then

$$level(x.p) \ge i$$

holds just preceding that Find. This means the $\texttt{erank}$'s of the parent $x.p$ and grandparent $x.p^2$ belong to different $i - 1$st level blocks. As $\texttt{erank}$'s are strictly increasing along a find-path, after this step, the new parent of x has $\texttt{erank}$ in a new $i - 1$st level block. But the indices of the $i - 1$st level blocks of $x.p$ are nondecreasing over time. Thus, after at most $b(i, j(i))$ charges, $\texttt{erank}(x.p)$ and $\texttt{erank}(x)$ lie in different level i blocks. This means the level of x has become strictly greater than i, and henceforth x is no longer charged at level i. This proves our claim.

Let $n(i, j)$ denote the number of items whose $\texttt{erank}$'s lie in $block(i, j)$. Clearly, for any i,

$$\sum_{j \ge 0} n(i, j) = n. \tag{12}$$

The total level charges, summed over all levels, is at most

$$\sum_{i=1}^{\alpha} \sum_{j \ge 0} n(i, j) b(i, j). \tag{13}$$

¶19. Bound for pure path compression heuristic. Suppose we use the path compression heuristic but not the rank heuristic. Let us choose $A^*(i,j)$ to be the binary partition function and so we may choose $\alpha = \lg(n+1)$. Then the level charges, by equations (12) and (13), is at most

$$\sum_{i=1}^{\lg(n+1)} \sum_{j\ge 0} n(i,j)2 = 2n\lg(n+1).$$

If there are m' Find requests, the upfront charges amount to $\mathcal{O}(m'\log n)$. This proves:

Theorem 7. *Assume the path compression heuristic but not the rank heuristic is used to process a sequence of m Link/Find Requests on a universe of n items. If there are m' Find requests, the total cost is $\mathcal{O}(m + (m'+n)\log n)$.*

In this result, the number m' of Find operations is arbitrary (in particular, we do not assume $m' \ge n$). This result has application in a situation where path compression is allowed but not the rank heuristic (see [7]).

Remark: The above definition of level is devised to work for any of the compaction heuristics noted in §2. A simpler notion of level will work for path compression heuristic (see Exercise).

Exercises

Exercise 3.1: (Chung Yung) Define the **level** of x to be the least i such that both x and the root r (of the tree containing x) have `eranks` in the same ith level block. Give a charging scheme and upper bound analysis for path compression using this definition of level.

◇

End Exercises

§4. Combined Rank and Path Compression Heuristics

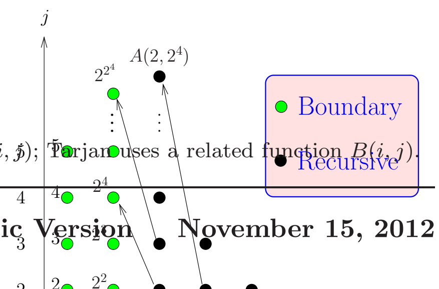
We now prove an upper bound on the complexity of Links-Find in case we combine both the rank and path compression heuristics.[1] Now all ranks lie in the range $[0..\lg n]$. We noted above that if we use the binary partition function A^*, then the level of any node lies in the range $[0..\lg\lg n]$ (see equation (11)). This immediately gives a bound of

$$\mathcal{O}((m+n)\lg\lg n)$$

on the problem. To get a substantially better bound, we introduce a new multilevel partition function.

¶20. Ackermann Functions. A number theoretic function $A : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ is called an **Ackermann function** if satisfies the following fundamental recurrence:

$$A(i,j) = A(i-1, A(i,j-1)) \qquad (14)$$



[1] Our multi-levels directly uses the the Ackermann function $A(i,j)$; Tarjan uses a related function $B(i,j)$.

for i, j large enough. A particular Ackermann function depends on the choice of the boundary conditions. In the following, we fix the following boundary conditions:

$$\left\{\begin{array}{lcll} A(i,0) & = & 0, & i \geq 0, \\ A(0,j) & = & j, & j \geq 0, \\ A(1,j) & = & 2^j, & j \geq 1, \\ A(i,1) & = & A(i-1,2), & i \geq 2. \end{array}\right. \tag{15}$$

The conditions (15) determine the values of $A(i,j)$ when $i < 2$ or when $j < 2$. For $i \geq 2$ and $j \geq 2$, the general recurrence (14) is applicable.

Some Values of the Ackermann function. Figure 10 is a useful visual aid: the domain $\mathbb{N} \times \mathbb{N}$ is represented by an infinite array of grid points. The values of A on the green points are given by the boundary conditions (15). The values at black points are given by the recurrence (14). It follows that

$$\begin{array}{rcll} A(2,j) & = & A(1, A(2,j-1)), & j \geq 2 \\ & = & \left\{\begin{array}{ll} 2^2, & j = 2 \\ 2^{A(2,j-1)}, & j > 2 \end{array}\right. & \\ & = & \exp_2^{(j)}(2) = 2^{2^{\cdot^{\cdot^{2}}}} & \end{array}$$

(a stack of j 2's). Also $A(3,1) = A(2,2) = A(1,4) = 16$ and $A(4,1) = A(3,2) = A(2,16)$. The last number is more than the number of atoms in the known universe (last estimated at 10^{80}). The dominant feature of an Ackermann function is its explosive growth – in fact, it grows faster than any primitive recursive function.

Lemma 8. *The Ackermann function $A(i,j)$ is a multilevel partition function.*

Proof. Conditions a) and b) in the definition of a multilevel partition function are immediate. For condition c) we must show that the ith level partition determined by A is a coarsening of the $i-1$st level partition. This amounts to noting that for all $i \geq 1, j \geq 1$, we can express $A(i,j)$ as $A(i-1,j')$ for some j'. **Q.E.D.**

Let us note some simple properties the Ackermann function.

- The 1st level partition consists of exponentially growing block sizes,

$$block(1,j) = [2^j..2^{j+1}-1].$$

- We want to determine an upper bound α on the level of an item. This can be taken to be the smallest i such that

$$[0..\lg n] \subseteq block(i,0).$$

Clearly, this is the smallest i such that $A(i,1) > \lg n$. In fact, let us define the following **inverse Ackermann function**:

$$\alpha(n) := \min\{i : A(i,1) > \lg n\}.$$

It follows that the level of every item is at most $\alpha = \alpha(n)$. The dominant feature of $\alpha(n)$ is its very slow growth rate. For all practical purposes, $\alpha(n) \leq 4$ (see exercise below).

- The number of $i-1$st level blocks that form $block(i,j)$ is given by the following:

$$b(i,j) = \left\{\begin{array}{ll} 2 & \text{if } j = 0, \\ A(i,j) - A(i,j-1) & \text{if } j \geq 1. \end{array}\right. \tag{16}$$

To see that $b(i,0)=2$, we note that $A(i,1)=A(i-1,2)$. For $j\geq 1$, we note that $A(i,j+1)=A(i-1,A(i,j))$.

- The number $n(i,j)$ of items whose `erank`s lie in $block(i,j)$ is bounded by:

$$n(i,j)\leq \sum_{k=A(i,j)}^{A(i,j+1)-1}\frac{n}{2^k}\leq \frac{2n}{2^{A(i,j)}},$$

where the first inequality comes from lemma 1(c).

- It follows that the number of charges at level $i\geq 1$ is bounded by

$$\sum_{j\geq 0} b(i,j)n(i,j)\leq b(i,0)n(i,0)+\sum_{j\geq 1}\frac{A(i,j)\cdot 2n}{2^{A(i,j)}}\leq 2n+2n\sum_{k\geq 2}\frac{k}{2^k}\leq 5n,$$

using the basic summation

$$\sum_{j=i}^{\infty}\frac{j}{2^j}=\frac{i+1}{2^{i-2}}.$$

Since there are $\alpha(n)$ levels, there is a bound of $\mathcal{O}(n\alpha(n))$ on all the level charges. Combined with the $\mathcal{O}(\alpha(n))$ Upfront charge for each Find, we conclude:

Theorem 9. *When the rank and path compression heuristics are employed, any sequence of m Links/Find request incurs a cost of*

$$\mathcal{O}((m+n)\alpha(n))$$

¶21. Amortization Framework. In Lecture VI, we presented the potential framework. We briefly review this in the setting of the Union-Find problem. We are given a sequence $p_1,\ldots,p_m$ of requests to process. Our amortization scheme establishes a finite set of **accounts** $A_1,\ldots,A_s$ such that the cost of each request p_i is distributed over these s accounts. More precisely, the scheme specifies a **charge** by p_i to each account A_j.

In our Link/Find analysis, we set up an account for each operation p_i (this is called the up-front charges for find requests) and for each item x at each level ℓ, we have an account $A_{x,\ell}$. Note that the charges to each account is non-negative value (these are pure debit accounts).

If $Charge(A_j,p_i)$ units is charged to account A_j by request p_i, we require

$$\sum_{j=1}^{s} Charge(A_j,p_i)\geq Cost(p_i). \tag{17}$$

If $Charged(A_j)$ is the sum of all the charges to A_j, then the amortization analysis amounts to obtaining an upper bound on $\sum_{j=1}^{s} Charged(A_j)$.

We may combine the charge account framework with the potential framework. We now allow an account to be **credited** as well as debited. To keep track of credits, we associate a **potential** $\Phi(A_j)$ to each A_j and let $\Delta\Phi(A_j,p_i)$ denote the increase in potential of A_j after request p_i. Then (17) must be generalized to

$$\sum_{j=1}^{s}(Charge(A_j,p_i)-\Delta\Phi(A_j,p_i))\geq Cost(p_i). \tag{18}$$

Without loss of generality, assume that $\Phi(A_j)$ is initially 0 and let Φ_j denote the final potential of A_j. The total cost for the sequence of operations is given by

$$\sum_{j=1}^{s}(Charged(A_j) - \Phi_j).$$

If we are interested in the "amortized cost" of each type operation, this amounts to having an account for each operation p_i and charging this account an "amortized cost" corresponding to its type. We insist that $\sum_{j=1}^{s} \Phi_j \geq 0$ in order that the amortized cost is meaningful.

Exercises

Exercise 4.1: Verify the claim that $A(4,1) > 10^{80}$ (number of atoms in the known universe). ♢

Exercise 4.2: Compute $A(3,3)$. ♢

Exercise 4.3: What is the smallest j such that $A(2,j)$ is larger than a virgintillion (the number 10^{63})? Larger than a googleplex (the number $10^{10^{100}}$)? ♢

Exercise 4.4:
(i) Show $A(i,j)$ is strictly increasing in each coordinate.
(ii) When is $\alpha(m,n) \leq 1$? When is $\alpha(m,n) \leq 2$?
(iii) Compute the smallest ℓ such that $\alpha(\ell)$ is equal to $0, 1, 2, 3, 4$. ♢

Exercise 4.5: (Tarjan) Define a two argument version of the inverse Ackermann function, $\alpha'(k,\ell)$, $k \geq \ell \geq 1$, where

$$\alpha'(k,\ell) := \min\{i \geq 1 : A(i, \left\lfloor \frac{k}{\ell} \right\rfloor) > \lg \ell\}.$$

The one-argument function $\alpha'(\ell)$ may be defined as $\alpha'(\ell,\ell)$, and could be used instead of the $\alpha(\ell)$ function used in the text. Note that $\alpha'(k,\ell)$, for fixed ℓ, is actually a decreasing function in k. Improve the amortized upper bound to $\mathcal{O}(m\alpha'(m+n,n))$. ♢

Exercise 4.6: Consider the following multilevel partition function: $A(i,j) = \left\lfloor \lg^{(i)}(j) \right\rfloor$ (i-fold application of lg to j). Fix your own boundary conditions in this definition of $A(i,j)$. Analyze the union-find heuristic using this choice of multilevel partition function. ♢

Exercise 4.7: Prove similar upper bounds of the form $\mathcal{O}(m\alpha(n))$ when we replace the path compression heuristic with either the splitting or halving heuristics. ♢

End Exercises

§5. Three Applications of Disjoint Sets

There are many applications of the disjoint set data structure. Three will be given here. The first problem is in the implementation of Kruskal's algorithm for minimum spanning tree. The second problem concerns restructuring of algebraic expressions, a problem which arise in optimizing compilers and in Exact Geometric Computation. The last problem concerns the computation of Betti numbers in in Computational Topology.

¶22. 1. Implementing Kruskals' Algorithm. In Lecture V, we presented Kruskal's algorithm for minimum spanning tree (MST). We now show how to implement it efficiently using the Union Find data structure.

Recall the problem of computing the MST of an edge-costed bigraph $G = (V, E; C)$, with edge costs $C(e), e \in E$. The idea of Kruskal is to sort all edges in E in increasing order of their costs, breaking ties arbitrarily. We initialize $S = \emptyset$ and at each step, we consider the next edge e in the sorted list. We will insert e into the set S provided this does not create a cycle in S (otherwise e is discarded). Thus, inductively, S is a forest. When the cardinality of S reaches $|V| - 1$, we stop and output S as the MST.

Implementation and complexity. We must be able to detect whether adding e to a forest S will create a cycle. This is achieved using a Union-Find structure to represent the connected components of S. If $e = (u, v)$, then $S \cup \{e\}$ creates a cycle iff $Find(u) = Find(v)$. Moreover, if $Find(u) \neq Find(v)$, we will next form the union of their components via $Union(u, v)$. To initialize, we must set up the Union-Find structure for the set V with the trivial equivalence relation. The algorithm makes at most $2m$ Finds and n Unions. But since each union on u, v is performed right after a $Find(u)$ and $Find(v)$, we can replace each Union by a single link. The amortized cost for these Link/Finds is

$$\mathcal{O}(m\alpha(n)).$$

This cost is dominated by the initial cost of $\mathcal{O}(m \log n)$ for sorting the edges. Hence the complexity of the overall algorithm is $\mathcal{O}(m \log n)$.

¶23. 2. Expression Optimization Problem. Suppose you are given an arithmetic expression, represented by a directed acyclic graph (DAG) G, with operators in the internal nodes and numerical constants or variables at the leaves. For instance, G might represent the expression

$$\sqrt{2} + \sqrt{3} - \sqrt{2 + 3 - \sqrt{6}}$$

This expression is an instance of the more general expression with variables instead of constants:

$$\sqrt{x} + \sqrt{y} - \sqrt{x + y - \sqrt{xy}}. \tag{19}$$

Thus one possible representation of the expression (19) is given by G in Figure 11. We want to compute a "reduced expression" G' that is equivalent to G. In Figure 11, we show such a G'. What is "reduced" about G'? Notice that it has one fewer node. But in this application, the more important fact is that G' has only three square-root operators, in contrast to the four square-roots in G.

This is an important problem in the area of Exact Geometric Computation where we are interested in numerical approximations for the values $val(G)$ of expressions. Note that even if there are variables like x or y in G, it is assumed that we know explicit constants for these variables so that $val(G)$ is actually a number. The key problem is to decide when $val(G)$ is really zero. We can approximate $val(G)$ to as many bits of accuracy as we like (using big number arithmetic). But if the expression is really zero, how do we know to stop? This problem can be solved when the expression involves only algebraic operations such as $+, -, \times, \div, \sqrt{\cdot}$. It turns out that we can compute a so-called **zero bound** $\beta(G) > 0$ for any expression G such that if $val(G) \neq 0$ then $|val(G)| > \beta(G)$. The graph G might be automatically generated by a program and can be quite large. As a result, $\beta(G)$ can be astronomically small, say $\beta(G) \simeq 2^{-10,000}$. This means that in order to decide if $val(G) = 0$, we need to approximate $val(G)$ to over $10,000$ bits. Suppose we have a reduced G' whose zero bound $\beta(G')$ is considerably larger (sic!) than $\beta(G)$, say $\beta(G') \simeq 2^{-100}$. The equivalence of expressions G and G' simply means that $val(G) = val(G')$. In this case, we only need to approximate $val(G') = val(G)$ to about 100 bits, and thus deciding if $val(G) = 0$ is greatly sped up. The theory behind such zero bounds is beyond our scope.

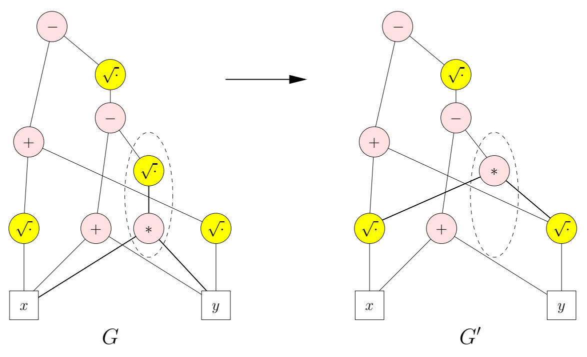


Figure 11: Reducing G to G'

We assume that the DAG G is represented as an adjacency list. The nodes with indegree 0 are called sources, and the nodes with outdegree 0 are called sinks. Each sink of G is labeled with a distinct variable name (x_1, x_2, etc). Each non-sink node of G has outdegree 2 and are labelled with one of four arithmetic operations $(+, -, *, \div)$. See Figure 12, where the directions of edges are implicitly from top to bottom. Thus, every node of G represents an algebraic expression over the variables. E.g., node b represents the expression $x_2 - x_3$.

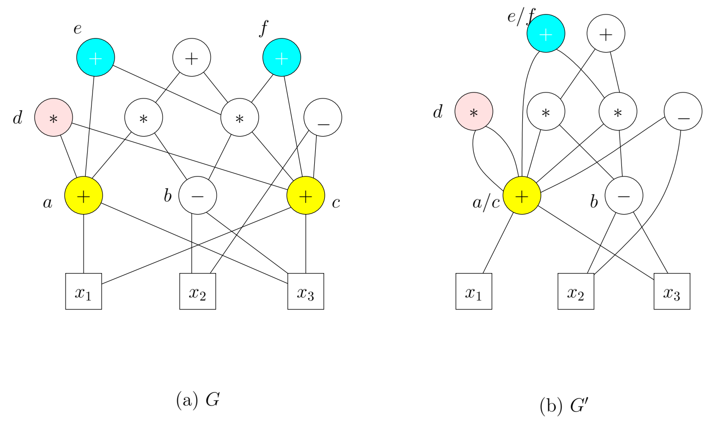


Figure 12: Two DAGs: G and G'

The two edges exiting from the '$-$' and '$\div$' nodes are distinguished (i.e., labeled as "Left" or "Right" edges) while the two edges exiting from $+$ or $*$ nodes The Left/Right order of edges are implicit in our figures. are indistinguishable. This is because $-$ and $\div$ are non-commutative ($x-y \neq y-x$, and $x \div y \neq y \div x$ in general) while $+$ and $*$ are commutative ($x+y=y+x$ and $x*y=y*x$). In general, G is actually a multigraph because it is possible that there there is more than one edge from a node to another. E.g., the graph G' in Figure 12(b) has two edges from d to a.

Define two nodes to be **equivalent** if they are both internal nodes with the same operator label, and their "corresponding children" are identical or (recursively) equivalent. This is a recursive definition. By "corresponding children" we mean that the order (left or right) of the children should be taken into account in case of $-$ and $\div$, but ignored for $+$ and $*$ nodes. For instance, the nodes a and c in Figure 12(a) are equivalent. Recursively, this makes e and f equivalent.

The problem is to construct a **reduced DAG** G' from G in which each equivalence class of nodes in G are replaced by a single new node. For instance with G in Figure 12(a) as input, the reduced graph is G' in Figure 12(b).

The solution is as follows. The height of a node in G is the length of longest path to a sink. E.g., height of node e in Figure 12(a) is 3 (there are paths of lengths 2 and 3 from e to sinks). Note that two nodes can only be equivalent if they have the same height. Our method is therefore to checking equivalent nodes among all the nodes of a given height, assuming all nodes of smaller heights have been equivalenced. To "merge" equivalent nodes, we use a disjoint set data structure. If there are k nodes of a given height, the obvious approach takes $\Theta(k^2)$ time.

To avoid this quadratic behavior, we use a sorting technique: let A be the set of nodes of a given height. For each $u \in A$, we first find their children u_L and u_R using the adjacency lists of graph G. Then we compute $U_L = Find(u_L)$ and $U_R = Find(u_R)$. Hence U_L and U_R are the representative nodes of their respective equivalence classes. Let $op(u)$ be the operator at node u. In case $op(u)$ is $+$ or $*$, we compare U_L and U_R (all nodes can be regarded as integers). If $U_R < U_L$, we swap their values. We now construct a 3-tuple

$$(op(u), U_L, U_R)$$

which serves as the key of u. Finally, we sort the elements of A using the keys just constructed. Since the keys are triples, we compare keys in a lexicographic order. We can assume an arbitrary order for the operators (say $'+' <' -' <' *' <' \div'$). Two nodes are equivalent iff they have identical keys. After sorting, all the nodes that are equivalent will be adjacent in the sorted list for A. So, we just go through the sorted list, and for each u in the list, we check if the key of u is the same as that of the next node v in the list. If so, we perform a $merge(u, v)$. This procedure takes time $O(k \lg k)$.

We must first compute the height of each node of G. This is easily done using depth-first search in $O(m)$ time (see Exercise). Assume that all the nodes of the same height are put in a linked list. More precisely, if the maximum height is H, we construct an array $A[1..H]$ such that $A[i]$ is a linked list of all nodes with height i. Separately, we initialize a disjoint set data structure for all the nodes, where initially all the sets are singleton sets. Whenever we discover two nodes equivalent, we form a union. We will now process the list $A[i]$ in the order $i = 1, 2, \ldots, H$, using the sorting method described above.

Complexity: computing the height is $O(m+n)$ (Exercise). Union Find is $m\alpha(n)$. Sorting of all the lists is $\sum_{i=1}^{H} O(k_i \log k_i) = O(n \log n)$ where $|A[i]| = k_i$. Hence the overall complexity is $O(m + n \log n)$.

Remark: We have actually solved a simple form of the general problem. For instance, the example in Figure 11 requires an additional transformation, $\sqrt{xy} \to \sqrt{x} \cdot \sqrt{y}$ before we can achieve the indicated reduction.

¶24. 3. Computing Betti Numbers. Let K be a non-empty set of tetrahedra, triangles, edges and vertices, where each $s \in K$ is a closed set of $\mathbb{R}^3$. Note that a tetrahedron, triangle, edge and vertex is also known as a **simplex** of **dimension** $3, 2, 1, 0$, respectively. We assume that the relation s is a **proper face** of s' is understood. For instance, a tetrahedron has 3 proper faces of dimension 2, 6 proper faces of dimension 1 and 4 proper faces of dimension 0. The empty set $\emptyset$ is regarded as a simplex of dimension -1. Then any simplex s has two **improper faces**, namely itself and $\emptyset$. All the other faces are **proper**.

For K to be a **triangulation**, we require that for all $s, s' \in K$, (a) any face of s also belongs to K, and (b) $s \cap s'$ is a face of s and of s'. Thus, $\emptyset \in K$. The underlying space or support space of K is given by $|K| := \cup_{s\in K}\, s$. Delfinado and Edelsbrunner shows how we can compute $\beta_i(K)$, the i-th Betti number of K, using Union Find as follows. The Betti numbers are defined algebraically and is a topological invariant of $|K|$. For $|K| \subseteq \mathbb{R}^3$, these numbers have an intuitive meaning: β_0 is the number of connected components of $|K|$, β_1 is the number of holes of $|K|$ (e.g., the donut as well as the torus has $\beta_1 = 1$), and β_2 is the number of voids on $|K|$ (e.g., the torus and the sphere S^2 has one void ($\beta_2 = 1$) each, but the donut has none, $\beta_2 = 0$). Let us just consider the case where K is just a graph, i.e., a set of vertices and edges only. The following basic lemma is all that we need:

Lemma 10. *Let K and K' be 1-dimensional complexes, and $K' = K \cup \{s\}$ where s is an edge connecting u, v. If u, v lies in the same connected component of K, then $\beta_1(K') = \beta_1(K) + 1$; otherwise $\beta_0(K') = \beta_0(K) - 1$. The other Betti number is unchanged.*

Note that if $K' = K \cup \{s\}$ and s is a vertex, then it is clear that $\beta_0(K') = \beta_0(K) + 1$ and the other Betti number is unchanged. Armed with this lemma, we can now give a simple incremental algorithm to maintain Betti numbers of K: we initialize b_0 to n, the number of vertices in K, and b_1 to 0. The n vertices are used to initialize a union-find data structure. Now we process each edge of K in turn. For each edge (u, v), if $Find(u) = Find(v)$, then we perform $Union(u, v)$ and update $b_1 = b_1 + 1$; otherwise we update $b_0 = b_0 - 1$. The final values of b_0, b_1 is the Betti number we seek.

The problem with extending this to 2-dimensional complexes K is that we do not have an efficient way to detect when the addition of a new triangle will create a void (although this is possible to do, by using tools of linear algebra). But suppose by embedding K in another complex L where $|L|$ is a tetrahedron in $\mathbb{R}^3$, then we can exploit duality for this purpose (Exercise).

¶25. Final Remarks. Tarjan has shown that the upper bounds obtained for disjoint sets is essentially tight under the pointer model of computation. Ben-Amram and Galil [1] proved a lower bound of $\Omega(m\alpha(m, n))$ lower bound on Union-Find problem under a suitable RAM model of computation. Gabow and Tarjan shows that the problem is linear time in a special case. There are efficient solutions that are not based on compressed trees. For instance, the exercise below shows a simple data structure in which Find requests take constant time while Union requests takes non-constant time. Another variation of this problem is to ensure that each request has a good worst case complexity. See [2, 5].

The application to expression optimization arises in the construction of optimizing compilers in general. In an application to robust geometric computation [4], such optimizations are critical for reducing the potentially huge bit complexity of the numerical computations.

Exercises

Exercise 5.1: Hand simulate Kruskal's algorithm on the graph in Figure 13. Specifically:
(a) First show the list of edges, sorted by non-decreasing weight. View vertices v_1, v_2, v_3, etc as the integers $1, 2, 3$, etc. We want you to break ties as follows: assume each edge has the form (i, j) where $i < j$. When the weights of (i, j) and (i', j') are equal, then assume the weight of (i, j) is less than that of (i', j') if (i, j) is lexicographically less than (i', j').
(b) Maintain the union-find data structure needed to answer the basic question in Kruskal's algorithm (namely, does adding this edge creates a cycle?). The algorithms for the Union and Find must use both the rank heuristic and the path compression heuristic. At each stage of Kruskal's algorithm, when we consider an edge (i, j), we want you to perform the corresponding $Find(i), Find(j)$ and, if necessary, $Union(Find(i), Find(j))$. You must show the result of each of these operations on the union-find data structure. ♢

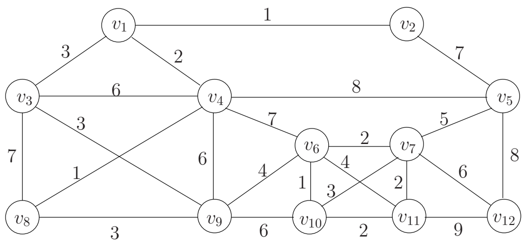


Figure 13: Another House Graph

Exercise 5.2: Let $G = (V, E; C)$ be the usual input for MST. We consider a variant where we are also given a forest $F \subseteq E$. We want to construct a minimum cost spanning tree T of G subject to the requirement that $F \subseteq T$. Call this the **constrained MST** problem. As usual, let us just consider the cost version of this problem, where we only want to compute the minimum cost. Describe and prove the correctness of a Kruskal-like algorithm for this problem. Analyze its complexity. ♢

Exercise 5.3: Let us define a set $S \subseteq E$ to be **Kruskal-safe** if (i) S is contained in an MST and (ii) for any edge $e \in E \setminus S$, if $C(e) < \max\{C(e') : e' \in S\}$ then $S \cup \{e\}$ contains a cycle. Note that condition (i) is what we called "simply safe" in §V.3. Show that if S is Kruskal-safe and e is an edge of minimum costs among those edges that connect two connected components of S then $S \cup \{e\}$ is Kruskal safe.

♢

Exercise 5.4: Describe an algorithm to determine the height of every node in a DAG G. Assume an adjacency list representation of G. Briefly show its correctness and analyze its running time. ♢

Exercise 5.5: Let K be a simplicial complex and $|K| \subseteq \mathbb{R}^3$.

(a) Show that we can embed K in another simplicial complex L such that $|L|$ is a tetrahedron.

(b) Suppose we have an incremental algorithm to compute the Betti numbers of L. Show how we can obtain use it to compute the Betti numbers of K.

(c) Suppose that the triangles and tetrahedrons (or cells) of L are $t_1, \ldots, t_m$ and $c_1, \ldots, c_n$. Consider the simplicial complex K_i $(i = 0, \ldots, m)$ such that K_i has all the vertices and edges of K but only the triangles $\{t_1, \ldots, t_i\}$. (K_i has no tetrahedrons. We want to detect the situation when a transition from K_{i-1} to K_i creates a new void. Consider the dual graph G_i $(i = 0, \ldots, m)$ whose vertex set is $\{c_1, \ldots, c_n\}$ and whose edges are $\{t_{i+1}, \ldots, t_m\}$. NOTE that in G_i, we identify a triangle t with the edge c, c' where $c \cap c' = t$. Show that adding t_i to K_{i-1} creates a new void in K_i iff the number of connected components increased by one in going from G_{i-1} to G_i.

(d) Show that if we construct the graph G_i's in reverse order (starting from G_m down to G_0) then we can detect all the i's such that adding t_i creates a new void. Describe this algorithm's implementation using Union-Find.

(e) Conclude that we can maintain the Betti numbers of L in time $O(n\alpha(n))$ where n is the number of simplices in L. ◇

End Exercises

References

[1] A. M. Ben-Amram and Z. Galil. Lower bounds of data structure problems on rams. *IEEE Foundations of Comp. Sci.*, 32:622–631, 1991.

[2] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM J. Computing*, 15:1021–1024, 1986.

[3] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computmg Surveys*, 23(3):319–344, 1991.

[4] C. Li and C. Yap. Recent progress in Exact Geometric Computation, 2001.

[5] M. H. M. Smid. A datastructure for the union-find problem having a good single-operation complexity. *Algorithms Review*, 1(1):1–12, 1990.

[6] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA, 1974.

[7] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[8] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31:245–281, 1984.