Lecture XII
DISJOINT SETS

Let S be a set of items, and let $\equiv$ be an equivalence relation on S. The **union-find problem** concerns the maintainence of this equivalence relation, subject to two kinds of "requests": The equivalence relation $\equiv$ can be queried using the **Find** request, and it can be modified using the **Union** request. The problem is also called the **set union**, **disjoint set** or **set equivalence problem**. Among its many applications, we will illustrate its use in Kruskal's algorithm for minimum spanning tree. The analysis uses a sophisticated form of amortization analysis.

§1. Union Find Problem

Consider a collection
$$P = \{S_1, \ldots, S_k\}$$
of pairwise disjoint non-empty sets where each S_i has a **representative item** $x_i \in S_i$. The choice of this representative item is arbitrary. It is convenient in the following to assume that P forms a partition of the set
$$[1..n] := \{1, 2, \ldots, n\}.$$
The items in the sets of P have no particular properties except that two such items can be checked for equality. In particular, there is no ordering property on the items. For any $x \in [1..n]$, let $set(x) \subseteq [1..n]$ denote the set in P that contains x, and let $rep(x)$ be the representative element of $set(x)$. Thus $rep(x) \in set(x)$. We say that two items in $[1..n]$ are **equivalent** if they belong to the same set. Each $S_i \in P$ is also called an **equivalence class**. Hence x, y are equivalent iff $rep(x) = rep(y)$ iff $set(x) = set(y)$.

The partition P is dynamically changing and subject to requests that have one of two forms:

- $Find(x) \to z$: this returns the representative item $z = rep(x)$.
- $Union(x, y) \to z$: this modifies P so that $set(x)$ and $set(y)$ are replaced by their union $set(x) \cup set(y)$. It returns an arbitrarily chosen representative item z in the new set $set(x) \cup set(y)$.

The problem of processing a sequence of such requests is called the **Union-Find** or **disjoint set** or **set equivalence problem**. Note that our sets can grow but not shrink in size.

Example. Let $n = 4$ and P consists initially of the singleton sets $S_i = \{i\}$ for $i \in [1..4]$. Let the sequence of requests be
$$Union(2,3), Find(2), Union(1,4), Union(2,1), Find(2).$$
The partition P finally consists of just one equivalence class, the entire set $[1..4]$. 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.

A sequence of Union/Find requests can be viewed as a sequence of **equivalence assertions** and **equivalence class queries**. Each $Find(x)$ amounts to querying the equivalence class of x and each $Union(x, y)$ amounts to asserting the equivalence of x and y (from that moment onwards). There are many applications of this. One original motivation is in FORTRAN compilers, where one needs to process EQUIVALENCE statements of the FORTRAN language, asserting the equivalence of programming variables. Another application is to in finding connected components in bigraphs. For more details on this problem, see [4, 6].

Complexity parameters. The complexity of processing a sequence of these requests will be given as a function of m and n, where n is the size of the universe $[1..n]$ and m the number of requests (i.e., Union or Find operations). For simplicity, we assume that there are initially n singleton sets and $m \geq n/2$. This inequality comes from insisting that every item in the universe must be referenced in either a Find or a Union request.

However, the parameter m in the following discussions will given a slightly different meaning: recall that the Union operation takes two arbitrary items x, y as argument. In the a set S is represented by a rooted unordered tree whose node set is S; the root serves as the representative of S. Each node $x \in S$ has a **parent pointer** $p(x)$; the tree structure is represented by these pointers. The root is the unique node $x \in S$ such that $p(x) = x$. If x, y are roots of trees, we have the basic operation

$$Link(x, y)$$

where we **link** x to y by setting $p(x)$ to y. The result is a new tree rooted at y, representing the union of the original two sets. [If $x = y$, then $Link(x, y)$ is a null operation.] We can implement a Union requests by two Finds and a Link:

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

So we may replace a sequence of m Union/Find requests by an equivalent sequence of $\leq 3m$ 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. Therefore, we will simple analyze a sequence of m Link/Find operations. In some of our analysis, we use another parameter m' ($m' \leq m$) which is the number of Find requests in the m operations.

Three Solutions. We now present three 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. Following links from any node will lead us to the head of a list. Hence it is natural to make the head of each list serve as the representative of the set represented by the list. See figure 1(a). Note that linking takes constant time, *provided* we maintain at each head (*i.e.*, set representative) a pointer to the tail of its list. On the other hand, Find operation requires, in the worst case, time proportional to the length of the list. Clearly, the complexity of a sequence of m requests is $\mathcal{O}(mn)$.

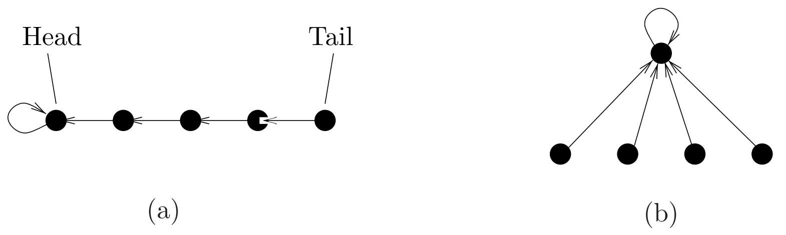


Figure 1: (a) Linked list solution. (b) Anti-list solution.

Anti-list Solution Another solution is to represent each set in P as a tree in which every node points to the root. See figure 1(b). This data structure is the antithesis of lists; we may call such trees "anti-lists". The root serves as the representative of the set. Clearly a Find request takes constant time. On the other hand, linking x to y takes time proportional to the size of *set*(x) since we need to make every element in *set*(x) point to y. It is easy to see that the overall complexity of processing m Link/Find requests is $\mathcal{O}(m + n^2)$.

Compressed Tree Solution In the previous solutions, Linking is easy using linked list, while Find is easy using anti-lists. We now combine the advantages of both solutions. Following Galler and Fischer, we represent each set S in P by a **compressed tree**. This is an unordered rooted tree $T(S)$ whose set of nodes may be identified with S. The root of $T(S)$ is the unique node x such that $p(x) = x$ also serves as the representative of S. Let the root of $T(S)$ be the representative of S. We emphasize that in these trees, there are neither child-pointers nor any á priori requirement on their structure. Indeed, it is not feasible to have child-pointers (why?). Clearly, linked list and anti-lists are just two extreme cases of compressed trees.

Example. 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 2.

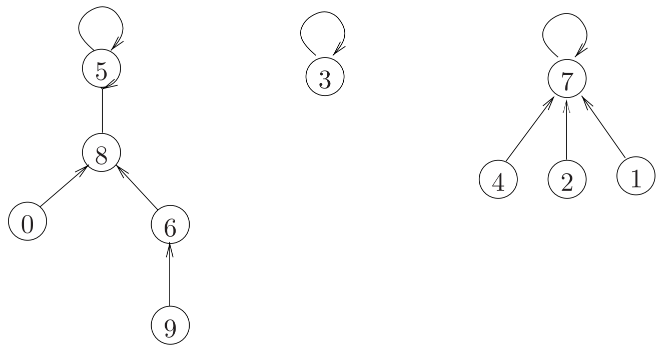


Figure 2:

Naive compressed tree algorithms. Given the above representation of sets, there are obvious algorithms for $Find(x)$ (just follow parent pointers until the root, which is returned) and $Link(x, y)$ (if $x \neq y$ then $p(x) = y$). These simple algorithms can lead to a degenerate tree that is a linear list of length $n-1$. Clearly $\mathcal{O}(mn)$ is an upper bound.

Exercises

Exercise 1.1: Show the stated upper bounds for the above data structures are tight by demonstating matching lower bounds.
(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: Generalize any of the above Union-Find data structures to support a new operation "In-equiv(x,y)". This amounts to detaching x from the set *set*(y), and setting up x as a new singleton set. Analyze its complexity. ♢

Exercise 1.3: Assume that we use anti-lists and when linking two trees, we use the "size heuristic": we always append the smaller set to the larger set. We need to keep track of the length of lists to do this, but this is easy. Show that this scheme achieves $\mathcal{O}(m + n \log n)$ complexity. Prove a corresponding lower bound. ♢

Exercise 1.4: 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 $2\lg n$.
(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**.

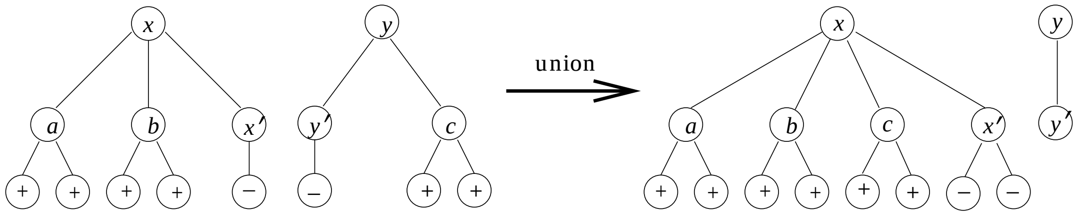


Figure 3: Illustration of Union.

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 $\mathcal{O}(n \log\log n)$. ♢

End Exercises

§2. Rank and Path Compression 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.

Size and Rank heuristics. Consider how we can improve the performance of Links. 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:

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

Note that the rank of y never decrease by this assignment. This assignment is the only way by which a rank changes. If compressed trees are never modified except through linking, then it is easy to see that $rank(x)$ is simply the height of x. In general, the rank of x turns out be just an upper bound on the height of x. The **rank heuristic** is this:

When we union two trees rooted, respectively, at x and y, we link x to y if $rank(x) \le rank(y)$, and otherwise we link y to x.

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

Path compression heuristic. Now consider a heuristic to improve the performance of Finds. This was introduced by McIlroy and Morris. Whereas the rank heuristic is used during a Links request, this heuristic is used during a Find request. When we do a Find on an item x, we traverse some path from x to the root u of its tree. This is called the **find-path** of x. The **path compression heuristic** is this:

After we perform a Find on x, we modify the parent pointer of each node y along the find-path of x to point to the root u.

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

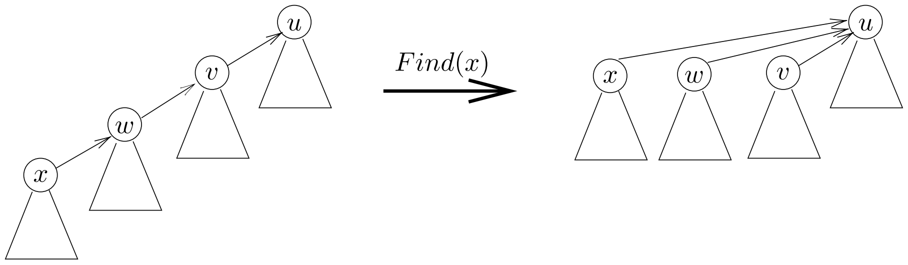


Figure 4: Path compression heuristic.

This algorithm 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.

Analysis of the rank heuristic. We state some simple properties of the rank function. Notice that the rank function is defined according to equation (1), whether or not we use the rank heuristic.

Lemma 1.
The rank function (whether or not the rank heuristic is used, and even in the presense 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. A node has rank 0 iff it is a leaf.
(iv) If the rank heuristic is used, the rank function has additional properties:

(a) $rank(x) \leq degree(x)$ and $2^{rank(x)} \leq size(x)$.
(b) No path has length more than $\lg n$.
(c) In any sequence of Links/Find requests, the number of times that the rank of any item gets promoted from $k-1$ to k is at most $n2^{-k}$, for any $k \geq 1$.

Proof. (i)-(iii) are immediate.

pf of old version:

(iv) (a) This follows by induction. It is true when x is a root and when it dies.
(b) This follows from (ii) and (a), since a path of length ℓ occurs in a tree of size at least 2^ℓ and so $2^\ell \leq n$.
(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 subtree rooted at x and y immediately after the respective promotions. Clearly, T_x and T_y are disjoint and $|T_x|, |T_y| \geq 2^k$. The result follows. **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.

Analysis of the path compression heuristic. We will show below (see §3) 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 $\mathcal{O}(m \log n)$.

This suggests that path compression heuristic alone and rank heuristic alone give the same complexity results. 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. 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 be made.

To prove a lower bound on path compression heuristic, we use binomial trees. A **binomial tree** is any tree that has the shape of some B_i, $i \geq 0$, where

$$B_0, B_1, \ldots,$$

is an infinite family of trees defined recursively as follows: B_0 is the singleton. 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 5). Clearly, the size of B_i is 2^i.

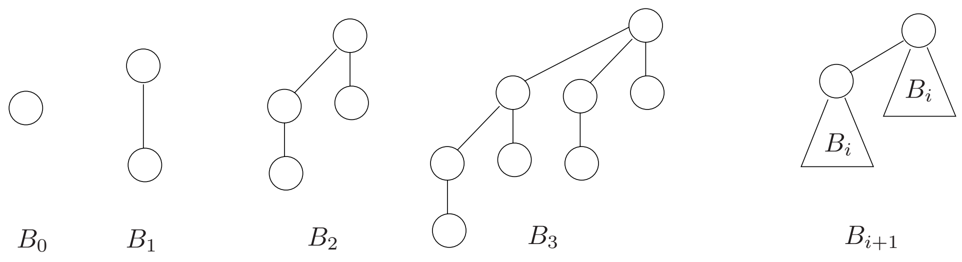


Figure 5: Binomial trees

The next lemma shows some nice properties of binomial trees (they are easily left as an Exercise).

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

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

(ii) 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.
(iii) B_i has 2^i nodes.

Property (ii) is the reason for the name "binomial trees".

A self-reproducing property of binomial trees. M. 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 6 shows an instance of $B_{3,1}$.

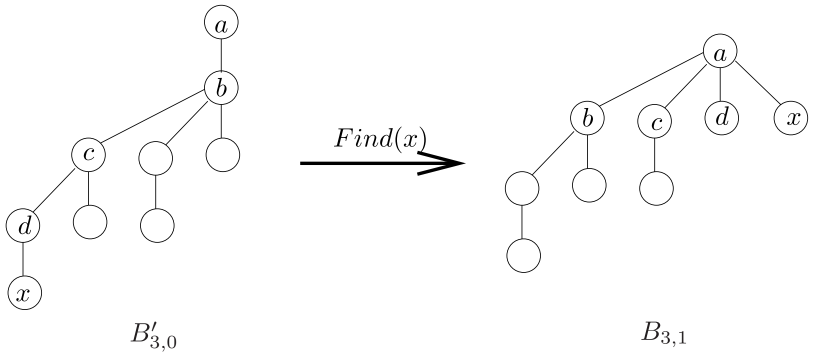


Figure 6: Self-reproducing binomial tree B_3.

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 6 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 $B_{i,k+1}$.*

This lemma is illustrated in figure 6 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).

Compaction Heuristics. Path compression can be seen as a member of the family of **path compaction heuristics**. A path compaction heuristic is any heuristic in which, along a find-path, we modify the parent pointer of each node z along that path to point to some ancestor of z. If z is the root or a child of the root, then path compaction never change $p(z)$.

The “trivial path compaction heuristic” is the one that never changes $p(z)$ for all z along a find path. Let us note two other members of this family introduced by van Leeuwen and van der Weide. The **splitting heuristic** says that each node z along the find-path, $p(z)$ should next point to $p(p(z))$. The effect of this is to split the find-path into two paths of half the original length, comprising the odd and even nodes, respectively. The **halving heuristic** says that for every other z along the path, we make $p(z)$ point to $p(p(z))$. This again has the effect of halfing the depth of each node on the find-path. The advantage of both these heuristics over path compression is that they are single-pass algorithms. Moreover, their performance matches the path-compression heuristic.

Exercises

Exercise 2.1: 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 12 Link/Find operations that achieves the worst case cost.
(b) Suppose the cost of a Find is *twice* the number of nodes in the find-path (this seems more realistic in any implementation). This obviously impacts the actual worst case cost, but would your sequence of 12 operations in (a) have changed? ♢

Exercise 2.2: Give a compact proof that when the size heuristic is used, then height of x is at most $\lg(size(x))$. ♢

Exercise 2.3: 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 preallocate $\mathcal{O}(\log\log n)$ bits to each node.] ♢

Exercise 2.4: Prove lemma 2 and lemma 3. ♢

Exercise 2.5: (Chung Yung) Assuming naive linking with the path compression heuristic, construct for every $i \ge 0$, 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 2.6: In the lower bound proof for the path compression heuristic, we have to make the following basic transformation:

$$B_{i,k} \longrightarrow B'_{i,k} \longrightarrow B_{i,k+1} \qquad (2)$$

Beginning with $B_{i,0}$, we repeat the transformation (2) to get an arbitrarily long sequence

$$B_{i,1}, B_{i,2}, B_{i,3} \ldots \qquad (3)$$

(a) The question is: can you always choose your transformations so that of the trees in this sequence has the form $B_{i+1,k}$ (for some $k \ge 0$)? (a) 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 $i = 2$. If you like, you can restrict the transformation (2) in the sense that the node for Find is specially chosen from the possible candidates.

♢

Exercise 2.7: 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 $\mathcal{O}(m)$. $\diamondsuit$

Exercise 2.8: 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 stucture to store this information about the heaviest components and which can be updated. Analyze the complexity of your algorithm.

$\diamondsuit$

Exercise 2.9: 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

$$t_1, t_2, \ldots, t_d \tag{4}$$

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

$$s_0, s_1, \ldots, s_p, \quad p = \lfloor (i+1)/k \rfloor \tag{5}$$

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 (4) and (5) are called the **horizontal** and **vertical decompositions** of the respective trees. Also, s_p in (5) 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 $m = \Omega(n^2)$. $\diamondsuit$

Exercise 2.10: Explore the various compaction heuristics. $\diamondsuit$

§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.

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 levei i partition. For $i \geq 1$, let

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

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

A simple example of a multilevel partition function is

$$A^*(i,j) = j2^i, \qquad (i,j \geq 0). \tag{6}$$

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

[SHOW FIGURE ILLUSTRATING A^*]

Basic Goal. Our goal is to analyze path compaction heuristics (§2), *without necessarily assuming that the rank heuristic is used*. Recall the rank function is defined as in equation (1), and 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 this information in our data structure. In this analysis, we assume some fixed sequence

$$r_1, r_2, \ldots, r_m \tag{7}$$

of m Links/Find request on a universe of n items. Note that up to constant factors in our analysis, we could also view (7) as a sequence of Link/Find requests.

The eventual rank of an item. The **rank** of an item is monotonically non-decreasing with time. Relative to the sequence (7) 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 $erank(x)$ to refer to its eventual rank. For emphasis, we say "current rank of x" to refer to the time-dependent concept of $rank(x)$.

The level of an item. The level notion is defined relative to any choice of a multilevel partition function $A(i,j)$, Notice that $A(i,j)$ is used purely in the analysis, and does not figure in the actual 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 $p(x)$ 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 $A^*(i,j)$ in (6), if $erank(x) = 2$ and $erank(p(x)) = 5$ then $level(x) = 3$ (since $2, 5$ both belong in the level 3 block $[0, 7]$ but are in different level 2 blocks).

Although $erank(x)$ is fixed, $p(x)$ can change and consequently the level of x may change with time. Notice that if x is a root if and only $erank(p(x)) = erank(x)$. Hnece 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, $erank(p(x))$ is non-decreasing with time.
(iii) The level of node x is 0 iff x is a root.
(iv) The level of any x 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.

Bound on the maximum level. Let $\alpha \geq 0$ 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 (6) 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

$$\alpha = \lg\lg(2n). \tag{8}$$

Charging scheme. We charge one unit for each Link operation. 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 $i \geq 1$, 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 $p(y) \neq root$ and$level(p(y)) \geq level(y)$ then we **charge one unit to y's account at level** i for each $i \in [level(y)..level(p(y))]$.

Thus y incurs a total debit of $level(p(y)) - level(y) + 1$ units over all the levels.

Justification for charging scheme.

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

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 in figure 7, this is a discrete set $v_1, v_2, \ldots, v_h$ of points which are connected up in order of increasing i. Note that by lemma 4(v), the level of v_h is 0.

The graph is a polygonal path with $h-1$ edges. An edge is called **positive, zero** or **negative**, according to the sign of its slope from left to right. Our level charges amount to charging $k+1$ units to a non-negative edge whose slope is $k \geq 0$; negative edges have no charges. In figure 7, edge (v_8, v_9) has slope 2 and so has a level charge of 3. But each negative edge can be matched with any positive edge that it can "horizontally see across the valley". (There may be several that it can see.) Thus both negative edges (v_5, v_6) and (v_6, v_7) can see the positive edge (v_8, v_9). The zero edge (v_7, v_8) does not see (v_8, v_9), by definition. Note that a non-negative edge that is charged $k+1$ units may be seen by no more than k negative edges, and so the $k+1$ units can pay for work associated with these $k+1$ edges. The negative edges that are **not yet** accounted for are those that can only see to infinity (these can be associated to those x_j's whose level is the last one along the find-path at this level number). There are at most α of these, and they can be paid for with the Upfront charge.

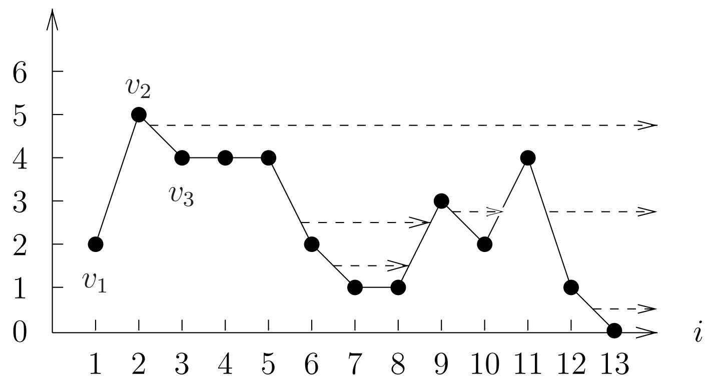


Figure 7: Plot of $level(x_i)$ against i.

The number of level charges. Let x be an item. For each $i \geq 1$, $erank(x)$ belongs to $block(i, j(i))$ for some $j(i)$.

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

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

$$level(p(x)) \geq i$$

holds just preceding that Find. This means the *erank*'s of the parent $p(x)$ and grandparent $p^2(x)$ belong to different $i-1$st level blocks. As *erank*'s are strictly increasing along a find-path, after this step, the new parent of x has *erank* in a new $i-1$st level block. But the index of these $i-1$st level blocks of $p(x)$ are nondecreasing. Thus, after at most $b(i, j(i))$ charges, $erank(p(x))$ and $erank(x)$ 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 *erank*'s lie in $block(i, j)$. Clearly, for any i,

$$\sum_{j \geq 0} n(i, j) = n. \tag{9}$$

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

$$\sum_{i=1}^{\alpha} \sum_{j \geq 0} n(i, j) b(i, j). \tag{10}$$

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 (9) and (10), is at most

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

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 $m' \ge n$). One application of this result has been pointed out by Tarjan [5] (the application allows path compression but not the rank heuristic).

Remarks:: The above definition of level is actually devised to work for any of the compaction heuristics noted in §2. The following exercise shows that a slightly simpler notion of level will work for path compression heuristic.

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 *erank*s in the same ith level block. Give a charging scheme and upper bound analysis for path compression using this definition of level.

$\diamondsuit$

§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 (8)). 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.

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

$$A(i,j) = A(i-1, A(i,j-1)) \tag{11}$$

for i, j large enough. A particular Ackermann function depends on the choice of boundary conditions. In our case, these are:

$$\begin{array}{rcll} A(i,0) &=& 0, & i \ge 0, \\ A(0,j) &=& j, & j \ge 0, \\ A(1,j) &=& 2^j, & j \ge 1, \\ A(i,1) &=& A(i-1,2). & i \ge 2. \end{array}$$

For $i \ge 2$ and $j \ge 2$, the recurrence (11) holds. It follows that

$$\begin{array}{rcl} A(2,j) &=& \begin{cases} A(1,2) = 2^2, & j = 1 \\ A(1, A(2,j-1)) = 2^{A(2,j-1)} & j \ge 2 \end{cases} \\ &=& \exp_2^{(j)}(2) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \end{array}$$

(a stack of $j+1$ 2's). See figure 8. 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 this function is its explosive growth – in fact, it grows faster than any primitive recursive function. It is not hard to verify:

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

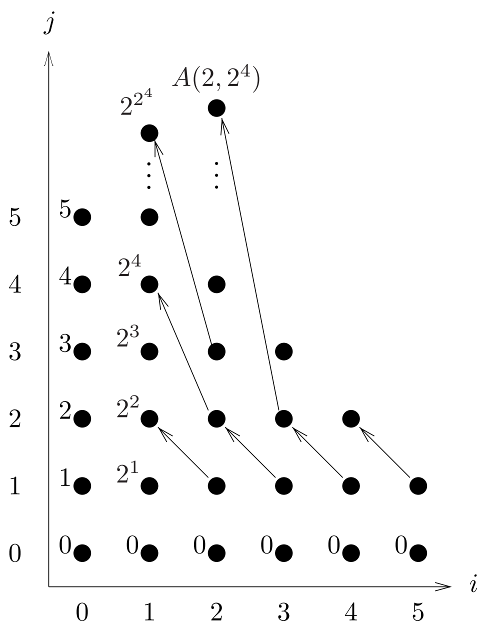


Figure 8: Ackermann's function

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 $i \ge 1, j \ge 1$, 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, $\alpha(n) \le 4$ (see exercise below).

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

$$b(i,j) = \begin{cases} 2 & \text{if } j = 0, \\ A(i,j) - A(i,j-1) & \text{if } j \ge 1. \end{cases} \tag{12}$$

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

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

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*

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

The other operations are at most $\mathcal{O}(m)$.

Generalized Amortization Framework. In Lecture V, we presented the potential framework. The present analysis suggests a different framework: assume that we are given a sequence $p_1, \ldots, p_m$ of requests to process. Our amortization scheme first 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 $Charge(A_j, p_i)$ is charged to A_j by p_i, we require

$$\sum_{j=1}^{s} Charge(A_j, p_i) \geq Cost(p_i). \tag{13}$$

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 (13) must be generalized to

$$\sum_{j=1}^{s} (Charge(A_j, p_i) - \Delta\Phi(A_j, p_i)) \geq Cost(p_i). \tag{14}$$

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:
i) $A(i, j)$ is strictly increasing in each coordinate.
ii) See the definition of $\alpha(m, n)$ in the previous exercise. 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 4.2: (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 $\alpha(\ell)$ in our text is equal to $\alpha(\ell, \ell)$. Note that $\alpha(k, \ell)$, for fixed ℓ, is a decreasing function in k. Verify that $\alpha(k, \ell) \leq \alpha(\ell)$. Improve the upper bound above to $\mathcal{O}(m\alpha(m + n, n))$. ♢

Exercise 4.3: 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 4.4: 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. ♢

§5. Three Applications of Disjoint Sets

We give three applications of the disjoint set data structure. The first problem is in the implementation of Kruskal's algorithm for MST. The second problem concerns restructuring of algebraic expressions, a problem which typically arise in optimizing compilers.

1. Implementing Kruskals' Algorithm. In Lecture IV, we presented Kruskal's algorithm for MST as one instance of the general MIS algorithm for matroids. At that time, its detailed implementation was omitted. We now show how to implement it efficiently using any Links/Find data structure.

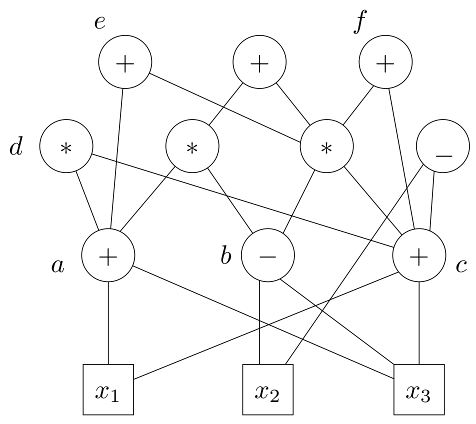
Recall that the problem is to compute the MST of an edge-costed graph $G = (V, E; C)$, with edge costs $C(e), e \in E$. The idea is to sort all edges in E in non-decreasing order of their costs. We initialize $S = \emptyset$ and at each step, we consider the next edge e in the sorted list. We will insert e into the set S provided this does not create a cycle in S (otherwise e is permanently discarded). Thus, inductively, S is a forest. When the cardinality of S reaches $|V| - 1$, we stop and present 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 of vertices which are initially pairwise non-equivalent. The algorithm makes at most $3m$ Link/Find requests, where m is the number of edges. We may assume $m \geq n$. The amortized cost is

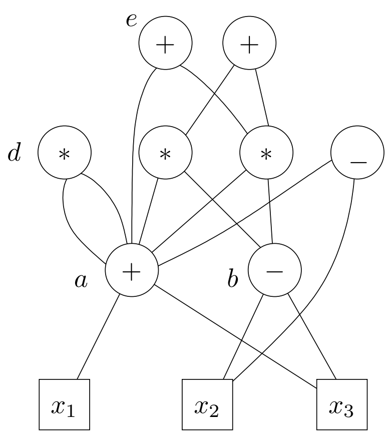
$$\mathcal{O}(m\alpha(n)).$$

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

2. Expression Optimization Problem. Suppose you are given a directed acyclic graph (DAG) G 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 9, 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 $x_2 - x_3$.



(a) G



(b) G'

Figure 9: Two DAGs: G and G'

Furthermore, 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 are indistinguishable. This is because $x - y$ is different from $y - x$, and $x \div y$ is different from $y \div x$, while $x + y = y + x$ and $x * y = y * x$. Also, this is actually a multigraph because it is possible that there there are two edges from a node d to another node a (e.g., see the graph G' in figure 9(b). Also, the Left/Right order of edges are implicit in our figures.

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 9(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 9(a) as input, the reduced graph is G' in figure 9(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 9(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 representive 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: somputing 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 $O(m + n \log n)$.

3. Computing Betti Numbers. Suppose we have a triangulation K in $\mathbb{R}^3$. This means that K is a set of tetrahedras, triangles, edges and vertices. For completeness, let K^+ denote $K \cup \{s_0\}$ where s_0 is the empty set. We call K^+ an augmented triangulation. Each $s \in K$ is a closed subset of $\mathbb{R}^3$. For K to be a **triangulation**, we require that if $s, s' \in K$ implies $s \cap s' \in K^+$. Delfinado and Edelsbrunner shows how we can compute the i-th Betti number of K using Union Find as follows.

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 nonconstant time. Another variation of this problem is to ensure that each request has a good worst case complexity. See [2, 3].

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

Exercises

Exercise 5.1: Hand simulate Kruskal's algorithm on the graph in figure 10. 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 we want (i, j) to appear before (i', j') iff (i, j) is less than (i', j') in the lexicographic order, i.e., either $i < i'$ or ($i = i'$ and $j < j'$).

(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-fine data structure. ♢

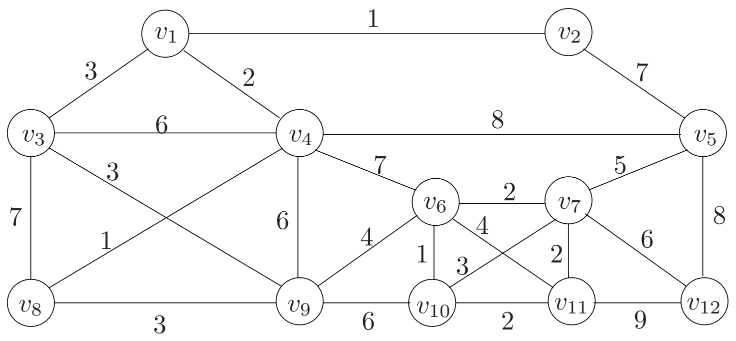


Figure 10: Another House Graph

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 §IV.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.

ANSWER:

We use depth-first search. Initially color all nodes u as "unseen" and assign $h(u) = 0$. For each node u in the graph, if u is "unseen", call $DFS(u)$. We define $DFS(u)$ as follows:

```
DFS(u)
    Color u "seen".
    For each v adjacent from u,
        If v is "unseen", DFS(v)
        $h(u) = \max\{h(u), 1 + h(v)\}$ (*)
```

The running time is $O(m+n)$ as in standard DFS. Why is this correct? It is enough to show that $h(u)$ is the height at the end. It is clear that sinks have $h(u) = 0$ since there are no nodes that are adjacent FROM a sink. Inductively, if every node v of height less than k has the correct value of $h(v)$, then the assignment to $h(u)$ in (*) is correct.

♢

End Exercises

References

[1] A. M. Ben-Amram and Z. Galil. Lower bounds of data structure problems on rams. *IEEE Foundations of Computer 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] M. H. M. Smid. A datastructure for the union-find problem having a good single-operation complexity. *Algorithms Review*, 1(1):1–12, 1990.

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

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

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