CS 124 Section #4 **Union-Find, Greedy Algorithms** 2/20/17

1 Disjoint-set data structure.

1.1 Operations

Disjoint-set data structure enable us to efficiently perform operations such as placing elements into sets, querying whether two elements are in the same set, and merging two sets together. (Thus they're very useful for simulating graph connectivity.) Must implement the following operations:

- MAKESET(x): create a new set containing the single element x.
- UNION(x, y): replace sets containing x and y by their union.
- FIND(x): return name of set containing x.

We add for convenience the function LINK(x, y) where x, y are roots: LINK changes the parent pointer of one of the roots to be the other root. In particular, UNION(x, y) = LINK(FIND(x), FIND(y)), so the main problem is to make the FIND operations efficient.

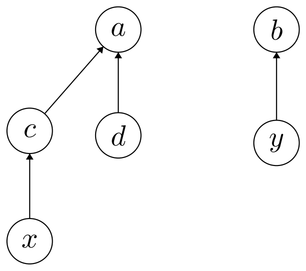
1.2 Optimization Heuristics

We have two main methods of optimization for disjoint-set data structures:

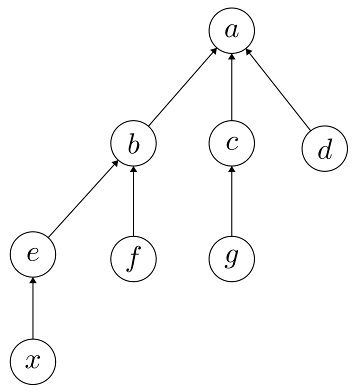
- **Union by rank.** When performing a UNION operation, we prefer to merge the shallower tree into the deeper tree.
- **Path compression.** After performing a FIND operation, we can simply attach all the nodes touched directly onto the root of the tree.

Exercise 1. Draw how the disjoint set data structure changes after each of the following operations using a particular heuristic:

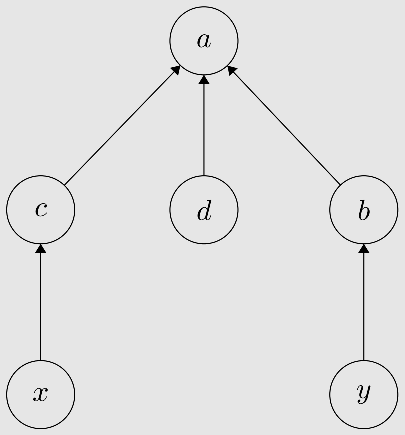
(a) UNION(x, y) with union by rank.

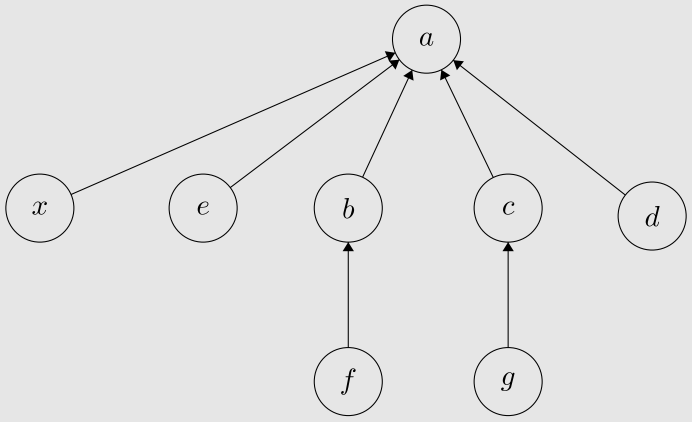


(b) FIND(x) with path compression.



Solution





Exercise 2. When using the union by rank optimization *only*, what is the asymptotic runtime of the operation FIND(x)?

Solution
When we use the union by rank heuristic, we have the property that the rank of a tree only increases when we union together two trees of the same rank. (Recall that rank = height). If you union a tree with rank 3 with one of rank 4, you get a tree of rank 4. Only when you union two trees of rank 4 can you get a tree of rank 5.

With this property, we can show that a tree of rank h has at least 2^h nodes. This can be proved using induction.

Base Case: When the rank of a tree is 0, there is always 1 node, which is indeed at least 2^0.

Inductive Case: Suppose it is true that every tree of rank h has at least 2^h nodes. If we had a tree of rank $h+1$, we know that it must have come about through the union of two trees of rank $h+1$ at some point in the history of this tree. By the inductive hypothesis, those two trees of rank h must have had $\geq 2^h$ nodes each, and thus the number of nodes in this tree of rank $h+1$ must be at least $2^h + 2^h = 2^{h+1}$.

Therefore, we have that the run-time of FIND(x) when using only the union by rank heuristic is $O(\log n)$ because if there are n nodes, then the maximum rank of any tree is $\log n$.

2 Greedy Algorithms

A **greedy algorithm** is an algorithm which attempts to find the globally optimal solution to a problem by making *locally optimal* decisions.
Examples from Lecture:

- **Kruskal's:** We greedily choose the lightest edge in the graph that doesn't form a cycle.
- **Prim's:** We greedily choose the lightest edge adjacent to the spanning tree we've formed so far.
- **Horn Formula:** We start by assigning all variables to FALSE and only set variables to true when an implication forces you to.
- **Huffman Encoding:** We greedily construct the encoding by taking the two least used characters and merging them into one new character.
- **Set Cover** ($O(k \log n)$ approximation): We greedily choose the set that covers the most number of the remaining uncovered elements at the given iteration.

In order to prove that a greedy algorithm is correct, we need to show that indeed choosing the locally optimal solution keeps us *on track* to finding the globally optimal solution. The cut property with MST's guarantees that the greedy MST algorithms are correct.

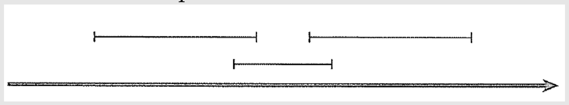
Problems with correct greedy algorithms are rare. We will soon be introduced to **Dynamic Programming**, a technique that lets you solve a much wider range of optimization problems. It is arguably the most useful technique you will learn in CS 124.
Exercise 3. Suppose that we have a set $S = \{a_1, \ldots, a_n\}$ of proposed activities. Each activity a_i has a start time s_i and a finish time f_i. We can only run one activity at a time. Your job is to find a maximal set of compatible activities. Which of the following greedy algorithms is correct?

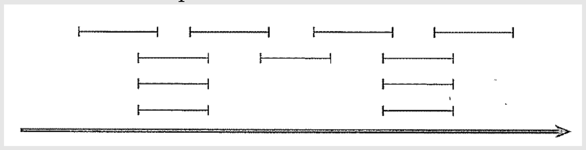
(a) Sort all the activities by their duration and greedily picking the shortest activity that does not conflict with any of the already chosen activities.

(b) Pick the activity that conflicts with the fewer number of remaining activities. Remove the activities that the chosen activity conflicts with. Break ties arbitrarily.

(c) Sort all the activities by their end time and greedily pick the activity with the earliest end time that does not conflict with any of the already chosen activities.

Solution

(a) *Incorrect.* It's not always best to choose the shortest length activities
Counter-example:


(b) *Incorrect.* This one is tricky, but it's not always best to choose the activity with the least a mount of conflicts.
Counter-example:


(c) *Correct.*

Proof: Let g_1 be the first activity chosen by the greedy solution and let t_1 be the first activity by the optimal solution. We know that based on our selection technique, g_1 is has the earliest end-time out of all the activities, and thus the end time of t_1 must be equal to or after that of g_1. A similar argument can be made for t_2 and g_2. We know that t_2 does not overlap with g_1 because g_1 ended before t_1 ended. Therefore t_2 must end at the same time or later than g_2. This argument can be made inductively to show that the end time of g_i is always earlier than the end time of t_i.

Why does this imply that the greedy algorithm is optimal? Imagine a situation where we have one more activity in the optimal solution than in greedy. Let t_ℓ be the last event chosen by the optimal solution, but suppose that greedy only goes up to $g_{\ell-1}$. Well we know by the informal induction we did that the end time of $g_{\ell-1}$ is no later than the end time of $t_{\ell-1}$. This implies that t_ℓ does not conflict with $g_{\ell-1}$. So why did the greedy algorithm not choose t_ℓ after choosing $g_{\ell-1}$? Contradiction.

Exercise 4. Let's go back to *greedy.c* from the first CS 50 problem set. The question was to determine the fewest number of US coins necessary to make change for a given amount of money.

(a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

(b) Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Solution

(a) Our greedy algorithm will be to use the largest coin at each iteration. Suppose we are making change for n cents.

- If $n < 5$, then greedy will use n pennies which is optimal.
- If $6 \leq n < 10$, then greedy will use one nickle and then $n - 5$ pennies, which is also optimal.
- If $11 \leq n < 25$, then greedy will use a dime first. Can the optimal solution not use any dimes? Notice that the optimal solution will never use more than 4 pennies or 1 nickel (because otherwise you could just always replace some of them with a nickle or a dime). 4 pennies and a nickle is only 9 cents, so it is definitely sub-optimal to not use a dime first when n is in this range.
- If $n \geq 25$, by a similar reasoning as before, we will never use more than 2 dimes because if you use 3 dimes, you might as well replace it with a nickle and a quarter. Therefore, if $n > 29$, then certainly we will use a quarter first. You can easily check that if n is between 25 and 29, then taking the quarter and then $n - 25$ pennies is optimal.

(b) We will *generalize* the argument used in the example above. The important realization was that you will never use more than $c - 1$ copies of a particular coin because otherwise you should replace c copies of that coin with 1 copy of the next higher coin. The exception is of course for the largest coin. Therefore, for any denomination n, let c^m be the size of the largest denomination less than n. If I only used coins of denomination c^{m-1} and less, while using only at most $c - 1$ copies of each, the maximum amount of money I could make change for is:

$$\sum_{i=0}^{m-1} (c-1) \cdot c^i = (c-1) \cdot \frac{c^m - 1}{c - 1} = c^m - 1$$

Given our assumption that $n \geq c^m$, we know that it is impossible to have an optimal solution that doesn't use the coin with value c^m.

(c) Suppose we only had pennies, dimes and quarters. Then, the optimal solution for $n = 31$ would be to take 3 dimes and a penny (4 coins total), but the greedy algorithm would take 1 quarter and 6 pennies (7 coins total). Note that in this case, it was best to take a coin with smaller value first (the penny) rather than the one with the highest value (the quarter).

3 Divide and Conquer

Divide and Conquer algorithms work by recursively breaking the problem into smaller pieces and solving the subproblems. You need to think about how to recursively break down the problem as well as how to combine the various pieces together.

Examples from Lecture:

- **Mergesort and StoogeSort:** We recursively sorted a fraction of the list and then did some work to combine those fractions of the list together.

- **Integer Multiplication:** We split two integers in half each, forming 4 pieces and then performed only 3 multiplications on half-sized numbers.
- **Strassen's Algorithm:** We perform matrix multiplication on two $n \times n$ matrices by performing 7 multiplications on $n/2 \times n/2$ matrices.

Exercise 5. Given an array of n elements, you want to determine whether there exists a majority element (that is an element which occurs at least $\lceil \frac{n+1}{2} \rceil$ times) and if so, output this element. Show how to do this in time $O(n \log n)$ using Divide and Conquer.

Solution
Split the list into 2 halves and recursively find the majority element of each half. If both sides return "no majority", then return "no majority". This is true because if there exists a majority element, then it also has to be the majority on at least one of the two. If both sides return an element, check to see if the elements are the same and if so return it. Otherwise, for each element returned by the recursive call, do a linear time traversal through the entire list to see if it is the majority element. Return it if it is and "no majority" if you don't detect a majority element this way.

It is not hard to convince yourself that this algorithm is correct, but what is its run-time? We make 2 recursive calls on lists of size $n/2$ and then perform up to 2 linear traversals through the list. Thus:

$$T(n) = 2T(n/2) + O(n)$$

which solves to $O(n \log n)$ just like mergesort.

Exercise 6. (2015 Problem Set 4) You are given an n-digit positive integer x written in base-2. Give an efficient algorithm to return its representation in base-10 and analyze its running time. Assume you have black-box access to an integer multiplication algorithm which can multiply two n-digit numbers in time $M(n)$ for some $M(n)$ which is $\Omega(n)$ and $O(n^2)$.

Solution
Consider the n-digit base-2 integer x as a string of base-2 digits, $x[0...n]$. Assume n is a power of 2. If not, we can pad the left side of the input integer with zeros. This will increase the number of digits by less than a factor of 2, which will not affect the big-O runtime.

We offer a recursive algorithm. In the base case, for an input of size 0, output 0. Divide the integer into two halves, $x[0...\frac{n}{2}]$ and $x[\frac{n}{2}...n]$. Then, recursively convert each half into a base-10 integer. Call these (converted) base-10 integers y_ℓ and y_r. Then, the desired integer can be found by computing $y_\ell \cdot 2^{n/2} + y_r$. Note that, $p = 2^{n/2}$ must be in base-10 also. One way we could get the base-10 representation of p is by converting it recursively, too, along with the halves of the input integer. Then we make three recursive calls to our procedure, each with a base-2 integer which is roughly half the size of the problem input. After we obtain these base-10 integers, we have to perform an integer multiplication $(y_\ell p)$ and an integer addition $(y_\ell p) + (yr)$.

The run-time is:

$$T(n) = 3T(n/2) + M(n/2) + O(n)$$

Suppose $M(n) = O(n^k)$ for some $1 \le k \le 2$. Then, $M(n/2) + O(n) = O(M(n/2)) = O(n^k)$. Thus, we can apply the master theorem on what k is:

If $k > \log_2 3$, then $T(n) = O(n^k)$. If $k = \log_2 3$, then $T(n) = O(n^k \log n)$ and if $k < \log_2 3$, then $T(n) = O(n^{\log_2 3})$.