Data Structures and Algorithms - Homework 6

HK Rho

March 6, 2020

Problem 1 - a

Information Provided

1. List of all the students at Olin
2. List of all the enrolled students given a class

Preprocessed Data

1. A hash-map of all the classmates to a particular student.
   Key = given student
   Value = all the other students in all the classes the given student takes.
   e.g. {student0: student1, ... student$x$,
   student1: student4, ... student$z$, ...}
2. A hash-map of true/false values to each student to record their status. The values of all the students are all initially set to False. This hash-map can be constructed since we are provided with the list of all the students at Olin.
   e.g. {student0: False,
   student1: False, ...}

Algorithm

This algorithm for this problem with an O(n) run-time is implemented with a combination of the preprocessed data and a queue to keep track of all the potentially infected students.
Given the patient zero and the classes that this patient takes initially, we enqueue patient zero to the queue.

Afterwards, we dequeue patient zero from the queue and iterate through all the students that patient zero has interacted with in all their classes. Put in other words, this process iterates through all the values of the first hash-map that stores all the classmates of a given student, with the key being patient zero. During this iteration, each student/value is checked if their infected status is set to true or false in the second hash-map that stores true/false values to each student. If the value of the student is set to true, this student is not enqueued to the queue. However, if the student is set to false, this student is enqueued to the queue and the infected status is updated to be true.

Structure of Algorithm

```
# used for storing all the potentially infected students
l = []
l.enqueue()
while len(l) > 0:
    current_student = l.dequeue
    for (all the classmates a given student interacted with):
        if the classmate is not in l:
            add to l
            change false -> true

```

Run-time

This algorithm satisfies the $O(n)$ run-time, as going through all the students a given student as interacted with and therefore has potentially infected takes $O(1)$ run-time and since looping through all the students at Olin takes $O(n)$ run-time. Iterating through all of the classmates that a given student has interacted with takes an $O(1)$ run-time because the number of classmates that student could have interacted with is at most $5 * 25 = 125$ students. (Utilizing the given information that there are at most 25 students in a class and a student takes no more than 5 classes).

Problem 1 - b

Suppose that a student A caught academitis but was not added to the list of potentially infected students. This would only result from a fault in the information that is provided to us prior to solving the problem (the list of all the enrolled students given a class). This fault in the list continues in the first hash-map that stores all the classmates of a given student, resulting in a wrong hash-map. Using this erroneous hash-map leads us to fail updating the true/false value for that student A properly in the second hash-map under the preprocessed data. Because we are given the second piece of information under the information provided that is reliable, it is impossible to have a student A (who is potentially infected) that would mistakenly not be added to the list of potentially infected students.

Problem 2

Base Case

If the length of a sub-array is equal to 1, then that sub-array is sorted.

Inductive Hypothesis

Assuming that a merge sort applied on a list of length $n/2$ returns a sorted list, then a merge sort applied on a list of length n also returns a sorted list.

Inductive Step

Given that $n = 2$, and therefore $n/2 = 1$, $n/2$ is guaranteed to be sorted because this case satisfies our base case. When these two sub-arrays of length $n/2 = 1$ are merged together by comparing all the values from each sub-arrays, this results in a joined sorted sub-array. The process of merging is the same as an insertion sort, where each element in one array is compared to all other elements in the other array until the smaller one is found. Because the case above holds true by returning a sorted sub-array, the merging of the sub-arrays with longer lengths is also guaranteed to be sorted, since all the arrays/sub-arrays with length greater than two will eventually end up as the above base case. Therefore, the inductive hypothesis holds true.

Problem 3 - a

make_set(i)

Given a node i, the run-time for making a set with this single node is $O(1)$ because this node simply has to be either set as the head node by pointing the head to this node i. If this node i had to be attached to an existing head node or its child nodes, This would also take an $O(1)$ run-time since only the appropriate pointers (e.g. next, prev) has to be updated.

union(A, B)

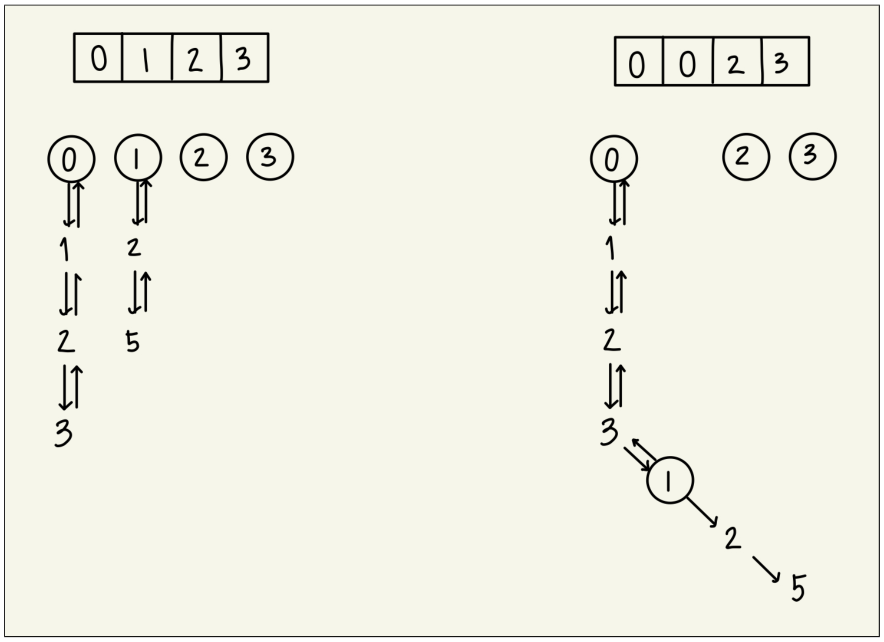

Figure 1: Visualization the structure and process for union-find when the run-time of union(A,B) is $O(min(n_A, n_B))$ with n_A being the number of elements in A and n_B being the number of elements in B.

To achieve the run-time $O(min(n_A, n_B))$ when merging two sets A and B, we attach the set with the smaller number of nodes to the other set. When adding each node from the smaller set to the larger set, we update the parent node of each node (originally in the smaller set) from the head node of the smaller set to the head node of the larger set. For example, we are given two doubly-linked lists where 0 is the head node for one and 1 is the head node for the other.

$$[0, 1, 2, 5] \; and \; [1, 2, 3]$$

When we merge these two doubly-linked lists together, then the doubly-linked list that has 1 as its head node is attached to the end of the other doubly-linked list. During this attachment, the parent node of 1, 2, and 3 are each updated to be 0. A graphical overview of this process is summarized by Figure 1.

find(i)

Before getting into the part of the algorithm that finds the head of a given node i in an $O(1)$ run-time, each node will store the head of the doubly-linked list that it belongs to by storing the head node of a given node in the node class. Storing the head node enables us to simply query the head node given the node i, which is an $O(1)$ run-time.

Problem 3 - b

make_set(i)

This operation will work the same as the make_set from Part *a* above; therefore, the run-time is $O(1)$ in this case as well.

Given a node i, the run-time for making a set with this single node is $O(1)$ because this node simply has to be either set as the head node by pointing the head to this node i. If this node i had to be attached to an existing head node or its child nodes, This would also take an $O(1)$ run-time since only the appropriate pointers (e.g. next, prev) has to be updated.

union(A,B)

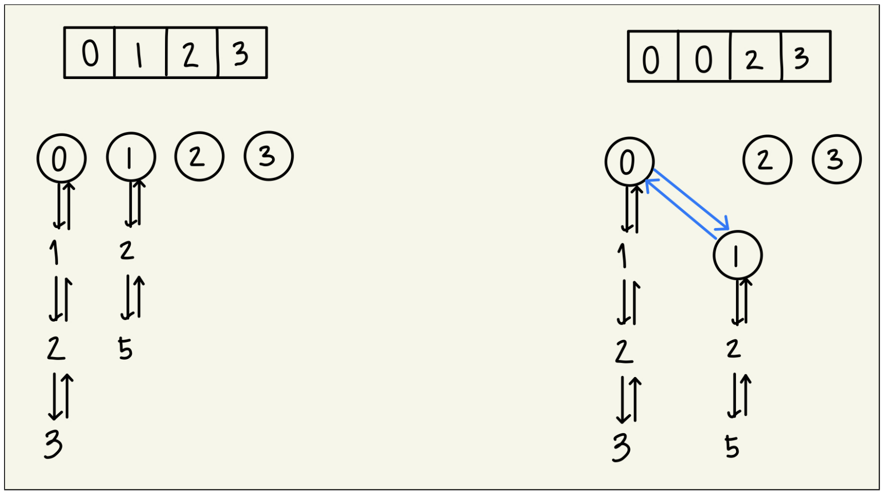

Figure 2: Visualization the structure and process for union-find when the run-time of union(A,B) is $O(1)$

To achieve the run-time $O(1)$ for merging two sets A and B, we simply set the pointers between the two sets or doubly-linked lists so that the longer set owns the shorter set as its child. A graphical overview of this process is encapsulated in Figure 2. If we union the two sets that each have 0 and 1 as their head nodes, then the set with 1 as the head node now falls as a child under the set with 0 as the head node. Deciding which set will become the child of another set will depend on how the union method is constructed.

find(i)

To find the head of a given node i, we would recurse over the parents starting from node i, which results in a run-time of $O(log\ n)$ (i.e. Given node i, we would find the parent of node i, parent of the parent node, and so on). Since there is no limit to the number of children that a node can have, the base of the log from the run-time is the maximum number of children that any node has. This is different from binary trees with 2 as the base for the log for the run-time.