Assignment 2

Kelly Jo Law and Riley Primeau

October 6, 2022

1 Graphs

1.1 Consider the following graph.

a. What is the cost of its minimum spanning tree?
The cost of this graph's minimum spanning tree is 19.

b. How many minimum spanning trees does it have?
Using Kruskal's algorithm, this graph has 1 unique minimum spanning tree.

c. Suppose Kruskal's algorithm is run on this graph. In what order are the edges added to the MST? For each edge in this sequence, give a cut that justifies its addition.
The order that the edges would be added via Kruskal's algorithm is as follows:

(a) A - E
(b) E - F
(c) E - B
(d) F - G
(e) G - H
(f) G - C
(g) G - D

We can justify the addition of these edges with the following cuts (using — to symbolize the cut):

Edge Added	Cut to Justify Addition
A-E	A B C D — E F G H
E-F	A B C D E — F G H
B-E	A E F — B C D G H
F-G	A B E F — C D G H
G-H	A B E F G — C D H
C-G	A B C D — E F G H
G-D	A B C D H — E F G

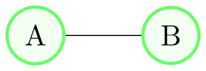
1.2 Suppose we want to find the minimum spanning tree of the following graph.

a. Run Prim's algorithm; whenever there is a choice of nodes, always use alphabetical ordering (e.g., start from node A). Draw a table showing the intermediate values of the cost array.
Intermediate Cost Values:

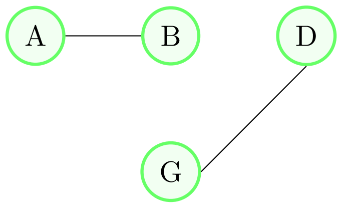
Edge	Cost of Edge	Total Intermediate Cost
A-B	1	1
B-C	2	3
C-G	2	5
G-D	1	6
G-F	1	7
G-H	1	8
A- E	4	12

b. Run Kruskal's algorithm on the same graph. Show how the disjoint-sets data structure looks at every intermediate stage (including the structure of the directed trees), assuming path compressing is used.
Disjoint-Set Data Structures:

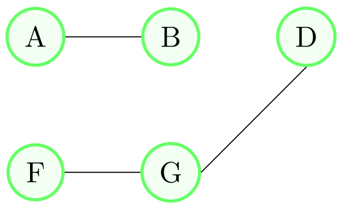
(a) Intermediate Stage 1



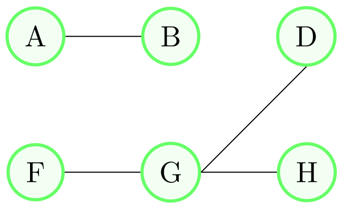
(b) Intermediate Stage 2



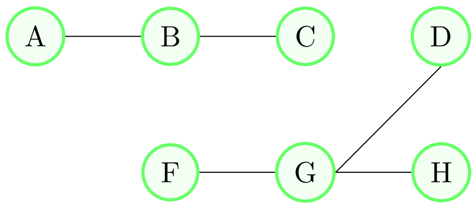
(c) Intermediate Stage 3



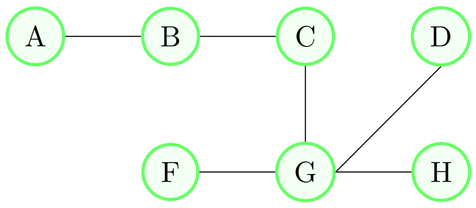
(d) Intermediate Stage 4



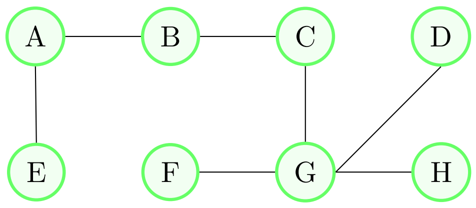
(e) Intermediate Stage 5



(f) Intermediate Stage 6



(g) Intermediate Stage 7



1.3 Design a linear-time algorithm for the following task.

Input: A connected, undirected graph G
Question: Is there an edge you can remove from G while still leaving G connected?

One algorithm that would achieve the task would start by removing an edge. Then, from any vertex, scan through the graph to find all the reachable vertices. If the number of reachable vertices is equal to the total number of vertices, then you are able to remove the edge. Otherwise, removing it would disconnect the graph. This algorithm has a time complexity of $O(V+E)$ with V being the number of vertices and E the number of edges.

Can you reduce the running time of your algorithm to $O(V)$?

The running time of the algorithm can be reduced to $O(V)$ if the first edge selected can be removed. This is because E will be a constant, making the time complexity $O(V+1)$. Since 1 is a constant, it can then be disregarded which results in the desired time complexity.

2 Cost / Complexity / Big O

Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems by dividing them into five subproblems of half the size recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by dividing them into nine subproblems of size n/3, and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into nine subproblems of size n/3, recursively solving each subproblem, and then combining the solutions in O(n2) time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

- Algorithm A

$$T(n) = n* \sum_{i=0}^{\log_2 n} (\frac{5}{2})^i = n* \frac{(\frac{5}{2})^{\log_2 n} - 1}{(\frac{5}{2}) - 1} = \frac{2n}{3}(\frac{5}{2}(\frac{n^{\log_2 5}}{n}) - 1) = \Theta(n^{\log_2 5})$$

- Algoritm B

$$T(n) = \sum_{n=1}^{\infty} 2^i c = c\frac{2^n - 1}{2 - 1} = \Theta(2^n)$$

- Algorithm C

$$T(n) = \sum_{i=0}^{\log_3 n} n^2 = \Theta(n^2 \log_3 n)$$

Out of these three algorithms, we would choose Algorithm C. This is because Algorithm B runs in exponential time so it will be the slowest of the three. Based on the Big O notation and looking at the behavior of Algorithms A and C, as n approaches infinity, we find that Algorithm C has the fastest runtime.

3 Algorithmic Efficiency

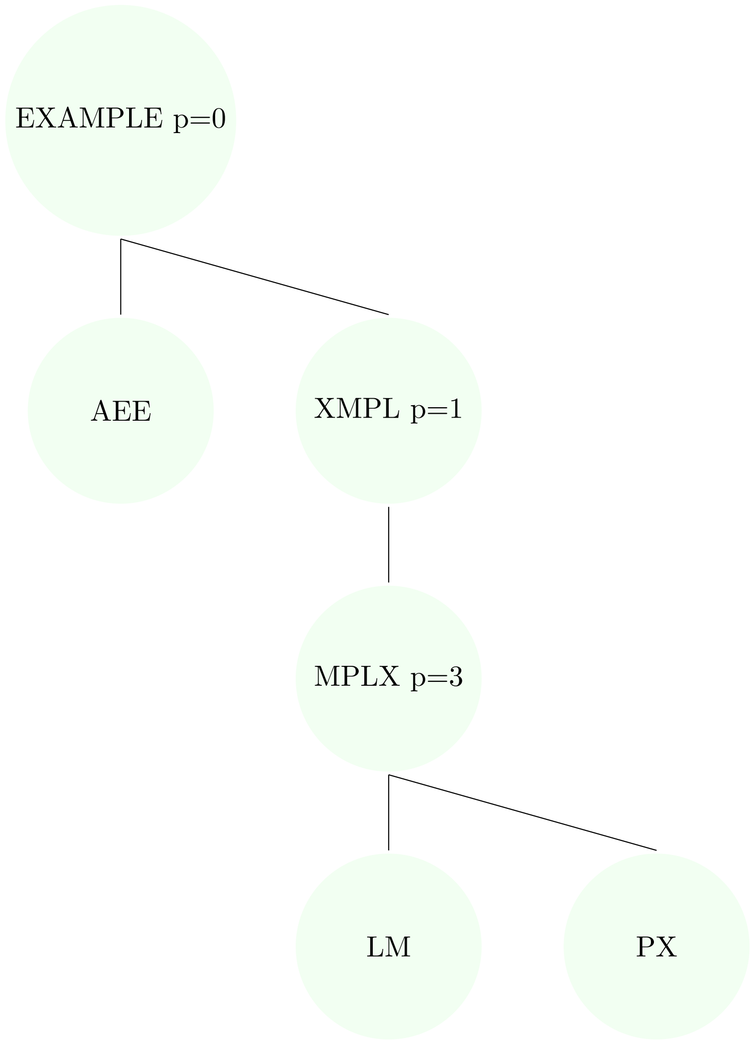
You are given an infinite array A[-] in which the first n cells contain integers in sorted order and the rest of the cells are filled with (infinity). You are not given the values of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(logn)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message (infinity) whenever elements A[i] with $i > n$ are accessed.

If given the value of n, binary search would be a solution with the desired time complexity. However, n is not known so the search area cannot be determined. In order for it to be determined, use the value you are searching for, the key, and compare it to the second element. If the key is greater than the second element, then you would change the lows and highs of the search bounds to, essentially, create a new subarray. The new bounds would start at the second element and end at that element times 2. This would continue recursively until the key is less than the upper-bound value. From here, you then have the ability to preform binary search to find the key. Since doubling the high bound acts as a method to divide the infinite array by 2 every time, that step has a time complexity of $O(logn)$. Since binary search has the same time complexity this algorithm would be $2 * O(logn)$. Discarding the constant results in a time complexity of $O(logn)$ as desired.

4 QuickSort

i. Apply QuickSort to the list E,X,A,M,P,L,E in alphabetical order. Draw the tree of the recursive calls made.

0	1	2	3	4	5	6
E	X	A	M	P	L	E



ii. For the partitioning procedure outlines in Levitin's section 5.2: Explain why if the scanning indices stop while pointing to the same element, i.e, i = j, the value t they are pointing to must be equal to p.

When the scanning indices stop while pointing to the same element, that element must be equal to p because of the criteria that determines the scans. For i, the scan begins on the left side of the array and increments, or scans, until it finds a value that is greater than or equal to the partition. On the other hand, j starts on the right side of the array and scans by decrementing until it reaches a value that is less than or equal to p. That said, if i and j are pointing to the same element, it would be impossible for that element to be both greater than and less than p. Therefore, that element must be equal to p.

iii. Explain why when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i.

When the scanning indices stop, j cannot point to an element more than one position to the left of the index of i because $i < j$ represents one of the cases where the array has almost been completely partitioned. When this case is reached, A[j] is swapped with the partition which results in the fully partitioned array. This means j will never be more than one position to the left of i. This is because this case will always be triggered at one position to the left of i, at which point j is swapped.

iv. Give an example showing that QuickSort is not a stable sorting algorithm.

One example that shows Quicksort is unstable is when there are duplicate values. In these scenarios, Quicksort has the possibility of reversing the two duplicate values since it works by swapping nonadjacent values. With some programs, this would not be an issue, however, if the sorting occurs with data such as a map or a pair, then it becomes clear that the relative order is violated which in turn makes it an unstable algorithm.

v. Give an example of an array of n elements for which the sentinel mentioned in Levitin's text is actually needed. What should be its value? Also explain why a single sentinel suffices for any input.

The sentinel mentioned in Levitin's text is not needed as much for an array with a specific n elements, but rather when all the elements in an array of any length are less than the pivot. In this case, the algorithm will continue incrementing and scanning to the right and eventually will go out of bounds. By adding a sentinel, a value equal to or greater than the pivot, to the end of the array it will prevent the scan from going out of bounds because the scan's condition will be met. That said, you only ever need one sentinel because the scan only compares against one value, the partition.

vi. For the version of QuickSort given in section 5.2 of Levitin's text: 1. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither? Worst-case
2. Are strictly decreasing arrays the worst-case input, the best-case input, or neither? Worst-case

vii. Nuts and bolts. You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency in theta(nlogn).

One way to solve this problem would be to implement an algorithm similar to Quicksort. The first step would be to select a partition from the collection of nuts. This nut could be compared to find the corresponding bolt and then used to partition the rest of the bolts based on whether they are larger or smaller then the nut. Then, you use the bolt that is an exact fit for the initial nut as a partition for the collection of nuts. With this, you then partition the collection of nuts based on whether they are smaller or larger than the selected bolt. Next, repeat this partitioning through recursion to match the rest of the pairs. Since this strategy essentially uses Quicksort twice, which has a time complexity of $\Theta(n * log(n))$, its time complexity would be double that of Quicksorts. However, multiplying by 2 is a constant so it is disregarded. This results in the average case for this algorithm being $\Theta(n * log(n))$.