# CSM 61B Spring 2019

# More MSTs & Sorting

Mentoring 10: April 8, 2019

## 1 Prim's Algorithm

The cut property states that given any cut, the minimum weight crossing edge is in the MST. The converse is also true, that if an edge is in the MST, then it must be the minimum weight crossing edge across some cut. This property is a key idea behind Prim's algorithm.

1.1 Describe Prim's algorithm.

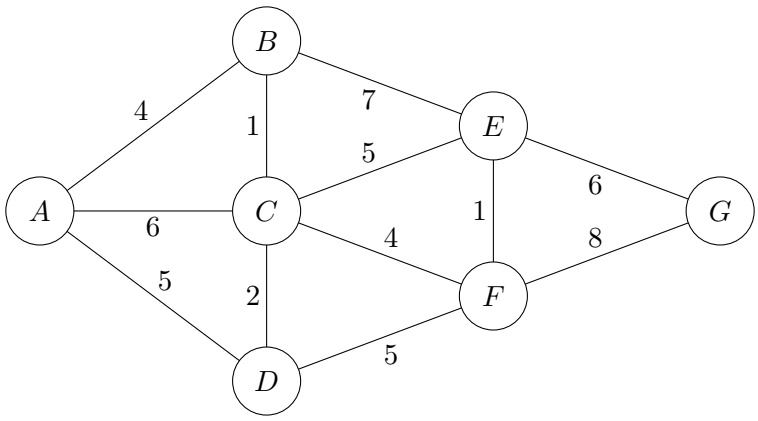
Starting from any arbitrary source, repeatedly add the shortest edge that connects some node in the tree to some node outside the tree.

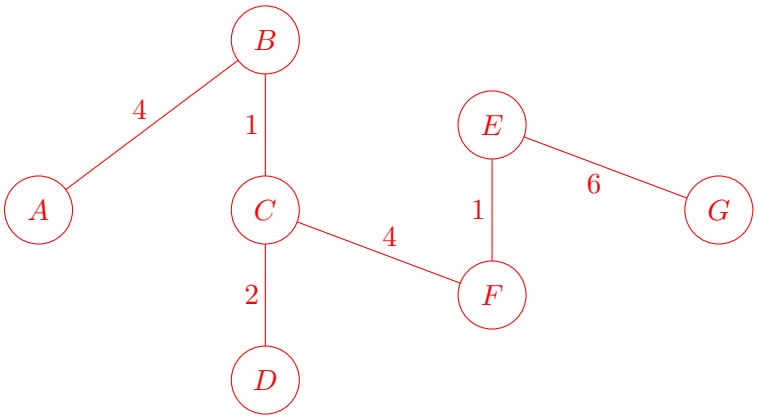
Another way of thinking about Prim's algorithm is that it is basically just Dijktra's algorithm, but where we consider node in order of the distance from the *entire tree*, rather than the distance from the start.

1.2 We can use a binary heap priority queue to implement Prim's. What would be the runtime of Prim's using this implementation?

In the worst case, we perform V insertions and deletions, each costing $O(\log V)$ time. We also decrease priorities E times, also costing $O(\log V)$ time. In total, assuming that E>V, Prim's will take $O(E \log V)$ time to complete.

1.3 Consider the telephone network from last week. Construct a minimum spanning tree by running Prim's Algorithm from node A.






**Meta**:
Intro: The cut property is introduced in the general statement for this question. Make sure to have a simple example to use as a visual when explaining it to students.

1.3: Your board work for how to run through Prim's should be very similar to how you ran through Djikstra's algorithm in the previous worksheet. Only add edges to your MST as you pop off the Fringe (PQ). Additionarlly, after running through the problem, explain how at a high level, for every edge added, we just added the shortest edge not in the MST into the MST. Use the cut property to explain this.

# 2 Maximum Spanning Trees

2.1 We have two algorithms, Kruskal's and Prim's, that allow us to find a Minimum Spanning Tree. Consider the problem of finding a Maximum Spanning Tree

(a) Describe a modification to Kruskal's algorithm that would allow us to find a Maximum Spanning Tree of a graph

Negate all the edge weights and find the minimum spanning tree using Kruskal's algorithm. Nothing in Kruskal's algorithm assumes the weights are positive. Therefore, the minimum of the weights negated, is achieved by the maximum of the original weights, and we will have a Maximum Spanning Tree.
Similar logic applies for using Prim's algorithm with negated edge weights.

(b) Can we use a similar approach to modify Djikstra's algorithm to find the Maximum Path between two nodes?

No, because Djikstra's doesn't work with negative edge weights. This is because Djikstra's relies on the assumption that if all weights are non-negative, adding an edge can never make a path shorter. Therefore, we cannot simply negate edge weights and use Djikstra's to find the Maximum path.

# 3 Feeling Out of Sorts?

So far, we've learned a few different types of basic sorting algorithms. While sorting might seem like a simple idea, there are many real-world applications of sorting, and several different algorithms that we can use depending on the situation.

In the table below, fill out the best and worst-case runtimes for each of the sorting algorithms provided.

| Algorithm | Best-case | Worst-case |
|---|---|---|
| Selection Sort | $\Theta(N^2)$ | $\Theta(N^2)$ |
| Insertion Sort | $\Theta(N)$ | $\Theta(N^2)$ |
| Merge Sort | $\Theta(N \log N)$ | $\Theta(N \log N)$ |
| Heapsort | $\Theta(N)$ | $\Theta(N \log N)$ |

[Selection Sort] In selection sort, we loop through the array to find the smallest element. Next, we swap the element at index-0 with the smallest element. Next, we repeat this procedure, but only looking at the array starting at index-1.

*Runtime, Best, Worst Case:* Since it takes $O(N)$ time to loop through the array, and we loop through the array $N$ times, this algorithm has a runtime of $\Theta(N^2)$. Note that even if the array is already sorted, we need to iterate through it to find the minimum, and then iterate through it again, and again, $N$ times.

[Insertion Sort] This is the way an adult would normally sort a pack of cards. Iterating through the array, swapping each element left-wards.

*Best Case:* Given a sorted array, `{ 1, 2, 3, 4 }`, this algorithm would iterate through the array just once, and do 0 swaps, since all elements are already as left-wards as they can be. *Worst Case:* Given a fully unsorted array, `{ 4, 3, 2, 1 }`, this algorithm would first swap $(3, 4)$, then to move 2 left-wards, it needs to do 2 swaps. Finally to move 1 left-wards, it needs to do 3 swaps. This is of the ordering of $O(n^2)$ swaps.

[Merge Sort] Given an array, divide it into two equal halves, and call merge-sort recursively on each half. Take the recursive leap of faith and assume that each half is now sorted. Merge the two sorted halves. Merging takes a single iteration through both arrays, and takes $O(N)$ time. The base case is if the input list is just 1 element long, in which case, we return the list itself.

*Best case, Worst Case, Runtime:* Since the algorithm divides the array and recurses down, this takes $\Theta(N \log N)$ time, no matter what.

[Heap Sort] Place all elements into a heap. Remove elements one by one from the heap, and place them in an array.

*Recall:* Creating a heap of $N$ elements takes $N \log N$ time, because we have to bubble-up elements. Removing an element from a heap takes $\log N$ time, also because of bubbling and sinking. *Best Case:* Say that all the elements in the input array are equal. In this case, creating the heap only takes $O(N)$ time, since there is no bubbling-down to be done. Also, removing from the heap takes constant time for the same reason. Since we remove $N$ elements, and creating the heap takes $O(N)$ time, the overall runtime is $O(N)$. *Worst Case:* Any general array would require creating the heap with bubbling which itself takes $N \log N$ time.

3.1 Give a best and worst case input for insertion sort.

Best case is a completely sorted array with 0 inversions while the worst case is a reverse-sorted array with $\Theta(N^2)$ inversions. Recall that the runtime for insertion sort is given by $\Theta(N + K)$ where $K$ is the number of inversions.

3.2 Do you expect selection or insertion sort to run more quickly on a reverse list?

Asymptotically, both algorithms operate run in $\Theta(N^2)$ in this scenario where $N$ is the length of the reversed list.

Selection sort might be better since it performs only $\Theta(N)$ swaps as opposed to $\Theta(N^2)$ swaps in insertion sort's case.

3.3 In Heapsort do we use a min-heap or max-heap? Why?

We use a max-heap because then we can fill in the array of sorted elements from the back to the front in the same array we use to represent our heap.

3.4 Sort the following array using Heap Sort. [3, 2, 1, 5, 6, 8, 7]

Heapify the array (may be easier to visualize with a tree structure)
[3, 2, 1, 5, 6, 8, 7]
[3, 6, 1, 5, 2, 8, 7]
[3, 6, 8, 5, 2, 1, 7]
[8, 6, 3, 5, 2, 1, 7]
[8, 6, 7, 5, 2, 1, 3]
Then delete the largest element and place it at the back of the array. Do this until the array is sorted. [1, 2, 3, 5, 6, 7, 8] **Meta**: Walkthrough heapification of the array when you go over solutions. No need to justify why/when we

sink and swim nodes. Students should be comfortable with this already. Redirect students to the previous part when they answered why we use a max heap, and emphasize that we can do everything within one array.

# 4 Vertigo

4.1 We have a list of $N$ elements that should be sorted, but to our surprise we recently discovered that there are at most $k$ pairs out of order, or $k$ **inversions**, in the list. The list `{ 0, 1, 2, 6, 4, 5, 3 }`, for example, contains 5 inversions: $(6,4), (6,5), (6,3), (4,3), (5,3)$.

For each value of $k$ below, state the most efficient sorting algorithm and give a tight asymptotic runtime bound.

(a) $k \in O(\log N)$

Insertion sort is the most efficient in this case because its runtime is $O(N+k)$. The overall runtime bound for insertion sort in this scenario is $O(N)$.

(b) $k \in O(N)$

Insertion sort for the same reason above. The overall runtime bound for insertion sort in this scenario is $O(N)$.

(c) $k \in O(N^2)$

Merge sort, quicksort, or heap sort would be ideal here since the number of inversions causes insertion sort to run in $O(N^2)$ runtime. Using one of the three sorts listed earlier yields a runtime in $O(N \log N)$ in the normal case.