Homework 3

Solution Key

This homework must be typed in LaTeX and handed in via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

Problem 1

1. Let E be the set of edges of a finite undirected graph and ℓ be the collection of subsets of E such that $A \in \ell$ if and only if A is an acyclic subset of E. Show that the ordered pair (E, ℓ) is a matroid.

2. Let $w : E \to \mathbb{R}^+$ be a function that assigns a non-negative weight to each element of E. Given a subroutine that takes as inputs an edge (u, v) in E and an acyclic subset A of E, and returns *true* if $A \cup (u, v)$ is acyclic and $false$ otherwise, design an efficient algorithm to find an acyclic subset of E of maximum weight. Analyze its runtime and argue the correctness/optimality.

1. By the definition of a matroid, we must show that:
   (a) E is a finite set
   (b) ℓ is the family of independent subsets of E, such that if $B \in \ell$ and $A \subseteq B$ then $A \in \ell$
   (c) If $A \in \ell$, $B \in \ell$ and $|A| < |B|$, then $\exists x \in B \setminus A$ such that $A \cup \{x\} \in \ell$

   (a) This is true since we know our graph is finite.

   (b) If $B \in \ell$, then B is a acyclic subset of E. Suppose, for the sake of contradiction, that $A \subset B$ but $A \notin \ell$. Then A is a subset of B containing a cycle. However, B contains no cycles. Therefore, $A \in \ell$.

   (c) Consider $A, B \in \ell$, such that $|A| < |B|$. Recall, that a connected acyclic graph is called a tree. Furthermore, since a single vertex is itself trivially acyclic and connected it is a tree.

   Consider the sub-graphs of G defined by $G_A = (V, A)$ and $G_B = (V, B)$. We can represent these as a disjoint union of trees in G also known as a *forest* of trees in G.

   Since $|B| > |A|$ we have that the number of connected components in G_B is less than the number of connected components in A. To verify this notice that without considering any of the edges in A and B, G_A and G_B begin with the same number of connected components. When we consider adding an edge to one of these graphs, this edge must connect to *disjoint* connected components. If this were not the case, i.e. an edge was added connecting to vertices in the same connected component, then we would construct a cycle in the graph, a contradiction! Therefore, the number of connected components in an acyclic graph strictly decreases as more edges are added.

Since each vertex in G is contained in some connected component of G_A and G_B, but G_B has fewer connected components than G_A, by the pigeonhole principle, there exists some connected component in G_B that contains two vertices in two distinct connected components of G_A. Since these vertices are contained in the same connected component of G_B, there exists a path from one of these vertices to the other in G_B. Furthermore, since these two vertexes are contained in disjoint components in A, there must be an edge in the path in G_B not contained in G_A such that the edge connects the two disjoint components in G_A. By adding this edge to A, the resulting graph $G_{A'}$ would be acyclic and therefore a valid extension of G_A.

2. **Algorithm** We modify the algorithm presented in Lectures 5 and 6, using the knowledge that (E, ℓ, w) is a weighted matroid.

Algorithm 1 Max Weight Acyclic Subset of a Graph

```
Input: A set of edges for a graph, E
       A predicate checking if a set of edges is acyclic, isAcyclic
       An edge weight function, w
Output: A maximum weight acyclic subset of E according to w
procedure MaxWeightSubset(E, isAcyclic, w)
    Initialize A to be an empty set
    sort(E) such that E is in decreasing order with respect to w
    for e ∈ E do
        if isAcyclic(A ∪ {e}) then
            Add e to A
    return A
```

Correctness Note that since we have shown that (E, ℓ) is a matroid and we know that w is a valid weight function, the theorem presented in lecture shows that the above algorithm will always produce an optimal subset of E.

Runtime Suppose our subroutine runs in $O(T)$ time and our input list of edges has length n. Using a sorting algorithm such as mergesort or heap sort we can perform the initial sorting of our list of edges in $O(n \log n)$ time. Initializing and adding elements to a set can be done in $O(1)$ time. Since the call to our subroutine takes $O(T)$ time, and this is done for each of the n elements of E, the total runtime of our loop is $O(nT)$. Thus, the overall runtime of the algorithm is given by $O(1) + O(n \log n) + O(nT) = O(n \log n + nT)$.

Problem 2

A *subsequence* of $[a_1, a_2, \cdots, a_n]$ is a list of numbers $[a_{i_1}, a_{i_2}, \cdots, a_{i_k}]$ such that $i_1 < i_2 < \cdots < i_k$.

Consider the list $[5, 1, 6, 2, 3, 10, 11]$. Two strictly increasing subsequences are given by $[5, 6, 10]$ and $[1, 2, 3, 10, 11]$. Moreover, $[1, 2, 3, 10, 11]$ is the longest strictly increasing subsequence of this list of numbers.

1. Given an arbitrary list of numbers, design a greedy algorithm to determine the length of the longest strictly increasing subsequence. Your algorithm should run in $O(n \log n)$ time where n is the length of the input list. Analyze the runtime of your algorithm and argue correctness/optimality.
2. Modify your algorithm to instead output the longest increasing subsequence of the original list of numbers. In the case of multiple longest increasing subsequences, outputting any of them shall suffice. Argue the correctness of your modified algorithm.

1. **Algorithm**

Algorithm 2 Length of Longest Increasing Subsequence

```
Input: A list of numbers of length $n$
Output: The length of the longest strictly increasing subsequence
procedure LengthLIS(A)
    Initialize piles to be an empty list of lists
    for $a_i \in A$ do
        if piles is empty then
            Insert $[a_i]$ into piles
        // One can accomplish the following using binary search on piles
        k ← index of leftmost pile whose first element is $\geq a_i$
        Insert $a_i$ at the front of piles[k]
    return length(piles)
```

Correctness Let L be the output of running LengthLIS on the list $[a_1, \cdots, a_n]$. Suppose, for the sake of contradiction, there exists an increasing subsequence $a_{i_1}, \cdots a_{i_k}$ of this input such that $L < k$. In other words, there exists a longer increasing subsequence of the input that our algorithm did not find. Since the length of the longest increasing subsequence is equal to the number of piles formed in our algorithm, by the pigeonhole principle there exists a pile such that at least two elements, a_{i_m} and a_{i_n}, of the subsequence share. Without loss of generality suppose $i_m < i_n$ and therefore $a_{i_m} < a_{i_n}$. By construction, a_{i_m} is placed into a pile before a_{i_n}. We therefore, try to arrive at a contradiction by showing that a_{i_n} can't possibly be put into the same pile. Let T be the top (last) number placed on the pile containing a_{i_m}. Recall a number is only placed on top of a pile if and only if it is less than or equal to the previous top of the pile. Therefore, for T to be on top of a_{i_m}'s pile, $T \leq a_{i_m}$. By this same property, for a_{i_n} to be placed on top of T, $a_{i_n} \leq T$. However, by the transitivity of order we have that $a_{i_n} \leq T \leq a_{i_m}$. This is a contradiction since we have assumed $a_{i_m} < a_{i_n}$. Thus, $k \leq L$, and the output of the algorithm is maximal.

Runtime The runtime of this algorithm is $O(n \log n)$. Notice that in the worst case the length of `piles` is n where each element of the input list forms a separate pile. This means when we search for the leftmost pile to insert element a_i to via binary search we require $O(\log n)$ operations. Furthermore, insertion of an element into the front of a linked list can be done in $O(1)$ time. Since we repeat this for all n elements of our list, the total runtime of the `for` loops is $O(n \log n)$. Finding the length of `piles` is another $O(n)$ operation. Therefore, the total runtime of this algorithm is $O(n \log n) + O(n) = O(n \log n)$.

2. **Algorithm**

Algorithm 3 Longest Increasing Subsequence

```
Input: A list of numbers of length $n$
Output: One of the longest strictly increasing subsequence
procedure LengthLIS(A)
    Initialize piles to be an empty list of lists
    for $a_i \in A$ do
        if piles is empty then
            piles ← piles ∪ [($a_i$, NULL)]
        // One can accomplish the following using binary search on piles
        k ← index of leftmost pile whose first element is $\geq a_i$
        back ← pointer to the first element of piles[k - 1], NULL if $k - 1 < 0$
        Insert ($a_i$, back) at the front of piles[k]
    subseq ← backtrack using the pointers from the first card in pile[-1]
    return reverse(subseq)
```

Correctness From above we know that our algorithm determines the length of the longest increasing subsequence. Therefore, it suffices to show that the output is a subsequence that is increasing. To show that this is a subsequence, notice that the backpointer of an element always points to an element that appeared in the list previously. This is because we scan through the list from left to right, and the backpointer always points to the top element of the pile that the element was compared with. Therefore, when reconstructing the subsequence using these backpointer we revisit elements moving right to left in the list, and reversing the list in the end gives a proper ordering. To show that the subsequence is increasing, notice that for any element in the list, the backpointer always points to a number strictly smaller than it. This is because the backpointer always points to the top element of the *previous* pile. If an element isn't placed on top of the pile we must have that its value is strictly greater than the top element of the pile, since otherwise it would be placed on top of it by construction. Thus, when reconstructing the list using the backpointers we construct a strictly decreasing list that is then reversed into a strictly increasing list. Therefore, the algorithms constructs a strictly increasing subsequence of maximal length and is a valid solution to the longest increasing subsequence problem.

Runtime The analysis of runtime is very similar to the above. If we consider our `for` loop, constructing a backpointer to the first element in the previous pile requires $O(1)$ time. Therefore, this does not modify the asymptotic runtime of the `for` loop. Backtracking through the piles will require $O(n)$ time in the worst case where we would have to visit each element of the list exactly once. Note, that we can't revisit a particular element since each backpointer always points to an element in a pile to the left. Reversing the list will also require $O(n)$ time. Therefore, the overall runtime of this algorithm is $O(n \log n) + O(n) + O(n) = O(n \log n)$.