Adam Paz
1363522

Homework 2

## 1 Car Watching

**Algorithm** Given Sequence S $(s_1, s_2, ..., s_n)$ and sub sequence P $(p_1, p_2, ..., p_k)$ using a greedy algorithm we can find out if the cow will be happy at the end of the day. If $|P| > |S|$ the cow will be unhappy. We start by comparing each passing car with p1 until there is a match, once there is a match we compare each following car with $p_i$ where $1 < i < k$, and incrementing i by one each time there is a match. Once the last element in P has a match in S we know that Betty will be happy and we no longer need to check any cars in S. If we get to the last element in S and $i! = k$ then we know Betty will not be happy. If $i = k$ and $p_k! = s_n$ then we will also know Betty is not happy.

**Proof of Correctness** Given the sequence S of cars and Betty's sub sequence P, for proving purposes lets say through our algorithm we return the indexes of our matches found in the sequence S, and Betty will be happy if there is no empty spot in our array. Lets call this array K[ ]. Where the length of K[ ] is equal to $|P|$. And we have an array J[ ] which is an optimal solution to the problem. Now lets assume J[ ] and K[ ] have some difference, meaning if both the optimal and the greedy have spots in their array, the two algorithms have chosen different points in S that satisfy a part of the sub sequence P. Now if we swap where this element is found in K[ ] with its location in J[ ] there are two things that can follow: (a) Either the element that was found by J and K were the last elements in P, in which case they are both equally optimal, because they are the same up until this point. (b) Or if the element is followed by other elements in P, but not necessarily contiguously, then we swap our K[p], p being any arbitrary number $< |P|$, with the optimal J[p]. Since our algorithm always takes the nearest match, we can assume K[p] is less than or equal to J[p]. Therefore, J[p+1] is strictly greater than K[p], so if K[p] is switched with J[p] the rest of the optimal solution can continue to stay unchanged. One of these two events apply to every possible swap between K[p] with J[p] and therefore K[p] is equally as optimal as the optimal solution J.

**Proof of time complexity** The worst case scenario is when the sub sequence P does not exist in the sequence S or when the last element of p is equal to the last element in S. In this case, since there are $|S|$ cars in S, and no car is ever compared twice we know there are $|S|$ comparisons. We take one element of P and compare it with some elements in S, once we move on to the next item in P we do not move back to recheck any elements in S, we simply move to the next item in S. Therefore there are at most $|S|$ comparisons. Therefore it is O($|S|$).

Adam Paz
1363522

Homework 2

## 2 Transportation

**Algorithm** Assume there is no cow that weighs more than W, the max weight the truck can carry. This greedy algorithm will begin by taking the list of cows not yet transported, finding the first non-transported cow, and adding it's weight with the weight of the next cow on the list. If their total weights are greater than W than just take the first cow. If their weights are less than W continue adding following cows from the list on to the truck. Continue comparing the total weight of cows on the truck with W. Once the weight is greater than W, remove the last cow from the truck and transport those cows still on the truck to the new farm. Remove the transported cows from the list of cows. Repeat recursively with the new first cow on the list.

**Proof of correctness** Lets say we have an optimal solution to this problem, lets call this set S. Lets say that in set S we have an array S[ ] in which each index represents the amount of trips taken up until this point, (i.e. index 0 no trips have been taken, index 1 one trip has been taken, etc...). And in each index is the amount of cows put on the truck on each trip. And we have the greedy algorithm's solution set as S' and the array S'[ ]. Lets say that $S[0] = n$ and that $S'[0] = n+k$, n and k being arbitrary numbers where $n+k < W$ and $0 <= k < S[1]$. We can assume that K will never be negative because the greedy algorithm takes as many cows as possible, so if the optimal solution were to be different it would have to have less cows on some trip then the greedy one. Now if we were to swap S'[0] with S[0] and take k cows away from S[1] then the rest of S would continue on the same. So there are no changes other than the one to the current index and the following index. Even though the following index must make a small change it does not change the amount of indexes needed, it simply reduces the amount of cows in that index by k. Which is then swapped in the next set of swaps with this index's amount of cows in S'[ ]. This swap can be done with every index in S'[ ] with its corresponding index in S[ ]. While causing no change to any index but the current one and the following one. Therefore it follows that the amount of indexes needed to transport the cows does not increase in S[ ] when indexes from S'[ ] are switched in and therefore the greedy algorithm is also an optimal solution.

**Proof or run time** The run time of this algorithm is O(n). Lets say we have a list of cows $(c_1, c_2, ...c_k, c_{k+1}, ..., c_n)$ We start by putting two cows on to the truck and comparing with W. The algorithm begins by comparing the first possible contiguous pair, and every time cows are brought to the new farm the next contiguous pair is compared with W. Meaning the first cow on the list gets "a free ride" on to the truck without a comparison. We can think of the comparisons as comparing cow $c_{k+1}$ with $W - c_k$ since we assume that the first individual cow will always fit on the truck. If the two cows fit then more cows are added to the truck one by one each time their weight being added to the total and compared with W, until the total weight is greater than W or until all the cows are gone. No cow is compared twice because if it is the singular cow that made the weight of the last truck run greater than W, then it is added to the next truck first with no comparison.

Adam Paz
1363522

## 3 Milking

**Algorithm** Given the triples of each cows Milk, Eat and drink time we can have it run fastest through a greedy algorithm. The algorithm takes the time to eat and drink and adds them into one simultaneous time and stores it into an array lets call it ED[ ] and the milking time in a separate array called M[ ]. It then uses merge sort to sort the array ED[ ] from longest time to shortest time. So the cows are listed from longest eating and drinking time to shortest independent of milking time. It follows that any change to ED[ ] would be made to M[ ].

**Proof of Correctness** For this Algorithm we are given triples of time taken for each cow to get milked, which can be done only one by one, and eat and drink which can be done simultaneously by any number of cows. No matter how they are ordered for milking the total milking time will always be the same, because if you added the first element in each triple you would get the same number independent of ordering. Lets say we have an array of each cows eating and drinking time summed, lets call it ED[ ]. Lets take the cow with longest ED time and call it j. Then lets swap ED[j] with ED[0]. There is the obvious case in which if ED[j] in the original ordering was the last cow to finish, in which case we have further optimized the order because ED[j] can start earlier and therefore finish earlier. In the original ordering if the cow in the jth index was not the last one to finish, then we have not proven yet that this algorithm works. Because the milking time of elements 0-j is the same whether ED[j] and ED[0] are swapped or not, the task ED[j] in the original ordering would start at the same time as ED[0] in the new ordering. We therefore have guaranteed that ED[0] will finish faster in the new order than ED[j] would have in the original ordering, meaning that we have surely not made the total time of the process any longer. And by doing this switch we have guaranteed that ED[j] and ED[1] will finish sooner in the new ordering, than ED[1] and ED[j] respectively in the original order. Therefore this greedy algorithm produces an optimal solution.

**Proof of run time** Since this algorithm really only requires sorting as far as time complexity, it's time complexity is O(nlogn), which is the worst case for merge sort. This is because once they are sorted in order of their task time of eating and drinking in descending order, making the greedy choice is a constant time action of just taking the next cow on the list once it is sorted.

## 4 FSP

**Proof of Correctness: Proposition 1** This proposition is true. Lets say that there is an edge e, which exists in a path K from vertex u to vertex v. Lets say we have a path K' which is the path from u to v in the MST. K is not equal to K', and K contains an edge e which has less length than some edge in K', therefore it follows that K is less tedious than K'. Since u and v exist in both paths K and K' it follows that the existence of both paths create a cycle. By deleting the longest edge in this cycle we can rid of the cycle and decreasing the maximum tediousness. If K has an edge e which is shorter than an edge in K' then we would delete an edge from K' that is longer than the edge e. However, by the definition of an MST we know that S contains the shortest edges possible between any two points, and since K' is in S and K is not, this proves that by contradiction path K cannot contain an edge that is shorter than the longest edge in K', and therefore the K' is the least tedious path from u to v.

**Proof of Correctness: Proposition 2** This proposition is true. With the assumption that proposition one is true, it follows that condition (c) holds. By definition an MST connects all the vertexes together with the minimal total weighting for its edges. This proves both (a) and (b). For (a), by this definition every vertex is connected. The definition also proves (b), because it has the smallest total "weight", which in this case would be the smallest total length, which (b) asks for. It follows that the MST has covered conditions (a)(b) and (c) which define it as an FSP.

**Proof of Correctness: Proposition 3** This proposition is true. Under the assumption that all edge lengths are distinct we know there is one unique MST. And since the FSP's condition a,b and c must hold true, the MST is the unique solution to the FSP. Lets say there is some edge e that is not in the MST but is in the FSP. In order for condition (b) to be true there cannot be a cycle in the FSP because this means the weight of the combined edges is not as small as possible. Therefore if e is added to the FSP and an edge in the MST is taken out it is possible to have a fully connected graph; however, by definition of the MST, this edge e cannot have shorter length than the edge taken out to put edge e into the graph. Therefore there can be no edge from the MST that is not in the FSP. The MST covers conditions (a) because it connects all vertices, (b) because it has the smallest total weight (or length in this case), and (c) due to the proof of proposition 1. Therefore the MST is the unique solution to the FSP and no edge can exist in the FSP that is not in the MST.