Written 2 - Greedy

Brian Aguirre Parada ba5bx

February 5, 2015

Problem 1:
Problem Description: To maximize the distance covered during a trip by minimizing stops along the way.
Inputs: Distance L, which stands for the total distance to be covered during the trip. A set of stopping points $(x_1, x_2, x_3..., x_n)$.
Outputs: In order to track the distance covered per day and whether the sum of the distances traveled is less than or equal to L, let the daily distance traveled be denoted by G_i, where i stands for the day of the trip, $i \geq 1$. Then the total distance covered would be:

$$G_{total} = \sum_{i=1}^{n} G_i \text{ - } G_{i-1}$$

Where $G_0 = 0$ miles.
Assumptions: Units are assumed to be in miles. All distances between any two successive points $(x_1, x_2, x_3..., x_n)$ are positive and less than or equal to d. The most one can travel in a day is d miles and one is able to determine arrival time to the next stopping point perfectly.
Strategy: A greedy algorithm that drives the maximum distance possible (d) per day.
Description: Since the most one can drive a total distance of d miles per day, the greedy algorithm makes sure to come as close as possible to that distance. The algorithm goes through the following steps:

-The algorithm verifies whether the first distance traveled, from x_1 to x_2 is less than or equal to d.
-If the distance from x_1 to $x_2 < d$, the distance is added to G_{total} and checks if $G_{total} = L$. If that's true, the algorithm stops completely. If it's false, the algorithm checks the next distance from x_2 to x_3 since it hasn't met it's daily limit of d miles.
-If the sum of x_1 to x_2 and x_2 to $x_3 = d$. The the distance is added to G_{total}. Then the algorithm checks to see if $G_{total} = L$. If that's true, it stops. If it's false, the algorithm continues.
-If the sum of x_1 to x_2 and x_2 to $x_3 < d$, it computes the next distance. In this case from x_3 to x_4 in order to see whether the new sum is strictly $\leq d$. Then again checks to see if $G_{total} = L$.

-But if the sum of x_1 to x_2 and x_2 to $x_3 > d$, we consider x_2 the first stopping point since we cannot drive over d miles. -The algorithm repeats the previous steps unless the total distance is met. Note that it's been constructed to check whether the sum is equal to L, in order to not go over the overall required distance to be traveled. The algorithm also verifies whether the daily distance covered is $\leq d$
-This greedy approach continues to try to cover as much distance as possible, without going over d miles per day, and checking if $G_{total} = L$ each time a new distance is added.

Problem 2:
LOGIC OF THE FOLLOWING PROOF: Let G represent the output of the algorithm while O be the optimal solution to the problem. By way of contradiction, assume that G and O are not the same, such that $O(L)$ returns a smaller number, thus less stops and therefore more efficient. Similarly, let $O[x_i]$ and $G[x_i]$ represent the set of stops along the way of each. At $O[1]$, meaning the first stop, the optimal algorithm certainly returns the optimal solution. Assuming 1 represents having traveled at most d without going over. Well $G[1]$, by construction also returns the most efficient solution of attaining d, but not going over, and so currently the size of both lists is the same for $G[1]$=$O[1]$. Then assume that $O[x_n - 1]$ represents the set of all optimal solutions for the given problem for the remaining number of necessary stops n. Since $G[1]$=$O[1]$, we can substitute that by saying $O[n] - G[0]$. By way of contradiction, assume that there exists a $C[n]$ which has less elements than $O[n]$ such that $C[n]$+$G[0]$ $< O$, meaning C has less stops and is the best solution for the problem. But by definition the set O has the minimum number of stops, for which the greedy algorithm attains the same number of values, thus the required contradiction. As for if the value $G[x_i]$ is not optimal, and x_j is, then the algorithm then chooses x_j distance before stop, assuming it doesn't break the rule of going over d. So the number of stops G and O have have to be the same. Having s
Proof: Let $O = \{o_1, o_2, ..., o_n\}$ be the set of optimal stops selected by the optimal solution. Let $G = \{g_1, g_2, ..., gm\}$ be the set of stop selected by the greedy algorithm developed. Since the optimal solution contains the least amount of stops in order to reach destination L, $n \leq m$. This results in the following lemma:

$$\forall_{\ 1 \leq m} g_i \leq o_i$$

Meaning that for any given value, letting that value be the i the any arbitrary stopping point, the greedy algorithm finds less stops.

By way of induction let the base case be when $i = 1$. This means that g_i $\leq o_i$. Meaning the stop for the greedy algorithm came closer to a distance d than the optimal solution. By way of contradiction, assume that $g_i > o_i$, but this goes against the construction of the greedy algorithm since it was designed to come as close to the distance d and not stop until it was either equal to it or just under it. If $g_i > o_i$, then that would mean that the algorithm chose not to keep going, which would be a contradiction to what was set to do. So the greedy stoping point should be the same as o_1.

Inductive Hypothesis: Assume $\forall\ _{1 \leq_k - 1} g_i \leq o_i$

Inductive Step: Show that $g_k \leq o_k$
By way of contradiction, assume that $g_k > o_k$. Then by the hypothesis, one can say that $g_{k-1} \leq o_{k-1} \rightarrow g_{k-1} \leq f(o_k)$. Meaning the optimal solution came closer to the distance d in order to cover more distance and make less stops but our greedy solution stopped and chose not to keep going. Thus the required contradiction.

Since $m = n$ by the previous lemma, then it is then only necessary to show that the set of stops for the greedy algorithm contains the same number of elements such as the optimal algorithm; such that $|G| = |O|$.

Assume that $|G| \neq |O|$
Problem 3:
Problem Description: Find the best path from the starting node (C-Vile) s to the end node (Montreal) t.
Inputs: Start nodes u and end nodes v within the graph G. Starting time, so that the function $f(u, v, t)$ returns the time it takes to travel from node u to v.
Outputs: Time t in milliseconds.
Assumptions: Nodes represents roads, such that from when referring to going from one node to another it logically means taking a road in order to arrive to another road. Time t is always positive and linear, such that if $t_1 > t_2$ then $f(u, v, t_1) \geq f(u, v, t_2)$. Graph G represents the various path one can take from s to t. Assume also that u and v are valid nodes and there exists a path within the graph G that connects the nodes u and v.
Strategy: Apply Dijkstra's Algorithm in order to find the shortest path from the starting node s to the target node t.
Description: Apply Djikstra's Algorithm in order to find the fastest path from the starting node to the target node.
Description: Using Djikstra's Algorithm, one can apply the same logic in order to solve this problem. In this case, the weight between the nodes is time rather than distance. Since the graph G contains the various paths from the C-Ville node s to the target node t, Montreal, the algorithm can begin by assessing the efficiency of the nodes in between the start node and end node. In order to do that, the algorithm follows the following steps until it attains the most efficient path:

- The algorithm begins only knowing node s, the starting vertex. From the this node, the algorithm starts to asses the efficiency of all of the neighboring nodes, i.e. the algorithm sees which roads are faster and which ones are slower in order to arrive at node t. For the algorithm to do that, every unknown node is set an upper bound of $t = \infty$. For

example, let s_1 denote the following node such that $f(s, s_1, t) = k$, for some millisecond value of k. Since we know that $s_1 \neq s$, then $k \geq 0$. Similarly, since know that the distance from one road to another cannot be infinate, $k < \infty$. Thus k must be in between such that $0 \leq k < \infty$. Setting this bound allows us to then store the time it takes from the starting point s to the following node s_1 into an array, so that we keep track of the effeciency of each road.
- Repeating the previous step for all of the neighboring nodes, storing all of these time values in an array of times, the algorithm can then start to evaluate which is the fastest path. In order to do that, the algorithm calculates the times from node s to node s_i.
- In the case in which we know a previous time t_i from node s, our starting node to another node s_i, since our algorithm is greedy in nature and wishes to find the fastest way to s_1, we must update our path so that only the fastest times are considered. In order to do so, we go through the array which contains the times for all of the known nodes (roads) and we compare the times tracked with any new times recorded as new nodes are discovered. At each edge, the algorithm compares the previous time and see whether the new path returns a shorter path, such that if t_{prev} $> t_{current}$, both computed by $f(s, s_i, t)$, except $t_{current}$ is shorter due to having taken another road which is faster, then we update the time in our array. In order to update it, we simply find the spot in our array which returns the time from s to s_i and change $t = t_{current}$. Deleting also the slower time.
- The algorithm continues to do this for every unknown node until we check whether the next node is the end node t. If so, we stop and calculate the fastest time in order to reach t by simply seeing which nodes, meaning roads, had the smallest weighted edges (times).

When using an array in order to keep the weighted edges and their values, the total run time of Dijkstra's algorithm is $O(v^2)$, where v stands for the number of edges.

Problem 4:

Proof: Let set $S = \{s_0, s_1, s_2, ..., s_{n-1}\}$ contain all of the nodes that the algorithm finds in order to reach node t. The time it takes on that node (road) is given by $f(s, s_i, t)$, where s_i is an element with S. By way of induction, we prove that $f(s, s_i, t)$ returns the smallest time from s to s_i such that all of the fastest roads are taken, and so the algorithm's solution is equal to the optimal solution of this given problem.
Base Case: The time it takes from node s to s_0 is optimal is s_0 is the starting road in C-Ville for our algorithm. Thus the time from where we are to where we are is 0 milliseconds.
Inductive Hypothesis: $\forall\ i \leq k - 1$, function $f()$ is minimal.
Inductive Step: Assume that function $f()$ returns a minimal time for s_k such that $f(s, s_k, t)$ is the minimal solution to the road s_k from s, and so the fastest route. By definition, the time to s_k is given by: $f(s, s_k, t) = f(s, s_i, t) + cost(e)$.

Since we assumed that the algorithm returns the fastest time in up to the road i, $\forall\ i \leq k-1$, then by way of contradiction, assume that the time from s_i to k is not optimal, in order to prove it is.

Assume that the time to s_k is not optimal, meaning for $f(s, s_k, t)$, $\exists\ s_{k'}$ for which the route is faster. Such that, the following is also true: $f(s, s_j, t_j) + cost(e') + \delta < f(s, s_i, t_i)$+$cost(e)$, implting that j ¡ i, meaning the times for the first one is faster than the second one. But since $\delta \leq 0$, the best δ can be is 0. So if $\delta = 0$, then $f(s, s_j, t_j) + cost(e') < f(s, s_i, t_i)$+$cost(e)$, thus the required contradiction. Since Dijkstra's algorithm selects the most efficient path, in this case based on time from one node to another node, it the above statement cannot be strictly less than. So the algorithm is in fact optimal.

Problem 5:

1. Yes, every MST of G is an *Elite Tree* by definition. A *bottleneck edge* is the the most expensive edge on a spanning tree T. Given that T represents any arbitrary spanning tree in G, T could very well be a minimum spanning tree of G. Assuming that it is, let $T = \{t_0, t_1, t_2, ..., t_n - 1\}$ where t_i was the most expensive edge of of that tree. Then $\forall\ t$ values in T, T contains only the smallest edges connecting two nodes. Then T is indeed an *Elite Tree* since T contains the minimum value possible *bottleneck edge*, which by construction used to be t_i but since reducing that edge, T is a minnimum spanning tree.

2. No, not every *Elite Tree* of G is an MST. An *Elite Tree* by definition just contains the minimum *bottleneck edge* of an arbitrary spanning tree T. Since T could contain other non optimal edges, it is not an MST. For example, let $T = \{5,\ 9\ ,\ 5\}$, three edges that connect 4 nodes within G. If we let the MST of G equal $\{5,\ 4,\ 2\}$, such that there are two edges more optimal than the ones in T. The *Elite Tree* would be a tree that reduces the *bottleneck edge* of T, meaning the edge 9, which is the most expensive edge. Then the *Elite Tree* $= \{5,\ 4,\ 5\}$, satisfying the criteria that an *Elite Tree* contains the minimum *bottleneck edge* of a given spanning tree T within G.