Homework 3 - Solutions

Deadline: 11:59pm, March 29, 2013
Available points: 112. Perfect score: 100.

Problem 1 (24 points): The solution to this problem follows a Dynamic Programming approach.

A. The algorithm will make several passes over the tree structure, filling in a single base of the sequence in each node at a time. Begin the dynamic programming approach bottom up:
1 Process the tree bottom-up, so that any node which we process, v has already had its children processed. For this node v.

1.a Define $C(v)$ as the cost of the optimal solution to the subproblem corresponding to the tree rooted at v.

1.b Define $C(v, \sigma)$ be the cost of the best labeling of this subtree when we choose node v to use base σ. From this, it follows that $C(v) = min_{\sigma} C(v, \sigma)$.

1.c If v is a leaf and is assigned base σ, then $C(v, \sigma) = 0$, and if v is not assigned that base, then $C(v, \sigma) = \infty$.

1.d If v is an internal node, compute $C(v, x)$ as:

$$C(v, x) = \sum_{children\ v'\ of\ v} (min(C(v') + 1, C(v', \sigma)))$$

2 After reaching the root, begin the backtracking phase, moving top-down:

2.a Assign the root r, character σ_r, such that $C(r) = C(r, \sigma_r)$.

2.b For any other node, v' which is a child of node v, assign $\sigma_{v'}$ according to the following:

$$\sigma_{v'} = \begin{cases} \sigma_v & : \text{if } C(v') + 1 > C(v', x_v) \\ y \text{ such that } C(v') = C(v', y) & : \text{ otherwise} \end{cases}$$

The running time of a single pass of this algorithm is $O(n)$, as we must process all nodes above the n leaves, of which there are $n - 1$. For each of these leaves, we must compute the cost for each of the 4 possible choices of base, therefore giving us the running time of $4 * (n - 1) = O(n)$. Then, in order to complete the tree, we repeat the above steps for all k bases in the sequence. This results in a total running time of $O(nk)$.

To argue for the correctness of this approach, it is first argued that construction base-by-base is valid. This is apparent from the fact that the choice for each base is completely independent from all the other bases in the sequence. Furthermore, it is easy to see for a single base that we are building optimal solutions to sub-problems at every step. Assume we made a different choice for a base at a given level. It follows then that the agreement between this node, v and its children must be decreased. This choice could potentially increase agreement with the parent of v, but this increase is at most 1 while agreement with children is guaranteed to be decreased by at least 1. We then compose these sub-problems together to create larger optimal solutions until we have the global solution.

B. See Figure 2.

Problem 2 (20 points):
A. Begin by ordering the vertices of the *DAG*. Start with the vertex which has no out edges, and assign its payoff to its own value. Then, iterate up the ordering of vertices, setting the payoff to the maximum of the node's value and the payoff of the last iteration. This can equivalently be done through a depth-first traversal of the DAG, where we recursively return the maximum between a node's value and the value returned by the DFS. Note that because this is a DFS, the running time is linear.

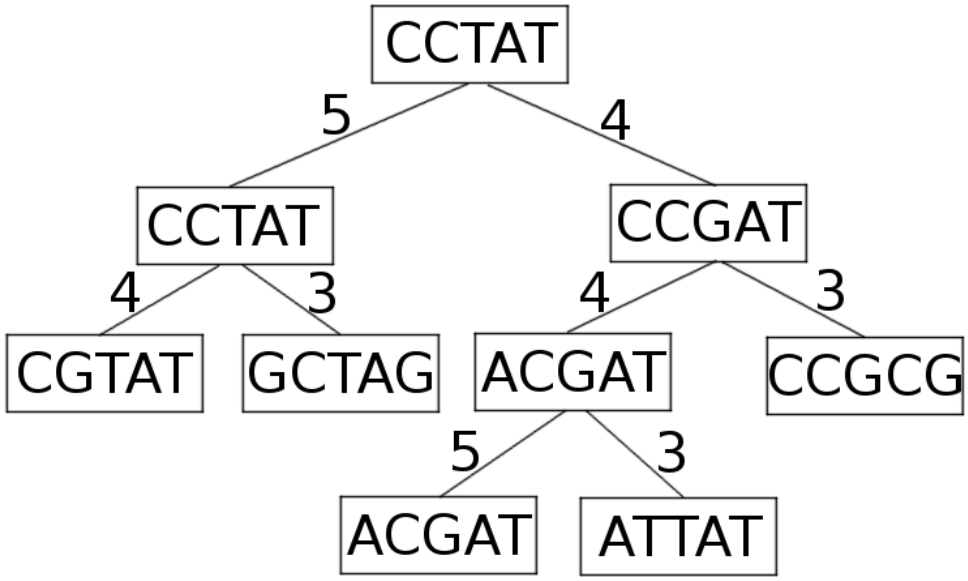

Figure 1: An example maximum likelihood tree for the genome reconstruction. Note that the values on the edges represent `agreement` between nodes.

B. Similar to the solution provided in part A. Create the hierarchy of nodes, where each level contains a cycle or a single node. Apply the same algorithm as in A., but ensure that all nodes which are in a cycle share the same payoff. Again, this is equivalent to performing a DFS where the cycles are "collapsed" into a single node first, where all nodes in the cycle share the same payoff. Again, because we are running DFS, the running time is linear.

Problem 3 (20 points): Begin by modeling the problem as a graph, where the n articles are vertices, and the x relations are edges which indicate differences in ideology. This problem can then be solved by using a graph-search algorithm to color the vertices of the graph. A simple way of doing this is to perform a breadth-first search, and using two colors to label the graph. Begin by assigning a label 'red' to the first node of the search, and then color all of its children 'blue'. The children of 'blue' nodes are colored 'red'. Repeat this process until the entire graph is colored. If ever the algorithm tries to color a child which is already colored the opposite color, then this graph contains an odd cycle, and this coloring fails.

It is important to note that if there are disconnected components in this graph, the algorithm cannot accurately match articles between disconnected components. For example, if there are any singular nodes with no edges, then it is unknown if it should be labeled 'red' or 'blue', or in this case, 'Tory' or 'Whig'. Again, we argue the running time of this algorithm to be linear, as it is just a modified BFS.

Problem 4 (24 points):
A. There are multiple ways to answer this problem. For instance, it is possible to perform depth-first search over the provided graph, G, with the modification that you only add a node to the open list if it is a correct descendant of it's parent node, for the given sequence N. If the search reaches a depth of k, return the path to that node, otherwise, if the the open set becomes empty, report that no such path exists. In the worst case, the complexity of this search is $O(|E|)$, though it is expected that relatively few out-edges will contain the labels we will follow.

An alternative dynamic programming solution can be easily adapted to answer part B as well. The original problem asks for a path in the graph starting at v_{init} that is labeled with $\{d_1, d_2, \ldots, d_k\}$. We can define subproblems as follows: paths starting at v_{init}, labeled with $\{d_1, d_2, \ldots, d_j\}$, where $j \leq k$, and ending at a specific node v_m. Thus, each subproblem $S(j, m)$ can

be identified by two parameters. Parameter j, the index of the last digit identified, and parameter m, the index of the vertex that the $j-th$ prefix of the number is able to reach. The solution to each subproblem then is a boolean output, either 1 ("yes, a path exists") or 0 ("no"). The base case for each subproblem can be formulated as follows: $S(0,0) = 1$ and $S(0,m) = 0,\ \forall\ 1 \leq m \leq n-1$. We can also define the incremental step for solving larger problems given solutions to smaller ones:

$$S(j,m) = \bigvee_{n:(v_n,v_m)\ \in\ E\ \wedge\ tag(v_n,v_m)=d_j} S(j-1,n)$$

This means that a subproblem for the j first digits can be constructed from subproblems for the $j-1$ first digits as follows: there has to be a node v_n that a path $\{d_1, d_2, \ldots, d_{j-1}\}$ reaches from v_{init} and there is an edge (v_n, v_m) in the graph that is tagged with digit d_j.

When we reach the subproblems for which $j = k$ (i.e., the length of the input digit sequence), then the overall problem has a solution if there is at least one m index for which $S(k,m) = 1$. This means that there is a vertex v_m that corresponds to a path from v_{init} that has the label $\{d_1, d_2, \ldots, d_k\}$.

B. We can solve this problem in a similar way to the dynamic programming solution for part A. The subproblems are the same but instead of returning boolean values, they return real numbers indicating the maximum probability of a path that produces that prefix and finishes at the corresponding vertex. A probability of 0 means that no such path exists. The initial conditions are the same as before: $S(0,0) = 1$ and $S(0,m) = 0,\ \forall\ 1 \leq m \leq n-1$. The recurrence can be defined as follows:

$$S(j,m) = \max_{n:(v_n,v_m)\ \in\ E\ \wedge\ tag(v_n,v_m)=d_j} \{p(v_n,v_m) \cdot S(j-1,n)\}$$

The time complexity can be computed by identifying the number of cells that we have to compute and the effort we need to spend in order to evaluate each cell. The size of the table is $k|V|$, where $|V|$ is the number of vertices and k is the length of the number sequence. The time to fill in a cell depends on how many edges there are entering that cell. In order to compute a new row of the S matrix, we need to look all the edges in the graph (as we have to consider all vertices in the graph and for each vertex, all its incoming edges). Therefore it takes $O(|V|+|E|)$ to fill in each row and there are k rows to fill. Therefore, the running time of the algorithm is in the order: $O((|V|+|E|)\cdot k)$.

Problem 5 (24 points):

A. To show this, it must be proven in both directions. We begin by showing that if a solution exists, then there are no junctions with odd degree. It is clear that if a solution exists, it cannot pass through a junction of degree 1, as any tour which would visit this node must necessarily visit the only edge connecting it to the rest of the graph multiple times. For junctions of degree > 2, consider that a tour must pass through this junction multiple times. Each visit will invalidate exactly 2 edges connected to this junction: one edge used to reach the junction, and one used to leave the junction. Therefore, for odd degree, the edges will eventually degenerate to the case of a dead end, having degree 1. From this it follows that if a closed tour exists, then there can be no odd-degree junctions.

Next, it must be shown that if no odd degree junctions exist, then there necessarily exists a closed tour solution. Consider such a graph and perform a random walk until reaching the junction from which the walk started. For each junctions visited, exactly 2 edges were invalidated for future exploration; however, there may be many junctions with valid edges. The number of such edges is necessarily even. Continue performing new random walks from any of these junctions until reaching a junction already visited. It must be that such a node is reached, because being unable to reach one of these nodes implies there is a dead-end in the road network. This process will iteratively invalidate edges until the network is covered.

This completes the proof in both directions.

B. The road network must have exactly two junctions which have odd degree. In this case, the trail followed must start at one of these junctions and necessarily ends at the other.

C. For the directed case, it must be that all of the junctions have equal out-degree and in-degree, for example, if some node has x in-edges, it must also have x out edges. A further requirement is that all such junctions must belong to a single strongly connected component.