Problem Set 1

Dave Lu

September 20, 2022

1. (a) This can be reformulated as a k-coloring problem.

   Input: A graph $G = (V, E)$, where vertices in V represent reads, and edges in E show us that the reads are incompatible with each other (ex. a vertex representing read ACG and a vertex representing ATT would have an edge between them, as CG and TT are different).

   Output: A number k which is the smallest number of colors that could be used to perform a k-coloring of the graph.

   Objective: Minimize k.

   (b) This is essentially the longest substring problem.

   Input: a set of sequences S.

   Output: a sequence x that is a substring of all sequences $s \in S$.

   Objective: maximize $|x|$.

   (c) This is essentially a maximum flow problem.

   Inputs: a graph $G = (V, E)$ (where V represents all of the compartments and E represents the connections between the compartments), a source node $s \in V$, a target node $t \in V$, and a function $f : E \to \mathbb{R}$ mapping edges to the maximum possible flow through that edge.

   Output: a number $b \in \mathbb{R}$ which is the maximum possible flow from the source s to the target t.

   Objective: maximize b.

   (d) This is essentially a minimum test-set problem.

   Inputs: a set E of all exons, a set of splice forms S containing subsets of E
   (for each element $s_i \in S$, $s_i \subseteq E$, aka. each splice form contains a subset of all exons)

   Output: a set $S' \subseteq S$, such that for each $e_i, e_j \in E$, $\exists s \in S'$ such that either $e_i \in s$ or $e_j \in s$, but not both.

   Objective: minimize $|S'|$.

   (e) This can be converted to a shortest path problem.

   Inputs: A graph $G = (V, E)$ (where V represents each target gene and E represents the interactions between genes), a start node $x \in V$, an end node $y \in V$, and a function $f : E \to \mathbb{R}$ mapping edges to $-log(p_e)$, or the negative log of the probability of interaction.

   Output: An ordered list containing a path from s to t.

   Objective: minimize the total weight of the path from s to t (which in turns maximizes the probability of interaction).

2. We assume that the genes are represented as vertices in a graph, and co-expression is represented as edges in a graph.

   (a) We can use a simple graph traversal to find out which vertices are connected to each other; any vertices that we see in the traversal, we toss into a set. All vertices in that set are then in the same co-expressed module.

   (b) This looks similar to the clique problem, which is NP-complete. One possible heuristic is to start with a set containing one vertex, and then add vertices to the set if it has edges to all of the vertices in the set. If we don't add a vertex after seeing it, ignore it for that set. Repeat until we have no more vertices to look at, then start again with a new set containing one vertex.

   (c) This looks like a max k-cut problem, which is NP-complete. One possible greedy algorithm to solve this is to contract edges that have the smallest values until we end up with k partitions.

   (d) The model in part A is probably the best one, since we can efficiently find the correct answer.

   (e) The best model to use is probably the one from part C, as it is forcing us to group the genes up into k groups no matter what, and not let any of the genes out of the groups. This makes it more resilient against lots of errors compared to the other two models, which are very reliant on the values of the edges.

   (f) The model from part B is probably the best, since even if there are some false positives, it's unlikely for a gene to have false positives with all of the other genes in a group, so even then it won't make it into the clique and will not be considered as part of the group.

3. (a) Inputs: a graph $G + (V, E)$, a function mapping edge weights to log probabilities of random chance overlap $f : E \to \mathbb{R}$, and the number of reads k.

   Output: a path of length k (as an ordered list of vertices in V) that does not contain repeats of vertices.

   Objective: minimize the sum of the log probabilities of the paths.

   (b) Note that in the decision problem variant, the only difference is the objective function: the sum should be less than a given bound $B \in R$.

   To verify a solution is valid, we simply follow the vertices in the ordered list: checking that there does exist an edge between the given vertices, and checking at the end whether the sum of all of the edges' probabilities is less than B. This can obviously be done in polynomial time as this is a simple graph traversal.

   (c) We will reduce TSP to Assembly-TSP (our current problem) by converting any TSP problem to an Assembly-TSP problem.

   A TSP problem takes as input a graph $G = (V, E)$, a function $f : E \to \mathbb{R}$ linking edges to edge weights, and a bound B. We can easily convert this to an Assembly-TSP problem, which only has one extra input: the number of vertices used, k.

   By setting $k = |V|$, we have converted a TSP problem to an Assembly-TSP problem. A solution to the Assembly-TSP problem is also a solution to the original TSP problem, as it will give us a path with all vertices in it, no overlaps, and the total weight $\leq B$.

   (d) We could use a branch and bound algorithm. A branch and bound algorithm for our assembly problem should give us the optimal solution for the reads, but faster than just brute forcing through all possible reads.

4. (a) We use the normal union algorithm, but with one modification: after taking all of the valid edges with the lowest weight and adding them to the MST, we then look at those edges and see if they are NOT in any cycle in the graph. If they are not in any cycles, we keep them. Else, we toss the ones that are in cycles.

   (b) For each edge $e = (u, v) \in E$, we run DFS from u to v, but ignoring edge e. If we cannot reach v from u when ignoring edge, then there must not be any cycles containing $e = (u, v)$.

   (c) Given a weighted graph, we do the following:

      1. start with an initial empty set T
      2. for each node v, create a set S_v containing only v
      3. sort the edges in increasing weight into a list L
      4. for all edges $e = (u, v)$ with the lowest weights: (if there are multiple edges with the lowest weight, take all of them through this at once)
         if $s(u) \neq s(v)$:
         i. $s(v) \leftarrow s(u) \cup s(v)$
         ii. $s(u) \leftarrow s(v)$
         iii. $T \leftarrow T \cup \{e\}$
         iv. for the edges we just added (which all have the same weights), check each edge if it is in a cycle in T, and remove it from T if it is part of a cycle.
         (we can check if an edge is in a cycle using the algorithm from part b.)
      5. return T.

   (d) See the code given.

   (e) For sample1.txt: $[(0, 1)]$
   For sample2.txt: $[(0, 1), (0, 3)]$