Homework 2

CS 4364/5364
Spring 2022

Due: 9 February 2022

Because of the reliance of the particular assignments in this class on mathematical notation, and the fact that all assignments will be submitted electronically, students are encouraged to use LaTeX to formalize their responses. **For those enrolled in the graduate section the use of latex is *required*.** This assignment (like all others) will be posted on the course `github`[1] as source code as well as in PDF form on the course website. Please submit your assignment to the professor via email, either as a link to your assignment online (i.e. overleaf or github) or as an attachment. Graduate students will need to include the `.tex` files as well as a PDF, this is optional but encouraged for undergraduates.

1. **(25 points) The first method we saw to find a pattern P in a text T was using the maximum prefix overlap values on a special string (calculate M_i for each i in the string $P\$T$). We know that we can compute these M_i values in $O(m+n)$ time (assuming $|P| = n$ and $|T| = m$). The solution described in class assumes both strings are over the same alphabet: what if they were not?**

   Consider the following: Given a protein sequence pattern $P \in \Sigma^*_{AA}$ over the amino acid alphabet, and a RNA text $T \in \Sigma^*_{RNA}$ (see footnote [2]). Develop an algorithm that uses the maximum prefix overlap method to determine where the pattern P is in the text T (if it exists), and runs in $O(m+n)$ time.

   Translating from RNA codons (3 nucleotides) to amino acids is done using the standard codon table (can be found in the slides, though its not actually needed for this exercise). The problem is that to reverse the translation of an amino acid $p_i \in P$ there are multiple choice of codon that could have produced it, thus there is an exponential number of possible un-translations of P. Therefore, we cannot simply enumerate all possible

[1] `github.com/deblasiolab/CS4364-documents`

[2] $\Sigma_{AA} = \{\mathtt{A}, \mathtt{R}, \mathtt{N}, \mathtt{D}, \mathtt{B}, \mathtt{C}, \mathtt{E}, \mathtt{Q}, \mathtt{Z}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{L}, \mathtt{K}, \mathtt{M}, \mathtt{F}, \mathtt{P}, \mathtt{S}, \mathtt{T}, \mathtt{W}, \mathtt{Y}, \mathtt{V}\}$ and $\Sigma_{RNA} = \{\mathtt{A}, \mathtt{C}, \mathtt{U}, \mathtt{G}\}$

RNA un-translations of the pattern and run the maximum prefix overlap, it would be exponential time which is not feasible.

Hint: you can run the algorithm multiple times, but as long as this number of repetitions is constant it is absorbed by the big-O.

Assume there is a function $L : \Sigma^*_{RNA} \to \Sigma^*_{AA}$ that will convert an RNA string to amino acid, removing up to 2 characters at the end for incomplete codons.

- define $\hat{T}_1 = L(T[1...n])$
- construct a new string $S_1 = P\$\hat{T}_1$ (where $\$ \notin \Sigma_{AA}$)
- calculate the maximum prefix match values $M_i(S_1), \forall 2 \leq i \leq (m + n/3 + 1)$
- if any $M_i(S_1) == m$, return `true` and exit
- define $\hat{T}_2 = L(T[2...n])$
- construct a new string $S_2 = P\$\hat{T}_2$ (where $\$ \notin \Sigma_{AA}$)
- calculate the maximum prefix match values $M_i(S_1), \forall 2 \leq i \leq (m + n/3 + 1)$
- if any $M_i(S_2) == m$, return `true` and exit
- define $\hat{T}_3 = L(T[3...n])$
- construct a new string $S_3 = P\$\hat{T}_3$ (where $\$ \notin \Sigma_{AA}$)
- calculate the maximum prefix match values $M_i(S_3), \forall 2 \leq i \leq (m + n/3 + 1)$
- if any $M_i(S_3) == m$, return `true` and otherwise return `false`

We know that for any $i > 3$, $L(T[i...n])$ will be a suffix of $L(T[(i\%3) + 1...n])$. Therefore if P is contained in $T[i...n]$, it will be contained in $T[(i\%3) + 1...n]$. This is why only a constant number of searches need to be performed. Because there is only one translation of RNA into amino acids, we know that we cannot miss P if we don't make the right translation. Therefore we can be assured that we are finding P if it exists using the 3 searches described.

The running time of L is linear with respect to the string given, this all of the conversions take $O(m)$ time each. The construction of the Ss takes $O(m + n + 1)$ time each, as does the calculating of M_i (these are the dominating factors). The searching across M_is takes $O(m)$ time. Since we construct a constant number of Ss and calculate M_is on them, the total running time is $O(m + n)$.

2. **(20 points) Given a directed graph $G = (V, E)$, source and sink vertexes $s, t \in V$, and a limit on the flow allowed through nodes m_e (defined $\forall e \in E$). Write the max flow problem as a linear program, and explain each (set of)**

equation in your definition. Remember the max flow problem assigns a flow (weight), f_e, to each *edge* in a graph while maximizing the total flow going from the source to the sink. For each node (other than the source and sink, i.e. $v \in V \setminus \{s, t\}$) the total in flow must equal the out flow. Some helpful notation to use:

- **a directed edge $e_{(ab)} \in E$ goes from a to b**
- **the set of *in* edges to a vertex v can be written as $E_{(*v)} = \left\{e_{(a,b)} | b = v, e_{(ab)} \in E\right\}$**
- **the set of *out* edges to a vertex v can be written as $E_{(v*)} = \left\{e_{(a,b)} | a = v, e_{(ab)} \in E\right\}$**
- **we can say the sum of the *out* flow to a node v is $\sum_{e \in E_{(v*)}} f_e$**

Your variables will be the set of f_e's, some of these (but not all) will end up in the optimization function. The things to keep in mind that must be satisfied are: (1) conservation of flow across nodes, (2) maximum flow across an edge, and (3) the outflow at the source should equal the inflow at the sink (though this may not need to be explicit in your program).

$$\begin{aligned}
&\text{minimize} && \sum_{e \in E_{(s*)}} f_e && && (1)\\
&\text{subject to} && f_e \leq m_e, && \forall e \in E && (2)\\
& && \sum_{e_i \in E_{(v*)}} f_{e_i} - \sum_{e_o \in E_{(*v)}} f_{e_0} = 0, && \forall v \in V \setminus \{s, t\} && (3)
\end{aligned}$$

The objective, (1), sums the amount of flow out of the source. Since the source and the sink are the only two nodes who's flow is not conserved, maximizing the flow out of this node maximizes the flow into the sink (they will also have matching values because of this without the need for an extra constraint). The conservation of flow across nodes is guaranteed by the equations (3), there is one for each non source/sink node in the graph. The flow constraints are enforced by (2), because m_e is a given constant it can be on the right side of the inequality.