CS302 - Final Review

1. Given two strings x and y, create a dynamic programming algorithm to compute their *optimal edit distance*, where the edit distance the numbers of insertions, deletions, substitutions, or transpositions (switching the order of two adjacent letters), in some sequence that changes x into y. The optimal edit distance is the smallest possible edit distance. For example, the optimal edit distance of SHOAL and COLA is 3, as we can remove S, changing $H \to C$, and transposing the L and the A. Edit distance is useful for spell checking applications and genomic applications.

   Solution Let x_i be the ith character in x, and let $\vec{x}_i$ be the first i letters of x. Likewise for y. Let $T(\vec{x}_n, \vec{y}_m)$ be the set of optimal transformations to turn $\vec{x}_n$ into $\vec{y}_m$.

   We note that the choice to do a transposition is only better than inserting/deleting/substituting if the transposition will put both letters in the correct position.

   Then

$$T(\vec{x}_n, \vec{y}_m) = \begin{cases} T(\vec{x}_{n-1}, \vec{y}_{m-1}) & \text{if } x_n = y_m \\ T(\vec{x}_{n-1}, \vec{y}_m) + \text{ delete } x_n \text{ from } \vec{x}_n & \text{if last transformation is delete } x_n \\ T(\vec{x}_n, \vec{y}_{m-1}) + \text{ add } y_m \text{ to } \vec{x}_n & \text{if last transformation is add } y_m \text{ to } \vec{x}_n. \\ T(\vec{x}_{n-1}, \vec{y}_{m-1}) + \text{ sub } x_n \text{ with } y_m \text{ in } \vec{x}_n. & \text{if last transformation is sub } x_n \text{ with } y_m \\ T(\vec{x}_{n-2}, \vec{y}_{m-2}) + \text{ transpose } x_n,\, x_{n-1} \text{ in } \vec{x}_n. & \text{if last transformation is transpose } x_n,\, x_{n-1} \end{cases} \tag{1}$$

   Converting this into pseudocode, we have

Algorithm 1: `Edit`(x, y)

```
Input  : Two strings $x$ and $y$.
Output: Minimum edit distance between $x$ and $y$.
Initialize 2D array $A$ of size $(|x|+1) \times (|y|+1)$ where $A[i,j]$ will contain the edit
  distance between $\vec{x}_i$ and $\vec{y}_j$;
// Base cases: To transform a zero-length string into a word, need
   to add all letters. To transform a word into a zero-length
   string, need to delete all letters
for $i = 0$ to $|x|$ do
    $A[i,0] = i$;
end
for $j = 1$ to $|y|$ do
    $A[0,j] = j$;
end
// Filling in the main array
for $i = 1$ to $|x|$ do
    for $j = 1$ to $|y|$ do
        if $x[i] = y[j]$ then
            $A[i,j] = A[i-1,j-1]$;
        else
            if $i, j \geq 2$ and $x[i] = y[j-1]$ and $x[i-1] = y[j]$ then
                $A[i,j] = 1 + \min\{A[i,j-1], A[i-1,j], A[i-1,j-1], A[i-2,j-2]\}$;
            else
                $A[i,j] = 1 + \min\{A[i,j-1], A[i-1,j], A[i-1,j-1]\}$;
            end
        end
    end
end
return $A[|x|,|y|]$;
```

2. DOUBLE-k-INDSET is the problem whose input is a graph, and the output is YES if and only if there are at least 2 distinct independent sets of size at least k in the graph. By distinct, I mean that the intersection of the two independent sets is empty. Prove DOUBLE-k-INDSET is NP-Complete.

Solution First we prove DOUBLE-k-INDSET is in NP. The witness y should be the description of the vertices in the two independent sets. Then to check the witness, we first check that no vertex appears more than once. Next we check that each set has at least k vertices in it. Next we check each pair of vertices within each set, to make sure that there is no edge in the graph between those two vertices. These checks take polynomial time in the size of the input. Also, the witness y has size less than the total number of vertices, so it has size that is polynomial in the input size. Thus DOUBLE-k-INDSET is in NP.

To prove DOUBLE-k-INDSET is NP-Hard, we use the same reduction from 3-SAT to k independent set that was in the problem set, except we add an extra k vertices to the graph, with no edges between them, and edges from those vertices to every other edge in the graph. Thus these k vertices constitute a k-independent set, but none of them can be involved in an independent set with any other vertices. Then there will be another k-independent set amongst the remaining vertices if and only if the 3-SAT instance is satisfied (see previous proof for why this is true). Thus there will be at least two distinct k-independent sets in the whole graph if and only if there is a satisfying assignment to 3-SAT.

3. What is the average runtime of randomized search without replacement if there are c copies of the item we are looking for out of an array of size n. Please use indicator random variables and go through the usual routine. You do not need to simplify your final answer - it can be a messy sum.

   Solution The sample space is the set of all possible choices of guessed elements. Let X be the random variable that is the number of guesses required to find an item we are looking for. Let X_i be the indicator random variable that takes value 1 if we have at least i rounds. Then

$$X = \sum_{i=1}^{n-c+1} X_i \tag{2}$$

   so using linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n-c+1} \mathbb{E}[X_i] \tag{3}$$

   and using the properties of indicator random variables:

$$\mathbb{E}[X] = \sum_{i=1}^{n-c+1} Pr(\text{at least } i \text{ rounds occur}) \tag{4}$$

   Now the probability that there are at least i rounds is the probability that we haven't found one of the items in the first $i-1$ rounds. This is

$$\frac{n-c}{n} \times \frac{n-c-1}{n-1} \times \frac{n-c-2}{n-2} \times \cdots \times \frac{n-c-(i-2)}{n-(i-2)} \tag{5}$$

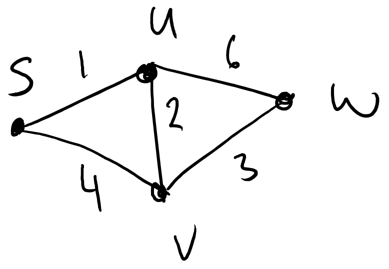
   Plugging in

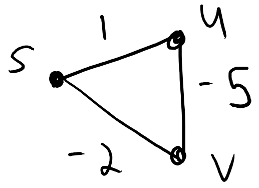
$$\mathbb{E}[X] = 1 + \sum_{i=2}^{n-c+1} \frac{n-c}{n} \times \frac{n-c-1}{n-1} \times \frac{n-c-2}{n-2} \times \cdots \times \frac{n-c-(i-2)}{n-(i-2)} \tag{6}$$

4. For the following statements regarding Dijkstra's algorithm, either explain why it is true (formal proof not required), or provide a counter example.

   (a) Consider a graph G that is directed, has negative edge weights, but no negative cycles (a negative cycle is a cycle where the sum of edge-weights in the cycle have negative value.) Then there will always be a vertex where the incorrect distance is calculated.

   (b) Consider a graph G that is directed, and that has a negative cycle that is reachable from s. Then there will always be a vertex where the incorrect distance is calculated.





Solution

(a) This statement is false. Consider the graph on the right above. If we make it directed so that the edges are (s, u), (s, v), and (v, u), and if we change the edge weight on (u, v) to be 5, then the algorithm will find the shortest paths to all vertices.

(b) This statement is true. For every vertex on the negative cycle, the shortest distance is negative infinity, because you can just keep going around the cycle to get to shorter and shorter paths. But Dijkstra's algorithm will give a finite distance to each of the vertices on the cycle, which is incorrect.

5. Return to conference scheduling: Suppose you have n events, each with a start time s_i and end time f_i, for $i \in \{1, \ldots, n\}$. Unfortunately, you only have one auditorium, and you can't schedule conflicting events (events where a start time of one is between the start time and end time of another.) You would like to maximize the number of events that are held. Consider an algorithm that at each iteration, picks the remaining event with the earliest finish time. Prove this algorithm is optimal.

Solution Let's re-label the events in the order in which they are chosen by this greedy approach. Suppose K is the number of events that are actually scheduled. Events that can not be scheduled/chosen are given labels from $K + 1$ to n. Let σ be this ordering.

Now suppose for contradiction that there is another algorithm that chooses events according to order σ^*, and assume it can schedule more events than σ. We will display a series of exchanges which transforms σ^* to σ, while never decreasing the number of events that are scheduled, proving a contradiction.

First note that if σ^* chooses to hold events i, j such $1 \leq i < j \leq K$, then σ^* must also hold event i before event j. Since $i < j$ in σ, that means $s_i < s_j$, so there would be no way to schedule j before i.

Let i be the first event scheduled by σ^* that is not in the same order as σ. There must be some event like this, or $\sigma^* = \sigma$. There are two cases. First, suppose $i \leq K$. (In other words, i is included is σ's schedule.) By the above argument, i can only occur earlier in σ^* than it does in σ. By the above argument, this means that event $i - 1$ is excluded from σ^*. But because the order of σ^* is the same as σ, that means we could consider a new schedule $\sigma^{*\prime}$ that is the same as σ^* but with $i - 1$ inserted before i. (Such a schedule is viable because σ is a viable.) This new schedule would have more events scheduled than σ^*, a contradiction.

For the second case, suppose $i > K$. (In other words, i is excluded by σ's schedule.) Let q be the event that comes immediately before i in σ^*. Then let's consider the schedule $\sigma^{*\prime}$

which is the same as σ^*, but with event i replaced with the event $q+1$. We need to show that $\sigma^{*\prime}$ is still a valid ordering. But recall that event $q+1$ is the event with the earliest end time among events with labels greater than q. This means that event i must have an end-time later than event $q+1$. Since all events after event i must start after its end-time, we can safely replace event i with event $q+1$ without conflicting with any later events in σ^*. This transformation from σ^* to $\sigma^{*\prime}$ preserved the total number of events scheduled.

If we continue transforming the schedule according to the above two rules, we will either encounter a contradiction (case 1), or end up (via sorting) in the same order as σ, without increasing the number of events scheduled. Thus our original ordering σ must be optimal.