CS561, HW6

Ryan Scherbarth, University of New Mexico

September 2024

1. You are buying pizza for n friends. You ask each friend to give you a $1 coin. When you get to the pizza place, it's closed, so you return the coins to your friends in a random order.

   - (a) Consider a single friend. What is the probability that they get back their own coin?

   If we have n friends, the probability that one single friend out of the group will get his own coin back is given by $p(n) = \frac{1}{n}$.

   - (b) What is the expected number of friends who will get back their own coins?

   We define x_i as a discrete random variable s.t. $x_i = 1$ if a friend receives his coin back, and 0 otherwise. We can then calculate the expected value $E[x_i]$;

$$
\begin{aligned}
&= \sum_{i=1}^{n} x_i \\
&= \sum_{i=1}^{n} \frac{1}{n} \\
&= (\frac{1}{n})(n) \\
&= 1
\end{aligned}
$$

   So the expected number of friends out of n that will receive their own coin back is 1.

   - (c) Use Markov's inequality to get an upper bound on the probability that at least 2 of your friends will get their coin back.

   We use Markov's inequality to prove an upper bound on the probability of a non-negative discrete random variable. Given the variable X, for any $\alpha > 0$, $P(X \geq \alpha) \leq \frac{E[X]}{\alpha}$. We want to choose $\alpha = 2$ to find the probability that at least 2 friends get their coin back.

$$
\begin{aligned}
P(X \geq 2) &\leq \frac{E[X]}{2} \\
&\leq \frac{1}{2}
\end{aligned}
$$

   Given that Markov's inequality defines an upper bound on discrete random variables, this conclusion tells us that the probability at least 2 friends get their coin back is at most $\frac{1}{2}$, or 50%.

- (d) What is the expected number of pairs of friends i and j, such that friend i gets j's coin and friend j gets i's coin?

Start by defining a discrete random indicator variable $X_{i,j}$ s.t. $X = 1$ when i gets j's coin, and j gets i's coin, and 0 otherwise.
For $X_{i,j}$ to happen, we have to correctly guess 2 coin results, with each at a probability of $\frac{1}{n}$, the probability of $X_{i,j} = (\frac{1}{n})^2 = \frac{1}{n^2}$.
The total number of pairs where it is possible to select 2 would be given by $\binom{n}{2} = \frac{n(n-1)}{2}$ possible pairs. We can calculate the possibility now with

$$\begin{aligned} E[X_{i,j}] &= \sum_{1 \le i < j \le n} E[X_{i,j}] \\ &= \sum_{1 \le i < j \le n} \frac{1}{n^2} \\ &= \frac{n(n-1)}{2}(\frac{1}{n^2}) \\ &= \frac{n-1}{2n} \end{aligned}$$

Therefore, the expected number of pairs of friends who swap coins, where i gets j's coin and j gets i's coin, is $\frac{n-1}{2n}$.

- (e) Say that you lose a random subset of $\frac{n}{2}$ coins on the way back, and you randomly distribute the remaining coins to a random subset of $\frac{n}{2}$ of your friends. Now use a union bound to get an upper bound on the probability that at least one of your friends gets their own coin back.

When we lose the first subset of $\frac{n}{2}$, that means we already have a 50% less chance of receiving our coin for any member of n. Once we get back, we further split the friends into another $\frac{n}{2}$ subset, which means we have $\frac{1}{\frac{1}{n}} = \frac{2}{n}$. Therefore, the probability that a given friend gets their own coin back is $\frac{1}{n}$.

To create a union bound, we find the probability that at least one of the $\frac{n}{2}$ friends gets their coin back. The union bound will say $P(\text{at least 1 friend gets his coin}) \le \sum_{i=1}^{\frac{n}{2}} P(\text{friend } i \text{ gets their coin})$.

$$P(\text{at least one friend gets their coin}) \le \sum_{i=1}^{\frac{n}{2}} \frac{1}{n}$$

$$P(\text{at least one friend gets their coin}) \le \frac{1}{2}$$

Therefore by the union bound we see there is at most a 50% chance that at least one friend gets their own coin back in this case.

2. Dynamic programming on a directed graph can play a part in speech recognition. A directed graph $G = (V, E)$ with labeled edges forms a formal model of a person speaking a restricted language. Each edge $(u, v) \exists V$ corresponds to a possible sequence of sounds produced by the model, with the label of a part being the concatenation of the labels of edges on that path.

   - (a) Describe an efficient algorithm that, given an edge-labeled directed graph G with distinguished vertex v_0 and a sequence $s =< o_1, o_2, \ldots, o_k >$, of sounds from $\sum$, returns a path in G that begins at v_0 and has s as it's label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (Hint: You may find concepts from Chapter 20 useful)

   We will start by defining a table $DP[v][i]$ which will represent whether there exists a valid path from vertex v matching the sequence $< \sigma_1, \ldots, \sigma_n >$.

   The first step is to solve the sub-problem whether the path exists in the first place. For this, we simply will start at the initial vertex v_0, defined on a directed graph as the vertex that contains only outgoing paths.
   Next, we simply check each outgoing path from the current v_i if there is a path taking us to the next v_i corresponding to the next item in the sequence, σ_i.

   If at any point in this process we scan every single path and do not find one corresponding to the next σ_i, we will return NO-SUCH-PATH.
   If we reach the path σ_{i-1} and there exists a path to σ_i, we return the full path that we found as a valid solution.

   Now that we've defined our base problem and recursive step, we can implement a dynamic approach to improve the efficiency. Rather than continuing down the path for every single iteration, we will instead intentionally explore every single path connecting every single vertex in our problem space.
   For each corresponding sector in our DP table, we will fill in 1 if a path exists between those two sounds. So if a path exists from v_i to v_j then we will set $DP[v_i][i] = 1$. Otherwise it will be 0.

   After our dynamic approach is implemented, we will have a runtime of $O(k(|V| + |E|))$ where k represents the length of the sound, V is the number of vertices, and E the number of outward edges. Because of our dynamic approach, our runtime will be improved to $O(k)$ for subsequent searches given the same problem space G. We will need to rebuild the entire DP table any time we introduce a new graph, G.

Now suppose that every edge, $(u, v) \exists E$ has an associated non negative probability $p(u, v)$ of being traversed, so that the corresponding sound is produced. The sum of probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. Think of the probability of a path beginning at vertex v_0 as the probability that a "random walk" beginning at v_0 follows the specified path, where the edge leaving a vertex u is taken randomly, according to the probabilities of the available edges leaving u.

- (b) Extend your answer to part (a) so that if a path is returned, it is a most probable path starting at vertex v_0 and having label s. Analyze the running time of your algorithm.

Since we already have our table DP, we just need to update our algorithm to calculate and store the probabilities of given paths as we fill in the table rather than a simple 1 or 0.

Let us define a new function $p(v, u)$ that will calculate the probability any given edge is chosen between two vertices. All values where there is not a possible edge will be initialized in the graph to 0. When filling out the graph, we will now use the relation; $DP[v][i] = \max(p(u, v) * DP[u][i + 1])$.

Here, we're assigning positive weight to each value in the table corresponding to the probability that vertex is chosen. We now simply need to scan the table for all possible paths for a given solution $< \sigma_1, \ldots, \sigma_n >$ and choose the one that results in the highest "score". This will be the most probable path.

This implementation will have the same time complexity of $O(k(|V| + |E|))$ for completing the table and $O(k)$ for solving for our solution, since we have only introduced a constant amount of additional work that we do on each iteration when building the table and searching for solutions in calculating the probability.

3. Dance, Dance Revolution (DDR) is played on a platform with 4 squares. You are given an input sequence σ over the symbols, A, B, C, and D, representing the four squares. In round $i \geq 1$, one of your feet must be on the square $\sigma[i]$. Your feet must always be in different squares, and you can move at most one foot at the start of each round to any new square. Your left foot starts in square A and right foot in square B.
You are a lazy dancer. Your goal is to maximize the following laziness score: the number of rounds in which neither foot moves.

- (a) Write a recurrence relation for the value $m(i, \ell, r)$ which gives the maximum score possible on the first i symbols of σ if your left foot ends in square ℓ and your right foot ends in square r.

Initially, our left foot is on square A and right on B, so we get our base case for laziness score $m(0, A, B) = 0$

$$m(i, \ell, r) = \begin{cases} m(i-1, \ell, r) + 1, & \text{if } \sigma[i] \in [\ell, r] \\ \max(m(i-1, \sigma[i], r), m(i-1, \ell, \sigma[i])), & \text{if DNE } [\ell, r] \end{cases}$$

If the current symbol $\sigma[i]$ is either ℓ or r, both feet stay in their current position and the laziness score is incremented by 1.
If neither foot is on $\sigma[i]$, we must move one of our feet to that position. We choose the maximum score possible by moving either of our feet, and in this case we cannot increase our laziness score.

- (b) Describe a dynamic program to return the max score for any input σ of length n based on your recurrence. What are the dimensions of your table? How do you fill it in? What is the final value returned? What is the runtime of your algorithm?

Lets create a 3 dimensional table $DP[i][\ell][r]$ where i stores the number of rounds processed, ℓ the left foot position, and r the right foot. The table will end up with dimensions $n \times 4 \times 4$ since the number of possibilities for ℓ, r are constant at 4.

Starting at position $(0, A, B)$ and round $i = 0$, we update the table for every combination of ℓ & r based on the recurrence in part (a). Once DP is filled, the final value will be the maximum score that we can achieve when moving through the table incrementing i from $1..n$. This return result will then be the laziness score for the ideal set of movements when given a particular input sequence of dance steps.

The runtime of our algorithm when creating the table will be $O(n)$, since two of the dimensions are stagnant. The runtime of the algorithm for subsequent runs will also remain at $O(n)$. While the subsequent runs after table creation will be a constant fraction of the runtime of the table creation phase, this speed increase is absorbed into the rules of big O notation.

4. You are given n balloons. Each balloon is painted with a number on it represented by an array $nums$. You are asked to burst all the balloons. If you pop the balloon at index i; and ℓ is the index of the closest un-popped balloon to the left, r is the index of the closest un-popped balloon to the right, you get a number of coins equal to $nums[\ell] * nums[i] * nums[r]$. If ℓ or r goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it. Your goal is to return the maximum coins you can collect by bursting the balloons wisely.

   - (a) Give an example input showing that if every step, you greedily select the balloon that gives you the largest number of coins for that step, this may not maximize the total number of coins.

   In a greedy algorithm, we will simply choose the balloon i such that the resulting $nums[\ell] * nums[i] * nums[r]$ is the largest, and continue downward to the base case. To illustrate, lets define an example $nums = [3, 1, 5, 8]$.

$$\begin{aligned} nums = [3,1,5,8],\ i = 3 &\implies 1 * 8 * 1 = 8 \\ nums = [3,1,5],\ i = 2 &\implies 8 + 1 * 5 * 1 = 13 \\ nums = [3,1],\ i = 0 &\implies 13 + 1 * 3 * 1 = 16 \\ nums = [1],\ i = 0 &\implies 16 + 1^3 = 17 \end{aligned}$$

   So with this algorithm we will end up with a total of 17 coins.

   - (b) To set up a dynamic program, describe in words the smaller problem(s) whose solutions can help you solve the big problem.

   The sub problem we have here is; Given two indices, ℓ, and r, find the maximum number of coins that can be obtained by bursting all balloons between those points.

   - (c) Write a recurrence relation for the dynamic program

   We will create a table $DP[\ell][r]$ representing the maximum number fo coins that can eb collected by bursting all the balloons between indices ℓ and r exclusively.
   The recurrence relation will be $DP[\ell][r] = \max_{\ell \le i \le r}(DP[\ell][i] + DP[i][r] + nums[\ell] \times nums[i] \times nums[r]])$

   We also have the base case $DP[\ell][r] = 0$ when $r - \ell \le 1$, since at this point there are no balloons between ℓ and r. Note also that our table will only actually end up being similar to that of an upper triangular matrix since r must always be greater than ℓ.

- (d) Describe the dynamic program to give the runtime

Since we are using a DP table, our runtime is primarily bounded by the time it takes to evaluate each position in this table on the first run. Here, $DP[\ell][r]$ where ℓ and r represent two points on the array $nums$ will end up taking $(n+2)(n+2) = O(n^2)$ time.

After we've created the table, it will also take $O(n)$ time to parse the information for a given input to calculate the total value for each input. Therefore on the first iteration we will run in $O(n^3)$ time, but subsequent runs will run in $O(n)$ time, until we give a new input array.

Attached below is the full DP table after we calculate our recurrence for each element in $nums = [3, 1, 5, 8]$. We see here that the optimal solution of 280, much better than what we found in (a).

$\ell \backslash r$	0	1	2	3
0	0	9	50	280
1	0	0	5	208
2	0	0	0	200
3	0	0	0	0