CIS 5511. Programming Techniques

Probabilistic Analysis

This lecture covers two topics: (1) analyzing a deterministic algorithm using probability theory, and (2) analyzing an algorithm which makes probabilistic choices.

1. Average cost

For a given problem *S* and an algorithm solving it, if each instance of size *n* s_i (i = 1..m) has a cost c_i, then the best-case and the worst-case cost (also for *n*) are *Minimum*(c_i, *i = 1..m)* and *Maximum*(c_i, *i = 1..m)*, respectively, while the average-case cost is $\sum p_i c_i$, where i = 1 .. m, and p_i is the probability for s_i to be taken as the input of the algorithm. In the current context, the probability p_i is defined by $\text{Lim}_{[t \to \infty]} (t_i / t)$, where *t* is the total number of times the algorithm is used on instances of that size, and t_i is the number for instance s_i to occur in the process, i.e., its occurrence frequency.

As a special situation, when all instances have the same probability to show up, that is, $p_i = p$. Since $\sum_{[i = 1..m]} p_i = 1$, then $\sum_{[i = 1..m]} p_i = mp = 1$, so $p = 1/m$, and the average cost $\sum_{[i = 1..m]} p_i c_i = (\sum_{[i = 1..m]} c_i) / m$.

To get the probability values is not always easy. Usually, they are based on the knowledge and assumptions about the problem instances to be processed, so are application-dependent and not a property of the algorithm itself.

2. Example: the hiring problem

For example, the following "algorithm" is used to hire a new office assistant right away as far as the new candidate is the best so far:

```
HIRE-ASSISTANT(n)
1  best = 0          // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

There are some unusual properties in this example:

- As a practical problem, the steps are not fully formalized.
- The goal is to analyze the cost of the procedure, reflected by line 3 and 6, not the

running time of the algorithm.
- The input does not have to be available at the same time (at the very beginning), and the output does not have to be available at the same time (at the very end), neither.

Even so, this algorithm can be analyzed using a similar method. Assuming the cost of interviewing and hiring is c_i and c_h, respectively, then the cost of the above algorithm is $O(n c_i + m c_h)$, where m is the number of hiring. Since the interview cost $n c_i$ remains unchanged, our analysis will focus on the hiring cost $m c_h$.

Obviously, for this problem the best case happens when the best candidate comes first (m = 1), and the worst case happens when every candidate is better than the previous ones (m = n).

To calculate the average hiring cost, we assume that the candidates come in *random* order, meaning that each possible order is equally likely. Please notice that such an assumption is different from a "complete ignorance" or "pure arbitrary" assumption --- we don't know which input the algorithm will meet each time, but we do assume that in the long run all possibilities will happen equally often. In general, it is not always valid to treat a variable with an unknown value as a *random* variable.

To calculate the probability for line 6 to be executed in the above algorithm, we can see that candidate *i* is hired, exactly when he/she is better than all candidates *1* through *i – 1*. Assuming any one of the first *i* candidates is equally likely to be the best so far, candidate *i* has probability of *1/i* to be hired. The expected number of hires is $\sum_{[i = 1..n]} 1/i = \ln n + O(1)$, which is $O(\ln n)$ (Page 1142, equation A.9).

3. Randomized algorithm

We cannot always assume that the input of an algorithm contains random variables with a known probabilistic distribution. For example, in the Hire-Assistant algorithm, when the coming order of candidate cannot be assumed to be random, the above average cost estimation is no longer valid.

One way to avoid this situation is to actually *impose* a probabilistic distribution onto the input, so as to turn it into a random variable. The common way to do so is to use a pseudo-random-number generator to rearrange the input. In this way, no matter how the input is presented to the system, the average computational cost is guaranteed to fit the probabilistic analysis. Such an algorithm is called "randomized". Please note that this solution assumes that all candidates are all available for processing at the very beginning, as well as that the hiring process will be repeated for many times.

For such a randomized algorithm, no particular input always has the best-case, or the worst-case, of time cost.

Many randomized algorithms work by shuffling the given input array *A*. One way to do so is to assign each element in the array *A[i]* a random priority *P[i]*, and then sort the elements of *A* according to their priorities:

```
PERMUTE-BY-SORTING(A)
1  $n = A.length$
2  let $P[1..n]$ be a new array
3  for $i = 1$ to $n$
4      $P[i] = \text{RANDOM}(1, n^3)$
5  sort $A$, using $P$ as sort keys
```

The range [1, n^3] is chosen to reduce duplicates in the random numbers generated.

A better method for generating a random permutation is to permute the given array in place. In iteration *i*, the element to be put at *A[i]* is chosen randomly from subarray *A[i..n]*, then remain unchanged.

```
RANDOMLY-PERMUTE(A, n)
1  for $i = 1$ to $n$
2      swap $A[i]$ with $A[\text{RANDOM}(i, n)]$
```

It can be proven that both algorithms generate random permutation as desired. For why the Random(i, n) cannot be replaced by Random(1, n), see this explanation.

4. Example: the on-line hiring problem

As a variant of the hiring problem, suppose now we can only hire once immediately after an interview (so we may not interview all the candidates). This is an example of online algorithms.

For this new problem, what is the trade-off between minimizing the amount of interviewing and maximizing the quality of the candidate hired?

Let's assume that after an interview we can assign the candidate a score, and no two candidates have the same score. One solution is: first select a positive integer $k < n$, interview but reject the first *k* candidates, then hire the first candidate thereafter who has a higher score than the best of the first *k* (which is also the best of all the preceding) candidates. If no such one can be found, hire the last one.

```
ONLINE-MAXIMUM(k, n)
1  $best\text{-}score = -\infty$
2  for $i = 1$ to $k$
3      if $score(i) > best\text{-}score$
4          $best\text{-}score = score(i)$
5  for $i = k + 1$ to $n$
6      if $score(i) > best\text{-}score$
7          return $i$
8  return $n$
```

Now the problem is to decide the best choice of *k*. Here we define it as the value that gives the highest probability for the best candidate to be hired, Pr{S}. This event is the summation of the events where the hiring happens when the best candidate is at position *i*: Pr{S} = $\sum_{[i = 1..n]}$ Pr{S_i}. Since the first *k* candidates are all rejected, it is actually $\sum_{[i = k+1..n]}$ Pr{S_i}.

The event S_i happens if and only if (1) the best candidate is at position *i*, and (2) nobody before it is better than the best of the first *k* candidates. Under the assumption of equal probabilities, the first value is *1/n*, and the second one is *k/(i – 1)*. In summary,

$$\begin{aligned}
\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\
&= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} \, .
\end{aligned}$$

It can be proved (see textbook page 152) that Pr{S} is maximized when *k* = *n/e*, with the value *1/e* (about 37% or 1/3).

Please note that the above Pr{S} is just one of the possible ways to define the optimal result.