Online Algorithms

1 Intro to Online Algorithms

An **online algorithm** is used when irrevocable decisions must be made with only partial knowledge of the input (often because the rest of the input hasn't arrived yet). With approximation algorithms, the algorithm outputs suboptimal solutions because there are not enough resources to solve the problem optimally (usually time because the problem is **NP**-hard). In contrast, online algorithms output suboptimal solutions because it doesn't know what inputs may occur in the future.

1.1 Areas of Interest

- Profit-maximizing job selection: jobs arrive over time with value unknown
- Data structure management and caching: requests for items arrive over time with future requests unknown
- Machine scheduling: jobs arrive over time with future durations unknown
- Routing of network traffic: packets/cars arrive over time with future network conditions unknown
- Multi-robot routing: tasks arrive over time with future task locations unknown

1.2 Competitive Analysis

Let OPT be the optimum solution for the whole input with full hindsight. Then, the analogy to the approximation ratio of approximation algorithms is the **competitive ratio**.

Definition 1. Competitive ratio: An online algorithm A is c-competitive is there is some α such that $\text{cost}(A) \leq c * \text{cost}(\text{OPT}) + \alpha$. A is **strictly** c-competitive if this condition holds with $\alpha \leq 0$. The competitive ratio of A is the infimum of c such that A is c-competitive.

Online algorithm instances usually have multiple parameters where one or more of them grows with time (like the number of jobs or requests) while the others stay constant. We would like c to not depend on the growing parameter – either c is constant or depends on a constant parameter.

2 The Surfboard Rental Problem

Imagine that you've just moved to Los Angeles from the frozen wasteland of Ithaca, NY and are excited to start surfing. However, all is not well – it's shark attack season, and each day you go surfing, a great white shark eats you with probability p. Because p can be nonzero, you don't know how many days d you'll be surfing until you get eaten. If surfboards can be rented at cost R or purchased at cost $P > R$ (and once purchased, it can be used forever for free), when is the optimal time to purchase a surfboard?

The offline algorithm here is trivial because we would know d. Thus, if $dR > P$ we purchase a surfboard on day one; otherwise, we rent every day. If we call the offline algorithm A, it's clear that $\text{cost}(A) = \min(dR, P)$.

The online algorithm is more subtle, but rather elegant. Notice that each possible online algorithm here is fully characterized by the purchase time $t \in \{\mathbb{N} \cup \infty\}$. If we call the online algorithm which purchases at time t B_t, it's clear that $\text{cost}(B_t) = R(t-1) + P$ if $t \leq d+1$.

For $t = \infty$, the competitive ratio is ∞; otherwise, the worst-case scenario is that we buy the surfboard on the day we get eaten, so $d = t$. Then, we can rephrase $\text{cost}(B_t) = R(d-1) + P$.

$$\frac{\text{cost}(B_t)}{\text{cost}(A)} = \frac{R(d-1)+P}{\min(dR, P)}$$

$$= \max(\frac{R(d-1)+P}{dR}, \frac{R(d-1)+P}{P})$$

$$= \max(1 + \frac{P-R}{dR}, 1 + \frac{R(d-1)}{P})$$

$$= 1 + \max(\frac{P-R}{dR}, \frac{R(d-1)}{P})$$

Because $d = t$ and we get to choose t, we effectively get to choose d. Notice that the first term in the max decreases with d, while the second term increases with d. So, to minimize the maximum, we set them equal.

$$\frac{P-R}{dR} = \frac{R(d-1)}{P}$$

$$dR(dR - R) = P(P - R)$$

These are of the same form, so:

$$dR = P$$

$$d = \frac{P}{R}$$

Therefore, we should purchase the surfboard on the day when the cumulative rental cost reaches the purchase cost. For example, if it costs \$20 to rent and \$200 to purchase, we rent for 10 days and purchase on the 11th day. Then, to determine c:

$$\frac{\text{cost}(B_t)}{\text{cost}(A)} = 1 + \frac{P-R}{dR}$$

$$= 1 + \frac{P-R}{P}$$

$$= 2 - \frac{R}{P}$$

Thus $c = 2$ and $\alpha = -\frac{R}{P}$, so B_t is better than 2-competitive.

Remark 2. For a deterministic algorithm, we can analyze as though the adversary observes the algorithm in action – in fact, it can predict the full behavior, and design the entire input ahead of time. So if the shark was really smart and could do these calculations, he could choose to eat us on the day we buy our surfboard. The solution to this is randomization – it lowers the predictablity of the algorithm, and can allow better competitive ratios.

Remark 3. Competitive worst-case analysis is only one way of analyzing online algorithms. One could also make statistical assumptions about the input: for example, if we knew the distribution of shark attack probabilities, we could design an algorithm which takes advantage of that information.

Remark 4. There are several variations of the surfboard rental problem (sometimes called the ski rental problem). For example, the Bahncard problem reflects the German metro system, where one may purchase a membership card for a fee F and receive an $N\%$ discount on all metro rides for a year. However, all these variations can be reduced to the basic rental problem.

3 The Porta-Potty Problem

3.1 The Basic Porta-Potty Problem

Imagine you're at Coachella and you really, really need to pee. There are 100 porta-potties at the concert, with varying levels of cleanliness c_i. The c_i can be chosen by an adversary, but the order in which the porta-potties appear is uniformly random. We want to maximize our probability of picking the cleanest porta-potty (or equivalently, picking the porta-potty which gives us the highest expected value of cleanliness), but we can only check porta-potty cleanliness one at a time, and our decision is irrevocable.

To succeed with probability $\frac{1}{4}$, we without exception reject the first half of the input, but remember the cleanest porta-potty we saw. This effectively calibrates our algorithm. Then, while checking the second half, we choose the first porta-potty with higher cleanliness than the one we remember.

Failure $(\frac{1}{2})$: The cleanest porta-potty is in the first half we checked.

Failure $(\frac{1}{4})$: Both the second-cleanest and the cleanest porta-potties are in the second half we checked.

Success $(\frac{1}{4})$: The second-cleanest porta-potty is in the first half we checked, and the cleanest porta-potty is in the second half.

To succeed with a probability $\frac{1}{e}$, we reject the first $\frac{1}{e}$ of the input and follow the algorithm for the remainder. This problem is also called the "secretary model".

3.2 Generalization: The Matroid Porta-Potty Problem

Given a matroid on n elements, we may include any independent set of elements. Values are adversarially assigned to elements and revealed in uniformly random order. When we see an element, we may irrevocably choose to include it. It is an open question whether there is a constant-competitive algorithm to solve this problem.