CSE586: Algorithms Under Uncertainty
Homework 2 Solutions

Ashlesha Gupta (2021380) Divyajeet Singh (2021529)

Solution 1.

Part (a)

We are given a cycle graph of n vertices. The given algorithm (let's say $\mathbf{\Lambda}$) breaks the cycle by removing an arbitrary (but fixed) edge. To show that the competitive ratio of the $\mathbf{\Lambda}$ is $\Omega(n)$, we simply provide an example where it performs n times worse than an offline optimal algorithm.
Consider a cycle graph of n vertices with $k = 2$. We label the vertices in order as $v_1, v_2, \ldots, v_n$. Without loss of generality, let's say $\mathbf{\Lambda}$ removes the edge (v_n, v_1). So, we have the following scenario.

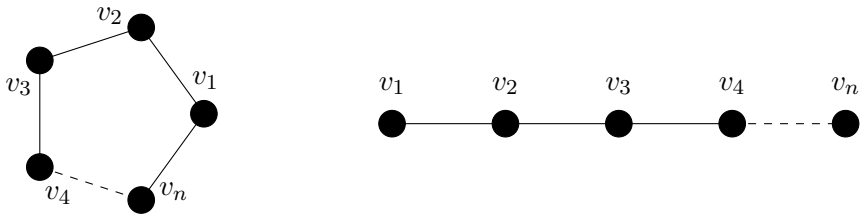

Figure 1: Graphs used by the optimal and the given algorithm.

Let the servers be initially located at the vertices v_1 and the *middle* vertex[1] $v_{\frac{n}{2}}$. An adversary can take advantage of the knowledge of the removed edge - consider the input sequence $\sigma = \langle v_1, v_n, v_{\frac{n}{2}}, v_1, v_n, v_{\frac{n}{2}}, \ldots \rangle$.
Let us analyze the cost incurred by both the algorithms on σ.
Evaluate the performance of $\mathbf{\Lambda}(\sigma)$. First, the server at v_1 stays. To serve the request at n, a server must move from $v_{\frac{n}{2}}$ to v_n, incurring a cost of $\frac{n}{2}$. Then, by the double cover algorithm, both the servers move to the *middle*[2] incurring a cost of $2 \cdot \frac{n}{2} = n$. After this, each request at v_1 and v_n costs $\frac{n}{2}$ (since before this, these servers must be in the middle), and each request at $v_{\frac{n}{2}}$ costs n, since a request in the middle forces both servers to jump to the middle. So, we have the total cost

$$\mathbf{\Lambda}(\sigma) = \left(0 + \frac{n}{2} + n\right) + \frac{|\sigma| - 3}{3} \cdot \left(\frac{n}{2} + \frac{n}{2} + n\right) \tag{1}$$

$$\leq \frac{|\sigma|}{3} \cdot \left(\frac{n}{2} + \frac{n}{2} + n\right) = \frac{2n|\sigma|}{3} \tag{2}$$

In contrast to this, the optimal algorithm $\mathbf{OPT}$, working on a circle, can keep one circle to alternate between v_1 and v_n, and the other circle to stay fixed at $v_{\frac{n}{2}}$. So, the cost incurred by $\mathbf{OPT}(\sigma)$ is

[1] If n is odd, the middle vertex is $v_{\frac{n+1}{2}}$ and if n is even, the middle vertex is $v_{\frac{n}{2}}$.

[2] When n is odd, both servers move to $v_{\frac{n+1}{2}}$, and otherwise, one server moves to $v_{\frac{n}{2}}$ and the other moves to $v_{\frac{n}{2}+1}$.

$$\mathbf{OPT}(\sigma) = (0+0+1) + \frac{|\sigma|-3}{3} \cdot (1+0+1) \tag{3}$$

$$\leq \frac{|\sigma|}{3} \cdot (1+0+1) = \frac{2|\sigma|}{3} \tag{4}$$

Thus, we have the competitive ratio

$$\frac{\mathbf{\Lambda}(\sigma)}{\mathbf{OPT}(\sigma)} = \frac{2n|\sigma|}{3} \cdot \frac{3}{2|\sigma|} = n \tag{5}$$

Hence, the competitive ratio of the given algorithm is at least $\Omega(n)$.

Part (b)

The given algorithm (let's say $\mathbf{\Lambda}$) breaks the cycle by removing any edge uniformly randomly and performs the double cover algorithm on the resulting line. We need to prove that $\mathbf{\Lambda}$ is $\mathcal{O}(k)$-competitive.
The expected cost of $\mathbf{\Lambda}$ on any input sequence σ will be

$$\mathbb{E}[\mathbf{\Lambda}(\sigma)] = \sum_{i=1}^{n} \frac{1}{n} \cdot \mathbf{\Lambda}_i(\sigma) \tag{6}$$

where $\mathbf{\Lambda}_i$ is the algorithm that breaks the cycle by removing the edge (v_i, v_{i+1}), for $i = 1, 2, \ldots, n-1$ and (v_n, v_1) for $i = n$, and then performs the double cover algorithm on the resulting line. Let the line formed by $\mathbf{\Lambda}_i$ be l_i.
There is an equivalent view point of (6). The cost paid by each line l_k is weighed by $\frac{1}{n}$. It is equivalent to say that for each requested vertex $\sigma_j (1 \leq j \leq |\sigma|)$, we move $\left(\frac{1}{n}\right)^{th}$ of a server from each line to the requested vertex. For the lines l_k where the requested vertex is between two serves (i.e. double coverage applies), it is implicit that $\left(\frac{1}{n}\right)^{th}$ of at most 2 servers will be moved to σ_j, i.e. the total movement is $2 \cdot \left(\frac{1}{n}\right)$ in some sense.
The analysis of this point of view of the algorithm will be done in a similar way as the analysis of the double coverage algorithm covered in class. We define a non-negative potential function Φ such that when the optimal algorithm $\mathbf{OPT}$ makes a move (incurs a cost x), $\Delta\Phi \leq 2k \cdot x$, and when $\mathbf{\Lambda}$ incurs a cost x, the difference is $\Delta\Phi \leq -x$. We define

$$\Phi = 2k\Psi + \Theta \tag{7}$$

where Ψ is the weighted distance between the servers of $\mathbf{OPT}$ and $\mathbf{\Lambda}$ and Θ is the weighted sum of the distances of pairs of servers of $\mathbf{\Lambda}$ weighted by the fractions they move. It is easy to see that Ψ changes only when $\mathbf{OPT}$ makes a move, and changes only by integer amounts, since $\mathbf{OPT}$ moves a server from one vertex to another.

$$\Psi = \sum_{i=1}^{k} d(s_i^*, s_i) \cdot w_{s_i} \tag{8}$$

$$\Theta = \sum_{\text{unique } (s_i, s_j)} d(s_i, s_j) \cdot w_{s_i} \cdot w_{s_j} \tag{9}$$

where s_i^* is the location of the i^{th} server of $\mathbf{OPT}$, s_i is the location of the i^{th} server of $\mathbf{\Lambda}$, and w_{s_i} is the weight of servers at s_i. So, Ψ measures the cost of moving one server in $\mathbf{OPT}$.
Let's say $\mathbf{OPT}$ incurs a cost x when it makes a move. Then, Θ remains unchaged. However, Ψ can increase by at most x. This is because $\mathbf{OPT}$ moves a server from one vertex to another, so, the weighted distance

between the servers of $\mathbf{OPT}$ and $\mathbf{\Lambda}$ can increase by at most x. In this case, the potential difference is $\Delta\Phi \leq 2k \cdot x$, i.e. the potential increases by at most $2k \cdot x$.
Now, let's say $\mathbf{\Lambda}$ incurs a cost x. There can be two cases for each of the n lines. Either the requested vertex will lie to the extreme left or right of the line, or it will lie between two servers.
Let's say that the requested lies to the extreme sides of p out of the n lines. Then, there are p lines on which a server moves from one vertex to another with a weight of $\frac{1}{n}$. x is the total distance moved to serve the request, so Ψ decreases by at most $\frac{p}{n} \cdot x$ (because p servers move by $x \cdot \frac{1}{n}$ each). Also, Θ increases by $\frac{p}{n}(k - \frac{p}{n}) \cdot x$, because the distance of the p servers moved from the other $k - \frac{p}{n}$ servers increases by $x \cdot \frac{p}{n}$ each. So, the potential difference in total is

$$\Delta\Psi_p \leq -\frac{p}{n} \cdot x \tag{10}$$

$$\Delta\Theta_p \leq \frac{p}{n}\left(k - \frac{p}{n}\right) \cdot x = kx \cdot \frac{p}{n} - x \cdot \frac{p^2}{n^2} \tag{11}$$

$$\implies \Delta\Phi_p = 2k \cdot \Delta\Psi_p + \Delta\Theta_p \tag{12}$$

$$= -2k \cdot x \cdot \frac{p}{n} + k \cdot x \cdot \frac{p}{n} - x \cdot \frac{p^2}{n^2} \tag{13}$$

$$\leq -2k \cdot x \cdot \frac{p}{n} + 2k \cdot x \cdot \frac{p}{n} - 2x \cdot \frac{p^2}{n^2} \tag{14}$$

$$= -2x \cdot \frac{p^2}{n^2} \tag{15}$$

At the same time, there are $n - p$ on which the requested vertex lies between two servers. On these lines, the servers move $\frac{2}{n}$ each by the property of double coverage - 2 servers are moving $\frac{1}{n}$ on each line. In this case, Ψ either decreases or remains the same. This can be understood by realizing that the distance that increases by the servers on the left of the requested vertex is the same as the distance that decreases by the servers on the right of the requested vertex, and vice versa. So, $\Delta\Psi \leq 0$. On the other hand, if the total distance moved is x, and 2 servers move $\frac{1}{n}$ each, then Θ decreases by $\left(\frac{n-p}{n}\right)^2 \cdot x$. This is because the the total weight of the servers moved is $\left(\frac{n-p}{n}\right)^2$, and x is the total distance moved. So, the potential difference is

$$\Delta\Psi_{n-p} \leq 0 \tag{16}$$

$$\Delta\Theta_{n-p} \leq -\left(\frac{n-p}{n}\right)^2 \cdot x \tag{17}$$

$$\implies \Delta\Phi_{n-p} = 2k \cdot \Delta\Psi_{n-p} + \Delta\Theta_{n-p} \tag{18}$$

$$\leq -\left(\frac{n-p}{n}\right)^2 \cdot x \tag{19}$$

So, the overall potential difference is

$$\Delta\Phi = \Delta\Phi_p + \Delta\Phi_{n-p} \tag{20}$$

$$= -2x \cdot \frac{p^2}{n^2} - \left(\frac{n-p}{n}\right)^2 \cdot x \tag{21}$$

$$= -\frac{x}{n^2}\left(2p^2 + p^2 + n^2 - 2np\right) \tag{22}$$

When $p = 0$, $\Delta\Phi = -\frac{x}{n^2} \cdot n^2 = -x$. When $p = n$, $\Delta\Phi = -\frac{x}{n^2} \cdot 2n^2 = -2x$. Clearly, $-2x \leq \Delta\Phi \leq -x$ when $\mathbf{\Lambda}$ makes a move.
We have proved that $\Delta\Phi \leq 2k \cdot x$ when $\mathbf{OPT}$ incurs a cost x and $-2x \leq \Delta\Phi \leq -x$ when $\mathbf{\Lambda}$ incurs a cost x. The existence of this potential function Φ proves that $\mathbf{\Lambda}$ is $\mathcal{O}(k)$-competitive.

Solution 2.

Part (a)

We are required to follow a greedy strategy to maintain a maximum-size subset of disjoint intervals while selecting intervals whose lengths are in $[2^l, 2^{l+1})$ for some non-negative integer l.

Claim 1. *The greedy algorithm, let's say* $\mathbf{\Lambda}$*, for picking intervals of length* $[2^l, 2^{l+1})$ *is* $\frac{1}{3}$*-competitive.*

Proof. To prove the claim, we only need to show that for any input sequence σ,

$$\frac{\mathbf{\Lambda}(\sigma)}{\mathbf{OPT}(\sigma)} \geq \frac{1}{3} \tag{23}$$

where $\mathbf{ALG}(\sigma)$ represents the size of the subset of intervals picked by the algorithms on an input seqeunce σ. It is easy to see that picking smaller intervals is *better*. Since $\mathbf{\Lambda}$ picks greedily, every time it selects an interval, it potentially blocks upto 3 smaller intervals that $\mathbf{OPT}$ would select. In the worst case, $\mathbf{\Lambda}$ selects one interval of length $2^{l+1} - 1$ while $\mathbf{OPT}$ picks 3 intervals of length 2^l each. This may happen for some $k \leq \frac{|\sigma|}{3}$ times, where each time, $\mathbf{\Lambda}$ picks a subset and blocks three subsets of *half* the size. So, we have

$$\frac{\mathbf{\Lambda}(\sigma)}{\mathbf{OPT}(\sigma)} = \frac{k}{3 \cdot k} = \frac{1}{3} \tag{24}$$

This proves that the claim in (23). □

Solution 3.

Part (a)

We need to modify the exponential update algorithm to show a competitiveness of $\mathcal{O}(\log f_{\max})$ where $f_{\max}$ is the *maximum* number of subsets to which any element can belong. Our proposed algorithm is given in 1. Here is an explanation of the algorithm.

Algorithm 1 A $\mathcal{O}(\log f_{\max})$ Exponential Update Algorithm for Fractional Set Cover

$f_S \leftarrow 0 \quad \forall S \in \mathcal{F}$
while elements $e \in U$ are arriving **do**
  If all sets $S_e : e \in S_e$ have $f_{S_e} = 0$, then $f_{S_e} \leftarrow \frac{1}{f_{\max}} \quad \forall S_e : e \in S_e$
  If $\exists S^* : e \in S^*$ with $f_{S^*} \neq 0$, then $f_{S_e} \leftarrow \left(\frac{1}{f_{\max}}\right)^{t+1} \quad \forall S_e : e \in S_e, f_{S_e} = 0$ where f_{S^*} was initialized to $\left(\frac{1}{f_{\max}}\right)^t$
  while $\sum_{S:e \in S} f_S < 1$ **do**
    $f_S \leftarrow \min\{1, 2f_S\} \quad \forall S : e \in S$
  end while
end while

The algorithm maintains a fraction f_S for each subset $S \in \mathcal{F}$, initialized to 0. When an element e arrives for the first time that has never been covered in any set, we set its fraction to $\frac{1}{f_{\max}}$ for all the subsets S_e that contain e.
If an element arrives that has already been covered (partially) in some subset, the algorithm assigns a fraction

of $\left(\frac{1}{f_{\max}}\right)^{t+1}$ to all the subsets S_e that contain e and have $f_{S_e} = 0$, where t is the power of $\frac{1}{f_{\max}}$ assigned to the subset S^* that contains e and has $f_{S^*} \neq 0$. Then, the algorithm simply follows the exponential update rule of doubling.
Now, we prove that Algorithm 1 is indeed $\mathcal{O}(\log f_{\max})$-competitive.

Claim 2. *Algorithm 1 is $\mathcal{O}(\log f_{\max})$-competitive.*

Proof. The proof is very simple. For simplicity, let's call the algorithm covered in class as $\mathbf{M}$ and the proposed algorithm as $\mathbf{\Lambda}$. The first modification is that the fractions f_S assigned to each subset S are initialized from 0, instead of $\frac{1}{m}$ as done in $\mathbf{M}$.
The intuition behind the algorithm is that if some subset has already been given some weight before, then instead of giving more weight to incoming/newer sets, we can increase the weight given to the older sets by doubling them (and setting the newer ones to higher powers of $\frac{1}{f_{\max}}$). This makes sure that the older sets get covered first, covering the new element more *quickly*.
Similar to the proof covered in class, in each (inner) iteration of $\mathbf{\Lambda}$, the weights corresponding to all the subsets in which the element e occurs will be doubled, which means that there cannot be any iteration in which no subset of $\mathbf{OPT}$ participates. Note that even if the element e occurs in previous subsets, every subset would have been initialized to **at most** $\frac{1}{f_{\max}}$, and therefore, when we double them, there can be **at most** $\mathcal{O}(\log f_{\max})$ iterations. □

An initialization of all weights to $\frac{1}{f_{\max}}$ does not work because

$$1 \leq f_{\max} \leq m \tag{25}$$

$$1 \geq \frac{1}{f_{\max}} \geq \frac{1}{m} \tag{26}$$

$$\implies m \geq m \cdot \frac{1}{f_{\max}} \geq 1 \tag{27}$$

which means that the total initialized weight can exceed 1. As it is easy to see, once we start doubling, we end up paying much more than an offline optimal algorithm.

Part (b)

For the rounding scheme, we assume that we know the value of $f_{\max}$. Let the final weights assigned to each subset S by the algorithm $\mathbf{\Lambda}$ in Part (a) be f_S. We multiply each f_S by $f_{\max}$ to get a value. If this value exceeds/equals 1, we round this subset up to 1. Otherwise, we round it down to 0.
Let us analyse this deterministic rounding scheme. The final integer set cover solution will have a cost of

$$C = \sum_{S \in \mathcal{F}} f_S \cdot f_{\max} = f_{\max} \sum_{S \in \mathcal{F}} f_S \tag{28}$$

$$\leq f_{\max} \cdot \mathcal{O}(\log f_{\max}) \cdot \mathbf{OPT}(\sigma) \quad \text{by the analysis in Part (a)} \tag{29}$$

$$= \mathcal{O}(f_{\max} \log f_{\max}) \cdot \mathbf{OPT}(\sigma) \tag{30}$$

Therefore, the cost paid by this rounding scheme is $\mathcal{O}(f_{\max} \log f_{\max})$.

Solution 4.

The given cat and mouse game is analogous to the Online Paging Problem that was discussed in class. The n vertices of the graph represent the pages[3]. The cache size can be assumed to be $n - 1$. We say that the

[3]The terms *vertex* and *page* are used interchangeably in this solution.

only page not in the cache is the one that the mouse is currently on. The cat behaves as an online adversary trying to hit a cache miss[4]. The mouse acts as the algorithm[5] trying to minimize its movement (i.e. the number of cache misses).

Part (a)

Given the constraint that each move costs 1 when $m_{t+1} \neq m_t$, we propose the following 1-bit marking algorithm for the mouse. We begin with all vertices unmarked. At each time step t, when the cat moves to any vertex c_t, one of two cases can arise:

1. $c_t \neq m_{t-1}$, i.e. c_t is *in cache*. In this case, we simply mark the vertex if it is not already marked. The mouse stays on $m_t = m_{t-1}$.

2. $c_t = m_{t-1}$, i.e. c_t is a cache miss. In this case, we mark c_t and the mouse moves to an arbitrary unmarked vertex m_t. If all vertices were marked, we unmark all of them and execute this step.

By **Theorem 3.** in [1], the marking algorithm is k-competitive where k is the cache size. So, the proposed algorithm is $(n-1)$-competitive.

Part (b)

We need to prove a claim similar to **Theorem 2.** in [1] for the given cat and mouse game.
Let us fix any deterministic algorithm $\mathbf{A}$ for the mouse. Note that the number of pages is fixed to n and the cache size is $k = n - 1$. This means that there exists exactly one page at each time step, in our case, m_{t-1}, that is not present in the cache. So, at every time step t, the cat can move to the vertex m_{t-1} (i.e. *request for page* m_{t-1}), forcing the mouse to move (to serve the cache miss), incurring a cost of 1 in each step. So, for an input sequence σ, $\mathbf{A}$ suffers a cost of $|\sigma|$.
However, an offline optimal mouse (something similar to LFD for paging) knows in advance the subsequent moves of the cat. So, for every cache miss, the optimal mouse moves to a vertex v where the cat would arrive the farthest in the future. The cat will reach v after at least $k-1$ or $n-2$ time steps. If the cat moves to v before $n-2$ requests that were *in the cache* when the mouse moved to v, then it wouldn't be the farthest-in-the-future vertex, as there would be some other vertex that is accessed later than v. Hence, the optimal mouse suffers a cost of at most $\frac{|\sigma|}{n-1}$.
Thus, the competitive ratio of any deterministic algorithm $\mathbf{A}$ is at least $n - 1 = \Omega(n)$.

Part (c)

Following the same argument as class notes, we propose the randomized marking algorithm as an $\mathcal{O}(\log n)$-competitive algorithm for the cat and mouse game. The description of the algorithm is as follows:

1. When the cat moves to a vertex $v_t \neq m_{t-1}$, the mouse stays at $m_t = m_{t-1}$ and marks v_t if it is not already marked.

2. Otheriwse, we mark v_t and the mouse moves to an unmarked vertex m_t chosen uniformly at random. If all vertices are marked, we unmark all of them and execute this step.

[4]By the problem definition, hitting a cache miss does not mean that the cat catches the mouse - it simply means that the mouse **must** move

[5]The terms *mouse* and *algorithm* are used interchangeably in this solution.

It is shown in **Theorem 4.** in [1] that the cost paid by an offline optimal mouse is bounded below by

$$\mathbf{OPT}(\sigma) \geq \sum_{i=1}^{p} \frac{d_i}{2} \tag{31}$$

where the mouse goes through p many *phases* and d_i is the number of vertices visited by the cat which are distinct from all the vertices visited by it in phase P_{i-1}. Let q_i be the number of times $c_t = m_{t-1}$, i.e. the number of times the cat hits a cache miss in phase P_i.
Now we bound the expected number of cache misses suffered by randomized marking. It is obvious that in each phase P_i, the randomized mouse suffers at least d_i misses, since these are new requests which are not in cache at the beginning of P_i. Other than these, there can be at most $n - 1 - d_i$ *old* vertices which the mouse may have visited (since they were unmarked at the beginning P_i) but then the cat visits again in P_i. The mouse picks these vertices to visit randomly.
Let us call the old vertices $o_1, o_2, \ldots, o_{n-1-d_i}$ according to their order of first visits by the cat in P_i. Define a random variable X_j as follows

$$X_j = \begin{cases} 1 & \text{if } o_j \text{ was a cache miss, i.e. the cat visited } o_j = m_{t-1} \\ 0 & \text{otherwise} \end{cases} \tag{32}$$

The total number of cache misses due to the *old* vertices is X, the sum of all X_j for $j = 1, 2, \ldots, k-1 = n-2$. So, the expected number of cache misses suffered by randomized marking is

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{j=1}^{n-2} X_j\right] = \sum_{j=1}^{n-2} \mathbb{E}[X_j] = \sum_{j=1}^{n-2} \mathbb{P}[X_j = 1] \quad \text{since } X_j \text{ are an indicator variables} \tag{33}$$

Now, we bound the probability that each indicator variable takes the value 1. The mouse will visit vertex o_1 because the cat visited its earlier vertex m_{t-1}. Let $\bar{n}_1$ be the number of vertices visited by the cat before its first visit to o_1 in P_1. So, we select a subset of size $\bar{n}_1$ from $n-1$ old vertices uniformly at random. So, the probability of o_1 being among these selected vertices is

$$\frac{\bar{n}_1}{n-1} \leq \frac{d_i}{n-1} \tag{34}$$

Similarly, let n_2 be the number of new vertices visited by the cat before its first visit to o_2. However, the mouse may have already visited o_2 because it was forced to leave vertex o_1 when the cat visited it. At this stage, the total number of unmarked pages is $n-2$, since o_1 would have been marked. So, the probability of the cat visiting o_2 when the mouse is at it becomes

$$\frac{n_2}{n-2} \leq \frac{d_i}{n-2} \tag{35}$$

Continuing this way, we get the probability of $o_j (j = 1, 2, \ldots, n-2)$ being a cache miss as

$$\mathbb{P}[X_j = 1] \leq \frac{d_i}{n-1-j} \tag{36}$$

Thus, we have the expected number of cache misses

$$\mathbb{E}[X] = \sum_{j=1}^{n-2} \mathbb{P}[X_j = 1] = \sum_{j=1}^{n-2} \frac{d_i}{n-1-j} = d_i \cdot \mathcal{O}(\log n) \tag{37}$$

Then, adding up over all phases P_i for $i = 1, 2, \ldots, p$, we get the upper bound on the expected number of misses for a randomized mouse, which is

$$\sum_{i=1}^{p}(d_i + d_i \cdot c \log n) \leq c \log n \cdot \sum_{i=1}^{p} d_i = \mathcal{O}(\log n) \cdot \mathbf{OPT}(\sigma) \tag{38}$$

where the last inequality follows from the lower bound on $\mathbf{OPT}(\sigma)$ in (31). This proves that the proposed randomized marking algorithm is $\mathcal{O}(\log n)$-competitive.

Solution 5.

For the given randomized algorithm for vertex cover with online edge arrivals, we need to prove that

$$\mathbb{E}[\mathbf{\Lambda}(\sigma)] \leq 2 \cdot \mathbf{OPT}(\sigma) \tag{39}$$

We try to estimate and bound the total expected cost paid by our algorithm on any input sequence, as compared to an offline optimal algorithm which knows the input of edges.
It is clear that $\mathbf{OPT}$ may or may not select an edge at every time step $t \leq |\sigma|$. So, let's say that $\mathbf{OPT}$ selects an edge at time steps $t_1, t_2, \ldots, t_N$ where $N \leq |\sigma|$ and $t_i < t_{i+1} \quad \forall i = 1, 2, \ldots, N-1$. Let the time steps $[t_i, t_{i+1})$ define a *phase*, say P_i, for every $i = 1, 2, \ldots, N-1$. In each phase P_i, $\mathbf{OPT}$ makes only 1 selection. So, we prove that in expectation, the number of vertices chosen by $\mathbf{\Lambda}$ in each phase is upper bounded by 2.
Let the number of vertices chosen in each phase P_i be S_i. Then, we have

$$\mathbb{E}[S_i] = \sum_{j=1}^{k} j \, \mathbb{P}[S_i = j] \quad \text{where } k = t_{i+1} - t_i, \text{ the no. of time steps in } P_i \tag{40}$$

Claim 3. *The probability that $\mathbf{\Lambda}$ chooses j vertices in phase P_i is*

$$\mathbb{P}[S_i = j] = \frac{1 + \mathbb{I}_{\{j=k\}}}{2^j} = \begin{cases} \frac{1}{2^j} & j \neq k \\ \frac{2}{2^j} & j = k \end{cases} \tag{41}$$

where k is the number of time steps in phase P_i, i.e. $k = t_{i+1} - t_i$.

Proof. First, it is easy to see that the distribution of probabilities is valid, i.e. it sums to 1. This is because

$$\sum_{j=1}^{k-1} \frac{1}{2^j} + \frac{2}{2^k} = \sum_{j=1}^{k-1} \frac{1}{2^j} + \frac{1}{2^{k-1}} \tag{42}$$

$$= \sum_{j=1}^{k-2} \frac{1}{2^j} + \frac{2}{2^{k-1}} \tag{43}$$

$$= \cdots = \frac{1}{2} + \frac{2}{2^2} = 2 \cdot \frac{1}{2} = 1 \tag{44}$$

Now, we prove the claim.
Let v^* be the vertex chosen by $\mathbf{OPT}$ in phase P_i. $\mathbf{\Lambda}$ picks any vertex of an input edge with half probability, which means that it picks v^* at time t_i with probability $\frac{1}{2}$. If it picks v^*, then it does not need to pick any other vertex till t_{i+1}, thereby matching $\mathbf{OPT}$. This means that $\mathbb{P}[S_i = 1] = \frac{1}{2^1}$.

If $\mathbf{\Lambda}$ does not pick v^* at t_i, then it picks v^* at $t_i + 1$ with half probability. This means that $\mathbb{P}[S_i = 2] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2^2}$. This continues till time[6] $t_{i+1} - 2$, i.e. $\mathbb{P}[S_i = j] = \frac{1}{2^j}, 1 \leq j < t_{i+1} - 1$.
At the last step of the phase, $t_{i+1} - 1$, it does not matter which vertex $\mathbf{\Lambda}$ picks; the size of the picked vertex set will be $k = t_{i+1} - t_i$. If it picks either, the phase ends anyway. So, there are two possibilities of making $S_i = k$. So, of $\mathbb{P}[S_i = k] = \frac{2}{2^k}$. □

Now we have

$$\mathbb{E}[S_i] = \sum_{j=1}^{k} j \, \mathbb{P}[S_i = j] = \sum_{j=1}^{k-1} j \cdot \frac{1}{2^j} + k \cdot \frac{2}{2^k} \tag{45}$$

$$= \sum_{j=1}^{k} \frac{j}{2^j} + \frac{k}{2^k} \tag{46}$$

$$\leq \lim_{k \to \infty} \sum_{j=1}^{k} \frac{j}{2^j} + \frac{k}{2^k} = 2 + 0 = 2 \tag{47}$$

where the summation is bounded by 2 (sum of arithmetico-geometric series, given in (48)), and the other limit is trivially true (exponential grows faster than linear).

$$\sum_{j=1}^{\infty} j \cdot \frac{1}{2^j} = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = \frac{1}{2} \cdot \frac{4}{2} = 2 \tag{48}$$

We have proved that in each phase, if $\mathbf{OPT}$ selects one vertex, then $\mathbf{\Lambda}$ selects at most 2 vertices in expectation. Summing over all N phases, we get

$$\mathbb{E}[\mathbf{\Lambda}(\sigma)] = \sum_{i=1}^{N} \mathbb{E}[S_i] \leq \sum_{i=1}^{N} 2 = 2 \cdot N \leq 2 \cdot \mathbf{OPT}(\sigma) \tag{49}$$

where the last inequality holds because $\mathbf{OPT}$ picks one vertex in each of the N phases by definition. This proves that the given randomized algorithm is $\mathbf{2}$-competitive.

References

1. Lecture Notes on Online Paging, CSE586 (Monsoon 2023, Dr. Syamantak Das)
2. Lecture Notes on Online Computation & Network Algorithms, CS294-1 (Spring 1997, Yair Bartal)
3. A Deterministic $\mathcal{O}(k^3)$-competitive k-Server Algorithm for the Circle, A. Fiat et al.
4. The k-Server algorithm and Fractional Analysis, Duru Turkoglu
5. Wikipedia - Arithmetico-Geometric Series

[6] Simply put, everytime we choose something wrong, the probability decreases by half and the size of chosen set of vertices increases by 1.