# Advanced Algorithms:
# Exam 1 Review (Partial) Solutions

*Dana Randall Spring 2024*

**Sarthak Mohanty**

## Exercise 1

A three-way cut is a set of edges such that removing it from the graph yields three connected components. Modify the min-cut algorithm discussed in class to get a three-way min cut. Compute the probability of success of your design and discuss how to boost it to be 1 - $o(1)$.

*Hint: Stop looking stuff up, you don't need to! Use the process from class (taking special care near the end.)*

## Solution

Let $G = (V, E)$ be a multigraph without self loops. For $e = (u, v) \in E$, define the contraction with respect to $e$ (denoted $G/e$) as in lecture. Now our algorithm is as follows:

Starting from the input graph $G = (V, E)$, repeat the following process until only *three* vertices remain:

1. Choose an edge $e = (u, v)$ uniformly at random from $E$.
2. Set $G = G/e$.

**Claim:** Let $\delta^*(S)$ be a three-way cut of minimum size of the graph $G = (V, E)$ with $|\delta^*(S)| = k$. Then the probability that the algorithm above ends with the cut $\delta^*(S)$ is at least $\frac{1}{\binom{n}{4} \cdot n}$.

*Proof.* Denote the edges we contract in the algorithm as $\{e_1, e_2, \ldots, e_{n-3}\}$. The algorithm succeeds if none of the contracted edges are in $\delta^*(S)$.

First note that for every intermediate multigraph, the sum of the degrees for any *pair* of vertices is at least $k$, otherwise we have a three-way cut of size smaller than $k$ by disconnecting two singular vertices from the rest of the graph. Thus

the sum of the degrees for all pairs together is at least $k \times \binom{|V|}{2}$, so

$$\begin{aligned} & \sum_{v \in V} \deg(v) \times (|V| - 1) \geq k \times \binom{|V|}{2} \\ \Rightarrow \quad & \sum_{v \in V} \deg(v) \times (|V| - 1) \geq \frac{k|V|(|V| - 1)}{2} \\ \Rightarrow \quad & \sum_{v \in V} \deg(v) \geq \frac{k|V|}{2}. \end{aligned}$$

This implies that the sum of all the degrees for every vertex in the graph is at least $\frac{k|V|}{2}$. After $j$ contractions, the resulting multigraph $G_j$ contains $n - j$ vertices, so $G_j$ has at least $\frac{k}{2} \cdot \frac{n-j}{2} = \frac{k(n-j)}{4}$ edges.

We can now compute the probability the resulting cut from our algorithm, (which we denote $\delta(S)$) is equal to $\delta^*(S)$. To do so, all of our contracted edges must not be in $\delta^*(S)$:

$$\begin{aligned} \mathrm{P}\left(\delta(S) = \delta^*(S)\right) &= \mathrm{P}\left(e_1, \ldots, e_{n-3} \notin \delta^*(S)\right) \\ &= \mathrm{P}\left(e_1 \notin \delta^*(S)\right) \prod_{j=1}^{n-4} \mathrm{P}\left( e_{j+1} \notin \delta^*(S) \mid e_1, \ldots, e_j \notin \delta^*(S) \right) \\ &\geq \left( \prod_{j=0}^{n-4} \left( 1 - \frac{k}{(n-j)k/4} \right) \right) \times \mathrm{P}\left(e_{n-3} \notin \delta^*(S) \mid e_1, \ldots, e_{n-4} \notin \delta^*(S) \right) \\ &\geq \frac{\cancel{n-4}}{n} \times \frac{\cancel{n-5}}{n-1} \times \frac{\cancel{n-6}}{n-2} \times \frac{\cancel{n-7}}{n-3} \times \frac{\cancel{n-8}}{\cancel{n-4}} \times \ldots \\ &\qquad \times \frac{\cancel{5}}{\cancel{9}} \times \frac{4}{\cancel{8}} \times \frac{3}{\cancel{7}} \times \frac{2}{\cancel{6}} \times \frac{1}{\cancel{5}} \times \frac{1}{k+1} \\ &= \frac{4 \cdot 3 \cdot 2 \cdot 1}{(n)(n-1)(n-2)(n-3)} \times \frac{1}{k+1} \\ &\geq \frac{1}{\binom{n}{4} \cdot n} \end{aligned}$$

□

In order to boost the probability of success, we simply run the algorithm $c\left(\binom{n}{4} \cdot n\right)$ times. Then the probability that at least one run succeeds is

$$1 - \left(1 - \frac{1}{\binom{n}{4} \cdot n}\right)^{c\left(\binom{n}{4} \cdot n\right)} \geq 1 - e^{-c} = 1 - o(1).$$

**DISCLAIMER**: You can probably get this tighter (I added an extra $n$ in the denominator); but the general idea remains the same.

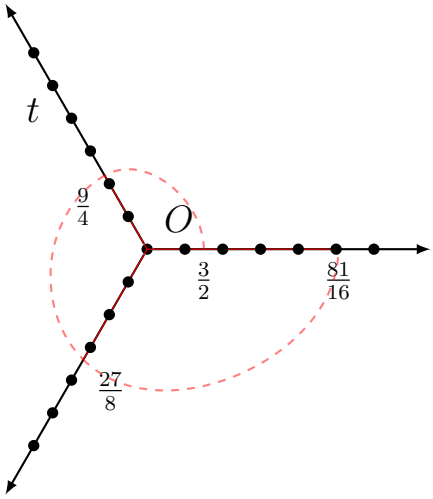
## Exercise 2

*(Navigating The Labyrinth)* The young maiden Ariadne is at the entrance of a simple labyrinth which branches in $m \geq 2$ corridors, as shown below. Unfortunately, she does not have her magic string with her! She knows the architect of the labyrinth, Daedalus, is somewhere in the labyrinth. She devises the following strategy to find him[1]:

Continue until we reach $t$: execute cycles of steps, where the function determining the number of steps to walk before the $i$th turn starting from the origin is

$$f(i) = \left(\frac{m}{m-1}\right)^i \quad \forall i \geq 1$$

What is the competitive ratio of this algorithm?




## Solution

Depending on how OPT is defined (whether OPT just knows the distance or the exact location of the $t$), you may have different competitive ratios, but it should be something like $1 + \frac{2m^m}{(m-1)^{m-1}}$.

[1]This strategy is known as *spiral search* and has been proven to be optimal. Looks like Ariadne was a Theory thread!

## Exercise 3

Define a preference list as *cyclic* when it results in a cycle, for instance, where A prefers B over C, B prefers C over A, and C prefers A over B. Assess the accuracy of the following statement: A cyclic preference list leads to the existence of at least one stable marriage.

## Solution

Not sure actually!

## Exercise 4

Consider FlushWhenFull algorithm for paging: when a cache miss occurs and the entire cache is full, evict all pages from the cache. What is the competitive ratio of FlushWhenFull?

## Solution

Obviously this is not a practical strategy. But, it turns out this has a competitive ratio of $k$. The proof is same as marking algorithm proof.

## Exercise 5

Consider adding the power of limited lookahead to an algorithm for paging. Namely, fix a constant $f \in \mathbb{N}$. Upon receiving the current page $p_i$, the algorithm also learns $p_{i+1}, p_{i+2}, \ldots, p_{i+f}$. Recall that the best achievable competitive ratio for deterministic algorithms without lookahead is $k$. Can you improve on this bound with lookahead?

## Solution

An on-line algorithm does not gain any advantage in the worst case, since any request sequence $\sigma = \sigma_1, \ldots, \sigma_p$ can be replaced by the request sequence $\sigma = (\sigma_1)^l, \ldots, (\sigma_p)^l$ in which each request is repeated $l$ times, thus, hiding the future requests from the algorithm.

## Exercise 6

*(Online Coloring)* We briefly covered coloring in our homework; now we will cover it in an online context. Our input sequence consists of the vertices $v_1, \ldots, v_n$ of an undirected graph $G = (\{v_1, \ldots, v_n\}, E)$. Together with each vertex $v_i$, we are given the list $E_i$ of all edges connecting $v_i$ to previously given vertices $v_1, \ldots, v_{i-1}$. Edges from $v_i$ to vertices $v_j$ with $j > i$ are only revealed once vertex $v_j$ is given. The number of edges and vertices in the graph is not known in advance.

When we are given $v_i$, we have to choose a color $c(v_i) \in \mathbb{N}$ for $v_i$ in such a way that no vertex adjacent to $v_i$ has color $c(v_i)$. As usual, this choice is final; we cannot change the color later on. We want to minimize the number of colors used.

We want to show that there is no deterministic $c$-competitive algorithm for this problem for any constant $c$. For any constant number $2 \leq k \in \mathbb{N}$ of colors and any deterministic online algorithm $A$, devise a strategy for an adversary that satisfies the following requirements.

- The strategy always produces a forest $T_{A,k}$.
- The online algorithm $A$ uses at least $k$ colors on $T_{A,k}$.

Offline, every forest can be colored with 2 colors; therefore, this implies that $A$'s competitive ratio is at least $k/2$.

## Solution

Given an arbitrary sequence of input items, the adversary can extend the sequence with disjoint trees $T_1, T_2, T_3, \ldots$ such that $ALG$ colors roots of the trees with $k$ different colors and the combined size of the trees is $\leq 2^k - 1$.

Let $T_1$ be the single vertex tree and assume that $T_1, T_2, \ldots, T_{n-1}$ have already been defined. Then $T_n$ is obtained as follows: We take for every $k$, $1 \leq k \leq n-1$, $|V(T_k)|$ disjoint copies of $T_k$ and in all copies we distinguish distinct vertices as roots. Tree $T_n$ is formed as the union of all these rooted copies of $T_1, \ldots, T_{n-1}$ plus a new vertex $x$ joined to every root.

Now we show how I has to play on $T_n$ against II, who can apply an arbitrary on-line coloring algorithm $A$. One can assume that there are strategies for I that forces at least $k$ distinct colors when playing on $T_k$ $(1 \leq k \leq n-1)$.

We argue that I is able to obtain $n-1$ distinct colors at the roots. Assume that I has only revealed vertices from copies of $T_1, \ldots, T_{k-1}$ and forced $k-1$ distinct colors at the corresponding roots $(1 \leq k \leq n-1)$. Now continue the game with a copy of $T_k$ until forcing a $k$th distinct color at some vertex $v$. Then I can freely identify $T_k$ with its copy in $T_n$, which has the root corresponding to $u$.

Obtaining in this way $n-1$ distinct colors at the roots, I wins by revealing vertex $x$ of $T_n$. This proves the theorem. □

## Exercise 7[2]

(*k-wise Independent Hashing*) We can extend our definition of $k$-wise independence to hashing as well. A hash family $\mathcal{H}$ is *k-wise independent* if for all $k$ distinct keys $x_1, x_2, \ldots, x_k \in \mathcal{U}$ and every set of $k$ values $v_1, v_2, \ldots, v_k \in \{0, 1, \ldots, m-1\}$, we have that

$$\mathrm{P}_{h \in \mathcal{H}} \left( h(x_1) = v_1 \wedge h(x_2) = v_2 \wedge \ldots \wedge h(x_k) = v_k \right) = \frac{1}{m^k}$$

Intuitively, this means that if you look at only up to $k$ keys, the hash family appears to hash them truly randomly.

(a) Is this hash family from $U = \{a, b\}$ to $\{0, 1\}$ (i.e., $m = 2$) one-wise independent (i.e., uniform)? how about pairwise independent?

| | $a$ | $b$ |
|---|---|---|
| $h_1$ | 0 | 0 |
| $h_2$ | 1 | 0 |

(b) Can you fill in the blanks in this hash family with values in $\{0, 1\}$ to make it pairwise independent? 3-wise independent?

[2]Taken from CMU's 15-451/651 recitation.

| | $a$ | $b$ | $c$ |
|---|---|---|---|
| $h_1$ | 0 | 0 | |
| $h_2$ | | | 1 |
| $h_3$ | 0 | | |
| $h_4$ | 1 | 1 | 0 |

## Solution

(a) i. One-wise independence implies that for each key, each hash value in the range $\{0, 1\}$ must occur with equal probability $\frac{1}{2}$. But for key $b$, we have $\mathrm{P}\left(h(b) = 0\right) = 1$ (since both $h_1(b)$ and $h_2(b)$ are 0), so $\mathcal{H}$ is not one-wise independent.

ii. Pairwise independence requires that for any two distinct keys, the probability of them mapping to any two hash values is $\frac{1}{4}$. But for keys $a, b$, we have $\mathrm{P}\left(h(a) = 0 \wedge h(b) = 1\right) = 0$, so $\mathcal{H}$ is not pairwise independent.

(b) i. For pairwise independence, each pair of distinct keys should map to any pair of values with equal probability $\frac{1}{4}$. One possible solution is as follows:

| | $a$ | $b$ | $c$ |
|---|---|---|---|
| $h_1$ | 0 | 0 | 0 |
| $h_2$ | 1 | 0 | 1 |
| $h_3$ | 0 | 1 | 1 |
| $h_4$ | 1 | 1 | 0 |

ii. For 3-wise independence, each trio of distinct keys should map to any trio of values with equal probability $\frac{1}{8}$. Achieving 3-wise independence requires 8 distinct hash functions for a range of $\{0, 1\}$. With only 4 hash functions, we cannot achieve 3-wise independence for this hash family.