Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science
Rasmus Kyng, Bernd Gärtner, David Steurer, Vera Traub

Algorithms, Probability, and Computing — Solutions KW48 — HS25

Solution 1

(a) By the definition of the determinant and by linearity of expectation, we have

$$\mathbf{E}[\det(B)] = \sum_{\pi\in\mathcal{S}_n} \operatorname{sign}(\pi)\, \mathbf{E}\Big[b_{1,\pi(1)}b_{2,\pi(2)}\dots b_{n,\pi(n)}\Big].$$

Now let $Z \subseteq \mathcal{S}_n$ be defined as

$$Z := \{\pi \in \mathcal{S}_n | a_{1,\pi(1)}a_{2,\pi(2)}\dots a_{n,\pi(n)} = 1\},$$

that is the set of transversals that do not contain a zero element of A. We then have that

$$\mathbf{E}[\det(B)] = \sum_{\pi\in Z} \operatorname{sign}(\pi)\, \mathbf{E}\Big[\epsilon_{1,\pi(1)}\epsilon_{2,\pi(2)}\dots \epsilon_{n,\pi(n)}\Big],$$

and by independence of the $\epsilon_{i,j}$,

$$\mathbf{E}[\det(B)] = \sum_{\pi\in Z} \operatorname{sign}(\pi)\, \mathbf{E}\Big[\epsilon_{1,\pi(1)}\Big]\,\mathbf{E}\Big[\epsilon_{2,\pi(2)}\Big]\dots \mathbf{E}\Big[\epsilon_{n,\pi(n)}\Big] = 0,$$

as each expectation is zero.

(b) This calculation is more involved. We first note that by definition (and reusing the set Z from (a)),

$$\mathbf{E}\Big[(\det(B))^2\Big] = \mathbf{E}\left[\left(\sum_{\pi\in Z}\operatorname{sign}(\pi)\epsilon_{1,\pi(1)}\epsilon_{2,\pi(2)}\dots\epsilon_{n,\pi(n)}\right)^2\right].$$

Expanding the multiplication and applying linearity of expectation yields

$$\mathbf{E}\Big[(\det(B))^2\Big] = \sum_{\pi_1,\pi_2\in Z}\operatorname{sign}(\pi_1)\cdot\operatorname{sign}(\pi_2)\cdot\,\mathbf{E}\Big[\epsilon_{1,\pi_1(1)}\epsilon_{1,\pi_2(1)}\epsilon_{2,\pi_1(2)}\epsilon_{2,\pi_2(1)}\dots\epsilon_{n,\pi_1(n)}\epsilon_{n,\pi_2(n)}\Big].$$

Now we start disentangling dependencies. First of all, since the $\epsilon_{i,j}$ are independent from one another, we can separate the expectation as

$$\mathbf{E}\Big[(\det(B))^2\Big] = \sum_{\pi_1,\pi_2\in Z}\operatorname{sign}(\pi_1)\cdot\operatorname{sign}(\pi_2)\cdot\,\mathbf{E}\Big[\epsilon_{1,\pi_1(1)}\epsilon_{1,\pi_2(1)}\Big]\,\mathbf{E}\Big[\epsilon_{2,\pi_1(2)}\epsilon_{2,\pi_2(1)}\Big]\dots\mathbf{E}\Big[\epsilon_{n,\pi_1(n)}\epsilon_{n,\pi_2(n)}\Big].$$

Now we observe that

$$\mathbf{E}\left[\epsilon_{i,j}\epsilon_{i,k}\right] = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise.} \end{cases}$$

For that reason, all the summands with $\pi_1 \neq \pi_2$ have at least one zero factor in the product and thus vanish. Remaining are the summands where the permutations are equal and thus

$$\mathbf{E}\left[(\det(B))^2\right] = \sum_{\pi \in Z} \operatorname{sign}^2(\pi) \cdot \mathbf{E}\left[\epsilon_{1,\pi(1)}^2\right] \mathbf{E}\left[\epsilon_{2,\pi(2)}^2\right] \ldots \mathbf{E}\left[\epsilon_{n,\pi(n)}^2\right] = |Z|.$$

On the other hand, obviously

$$\operatorname{per}(A) = \sum_{\pi \in Z} 1 = |Z|,$$

which establishes the claim.

Solution 2

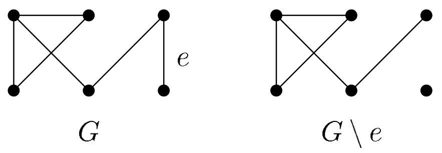
We are given an algorithm A for testing the existence of a perfect matching in a given graph, with running time at most $T(n)$ for any n-vertex graph.

(a) We want to find a perfect matching of a graph G by using repeated calls to algorithm A (supposed that G has a perfect matching).

First we call $A(G)$. If it says "No", G has no perfect matching. Done. If the algorithm says "Yes" G has a perfect matching. We have to find one.

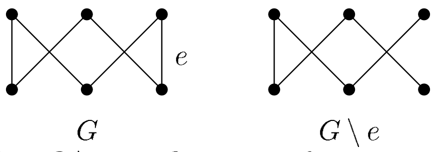
Choose an arbitrary edge e of the graph G. Now we are going to check whether e is part of *every* perfect matching of G. To do that, consider deleting e from G. Denote by $G \setminus e$ the result of the deletion. Then we call $A(G \setminus e)$. We have two cases.

Case 1. If the algorithm says "No", then e is a part of every perfect matching of G (since G contains a perfect matching but $G \setminus e$ does not).



In this case we keep e as an edge of the perfect matching that we will output later, and continue with the remaining graph, i.e., the graph obtained by removing the vertices incident to e (because they are already matched by e).

Case 2. In case the algorithm says “Yes”, $G \setminus e$ contains a perfect matching, which is also a perfect matching of G.



Therefore we continue with $G \setminus e$ to find a perfect matching in $G \setminus e$.

This is the idea of our procedure, as given by the following Algorithm 1 in pseudocode.

```
Algorithm 1:  Finding a Perfect Matching in a Graph
    Input:   a graph $G = (V, E)$
    Output:  a perfect matching $M$ of $G$ if exists and ‘No’ if not
    IF $A(G)$ = ‘No’ THEN
       RETURN ‘No’
    ELSE
       $M \leftarrow \emptyset$
       WHILE $M$ is not a perfect matching of $G$ DO
          $e \leftarrow$ an arbitrary edge in $E$
          IF $A(G \setminus e)$ = ‘No’ THEN
             $M \leftarrow M \cup \{e\}$
             $G \leftarrow$ the graph obtained by removing the vertices incident to $e$
          ELSE
             $G \leftarrow G \setminus e$
          END
       END
       RETURN $M$
    END
```

The correctness of the algorithm follows from the discussion above. What is the running time? One call to $A(G)$ takes $T(n)$ time. Then, we will potentially enter the while-loop. In each iteration of the loop at least one edge is removed from the graph and the number of vertices in the graph is always at most n. The time we need for the deletion of an edge or a vertex depends on a data structure used in the test A, so let us denote it by $t(n)$, when we are dealing with a graph with n vertices. Then the worst-case total running time is at most $T(n) + O(m \cdot (t(n) + T(n))) = O(mT(n))$. Here, $m := |E|$ as usual and we safely assume $t(n) < T(n)$.

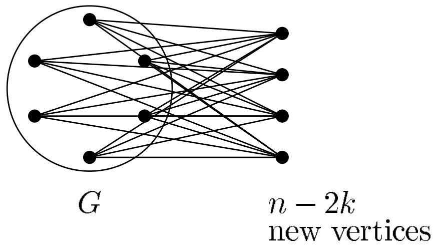
If we use the algorithm from the lecture as A, we get a running time $O(n^{4.376})$.

(b) How can the above algorithm be used for finding a maximum matching in a given graph?

Let G be a graph with n vertices. The basic step is to decide whether G has a matching of size k (i.e., consisting of k edges). With this subroutine, we search for the maximum k by performing the binary search on $\{0..\lfloor n/2 \rfloor\}$ (note that if G contains a matching of size k then it also contains a matching of smaller size).

Therefore, the overall time to decide the size of the maximum matching will be $O(\log n)$ multiplied by the time needed to decide whether G contains a matching of a given size.

Let us fix $k \in \{0..\lfloor n/2 \rfloor\}$. To decide whether G has a matching of size k, we construct an auxiliary graph G^* from G as follows. The vertex set of G^* is the vertex set of G plus additional $n - 2k$ vertices. The edge set of G^* is the edge set of G plus the following edges: we connect every vertex of G to each of the new vertices by an edge. This is our construction of G^*:



Lemma 1. G *has a matching of size* k *if and only if* G^* *has a perfect matching.*

Proof. Assume that G has a matching of size k. Take such a matching. There are $n - 2k$ vertices in G which are not incident to any edge of the matching. Then, in G^* these vertices can be matched with the additional vertices, which gives a perfect matching of G^*. Conversely, if we have a perfect matching of G^*, by removing the additional $n - 2k$ vertices and the edges incident to them we obtain a matching in G of size k.

In this way, deciding if G has a matching of size k reduces to deciding whether G^* has a perfect matching. We observe that G^* has $2n - 2k$ vertices and $m + n(n-2k)$ edges, which are at most $2n$ and $m + n^2 = O(n^2)$ respectively. Therefore, running the above binary search to find the appropriate maximum k and applying the result of (a) to finally find a matching of size k we can find a maximum matching of G in $O(\log(n)T(2n) + n^2T(2n)) = O(n^2T(2n))$ time. $\square$

Solution 3

(a) Let $k \in \mathbb{N}$ be such that 2^k is the smallest power of two that is at least N, i.e., $2^{k-1} < N \leq 2^k$. Take k random bits from the stream and interpret the sequence of k random bits as an integer i, written in its binary representation. Because the stream consisted of random bits, the number $i + 1$ is uniformly distributed in the set $\{1, \dots, 2^k\}$. If $i + 1 \leq N$, then $i + 1$ is uniformly distributed in $\{1, \dots, N\}$ because for every $j \in \{1, \dots, N\}$ it holds that

$$\Pr[j = i + 1 \mid i + 1 \leq N] = \frac{\Pr[j = i + 1 \text{ and } i + 1 \leq N]}{\Pr[i + 1 \leq N]} = \frac{1/2^k}{N/2^k} = \frac{1}{N}.$$

To sample the required number we would repeat the above process, always sampling a new integer $i \in \{1, \dots, 2^k\}$ by using k new random bits from the stream until

$i+1 \in \{1, \dots, N\}$. The success probability of one such sampling is $p := \frac{N}{2^k} > \frac{2^{k-1}}{2^k} = \frac{1}{2}$ and different repetitions are independent of each other. The number of repetitions needed until succeeding is geometrically distributed with parameter p. Therefore in expectation after $\frac{1}{p} < 2$ repetitions we will succeed. We conclude that the expected number of bits used is at most $2 \cdot k = O(\log N)$.

(b) We will first describe our algorithm SampleMatching for the problem and then prove its correctness and show that it satisfies the runtime requirement and that it uses only the required number of random bits. Given a graph $G = (V, E)$ and an edge $e \in E$ we let G_e denote the graph attained from G by removing both endpoints of e and all their adjacent edges from G. For a vertex $v \in V$ we denote by $\delta(v) \subseteq E$ the set of all edges adjacent to v. We let Oracle(G) denote the oracle function that takes a graph and returns the number of perfect matchings in G. Our algorithm SampleMatching for the problem is defined below.

```
Input: A nonempty simple graph $G = (V, E)$.
Output: A uniformly random perfect matching $M \subseteq E$ in $G$ or $\emptyset$ if there is no perfect matching.
SampleMatching(G):
    1. Check with the oracle that G has at least one perfect matching. If not, return $\emptyset$.
    2. If $|V| = 2$, return the single edge in G.
    3. Let $v \in V$ be an arbitrary vertex and fix an ordering of $\delta(v) = \{e_1, \dots, e_s\}$.
    4. For every $i = 1, \dots, s$ let $N_i :=$ Oracle$(G_{e_i})$ and let $N := \sum_{i=1}^{s} N_i$.
    5. Using (a) choose a uniformly random integer $k \in \{1, \dots, N\}$.
    6. Let j be the least index so that $\sum_{i=1}^{j} N_i \geq k$.
    7. Return $\{e_j\} \cup$ SampleMatching$(G_{e_j})$.
```

Correctness proof. We show that SampleMatching really outputs a uniformly random perfect matching given that there is at least one such matching. If G has an odd number of vertices or no perfect matching is recognized in step 1 and is correctly handled so we need to prove correctness only for graphs with at least one perfect matching.

We proceed by induction on n. The base case $n = 2$ is handled correctly in step 2 since there is a unique perfect matching, the single edge. Assume now that the algorithm is correct for all graphs with at most $n-2$ vertices, n even, and consider a graph G with n vertices. Fix also some perfect matching M in G. We observe first that the number N computed in step 4 is the number of perfect matchings in G. This is because every perfect matching of G contains exactly one of the edges $e \in \delta(v)$ and because every perfect matching M' that contains some edge $e \in \delta(v)$ has the property that $M' \setminus \{e\}$ is a perfect matching in G_e.

Let ℓ be such that M contains the edge $e_\ell \in \delta(v)$. For the algorithm to output M it has to be that $\ell = j$ and that the recursive call of step 7 returns the matching $M \setminus \{e_\ell\}$ when called on the graph G_{e_ℓ}. Notice that $\Pr[j = \ell] = \frac{N_\ell}{N}$ since there are N_ℓ values $k \in \{1, \dots, N\}$ for which ℓ is the least index satisfying the condition in step 6. By induction we have

$$\Pr[\textsc{SampleMatching}(G_{e_j}) = M \setminus \{e_\ell\} \mid j = \ell] = \frac{1}{N_\ell}.$$

Therefore the probability that the algorithm returns M is $\frac{N_\ell}{N} \cdot \frac{1}{N_\ell} = \frac{1}{N}$ which is uniform across all perfect matchings. This concludes the correctness proof.

Runtime analysis. In steps 1-3 we do one call to the oracle that uses time $T(n)$ and additionally we spend only $\text{poly}(n)$ time. In step 4 we do at most $n-1$ calls to the oracle, each taking time $T(n)$. Because of part (a) step 5 takes time $O(\log N)$ in expectation which is $\text{poly}(n)$ because N is certainly at most $n! \leq n^n$. Step 6 takes also $\text{poly}(n)$ time. Notice that when we recurse in step 7 we reduce the number of vertices by 2 so the depth of the recursion is $O(n)$. Therefore the total number of oracle calls is at most $(n-1) \cdot O(n) = O(n^2)$ and the total expected runtime is also bounded by $O(T(n)\text{poly}(n))$ as required.

Random bits. We already argued that $N \leq n^n$ which implies that the number of random bits we use in step 5 is in expectation $O(\log N) = O(n \log n)$. We do such sampling $O(n)$ times across the recursive calls so the total number of random bits we use is $O(n^2 \log n)$.

(c) The key ingredient to solving this problem is to realize that in a planar graph there always exists a vertex whose degree is at most 5. This is because the number of edges in a planar graph with n vertices is at most $3n-6$. Therefore the average degree is at most $\frac{2(3n-6)}{n} < 6$ which implies the claim. We change the algorithm from part (b) by choosing the vertex v in step 3 as a vertex whose degree is at most 5. The correctness of the algorithm stays unchanged.

Runtime analysis Since in every recursive step there are at most $1+5 = 6$ oracle calls, the total number of oracle calls across the algorithm execution is $O(n)$. Because there is always a vertex of degree at most 5, the number of perfect matchings in a planar graph is at most $5^{n/2}$. This means that N in step 4 of the algorithm is upper bounded by $5^{n/2}$ and the expected runtime of steps 4-6 is therefore at most $T(n) + O(\log N) = 5T(n) + O(n)$. Note also that steps 1-3 also take only linear time plus time $T(n)$ since planar graphs have only linearly many edges and in particular the vertex of degree at most 5 can be found in linear time.

If we let $t(n)$ denote the expected runtime of the modified algorithm SampleMatching we have deduced that $t(n)$ satisfies the recurrence

$$t(n) \leq 6T(n) + O(n) + t(n-2).$$

More concretely let c be a constant such that for $n \geq C$, for some large enough constant C, it holds that

$$t(n) \leq 6T(n) + cn + t(n-2).$$

Let us prove by induction that $t(n) \leq dnT(n)$ for some constant d when $n \geq C$. We don't need to prove the cases $n < C$ since these can be solved in constant time by enumerating all matchings (because C is constant). Also the base case $n = C$ be made to hold by choosing d large enough. For the inductive step we use the recurrence formula together with the induction assumption to conclude that

$$\begin{aligned} t(n) &\leq 6T(n) + cn + d(n-2)T(n-2) \\ &\leq 6T(n) + cn + d(n-2)T(n) \\ &= 6T(n) + cn - 2dT(n) + dnT(n) \\ &\leq dnT(n). \end{aligned}$$

Above $T(n-2) \leq T(n)$ since $T(n) \in \Omega(n)$. The last step also follows by choosing d large enough because $T(n) \in \Omega(n)$ then implies that $5T(n) + cn - 2dT(n) < 0$.

Random bits. In one recursive step we use by (a) and our previous remarks at most $O(\log 5^{n/2}) = O(n)$ random bits in expectation. Since there are at most n recursive steps, the total number of random bits is at most $O(n)$.