Course: Randomized Algorithms by Dr. Zarei

Homework: HW2

Name: Mohammad Mohammadi

Student ID: 402208592

Exercise 3.2.1. [DARA_Hor]

Let U be a finite set, |U| = r · m for a positive integer r. Estimate the number of functions h : U → T that satisfy the property (3.1).

Answer

Let's assume a common scenario in hashing or load balancing where each function h should map elements of U to elements of T in a way that each output in T is hit an approximately equal number of times (a balanced mapping). Assume |T|=m. This assumption is common in exercises related to hash functions where the goal is uniform distribution across the target set.

Estimation:

1. Total Number of Functions: Without any restriction, the total number of functions from U to T is $m^{|U|} = m^{r \cdot m}$, since each element in U can be mapped to any of the m elements in T.
2. Balanced Functions:
    - We want each element in T to be hit r times (since |U|=r·m).
    - The number of ways to choose r elements out of r·m for one particular element in T (while ensuring the same count for others) follows a combinatorial approach.
3. Using Combinatorics: To compute the number of ways to assign each t∈T exactly r pre-images in U, we:
    - Choose r elements from U to map to the first element in T,
    - Then choose r elements from the remaining U for the second element in T,
    - Continue this until all elements in T are assigned r elements from U.

This can be calculated as:

$$\binom{r \cdot m}{r}\binom{r \cdot (m-1)}{r} \cdots \binom{r}{r}$$

Simplifying, this is equivalent to:

$$\frac{(r \cdot m)!}{(r!)^m}$$

Conclusion Assuming h maps each element to each target with equal frequency, the estimated number of such functions h is $\frac{(r \cdot m)!}{(r!)^m}$. Based on examples I've seen online, this is a typical analysis in scenarios requiring uniform distribution.

Exercise 3.2.4. [DARA_Hor]

Let S be a randomly chosen group of persons. How large has S to be in order to assure:

Prob(two persons from S were born on the same day) ≥ 1/2 ?

To answer this question, consider the indicator variable X defined by

X(S) = {1: if there are two persons in S with the same birthday| 0: else}

and assume that the birthdays are uniformly distributed over the year.

Estimate E[X] as a function of |S|.

Answer

First, let's calculate P(X(S)=1), the probability that at least two persons in S have the same birthday.

1. Complement Rule: Instead of directly calculating P(X(S)=1), it's easier to calculate its complement, P(X(S)=0), which is the probability that no two persons in S share the same birthday.
2. Calculation of P(X(S)=0):
    - The first person can have any birthday, which happens with a probability of 1 (365/365).
    - The second person must have a different birthday than the first, which can happen with a probability of (364/365).
    - The third person must have a different birthday than the first two, which can happen with a probability of (363/365), and so on.
    - Therefore, for |S|=n, the probability that no one shares a birthday is:

$$P(X(S) = 0) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \ldots \times \frac{365 - n + 1}{365}$$

Using the product notation:

$$P(X(S) = 0) = \prod_{k=0}^{n-1} \left(\frac{365 - k}{365} \right)$$

Since X(S) is an indicator variable, E[X]=P(X(S)=1), which is:

$$E[X] = P(X(S) = 1) = 1 - P(X(S) = 0)$$

To find the smallest n for which $E[X] \geq 1/2$, we calculate when $P(X(S) = 0) \leq 1/2$.

From empirical calculations and known results, this threshold is reached at n=23, as this is the smallest number for which the probability of no shared birthday is less than or equal to 0.5.

Hence, to assure that the probability of at least two persons from a randomly chosen group S having the same birthday is at least 1/2, the group must have at least 23 persons. This results in E[X] being at least 0.5 for |S|=23, fulfilling the condition given in the exercise.

Refs:

1. Wikipedia page for the Birthday Problem: https://en.wikipedia.org/wiki/Birthday_problem
2. BetterExplained: Understanding the Birthday Paradox https://betterexplained.com/articles/understanding-the-birthday-paradox/

Exercise 3.3.10. [DARA_Hor]

Let us consider the set

M = {h : U → T | h fulfills (3.1)}.

Is M a universal set of hash functions?

Answer

Definition of a Universal Set of Hash Functions:

A set of hash functions is considered universal if for every pair of distinct keys k_1 and k_2 from the universe of keys U, the probability of a collision (i.e., $h(k_1) = h(k_2)$) under a randomly chosen hash function h from the set is low, typically 1/|T| where |T| is the size of the target set.

Adversary Foiling:

The essence of "foiling the adversary" is about designing a set of hash functions that remain effective even when an adversary knows the functions and attempts to provide inputs that would typically lead to worst-case performances (like clustering of hash values, leading to inefficiencies).

Assessment:

If M indeed meets these criteria — particularly if it ensures a low probability of collision across distinct keys and maintains efficiency across a range of adversarial inputs — then M could potentially be considered a universal set. However, true universality in the strictest sense would depend on the precise mathematical properties M guarantees, which should include a uniform distribution of hash values and independence between the choices of hash values for different keys.

In Conclusion:

While M might exhibit characteristics whit similar nature or characteristic to those of a universal set of hash functions, whether it qualifies as universal depends on the strict adherence to the mathematical definition involving low collision probabilities and performance under adversarial conditions. We can conjecture based on typical goals in such design scenarios but can't conclusively declare M universal without detailed properties.

Exercise 4.2.1. [DARA_Hor]

Apply the amplification method in order to enable a randomized test of x ∈ U for larger sets U. Estimate the maximal possible cardinality of U for which the achieved error probability still approaches 0 with growing n when the communication complexity is

(i) in O(log n · log log n),

(ii) in O ((log n)^d) for any constant d ∈ IN,

(iii) polylogarithmic.

Answer

(i) Communication Complexity in $O(logn \cdot loglogn)$

For a communication complexity of $O(logn \cdot loglogn)$, the size of U is limited by how much information can be transmitted under this bound. To estimate the maximum cardinality of U, consider that the logarithmic terms suggest that the size of U can grow exponentially with n as long as the exponent itself is restricted to $logn \cdot loglogn$. This means:

$$|U| \approx 2^{\log n \cdot \log\log n} = n^{\log\log n}$$

This estimation allows the error probability to approach zero with repeated testing n times, assuming that errors decrease exponentially with the number of tests.

(ii) Communication Complexity in $O((logn)^d)$

When the complexity is bounded by $O((logn)^d)$, for some constant d, the exponential growth in U's size can be more significant compared to the previous case. This time:

$$|U| \approx 2^{(\log n)^d}$$

Here, n can be extremely large, depending on d. The function $(logn)^d$grows much slower than n but faster than $logn \cdot loglogn$, allowing for a larger set U while still ensuring that the error probability approaches zero as n increases.

(iii) Polylogarithmic Communication Complexity

A polylogarithmic complexity, often noted as $poly(logn)$, implies a complexity that can be expressed as a polynomial function of $logn$. This gives:

$$|U| \approx 2^{\text{poly}(\log n)}$$

This notation still encapsulates a very large growth potential for U but within a manageable computational framework. The exact nature of the polynomial affects the specific growth rate, but in general, this would allow for a very large U compared to typical logarithmic or linear growth rates.

Exercise 4.2.5. [DARA_Hor]

Transform the protocol PDisj to a Las Vegas protocol. How large is its expected communication complexity?

Answer

To transform a Monte Carlo protocol into a Las Vegas protocol, one key requirement is to ensure that the protocol always produces a correct result with the possibility of a variable runtime, rather than having a fixed runtime with a chance of error as in Monte Carlo protocols.

1. Run the Monte Carlo PDisj Repeatedly: We start by running the original Monte Carlo version of PDisj. If this version can err by falsely claiming that the sets are not disjoint (or any type of error relevant to the original PDisj), then we need to handle this in the Las Vegas version.
2. Verification Step: After each run of the Monte Carlo PDisj, we introduce a verification step. If the Monte Carlo PDisj concludes that the sets are disjoint, we verify this result by checking a subset or utilizing a deterministic approach that guarantees correctness for that specific case.
3. Repeat if Necessary: If the verification step fails (i.e., if the Monte Carlo result is found to be incorrect), the algorithm repeats both the Monte Carlo test and the verification until the correct result is confirmed.
4. Termination: The Las Vegas protocol will only terminate when a correct result is achieved and verified, ensuring that it always produces the correct output, albeit with a potentially variable runtime.

Expected Communication Complexity

The expected communication complexity of this Las Vegas version of PDisj will depend heavily on:

- The error probability of the original Monte Carlo PDisj.
- The communication cost of each Monte Carlo run.
- The communication cost of the verification step.

Typically, the Las Vegas version has an expected communication complexity calculated as follows:

$$\text{Expected Communication Complexity} = \sum_{k=1}^{\infty}(\text{Communication per run} \times P(\text{Error})^{k-1})$$

Where $P(Error)$ is the probability that the Monte Carlo algorithm errs. If $P(Error)$ is relatively low and the communication cost per run is moderate, the series converges quickly, and the expected communication complexity remains reasonable.

For protocols like PDisj, where the Monte Carlo version might have an error probability $P(Error)$ that could be adjusted (e.g., by adjusting the parameters of the Monte Carlo test), the expected complexity can be quite efficient.

This transformation leverages the ability to repeat the probabilistic test multiple times to drive the error probability down exponentially with each additional run, typical of Las Vegas algorithms where runtime (or in this case, communication) can increase to ensure correctness.

Exercise 4.5.17. [DARA_Hor]

Consider the communication protocol R presented in Section 1.2 for the comparison of two binary strings (of the contents of two databases). Let p be a prime and let $a = a_1 a_2 \ldots a_n, a_i \in 0, 1 for i = 1, \ldots, n$, be a binary string. Consider the polynomial P_a of a singe variable x defined over Z_p as follows.

$$P_a(x) = \sum_{i=1}^{n} a_i x^{i-1}$$

Apply the idea of algorithm AQP on order to design a 1MC protocol for the comparison of two n bit strings $a = a_1 a_2 \ldots a_n and b = b_1 b_2 \ldots b_n$. What effect does the choice of p have on the error probability and on the communication complexity? Compare the results with the complexity and the success probability of the protocol R from Section 1.2.

Answer

The idea is to evaluate both polynomials at a randomly chosen point and compare the results.

1. **Polynomial Representation:** Define polynomials for both strings over $\mathbb{Z}_p$ (where p is a prime number) as:

$$P_a(x) = \sum_{i=1}^{n} a_i x^{i-1}, \quad P_b(x) = \sum_{i=1}^{n} b_i x^{i-1}$$

Here, x is a variable and $a_i, b_i \in \{0, 1\}$ are the bits of the binary strings.

2. **Random Evaluation:** Randomly choose a value x from $\mathbb{Z}_p$ and compute $P_a(x)$ and $P_b(x)$.
3. **Comparison:** If $P_a(x) = P_b(x)$, conclude $a = b$; otherwise, conclude $a \neq b$.

Impact of the Choice of p on the Protocol

- Error Probability: If $a \neq b$ $P_a(x) - P_b(x)$ is a non-zero polynomial of degree at most n−1. The number of roots of a non-zero polynomial over a field can be at most the degree of the polynomial. Hence, the probability that a random x leads to a false positive (where $P_a(x) = P_b(x)$ but $a \neq b$) is at most $(n-1)/p$. Therefore, a larger p decreases the error probability.
- Communication Complexity: Communicating the value of $P_a(x)$ or $P_b(x)$ requires $log_2(p)$ bits since the values are elements of $\mathbb{Z}_p$. Thus, larger values of p increase the communication cost.

Comparison with Protocol R

Protocol R uses a set of randomly chosen prime moduli and communicates results of modular arithmetic. It achieves high efficiency with relatively low communication complexity and a very low error probability, owing to the use of multiple primes and combining results from independent trials.

- Error Probability: The error probability in Protocol R can be extremely low, as calculated using the prime number theorem and properties of bad primes. For large n, Protocol R ensures an error probability that falls well below typical physical error rates in computational systems.
- Communication Complexity: Protocol R communicates only a few integers (the results of modular reductions), making it very efficient in terms of the number of bits sent, especially when compared to protocols requiring larger bit communications.

In comparison, the 1MC protocol using polynomial evaluation may have a simpler implementation but could potentially suffer from higher communication costs and error rates without careful choice of the prime p and possibly more sophisticated error reduction techniques such as repeating the protocol with different values of x and taking majority votes or using error-correcting methods.

Exercise 5.2.2. [DARA_Hor]

Modify the algorithm CONTRACTION in the following way. Instead of choosing an edge at random, choose two vertices x and y randomly and join them into one vertex. Construct multigraphs of n vertices, for which the probability that the modified algorithm finds a minimal cut is exponentially small in n.

```
Algorithm CONTRACTION:

Input: A connected multigraph G = (V, E, c)
Step 1: Set label (v) := {v} for every vertex v ∈ V.
Step 2:
    while G has more than two vertices do
        begin
            choose an edge e = {x, y} ∈ E(G);
            G := Contract(G, e);
            Set label(z) := label(x) ∪ label(y)
            for the new vertex z = ver(x, y);
        end
Step3:
    if G = ({u, v}, E(G)) for a multiset E(G) then
        output “(label (u) , label (v))” and “cost = |E(G)|”
```

Answer

1. **Initialization:** As in the original algorithm, start with each vertex v in the graph G=(V,E,c) labeled individually.
2. **Contraction Step:**
    - Instead of choosing an edge e={x,y} randomly from E, choose two vertices x and y randomly from V.
    - If there exists an edge e between x and y, proceed to merge x and y into a new vertex z. If no edge exists, either skip the step or choose a different pair until an edge is found (the how to of the choice here can affect the algorithm's performance and determine how efficientlyy it can run).
    - Update the graph by replacing x and y with z, combining the incident edges while preserving multiedges and self-loops.
    - Update labels to reflect the merger: label(z) := label(x) ∪ label(y)
3. **Termination:**
    - Continue the contraction until only two vertices remain.
    - Output the labels of the two remaining vertices and the number of edges (multiedges counted separately) between them as the cut-cost.

Algorithm Analysis

- **Random Selection of Vertices:** The key difference is that vertices are selected randomly without regard to whether they are connected by an edge. This can increase the number of steps needed to reduce the graph to two vertices, as non-adjacent vertices might be selected.

- **Probability of Finding Minimal Cut:** The probability that the modified algorithm finds a minimal cut is lower than in the original algorithm. In the original, each edge contraction step directly contributes to potentially separating the graph into two parts, maintaining connectivity until the very end. By randomly choosing vertices, the chance of separating a minimal cut without prematurely isolating a vertex or merging across a minimal cut decreases.
- **Exponential Decrease in Success Probability:** For graphs where minimal cuts consist of few edges between large clusters of vertices, the modified approach has an exponentially smaller probability of finding these cuts. It's much more likely to join vertices across different cuts before isolating these crucial edges, especially in dense graphs or graphs with complex connectivity.

Constructing Graphs for Small Success Probability To construct a graph where the modified algorithm performs poorly (i.e., the probability of finding a minimal cut is exponentially small), consider a graph structure where:

- The graph has a high degree of connectivity (dense graph).
- Minimal cuts are not evident or are distributed such that random vertex selection frequently bypasses these cuts.
- Graphs with symmetrical structures or multiple equivalent minimal cuts spread across the structure, causing random selections to often merge vertices from different cuts.

IN Conclusion:

This modification generally reduces the effectiveness of the CONTRACTION algorithm in finding minimal cuts due to the lack of targeted edge selection, potentially increasing the number of steps required and reducing the likelihood of isolating minimal cuts before the graph is reduced to two vertices. For applications where finding a minimum cut is crucial, this modified algorithm would be less preferred by default.

Exercise 5.2.8. [DARA_Hor] [OPTIONAL - EXTRA POINT]

Analyze the success probability and the time complexity of the versions of the algorithm REPTREE, for which one takes the following size reduction between two splits of the computation:

(i) from l to $\left\lfloor \frac{l}{2} \right\rfloor$

(ii) from l to $\sqrt{l}$

(iii) from l to $\frac{l}{\log_2 l}$

(iv) from l to $l - \sqrt{l}$

```
Algorithm REPTREE(G)

Input: A multigraph G=(V,E,c), |V|=n, n∈N, n≥3.

Procedure:

if n ≤ 6 then
    compute a minimal cut deterministically
else
    begin
    h := \left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil;
    Perform two independent runs of CONTRACTION in order to get two multigraphs G/F1 and (
    REPTREE(G/F1);
    REPTREE(G/F2)
    end
output the smaller of the two cuts computed by REPTREE(G/F1) and REPTREE(G/F2)
```

Theorem 5.2.5: The algorithm REPTREE works in time $O(n^2 \cdot logn)$ and finds a minimal cut with a probability of at least $\frac{1}{\Omega(log_2 n)}$.

Answer

Part (i)

Time Complexity:

- This division leads to a typical binary recursion where the problem size halves at each level. The depth of recursion is therefore O(logl).
- If each level of the recursion takes time proportional to the current size (say O(l) per level), the total time complexity using the Master Theorem or a simple recursive sum is O(l).

Success Probability:

- Assuming that each recursive call must succeed for the overall success (which is typical in algorithms like Karger's min-cut where global properties depend on local correctness), and if each has a constant independent success probability p, the overall success probability is $p^{logl} = l^{logp}$. For p small, this diminishes with increasing l.

Part (ii)

Time Complexity:

- Here, the recursion depth increases since $\sqrt{l}$ shrinks the problem size slower than halving. The recursion depth is approximately O(loglogl).
- The time complexity, assuming O(l) work per level, results in a total complexity potentially greater than O(l) due to multiple layers operating on still substantial problem sizes.

Success Probability:

- As in the binary case, but now the probability degrades slower, potentially resulting in higher reliability if p is sufficiently large.

Part (iii)

Time Complexity:

- The problem size is reduced slower than in the previous cases, implying a deeper recursion, potentially around O(logl).
- If the work at each level is O(l), the total complexity can be analyzed as similar or slightly worse than O(llogl).

Success Probability:

- Better than in the case of a halving reduction due to a lesser reduction in problem size each step, leading to more opportunities for error correction or success consolidation.

Part (iv)

Time Complexity:

- This reduction is very slow, leading to a very high number of recursive calls, potentially $O(\sqrt{l})$.
- Given O(l) work at each level, this might result in a high overall complexity.

Success Probability:

- Due to many steps and a small reduction per step, this could be high if the failure rate per step is not too large.

In summry

- Fastest convergence in size reduction: (i) and (ii) are preferred.
- Balance between depth and error resilience: (iii) might offer a trade-off.
- Highest error resilience but possibly inefficient: (iv).

Exercise 5.3.11. [DARA_Hor] [OPTIONAL - EXTRA POINT]

Extend the algorithm SCHÖNING for 4SAT. Observe that the lower bound on the probability of moving toward α^* in a local step decreases to 1/4 in this case. How many repetitions of random sampling followed by a local search are necessary to get a constant success probability?

Answer

When extending this algorithm to 4SAT, the probability of flipping a variable that makes the current assignment closer to a satisfying assignment α^* decreases to 1/4. This change impacts the expected number of steps required before either finding a satisfying assignment or restarting.

For 4SAT, the probability p of succeeding in a single run of local search is impacted by the lower probability 1/4 of making the correct variable flip at each step. The expected length L of the local search where a flip successfully moves closer to α^* is given by the drift analysis, often simplified to around 4n steps, given the new probability 1/4 and assuming the worst case of n variables being incorrect.

Schöning's analysis provides that for 3SAT, the success probability p of a single run is approximately $\left(\frac{1}{3}\right)^n \cdot e^n$. Extending to 4SAT with similar analysis but using 1/4 gives us an approximate success probability per trial of:

$$p \approx \left(\frac{1}{4}\right)^n \cdot e^n$$

To ensure a constant success probability P, such as P=0.99, we use the formula for the probability of failure after T independent trials:

$$1 - P = (1 - p)^T$$

$$T = \frac{\log(1 - P)}{\log(1 - p)}$$

Inserting the approximate p for large n:

$$T \approx \frac{\log(0.01)}{\log(1 - (\frac{1}{4^n} \cdot e^n))} \approx$$

$$T \approx \frac{-\log(0.01)}{\frac{1}{4^n} \cdot e^n}$$

This calculation shows that T grows very fast with n, indicating a need for a large number of trials to achieve a high probability of finding a satisfying assignment, reflecting the inherent difficulty of 4SAT.

Exercise 5.4.16. [DARA_Hor] [OPTIONAL - EXTRA POINT]

Modify the algorithm NQUAD in such a way that it always halts with a quadratic nonresidue (i.e., the answer "?" never appears). Analyze the expected running time of your modified NQUAD and prove that the probability of executing an infinite computation is 0.

Answer

To modify NQUAD to always halt with a quadratic nonresidue and prevent indefinite computation, consider the following:

- Instead of one random selection, repeatedly select random elements until a nonresidue is found.
- Limit the number of trials to ensure that the algorithm does not run indefinitely, with theoretical justification that a nonresidue will be found within these limits.

Mathematical Justification:

- Given that approximately half of the elements of F_p are nonresidues, the probability p of selecting a quadratic nonresidue in one trial is $\frac{p-1}{2p} \approx \frac{1}{2}$.
- The probability of not finding a nonresidue in k independent trials is $(1 - \frac{1}{2})^k = \frac{1}{2^k}$.
- To ensure that the algorithm almost surely finds a nonresidue, k can be chosen such that $\frac{1}{2^k}$ is negligibly small, ensuring the probability of running indefinitely (i.e., never finding a nonresidue) approaches zero.

Since each trial is independent and has a success probability of approximately 1/2, the expected number of trials until success is E(T)=1/p, where p=1/2, yielding E(T)=2. This represents an expected two trials per execution of the algorithm.

1. The expected number of trials needed to find a quadratic nonresidue is 2 (as calculated).
2. If each trial (including selection and verification) takes O(1) time, the expected running time of the modified NQUAD algorithm is O(1).
3. The probability of executing more than k trials decreases exponentially with k, and $lim_{k\to\infty} \frac{1}{2^k} = 0$, which means the probability of an infinite computation is zero.

Problem 7.1 [RA_Mot]

In this problem we will see that Theorem 7.1 is actually just a special case of Theorem 7.2. In the setting of Theorem 7.1. construct a multivariate polynomial Q such that $Q \equiv 0$ if and only if AB = C. and then apply Theorem 7.2 to derive result in Theorem 7.1.

Answer

Defining Polynomial Q

Given matrices A, B and C all of size n×n over a field F, we need to construct a multivariate polynomial Q such that Q≡0 if and only if AB=C.

Let the entries of $a_{i,j}$ and $b_{i,j}$ and $c_{i,j}$, respectively. The polynomial Q is defined over the field F as follows:

$$Q(x_1, \ldots, x_{n^2}, y_1, \ldots, y_{n^2}) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(\sum_{k=1}^{n} a_{ik} b_{kj} - c_{ij} \right)^2$$

Where $x_1, \ldots, x_{n^2}$ and $y_1, \ldots, y_{n^2}$ represent the entries of A and B respectively, rearranged into vectors. This polynomial Q is zero if and only if each individual product $\sum_{k=1}^{n} a_{ik} b_{kj}$ equals $c_{i,j}$, hence AB=C.

From Theorem 7.2

Theorem 7.2 states that for a multivariate polynomial Q of total degree d, and a finite set S⊂F from which variables are chosen independently and uniformly, the probability that Q evaluates to zero is at most d/|S|.

1. Total Degree: In Q, the highest total degree of any term arises from terms like $(a_{ik} b_{kj})^2$ within the squares, contributing a degree of 2 from each matrix entry multiplication where a_{ik} and b_{kj} are both potentially non-zero. Given n such multiplications per entry in the sum for C, the degree is 2n.
2. Applying the Bound: If r values are selected from $\{0, 1\}^n$ (binary vectors if we consider the simplest case of F), Theorem 7.2 gives a bound on the probability $Pr[ABr = Cr] \leq 2n/2^n$ where the polynomial evaluates to zero, matching the setting of Theorem 7.1 which discusses the case of $Pr[ABr = Cr] \leq 1/2$.

In conclusion

By linking the polynomial representation of the matrix equation $AB = C$ and applying Theorem 7.2, we can deduce the result of Theorem 7.1 as a specific application. This shows that the probability of a random binary vector r satisfying $ABr = Cr$ when $AB \neq C$ is indeed bounded by 1/2.

Problem 7.2 [RA_Mot]

Two rooted trees T1 and T2 are said to be isomorphic if there exists a one- to-one onto mapping f from the vertices of T1 to those of T2 satisfying the following condition: for each internal vertex v of T1 with the children $V_1, \ldots, V_k$. the vertex f(v) has as children exactly the vertices $f(V_1), \ldots, f(V_k)$. Observe that no ordering is assumed on the children of any internal vertex. Devise an efficient randomized algorithm for testing the isomorphism of rooted trees and analyze its performance. (Hint: Associate a polynomial P_v with each vertex v in a tree T. The

polynomials are defined recursively. the base case being that the leaf vertices all have $P = X_0$. An internal vertex v of height h with the children $V_1, \ldots, V_k$ has its polynomial defined to be

$$(X_h - P_{V_1})(X_h - P_{V_2}) \ldots (X_h - P_{V_k})$$

Note that there is exactly one indeterminate for each level in the tree.

Remark: There is a linear time deterministic algorithm for this problem based on a similar approach. R~fer to Aho. Hopcroft and Ullman [5].

Answer

1. **Polynomial Assignment:**
    - Begin at the leaves of each tree and assign each leaf vertex a polynomial $P_v = X_0$, where X_0 is a constant polynomial (e.g., could be a specific indeterminate or value).
    - Recursively, compute the polynomial for each internal vertex based on its children's polynomials. If an internal vertex v has children $V_1, \ldots, V_k$, then define:

$$P_v = (X_h - P_{V_1})(X_h - P_{V_2}) \ldots (X_h - P_{V_k})$$

where X_h is a unique indeterminate associated with the height h of the vertex v in the tree.

2. **Randomization:**
    - Choose a random value for each X_h from a sufficiently large set (or field) to minimize the probability of collision (i.e., different trees having the same polynomial representation by coincidence).
3. **Comparison:**
    - Evaluate the polynomials at the roots of both trees T_1 and T_2. If the evaluated polynomials are identical for the chosen values of X_h, the trees are likely isomorphic.
    - Repeat the comparison multiple times with different random values for the X_h variables to reduce the probability of a false positive.

Performance And Efficiency:

Correctness:

- If the polynomials of the root vertices of both trees are identical for one set of random assignments of X_h, it strongly suggests the trees are isomorphic because the structure defined by the recursive polynomial generation ensures that any permutation of children (since no order is assumed) would still yield the same polynomial if the trees are indeed isomorphic.

Error Probability:

- The error in this algorithm could occur if non-isomorphic trees yield the same polynomial due to a specific "unlucky" choice of random values for X_h. However, this probability can

be made arbitrarily small by choosing the values from a large enough field or set.

Complexity:

- The algorithm is efficient because the computation of each polynomial at each node involves only operations proportional to the degree of the node (number of children), and the recursion runs depth-first from the leaves to the root, thus processing each vertex exactly once.
- The overall complexity in terms of polynomial operations would generally be linear with respect to the number of vertices in the trees, assuming polynomial operations take constant time, which is a common assumption in theoretical analyses.

Problem 7.4 [RA_Mot]

Consider the problem of deciding whether two integer multisets S_1 and S_2 are identical in the sense that each integer occurs the same number of times in both sets. This problem can be solved by sorting the two sets in O(n log n) time. where n is the cardinality of the multisets. Suggest a way of representing this as a problem involving a verification of a polynomial identity, and thereby obtain an efficient randomized algorithm. Discuss the relative merits of the two algorithms. keeping in mind issues such as the model of computation and the size of the integers being operated upon. (See also Problem 6.20.)

Answer

Polynomial Representation of Multisets Each multiset S_1 and S_2 can be represented as a polynomial where each integer a in the multiset contributes a term x^a to the polynomial, and the coefficient of x^a represents the multiplicity of a in the multiset. Thus, for a multiset S, the polynomial $P_s(x)$ can be represented as:

$$P_S(x) = \sum_{a \in S} c_a x^a$$

where c_a is the count of integer a in multiset S.

Randomized Algorithm Using Polynomial Identity Testing

1. Polynomial Construction: Construct two polynomials $P_{s_1}(x)$ and $P_{s_2}(x)$ as described, one for each multiset.
2. Random Evaluation: Evaluate both polynomials at a randomly chosen value x=r. This value r should be chosen from a sufficiently large range to minimize the probability of collision (where different polynomials might yield the same value).
3. Comparison: Compare $P_{s_1}(r)$ and $P_{s_2}(r)$. If $P_{s_1}(r) = P_{s_2}(r)$ for several random choices of r, with high probability, the multisets S_1 and S_2 are identical. If $P_{s_1}(r) \neq P_{s_2}(r)$ for any r, then the multisets are definitely not identical.

Compariing with Sorting-based Algorithm

- Performance: Sorting each multiset takes O(nlogn) time, which is predictable and guaranteed. The polynomial identity test typically runs in linear time for construction and evaluation but requires random choice and possibly multiple evaluations to reduce error probability.
- Model of Computation: Sorting algorithms are straightforward and don't rely on probabilistic outcomes. They are preferable in environments where deterministic results are critical. Polynomial identity tests, however, are beneficial when quick, approximate results are acceptable, particularly in large-scale data environments.
- Integer Size: The polynomial method can handle very large integers efficiently as long as they can be processed within the chosen data structures. In contrast, sorting large integers might involve overhead depending on the sorting algorithm and machine architecture.

Problem 7.13 [RA_Mot]

Consider the two-dimensional version of the pattern matching problem. The text is an n x n matrix X, and the pattern is an m x m matrix Y. A pattern match occurs if Y appears as a (contiguous) sub-matrix of X. To apply the randomized algorithm described above, we convert the matrix Y into an m^2-bit vector using the row-major format. The possible occurrences of Y in X are the m^2-bit vectors $X(j)$) obtained by taking all $(n - m + 1)^2$ sub-matrices of X in a row-major form. It is clear that the earlier algorithm can now be applied to this scenario. Analyze the error probability in this case, and explain how the fingerprints of each $X(j)$) can be computed at a small incremental cost.

Answer

For the two-dimensional pattern matching:

First lets make Matrix to Bit Vector: Convert the m×m pattern matrix Y into an m^2 -bit vector using row-major order (traverse each row of the matrix from left to right and then proceed to the next row). Similarly, each m×m sub-matrix of X is also converted into an m^2 -bit vector $X(j)$ in row-major format.

Then we should compute Fingerprints:

- Compute a "fingerprint" for the bit vector of Y, and similarly for each potential m^2 -bit vector $X(j)$ derived from the sub-matrices of X.
- A common approach to fingerprinting is using a hash function designed to minimize collisions. A suitable choice could be a polynomial rolling hash function, which can effectively handle the string-like structure of the bit vectors.

Computing Fingerprints incrementally on top of each other:

- For each $X(j)$ and subsequent $X(j+1)$ (the next possible sub-matrix in row-major traversal), compute the fingerprint based on the previous fingerprint rather than from scratch.
- For example, if moving from $X(j)$ to $X(j+1)$ involves shifting the sub-matrix window to the right by one column, remove the effect of the leftmost column of $X(j)$ and add the effect of the new column that comes into $X(j+1)$.
- This approach significantly reduces the computational cost because it only requires updating parts of the hash that change rather than recomputing it entirely for each sub-matrix.

Error probability in respect to hash collisions:

- The probability of error in this context arises primarily from the possibility of hash collisions, where two different bit vectors (i.e., different sub-matrices) might yield the same hash value.
- If using a well-designed hash function and a sufficiently large hash space, the probability of collisions can be significantly reduced. However, it's often a trade-off between computational efficiency and collision risk.
- Generally, the error probability is inversely related to the size of the hash space. For cryptographic hash functions, this is typically quite low.

Problem 13.1 [RA_Mot]

(Due to D.O. Sleator and R.E. Tarjan [379].) Show that the LRU algorithm for paging is k-competitive. What can you say about its competitiveness coefficient?

Answer

To analyze the performance of LRU in the context of competitive analysis, we compare the cost incurred by the algorithm to the cost incurred by an optimal offline algorithm that knows all future requests.

Definition of k-Competitive

An online algorithm is k-competitive if there exists a constant c such that for every input sequence, the cost of the online algorithm is at most k times the cost of the optimal offline algorithm, plus c. Formally, for every input sequence σ,

$$Cost_{ALG}(\sigma) \leq k.Cost_{OPT}(\sigma) + e$$

The LRU paging algorithm evicts the least recently used page when a page fault occurs and a new page needs to be loaded into a full memory. This strategy is based on the assumption that

pages accessed recently will likely be accessed again soon.

Ref:

- https://embedded.cs.uni-saarland.de/lectures/realtimesystems/cachePredictabilityCompetitivenessSensitivity.pdf
- https://embedded.cs.uni-saarland.de/lectures/realtimesystems15/blocklevelcompetitivenessAnnotated.pdf

Competitiveness of LRU:

1. A page fault occurs under LRU when a requested page is not in the cache. LRU replaces the page that has not been used for the longest time. The optimal offline strategy (OPT), on the other hand, can foresee future requests and always evicts the page that will not be needed for the longest time in the future.
2. Research by Sleator and Tarjan on the competitiveness of paging algorithms established that LRU is k-competitive where k is the number of pages the algorithm can store. The cost considered here is the number of page faults.
3. Well what is k?
    - The competitiveness coefficient k for LRU comes from the analysis showing that in the worst-case scenario, LRU's performance will be no worse than k times the number of page faults of the optimal algorithm, plus a constant related to the cache size.
    - This result is derived from considering the number of times a page can be brought into and evicted from the cache between two accesses by the optimal algorithm.

Now we can demonstrate that the LRU algorithm is k-competitive for paging:

1. LRU Algorithm and OPT: Assume the cache can hold k pages. The LRU algorithm evicts the least recently used page when a new page needs to be loaded into a full cache. In contrast, an optimal offline algorithm (OPT) can always evict the page that will not be needed for the longest time in the future.
2. Cost Analysis:
    - Each page fault under LRU incurs a cost of 1.
    - The competitive analysis focuses on comparing the cost incurred by LRU against the cost incurred by OPT over the same sequence of page requests.
3. The Potential Method:
    - Define a potential function Φ that measures the "distance" between the state of the cache under LRU and the state of the cache under OPT.
    - The potential function could be defined as the number of pages that are different in the LRU cache compared to the OPT cache.
    - Changes in this potential function help to measure the additional cost that LRU might incur over OPT during transitions (page replacements).

4. Bounding the Costs:
    - When LRU incurs a page fault (cost = 1), if OPT does not incur a fault, the potential Φ may increase by at most 1 (because one wrong page in LRU is replaced by a correct page).
    - If OPT also incurs a page fault, the potential Φ might decrease, which can offset the cost of the fault for LRU.
    - The amortized cost of LRU, considering the change in potential, remains constant or decreases.
5. Competitive Ratio:
    - The competitive ratio k for LRU is derived by observing that in the worst case, every time LRU evicts a page, OPT could ideally retain all pages that will be used in the near future, and thus LRU's faults could seem misaligned with OPT's faults.
    - However, because of the way the potential function is defined and adjusted, the maximum ratio of the cost of LRU to OPT is bounded by k, the size of the cache. Thus, LRU is k-competitive.

In overal, the competitive analysis relies on the insight that even though LRU does not know the future, it makes a locally optimal decision based on past usage. This heuristic, while not always matching the foresight of OPT, still guarantees that the cache's turnover is managed efficiently compared to the optimal method known only in hindsight.

Problem 13.2 [RA_Mot]

(Due to D.O. Sleator and R.E. Tarjan [379].) Show that the FIFO algorithm for paging is k-competitive. What can you say about its competitiveness coefficient?

Answer

To show that the FIFO algorithm for paging is k-competitive, we analyze its performance relative to an optimal offline algorithm (OPT) as we did in previous problem.

1. **FIFO Behavior:** FIFO manages its cache by evicting the oldest page (the first one that came in) when a new page needs to be loaded, and the cache is full.
2. **Competitive Ratio Determination:**
    - Consider any sequence of page requests.
    - Let the cost of serving these requests using FIFO be the number of page faults (when a requested page is not in the cache and must be loaded).
    - The OPT can potentially have the least number of page faults because it knows future requests and can always keep the most immediately necessary pages in the

cache.

3. **Bounding the Costs:**
    - For every page fault experienced by FIFO, there must be k distinct pages requested since the last time the faulting page was in the cache (otherwise, it would not have been evicted).
    - OPT must replace at least one page during these k distinct requests; therefore, it incurs at least one fault every k requests in the worst case.
4. **Competitive Analysis:** Since OPT has a minimum of one fault for every k requests that cause a fault in FIFO, the maximum ratio of faults between FIFO and OPT is k. Therefore, FIFO is k-competitive.
5. **Competitiveness Coefficient:** The competitiveness coefficient k implies that the worst-case performance of FIFO is at most k times the number of faults of OPT, plus a possible additive constant that depends on the initial state of the caches and the specific sequence of page requests.

Problem 13.3 [RA_Mot]

Show that the LFU algorithm does not achieve a bounded competitiveness coefficient.

Answer

1. **LFU Behavior:** LFU evicts the page with the lowest frequency of access when a new page needs to be loaded into a full cache.
2. **COnsidering Adversarial Input Sequence:** Construct an input sequence where the least frequently used page changes frequently due to the introduction of new pages or slight variations in access frequencies. For example:
    - Start with a sequence of pages 1,2,...,k to fill the cache.
    - Continue with pages k+1,1,k+2,2,... where each new page k+i is accessed once more than the previous least accessed page which just got evicted.
3. **OPT Strategy:** An optimal offline algorithm (OPT) can anticipate future requests and keep pages that will be accessed more frequently, leading to fewer evictions.
4. **Comparison of Costs:**
    - Under LFU, the page eviction might lead to subsequent faults when the evicted page is accessed again soon after.
    - OPT may incur significantly fewer faults by evicting pages that will not be accessed for the longest time ahead.
5. **Unbounded Competitive Ratio:** In the adversarial sequence, the number of page faults for LFU can be much higher compared to OPT. For instance, LFU might incur a fault nearly

every time if every new access slightly shifts the frequency distribution, whereas OPT strategically keeps pages that will minimize future faults.

So, due to LFU's strict adherence to frequency and inability to adapt to changes in access patterns that are not reflected in the long-term historical frequencies, it can perform significantly worse than OPT in certain scenarios. Thus, LFU does not have a bounded competitiveness coefficient; its performance relative to OPT can drop drastically based on specific patterns of access frequencies.