Algorithms and Probability (FS2025)
Week 10

Rui Zhang

May 2, 2025

Contents

1 Content

From here on out in the course, most of the things we will do is going to be applications of our probabilistic methods for the sake of improving algorithms:

1.1 Prime Number Testing

Finding prime numbers is a very important process in cryptography. In order to generate prime numbers, what is often done is just choose a random number and testing if that number is prime (again, just gambling).

This testing of prime numbers can be done in multiple ways:

Naive Solution

The most simple solution is to go through all numbers i between 2 and $n-1$ and check if our number q is divisible by i. We can improve this strategy by not ranging through all numbers n, but instead going all the way up to $\sqrt{n}$.

Probabilistic Solution

Our probabilistic solution uses two fundamental results from discrete mathematics:

Theorem: If n is prime, then $\mathbb{Z}_n$ together with addition and multiplication modulo n is a field. In particular, $x^2 \equiv 1$ (mod n) has only the solutions $n-1$ and 1

Theorem: If n is prime then for all $0 < a < n$, we have

$$a^{(n-1)} \equiv 1 \text{ (mod n)} \tag{1}$$

Combining these results, we come up with the following procedure: Write $n-1 = d \cdot 2^k$ for some odd number d. Choose some number a between $\{2, \ldots, n-1\}$ randomly uniformly, then calculate $x = a^d$. If x is already 1 or $n-1$ at this point, then our procedure below (which will consist of iterative squaring) will always result in 1, meaning that we will never find proof that x is composite. Thus we immediately return "n is prime".

Continuously square a (via iterative squaring) to reach the exponent $n-1$. After each iteration, we check the following:

- Have we reached $n-1$? If yes, then we know that the next iteration will surely yield a 1, since squaring $n-1$ modulo n is 1. If we reached this point, then we don't really know anything anymore. Our number n could still be prime, but we will never really find proof of this (because the next iterations will make the number stay 1), and we thus return "n is prime".
- Have we reached 1 without our previous iteration resulting in a $n-1$ (or 1, but then we could repeat this current argument for the iteration before)? Then the 1st theorem from above is violated and we have found proof that our number n cannot be prime. Thus we return "n is composite"

Since we iteratively square, the algorithm is in $O(\ln n)$, exactly what we want. You will realize that the output "n is composite" is always correct, since we will have found proof in our algorithm in that case, whereas the output "n is prime" does not have to be, since we could have just got unlucky while choosing a. The probability that we make a mistake and output "n is prime" despite n being composite, is at most $1/4$ (trust me bro.)

1.2 Finding Arbitrarily Long Paths

You have found lots of algorithms for shortest paths in the precursor course to this one. Sometimes however, say in task scheduling, we want to model our tasks as vertices and directed edges as dependencies, with the goal of finding the longest path (also called critical path). We are going to reduce our scope for now and look at the following problem:

Definition: Long Paths: Given a graph G and a number B, find out if there exists a path of length at least B in G.

Maybe don't spend too much time thinking of possible efficient (polynomial time) solutions to this problem, because:

Theorem: If we can solve the long paths problem for graphs on n vertices in $t(n)$ time, then we can solve the hamiltonian cycle problem in $t(2n-2)+O(n^2)$ time

And since the hamiltonian cycle problem is NP-Complete, Long paths is so too.

1.3 Copium: Finding Kinda, Sorta Long Paths

After learning that we likely won't come up with an efficient solution to Long Paths, we go through the 5 stages of grief. Afterwards, we calm down and take mass amounts of copium and decide to manage our expectations a bit and find kinda, sorta long paths. From here on out, we will refer to the length of paths by the number of edges in them.

Definition: Kinda, Sorta Long Paths: Given a graph G find out if there exists a path of length at least $\log n$ in G.

Our idea is the following: We will develop a probabilistic algorithm which does multiple iterations of the following:

1. Color the graph arbitrarily (not a vertex coloring, just random color assignments)

```
BUNT(G, i)
1: for all v ∈ V do
2:     P_i(v) ← ∅
3:     for all x ∈ N(v) do
4:         for all R ∈ P_{i−1}(x) mit γ(v) ∉ R do
5:             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}}
```

Figure 1: Pseudo for the BUNT algorithm

2. Find all colorful paths := sequences of vertices which do not repeat colors
3. Among those colorful paths, if one of them is of length $\log n$, then we can return "true" otherwise repeat

If no iteration finds a colorful path, then return "false" (although this answer might be wrong). This is an instance of a monte-carlo algorithm. There are two things we have to look at in more detail for this to make sense: Step 2. of the procedure and the analysis of our algorithm.

First, let's find an algorithm for step 2. Let us fix a number k. Given a graph $G = (V, E)$ and a coloring $\gamma : V \to [k]$ we want to find colorful paths and determine in particular if there exists a path of length $k-1$ (edges). To this end, we will use the following DP algorithm (yay, your favorite.)

Definition: DP-Definition: Let $P_i(v)$ be our DP entry defined as:

$$P_i(v) := \left\{ S \in \binom{[k]}{i+1} \mid \exists \text{ a colorful path which ends in } v \text{ and traverses exactly the colors in } S \right\}$$

And we define our base cases and recursion as:

Definition:

- Base Case 1: $P_0(v) = \{\{\gamma(v)\}\}$, since a path of 0 edges is just the single vertex v and thus only contains the color $\gamma(v)$.
- Recursion: $P_i(v) = \bigcup_{x \in N(v)} \{R \cup \{\gamma(v)\} \mid R \in P_{i-1}(x) \text{ and } \gamma(v) \notin R\}$

  which intuitively means that we range over all possible neighbors of v and append the color of v to the colors of the paths ending in the neighbors of length one less, as long as that path does not contain our current color.

Using this we can define a function "BUNT" which takes as input i and calculates $P_i(v)$ for all vertices $v \in V$. We can iterate through all i's until $k-1$ and apply this function in each iteration. If $P_{k-1}(v)$ is empty for all vertices v, then we do not have a path of length $k-1$. Otherwise we do.

Runtime Analysis "BUNT": Instead of looking at the full algorithm, let us consider BUNT first. In this function, we have that the innermost for loop goes over all elements R in $P_{i-1}(x)$ with $\gamma(v)$ not contained in R. Due to the definitino of P_{i-1}, we get that $|P_{i-1}| \leq \frac{k}{i-1+1=i}$, so we have at most this many candidates for R. To check the condition of $\gamma(v) \notin R$, we still need to loop over R, which has a size of exactly i elements.

Finally, we have to loop over all neighbors from all vertices v, which is in $O(m)$ as we know from A&D. This leaves us with:

Theorem: The runtime of "BUNT" is $O\left(\binom{k}{i} \cdot i \cdot m\right)$ and the total runtime of the algorithm is

$$O\left(\sum_{i=1}^{k-1} \binom{k}{i} \cdot i \cdot m\right) \leq O\left(2^k km\right)$$

where we used the equality $\sum_{i=0}^{k} \binom{k}{i} = 2^n$

We are still not done, we proposed this algorithm with the idea in mind that we color the graph somehow and run this previous algorithm. Thus, our idea is the following: we assign colors randomly and independently to every vertex in each round / iteration with $k = \log n + 1$ and then apply the algorithm with the same k. Each iteration will take:

- $O(n)$ time to color the vertices randomly
- $O\left(2^{\log n}(\log n)m\right)$ (dominant factor)

And what is the probability of a mistake in each round / iteration: Remember, if we return that there is a colorful path of length k, then there definitely exists a path of length k. However, if we return that there is no colorful path, then we might have just got unlucky when coloring randomly. What is this probability of getting lucky, i.e. coloring a path of length $k-1$ with k distinct colors? Well, we have a total of k^k possible colorings (each vertex has k choices for colors) and we have $k!$ many colorful colorings. This leaves us with an error probability of

$$p_{error} = 1 - \frac{k!}{k^k} \leq 1 - e^{-k}$$

Where the second inequality follows from the taylor series of e^{-k}. For each iteration of our monte-carlo algorithm we have an error probability of at most $1 - e^{-\lambda}$ for some λ of our choice. If we want a total error probability of at most e^{-k}, then we can apply Theorem 2.74 with $\varepsilon = e^{-k}$ and $\delta = e^{-\lambda}$ and get that we have to repeat this algorithm $N = \lambda e^k$ times, resulting in a total runtime of

$$O(N \cdot 2^{\log n}(\log n)m) = O(\lambda 2^{\log n}(\log n)m) = O(\lambda(2e)^{\log n}(\log n)m)$$