# CS 161 Winter 2021 Section 5

February 11, 2021

## 1 Pattern Matching with Rolling Hash

In the Pattern Matching problem, the input is a *text* string $T$ of length $n$ and a *pattern* string $P$ of length $m < n$. Our goal is to determine if the text has a (consecutive) substring[1] that is exactly equal to the pattern (i.e. $T[i \dots i+m-1] = P$ for some $i$).

1. Design a simple $O(mn)$-time algorithm for this problem.
2. Can we find a more efficient algorithm using hash functions? One naive way to do this is to hash $P$ and every length-$m$ substring of $T$. What is the running time of this solution?
3. Suppose that we had a universal hash family $H_m$ for length-$m$ strings, where each $h_m \in H_m$ the sum of hashes of characters in the string:

$$h_m(s) = h(S[0]) + \cdots + h(S[m-1]). \tag{1}$$

   Explain how you would use this hash family to solve the pattern matching problem in $O(n)$ time.

   (Hint: the idea is to improve over your naive algorithm by **reusing your work**.)
4. Unfortunately, a family of "additive" functions like the one in the previous item cannot be universal. Prove it.
5. The trick is to have a hash function that looks almost like (1): the hash function treats each character of the string is a little differently to circumvent the issue you discovered in the previous part, but they're still related enough that we can use our work. Specifically, we will consider hash functions parameterized by a fixed large prime $p$, and a random number $x$ from $1, \dots, p-1$:

$$h_x(S) = \sum_{i=0}^{m-1} S[i] \cdot x^i \pmod{p}.$$

   For fixed pair of strings $S \neq S'$, the probability over random choice of $x$ that the hashes are equal is at most $m/p$, i.e.

$$\Pr[h_x(S) = h_x(S')] \leq m/p.$$

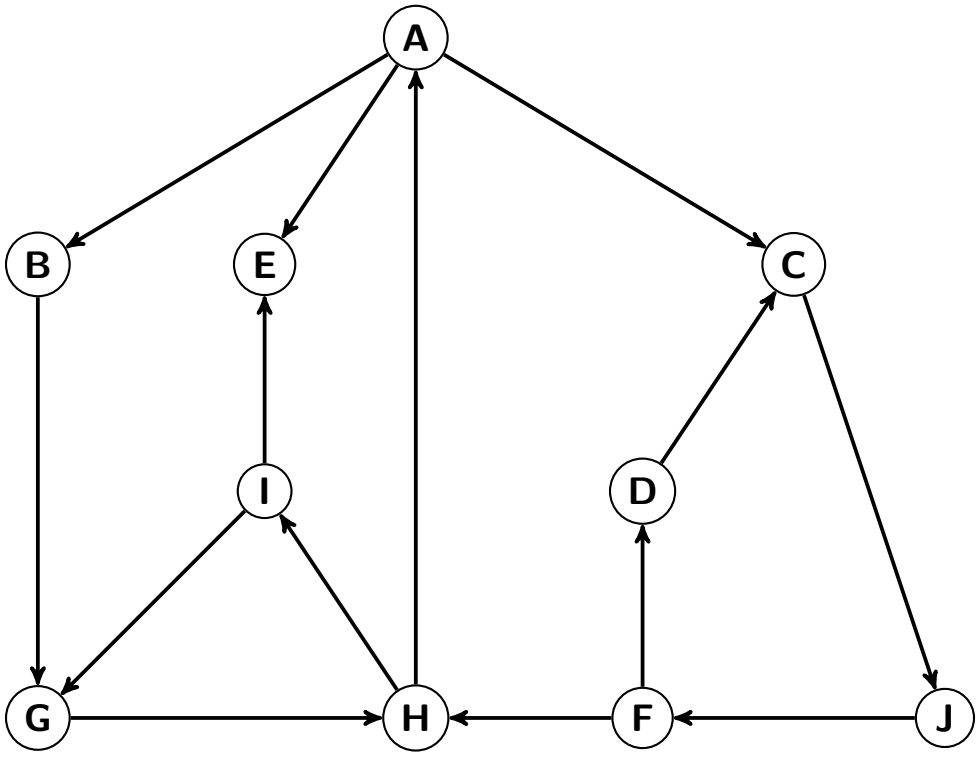
   (This follows from the fact that a polynomial of degree $(m-1)$ can have at most $m$ zeros. Do you see why?)

   Design a randomized algorithm for solving the pattern matching problem. The algorithm should have worst-case run-time $O(n)$, but may return the wrong answer with small probability (e.g. $< 1/n$). (Assume that addition, subtraction, multiplication, and division modulo $p$ can be done in $O(1)$ time.)

[1] In general, *subsequences* are not assumed to be consecutive, but a *substring* is defined as a consecutive subsequence.

6. How would you change your algorithm so that it runs in *expected* time $O(n)$, but always return the correct answer?
7. Suppose that we had one fixed text $T$ and many patterns $P_1, \ldots P_k$ that we want to search in $T$. How would you extend your algorithm to this setting?

## 2 Depth First Search




1. What are all the strongly connected components? (i.e. groups of vertices such that there exists a path between any two vertices in the group)
2. Perform DFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the start time and the finish time. Highlight the edges in the tree generated from the search.
3. Perform BFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the discovery order. Highlight the edges in the tree generated from the search.

## 3 True or False

1. If $(u, v)$ is an edge in an undirected graph and during DFS, $finish(v) < finish(u)$, then $u$ is an ancestor of $v$ in the DFS tree.
2. In a directed graph, if there is a path from $u$ to $v$ and $start(u) < start(v)$ then $u$ is an ancestor of $v$ in the DFS tree.

## 4 Bipartite Graphs

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, $U$ and $V$ such that every edge $(u, v)$ either connects a vertex from $U$ to $V$ or a vertex from $V$ to $U$. A graph is bipartite if there is a 2-coloring such that vertices in a set are colored with the same color. In lecture, we saw an algorithm using BFS to determine where a graph is bipartite.

Design an algorithm using DFS to determine whether or not a graph is bipartite.

## 5 Source Vertices

A source vertex in a graph $G = (V, E)$ is a vertex $v$ such that all other vertices in $G$ can be reached by a path from $v$. Say we have a directed, connected, acyclic graph that has at least one source vertex.

1. Describe a naive algorithm to find a source vertex.
2. Describe an algorithm that operates in $O(V + E)$ time to find a source vertex.