CS 1332 - Week 12

Monday

<u>Topics</u>

I. KMP Tracing

Failure Table										
idx	0	1	2	3	4	5	6	7	8	9
pat	b	a	b	a	a	b	a	b	a	b
fail	0	0	1	2	0	1	2	3	4	3

Tracing																		
text	b	a	b	a	b	a	b	a	b	a	b	a	a	b	a	b	a	b
pat	b	a	b	a	a	b	a	b	a	b								
			b	a	b	a	a	b	a	b	a	b						
					b	a	b	a	a	b	a	b	a	b				
							b	a	b	a	a	b	a	b	a	b		
									b	a	b	a	a	b	a	b	a	b

II. Rabin-Karp

A. Premise - computes hashcodes for substrings and only performs character comparisons when the hashes match

B. Calculating the hash of a string/substring

1. Select a base number - the ideal base number is prime and fairly large, this helps eliminate hashcode collisions
2. For each character:
    a) Convert char to an integer - eg. use the ASCII value
    b) Computer a power of base - to account for the position of the char in the string, we multiply the character by a power of base
3. Add all values together to form the entire hashcode
4. Eg. pattern “90210”, size = 5, base = 10
    a) hash(pattern)

$= 9*10^4 + 0*10^3 + 2*10^2 + 1*10^1 + 0*10^0$

$= p[0]*base^{(size-1-0)} + p[1]*base^{(size-1-1)} + p[2]*base^{(size-1-2)}...$

$= (n=0, size-1)\ \Sigma\ p[n]*base^{(size-1-n)}$

C. Rolling the text hash
   1. newHash = (oldHash - hash of first char in oldHash) * base + next char
   2. Eg. text “48902107” → roll hash from hash(“48902”) to hash(“89021”)
      a) oldHash = 48902, base = 10
      b) remove hash of first character: 48902 - 40000 = 8902
      c) make room for next character: 10 * 8902 = 89020
      d) add on the next new character: 89020 + 1 = 89021

D. Algorithm
   1. Calculate the hash of the pattern
   2. Calculate the initial hash of the text (the first m characters of the text)
      a) Recommendation: calculate the initial text hash and the pattern hash in the same loop starting from index m-1 and going to 0, this way you can calculate the power of base needed by starting a variable at 1 and multiplying it by base for each iteration
   3. Compare the pattern hash to the text hash
      a) If they match → compare the actual characters
      b) If they don’t → roll the hash and compare again

E. Hashcode Collisions
   1. If we have a bad hashcode (eg. small or non-prime base), we’ll have lots of places where the hashcodes are equal, but the characters are different
      a) Eg. text “aabbcaba”, pattern “cab”, base = 1, hash a to z = 0 to 26
         (1) Pattern hash = c + a + b = 2 + 0 + 1 = 3
         (2) Initial text hash = a + a + b = 0 + 0 + 1 = 1 → no match
         (3) Rolled hash = 1 - a + b = 1 - 0 + 1 = 2 → no match
            (a) (old - oldChar) * base + newChar
         (4) Rolled hash = 2 - a + c = 2 - 0 + 2 = 4 → no match
         (5) Rolled hash = 4 - b + a = 4 - 1 + 0 = 3 → match
            (a) compare t: “bca” to p: “cab” → no match
         (6) Rolled hash = 3 - b + b = 3 - 1 + 1 = 3 → match
            (a) compare t: “cab” to p: “cab” → match!

Wednesday

<u>Topics</u>

I. Pattern Matching Efficiencies

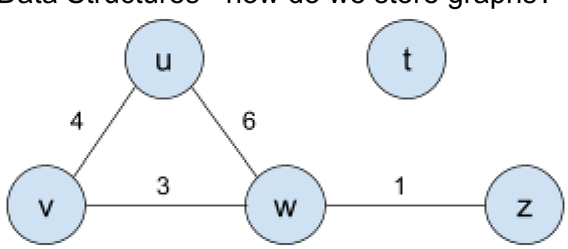
	Worst	Best (first occurrence)	Best (all)
Brute Force	O(mn)	O(m)	O(mn)
Boyer-Moore	O(mn)	O(m)	O(n/m)
Knuth-Morris-Pratt	O(m + n)	O(m)	O(m + n)
Rabin-Karp	O(mn)	O(m)	O(m + n)

II. Graphs ADT
   A. Graphs are made up of a set of vertices, V, and a set of edges, E
   B. Terminology
      1. Order(G) - number of vertices in graph G
      2. Size(G) - number of edges in graph G
      3. Sparse - few edges relative to the number of vertices
      4. Dense - many edges relative to the number of vertices
      5. Path - edges you traverse to go from one vertex to another
         a) Simple - vertices traversed in the path are only visited once
         b) Cycle - at least one vertex is visited more than once in a path
         c) Length of path - number of edges traversed
      6. Edge - connects two vertices, edge(a, b)
         a) Undirected Edge - can go from a to b and b to a
            (1) Undirected Graph - ALL edges are undirected
         b) Directed Edge - can only go from a to b
            (1) Directed Graph - ALL edges are directed
         c) Weighted Edge - an edge has a “cost” associated with traversing it, eg. if vertices are locations, edge weights could be miles between two locations
            (1) Weighted Graph - ALL edges are weighted
         d) Self Loop - an edge that connects a vertex to itself
         e) Parallel Edges - two edges with the same source and destination
   C. Information Methods
      1. vertices() - returns iteration (list generated by an iterator) of all vertices
      2. edges() - returns iteration of all edges
      3. numV() - returns count/number of vertices
      4. numE() - returns count/number of edges

Friday

<u>Topics</u>

I. Graph Data Structures - how do we store graphs?



   A. Edge List - stores all the edges of the graph in a list
      1. Edge list - [(v, u, 4), (u, w, 6), (w, v, 3), (w, z, 1)]
      2. Pros - easy access to all edges
      3. Cons - hard to access all vertices, doesn’t store disconnected vertices

B. Adjacency List - like a hashmap, the keys are the vertices and each vertex key's value is a list of the edges adjacent to it
    1. Adjacency list

        w → [(w, u, 6), (w, v, 3), (w, z, 1)]
        u → [(u, v, 4), (u, w, 6)]
        v → [(v, u, 4), (v, w, 3)]
        z → [(z, w, 1)]
        t → []

    2. Pros - easy access to all vertices and edges incident to a given vertex
    3. Cons - hard to access all edges, undirected edges are stored twice

C. Adjacency Matrix - stores a 2D array where matrix[v][w] would give the weight of the edge between vertices v and w if it exists
    1. Adjacency matrix

	w	u	v	z	t
w		6	3	1	
u	6		4		
v	3	4			
z	1				
t					

    2. Pros - easy to find an edge between two vertices (O(1) array access)
    3. Cons - extra memory is taken up when the graph is sparse or undirected

II. Graph Traversals

A. Depth-First Search (DFS)
    1. Premise - follow one path as deep as possible before backtracking and going down another path
        a) The pre-, post-, and inorder BST traversals were depth-first
    2. Algorithm - can be done with a regular stack or recursive stack
        a) Recursive pseudocode

```
dfs(vertex v):
      mark v visited, output v (print or record)
      for each neighbor w of v:
            if w is unvisited:
                  dfs(w)
```