*Owen Lindsey*

*Professor Demland, David*

*CST-201 Exercise 7*

*10/20/2024*

**Exercise 3.2 - 5**

*Using String matching algorithm:*

***1. Keep the text length n = 1000.***

***2. Keep the pattern length m = 5.***

*3. +1 to count the last possible starting position for our algorithm.*   
 *n – m + 1 = 1000 – 5 + 1 = 996*  
***4. Subtract 5 because the last 5 positions in the text can't be starting positions.***

***5. Create sperate calculations for each pattern:***

***i. Pattern 0001:***

***ii. Calculate possible starting positions:***   
  *n – m + 1 = 1000 – 5 + 1 = 996.2.*   
 ***iii. For each starting position:***

*Compare characters until mismatch or end of pattern.*   
 *995 positions: 4 comparisons each (4 matches, mismatch on first character)*

***Iv. Total comparisons:***  
 *996 \* 4 = 3,984*

**Exercise 3.2 - 5**

*Using String matching algorithm:*

***i. Pattern 1000:***

***ii. Calculate possible starting positions:***

*n – m + 1 = 1000 – 5 + 1 = 996\**

***iii. For each starting position:***

*Compare all characters it will always match:*   
 *996 positions: 5 comparisons each (all match)*

***iii. Total comparisons:***  
 *996 \* 5 = 4980*

***I. Pattern 01010:***

***ii. Calculate possible starting positions:***  
 *n – m + 1 = 1000 – 5 + 1 = 996*

***iii. For each starting postion:***  
 *compare first character (match)*

*compare second character (mismatch)*

***iv. Total comparisons:***

*996 \* 2 = 1,992*

***6. Total of all comparisons:***   
 *a. 00001: 3,984 comparisons*

*b. 1000: 4,980 comparisons*

*c. 01010: 1,992 comparisons*

**Exercise 3.5 - 1**

*Graphing a matrix with adjacency lists:*

***a. Adjacency matrix and adjacency lists:***

***First, we list the vertices in alphabetical order: a, b, c, d, e, f, g***

1. *W****e create a 7x7 matrix (because there are 7 vertices), with rows and columns labeled in alphabetical order.***
2. ***We fill the matrix with 0s and 1s:*** 
   1. *Put 1 if there's an edge between the vertices*
   2. *Put 0 if there's no edge or it's the same vertex*
3. ***Looking at the graph, we fill in the 1s:*** 
   1. *a is connected to b, d, e*
   2. *b is connected to a, c, f*
   3. *c is connected to b, g*
   4. *d is connected to a, f*
   5. *e is connected to a, g*
   6. *f is connected to b, d*
   7. *g is connected to c, e*
4. ***The result is:***

*a b c d e f g*

*a*  *0 1 0 1 1 0 0*

*b* *1 0 1 0 0 1 0*

*c* *0 1 0 0 0 0 1*

*d*  *1 0 0 0 0 1 0*

*e* *1 0 0 0 0 0 1*

*f*  *0 1 0 1 0 0 0*

*g*  *0 0 1 0 1 0 0*

**Exercise 3.5 - 1**

*Graphing a matrix with adjacency lists:*

***Adjacency Lists:***

*a: b, d, e*

*b: a, c, f*

*c: b, g*

*d: a, f*

*e: a, g*

*f: b, d*

*g: c, e*

***b. Depth-First Search (DFS) starting from vertex a:***

***DFS Tree:***

*a*

*/*

*b*

*/*   
*c*  *f*

*\*   
*g*  *d*

*e*

**Exercise 3.5 - 1**

*Graphing a matrix with adjacency lists:*

***Order vertices were reached (pushed onto stack):***

*a, b, c, g, e, f, d*

***Order vertices became dead ends (popped off stack):***

*d, f, e, g, c, b, a*

***DFS process:***

1. *Start at a, push a*
2. *Visit b (first alphabetical neighbor of a), push b*
3. *Visit c (first unvisited alphabetical neighbor of b), push c*
4. *Visit g (only unvisited neighbor of c), push g*
5. *Visit e (unvisited neighbor of g), push e*
6. *Backtrack to b, visit f (next unvisited alphabetical neighbor), push f*
7. *Visit d (unvisited neighbor of f), push d*
8. *All vertices visited, pop off stack: d, f, e, g, c, b, a*

*This DFS traversal resolves ties by choosing the alphabetically first unvisited neighbor at each step.*

**Exercise 3.5 - 8**

**Bipartite graph DFS vs BFS:**

***a. DFS-based algorithm for checking whether a graph is bipartite:***

1. ***Choose two colors: red and blue.***
2. ***For each uncolored vertex v in the graph:***
   1. *Assign red color to v*
   2. *Call DFS\_Color(v)*
3. ***DFS\_Color(vertex) process:***
   1. *For each adjacent vertex u of vertex:*
      1. *If u is uncolored:*
         1. *Assign the opposite color of vertex to u*
         2. *Recursively call DFS\_Color(u)*
      2. *If u is colored and has the same color as vertex:*
         1. *Return false (graph is not bipartite)*
   2. *If all adjacent vertices are processed without conflicts, return true*
4. ***If DFS\_Color never returns false, the graph is bipartite. Return true.***

*The key idea is to color vertices alternating between red and blue as we explore deeper into the graph. Any color conflict between adjacent vertices immediately indicates the graph is not bipartite.*

**Exercise 3.5 - 8**

*Bipartite graph DFS vs BFS:*

***b. BFS-based algorithm for checking whether a graph is bipartite:***

1. ***Choose two colors: red and blue.***
2. ***Create an empty queue Q.***
3. ***For each uncolored vertex v in the graph:***
   1. *Assign red color to v*
   2. *Enqueue v into Q*
   3. *While Q is not empty:*
      1. *Dequeue a vertex u from Q*
      2. *For each adjacent vertex w of u:*
         1. *If w is uncolored:*
            1. *Assign the opposite color of u to w*
            2. *Enqueue w into Q*
         2. *If w is colored and has the same color as u:*
            1. *Return false (graph is not bipartite)*
4. ***If all vertices are processed without conflicts, the graph is bipartite. Return true.***

*This BFS approach colors vertices level by level, ensuring vertices at even distances from each starting point are one color (e.g., red) and those at odd distances are the other color (e.g., blue). This naturally creates the two-color partition required for a bipartite graph.*