**Encryption and Decryption using Graph Theory**

**Student Information:**

**Name:** Om Thanage

**Roll No:** 16010123217

**Course Name/Code:** DSM

**Date of Submission:** 25/10/24

**Name:** Omkar Dinde

**Roll No:** 16010123219

**Course Name/Code:** DSM

**Date of Submission:** 25/10/24

**Problem Statement**

The task is to develop a tool for encrypting and decrypting messages based on graph theory and lattice/logic principles. Specifically, the encryption should use graph traversal techniques, with the message being encoded as nodes in a graph and traversed based on random edges between characters. The tool should have a GUI to allow users to encrypt and decrypt messages using a key, and the tool should be able to decrypt an encrypted message when the correct encryption path is provided.

Input Requirements:

For encryption: A plain text message and an integer key.

For decryption: An encrypted message, a corresponding encryption path (as a list of indices), and the same integer key used during encryption.

Output Requirements:

For encryption: An encrypted message and the encryption path (a sequence of character indices used during the encryption process).

For decryption: The original (decrypted) message.

**Approach and Algorithm**

Graph Construction:

* We build a graph using an adjacency matrix where each character in the alphabet corresponds to a node. The matrix is seeded using the provided key to ensure deterministic encryption.
* Self-loops are added for each node, allowing nodes to connect to themselves.
* Encryption: Each character of the input message is converted to its corresponding index (using a predefined character map).
* For each character, a random adjacent node is selected using the adjacency matrix, and the new node (character) is used in the encrypted message. The path (original indices) is stored to facilitate decryption.
* Decryption: The path (provided during encryption) is used to retrieve the original indices of the characters.
* These indices are converted back into characters to form the original message

**Code Implementation:**

import numpy as np  
import random  
import string  
import tkinter as tk  
from tkinter import messagebox  
  
  
class GraphEncryption:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.letters = string.ascii\_lowercase + ' '  
 self.graph = self.\_build\_graph()  
  
 def \_build\_graph(self): #Build the graph  
 size = len(self.letters)  
 np.random.seed(self.key)  
 matrix = np.random.randint(0, 2, (size, size))  
 np.fill\_diagonal(matrix, 1)  
 return matrix  
  
 def \_char\_to\_index(self, char):  
 *"""Convert a character to its corresponding index in the graph."""* return self.letters.index(char)  
  
 def \_index\_to\_char(self, index):  
 *"""Convert an index back to its corresponding character."""* return self.letters[index]  
  
 def encrypt(self, message):  
 *"""Encrypt the message using graph traversal and store the traversal path."""* message = message.lower()  
 encrypted\_message = []  
 path = [] # Store path to use for decryption  
  
 for char in message:  
 current\_index = self.\_char\_to\_index(char)  
 # Find a connected node from the current node using the adjacency matrix  
 connections = np.where(self.graph[current\_index] == 1)[0]  
 next\_node = random.choice(connections) # Randomly pick a connected node  
 encrypted\_message.append(self.\_index\_to\_char(next\_node))  
 path.append(current\_index) # Store the original index for decryption  
  
 return ''.join(encrypted\_message), path  
  
 def decrypt(self, encrypted\_message, path):  
 *"""Decrypt the message by retracing the recorded path."""* decrypted\_message = []  
  
 for index in path:  
 # Using the recorded path to trace back the result  
 decrypted\_message.append(self.\_index\_to\_char(index))  
  
 return ''.join(decrypted\_message)  
  
  
# GUI code  
def encrypt\_message():  
 message = input\_message.get()  
 key = int(input\_key.get())  
  
 if not message:  
 messagebox.showwarning("Input Error", "Message cannot be empty!")  
 return  
  
 ge = GraphEncryption(key)  
 encrypted\_message, path = ge.encrypt(message)  
  
 # Results  
 output\_encrypted\_message.set(encrypted\_message)  
 output\_decrypted\_message.set(ge.decrypt(encrypted\_message, path))  
 output\_encryption\_path.set(str(path))  
  
  
def decrypt\_message():  
 encrypted\_message = input\_encrypted\_message.get()  
 path\_input = input\_encryption\_path.get()  
  
 # Parse the encryption path  
 try:  
 path = list(map(int, path\_input.strip('[]').split(',')))  
 except ValueError:  
 messagebox.showwarning("Input Error", "Invalid encryption path format! Use comma-separated integers.")  
 return  
  
 key = int(input\_key.get())  
 ge = GraphEncryption(key)  
 decrypted\_message = ge.decrypt(encrypted\_message, path)  
  
 output\_decrypted\_message.set(decrypted\_message)  
  
  
# GUI window  
root = tk.Tk()  
root.title("Graph-Based Encryption/Decryption")  
  
  
  
def toggle\_mode():  
 if mode.get() == "Encrypt":  
 frame\_encrypt.grid()  
 frame\_decrypt.grid\_remove()  
 else:  
 frame\_encrypt.grid\_remove()  
 frame\_decrypt.grid()  
  
  
mode = tk.StringVar(value="Encrypt")  
tk.Radiobutton(root, text="Encrypt", variable=mode, value="Encrypt", command=toggle\_mode).grid(row=0, column=0, padx=10,  
 pady=10)  
tk.Radiobutton(root, text="Decrypt", variable=mode, value="Decrypt", command=toggle\_mode).grid(row=0, column=1, padx=10,  
 pady=10)  
  
# Inputs  
tk.Label(root, text="Enter Key (Integer):").grid(row=1, column=0, padx=10, pady=10)  
input\_key = tk.Entry(root, width=50)  
input\_key.grid(row=1, column=1, padx=10, pady=10)  
  
frame\_encrypt = tk.Frame(root)  
frame\_encrypt.grid(row=2, column=0, columnspan=2)  
  
# Input for message to encrypt  
tk.Label(frame\_encrypt, text="Enter Message:").grid(row=0, column=0, padx=10, pady=10)  
input\_message = tk.Entry(frame\_encrypt, width=50)  
input\_message.grid(row=0, column=1, padx=10, pady=10)  
  
tk.Button(frame\_encrypt, text="Encrypt", command=encrypt\_message).grid(row=1, column=0, columnspan=2, pady=10)  
  
# Output for encryption  
tk.Label(frame\_encrypt, text="Encrypted Message:").grid(row=2, column=0, padx=10, pady=10)  
output\_encrypted\_message = tk.StringVar()  
tk.Entry(frame\_encrypt, textvariable=output\_encrypted\_message, state="readonly", width=50).grid(row=2, column=1,  
 padx=10, pady=10)  
  
tk.Label(frame\_encrypt, text="Encryption Path:").grid(row=3, column=0, padx=10, pady=10)  
output\_encryption\_path = tk.StringVar()  
tk.Entry(frame\_encrypt, textvariable=output\_encryption\_path, state="readonly", width=50).grid(row=3, column=1, padx=10,  
 pady=10)  
  
  
frame\_decrypt = tk.Frame(root)  
frame\_decrypt.grid(row=2, column=0, columnspan=2)  
frame\_decrypt.grid\_remove() # Hide the frame initially (encryption mode is the default)  
  
# Input for encrypted message  
tk.Label(frame\_decrypt, text="Enter Encrypted Message:").grid(row=0, column=0, padx=10, pady=10)  
input\_encrypted\_message = tk.Entry(frame\_decrypt, width=50)  
input\_encrypted\_message.grid(row=0, column=1, padx=10, pady=10)  
  
# Input for encryption path  
tk.Label(frame\_decrypt, text="Enter Encryption Path (comma-separated):").grid(row=1, column=0, padx=10, pady=10)  
input\_encryption\_path = tk.Entry(frame\_decrypt, width=50)  
input\_encryption\_path.grid(row=1, column=1, padx=10, pady=10)  
  
tk.Button(frame\_decrypt, text="Decrypt", command=decrypt\_message).grid(row=2, column=0, columnspan=2, pady=10)  
  
# Output  
tk.Label(root, text="Decrypted Message:").grid(row=3, column=0, padx=10, pady=10)  
output\_decrypted\_message = tk.StringVar()  
tk.Entry(root, textvariable=output\_decrypted\_message, state="readonly", width=50).grid(row=3, column=1, padx=10,  
 pady=10)  
  
root.mainloop()

**Code Efficiency**

Time Complexity: The time complexity of the encrypt method is O(n), where n is the length of the input message. For each character in the message, the algorithm performs a constant amount of work (finding connections and picking a random node).

The time complexity of the decrypt method is also O(n) for the same reasons, as it iterates over the recorded path.

Space Complexity: The space complexity for both encrypt and decrypt methods is O(n) because they use lists (encrypted\_message and decrypted\_message) proportional to the size of the input message.

Additionally, the adjacency matrix used for the graph has a space complexity of O(m²), where m is the number of unique characters (in this case, 27, for the letters and space).

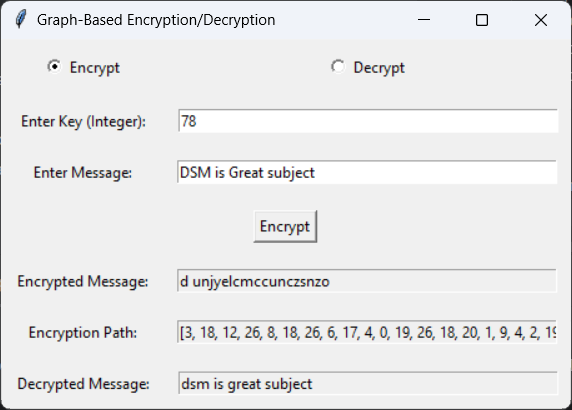
Efficiency Improvements: The random selection of connections can lead to different encrypted messages for the same input. If predictability is desired, a deterministic traversal method (like depth-first search) could be employed.

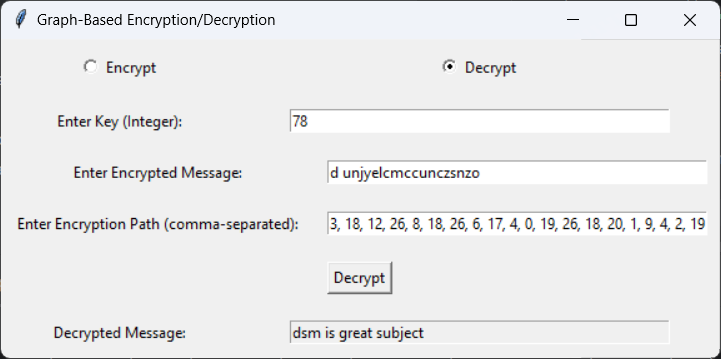
Instead of storing the entire encryption path, if only the resulting encrypted message is needed, one could potentially optimize space usage further by maintaining just the last node.

Test Cases and Results

|  |  |  |  |
| --- | --- | --- | --- |
| TC | Input | Expected Op | Actual Op |
| 1 | hello | vzgccd | vzgccd |
| 2 | world | fyoxil | fyoxil |
| 3 | test | igexf | igexf |
| 4 | abc | ilbm | ilbm |

Output:





**Challenges and Error Handling**

Difficulties Faced:

Understanding how to construct a random adjacency matrix that maintains the properties of a graph for encryption.

Implementing the GUI and ensuring smooth transitions between encryption and decryption modes.

Error Handling:

The code includes checks for empty messages and invalid encryption path formats, prompting the user with warning messages when necessary.

If the input path for decryption is not correctly formatted, it raises a warning.

**Conclusion:**

Outcome Summary:

The implementation of a graph-based encryption and decryption method was successfully achieved using a GUI for user interaction. The program encrypts and decrypts messages based on a randomly generated adjacency matrix, providing an innovative approach to encryption.

Key Takeaways:

Understanding the principles of graph traversal can significantly aid in creating secure encryption methods.

The importance of error handling in user interfaces is crucial for a good user experience.

**References:**

Books: "Introduction to Cryptography" by William Stallings

Online Resources:

[Numpy Documentation](https://numpy.org/doc/)

[Python Tkinter Documentation](https://docs.python.org/3/library/tk.html)