Aufgabe 15 “Vigenère Bruteforce Cracker”

Einleitung

Implementierung eines Brute-Force-Angriffs auf eine Vigenère-Verschlüsselung. Die Implementierung verwendet Multiprocessing, um die Entschlüsselungsaufgaben parallel auszuführen und die Berechnungszeit zu verkürzen.

Funktionsbeschreibung

Funktionen:

1. **brute_force_worker(key, cipher_text, known_plaintext_start)**
   - Diese Funktion wird von den Worker-Prozessen aufgerufen und führt die tatsächliche Entschlüsselung für einen bestimmten Schlüssel durch.
   - **Parameter**:
     - `key`: Der aktuell zu testende Schlüssel als Tuple von Buchstaben.
     - `cipher_text`: Der zu entschlüsselnde Text.
     - `known_plaintext_start`: Der bekannte Klartextanfang (“hello”).
   - **Funktionsweise**: Der Text wird mit dem aktuellen Schlüssel entschlüsselt, und falls der entschlüsselte Text mit dem bekannten Klartextanfang beginnt, wird der Schlüssel zurückgegeben und der entschlüsselte Text ausgegeben. Ansonsten wird zweimal `None` zurückgegeben.
2. **brute_force_vigenere(cipher_text, known_plaintext_start, max_key_length=5)**
   - Übernimmt die Steuerung des Brute-Force-Angriffs und koordiniert die parallele Entschlüsselung des verschlüsselten Textes.
   - **Parameter**:
     - `cipher_text`: Der verschlüsselte Text.
     - `known_plaintext_start`: Der bekannte Klartextanfang.
     - `max_key_length`: Die maximale Länge des zu testenden Schlüssels.
   - **Funktionsweise**:
     - Die Funktion testet alle Schlüssellängen bis hin zur maximalen Schlüssellänge `max_key_length`.
     - Für jede Schlüssellänge werden alle möglichen Buchstabenkombinationen generiert. Dazu wird die `itertools.product`-Funktion verwendet.
     - Die `partial`-Funktion von `functools` wird verwendet, um die `brute_force_worker`-Funktion partiell zu instanziieren, sodass `cipher_text` und `known_plaintext_start` bereits festgelegt sind. Die beiden Variablen sind konstant und müssen nicht bei jedem Worker-Aufruf neu übergeben werden.
     - Ein Pool von Prozessen (in Anzahl der verfügbaren CPU-Kerne) wird erstellt und die Entschlüsselungsaufgaben werden mithilfe der `imap_unordered`-Methode parallelisiert. `imap_unordered` gibt die Ergebnisse sofort zurück, sobald sie verfügbar sind und kann daher hier verwendet werden, da die Reihenfolge der Ergebnisse nicht wichtig ist.
     - Sobald der richtige Schlüssel gefunden wird, wird `pool.terminate()` aufgerufen, um alle laufenden Prozesse zu beenden.

Implementierte Optimierungen

- **Verwendung von `functools.partial`**: Die Verwendung von `partial` sorgt dafür, dass `cipher_text` und `known_plaintext_start` nur einmal festgelegt und nicht bei jedem Task-Aufruf an die Worker-Funktion übergeben werden müssen. Dies

spart Speicherplatz und verringert die Anzahl der zu übergebenden Argumente.

- **Multiprocessing**: Die Implementierung verwendet die `multiprocessing.Pool`-Bibliothek, um die Brute-Force-Operation parallel auszuführen. Jeder Prozess bearbeitet einen Teil der Schlüssel, was die gesamte Berechnungszeit zu verkürzen. Für die parallele Verarbeitung wird die `imap_unordered`-Methode verwendet. Diese gibt die Ergebnisse sofort zurück, sobald sie verfügbar sind und kann daher hier verwendet werden, da die Reihenfolge der Ergebnisse nicht wichtig ist. Somit kann auch frühzeitig gestoppt werden, sobald der richtige Schlüssel gefunden wird.
- **Verwendung von `itertools.product` und `itertools.chain`**: Die `itertools.product`-Funktion wird verwendet, um alle möglichen Buchstabenkombinationen für einen Schlüssel zu generieren. Die `itertools.chain`-Funktion wird verwendet, um die Ergebnisse der `product`-Funktion in einem einzigen Generator zu kombinieren. Dadurch wird Speiherplatz gespart und die einzelnen Task können über alle Schlüssellängen hinweg effizient verteilt werden.
- **Optimale Chunk-Größe**: Die Funktion berechnet eine optimale Chunk-Größe, sodass die einzelnen Task gleichmäßig auf die verfügbaren CPU-Kerne verteilt werden. Die Chunk-Größe wird so berechnet, dass sie entweder auf die Anzahl der CPU-Kerne abgestimmt ist oder eine Obergrenze von 30.000 Aufgaben pro Chunk hat.
- **Frühes Stoppen**: Sobald der richtige Schlüssel gefunden wird, beendet `pool.terminate()` alle laufenden Prozesse, um die Ressourcennutzung zu minimieren.

Beispielaufrufe

1. **Validierung der Verschlüsselungs- und Entschlüsselungsfunktionen**
   - Im ersten Schritt wird die Funktionalität der `vigenere_encrypt` und `vigenere_decrypt` Funktionen getestet.
   - Hierzu wird der Text “test” mit dem Schlüssel “key” verschlüsselt und anschließend wieder entschlüsselt:

     ```
     # Define plain text and print
     plain_text = 'test'
     key = 'key'

     print("Step1: Validate the cypther and decypher functions")
     print(f"Plain text: {plain_text}")

     # Encrypt the plain text
     cipher_text = vigenere_encrypt(plain_text, key)

     # Print the cipher text
     print(f"Cipher text: {cipher_text}")

     # Validate the functions
     decrypted_text = vigenere_decrypt(cipher_text, key)
     print(f"Decrypted text: {decrypted_text}")
     ```

   - Das Ergebnis wird überprüft, ob der entschlüsselte Text mit dem ursprünglichen Text übereinstimmt:

     ```
     assert decrypted_text == plain_text, "Decryption failed"
     ```

   - **Konsolenausgabe**:

     ```
     Step1: Validate the cypther and decypher functions
     ```

```
Plain text: test
Cipher text: diqd
Decrypted text: test
```

2. **Brute-Force-Angriff**

   - **Schritt 2**: Ein Brute-Force-Angriff mit einer maximalen Schlüssellänge von 5 wird ausgeführt, um den verschlüsselten Text “eqvpmtabpb” zu knacken:

```
cipher_text = "eqvpmtabpb"
known_plaintext_start = "hello"
print("\nStep2: Perform brute force attack (max key length = 5)")
print(f"Cipher text: {cipher_text}")
print(f"Known plaintext start: {known_plaintext_start}")

# Start timer
start_time = time.time()

# Perform brute force attack
key, decrypted_text = brute_force_vigenere(cipher_text, known_plaintext_start,
       max_key_length=5)

# End timer
end_time = time.time()
# Print results
if key and decrypted_text:
   print(f"Key found: {key}")
   print(f"Decrypted text: {decrypted_text}")
   print(f"Time taken: {end_time - start_time} seconds")

else:
   print(f"No valid key found. Time taken: {end_time - start_time} seconds")
```

   - **Konsolenausgabe**:

```
Step2: Perform brute force attack (max key length = 5)
Cipher text: eqvpmtabpb
Known plaintext start: hello
Total number of possible combinations: 12356630
Key found: xmkey
Decrypted text: helloworld
Time taken: 24.317224979400635 seconds
```

   - **Schritt 3**: Der gleiche Angriff wird mit einer maximalen Schlüssellänge von 6 durchgeführt, um die Dauer der Schlüsselgenerierung und Entschlüsselung mit einer längeren Schlüssellänge zu vergleichen:

```
print("\nStep3: Perform brute force attack (max key length = 6)")
print(f"Cipher text: {cipher_text}")
print(f"Known plaintext start: {known_plaintext_start}")

# Start timer
start_time = time.time()

# Perform brute force attack
key, decrypted_text = brute_force_vigenere(cipher_text, known_plaintext_start,
       max_key_length=6)

# End timer
end_time = time.time()
# Print results
if key and decrypted_text:
   print(f"Key found: {key}")
   print(f"Decrypted text: {decrypted_text}")
```

```
        print(f"Time taken: {end_time - start_time} seconds")

    else:
        print(f"No valid key found. Time taken: {end_time - start_time} seconds")
```

- **Konsolenausgabe**:

```
Step3: Perform brute force attack (max key length = 6)
Cipher text: eqvpmtabpb
Known plaintext start: hello
Total number of possible combinations: 321272406
Key found: xmkey
Decrypted text: helloworld
Time taken: 23.0162672996521 seconds
```