AI1_Actividad Integradora 1

Alumnos:

Jorge Emiliano Pomar Mendoza | A01709338
Eliuth Balderas Neri | A01703315

Análisis y diseño de algoritmos avanzados (Gpo 601)

Profesor: Ramona Fuentes Valdéz

Tecnológico de Monterrey, campus Querétaro

Febrero- Junio 2024

15 de abril del 2024

Índice

Parte 1

Descripción

El programa analiza si el contenido de los archivos mcode1.txt, mcode2.txt y mcode3.txt están contenidos en los archivos transmission1.txt y transmission2.txt y despliega un true o false si es que las secuencias de chars están contenidas o no. En caso de ser true, muestra *true*, seguido de exactamente un espacio, seguido de la posición en el archivo de transmisiónX.txt donde inicia el código de mcodeY.txt

main()

```
// busca códigos maliciosos
  std::cout << "\n////// Parte 1 //////" << std::endl;
  for (const auto &nombreArchivo : nombresTransmision) {
    std::string contenidoTransmision = leerArchivo(nombreArchivo);
    for (const auto &secuencia : secuencias) {
      auto start = std::chrono::steady_clock::now();
      int pos = buscaSecuencia(contenidoTransmision, secuencia);
      auto end = std::chrono::steady_clock::now();
      std::chrono::duration<double, std::milli> elapsed_seconds = end - start;
      if (pos != -1) {
        std::cout << "\n(true) Secuencia\n <<" << secuencia
                  << ">> encontrada en el archivo: \n"
                  << nombreArchivo << " en la posicion: \n"
                  << pos << std::endl;
      } else {
        std::cout << "\n(false) Secuencia no encontrada" << std::endl;
      }
      std::cout << "Tiempo de ejecución: " << elapsed_seconds.count() << "ms"
                << std::endl;
    }
  }
```

buscaSecuencia()

```
int buscaSecuencia(
    const std::string &contenido,
```

```
    const std::string &secuencia) { // Procesa el patron y busca en el archivo
  // usando algoritmo KMP O(m + n)
  int n = contenido.length();
  int m = secuencia.length();
  std::vector<int> lps(m, 0);
  construyeArregloLPS(secuencia, m, lps);

  int i = 0;
  int j = 0;

  while (i < n) {
    if (secuencia[j] == contenido[i]) {
      j++;
      i++;
    }

    if (j == m) {
      return i - j;
    } else if (i < n && secuencia[j] != contenido[i]) {
      if (j != 0) {
        j = lps[j - 1];
      } else {
        i = i + 1;
      }
    }
  }

  return -1;
}
```

Análisis de Complejidad

Para la búsqueda de secuencia, se tiene una complejidad computacional de O(m+n)

Parte 2

Descripción

El programa detecta si el código es malicioso, suponiendo que el código malicioso tiene siempre código "espejeado" (palíndromos de chars), busca este tipo de código en una transmisión. El

programa después busca si hay código "espejeado" dentro de los archivos de transmisión. (palíndromo a nivel chars, no meterse a nivel bits). El programa muestra en una sola línea dos enteros separados por un espacio correspondientes a la posición (iniciando en 1) en donde inicia y termina el código "espejeado" más largo (palíndromo) para cada archivo de transmisión. Puede asumirse que siempre se encontrará este tipo de código.

main()

```
// busca palíndromos
 std::cout << "\n////// Parte 2 //////" << std::endl;
 std::cout << "\nEncuentra palindromo más largo por Manacher ---> \n"
           << std::endl;
 for (const auto &nombreArchivo : nombresTransmision) {
   std::string contenidoTransmision = leerArchivo(nombreArchivo);
   auto start = std::chrono::steady_clock::now();
   int maxLen = encuentraPalindromoManacher(contenidoTransmision);
   auto end = std::chrono::steady_clock::now();
   std::chrono::duration<double, std::milli> elapsed_seconds = end - start;
   if (maxLen > 0) {
     std::cout << "\n(true) Palindromo de longitud: " << maxLen
               << " encontrado en el archivo: \n"
               << nombreArchivo << std::endl;
   } else {
     std::cout << "(false) Palindromo no encontrado" << std::endl;
   }
   std::cout << "Tiempo de ejecución: " << elapsed_seconds.count() << "ms"
             << std::endl;
 }

 std::cout << "\nEncuentra palindromo más largo por busqueda bruta ---> \n"
           << std::endl;
 for (const auto &nombreArchivo : nombresTransmision) {
   std::string contenidoTransmision = leerArchivo(nombreArchivo);
   auto start = std::chrono::steady_clock::now();
   int maxLen = encuentraPalindromoBruto(contenidoTransmision);
   auto end = std::chrono::steady_clock::now();
   std::chrono::duration<double, std::milli> elapsed_seconds = end - start;
   if (maxLen > 0) {
     std::cout << "\n(true) Palindromo de longitud: " << maxLen
               << " encontrado en el archivo: \n"
               << nombreArchivo << std::endl;
   } else {
     std::cout << "(false) Palindromo no encontrado" << std::endl;
   }
   std::cout << "Tiempo de ejecución: " << elapsed_seconds.count() << "ms"
```

```
              << std::endl;
  }
```

encuentraPalindromoManacher()

```
int encuentraPalindromoManacher(
    const std::string &contenido) { // Implementación de Manacher O(n) para
  // encontrar palíndromos
  std::string secuencia = "$#";
  // Agregamos caracteres especiales para manejar casos de longitud par e impar
  for (const auto &c : contenido) {
    secuencia += c;
    secuencia += "#";
  }
  secuencia += "@";
  int n = secuencia.length();
  std::vector<int> lps(n, 0);
  int c = 0, r = 0;

  // este ciclo calcula los valores de lps y actualiza c y r para cada i en la
secuencia de entrada
  for (int i = 1; i < n - 1; i++) {
    int mirr = 2 * c - i;
    if (i < r) {
      lps[i] = std::min(r - i, lps[mirr]);
    }
    while (secuencia[i + (1 + lps[i])] == secuencia[i - (1 + lps[i])]) {
      lps[i]++;
    }
    if (i + lps[i] > r) {
      c = i;
      r = i + lps[i];
    }
  }

  int maxLen = 0;
  int centerIndex = 0;

  for (int i = 1; i < n - 1; i++) {
    if (lps[i] > maxLen) {
      maxLen = lps[i];
      centerIndex = i;
```

```
    }
  }
  return maxLen;
}
// Referencia: https://cp-algorithms.com/string/manacher.html
```

encuentraPalindromoBruto()

```
int encuentraPalindromoBruto(
    const std::string &contenido) { // Implementacion de búsqueda bruta O(n^3)
  // para encontrar palindromos
  int n = contenido.length();
  int maxLen = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      bool esPalindromo = true;
      for (int k = i; k <= (i + j) / 2; k++) {
        if (contenido[k] != contenido[j - (k - i)]) {
          esPalindromo = false;
          break;
        }
      }
      if (esPalindromo) {
        maxLen = std::max(maxLen, j - i + 1);
      }
    }
  }
  return maxLen;
}
// basado en https://www.tamps.cinvestav.mx/~ertello/algorithms/sesion07.pdf
```

Análisis de Complejidad

El algoritmo de Manacher tiene complejidad de O(n), mientras que el algoritmo de Fuerza Bruta tiene una complejidad O(n^3) debido a que contiene 3 bucles anidados.

Parte 3

Descripción

El programa analiza qué tan similares son los archivos de transmisión, y muestra la posición inicial y la posición final (iniciando en 1) del primer archivo en donde se encuentra el substring más largo común entre ambos archivos de transmisión.

main()

```
// encuentra el substring mas largo comun entre dos archivos
  std::cout << "\n////// Parte 3 //////" << std::endl;
  for (const auto &nombreArchivo : nombresTransmision) {
    std::string contenidoTransmision = leerArchivo(nombreArchivo);
    auto start = std::chrono::steady_clock::now();
    int maxLen = encuentraSubMasLargo(contenidoTransmision);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double, std::milli> elapsed_seconds = end - start;
    if (maxLen > 0) {
      std::cout << "\n(true) Substring de longitud: " << maxLen
                << " encontrado entre los archivos: \n"
                << nombresTransmision[0] << " y " << nombresTransmision[1]
                << std::endl;
    } else {
      std::cout << "(false) Substring no encontrado" << std::endl;
    }
    std::cout << "Tiempo de ejecución: " << elapsed_seconds.count() << "ms"
              << std::endl;
  }
```

encuentraSubMasLargo()

```
int encuentraSubMasLargo(const std::string &contenido) {
  int n = contenido.length();
  int maxLen = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      if (contenido[i] == contenido[j]) {
        int len = 0;
        int k = i;
        while (k < n && contenido[k] == contenido[j]) {
          len++;
          k++;
          j++;
        }
```

```
        maxLen = std::max(maxLen, len);
      }
    }
  }
  return maxLen;
}
```

Análisis de Complejidad

Para la función encuentraSubMasLargo, se tiene una complejidad computacional de O(n^3) debido a que también contiene 3 bucles anidados.

Reflexión

Emiliano Pomar:

En muchas aplicaciones, sobre todo en la bioinformática, conviene que los algoritmos utilizados para ordenar y buscar sean lo más rápidos posibles para detectar posibles virus y variantes de enfermedades. Igualmente en la ciberseguridad, poder detectar este tipo de ataques filtrando entre muchas direcciones ip por ejemplo necesita de una gran eficiencia para manejar mucha información. Y creo que en esta primera entrega me dí cuenta que la eficiencia en memoria y el tiempo son muy valorados. Y por eso escogimos en el caso de los palíndromos el mejor algoritmo dedicado. Igualmente creo que si bien se implementaron bien y al final pudimos medir los milisegundos, siento que me costó mucho trabajo entender la forma en la que se eficientiza y sobre todo cambia la complejidad al cambiar de un algoritmo a otro. Pero es ahí que me sirvió mucho comparar usando los algoritmos extremos. El mejor y el peor, por así decirlo. Y esta comparación con más contraste me permitió ver la importancia de las estructuras que se escogen dentro del mismo algoritmo y la forma en la que se iteran los valores.

Eliuth Balderas Neri:
En todo contexto se debe analizar lo que se quiere, así como las prioridades, como por ejemplo, memoria, velocidad, eficiencia, etc. Esto debido a que en ocasiones algunos algoritmos pueden ser más rápidos pero ocupan mucha memoria, entonces se debe definir qué cosa se prioriza más. En este caso, no se tenía certeza de que algoritmo usar, por lo que utilizar diferentes algoritmos

definitivamente era un excelente acercamiento a la solución, en este caso se utilizó el algoritmo de fuerza bruta que es uno de los algoritmos más simples pero menos eficientes en este caso debido a que hay muchos archivos y elementos dentro de estos. Una vez que supimos cómo funcionaba, así como las respuestas que se buscaban, se pudo hacer una investigación más extensa en donde se definió que usar otro algoritmo como manacher podrías ser mucho más eficiente. Este algoritmo fue mucho más rápido que el de fuerza bruta por lo que se decidió que era la mejor solución. Sin embargo, quizá no se han tomado en cuenta todos los aspectos del contexto, los cuáles se espera que se puedan visualizar más adelante en el curso.

Referencias

Bultrón, J. (2017). Algoritmo KMP [Presentación]. Recuperado de https://prezi.com/p/9fxptyl1xpzz/algoritmo-kmp/

Bustos, B. (s.f.). Búsqueda de texto. Recuperado de https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/BusqTexto/

Universidad Mayor de San Andrés. (s.f.). Algoritmo de fuerza bruta. Recuperado de https://www.studocu.com/bo/document/universidad-mayor-de-san-andres/taller-de-programacion-lab-inf/algoritmo-de-fuerza-bruta/4599639