Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructura de Datos y Algoritmos

Informe Tarea 1

2 de junio de 2024
Lucas Fernández
20639112

Motivación

Este problema nos permite aprender diferentes habilidades que son de alta utilidad para desarrollos futuros. Por una parte, nos permite experimentar con estructuras y algoritmos de búsqueda de secuencias, tanto para secuencias lineales como no lineales, en donde tanto la forma de armar la estructura como buscar en ella impacta directamente en la complejidad de la resolución del problema. Por otra parte, nos permite experimentar con algoritmos de resoluciones de problemas, en donde probar todas las soluciones es una opción, pero con los algoritmos propuestos, backtracking, se logra realizarlo de una manera mucho más rápida. Estas dos técnicas se ven en la vida real, la búsqueda lineal y no lineal se puede encontrar aplicada en bioinformática, procesamiento de texto y compresión de datos, entre otros campos. El backtracking se puede observar en procesamiento de lenguaje natural, bioinformática y optimización combinatoria, entre otros campos.

Informe

0.1. Parte 1

Para resolver la primera parte se utilizó la estructura de datos conocida como Suffix Automaton. Esta estructura es eficiente para manejar problemas relacionados con los sufijos de cadenas, permitiendo encontrar las posiciones de una subcadena en la cadena original en tiempo lineal respecto al largo de la subcadena buscada.

Un Suffix Automaton de una cadena es un Autómata Finito Determinista (DFA) mínimo que contiene todas las subcadenas de la cadena. En otras palabras, esta estructura es un grafo acíclico orientado en el cual los vértices y los bordes son llamados estados y transiciones, respectivamente. Existe un estado inicial desde el cual se puede alcanzar a todos los otros estados. Cada transición está relacionada con un carácter de la cadena transformado en un número, por lo que cada transición generada desde un estado debe tener un carácter exclusivo. Finalmente, uno o más estados están marcados como estados terminales, lo que significa que si partimos desde el estado inicial y recorremos un substring hasta llegar a un estado final, ese substring está contenido en la cadena que representa el autómata. Dado que este tipo de autómata es minimalista en términos de número de vértices, se considera altamente eficiente.

La construcción de esta estructura se realiza recorriendo una vez la cadena, añadiendo caracteres transformados a números, y agregando transiciones y enlaces para mantener la estructura y asegurar una construcción eficiente. En este problema, dado que la cantidad de letras es pequeña (4), se puede mantener un array por cada estado donde cada índice representa una letra. Esto asegura la linealidad de búsqueda y construcción, sacrificando O(nk) memoria, donde n es la longitud de la cadena y k la cantidad de letras posibles (4).

Para realizar las búsquedas, se recorre el substring hasta no encontrar una transición, lo que indica que no es substring de la cadena, o hasta llegar a un estado final. En este caso, se imprime la primera ocurrencia de ese estado y luego las recurrencias, todas almacenadas en arrays, permitiendo acceso O(1) a esta información.

El algoritmo toma O(n) en construir el autómata y, en el peor de los casos, O(Q x M) en realizar todas las búsquedas, siendo así O(N) si N ¿Q x M y O(Q x M) si N ¡Q x M.

En cuanto a memoria, el suffix automaton tendrá un máximo de 2n - 1 estados y 3n - 4 transiciones. Por lo tanto, se debe considerar suficiente espacio para las matrices que representan todos los estados y transiciones. No se utiliza hashing en este algoritmo; lo más cercano es la transformación de letras a índices: 'A' -¿0, 'G' -¿1, 'T' -¿2 y 'C' -¿3.

Dada la complejidad de esta estructura y algoritmo, no se pudo abordar en su totalidad en esta explicación. Para mayor información, revisar las referencias teóricas y de código en la sección de bibliografía.

0.2. Parte 2

Para resolver el problema se utilizó el algoritmo de backtracking. Primero se guarda la distribución de la bodega, agregando dos filas y columnas para integrar los casos borde donde hay tres ceros en las primeras dos filas o columnas del paquete. Luego, se guardan los paquetes entregados y se inicia el algoritmo de backtracking.

Este algoritmo hace una asignación recursiva de los paquetes. En la asignación, se recorren todas las filas y columnas de la distribución, verificando si el paquete se puede colocar en la posición de la bodega. Si es posible, se coloca el paquete y se ingresa recursivamente al siguiente paquete a incluir, repitiendo el proceso. Si algún paquete no se puede colocar, se retrocede, se quita el paquete anterior y se prueban las posiciones restantes. Si se logran colocar todos los paquetes, se imprime la distribución final, eliminando las filas y columnas adicionales.

Posibles optimizaciones incluyen precomputar las posiciones posibles para cada paquete, reduciendo la cantidad de posiciones a revisar y, en consecuencia, las iteraciones.

El recorrido del árbol se realiza revisando todas las posiciones posibles para cada paquete, considerando la ubicación de otros paquetes. Si algún paquete no se puede colocar, se retrocede y se prueban nuevas posiciones, recorriendo todas las opciones posibles de manera recursiva.

El manejo de memoria se realiza utilizando arrays de punteros a la distribución y paquetes, permitiendo modificaciones a medida que se avanza en las recursiones. Toda la memoria reservada se libera al terminar el algoritmo.

0.3. Parte 3

Para esta parte se utilizó hashing. Primero, se inicializan todos los arrays necesarios, incluyendo los desplazamientos para el hashing, las posiciones de los nodos según el hashing y las copias de nodos con sus posiciones correspondientes, optimizando así el problema. Luego, se hashea el árbol entregado recorriendo cada nivel y guardando la posición y nivel de cada nodo con el hashing correspondiente. Esto permite encontrar rápidamente los nodos desde donde puede partir una consulta y comparar los niveles de la consulta con el hashing.

Se procesan las consultas hasheando por nivel, similar al árbol principal, y guardando en un array. Luego, se obtiene la posición y nivel de la primera ocurrencia del nodo 0 de la consulta y se compara recurrentemente con los niveles del árbol principal. Se desplaza el hashing de la consulta a la posición inicial correspondiente en el árbol principal y se ajusta el hashing del nivel del árbol principal al mismo largo que el nivel de la consulta. Si todas las comparaciones son exitosas, se retorna 1 e imprime la posición de la consulta.

Este proceso se repite para todas las posiciones posibles del inicio de la consulta, revisando solo las posiciones dentro del rango permitido. Para reducir colisiones, se usan números primos grandes y dos funciones de hashing con números distintos. Las posiciones posibles se revisan en orden, ya que se guardan de manera ordenada durante el hasheo inicial del árbol principal.

En cuanto a la complejidad, el hasheo inicial del árbol se realiza en O(N), ya que se recorre una vez la cadena. Las consultas toman O(M) para hashear y O(log(M) x L) para encontrar las posiciones factibles del subárbol, donde L es el número máximo de impresiones necesarias,

por lo que se desprecia en el calculo. Por lo tanto, el tiempo total es O(M x K) para encontrar los K patrones de tamaño M.

Para mayor información, se puede consultar la página sobre hasheo de strings en cp-algorithms referenciada en la bibliografía.

Conclusión

Concluyendo, el problema planteado nos entregó la oportunidad de aprender diferentes algoritmos y estructuras que son de alta utilidad para desarrollos futuros. Por un lado, nos permitió poner en práctica el uso de estructuras y algoritmos de búsqueda de secuencias, tanto lineales como no lineales, aprendiendo de esta forma como la construcción de la estructura y la eficiencia de las búsquedas impactan significativamente en la complejidad de la solución del problema. Por otro lado, nos permitió trabajar algoritmos de resolución de problemas, backtracking, el cual optimiza considerablemente la búsqueda de soluciones en comparación con el enfoque exhaustivo de probar todas las posibilidades.

Bibliografía

- McNanoL-ICPC-Team-Notebook/Strings/SuffixAutomatonFULL.cpp

  en mc-cari/McNanoL-ICPC-Team-Notebook. (s. f.). Recuperado el 2 de junio de 2024, de https://github.com/mc-cari/McNanoL-ICPC-Team-Notebook

- Suffix Automaton - Algoritmos para Programación Competitiva. (s. f.). Recuperado el 2 de junio de 2024, de https://cp-algorithms.com/string/suffix-automaton.html

- String Hashing - Algoritmos para Programación Competitiva. (s. f.). Recuperado el 2 de junio de 2024, de https://cp-algorithms.com/string/string-hashing.html