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

Informe Tarea 2

1 de junio de 2024

Manuel Espinoza Quintero
20642598

Motivación

La tarea busca aplicar estructuras de datos y algoritmos para resolver problemas complejos y practicar hashing y backtracking. Estos se detallan a continuación.

Parte 1: Búsqueda de Subcadenas en una Secuencia

Algoritmos Utilizados

Fuerza Bruta El algoritmo de fuerza bruta implementado en la función `brute_force_search` compara cada cadena de consulta con cada subcadena posible en el documento. Este enfoque tiene una complejidad temporal de $O(N \times M)$, lo que lo hace ineficiente para secuencias largas debido a la necesidad de realizar múltiples comparaciones para cada subcadena. En términos de memoria, este algoritmo utiliza una cantidad proporcional al tamaño de las cadenas de consulta y la secuencia, lo cual es mínimo pero suficiente para pequeñas entradas.

Boyer-Moore La función `boyer_moore_search` utiliza una heurística de carácter malo, que permite saltar posiciones en la secuencia basándose en la información de la última aparición de los caracteres. Esto mejora la eficiencia en ciertos casos. La complejidad puede ser sublineal en el mejor caso, especialmente cuando los caracteres no coinciden frecuentemente, pero en el peor caso sigue siendo $O(N \times M)$. Este algoritmo es eficiente cuando las cadenas de consulta son largas y hay muchas diferencias entre las subcadenas.

Rabin-Karp La función `rabin_karp_search` emplea rolling hashing para comparar los hashes de las subcadenas y las cadenas de consulta. Esto permite realizar comparaciones en tiempo promedio $O(N+Q)$, donde N es el tamaño de la secuencia y Q el número de consultas. El uso de un hash rodante optimiza la actualización del hash de subcadenas consecutivas, permitiendo comparar eficientemente grandes cantidades de datos. Sin embargo, la gestión de colisiones es crucial para mantener la eficiencia, lo que requiere una función de hash robusta.

Consideraciones de Memoria

Estos algoritmos utilizan memoria proporcional al tamaño de las cadenas de consulta M y la secuencia N. Rabin-Karp puede requerir memoria adicional para almacenar valores hash y manejar colisiones. Implementar una tabla de dispersión con una función de hash eficiente podría mejorar significativamente la gestión de memoria y tiempo.

Mejoras

Una mejora notable sería aplicar rolling hashing de manera más eficiente, ajustando los parámetros de hash para minimizar colisiones. También se podría implementar una tabla de dispersión para mejorar la eficiencia en la gestión de memoria y tiempo.

Parte 2: Organización de Paquetes en una Bodega

Estrategia Empleada

Backtracking La solución implementada en la función `place_packages` utiliza backtracking para probar todas las posibles posiciones de los paquetes en la bodega. Este enfoque recursivo coloca un paquete en una posición válida y retrocede si no se pueden colocar todos los paquetes, probando así todas las combinaciones posibles. Esta técnica es efectiva para problemas de búsqueda y optimización donde se necesita explorar todas las posibles configuraciones.

Optimización y Extensiones

- **Poda del Árbol de Decisiones**: Aplicar técnicas de poda para evitar explorar ramas innecesarias del árbol de decisiones puede mejorar significativamente la eficiencia. Esto se puede lograr eliminando configuraciones que no cumplen con los criterios necesarios temprano en el proceso.
- **Heurísticas**: Usar heurísticas para seleccionar primero las posiciones más prometedoras puede reducir el número de llamadas recursivas y mejorar el rendimiento general del algoritmo. Por ejemplo, priorizar las áreas más grandes y continuas de espacios disponibles en la bodega.

Análisis de Complejidad

La complejidad en el peor caso es exponencial, típico de los algoritmos de backtracking, ya que se exploran todas las posibles configuraciones. Sin embargo, en la práctica, la poda y las heurísticas pueden reducir significativamente el tiempo de ejecución al evitar exploraciones innecesarias.

Consideraciones de Memoria

El algoritmo requiere memoria para almacenar la matriz de la bodega y los paquetes. Utilizar estructuras de datos más compactas o representaciones eficientes podría reducir el consumo de memoria. Por ejemplo, representar la matriz de la bodega y los paquetes como bitmaps podría optimizar el uso de memoria.

Parte 3: Búsqueda de Subárboles en un Árbol Binario Perfecto

Técnica Empleada

Hashing de Subárboles La función `matches_at` utiliza hashing para comparar los subárboles de manera eficiente. Calcula un hash para cada subárbol en el árbol principal y lo compara con el hash del subárbol padre. Esta técnica permite verificar la igualdad de subárboles en tiempo constante, haciendo el proceso de búsqueda mucho más eficiente que una comparación directa.

Para lograr recorrer el árbol de posibles opciones, se sigue una estrategia recursiva:

1. **Decisión Inicial**: Comenzamos con el primer paquete y probamos todas las posiciones posibles en la bodega. Si encontramos una posición válida, colocamos el paquete en esa posición.
2. **Recursión**: Luego, hacemos una llamada recursiva para intentar colocar el siguiente paquete. Si logramos colocar todos los paquetes, hemos encontrado una solución válida.
3. **Backtracking**: Si no podemos colocar el siguiente paquete, removemos el paquete actual (deshacemos la decisión) y probamos la siguiente posición posible para el paquete actual. Este proceso continúa hasta que se han probado todas las posiciones posibles para todos los paquetes.

Optimización y Extensiones

- **Función de Hash Robusta**: Utilizar una función de hash robusta minimiza colisiones y mejora la precisión de las comparaciones. Una buena función de hash debería distribuir uniformemente las entradas posibles para reducir la probabilidad de colisiones.

- **Precomputación de Valores**: Precomputar ciertos valores de hash puede acelerar el proceso de comparación y reducir el tiempo de ejecución general. Esto es especialmente útil cuando se realizan múltiples búsquedas sobre el mismo conjunto de datos.

Análisis de Complejidad

La complejidad del algoritmo es $O(M \times K)$, donde M es el tamaño del patrón y K es el número de patrones. Esta complejidad es eficiente para este tipo de problemas, comparado con una búsqueda directa que tendría una complejidad mucho mayor.

Consideraciones de Memoria

El uso de hashing requiere memoria adicional para almacenar los valores hash de los subárboles. Sin embargo, la eficiencia ganada en tiempo de ejecución justifica este uso adicional de memoria. Implementar una tabla de dispersión eficiente puede ayudar a reducir el uso de memoria y mejorar la velocidad de acceso a los datos hash.

Conclusión

La implementación de algoritmos eficientes y técnicas adecuadas de hashing y backtracking permite resolver problemas complejos de manera efectiva. La selección del algoritmo adecuado según el contexto del problema es crucial para optimizar tanto el tiempo de ejecución como el uso de memoria. Las técnicas de poda y heurísticas en backtracking, así como el uso de funciones de hash robustas en algoritmos de búsqueda, son ejemplos clave de cómo mejorar la eficiencia de los algoritmos. Además, ajustar los parámetros y optimizar las estructuras de datos utilizadas puede llevar a mejoras significativas en el rendimiento, que incluso se vio reflejado en los tiempos de los test de la tarea.

En el primer problema, la comparación de diferentes métodos de búsqueda de subcadenas mostró la importancia de elegir el algoritmo correcto según las características del input. La fuerza bruta es simple pero ineficiente para grandes secuencias, mientras que Boyer-Moore y Rabin-Karp ofrecen mejoras significativas en ciertos escenarios, las cuales se ajustaron en el código.

El segundo problema destacó la efectividad del backtracking para resolver problemas de colocación complejos, con el potencial de mejorar el rendimiento mediante técnicas de poda y heurísticas. Hay que tener en consideración, la organización eficiente de la memoria, ya que es clave para manejar grandes matrices y múltiples objetos.

El tercer problema demostró cómo el hashing puede transformar la comparación de subestructuras complejas en una tarea manejable y eficiente. El uso de funciones de hash robustas y la precomputación de valores son estrategias efectivas para optimizar el rendimiento en la búsqueda de patrones en estructuras de datos grandes.

Referencias

[1] "Common string-matching algorithms: theoretical and practical performance", *Daan Kolthof*, Septiembre 2019. [Online]. Disponible: https://daankolthof.com/2019/09/01/common-string-matching-algorithms-theoretical-and-practical-performance/.