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

Informe Tarea 3

20 de junio de 2024
Lucas Fernández
20639112

Motivación

Este problema nos permite aprender diferentes habilidades que son de alta utilidad para desarrollos futuros. Nos da la oportunidad de experimentar con algoritmos codiciosos, programación dinámica, búsqueda en profundidad y estructuras de grafos, ampliando así nuestro conocimiento y equipándonos con herramientas de gran utilidad frente a la resolución de problemas complejos, cuando hay que mantener la eficiencia en mente. Estas técnicas son de gran importancia para la optimización de recursos, inteligencia artificial, seguridad informática y el desarrollo de software eficiente, dentro de las aplicaciones de estas técnicas en la vida real se encuentran la planificación de rutas, bioinformática, los circuitos electrónicos y los sistemas recomendadores.

Informe

0.1. Parte 1

Para resolver este problema se utiliza un algoritmo codicioso. Primero, se ordenan los planetas que se entregan con un insertion sort, luego de ordenarlos, se recorre la lista de planetas ordenados. Al recorrer la lista se revisa la distancia entre el primer planeta cubierto, más a la izquierda, hasta el planeta que se revisa actualmente, si la distancia entre estos dos planetas es mayor a dos veces el rango entregado se le suma 1 a la cantidad de naves necesarias y se guarda el índice del planeta que se estaba chequeando, ahora este es el primer planeta cubierto por la nueva nave. De esta forma se consigue minimizar la cantidad de naves, ya que se maximiza el rango cubierto por cada nave revisando la distancia entre planetas consecutivos.

La complejidad de este algoritmo es, en el peor caso, de $O(n^2)$, esto debido al algoritmo insertion sort. Luego de ordenar los planetas, el algoritmo tiene complejidad igual a $O(n)$. Considerando estas dos complejidades, el algoritmo completo tiene una complejidad de tiempo igual a $O(n^2)$. La complejidad de memoria es de $O(n)$, dado que solo se utiliza un array y el sort se realiza in place, por lo que no requiere memoria extra.

0.2. Parte 2

Para resolver este problema se utilizó programación dinámica. Primero, se inicializa el array que contiene todas las soluciones a los subproblemas, en esta, cada índice representa la cantidad de masa que se quiere obtener y el valor guardado en ese índice es cuantos basureros se necesitan, para este problema se les asignó un valor de -1, ya que nunca se van a utilizar -1 basureros. Luego de inicializada, se le asigna el valor de 0 al índice 0 del array, representando que se necesitan 0 basureros para obtener la masa 0. Con estas asignaciones realizadas, se comienzan las iteraciones hasta la cantidad de masa M que se quiere obtener, por cada masa que se itera se revisan todos los basureros disponibles, por cada uno de estos se revisan 4 condiciones:

1. Que la cantidad de masa que aporta el basurero sea menor que la cantidad de masa que se quiere obtener
2. Que exista una solución al valor de masa que se quiere cubrir menos la masa que el basurero que se está revisando aporta, siendo esta la parte en la cual se ocupan los subproblemas para resolver el problema
3. Que no exista una solución para la masa que se está tratando de cubrir o que la nueva solución sea mejor que la asignada previamente

Si estas tres condiciones se cumplen, se asigna al índice de masa que se está intentando cubrir el valor de la solución de la masa que se quiere cubrir menos la masa que el basurero que se está revisando aporta sumándole uno. De esta forma se utilizan los subproblemas para resolver el problema, una vez terminadas todas las iteraciones se retorna la solución a la masa M, que puede ser -1 si no se encontró solución o la solución óptima en el caso de ser encontrada.

La complejidad de este algoritmo es de $O(M \cdot N)$, siendo M la cantidad de masa que se quiere cubrir y N la cantidad de basureros disponibles. La complejidad de memoria es de $O(M + N)$, dado que solo se utilizan dos arrays, uno para las soluciones y otro para los basureros.

0.3. Parte 3

Para resolver este problema se utiliza DFS y las estructuras necesarias para representar un grafo y realizar DFS sobre él. Primero, se inicializan dos arrays, una que guarda si el nodo fue visitado y otra que guarda la cantidad de aristas que tiene el nodo, ambas parten con ceros y el índice indica que nodo es. Luego, a medida que se van recibiendo las aristas, se va aumentando la cantidad de aristas en el array mencionado previamente y realocando la memoria de un array de arrays que guarda el nodo con el cual el índice tiene una arista.

Luego de guardar todas las aristas en el tercer array mencionado, se comienza a contar los grupos no conectados. Para ello se itera desde 0 hasta la cantidad de nodos que existen, en cada iteración se revisa si el nodo que se está iterando ya fue visitado, en el caso de no haber sido visitado se aumenta la cuenta de grupos y se realiza DFS desde este nodo.

Para el DFS, se instancia un array que se utilizara como stack y un entero que se utiliza para contar la cantidad de elementos en el stack y para conseguir el último elemento que se encuentra en este. Luego, mientras todavia quede un elemento en el stack, se itera sobre todas las aristas que tiene el último elemento del stack. En estas iteraciones se revisa cada nodo conectado por una arista, en el caso de que un nodo revisado no haya sido visitado previamente se marca como visitado y se agrega al stack, aumentando también el entero que lleva la cuenta de elementos. Una vez que no quedan elementos en el stack se sale del algoritmo DFS y se continúa con la iteración descrita en el parrafo anterior.

Al terminar todas las iteraciones se logra conseguir la cantidad de grupos no conectados de nodos.

La complejidad de este algoritmo es, en el peor caso, de $O(N \cdot E)$, siendo N la cantidad de nodos y E la cantidad de aristas presentes. La complejidad de memoria es de $O(N + E)$, dado que las arrays utilizadas depende de la cantidad de nodos y aristas.

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 algoritmos codiciosos para tomar decisiones óptimas y, luego, lo llevamos un paso más allá con programación dinámica, encontrando así soluciones que codicioso puede no encontrar. Esto nos enseñó cómo la elección de estructuras de datos y algoritmos impacta directamente en la correctitud de nuestras soluciones. Por otro lado, DFS, nos permitió explorar conexiones y relaciones en estructuras de grafos. En resumen, este problema ha fortalecido nuestra capacidad para resolver problemas de manera efectiva y sistemática, preparándonos para enfrentar desafíos más complejos.

Bibliografía

- Finding Connected Components - Algorithms for Competitive Programming. (n.d.). Retrieved June 20, 2024,

  from https://cp-algorithms.com/graph/search-for-connected-components.html