GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMOS ÁVIDOS

1. Introducción

En ciencias de la computación, un algoritmo voraz (también conocido como goloso, ávido, devorador o greedy) es una estrategia de búsqueda por la cual se sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Este esquema algorítmico es el que menos dificultades plantea a la hora de diseñar y comprobar su funcionamiento. Normalmente se aplica a los problemas de optimización.

2. Conocimientos previos

2.1. Heurística

En computación, dos objetivos fundamentales son encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una heurística es un algoritmo que abandona uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando no existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.

2.2. Programación Dinámica

La Programación Dinámica la cual es una técnica que combina la corrección de la búsqueda completa y la eficiencia de los algoritmos golosos.

3. Desarrollo

Un algoritmo greedy es cualquier algoritmo que sigue la heurística de resolución de problemas de hacer la elección localmente óptima en cada etapa. En muchos problemas, una estrategia greedy no produce una solución óptima, pero una heurística codiciosa puede generar soluciones óptimas locales que se aproximan a una solución óptima global en un tiempo razonable.

Greedy es un paradigma algorítmico que construye una solución pieza por pieza, eligiendo siempre la siguiente pieza que ofrece el beneficio más obvio e inmediato. Por lo tanto, los problemas en los que elegir lo óptimo localmente también conduce a una solución global son los que mejor se adaptan a Greedy.

Los algoritmos greedy producen buenas soluciones en algunos problemas matemáticos, pero no en otros. La mayoría de los problemas para los que funcionan tendrán dos propiedades:

1. **Propiedad de elección codiciosa:** Podemos tomar la decisión que nos parezca mejor en ese momento y luego resolver los subproblemas que surjan más adelante. La elección realizada por un algoritmo codicioso puede depender de las elecciones realizadas hasta el momento, pero no de las elecciones futuras o de todas las soluciones al subproblema. De manera iterativa, toma una decisión codiciosa tras otra, reduciendo cada problema dado a uno más

pequeño. En otras palabras, un algoritmo codicioso nunca reconsidera sus elecciones. Esta es la principal diferencia con la programación dinámica, que es exhaustiva y garantiza encontrar la solución. Después de cada etapa, la programación dinámica toma decisiones basadas en todas las decisiones tomadas en la etapa anterior y puede reconsiderar la ruta algorítmica de la etapa anterior hacia la solución.

2. **Subestructura óptima:** Un problema exhibe una subestructura óptima si una solución óptima al problema contiene soluciones óptimas a los subproblemas.

3.1. Tipos

Los algoritmos greedy se pueden caracterizar como miope y también como no recuperables. Son ideales solo para problemas que tienen una subestructura óptima. A pesar de esto, para muchos problemas simples, los algoritmos más adecuados son los greedy. Sin embargo, es importante tener en cuenta que el algoritmo voraz se puede usar como un algoritmo de selección para priorizar opciones dentro de una búsqueda o un algoritmo de ramificación y enlace. Hay algunas variaciones del algoritmo greedy que nos permite clasificarlos como:

- Algoritmos greedys puros
- Algoritmos greedys ortogonales
- Algoritmos greedys relajados

Como ayuda para identificar si un problema es susceptible de ser resuelto por un algoritmo greedy vamos a definir una serie de elementos que han de estar presentes en el problema:

- Un conjunto de **candidatos**, que corresponden a las **n** entradas del problema.
- Una **función de selección** que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una función que compruebe si un cierto subconjunto de candidatos es **prometedor**. Entendemos por prometedor que sea posible seguir añadiendo candidatos y encontrar una solución.
- Una **función objetivo** que determine el valor de la solución hallada. Es la función que queremos maximizar o minimizar.
- Una función que compruebe si un subconjunto de estas entradas es solución al problema, sea óptima o no.

4. Implementación

Con estos elementos ya identificados que deben estar presentes en un problema para que el mismo sea susceptible a ser resuelto por un algoritmo greedy, podemos resumir el funcionamiento de los algoritmos greedy en los siguientes puntos:

1. Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.

2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.
3. Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadilo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.
4. Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. A la vista de todo esto no resulta difícil plantear un esquema general para este tipo de algoritmos:

```
PROCEDURE AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;
   VAR x:ELEMENTO; solucion:CONJUNTO; encontrada:BOOLEAN;
BEGIN
   encontrada:=FALSE;
   crear(solucion);
   WHILE NOT EsVacio(entrada) AND (NOT encontrada) DO
      x:=SeleccionarCandidato(entrada);
      IF EsPrometedor(x,solucion) THEN
         Incluir(x,solucion);
         IF EsSolucion(solucion) THEN
            encontrada:=TRUE
         END;
      END
   END;
   RETURN solucion;
END AlgoritmoAvido;
```

De este esquema se desprende que los algoritmos ávidos son muy fáciles de implementar y producen soluciones muy eficientes. Entonces cabe preguntarse ¿por qué no utilizarlos siempre? En primer lugar, porque no todos los problemas admiten esta estrategia de solución. De hecho, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global, como mostraremos en varios ejemplos de este capítulo. La estrategia de los algoritmos ávidos consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.

Desgraciadamente, y como en la vida misma, pocos hechos hay para los que podamos afirmar sin miedo a equivocarnos que lo que parece bueno para hoy siempre es bueno para el futuro. Y aquí radica la dificultad de estos algoritmos. Encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión. Por ello, una parte muy importante de este tipo de algoritmos es la demostración formal de que la función de selección escogida consigue encontrar óptimos globales para cualquier entrada del algoritmo. No basta con

diseñar un procedimiento ávido, que seguro que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema.

5. Aplicaciones

Los algoritmos greedy normalmente (pero no siempre) no logran encontrar la solución globalmente óptima porque, por lo general, no operan de manera exhaustiva en todos los datos. Pueden comprometerse con ciertas opciones demasiado pronto, lo que les impide encontrar la mejor solución general más adelante. Por ejemplo, todos los algoritmos de coloración greedy conocidos para el problema de coloración de grafos y todos los demás problemas NP-completos no encuentran soluciones óptimas de manera consistente. Sin embargo, son útiles porque son rápidos de pensar y, a menudo, dan buenas aproximaciones al óptimo.

Si se puede demostrar que un algoritmo greedy produce el óptimo global para una clase de problema dada, generalmente se convierte en el método de elección porque es más rápido que otros métodos de optimización como la programación dinámica. Ejemplos de tales algoritmos greedy son el algoritmo de Kruskal y el algoritmo de Prim para encontrar árboles de expansión mínimos y el algoritmo para encontrar árboles de Huffman óptimos.

Los algoritmos greedy también aparecen en el enrutamiento de la red. Mediante el enrutamiento codicioso, se reenvía un mensaje al nodo vecino que está "más cerca"del destino. La noción de ubicación de un nodo (y, por lo tanto, cercanía) puede determinarse por su ubicación física, como en el enrutamiento geográfico utilizado por redes ad hoc. La ubicación también puede ser una construcción completamente artificial como en el enrutamiento de mundo pequeño y la tabla hash distribuida.

6. Complejidad

Por lo general los algoritmos greedy son rápido y eficiente con una complejidad de temporal que puede estar entre O($n \log n$) u O(n) pero no significa que algunos casos tenga una complejidad mayor.

7. Ejercicios

En la programación competitiva es común encontrar ejercicios o problemas que la solución radica en aplicar este enfoque algorítmico, por lo cual es posible decir que existe un grupo de situaciones que se conoce que su solución parte de este enfoque. Por lo cual nos vamos a encontrar:

- Algoritmos greedy estándar.
- Algoritmos greedy en arreglos.
- Algoritmos greedy en grafos.
- Algoritmos greedy aproximado para NP completo

- Algoritmos greedy para casos especiales de la programación dinámica

A continuación una lista de ejercicios que se pueden resolver utilizando este enfoque algorítmico:

- DMOJ - Palíndromo
- DMOJ - Múltiplo de 2, 3 y 5
- DMOJ - Arreglos Injustos
- DMOJ - Aislador