GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: BÚSQUEDA COMPLETA

1. Introducción

La búsqueda completa o exhaustiva es una idea o técnica general que se puede aplicar para resolver cualquier problema algorítmico. Sobre dicha técnica sus ventajas y desventajas abordaremos en la siguiente guía.

2. Conocimientos previos

2.1. Fuerza Bruta

Los algoritmos de Fuerza Bruta son capaces de encontrar la solución a cualquier problema por complicado que sea. Su fundamento es muy simple, probar todas las posibles combinaciones, recorrer todos los caminos hasta dar con la situación que es igual que la solución. No le importa iniciar caminos malos o muy malos, al llegar a su final y ver que su destino no es la solución, se iniciará otro camino en busca del que conduzca a ella.

2.2. Retroceso (*Backtracking*)

Backtracking (o vuelta atrás) es una técnica algorítmica para encontrar soluciones a problemas que tienen una solución completa, en los que el orden de los elementos no importa, y en los que existen una serie de variables, a cada una de las cuales, debemos asignarle un valor teniendo en cuenta unas restricciones dadas. O lo que es lo mismo, es una estrategia algorítmica que busca todas las posibles soluciones dado un conjunto de variables inicial para encontrar el resultado definido por el problema.

2.3. Problema NP-difícil/completo (*NP-hard/complete*)

En teoría de la complejidad computacional, la clase de complejidad NP-hard (o NP-complejo, o NP-difícil) es el conjunto de los problemas de decisión que contiene los problemas H tales que todo problema L en NP puede ser transformado polinomialmente en H. Esta clase puede ser descrita como aquella que contiene a los problemas de decisión que son como mínimo tan difíciles como un problema de NP. Esta afirmación se justifica porque si podemos encontrar un algoritmo A que resuelve uno de los problemas H de NP-hard en tiempo polinómico, entonces es posible construir un algoritmo que trabaje en tiempo polinómico para cualquier problema de NP ejecutando primero la reducción de este problema en H y luego ejecutando el algoritmo A.

3. Desarrollo

La búsqueda completa es un método general que se puede utilizar para resolver casi cualquier problema de algoritmo. La idea es generar todas las posibles soluciones al problema utilizando la fuerza bruta, y luego seleccionar la mejor solución o contar el número de soluciones, dependiendo del problema.

La técnica de búsqueda completa, también conocida como fuerza bruta o retroceso (*backtracking*) recursivo , es un método para resolver un problema atravesando todo (o parte del) espacio de búsqueda para obtener la solución requerida. Durante la búsqueda, se nos permite podar (es decir, elegir no explorar) partes del espacio de búsqueda si hemos determinado que estas partes no tienen posibilidad de contener la solución requerida. De esta manera, la búsqueda completa debe devolver la respuesta mejor/óptima (si existe) al finalizar.

En ICPC, la búsqueda completa debe ser la primera solución considerada, ya que generalmente es fácil encontrar una solución de este tipo y codificarla/depurarla. Recuerda el principio *KISS*: Mantenlo breve y simple (*Keep It Short and Simple*). Una solución de búsqueda completa libre de errores nunca debería recibir una respuesta incorrecta (WA) respuesta en concursos de programación ya que explora todo el espacio de búsqueda que puede contener la respuesta. Sin embargo, muchos problemas de programación tienen soluciones mejores que la Búsqueda completa. Por lo tanto, una solución de búsqueda completa puede recibir un veredicto de límite de tiempo excedido (TLE). Con un análisis adecuado, puede determinar el resultado probable (TLE vs AC) antes de intentar codificar nada es un buen punto de partida. Si una búsqueda completa es fácil de implementar y es probable que no supere el límite de tiempo, continúe e implemente una. Esto le dará más tiempo (concurso) para trabajar en problemas más difíciles en los que la búsqueda completa será demasiado lenta.

En IOI, normalmente necesitará mejores técnicas de resolución de problemas, ya que las soluciones de búsqueda completa normalmente solo se recompensan con una fracción muy pequeña de la puntuación total en el esquema de puntuación de las subtareas. Sin embargo, la búsqueda completa debe usarse cuando no pueda encontrar una mejor solución; al menos le permitirá obtener puntuación.

A veces, ejecutar la búsqueda completa en instancias pequeñas de un problema desafiante puede ayudarnos a comprender su estructura a través de patrones en la salida (es posible visualizar el patrón para algunos problemas) que pueden ser explotados para diseñar un algoritmo más rápido. Ejemplo de estos son algunos problemas combinatorios que se pueden resolver de esta manera. Por tanto, la búsqueda completa también puede actuar como un verificador para instancias pequeñas, proporcionando una verificación adicional para el algoritmo más rápido pero no trivial que desarrolle.

Se puede tener la impresión de que la búsqueda completa solo funciona para *problemas fáciles* y, por lo general, no es la solución prevista para *problemas más difíciles*. Esto no es enteramente verdad. Existen problemas difíciles que solo se pueden resolver con algoritmos creativos de búsqueda completa.

Finalmente, se pueden usar algunas reglas empíricas a continuación para ayudar a identificar problemas que se pueden resolver. con búsqueda completa. Un problema es posiblemente un problema de búsqueda completa si el problema:

- Pide imprimir todas las respuestas y el espacio de solución puede ser tan grande como el espacio de búsqueda.
- Tiene un espacio de búsqueda pequeño (el total de operaciones en el peor de los casos es

<100M)

- Tiene una restricción de tiempo sospechosamente grande y tiene muchos potenciales de poda (temprana)
- Puede calcularse previamente
- Es un problema NP-difícil/completo conocido sin ninguna propiedad especial.

3.1. Consejos para implementar una búsqueda completa

La mayor apuesta al escribir una solución de búsqueda completa es si podrá o no pasar el límite de tiempo. Si el límite de tiempo es de 1 segundo (los jueces en línea no suelen utilizar grandes límites de tiempo para una evaluación eficiente) y su programa actualmente se ejecuta en más de 1 segundo en varios (puede haber más de uno) casos de prueba con el tamaño de entrada más grande como se especifica en la descripción del problema, pero aún se considera que su código es TLE, es posible que desee modificar el código crítico en su programa en lugar de volver a resolver el problema con un algoritmo más rápido que puede no ser fácil de diseñar o puede no existir.

3.1.1. Filtrar vs generar

Algoritmos que examinan muchas (si no todas) posibles soluciones y eligen las que son correctos (o eliminar los incorrectos) se denominan *filtros*. Por lo general, los algoritmos de *filtros* se escriben iterativamente.

Los algoritmos que construyen gradualmente las soluciones e inmediatamente eliminan las soluciones parciales inválidas se denominan *generadores*. Por lo general, los algoritmos *generadores* son más fáciles de implementar cuando escrito recursivamente ya que nos da mayor flexibilidad para podar el espacio de búsqueda.

En general, los filtros son más fáciles de codificar pero se ejecutan más lentamente, dado que suele ser mucho más difícil eliminar una mayor parte del espacio de búsqueda de forma iterativa. Haga los cálculos (análisis de complejidad) para ver si un filtro es lo suficientemente bueno o si necesita crear un generador.

3.1.2. Eliminación temprana del espacio de búsqueda no viable

Al generar soluciones utilizando el retroceso (*backtracking*) recursivo, podemos encontrar una solución parcial que nunca conducirá a una solución completa. Podemos podar la búsqueda allí y explorar otras partes del espacio de búsqueda. Continuar desde cualquiera de estas soluciones parciales no viables nunca va a conducir a una solución válida. Por lo tanto, podemos podar estas soluciones parciales aquí y concentrarnos en las otras posiciones válidas, reduciendo así el tiempo de ejecución general. Como regla general, cuanto antes pueda podar el espacio de búsqueda, mejor.

3.1.3. Utilizar simetrías

¡Algunos problemas tienen simetrías y deberíamos intentar explotar las simetrías para reducir el tiempo de ejecución!. Sin embargo, debemos señalar que es cierto que, a veces, considerar las simetrías puede complicar el código. En la programación competitiva, esta no suele ser la mejor manera (queremos un código más corto para minimizar los errores). Si la ganancia obtenida al tratar con simetría es no es significativo para resolver el problema, simplemente ignore este consejo.

3.1.4. Precálculo

A veces es útil generar tablas u otras estructuras de datos que aceleren la búsqueda de un resultado antes de la ejecución del programa en sí. Esto se llama Precálculo, en el que uno intercambia memoria/espacio por tiempo. Sin embargo, esta técnica rara vez se puede utilizar para problemas de concursos de programación recientes.

Aunque este consejo no se puede utilizar para la mayoría de los problemas de búsqueda completa, puede encontrar una lista de algunos ejercicios de programación donde se puede utilizar este consejo.

3.1.5. Intenta resolver el problema al revés

Algunos problemas de concurso parecen mucho más fáciles cuando se resuelven *hacia atrás* (desde una perspectiva menos ángulo obvio) que cuando se resuelven usando un ataque frontal (desde el ángulo más obvio como se describe en la descripción del problema). Esté preparado para intentar enfoques no convencionales a los problemas.

3.1.6. Compresión de datos

La restricción de entrada en algunos problemas creativos en los que la solución esperada por el autor del problema es la búsqueda completa puede estar *disfrazada* para parecer demasiado grande para que una solución normal de búsqueda completa funcione dentro del límite de tiempo. Pero luego de una inspección más cuidadosa, algunos comentarios, generalmente sutiles, en la descripción del problema en realidad reduce el espacio de búsqueda (significativamente) lo que luego hace factible una solución de búsqueda completa.

3.1.7. Optimización de su código

Existen muchas técnicas que puede utilizar para optimizar su código. Entendiendo la computadora hardware y cómo está organizado, especialmente el comportamiento de E/S, memoria y caché, puede ayudar diseñas mejor código.

1. Use C++ en lugar de Java (más lento que C++) o Python (más lento que Java). Un algoritmo implementado usando C++ generalmente se ejecuta más rápido que el implementado en Java o Python en muchos jueces en línea. Algunos concursos de programación, pero no todos, dan a los usuarios de Java/Python más tiempo para dar cuenta de la diferencia en el rendimiento (pero esto nunca es 100 % justo).

2. La manipulación de bits en los tipos de datos enteros incorporados (hasta el entero de 64 bits) es (mucho) más eficiente que la manipulación de índices en una matriz de valores booleanos. Si necesitamos más de 64 bits, use el conjunto de bits STL de C++ en lugar de vector<bool>.
3. Para usuarios de C/C++, use el estilo C más rápido scanf/printf en lugar de cin/cout (o al menos configure ios::sync con stdio(false); cin.tie(NULL); aunque todavía más lento que scanf/printf).
4. Para usuarios de Java, use las clases *BufferedReader/BufferedWriter* más rápidas para la lectura e impresión de datos
5. Acceda a una matriz 2D en forma de fila principal (fila por fila) en lugar de columna principal forma como las matrices multidimensionales se almacenan en un orden de fila principal en la memoria. Este aumentará la probabilidad de acierto de caché.
6. Utilice estructuras/tipos de datos de nivel inferior en todo momento si no necesita la funcionalidad adicional en los de nivel superior (o más grandes). Por ejemplo, use una matriz con un tamaño ligeramente mayor que el tamaño máximo de entrada en lugar de usar vectores de tamaño variable.
7. Declare la mayoría de las estructuras de datos (especialmente las voluminosas, por ejemplo, matrices grandes) una vez colocándolas en el ámbito global. Asigne suficiente memoria para manejar la entrada más grande del problema. De esta manera, no tenemos que pasar (o peor aún, copiar) las estructuras de datos como argumentos de función. Para problemas con múltiples casos de prueba, simplemente borrar/restablecer el contenido de la estructura de datos antes de tratar con cada caso de prueba.
8. Cuando tenga la opción de escribir su código de forma iterativa o recursiva, elija el versión iterativa.
9. El acceso a la matriz en bucles (anidados) puede ser lento. Si tiene una matriz A y accede con frecuencia al valor de A[i] (sin cambiarlo) en bucles (anidados), puede ser beneficioso usar una variable local temp = A[i] y trabajar con temp en su lugar.
10. Para usuarios de C++: el uso de matrices de caracteres de estilo C producirá una ejecución más rápida que cuando utilizando string STL de C++. En el caso de Java utilizar *StringBuilder*

4. Implementación

A la hora de implementar una búsqueda completa existen varios enfoques los cuales veremos cada uno de ellos.

4.1. Búsqueda completa iterativa

Este enfoque tiene como característica realizar la búsqueda completa con la iteración por todos los elementos que pueden componer el campo de búsqueda.

4.1.1. Búsqueda completa iterativa (Dos bucles anidados)

Como bien su nombre lo indica este enfoque se caracteriza por la utilización de dos estructuras repetitivas anidadas las cuales permiten bien combinar en pares todos los elementos del campo de búsqueda para ver cuales pares forma una solución deseable.

Ejemplo: Encuentre y muestre todos los pares de números de 5 dígitos que en conjunto usen los dígitos del 0 al 9 una vez cada uno, de modo que el primer número dividido por el segundo sea igual a un número entero N, donde $2 \leq N \leq 79$. *Eso es , abcde/fghij = N, donde cada letra representa un dígito diferente. Se permite que el primer dígito de uno de los números sea cero, por ejemplo, para N = 62, tenemos 79546/01283 = 62; 94736/01528 = 62. UVa - 00725 - Division*

4.1.2. Búsqueda completa iterativa (Muchos bucles anidados)

Enfoque similar al anterior pero con la diferencia que la utilización de la cantidad de estructuras repetitivas anidadas no está limitada a una cantidad en específica.

Ejemplo: Dado 6 <k <13 enteros (que ya están ordenados), enumere todos los subconjuntos posibles de tamaño 6 de estos enteros en orden ordenado. UVa - 00441 - Lotto

4.1.3. Búsqueda completa iterativa (Bucles + Poda)

Este enfoque es una optimización de los dos anteriores ya que podemos establecer cortes o podas en las iteraciones a realizar cuando estamos en una subregión del campo de búsqueda del cual no se obtendrá ninguna solución factible.

Ejemplo: Dados tres enteros A, B y C $(1 \leq A, B, C \leq 10000)$*, encuentre otros tres enteros distintos x, y y z tales que* $x + y + z = A$, $x \times y \times z = B$, *y* $x^2 + y^2 + z^2 = C$. *UVa - 11565 - Simple Equations*

4.1.4. Búsqueda completa iterativa (Permutaciones)

Este enfoque tiene como campo de búsqueda todas las permutaciones que pueden generar un grupo de elementos y de ellas ver cuales cumple con determinada condición. Aunque existe una solución recursiva la variante iterativa es mucho mas eficiente a la hora de realizar una búsqueda completa.

Ejemplo: Hay $0 < n \leq 8$ *asistentes al cine. Se sentarán al frente fila en* n *asientos abiertos consecutivos. Hay* $0 \leq m \leq 20$ *restricciones de asientos entre ellos, donde cada restricción especifica dos asistentes al cine* a *y* b *que deben ser como máximo (o al menos)* c *asientos separados. La pregunta: ¿Cuántas disposiciones de asientos posibles hay ?. UVa - 11742 - Social Constraints*

4.1.5. Búsqueda completa iterativa (Subconjuntos)

Este enfoque tiene como campo de búsqueda todos los subconjuntos que pueden generar un grupo de elementos y de ellas ver cuales cumple con determinada condición. Aunque existe una solución recursiva la variante iterativa es mucho mas eficiente a la hora de realizar una búsqueda completa.

Ejemplo: Dada una lista l que contiene $1 \leq n \leq 20$ enteros, ¿existe un subconjunto de la lista l que suma a otro entero dado X?. UVa - 12455 - Bars

4.2. Búsqueda completa recursiva

Este enfoque tiene como característica realizar la búsqueda completa con la ayuda de la recursividad ya que la forma iterativa no asegura explorar todo el campo de búsqueda o se hace demasiado engorroso hacerlo mientras la variante recursiva propone una solaución simple y limpia.

4.2.1. Retroceso simple (*Backtracking*)

Un algoritmo de retroceso comienza con una solución vacía y extiende la solución paso a paso. La búsqueda recursivamente pasa por todas las diferentes formas en que se puede construir una solución. Este enfoque va construyendo un árbol con todos los posibles estados o variantes de solución donde cada nodo representa un estado o variante y partir de el se pueden generar nuevos estados o variantes. Los nodos que pueden generar nuevos estados o variantes tendrán una solución parcial al problema mientras los nodos los cuales ya no generaron nuevos estados o variantes serán nodos hojas en el árbol. Los nodos hojas del árbol tendrán una solución completa lo cual podrá ser tomada o no en dependencia del problema.

Ejemplo: En el ajedrez estándar (con un tablero de 8×8), es posible colocar 8-Reinas en el tablero de manera que no haya dos Reinas que se ataquen entre sí. Determinar todo eso posibles arreglos dada la posición de una de las Reinas (es decir, la coordenada (a, b) debe contienen una Reina). Muestra las posibilidades en orden lexicográfico (ordenado). UVa - 00750 - 8-Queens Chess Problem

4.2.2. Retroceso simple (*Backtracking*) + Poda

Nosostros podemos optimizar la búsqueda completa con retroceso si aplicamos poda en el árbol de búsqueda que está idea genera. La idea es no seguir explorando aquellos subárboles de búsqueda que se pueden generar a partir de un nodo del árbol que la solución parcial que propone ese nodo ya no es factible, esto hace que los subárboles que se generan a partir de este nodo propondan soluciones parciales o completas no factibles. Aplicación de la poda puede tener excelentes efectos en la eficiencia de la búsqueda

Ejemplo: Dado un tablero de ajedrez $n \times n$ ($3 \leq n \leq 15$) donde algunas de las las celdas son malas (las reinas no se pueden colocar allí), ¿de cuántas maneras se pueden colocar N-reinas en el tablero de ajedrez para que no haya dos reinas que se ataquen entre sí? Las celdas malas no se pueden usar para bloquear el ataque de las reinas. UVa - 11195 - Another N-Queens Problem

Si un problema se puede resolver mediante la búsqueda completa, también estará claro cuándo utilizar los enfoques de seguimiento iterativo o recursivo. Los enfoques iterativos se utilizan cuando uno puede derivar fácilmente los diferentes estados con alguna fórmula relativa a un cierto contador y (casi) todos los estados deben verificarse, por ejemplo, escaneando todos los índices de una matriz, enumerando (casi) todos los subconjuntos posibles de un conjunto pequeño, generando (casi) todas las permutaciones, etc.

El retroceso recursivo (*backtracking*) se usa cuando es difícil derivar los diferentes estados con un índice simple y/o uno también quiere (en gran medida) podar el espacio de búsqueda. Si el espacio de búsqueda de un problema que se puede resolver con la búsqueda completa es grande, generalmente se utilizan enfoques recursivos de retroceso que permiten la poda temprana de secciones no factibles del espacio de búsqueda. La poda en búsquedas completas iterativas no es imposible, pero suele ser difícil.

5. Aplicaciones

La búsqueda completa es una buena técnica si hay tiempo suficiente para recorrer todas las soluciones, porque la búsqueda suele ser fácil de implementar y siempre da la respuesta correcta. Si la búsqueda completa es demasiado lenta, es posible que se necesiten otras técnicas algorítmicas para resolver el problema.

En los concursos de programación, un concursante debe desarrollar una solución de búsqueda completa cuando claramente no hay otro algoritmo disponible (por ejemplo, la tarea de enumerar todas las permutaciones de $0, 1, 2, \ldots, N-1$ claramente requiere O(N!), es decir, al menos N! operaciones) o cuando existen mejores algoritmos, pero son excesivos ya que el tamaño de entrada resulta ser pequeño.

6. Complejidad

La complejidad de los algoritmos de búsqueda completa es variada ya que depende en gran medida del problema y de la forma en que se implemente el algoritmo que siga este enfoque. Por lo general las soluciones algorítmicas que utilizan la búsqueda completa sus complejidades son iguales o superiores a O(N^2).

7. Ejercicios propuestos

La mejor manera de mejorar sus habilidades de búsqueda completa es resolver más búsqueda completa problemas para que su intuición de si un problema se puede resolver con la búsqueda completa se vuelve mejor. Aquí proporcionamos una lista de tales problemas:

- DMOJ - A*B*C
- DMOJ - ¿Cuántas fichas de dominó? (I)
- DMOJ -Encuentra los Dígitos
- DMOJ - Generating Words
- DMOJ - El Arado Robot
- DMOJ - Sum
- DMOJ - Cambiando Luces

- DMOJ - Vacas Claustrofóbicas