

Estrategias para el diseño de Algoritmos: Un Estudio Comparativo.

Ojeda R., Hermes,
*Estudiante de Ingeniería en Computación,
 Octavo Semestre, Grupo A, UTM*

Abstract— De las estrategias para el diseño de algoritmos, cada una tiene una gran importancia, además de características propias. Este documento trata de dar un panorama general de algunas estrategias tales como: vueltra atrás, ramificación y poda, divide y vencerás, algoritmos ávidos y programación dinámica, así como una comparación de las mismas.

I. INTRODUCCIÓN

El diseño de un algoritmo para solucionar algún problema muchas veces se realiza de una forma intuitiva, pero eso no asegura su eficiencia y una idea rebuscada puede hacer que su implementación sea muy complicada.

La eficiencia y la facilidad para su implementación deben ser 2 cuestiones a tomar en cuenta al momento de diseñar un algoritmo, y cada una de estas tendrá un peso mayor o menor, dependiendo de la situación. Por ejemplo: Si una compañía solicitará un algoritmo para el manejo cuestiones críticas de la empresa (como controlar un reactor nuclear, o dispositivos de seguridad), sería conveniente que este proporcionara una respuesta rápida y confiable (sin importar mucho, que su implementación sea complicada), otro ejemplo, sería durante un concurso de programación algorítmica (tales como el ACM-ICPC, o TopCoder en los concursos de algoritmos), en este tipo de concursos es necesario tener en cuenta que el algoritmo diseñado sea eficiente pero además fácil de implementar, ya que un algoritmo demasiado eficiente para el cual se requieran 10 o 20 horas para implementarlo es inútil, así también lo es un algoritmo fácil de implementar pero que no sea eficiente para solucionar el problema en cuestión.

Tomando en cuenta, los aspectos de eficiencia y la facilidad de implementación, la teoría de algoritmos nos brinda algunas técnicas generales para el diseño de algoritmos, las cuales, al conocer sus características, ventajas y desventajas, son una herramienta importante para diseñar algoritmos que sean eficientes y muy probablemente fácil de implementar.

A lo largo de la historia del estudio de los algoritmos, se han podido encontrar diversas técnicas generales para producir algoritmos eficientes y lograr la resolución de una gran cantidad de problemas, algunas de la más importantes y que serán tratadas en este documento son:

Dividir y conquistar, también conocido como “divide y vencerás” (divide and conquer), algoritmos ávidos (greedy algorithms), método de retroceso o vuelta atrás (backtracking), ramificación y poda (Branch and Bound) y programación dinámica (dynamic programming o simplemente DP) [1].

II. DIVIDE Y VENCERÁS

La técnica de divide y vencerás (divide and conquer) también conocida como dividir y conquistar, esta técnica, se basa en la idea “entre más simple, mejor”, dividiendo el problemas en problemas más pequeños que pueden ser resueltos más fácil que el problema principal, después al combinar la solución de esos subproblemas se obtiene la solución del problema principal [2].

En general, la técnica de divide y vencerás consta de 3 pasos, los cuales son los siguientes:

- Dividir: Divide el problema en problemas más pequeños (similares o iguales al problema original).
- Conquistar: Resolver los subproblemas, usualmente de forma recursiva, hasta que los subproblemas sean tan pequeños como para poder resolverlos directamente.
- Combinar: Unir las soluciones obtenidas en los subproblemas para obtener la solución del problema original.

Esta técnica tiene muchas ventajas, la primera es que después de dividir el problema en subproblemas parecidos (que sería la parte complicada) su diseño e implementación son sencillas, otra ventaja es que provee una forma fácil y directa de solucionar problemas complejos, además de proporcionar una forma natural de adaptación a su ejecución en entornos multiprocesador [3].

Una de sus desventajas es el gasto producido por usar recursión, y por esa misma razón, la cantidad de datos a manejar es limitado, y no es conveniente para solucionar problemas con una cantidad de entradas recursivas extremadamente grandes.

Algo importante que hay que tener en cuenta con esta técnica, es que muchos algoritmos eficientes usan este paradigma del diseño de algoritmos, algunos de los cuales son: QuickSort, MergeSort, BinarySearch (Búsqueda Binaria).

III. ALGORITMOS ÁVIDOS

Los algoritmos ávidos (Greedy Algorithms), también conocidos como algoritmos voraces, es una técnica de diseño de algoritmos, que se basa principalmente en usar la mejor opción para un momento dado, para obtener la mejor solución general.

El funcionamiento de esta técnica sigue los siguientes pasos:

- Para cada paso en el algoritmo se elige la mejor opción, que se tiene disponible.
- Se comprueba si la opción elegida podría llevar a la solución del problema.
- Si esa opción puede llevar a la solución se elige, sino se elige la siguiente mejor opción.
- Si el conjunto de “mejores opciones” escogidas es solución, se termina el algoritmo. Sino se continúa [4].

Esta técnica nos brinda muchas ventajas, las soluciones “ávidas” son muy fáciles de implementar, por lo tanto fáciles de depurar, la velocidad de ejecución es muy rápida además de usar muy poca memoria [5].

Aunque parecería que este es la mejor forma de solucionar un problema, existe una desventaja muy importante, no todos los problemas se pueden resolver con esta técnica, ya que no darían la solución apropiada para el mismo.

También hay que tomar en cuenta que existen muchos algoritmos conocidos que usan esta técnica, como son: Algoritmo de Kruskal, Algoritmo de Prim, Algoritmo de Dijkstra. Los 2 primeros, son algoritmos para encontrar el árbol de expansión mínimo de un grafo, y el tercero sirve para encontrar la ruta mínima entre un par de nodos de un grafo. En los 3 casos, los algoritmos dan una solución óptima y correcta.

Por lo tanto, si existe una solución voraz correcta para el problema, es una solución que proporciona grandes ventajas.

IV. BACKTRACKING

La técnica de vuelta atrás (Backtracking) también conocida método de retroceso, es una técnica muy potente para la solución de problemas.

El Backtracking es un método sistemático para iterar a través de todas las posibles configuraciones de un espacio de búsqueda, se puede decir que es una técnica o algoritmo general que puede ser adaptado para diferentes aplicaciones [6].

El backtracking es un método de solución completo que revisará todas las posibles soluciones, es comúnmente implementado de forma recursiva.

La forma en que funciona, se puede resumir en 2 pasos.

- Si el actual conjunto S es una solución al problema, de ser así procesarla (esto depende específicamente del problema).
- En otro caso, construir extensiones para el conjunto S e invocar el algoritmos con este nuevo conjunto.

Esta técnica tiene varias ventajas, una es la facilidad en la implementación, ya que con un poco de experiencia con recursividad implementar un backtracking es relativamente

sencillo, además este método se adapta fácilmente a muchos problemas.

Una de las desventajas que tiene esta técnica, es que no se puede trabajar con espacios de búsqueda muy grandes, además de que la implementación recursiva requiere ciertos gastos en memoria extra, también hay que tener en cuenta que posiblemente se haga el cálculo de alguna solución que ya fue probada, en otro momento.

Las soluciones a algunos problemas comunes son implementados por medio de backtracking, por ejemplo el problema de las 8 reinas, el recorrido del caballo o atravesar un laberinto. Aunque no es la solución más eficiente es la más directa para implementar.

V. RAMIFICACIÓN Y PODA

La ramificación y poda (Branch and Bound), es una técnica muy parecida al backtracking con la única diferencia que se agregan ciertas condiciones para evitar hacer cálculos que no son necesarios.

Si tomamos en cuenta que al usar el backtracking se crea un árbol de recursión, lo que se hace con esta técnica, es quitar las ramas de ese árbol que no lleguen a la solución deseada.

Para lograr esto su funcionamiento se podría plantear en los siguiente pasos:

- Tomar un nodo (que se supone no ha recibido podas) dependiendo de la estrategia a seguir.
- Ramificar este nodo (es decir, generar todas las posibles nodos hijos del nodo actual).
- Podar: De los nodos ramificados eliminar los que no lleven a la solución.

Esta técnica de solución tiene grandes ventajas, ya que toma los puntos fuertes del backtracking, quitando algunas de sus desventajas, ya que eligiendo una buena estrategia se puede eliminar el cálculo de duplicados, así como hacer que la convergencia a la solución sea mucho más rápida.

La desventaja más fuerte de esta técnica, es que depende mucho del problema a solucionar. Después de analizar el problema, se puede obtener una estrategia adecuada para la convergencia del problema, y si las estrategias son complicadas implica una mayor dificultad al implementar, además de que si la estrategia elegida es incorrecta puede arrojar soluciones también incorrectas.

Los problemas que se pueden solucionar con esta técnica son muy variados, ya que a la mayoría de problemas que se resuelven por backtracking se le puedan agregar estrategias para acelerar su convergencia.

VI. PROGRAMACIÓN DINÁMICA

La técnica de programación dinámica (dynamic programming o DP) es una técnica usada para optimizar algoritmos mediante la utilización de “subproblemas superpuestos” y “subestructuras óptimas” [7].

Esta estrategia, cuando existen subestructuras óptimas usa el

principio de Bellman: “En una secuencia de decisiones óptimas toda subsecuencia debe ser óptima”.

Para lograr la solución cuando existen subestructuras óptimas, se pueden seguir los siguientes pasos.

- Dividir el problema en subproblemas más pequeños.
- Resolver estos problemas de manera óptima usando el mismo proceso (actual) recursivamente.
- Usar las soluciones óptimas obtenidas para construir la solución del problema original.

Cuando un problema se dice que tiene problemas superpuestos es que un mismo problema es usado para solucionar problemas mayores, cuando esto existe, con un mala implementación estos problemas son calculados muchas veces, para este caso, se usa una técnica de programación dinámica conocida como memoización, la cual almacena las soluciones calculadas, para evitar hacer de nuevo los mismos cálculos.

Para solucionar un problema con programación dinámica se pueden usar 2 enfoques.

- Top-down: Es una combinación de recursión y memoización, ya que el cálculo se realiza recursivamente y solamente se almacenan las soluciones para no calcularlas de nuevo en alguna entrada recursiva.
- Bottom-up: Todos los subproblemas que se requieran para un problema general se calculan previamente, para así poder calcular soluciones mayores.

Una forma general para abordar la programación dinámica se puede dividir en 4 pasos [8].

- Definir la estructura de una solución óptima.
- Definir recursivamente el valor de una solución óptima.
- Calcular el valor de una solución óptima con un enfoque bottom-up.
- Construir la solución óptima con la información calculada.

Las ventajas de la programación dinámica son muy importantes, ya que de conseguir una solución correcta por esta técnica, la cantidad de cálculos y el uso de memoria se reducen mucho. Con esto se logra que en problemas que calculados recursivamente o por medio de backtracking, se reduzca su orden de complejidad de una forma considerable.

De las desventajas más importantes de esta técnica, es que no es fácil solucionar problemas con programación dinámica, requiere de un análisis profundo, así como experiencia. Además de que es dependiente del problema. En cuanto a la facilidad de implementación depende de que tan complicada haya sido la solución encontrada.

Uno de los algoritmos de programación dinámica más importante es el Floyd-Warshall, el cual nos permite calcular la ruta más corta entre todo par de nodos de un grafo. Además algunos problemas que se pueden resolver por esta técnica: Edit Distance [9], Matrix-Chain Multiplication, o Longest Common Subsequence [10].

VII. COMPARACIÓN DE LAS DIFERENTES TÉCNICAS

Hacer una elección de parámetros de comparación de las diferentes técnicas es complicado por las diferencias tan marcadas entre las técnicas, pero arbitrariamente se tomarán los siguientes parámetros, para realizar una comparación:

- Dependencia del problema (DEP): Este parámetro es la cantidad de profundidad en el análisis del problema necesario para el uso de la técnica. Valores: (A – Alta, M – Media, B – Baja).
- Facilidad de implementación (FAC): Este parámetro se refiere a la complejidad para pasar el algoritmo a algún lenguaje de programación (F – Fácil, M – Medio, C – Complejo).
- Cantidad de datos a usar (DAT): Este se refiere a la cantidad de datos que se pueden usar con esa técnica, y que el tiempo de ejecución del algoritmo sea aceptable. Valores: (P – Pequeño, M – Mediano, G – Grande).
- Experiencia (EXP): Este es un parámetro se refiere a la cantidad de experiencia necesaria para diseñar un algoritmo con esta técnica. No es muy objetivo el uso de este parámetro, pero puede servir como referencia. Valores: (P – Poca, M – Media, E - Experimentado).
- Uso de Recursividad (REC): Tomando en cuenta, que la recursividad gasta recursos extra, podemos tomar el uso de recursividad como un parámetro de comparación. Valores: (NN – No necesaria, N – Necesaria).

Usaremos además la siguiente notación: (DC – Divide y vencerás, BT – Backtracking, BB – Ramificación y podas, GA – Algoritmos ávidos, DP – Programación dinámica).

	DC	BT	BB	GA	DP
DEP	M	B	A	A	A
FAC	F	F	C	F	M
DAT	M	P	M	G	G
EXP	M	P	M	E	E
REC	N	N	N	NN	NN

Fig. 1. Tabla comparativa de las técnicas de diseño de algoritmos.

Como se puede ver, en esta tabla no se tomó en cuenta cosas como la complejidad temporal, ya que esto va a depender estrictamente del problema a resolver.

Revisando la tabla de la Fig. 1. se puede ver que en el aspecto de Facilidad de implementación se podría tomar a la programación dinámica, que es medianamente complicado, o también fácil de implementar, ya que después de diseñar un algoritmo con programación dinámica, si se hizo de forma correcta, la implementación no es muy complicada. En cuanto a la técnica de ramificación y poda, es complicado implementar, porque muchas veces las estrategias para hacer las podas y las ideas para hacerlo, es complicado codificarlo en un lenguaje de programación.

En el aspecto de Cantidad de datos a usar, tiene un poco que ver con la complejidad y la cantidad de memoria usada, y

pues el backtracking al probar todas las posibles soluciones es el que puede trabajar con conjuntos más pequeños de datos, si se implementa una Ramificación y poda eficiente se puede hacer que aumente el rango de datos con los que puede trabajar el backtracking, el divide y vencerás por la forma en que trabaja, la cantidad de datos con los que puede trabajar aumenta a comparación de un backtracking, y algoritmos ávidos o de programación dinámica, por la forma en que funcionan y al evitar redundancia en cálculos la mayoría de veces trabajan con una cantidad de datos grande, sin perder eficiencia el algoritmos.

En cuanto a la experiencia necesaria, el que requiere menos experiencia es el backtracking, ya que es una técnica muy general y fácilmente adaptable. En cambio diseñar un algoritmo con “divide y vencerás” o “ramificación y podas” requiere un poco más de experiencia, sobre todo al escoger las estrategias o al elegir la división del problema. Los algoritmos ávidos o la programación dinámica requieren una mayor experiencia para el diseño de algoritmos, elegir los componentes de un algoritmo de programación dinámica suele ser muy complicado, así también lo es saber si un algoritmo ávido nos va a dar o no, la solución correcta.

Por último el aspecto de la recursividad solamente es para saber si es necesario usar recursividad para el uso de las diferentes técnicas. Las técnicas “divide y vencerás”, “backtracking” y “ramificación y podas”, son naturalmente recursivas, aunque se pueden implementar de otras formas, pero en general es necesario usar recursividad. Las técnicas de “programación dinámica” y “algoritmos ávidos” no dependen directamente de la recursividad, ya que si se diseñara un algoritmo de programación dinámica, usando los 4 pasos generales completamente, el algoritmo no es necesario que use recursividad, así también los algoritmos ávidos, no dependen directamente de la recursividad, aunque eso no quita la posibilidad de implementar algún algoritmo con estas técnicas usando recursividad.

VIII. CONCLUSIÓN

Es difícil definir una técnica adecuada para solucionar un problema, ya que depende mucho del mismo. Pero se pueden usar algunos parámetros para elegir la mejor técnica.

Si lo que se requiere para la solución es eficiencia y corrección, a toda costa. Se puede optar por un análisis profundo del problema por medio de programación dinámica, o algún algoritmo voraz.

Si lo que se requiere es facilidad de implementación, sin que la eficiencia del algoritmo sea tan importante, se puede optar por técnicas como backtracking, y usar ramificación y poda según lo permita el problema en caso de que se quiera mejorar la eficiencia.

En caso de que se desee un punto intermedio entre eficiencia y facilidad de implementación, se debe tener en cuenta la técnica de divide y vencerás, ya que tal vez no nos proporcione la solución óptima en cuanto a eficiencia, pero si

el problema lo podemos adaptar con esta técnica, nos proporcionará una solución aceptable en términos de eficiencia y facilidad de implementación.

Para elegir la mejor solución acorde a un problema, se deben tener en cuenta todas las técnicas mencionadas en este documento (y tal vez algunas otras más), y resolver problemas diversos para lograr la experiencia que nos permita la mejor elección de la técnica a usar, en términos de eficiencia y facilidad de implementación.

REFERENCIAS

- [1] Víctor Valenzuela Ruz
Manual de análisis y diseño de algoritmos. Versión 1.0
<http://www.angelfire.com/my/jimena/algoritmos/Algoritmos.pdf>
- [2] Steven Halim
World of Seven
Basic Techniques
http://www.comp.nus.edu.sg/~stevenha/programming/prog_basictchniques.html#Divide%20and%20Conquer
- [3] Wikipedia
http://es.wikipedia.org/wiki/Divide_y_vencer%C3%A1s
- [4] Wikipedia
http://es.wikipedia.org/wiki/Algoritmo_voraz
- [5] Steven Halim
World of Seven
Greedy
http://www.comp.nus.edu.sg/~stevenha/programming/prog_greedy.html
- [6] Skiena, Steven S. y Revilla Miguel A.
Programming Challenges
Ed. Springer
2003, Pag. 167.
- [7] Wikipedia
http://es.wikipedia.org/wiki/Programaci%C3%B3n_din%C3%A1mica_%28computaci%C3%B3n%29
- [8] Cormen, Thomas H.
Introduction to Algorithms
Second Edition
MIT Press
Ed. McGraw Hill
2003, Pag. 323
- [9] Skiena, Steven S. y Revilla Miguel A.
Programming Challenges
Ed. Springer
2003, Pag. 246.
- [10] Cormen, Thomas H.
Introduction to Algorithms
Second Edition
MIT Press
Ed. McGraw Hill
2003, Pag. 331 y 350