PRÁCTICA 2: ALGORITMOS VORACES, DINÁMICOS Y VUELTA ATRÁS

Diego Viñals Lage y Carlos Vega

COMPLEJIDAD COMPUTACIONAL 3B

Tabla de contenido

Introducción

El objetivo de esta práctica es afianzar y aprender los algoritmos voraces, dinámicos y de vuelta atrás. Para lograrlo, se nos presenta el problema de la mochila, el cual debemos resolver utilizando estos tres tipos de algoritmos. Una empresa de café se ha comunicado con nosotros para que los ayudemos a decidir qué mercancías deben transportar en un vuelo internacional, teniendo en cuenta los costos de transporte y seguros. La empresa tiene un límite de peso que puede llevar y desea transportar el mayor valor posible.

Para esto, aplicaremos los diferentes paradigmas que hemos visto en clase: el paradigma voraz, el paradigma dinámico y el paradigma de vuelta atrás.

La capacidad límite que el avión puede transportar es de 600.000 kg, y nosotros disponemos de estos cafés.

Producto	Peso (kg)	Precio total ($)
Pales cafés de Tanzania	260.000	26.000.000
Pales cafés de Hawái	150.000	18.000.000
Pales cafés de Nicaragua	240.000	19.200.000
Pales cafés de Jamaica	190.000	34.200.000
Pales cafés de Colombia	130.000	19.500.000
Pales cafés de Kenia	140.000	9.800.000

Algoritmo “Voraz”

Código

```
public static List<Producto> SeleccionarMercanciasVoraz(List<Producto> productos, int capacidad)
{
    List<Producto> seleccionados = new List<Producto>();

    // Ordenar productos por su relación valor-peso de forma descendente
    productos = productos.OrderByDescending(p => (double)p.Valor / p.Peso).ToList();

    foreach (Producto producto in productos)
    {
        // Comprobar si el producto puede ser incluido en la selección actual
        if (producto.Peso <= capacidad)
        {
            seleccionados.Add(producto);
            capacidad -= producto.Peso;
        }
    }

    return seleccionados;
}
```

Complejidad en tiempo y espacio

La complejidad computacional de este algoritmo es de O(n log n), donde n es el número de productos. Usa OrderByDescending, que tiene esa complejidad, después isa foreach que tiene complejidad de N, pero al final la complejidad viene dada por la ordenación de los elementos.

La complejidad espacial es de O(n), ya que usa una lista llamada productos, que contiene todos los productos, es decir, n

Solución

La solución que devuelve este algoritmo es:

```
Mercancías seleccionadas (algoritmo voraz):
Nombre: Café de Jamaica, Peso: 190000, Valor: 34200000
Nombre: Café de Colombia, Peso: 130000, Valor: 19500000
Nombre: Café de Hawaii, Peso: 150000, Valor: 18000000
```

Características

El algoritmo voraz selecciona productos en base a su relación valor-peso, optando por aquellos que tienen la mejor relación en cada paso. En esencia, toma decisiones óptimas a nivel local sin tener en cuenta las implicaciones a largo plazo. Sin embargo, no ofrece garantías de encontrar la solución óptima en todos los casos, ya que puede llegar a una solución posible pero no necesariamente la mejor para el problema en cuestión.

Este enfoque es recomendado cuando se busca una solución rápida y se está dispuesto a aceptar una solución subóptima. Resulta útil en situaciones donde no es necesario encontrar la mejor solución global, pero se requiere una solución viable que sea lo suficientemente buena.

El algoritmo voraz es especialmente adecuado para problemas en los que cada paso puede resolverse de manera independiente, sin depender de las decisiones tomadas previamente. Esto significa que no se necesita tener en cuenta el contexto completo del problema en cada paso, lo que agiliza el proceso de selección.

Sin embargo, no se recomienda utilizar el algoritmo voraz cuando se requiere una solución óptima garantizada o cuando el problema tiene restricciones complejas que deben considerarse en cada paso. Además, puede no ser apropiado cuando la calidad de la solución es crítica y se busca maximizar el valor total o minimizar el peso total de las mercancías seleccionadas.

En resumen, el algoritmo voraz es una opción viable cuando se busca una solución rápida y aceptable, sin necesidad de encontrar la mejor solución global. Es adecuado para problemas donde cada paso puede resolverse de manera independiente y no se requiere una precisión extrema. Sin embargo, en casos que exijan la mejor solución posible o presenten restricciones complejas, es recomendable considerar otros enfoques más sofisticados.

¿Y si los productos fuesen divisibles?

Si los productos fuesen divisibles el algoritmo tendría fácil adaptación ya que, en cada fase en caso de no poder elegir el producto completo, elegiría la parte que le queda por completar. Esto permite aprovechar al máximo el espacio y obtener el mejor valor posible. En este caso si sería apropiado su uso.

Algoritmo “vuelta atrás”

Código

```
public static List<Producto> SeleccionarMercanciasVueltaAtras(List<Producto> productos, int capacidad)
{
    List<Producto> mejorSeleccion = new List<Producto>();
    List<Producto> seleccionActual = new List<Producto>();

    BuscarMejorSeleccion(productos, capacidad, 0, seleccionActual, ref mejorSeleccion);

    return mejorSeleccion;
}

public static void BuscarMejorSeleccion(List<Producto> productos, int capacidad, int indice, List<Producto> seleccionActual, ref List<Producto> mejorSeleccion)
{
    if (indice == productos.Count)
    {
        if (ObtenerPesoTotal(seleccionActual) <= capacidad && ObtenerValorTotal(seleccionActual) > ObtenerValorTotal(mejorSeleccion))
        {
            mejorSeleccion = new List<Producto>(seleccionActual);
        }

        return;
    }

    Producto productoActual = productos[indice];

    seleccionActual.Add(productoActual);
    BuscarMejorSeleccion(productos, capacidad, indice + 1, seleccionActual, ref mejorSeleccion);
    seleccionActual.Remove(productoActual);

    BuscarMejorSeleccion(productos, capacidad, indice + 1, seleccionActual, ref mejorSeleccion);
}

public static int ObtenerPesoTotal(List<Producto> productos)
{
    int pesoTotal = 0;

    foreach (Producto producto in productos)
    {
        pesoTotal += producto.Peso;
    }

    return pesoTotal;
}
```

```
public static int ObtenerValorTotal(List<Producto> productos)
{
    int valorTotal = 0;

    foreach (Producto producto in productos)
    {
        valorTotal += producto.Valor;
    }

    return valorTotal;
}
```

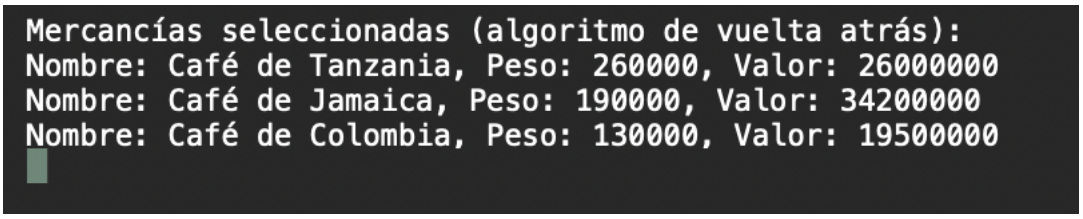
Complejidad computacional en tiempo y espacio

Este algoritmo tiene una complejidad temporal en el peor de los casos de O(2^n), ya que la función BuscarMejorSeleccion se ejecuta dos veces anidadas, lo que aumenta exponencialmente la complejidad.

La complejidad espacial de este algoritmo es de O(n), ya que utiliza una lista que almacena todos los productos, y el espacio requerido es proporcional al número de productos, que en este caso se representa por "n".

Salida

La salida de este algoritmo es la siguiente:



Características

El algoritmo de Vuelta Atrás es una técnica de búsqueda exhaustiva que se emplea para encontrar todas las soluciones posibles a un problema. Su principal característica radica en su enfoque sistemático al explorar el espacio de soluciones, retrocediendo cuando se determina que una solución parcial no puede conducir a una solución válida.

Cuando se busca obtener todas las soluciones posibles en lugar de una única solución óptima, se recomienda utilizar el algoritmo de Vuelta Atrás. Esta técnica resulta útil cuando se desea examinar minuciosamente el espacio de soluciones y cuando la estructura del problema permite la aplicación de una estrategia de retroceso. Es particularmente valioso cuando es necesario identificar todas las combinaciones o permutaciones viables dentro de un conjunto de elementos.

No se aconseja su utilización en casos donde el problema involucre una gran cantidad de elementos, ya que el algoritmo de Vuelta Atrás puede volverse ineficiente en términos de tiempo y recursos. En tales situaciones, puede resultar más eficiente emplear otras alternativas. Además, si el problema presenta restricciones complejas que

no se pueden manejar fácilmente mediante una estrategia de retroceso, puede ser necesario recurrir a algoritmos más avanzados o técnicas específicas diseñadas para abordar esas restricciones.

¿Y si los productos fuesen divisibles?

Si los productos son divisibles las posibles soluciones serian mayores por lo que la complejidad y los recursos irían en aumento. Seria el algoritmo menos recomendado de los propuestos.

Algoritmo “Dinámico”

Código

```
public static List<Producto> SeleccionarMercanciasDinamico(List<Producto> productos, int capacidad)
{
    // Crear una matriz para almacenar los resultados intermedios
    int[,] matriz = new int[productos.Count + 1, capacidad + 1];

    // Llenar la matriz con los resultados óptimos
    for (int i = 0; i <= productos.Count; i++)
    {
        for (int j = 0; j <= capacidad; j++)
        {
            if (i == 0 || j == 0)
                matriz[i, j] = 0;
            else if (productos[i - 1].Peso <= j)
                matriz[i, j] = Math.Max(productos[i - 1].Valor + matriz[i - 1, j - productos[i - 1].Peso], matriz[i - 1, j]);
            else
                matriz[i, j] = matriz[i - 1, j];
        }
    }

    // Recuperar la selección óptima de productos
    List<Producto> seleccionados = new List<Producto>();
    int fila = productos.Count;
    int columna = capacidad;

    while (fila > 0 && columna > 0)
    {
        if (matriz[fila, columna] != matriz[fila - 1, columna])
        {
            seleccionados.Add(productos[fila - 1]);
            columna -= productos[fila - 1].Peso;
        }

        fila--;
    }

    return seleccionados;
}
```

Complejidad computacional en tiempo y espacio

La complejidad temporal de este algoritmo es de O(n), donde "n" es el número de productos. El algoritmo utiliza dos bucles, pero no están anidados, lo que significa que se ejecutan secuencialmente uno después del otro. Por lo tanto, la complejidad del algoritmo viene dada por el bucle que itera sobre los productos, y en este caso, ese bucle tiene una complejidad lineal O(n).

La complejidad espacial de este algoritmo es de O(n), donde "n" es el número de productos. El algoritmo utiliza una lista para almacenar todos los productos disponibles. Dado que la lista contiene "n" productos, el espacio requerido para almacenarlos es proporcional al número de productos.

Salida

La salida de este algoritmo es la siguiente:

```
Mercancías seleccionadas (algoritmo dinámico):
Nombre: Café de Colombia, Peso: 130000, Valor: 19500000
Nombre: Café de Jamaica, Peso: 190000, Valor: 34200000
Nombre: Café de Tanzania, Peso: 260000, Valor: 26000000
```

Características

El algoritmo dinámico resuelve un problema dividiéndolo en subproblemas más pequeños y utilizando las soluciones óptimas de esos subproblemas para construir la solución óptima del problema original.

El algoritmo dinámico puede requerir una mayor cantidad de recursos en comparación con el algoritmo voraz, especialmente en términos de memoria. Esto se debe a que necesita almacenar y calcular los resultados de los subproblemas para poder construir la solución final.

Recomendaría utilizar el algoritmo dinámico cuando se busca una solución óptima a un problema de optimización y existen solapamientos entre los subproblemas. También es adecuado cuando se necesita realizar consultas repetidas a un conjunto de datos y se desea evitar recalcular los mismos resultados una y otra vez.

Sin embargo, no recomendaría utilizar el algoritmo dinámico en situaciones donde el problema no presente solapamiento de subproblemas, ya que el enfoque dinámico puede resultar innecesariamente complejo y requerir más recursos en comparación con otras técnicas más simples. Además, si la cantidad de elementos del problema es extremadamente grande y la tabla o matriz de resultados ocuparía una cantidad significativa de memoria, puede ser necesario considerar alternativas más eficientes en términos de recursos.

¿Y si los productos fuesen divisibles?

Si los productos son divisibles también seria un buen algoritmo a utilizar.