PROGRAMACIÓN DINÁMICA

El problema de el mayor beneficio invirtiendo en acciones de empresas

Autores
Alejandro Ramos Peña
Pedro Antonio Mayorgas Parejo

Contents

1 Introducción

En este problema se nos da un listado de empresas, cada cual tiene:

1. Una cantidad de acciones b_i
2. Un precio por acción p_i
3. Una comisión por acción c_i
4. Un beneficio por acción dado en porcentaje b_i

El beneficio real que te da comprar una acción se calcula multiplicando $p_i * b_i$

Dados esos datos de un número n de empresas, se nos pide calcular el mayor beneficio real posible dado un presupuesto.
Para ello, hemos diseñado 3 algoritmos distintos:

- Dos algoritmos simples
- Un algoritmo basado en programación dinámica.

2 Formato de los archivos .stock

Para especificar las características de las empresas que tendremos como entrada de nuestro programa hemos definido un tipo de archivo *.stocks*.
Los archivos acabados en *.stocks* vienen dados de la siguiente forma:
E
n
$a_0\ p_0\ c_0\ b_0$
$a_1\ p_1\ c_1\ b_1$
$a_2\ p_2\ c_2\ b_2$
...
$a_n\ p_n\ c_n\ b_n$

Donde n es el número de empresas que hay.

3 Algoritmos simples

Para los algoritmos simples nos planteamos comparar todas las combinaciones posibles de compra de acciones, teniendo en cuenta tan solo aquellas que no superaran el presupuesto y quedándonos así con la que más beneficio nos diera. Es así como llegamos a el primer algoritmo simple.

3.1 Primer algoritmo simple

El primer algoritmo simple lo que hace es hacer las partes del conjunto de acciones y comprobar el coste de cada subconjunto de las partes, comprobando que el precio no es mayor del presupuesto y quedándose con el que mayor beneficio tiene.

3.1.1 Pseudocódigo cálculo de las partes del conjunto de acciones

El pseudocódigo que calcula las partes del conjunto de acciones quedaría de la siguiente manera:

$i = 0$
while $i \neq Acciones.size()$ **do**
    $j = 0$
    while $j \neq Acciones[i]$ **do**
        $Partes.push(i)$
    end while
    $i++$
end while

Donde el vector *Acciones* contiene el número de acciones de la empresa de la siguiente manera: $\{a_0, a_1, a_2, ...a_n\}$

3.1.2 Pseudocódigo cálculo de la mejor opción de compra

Una vez tenemos las partes del conjunto de acciones, comprobamos el coste que tiene y si es menor que el presupuesto y su beneficio es mayor que el de la mejor opción, entonces actualizamos la mejor opción.
El pseudocódigo quedaría de la siguiente manera:

$MejorBeneficio = -\infty$
$MejorCombinacion = \{\}$
while $i < Partes.size()$ **do**
    if $Precio(Partes[i]) < Presupuesto$ && $Beneficio(Partes[i]) > MejorBeneficio$) **then**
        $MejorBeneficio = Beneficio(Partes[i]$
        $MejorCombinacion = Partes[i]$
    end if
    $i++$
end while
return $MejorBeneficio, MejorCombinacion$

De esta manera comprobamos todas las posibilidades de combinación de acciones.

El problema de este algoritmo es que es muy ineficiente, pues realiza muchísimas combinaciones repetidas al basarse en las partes del conjunto de acciones, pues combinaciones como $\{0, 0, 1\}$ podrían llegar a aparecer muchísimas veces si las empresas tuvieran un número elevado de acciones.

Es por ello que decidimos diseñar un algoritmo que fuera mucho más eficiente.

3.1.3 Orden de eficiencia

Se itera sobre todas las combinaciones posibles, lo cual es de orden 2^n y para cada combinación hay que comprobar el precio y beneficio, cuyas funciones tienen cada una un orden

de $O(n)$, por lo que la eficiencia del algoritmo es de $O(n * 2^n)$.

3.2 Segundo algoritmo simple

Para este algoritmo pensamos en implementar una manera distinta de iterar sobre las posibles combinaciones de acciones de las empresas.
Al igual que en el anterior algoritmo tendríamos el vector *Acciones* con el mismo formato. Inicializamos un vector *Combinacion* con el mismo tamaño que el vector *Acciones* pero con todos los valores a 0. Entonces vamos sumando 1 a el primer componente hasta llegar a $Acciones[i]$, donde volveríamos a comprobar y hacemos el módulo de $Acciones[i]$ y sumamos 1 a la componente $Combinacion[i+1]$ realizando también el módulo de $Acciones[i+1]$.

De esta manera habríamos implementado una manera de comprobar todas las combinaciones posibles de acciones.

3.2.1 Ejemplo

Por ejemplo, tenemos el siguiente vector $Acciones = \{2, 1, 3\}$.
Entonces las combinaciones de compra que se comprobarían serían las siguientes:
$\{0,0,0\}, \{1,0,0\}, \{2,0,0\}, \{0,1,0\}, \{1,1,0\}, ..., \{1,1,2\}, \{2,1,2\}, \{2,1,3\}$

Comprobando de esta manera todas las combinaciones posibles de compra de acciones, quedándonos con la que mejor beneficio tenga con un precio menor que el presupuesto.

3.2.2 Pseudocódigo

El pseudocódigo de el algoritmo simple más eficiente es el siguiente:

```
MejorCombinacion = {}
MejorBeneficio = −∞
while Combinacion ≠ Acciones do
    if Precio(Combinacion) < Presupuesto && Beneficio(Combinacion) > MejorBeneficio
then
        MejorBeneficio = Beneficio(Combinacion)
        MejorCombinacion = Combinacion
    end if
    i = 0
    Combinacion[0] + +
    while i ≠ Acciones.size() do
        if Combinacion[i] == Acciones[i] && Combinacion[i+1] ≠ Acciones[i+1] then
            if Precio(Combinacion) < Presupuesto && Beneficio(Combinacion) > MejorBeneficio
then
                MejorBeneficio = Beneficio(Combinacion)
                MejorCombinacion = Combinacion
            end if
            Combinacion[i] = 0
            Combinacion[i + 1] + +
        end if
        i + +
    end while
end while
return MejorCombinacion, MejorBeneficio
```

Y así comprobamos todas las combinaciones posibles de compra de acciones y conseguimos la mejor.

3.2.3 Orden de eficiencia

Tendríamos un orden de eficiencia $O(n^2)$ de, donde n es el número de empresas, pues en el peor caso iteramos sobre el vector de combinación todo el rato y tenemos que realizar n veces esa función.

4 Algoritmo basado en Programación Dinámica

Para realizar el algoritmo primero tenemos que reordenar el orden de las empresas para que, la primera empresa, sea la que tenga mejor ratio de beneficio real por coste real.
El beneficio real para la empresa i como hemos comentado al principio será el valor $b_i * p_i$, mientras que el coste real será $p_i + c_i$, calculando así el ratio $beneficioReal/costeReal$ y cambiando el orden en el que vienen las empresas.

Este cambio en el orden se realiza para que en general, la opción en la que se evalúa la empresa anterior para un presupuesto sea mejor opción ya que el beneficio real que te da una acción de esa empresa será mayor.

4.1 Tabla de beneficios

La tabla de beneficios tiene en las columnas presupuestos que van aumentando hasta llegar a el presupuesto máximo, el cual está en la última columna y tantas filas como empresas hay.
Para el algoritmo hemos usado un número de columnas de 11.
Los valores de las columnas los hemos calculado como el presupuesto máximo entre el número de columnas-1, ya que no contamos la columna 0, pues tiene un presupuesto de 0.
De esta manera definimos una variable *intervalo* que se calcula de la siguiente manera: $intervalo = (presupuestoMaximo)/(numeroDeColumnas - 1)$ y cada columna j tendría un valor $j * intervalo$, teniendo la última columna el valor del presupuesto máximo.
La posición de la tabla $[n-1][10]$ tendrá el valor que le pedimos, el mayor beneficio posible teniendo en cuenta todas las empresas disponibles y el presupuesto dado.

4.2 Struct de compra

Para almacenar la combinación de acciones compradas además de su coste y beneficio, hemos creado un struct llamado *Compra* el cual contiene un array del mismo tamaño que el vector *Acciones*, el coste de comprar esas acciones y el beneficio.
La tabla de beneficios estará formada por estructuras de este tipo, donde cada $mem[i][j]$ contiene la mejor compra posible considerando que podemos comprar acciones desde la empresa 0 hasta la i con un presupuesto de $j * intervalo$.

4.3 Inicialización de la tabla de beneficios

Inicializamos la tabla de la siguiente manera:

- La columna 0 tendrá un valor de 0 de beneficio y coste.
- La fila 0 se inicializará con el mayor beneficio de comprar el mayor número de acciones de la empresa 0 con un presupuesto de $j * intervalo$.
- Las demás celdas se inicializarán a -1.

4.4 Ecuación de recurrencia

Dado que buscamos el mayor beneficio, la ecuación de recurrencia devolverá el valor más grande entre distintos valores. Para ir rellenando la tabla con los mejores valores de beneficios comparamos **4 casos** distintos:

1. Caso en el que el beneficio teniendo en cuenta hasta la empresa anterior y con lo que sobra compramos acciones de la empresa actual.

2. Caso en el que cogemos la mejor compra para el presupuesto anterior para la empresa actual y con el dinero que sobre de hacer esa compra compramos el mayor número posible de acciones de la empresa actual.

3. Caso en el que cogemos la mejor compra para el presupuesto anterior y con el dinero que sobra cogemos la mejor compra para ese dinero sobrante teniendo en cuenta hasta la empresa anterior nada más.

4. Caso en el que compramos solo acciones de la empresa actual.

De esos cuatro casos sacamos una ecuación de recurrencia:

$$
\begin{aligned}
mem[i][j] = max(& \\
& mem[i-1][j] + \frac{restante}{p_i + c_i} * b_i, \\
& mem[i][j-1] + \frac{restante}{p_i + c_i} * b_i, \\
& mem[i][j-1] + mem[i-1][\frac{restante}{intervalo}] + \frac{restante}{p_i + c_i} * b_i, \\
& \frac{presupuesto}{p_i + c_i} * b_i \\
&)
\end{aligned}
\tag{1}
$$

Donde *restante* hace referencia a el dinero restante de comprar la compra anterior.

4.5 Orden de eficiencia

En el código que calcula la mejor compra posible para un presupuesto y unas empresas dadas se realizan 4 bucles for donde se comprueban los precios de los 4 casos posibles cuyas iteraciones son iguales a el número de empresas que hay (n) por lo que el orden de la función sería de **O(n)**.

Aunque aún siendo el orden de eficiencia tan bajo, hay que tener en cuenta que es una función recursiva y va a ejecutarse muchas veces, pues tiene que rellenar todas las celdas de la tabla, así que la constante oculta de el algoritmo será grande.

5 Implementaciones de los algoritmos

5.1 Estructura del proyecto

Para implementar los algoritmos hemos usado 3 archivos distintos:

- ***main.cpp*** **:** Este fichero contiene la implementación de el algoritmo por fuerza bruta más eficiente. Se compila en el archivo BFmain.bin cuando se ejecuta make build.
- ***main_ineficiente.cpp*** **:** Este fichero contiene la implementación de el algoritmo por fuerza bruta poco eficiente. Se compila en el archivo BFmainInefficiente.bin cuando se ejecuta make build.
- ***mainPD.cpp*** **:** Este fichero contiene la implementación de el algoritmo basado en programación dinámica. Se compila en el archivo DPmain.bin cuando se ejecuta make build.

Además, hemos incluido 3 archivos .stocks ejemplo para poder dárselos como entrada a los programas. Estos archivos tienen los siguientes nombres, ordenados de menos complejos a más:

- pruebaPapel.stocks
- prueba.stocks
- prueba1.stocks

5.2 Compilación y ejecución de los programas

Para facilitar la compilación y ejecución del programa hemos creado un **makefile** con algunos comandos para compilar y ejecutar un ejemplo sencillo. Estos comandos son los siguientes:

1. **make build**: compila los 3 archivos .cpp y crea sus respectivos ejecutables .bin.
2. **make clear**: elimina todos los archivos .bin de el directorio.
3. **make BFrun**: compila y ejecuta un ejemplo de la implementación de el algoritmo de fuerza bruta eficiente
4. **make BFrunInefficient**: compila y ejecuta un ejemplo de la implementación de el algoritmo de fuerza bruta ineficiente.
5. **make DPrun**: compila y ejecuta un ejemplo de la implementación de el algoritmo basado en programación dinámica, mostrando en cada paso la tabla actualizada.

5.3 Ejecuciones de los programas

Para la ejecución de ejemplo usaremos un presupuesto de 20 y el archivo prueba.stocks.

Se puede ver que la implementación de el algoritmo por fuerza bruta ineficiente tiene un tiempo de ejecución malísimo mientras que los otros dos programas tienen un tiempo de ejecución más considerable.

```
alexrp@pop-os:~/Documentos/ALG/algoritmica/p4i$ ./BFmainIneficiente.bin 20 prueba.stocks
Array: 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 2 2 2 2 2
Tenemos 4.1943e+06 combinaciones posibles
Mejor compra: 1 1 1 1 1 1 1 1 2 2 2 2
Coste: 19.16
Beneficio: 1.72
Tiempo de ejecución: 4.83813
```

Figure 1: Ejecución del programa ineficiente con un presupuesto de 20 y el archivo de empresas prueba.stocks

```
alexrp@pop-os:~/Documentos/ALG/algoritmica/p4i$ ./BFmain.bin 20 prueba.stocks
Total de combinaciones: 478
Mejor compra: 0 7 4
Coste: 19.16
Beneficio: 1.72
Tiempo de ejecución: 0.000229
```

Figure 2: Ejecución del programa eficiente con un presupuesto de 20 y el archivo de empresas prueba.stocks

```
-------------------------
Int.    0       2       4       6       8       10      12      14      16      18      20
0       0       0.115   0.345   0.46    0.575   0.575   0.575   0.575   0.575   0.575   0.575
1       0       0.115   0.345   0.525   0.705   0.885   1.065   1.245   1.425   1.605   1.72
2       0       0.115   0.345   0.525   0.705   0.885   1.065   1.245   1.425   1.605   -1
-------------------------
Int.    0       2       4       6       8       10      12      14      16      18      20
0       0       0.115   0.345   0.46    0.575   0.575   0.575   0.575   0.575   0.575   0.575
1       0       0.115   0.345   0.525   0.705   0.885   1.065   1.245   1.425   1.605   1.72
2       0       0.115   0.345   0.525   0.705   0.885   1.065   1.245   1.425   1.605   1.72
-------------------------
Tiempo de ejecución: 0.000561
Mejor compra: 0 7 4
Coste: 19.16
Beneficio: 1.72
```

Figure 3: Ejecución del programa basado en Programación Dinámica con un presupuesto de 20 y el archivo de empresas prueba.stocks, donde se muestran la tabla en las dos últimas iteraciones.