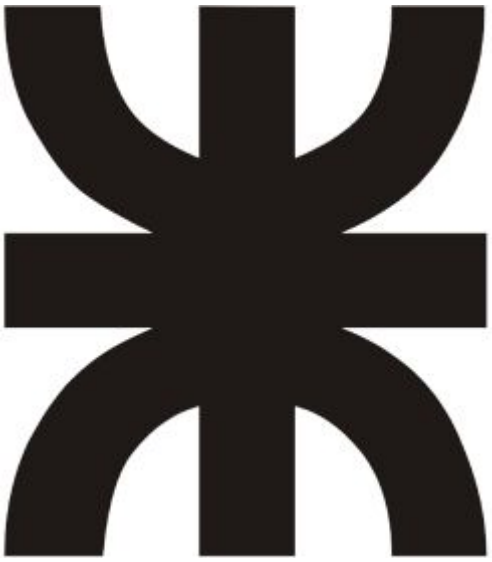
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL RESISTENCIA



INGENIERÍA EN SISTEMAS DE INFORMACIÓN

COMPLEJIDAD Y TÉCNICAS DE DISEÑO DE ALGORITMOS

TRABAJO PRÁCTICO INTEGRADOR

Profesor de Teoría: Ing. Acuña Cesar Javier
Profesor de Práctica: Ing. Tortosa Nicolás

Grupo 4:

-ACEVEDO, Fernando Enrique
-ACOSTA, Gastón Eduardo
-GETZEL, Martin Exequiel
-LUCAS, Dania
-NAVARRO, Victoria Cecilia

2020

Introducción

En este trabajo se pretende hallar la solución al problema “Intereses Bancarios” aplicando las distintas técnicas de diseño de algoritmos aprendidas durante la cursada de la materia, para luego compararlas y así poder elegir cuál es la más eficiente para este problema. Dichas soluciones serán programadas en el lenguaje Python versión 3.7, en las plataformas Visual Studio Code y Geany.

El problema propuesto consiste en que se tiene una cantidad de dinero a invertir M, de la cual se desea obtener el máximo beneficio. Para ello se dispone de ofertas de n bancos, correspondientes cada una de ellas a una función de interés fi (x) (i = 1..n) dependiente del capital invertido, que representa la cantidad de dinero que se obtiene al invertir en el banco i la cantidad x. Las funciones de interés cumplen que si x > y, entonces fi (x) ≥fi (y). En general no existe una oferta mejor que ninguna otra para cualquier cantidad, por lo que el beneficio máximo deberá buscarse en un reparto adecuado de la cantidad total a invertir entre todas las ofertas.

Hipótesis

El problema podrá ser resuelto mediante la aplicación de las técnicas de *programación dinámica* y *backtracking*. Sin embargo la técnica de *programación dinámica* será la que resuelva la problemática de manera más eficiente con respecto al coste computacional. A pesar de que hará un mayor uso de memoria que el resto de técnicas, lo compensará con un eficiente cálculo, hallando la solución en el mejor tiempo posible.

La técnica de *algoritmos ávidos* no logrará maximizar la función requerida en todos los casos, por lo cual, no dará solución a la problemática.

Desarrollo

Consigna formal

Dadas n funciones f1, f2, ..., fn y un entero positivo M, deseamos maximizar la función f1(x1) + f2(x2) + ... + fn(xn) sujeta a la restricción x1 +x2 + ... + xn = M, donde fi(0) = 0; (i=1,..,n), xi son números naturales, y todas las funciones son monótonas crecientes, es decir, x ≥ y implica que fi(x) > fi(y). Supóngase que los valores de cada función se almacenan en un vector. Este problema tiene una aplicación real muy interesante, en donde fi representa la función de interés que proporciona el banco i, y **lo que deseamos es maximizar el interés total** al invertir una cantidad determinada de dinero M. Los valores xi van a representar la cantidad a invertir en cada uno de los n bancos.

Programación Dinámica

Referencias:

fi= vectores que almacenan el interés del banco i
fi(xi)= función de interés que proporciona el banco i.
j=xi=cantidades de dinero invertidas en banco i (ej: 100$, 200€,etc).

m=cantidad Total determinada de dinero a invertir.
i=banco.
In(M)= interés máximo al invertir M dinero en **n** bancos.
F[i,j] representa el interés del banco i para j monedas.

Función a maximizar

In(M) = f1(x1) + f2(x2) + ... + fn(xn)

sujeta a la restricción x1+x2 + ... + xn = M.

Veamos cómo aplicar el principio de óptimo. Si In(M) es el resultado de una secuencia de decisiones y resulta ser óptima para el problema de invertir una cantidad M en n bancos, cualquiera de sus subsecuencias de decisiones ha de ser también óptima y así la cantidad.

In–1(M – xn) = f1(x1) + f2(x2) + ... + fn–1(xn–1) será también óptima para el subproblema de invertir (M – xn) pesetas en n – 1 bancos. Y por tanto el principio de óptimo nos lleva a plantear la siguiente relación en recurrencia:

$$I_n(x) = \begin{cases} f_1(x) & \text{si } n = 1 \\ \underset{0 \le t \le x}{Max}\{I_{n-1}(x-t) + f_n(t)\} & \text{en otro caso.} \end{cases}$$

Para resolverla y calcular In(M), vamos a utilizar una matriz I de dimensión nxM en donde iremos almacenando los resultados parciales y así eliminar la repetición de los cálculos. El valor de I[i,j] va a representar el interés de j pesetas cuando se dispone de i bancos, por tanto la solución buscada se encontrará en I[n,M]. Para guardar los datos iniciales del problema vamos a utilizar otra matriz F, de la misma dimensión, y donde F[i,j] representa el interés del banco i para j pesetas.

```
def intereses(F,I,n,m):
    for i in range (0,n):
        I[i][0]=0
    for j in range (m+1):
        I[0][j]=F[0][j]
    for i in range (1,n):
        for j in range (1,m+1):
            I[i][j]=IntMax(I,F,i,j)
    return(I[n-1][m])

def IntMax(I,F,i,j):
    maxim=I[i-1][j]+F[i][0]
    for t in range(1,j+1):
        maxim=max(maxim,I[i-1][j-t]+F[i][t])
    return (maxim)
```

```
def crearMatriz(m,n):
   matriz = []
   for i in range(0,m):
       fila=[]
       for j in range (0,n):
           fila=fila[0:j]+[0]
       matriz=matriz[0:i]+[fila]
   return matriz
#-----------------------------------------------------------------------
bancos = int(input("Coloque la cantidad de bancos:"))
dinero = int(input("Coloque la cantidad de dinero:"))
matF=[[0,43,56,99],[0,32,79,80],[0,51,62,71]]
matI=crearMatriz(bancos,dinero+1)
resultado=intereses(matF,matI,bancos,dinero)
print("La ganancia maxima con ",dinero,"pesos será: ",resultado)
```

Conclusiones

En cuanto a esta técnica, podemos concluir en que fue difícil la identificación de una función recurrente que cumpla con lo requerido, y en comparación con otras técnicas, el código no es claro, es decir, difícil de comprender. La solución utilizando esta técnica suele exigir gran capacidad de memoria, al realizar el cálculo del beneficio para una cantidad M de dinero en n bancos, necesitamos utilizar $n * (M + 1)$ variables, que serían la cantidad de elementos de la matriz matI.

Backtracking

El problema busca obtener el máximo beneficio al invertir una cantidad **M** de dinero, en los **n** bancos. Al implementar la técnica de backtracking estaremos generando combinaciones hasta agotar las posibilidades (búsqueda exhaustiva), es decir, hasta que todo el dinero **M** sea invertido, de lo contrario se continuará generando nodos a través de la llamada recursiva **Ganancia()**.

A medida que se recorre el árbol se rellenará la lista **I** de tamaño **n** en la cual se indicará cuánto dinero se invertirá en cada banco. Cada vez que se alcanza una hoja del árbol (es decir, todo el dinero ya ha sido invertido), se controla si el valor obtenido es el máximo hasta el momento, una vez que se termine el recorrido la variable *"maximo"* tendrá almacenado el beneficio máximo para **M** dinero.

```
M=3 #PlataAInvertirTotal
n=3 #CantidadDeBancos
I=[0,0,0] #Cada posicion indica cuantos pesos invierto en cada banco
F=[[0,43,56,99],[0,32,79,80],[0,51,62,71]] #Interés que genera c/banco
por cada peso
maximo=0

def Ganancia(F,n,M,I):
    global maximo
    tot=0 #Cantidad de dinero invertido hasta el momento
    suma=0
    for i in range(0,n):
        tot=tot+I[i]

    if tot==M:
        for i in range(0,n):
            suma= (F[i][I[i]]) + suma
            if suma>maximo:
                maximo=suma

    else:
        for i in range(0,n):
            nuevoI=I[:] #porCopia
            nuevoI[i]=nuevoI[i]+1
            Ganancia(F,n,M,nuevoI)
    return maximo
#----------------------------------------------------------------------
print("La ganancia maxima con ",M,"pesos será: ",Ganancia(F,n,M,I))
```

Conclusiones

La aplicación de la técnica de Backtracking para la resolución de este problema, prioriza la complejidad espacial antes que la temporal, por lo cual tarda más tiempo en ejecutarse. Al realizar el cálculo de interés para una cantidad de n bancos y una cantidad M de dinero, nos llevaría en backtracking a recorrer n^M nodos, es decir, a medida que aumente la cantidad de bancos y/o dinero a invertir, la cantidad de nodos aumentará notoriamente, y por ende la cantidad de cálculos a realizar también. Sin embargo, es una técnica muy fácil de implementar, una vez que se logra entender cómo recorrer el espacio del problema, lo demás se vuelve sencillo de realizar. Como producto final se obtiene un código muy simple debido a la utilización de llamadas recursivas.

Algoritmos ávidos

Para este problema la implementación de un algoritmo ávido no es factible, basándonos en la siguiente demostración:

Dados dos bancos B1 y B2, con sus respectivas funciones de interés f_1 y f_2, que generen los siguientes vectores:

$$f_1 = \{f_1(0), f_1(1), f_1(2), ..., f_1(m)\}$$
$$f_2 = \{f_2(0), f_2(1), f_2(2), ..., f_2(m)\}$$

Donde

$$f_1(x_n) > f_1(x_n - 1)$$
$$f_2(x_n) > f_2(x_n - 1)$$
$$f_1(0) = f_2(0) = 0$$

La función de selección que utilizaremos es aquella que para un valor de x, elige el $Max_{1 \le i \le n} \{f_i(x)\}$.

Un candidato es factible cuando el valor del dinero invertido (x) no supera la cantidad de dinero disponible para invertir (m).

Tomando los candidatos de la solución de menor a mayor con $m = 2$, nos podremos encontrar con la siguiente situación:

- Supongamos que $f_1(1) > f_2(1)$, nuestra función de selección tomará $f_1(1)$. Este candidato es un candidato factible, ya que $x < m$.
  Ahora $m' = m - x = 1$.
  En nuestro conjunto solución hasta ahora solo tenemos $[f_1(1)]$.
- Ahora visualizamos que con una inversión de 2 pesos podríamos tener que $f_2(2) > f_1(2) \wedge f_2(2) > f_1(1) + f_2(1)$, sin embargo como ya invertimos 1 peso en el banco B1, no podremos invertir los 2 pesos en el banco B2, por lo tanto la opción obtenida en el paso anterior no es un óptimo global.

Tomando los candidatos de la solución de mayor a menor con m>1, puede darse la situación de que al comenzar invirtiendo todo el dinero en un único banco, el interés obtenido sea menor que el que obtendríamos al dispersar esa inversión en más de un banco, en este caso con dos bancos:

$$f_1(m) < f_1(i) + f_2(m-i);\ con\ 1 \leq i < m$$

$$f_2(m) < f_1(i) + f_2(m-i);\ con\ 1 \leq i < m$$

Como puede verse en el ejemplo anterior, los algoritmos ávidos no serían capaces de encontrar un óptimo global, ya que en cada paso que demos hacia la solución nos podríamos encontrar con la situación mencionada.

El ejercicio pide conseguir la cantidad máxima de intereses que obtendría invirtiendo M pesos en n bancos, lo cual requiere que la solución sea el óptimo global, y con algoritmos ávidos no siempre encontraremos este óptimo global, por lo que no es factible la utilización de esta técnica de diseño para solucionar el problema.

Conclusión final

Como conclusión final, podemos decir que, a pesar de que cada técnica tiene sus ventajas y desventajas, la elección de una técnica por sobre las demás depende de la estructura de la problemática a resolver. En nuestro caso, la técnica que encontramos más óptima para resolver el problema planteado, es la técnica de programación dinámica cómo se expuso en la hipótesis, la cual a pesar de utilizar más recursos físicos del sistema que se ejecute, es la técnica que nos devolverá la solución más eficiente del mismo.

Dada la situación específica del problema requiere que la solución sea moderadamente rápida, en el caso de la utilización de backtracking el tiempo de ejecución aumenta notablemente, en cambio con programación dinámica preferimos optimizar la complejidad temporal sobre la espacial ya que hoy en día no es una preocupación la cantidad de memoria utilizada.

Para finalizar, en cuanto a la técnica de greedy, se pudo observar mediante la demostración expuesta que se cumple lo supuesto en la hipótesis ya que la función de selección no siempre encuentra el valor máximo.