Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ingeniería Industrial y de Sistemas
ICS3105 Optimización Dinámica (2018-2)
Profesor Mathias Klapp

Tarea 1

Ignacio Guridi (iguridi@uc.cl) - Raimundo Herrera (rjherrera@uc.cl)

1 Knapsack

Para resolver el problema de la mochila o *Knapsack*, se procedió con los dos métodos propuestos, por un lado `Python + Gurobi` y por otro lado programación dinámica en `Python`.

Para todos las ejecuciones el computador utilizado fue un MacBookPro que tiene las siguientes características:

Componente	Detalle
CPU	Core i5 @ 3.1 GHz
RAM	16 GB LPDDR3 @ 2133 MHz
GPU	Intel Iris Graphics 550 1536 MB
SSD	256 GB @ 3 GB/s

Se especifica la unidad de estado sólido puesto que el sistema de manejo de memoria RAM de `macOS` utiliza el disco eficientemente como unidad de *swap* cuando se ejecutan programas intensivos en memoria.

Por otro lado, cabe señalar que se modificó levemente el archivo `.xlsx` de instancias puesto que tenía un salto de línea extra en la instancia difícil que hacía menos generalizable la lectura del mismo.

Para efectos de las siguientes secciones, no se incluirá código de *parseo* de las instancias, pero cabe señalar que se programó un módulo que permite, a partir del nombre de la instancia deseada, obtener el n y B del problema, y los volumenes y valores de cada objeto.

En adelante, con la intención de no incluir código innecesario, se mostrarán fragmentos del código que, gracias a la similitud de `Python` con pseudocódigo, permiten entender lo realizado sin añadir ruido al informe.

1.1 Python + Gurobi

En esta sección, el código utilizado es bastante básico, y ejemplifica lo simple de utilizar la librería propuesta. Se incluye a continuación la función principal que resuelve el problema.

```
def knapsack(n, B, volumes, values, binary=True):
  n = [i for i in range(n)]
  model = Model('Knapsack')

  x = model.addVars(n, vtype=GRB.BINARY if binary else GRB.INTEGER, name='x')
  model.addConstr(quicksum(volumes[p] * x[p] for p in n) <= B)
  obj = quicksum(values[p] * x[p] for p in n)

  model.setObjective(obj, GRB.MAXIMIZE)
  model.optimize()
  return [i.X for i in model.getVars()]
```

El código anterior, en palabras lo que hace es declarar un modelo de tipo Knapsack, agregar las variables binarias correspondientes a los objetos, agregar la restricción de volumen, especificar la función objetivo y resolver. Finalmente retorna los valores de cada variable.

Para cada instancia se ilustran a continuación los valores objetivo, volumen alcanzado, tiempo de ejecución promedio y parámetros de entrada:

Instancia	n	B	Valor	Volumen	Tiempo
Fácil	50	995	8373	971	0.01s
Normal	1000	23827	21627	23827	0.03s
Difícil	2000	97258	174231	97258	0.05s
Desafío	10000	16693224	15051142	16693142	0.13s

El promedio corresponde a 5 ejecuciones del programa, y como se puede apreciar, los tiempos son extremádamente bajos, debido a las optimizaciones que tiene implementadas el solver utilizado, y al ser este problema uno extramemadamente estudiado.

1.2 `Python + DP`

En esta sección se implementó una versión muy sencilla de programación dinámica, basándose en las diapositivas de clases. El programa consiste fundamentalmente en dos `for` anidados, uno que va desde la cantidad de objetos a 0, y otro interior que recorre el volumen desde 0 a B.

Para ir almacenando los subproblemas, se utilizó una matriz `V`, de la forma de una lista bidimensional, donde el elemento óptimo `V[k][b]` se corresponde con el $V_k(b)$ del material de clases.

El código es el siguiente:

```
def binary_knapsack(n, B, volumes, values):
    V = [[0 for _ in range(B + 1)] for _ in range(n)] # matriz de subproblemas
    for i in range(B + 1): # valores terminales
        V[n - 1][i] = values[n - 1] if volumes[n - 1] <= i else 0

    for k in range(n - 2, -1, -1): # iteracion sobre objetos
        # (*) optimizacion de asignaciones
        vol_k = volumes[k]
        val_k = values[k]
        next_k = k + 1
        for j in range(0, min(vol_k, B + 1)): # (**) poda de minimo volumen
            V[k][j] = V[next_k][j]
        for i in range(vol_k, B + 1): # iteracion sobre volumen
            V[k][i] = max(V[next_k][i], val_k + V[next_k][i - vol_k])
    return V
```

De la matriz retornada, cabe indicar que para obtener el valor objetivo óptimo es necesario acceder al valor de `V[0][B]`.

Se añadieron ciertos comentarios para facilitar la comprensión. La primera consideración `(*)` es que en un programa minimizar las asignaciones y cálculos puede ser crucial, sobretodo cuando la cantidad de iteraciones es muy grande. Por este motivo, en vez de calcular el elemento `values[k] + V[k + 1][i - volumes[k]]`, correspondiente a la elección de considerar el elemento actual, sumando el valor de añadirlo y el valor óptimo de la matriz con el volumen reducido en el volumen del objeto, se decide asignar afuera del segundo `for` las variables. Esto consigue reducciones de un 30% en el tiempo de ejecución.

Una poda implementada para evitar cálculos innecesarios es la que se señala en `(**)`. Esta evita revisar la condición sobre el volumen del objeto que verifica si este es inferior al volumen b. Lo que se hace es definir los valores de la matriz inmediatamente como el valor de no añadir el objeto, puesto que al no caber el

objeto, no hay necesidad de chequearlo nuevamente, con esto, el `for` del volumen, comienza en el volumen del objeto y no en 0, y no realiza el chequeo de volumen dentro del ciclo.

Por otro lado, a diferencia del *solver* de Gurobi, obtener aquellos objetos o bien los índices de los objetos que se incluyeron en el problema, no viene resuelto inmediatamente, por lo tanto, se tiene que obtener el detalle de la solución a partir de la matriz de subproblemas. Para esto, se implementó la siguiente función:

```
def solution_from_matrix(n, B, volumes, values, V):
    objects = []
    value = V[0][B]
    for i in range(n - 1):
        if value <= 0:
            break
        if value != V[i + 1][B]:
            objects.append(i)
            value = value - values[i]
            B = B - volumes[i]
    return objects
```

Esta función lo que hace es iterar sobre cada fila de la matriz, buscando las diferencias entre fila y fila para un mismo volumen, de modo de saber cuando se añadió un objeto. Esto viene de la observación de que si no se añadió un objeto, el valor de `V[i][b]` es igual al de `V[i + 1][b]` para un mismo b, y es distinto para el caso contrario. Por lo que la iteración consiste en agregar un objeto a la lista de considerados sólo cuando hay diferencias entre filas, y volver a realizar el procedimiento para el volumen reducido en el volumen del objeto añadido. Se comprobó la correctitud del algoritmo teóricamente con lo explicado y empíricamente comparando los resultados con los obtenidos en la sección anterior con Gurobi.

En cuanto a los resultados obtenidos, se ilustran en la tabla a continuación. La instancia de desafío se intentó realizar con diferentes podas, y optimizaciones al código, como por ejemplo ordenar por volumen creciente los objetos e invertir el orden de los ciclos, de modo de, pasado cierto volumen de producto, dejar de calcularlo ya que ese volumen no cabría, o bien, la que se explicó anteriormente, pero ninguna de estas optimizaciones fue suficiente para poder computar el problema.

El programa empezó a utilizar mucha memoria de *swap*, que corresponde a utilizar espacio en el disco cuando la memoria RAM no alcanza, sin embargo tampoco fue suficiente. Es evidente que esto se debe a los cómputos a realizar, puesto que con un $n = 10000$ y $B = 16693224$, la cantidad de cómputos tiene como cota inferior $nB = 166932240000$, lo que, por la implementación, requiere de una cantidad de memoria exorbitante.

En la tabla a continuación se ilustran los tiempos de cómputo de las 3 instancias, donde los resultados óptimos son evidentemente los mismos. Cabe señalar que los tiempos son bastante superiores a los de Gurobi, y creemos que esto se debe principalmente a las optimizaciones que incluye la librería y el enfoque *memory-intensive* de nuestra implementación en DP.

Instancia	n	B	Valor	Volumen	Tiempo
Fácil	50	995	8373	971	0.02s
Normal	1000	23827	21627	23827	10.47s
Difícil	2000	97258	174231	97258	88.62s
Desafío	10000	16693224	-	-	-

Donde los tiempos corresponden al promedio de 5 ejecuciones.

1.3 Primeros elementos fijos

Para esta sección cabe considerar que es posible y muy útil usar los resultados obtenidos en la sección anterior para computar esta nueva versión. El caso particular de introducir 2 objetos fijos a la mochila

implica reducir el volumen disponible en los volumenes de cada objeto, y luego considerar los valores ya calculados para este nuevo volumen disponible.

Sean v_1, v_2 los volúmenes de los primeros dos objetos y w_1, w_2 los beneficios, este nuevo problema será simplemente obtener el valor del problema de la mochila con los productos $\{3, ..., n\}$, sumándole el beneficio de los dos primeros productos.

Uno podría pensar que existe un problema: quizás el subproblema de volumen $B - v_1 - v_2$ podría haber incluido en su solución óptima a estos dos productos. Sin embargo, la secuencialidad de las elecciones de los productos asegura que esto no puede haber ocurrido.

Pensemos que, por ejemplo, $B - v_1 - v_2 = B - v_3$. En ese caso, no podrá haber incorporado a 1 ni a 2, ya que ya se tomó la decisión de no incorporarlo. Esto se da porque el problema señala la incorporación de los dos primeros objetos, no de dos objetos cualquiera, y la forma en la que se arma la matriz de valores óptimos, garantiza la secuencialidad de la solución, y evita el cuestionamiento de repetir alguno de los dos primeros objetos.

1.4 Repeticiones

En esta variante del problema, se decidió resolver con ambas implementaciones, tanto en Gurobi como con programación dinámica.

1.4.1 `Python + Gurobi`

Para la implementación con Gurobi, la única modificación a realizar es la de modificar el tipo de variable de cada objeto, desde binaria a entera. En el código incluido en la primera sección se añadió una variable booleana en la función que indica si el problema será resuelto binariamente o enteramente, de modo que para ejecutar el código se debe llamar con `True` o `False`. Con esto se obtienen resultados similarmente rápidos y se resumen en la siguiente tabla:

Instancia	n	B	Valor	Volumen	Tiempo
Fácil	50	995	87010	971	0.002s
Normal	1000	23827	21627	23827	0.03s
Difícil	2000	97258	5770404	97258	0.04s
Desafío	10000	16693224	15174216	16693142	0.12s

1.4.2 `Python + DP`

Para modelar este problema como un problema de programación dinámica nos dimos cuenta que se podía modelar idénticamente pero en vez de añadir cada objeto i una vez, se podía elegir añadirlo hasta $B//v_i$ veces.

De este modo, el problema ya no solo considera añadir un objeto una vez, sino que potencialmente muchas veces. Así, la expresión recursiva consiste de:

$$V_k(b) = \begin{cases} max(w_k \cdot j + V_{k+1}(b - v_k \cdot j)) & \text{si } v_k \cdot j \geq b, \text{ con } j \in [0, B//v_k] \\ V_{k+1}(b) & \text{si } v_k < b \end{cases}$$

Y en cuanto a la modelación DP, con respecto a la modelación original propuesta en clases, se mantienen: el espacio de estados, etapas de decisión y función de transición de estados. Se modifican:

- Espacio de decisión por etapa y estado:

$$\mathbb{X}_k = \begin{cases} \{0\} & \text{si } v_k < b \\ \{j\} & \text{si } v_k \cdot j \geq b, \text{ con } j \in [0, B//v_k] \end{cases}$$

- Valor inmediato de etapa k al estar en un estado y tomar una cierta decisión:

$$c_k = \begin{cases} 0 & \text{si } x_k = 0 \\ x_k \cdot w_k & \text{e.o.c} \end{cases}$$

Esta modelación considerando exactamente las mismas consideraciones del problema original, con los mismos valores terminales y obteniendo el resultado en la misma forma, sugiere un programa así:

```
def integer_knapsack(n, B, volumes, values):
  V = [[0 for _ in range(B + 1)] for _ in range(n)]
  for i in range(B + 1):
      V[n - 1][i] = values[n - 1] if volumes[n - 1] <= i else 0

  for k in range(n - 2, -1, -1):
      vol_k = volumes[k]
      val_k = values[k]
      next_k = k + 1
      for j in range(0, min(vol_k, B + 1)):
          V[k][j] = V[next_k][j]
      for i in range(vol_k, B + 1):
          candidates = [V[next_k][i]]
          for j in range(0, (B + 1) // vol_k + 1):
              if vol_k * j <= i:
                  candidates.append(val_k * j + V[next_k][i - (vol_k * j)])
          V[k][i] = max(candidates)
  return V
```

Que funciona, pero está lejos de ser una implementación eficiente para el problema. Tras analizar el problema, es posible llegar a la conclusión de que no se necesita recorrer todos los productos ni generar una matriz bidimensional. Esto se concluye al notar que el problema es equivalente a poder elegir cualquier producto para cualquier volumen, y manteniendo la noción de subproblema óptimo, basta considerar únicamente cada volumen entre 0 y B, y computar el óptimo para cada caso.

Dicho de otro modo, el problema cuando hay repetición de productos es más simple que el binario, ya que el estado ya no requiere la información del volumen restante y los productos ingresados, sino que sólo el volumen. Con esto, el valor de un estado se calcula como el máximo entre los valores de ingresar cada producto más el valor óptimo del volumen restante, que debido al orden de iteración fue necesariamente calculado anteriormente. En código se ve así:

```
def integer_knapsack(n, B, volumes, values):
  V = [0 for _ in range(B + 1)]
  for i in range(B + 1):
      V[i] = values[n - 1] if volumes[n - 1] <= i else 0

  for i in range(B + 1):
      V[i] = max(values[k] + V[i - volumes[k]] if volumes[k] <= i else 0
                 for k in range(n))
  return V
```

Donde como se explicó, se calcula el óptimo para cada volumen en base a añadir el máximo producto posible. Notar que la expresión `if volumes[k] <= i else 0` es para cada valor de k y con esto la elección de un máximo se da entre los objetos que efectivamente podrían caber en la mochila dado el volumen i.

Reduciendo significativamente los tiempos de cómputo, incluso cabe notar, que con la simplificación anterior se consiguen tiempos de cómputo inferiores al problema binario y también que la memoria utilizada se reduce notoriamente. Por completitud se añaden los resultados en la siguiente tabla:

Instancia	n	B	Valor	Volumen	Tiempo
Fácil	50	995	87010	971	0.02s
Normal	1000	23827	21627	23827	7.28s
Difícil	2000	97258	5770404	97258	52.58s
Desafío	10000	16693224	15174216	16693142	∼10.1hrs

Cabe señalar que como el programa era mucho más óptimo en términos de memoria, se pudo computar el resultado de la instancia de desafío con uso intensivo de CPU, esto es, dejando correr el programa muchas horas, lo que no implica hacer colapsar el computador, ya que el uso de memoria se mantiene relativamente acotado, por el uso de una lista de una y no dos dimensiones.

2 Torre de cajas

2.1 Sin rotación

$$Y_{j,k} = \begin{cases} 1 & \text{si el elemento } j \text{ está en la posición } k \text{ de la torre} \\ 0 & \text{e. o. c} \end{cases}$$

Función objetivo:

$$\max_{Y \in \{0,1\}} \sum_{j=1}^{n} \sum_{k=1}^{n} h_j \cdot Y_{j,k}$$

R1: *máximo una caja en cada posición*

$$\sum_{j=1}^{n} Y_{j,k} \leq 1,\ k = 1, \ldots, n$$

R2: *el lado mayor de una caja debe ser menor al lado mayor de la caja debajo de ella*

$$\max\{l_j, a_j\} \cdot Y_{j,k} \cdot Y_{i,k-1} \leq \max(l_i, a_i),\ i = 1, \ldots, n,\ j = 1, \ldots, n,\ k = 2, \ldots, n$$

R3: *el lado menor de una caja debe ser menor al lado menor de la caja debajo de ella*

$$\min(l_j, a_j) \cdot Y_{j,k} \cdot Y_{i,k-1} \leq \min(l_i, a_i),\ i = 1, \ldots, n,\ j = 1, \ldots, n,\ k = 2, \ldots, n$$

DP

Se debe reordenar las cajas por el tamaño de su base, de forma que $l_k \cdot a_k \leq l_{k+1} \cdot a_{k+1},\ k = 1, \ldots, n-1$

Valor terminal para $l_b, a_b \in S$. Se aprovecha la definición de Valor Inmediato definido más adelante, para evitar redundancia

$$V_n(l_b, a_b) = \max\{c_n\}$$

Recursión. Se utiliza las definiciones de Función de Transición, Espacio de Desición y Valor Inmediato, definidos en el apartado subsiguiente.

$$V_k(l_b, a_b) = \max\{V_{k+1}(f(l_a, l_b)) + c_k\}$$

Nos interesa $V_1(B)$, en donde $k = 1$ corresponde al cubo con base más pequeña

- *Etapas.*

$$1, \ldots, n$$

- *Espacio de Estados.* Corresponden a las combinaciones de enteros con dos dimensiones entre 0 y el lado mayor del cubo más grande en tamaño

$$S = \{(x, y) : 0 < x \leq \max(l_i, a_i), 0 \leq y \leq \max(l_i, a_i)\},\ i = 1, \ldots, n$$

- *Espacio de Decisión* por etapa y estado:

$$X_k((l_b, a_b)) = \begin{cases} \{0\} & \text{si} \quad (l_k > l_b \wedge l_k > a_b) \vee (a_k > l_b \wedge a_k > a_b) \\ \{0, 1\} & \text{e. o. c.} \end{cases}$$

- *Función de Transición* desde el estado b en la etapa k. Para el problema, es indiferente el orden $((l_k, a_k) vs. (a_k, l_k))$ de las dimensiones al copiarlas para el siguiente estado, pues solo nos preocupa la base de la última caja

$$b : \text{estado actual}$$
$$l_{k+1}, a_{k+1} = f(l_b, a_b) = (l_b, a_b) \cdot (1 - X_k) + (l_k, a_k) \cdot X_k$$

- *Valor Inmediato* de la etapa k al estar en un estado b y tomar una decisión X_k:

$$c_k = \begin{cases} 0 & \text{si } X_k = 0 \\ h_k & \text{e. o. c.} \end{cases}$$

- *Cost-to-go.* Corresponde a sumar el valor inmediato de tomar una desición en una etapa con el valor del cost-to-go de la siguiente etapa. Nos sirve el cost-to-go de la etapa inicial, pues nos entregará el resultado dado por el óptimo del problema

$$C_k = \max_{x_k \in X_k} \{c_k(x_k) + C_{k+1}(x_k)\},\ k = 1, \ldots, n-1$$

2.2 Con rotación

Se debe definir la variable extra para este caso, Z_j, que señala de qué forma está orientada la caja

$$Y_{j,k} = \begin{cases} 1 & \text{si el elemento } j \text{ está en la posición } k \text{ de la torre} \\ 0 & \text{e. o. c} \end{cases}$$

$$L_j = \begin{cases} 1 & \text{si la dimensión } l_j \text{ hace las veces de la altura del elemento } j \\ 0 & \text{e. o. c} \end{cases}$$

$$A_j = \begin{cases} 1 & \text{si la dimensión } a_j \text{ hace las veces de la altura del elemento } j \\ 0 & \text{e. o. c} \end{cases}$$

$$H_j = \begin{cases} 1 & \text{si la dimensión } h_j \text{ hace las veces de la altura del elemento } j \\ 0 & \text{e. o. c} \end{cases}$$

Función objetivo: Se le añadió variables de forma que el modelo solo reconozca la altura de cada caja en el cálculo de la función objetivo.

$$\max_{Y \in \{0,1\}} \sum_{j=1}^{n} \sum_{k=1}^{n} (h_j H_j + l_j L_j + a_j A_j) Y_{j,k}$$

Se modifican las restricciones, de forma de representar el grado de libertad extra por poder rotar las cajas

R1: *máximo una caja en cada posición*

$$\sum_{j=1}^{n} Y_{j,k} \leq 1,\ k = 1, \ldots, n$$

R2: *el lado mayor de de la base de cada caja debe ser menor al lado mayor de la caja debajo de ella.* Se multiplica por un factor de forma que la dimensión que actúa como altura se vaya a 0 y no afecte la comparación

$$\max(h_j \cdot (1 - H_j), l_j \cdot (1 - L_j), a_j \cdot (1 - A_j))Y_{j,k} \cdot Y_{i,k-1} \leq \max(l_i, a_i),\ i = 1, \ldots, n,\ j = 1, \ldots, n,\ k = 2, \ldots, n$$

R3: *el lado menor de la base de cada caja debe ser menor al lado menor de la caja debajo de ella.* Se suma a cada dimensión de la caja un número muy grande M, que solo está presente en la dimensión que es altura, de forma que no interfiera con la restricción.

$$\min(h_j + M \cdot H_j, l_j + M \cdot L_j, a_j + M \cdot A_j)Y_{j,k}Y_{i,k-1} \leq \min(l_i, a_i),\ i = 1, \ldots, n,\ j = 1, \ldots, n,\ k = 2, \ldots, n$$

R4: *hay una y solo una altura por caja*

$$H_j + L_j + A_j = 1 \leq 1,\ j = 1, \ldots, n$$

DP

El planteamiento como problema de DP corresponde al mismo que para el caso sin rotación, debido a que está en función de los ingredientes del problema. Nos aprovechamos de esto para modificar únicamente estos ingredientes (*Etapa*, *Valor Inmediato*, etc.) para evitar redundancia.

- *Etapas.*

$$1, \ldots, n$$

- *Espacio de Estados.* Corresponden a las combinaciones de enteros con dos dimensiones entre 0 y el lado mayor del cubo más grande en tamaño, considerando sus 3 dimensiones

$$S = \{(x, y) : 0 < x \leq \max(h_i, l_i, a_i), 0 \leq y \leq \max(h_i, l_i, a_i)\},\ i = 1, \ldots, n$$

- *Espacio de Decisión* por etapa y estado. Ahora se tiene otro grado de libertad, por lo que hay más opciones para colocar una caja. Se toma en consideración que el estado actual b solo tiene dos dimensiones, pues corresponde a una caja que ya fue colocada.

$$X_k((l_b, a_b)) = \begin{cases} \{0\} & \text{si} \quad (l_k > l_b \wedge l_k > a_b) \vee (a_k > l_b \wedge a_k > a_b) \vee (h_k > l_b \wedge h_k > a_b) \\ \{0, 1\} & \text{e. o. c.} \end{cases}$$

- *Función de Transición* desde el estado b en la etapa k. Ahora se deben manejar las dimensiones de forma que el siguiente estado corresponda con la nueva base de la caja que se coloca

$$b : \text{estado actual}$$

$$l_{k+1}, a_{k+1} = f(l_b, a_b) = (l_b, a_b) \cdot (1 - X_k) + (l_k, a_k) \cdot X_k \cdot H_k + (h_k, a_k) \cdot X_k \cdot L_k + (l_k, h_k) \cdot X_k \cdot A_k$$

- *Valor Inmediato* de la etapa k al estar en un estado b y tomar una decisión X_k:

$$c_k = \begin{cases} 0 & \text{si } X_k = 0 \\ h_k & \text{e. o. c.} \end{cases}$$

- *Cost-to-Go.* Corresponde a sumar el valor inmediato de tomar una desición en una etapa con el valor del *cost-to-go* de la siguiente etapa. Se quiere el *cost-to-go* de la etapa inicial, pues nos entregará el resultado dado por el óptimo del problema

$$C_k = \max_{x_k \in X_k} \{c_k(x_k) + C_{k+1}(x_k)\},\ k = 1, \ldots, n-1$$

La torre más alta debiese ser sin lugar a dudas la del problema que permite rotación. Esto es pues permite más libertad que el problema sin rotación, el cual se puede ver como que sus posibles construcciones son un subconjunto de las posibilidades de este otro. Visto de forma intuitiva, si se tiene una caja de largo muy grande, y profundidad y altura pequeños, en el primer problema es difícil que quepa sobre alguna caja o que sume mucha altura. Sin embargo, si se rota, su largo pasa a ser su altura, y queda con una base muy pequeña. Esta características son ideales para lograr torres altas, dentro de las restricciones del problema. En definitiva, solo se puede ganar con este extra grado de libertad que se tiene, por lo que el problema que permite rotación tendrá torres iguales o más altas que el otro, considerando las mismas cajas.

3 Recursión y subproblemas

Para este problema se intentará resolver un problema básico, pero que no cumple con el principio de Bellman, pues cualquier subsecuencia de la secuencia de desiciones que lo lleva a su solución óptima no es óptima con respecto al subproblema correspondiente.

Se tiene que cierta persona M. está viajando por un bosque. Tiene que tiene plata inicial C, y se encuentran 3 piedras en un camino, las que poseen un poder especial que transforma la cantidad de plata del que las posee, de acuerdo a la siguiente tabla

Piedra	Poder f($plata$)
1	$(plata)^2$
2	$(plata)^{-1}$
3	$(plata)^{-2}$

Si se intenta ayudar a M. a decidir que piedras llevarse, resulta evidente que la mejor opción es llevarse las 3, pues, si se escojen todas las piedras (no hay ninguna limitación frente a esto), se queda con que

$$\text{plata}^{2^{-1^{-2}}} = \text{plata}^4$$

Esto lo deja muy bien parado con respecto a su capital inicial, eligiendo todas las piedras, lo cual será representado con la secuencia $S_{\text{óptima}} = \{1, 1, 1\}$, en donde cada 1 señala que esa piedra si fue escogida. Sin embargo, si se intenta resolver este problema con PD, se llega a otro resultado.

Sea $k = 1, 2, 3$, y se sabe que el valor en la etapa k es V_k, siendo V_{k-1} correspondiente a la cantidad de plata que se tiene en el estado anterior

$$V_k = \max\{V_{k-1}, f_k(V_{k-1})\}$$

Entonces, si se inicia con un capital de valor $C > 1$, podemos hacer de forma manual cada etapa de la optimización.

Etapa (k)	Poder		Desición (X_k)	V_k
1	$()^2$	$\max\{C; C^2\}$	1	C^2
2	$()^{-1}$	$\max\{V_1; (V_1)^{-1}\}$	0	C^2
3	$()^{-2}$	$\max\{V_2; (V_2)^{-2}\}$	0	C^2

En este caso nos interesa el valor final de V_3, pues hicimos la optimización desde la etapa 1 a la n. Se puede ver que nos dio un resultado final de C^2, lo que está muy por debajo del óptimo que se encontró previamente, que es C^4.

Por lo tanto, este problema no se puede resolver con PD. esto se debe a que hay subsecuencias dentro de $S_{\text{óptima}}(\{1,1,1\})$ que no corresponden a óptimos dentro del subproblema respectivo. Si se ve por ejemplo la elección de las dos primeras etapas, se obtiene que la mejor opción es la que arroja el algoritmo de PD, la que sería {1,0}. Sin embargo, es inmediato ver que esta secuencia de desiciones no corresponde a ninguna subsecuencia dentro de $S_{\text{óptima}}$. Luego, he ahí la razón de por qué con este algoritmo no se llega al óptimo.