.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

4 de noviembre de 2023

Milena Marchese
100962

Gianni Boccazzi
109304

1. Introducción

El objetivo de este trabajo es resolver un problema mediante programación dinámica y analizarlo en profundidad.

Scaloni se enfrenta al desafío de planificar los próximos $\boldsymbol{n}$ entrenamientos de la selección Argentina. Cada entrenamiento requiere un esfuerzo $\boldsymbol{e_i}$, que también consideramos como una ganancia fija. Además, los jugadores tienen niveles de energía $\boldsymbol{s_i}$. Debido al esfuerzo físico requerido en cada entrenamiento, los jugadores comienzan con un nivel de energía de $\boldsymbol{s_i}$, y después de cada sesión, su nivel de energía para el próximo día será s_{i+1}, manteniendo la restricción $\boldsymbol{s_1} \geq \boldsymbol{s_2} \geq \ldots \geq \boldsymbol{s_n}$. Si Scaloni decide darles un día de descanso, el nivel de energía de los jugadores antes del próximo entrenamiento volverá a ser $\boldsymbol{s_1}$. Si el esfuerzo de un entrenamiento excede la energía disponible en ese día, la ganancia será igual a la energía disponible.

Como los entrenamientos son fijos, Scaloni debe decidir qué días la selección debe entrenar, a fin de obtener la mayor ganancia posible en los $\boldsymbol{n}$ días. El presente trabajo se enfoca en encontrar una solución óptima a este desafío mediante el uso de programación dinámica y proporcionar un análisis detallado del mismo. La resolución se encuentra en este repositorio [1] de GitHub.

2. Análisis e interpretación del problema

Consideraciones del problema:

- **La forma que tienen los subproblemas**: El subproblema es obtener la mayor ganancia para cierto día, teniendo disponible distintas cantidades de energía dependiendo qué días se descansa y se entrena.
- **La forma en que dichos subproblemas se componen para solucionar subproblemas más grandes**: Se tienen varias cantidades de energía que puedo usar dependiendo qué día se descansa. La mejor sería aquella que maximice la cantidad total. Se prueban todas y se obtiene el máximo.

Entonces para armar la ecuación de recurrencia vamos a contemplar las siguientes opciones:

- No se entrenó el día de ayer entonces el día de hoy estará disponible la energía s_1
- Se entrenó el día de ayer entonces la energía disponible es s_{act}

Luego, la ecuación de recurrencia será:

$$OPT(n, act) = \text{máx} \begin{cases} \text{mín}\{e_n, s_1\} + \text{máx}\{OPT[n-2]\} \\ \text{mín}\{e_n, s_{act}\} + OPT[n-1, act-1] \end{cases}$$

3. Algoritmo para obtener la solución óptima

Para resolver este problema a nivel de programación, se utiliza una matriz de dimensiones n×n. Cada celda de esta matriz contendrá la ganancia máxima que se puede obtener hasta el día i con un nivel de energía s_i.

A continuación el código en Python que resuelve este problema:

```python
def scaloni_dinamico(n, ganancia, energia):
    res = [[0] * n for _ in range(n)]
    res[0][0] = min(ganancia[0], energia[0])
    res[1][0] = min(ganancia[1], energia[0])
    res[1][1] = res[0][0] + min(ganancia[1], energia[1])
    for dia in range(2, n):
        res[dia][0] = max(res[dia-2]) + min(energia[0], ganancia[dia])
        for e in range(1, n):
            if e > dia:
                break
            res[dia][e] = res[dia-1][e-1] + min(ganancia[dia], energia[e])
    return res
```

Esta función devolverá la matriz con todos los resultados óptimos que se pueden obtener para cada día y energía. Se puede reconstruir la secuencia óptima de días entrenados y descansados con la siguiente función:

```python
def obtener_resultado(res, n):
  optimo = max(res[n - 1])
  orden = ['Descanso'] * n
  dia_actual = n - 1 # Filas
  energia_actual = res[n-1].index(optimo) # Columnas
  while dia_actual >= 0:
    for i in range (energia_actual + 1):
      orden[dia_actual] = 'Entreno'
      if (i != energia_actual):
         dia_actual -= 1
      if (dia_actual < 0):
        break
    dia_actual -= 2
    energia_actual = res[dia_actual].index(max(res[dia_actual]))
  return (optimo, orden)
```

El resultado de esta función será el plan de entrenamiento que maximiza la ganancia teniendo en cuenta las restricciones planteadas por el problema.

3.1. Seguimiento del algoritmo

Con los siguientes datos:

$E = [10, 5, 7, 4]$

$S = [9, 8, 6, 2]$

El resultado del algoritmo es:

$$\begin{bmatrix} 9 & 0 & 0 & 0 \\ 5 & 14 & 0 & 0 \\ 16 & 12 & 20 & 0 \\ 18 & 20 & 16 & 22 \end{bmatrix}$$

Siendo las filas los días y las columnas los niveles de energía disponible. Es por eso que en la primera fila, que representa el dia 1, solo se dispone de un nivel de energía (e_1). Asi es como el algoritmo completa la matriz de forma diagonal inferior.
Es fundamental destacar que cada celda de la matriz contendrá la mejor opción posible si se tuviera la energía de la columna correspondiente en el día de la fila correspondiente.

Una vez completada la tabla de ganancias máximas, la ganancia óptima del día n será el máximo de la fila de dicho día. Podemos observar que no es relevante para el resultado el óptimo local de la

ganancia, es por eso que partiendo desde el primer máximo según su nivel de energía reconstruimos cuantos dias para atras se entrenaron consecutivamente, es decir si el nivel de energía del dia s_4 es 4 entonces los días 1, 2 y 3 se entrenó con energías s_1, s_2 y s_3 respectivamente. Una vez que se llega al primer nivel de energía entonces, si sigue habiendo matriz, se considera que hubo un día de descanso en el medio y se repite la lógica de buscar el máximo y contar cuantos días hacia atrás se entrenaron con el dato de la energía del nuevo máximo.

3.2. Análisis del algoritmo

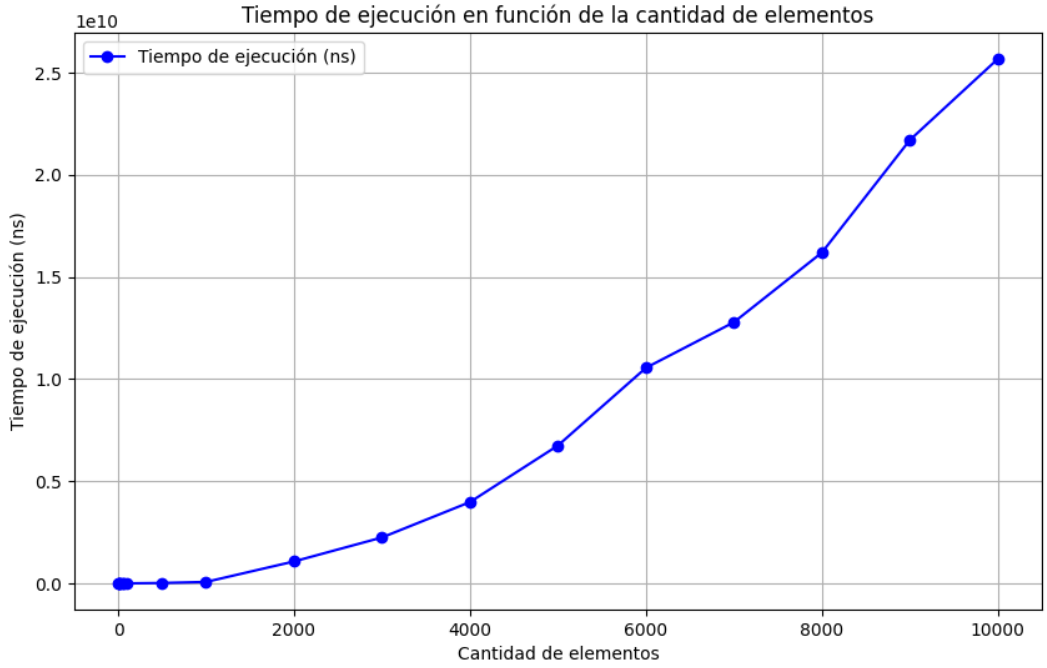
La complejidad temporal de este código es $O(n^2)$, ya que en primer lugar, la creación de la matriz requiere $O(n^2)$ operaciones. Las tres primeras asignaciones de valores iniciales a elementos de la matriz son operaciones de tiempo constante, por lo que no afectan la complejidad general. Luego, se itera sobre los días desde el tercer día hasta el día $n-1$, lo que implica $n-2$ iteraciones. Dentro de estas iteraciones, el cálculo del óptimo involucra la búsqueda del máximo en la fila $res[dia-2]$, lo que requiere examinar n elementos. El segundo bucle `for` e `in range(1, n)` itera sobre los niveles de energía desde el segundo nivel hasta el nivel $n-1$, realizando $n-1$ iteraciones. Dentro de este bucle, el cálculo del óptimo implica operaciones de tiempo constante. Por lo tanto, teniendo en cuenta que para obtener la solución es necesario calcular todos los subproblemas hasta completar la matriz la complejidad será de $O(n^2)$.
Respecto a la variabilidad de los valores, la complejidad es proporcional a la cantidad de posiciones de la matriz y el tamaño de la matriz solo depende de la cantidad n de días que disponga Scaloni para entrenar. Podemos afirmar que es de tiempo polinomial.
En cuanto a optimalidad, en caso de que haya un solo elemento entonces la máxima ganancia es el mín$\{\mathbf{e_1}, \mathbf{s_1}\}$ y es óptimo. Luego, para los $n \geq 1$, se deben tener en cuenta las decisiones tomadas en los días anteriores, desde el día 1 hasta el *n+1*, y las restricciones de energía disponibles en el día *n+1*, con lo cual el óptimo será el máximo entre el óptimo de haber entrenado el dia anterior y disponer de energia e_i y el óptimo de haber descansado el dia anterior y disponer de energia e_1. Entonces, el razonamiento inductivo demuestra que la solución propuesta es, de hecho, la mejor posible, ya que considera todas las decisiones anteriores y garantiza la máxima ganancia acumulada.

4. Ejemplos de ejecución y mediciones

Para estudiar el comportamiento de nuestro algoritmo respecto al tiempo presentamos las siguientes mediciones de ejemplos brindados por la catedra y propios:

Cantidad de elementos	Ganancia	Tiempo (ns)
3	7	2.000
10	380	12.000
10(bis)	523	11.000
10(entrena todo)	900	10.000
50	1.870	168.000
50(bis)	2.136	171.000
100	5.325	779.000
500	27.158	18.851.000
1.000	54.021	69.143.000
2.000	1.981.506	1.072.697.078
3.000	2.999.515	2.247.985.721
4.000	4.031.058	3.980.632.771
5.000	279.175	6.750.401.137
6.000	5.928.811	10.553.342.543
7.000	6.864.534	12.785.959.684
8.000	8.040.718	16.197.121.854
9.000	8.996.551	21.709.097.540
10.000	9.956.893	25.678.291.283

Gráficamente:



Finalmente, las mediciones de tiempos de nuestro algoritmo respecto a cantidad de elementos nos muestran un comportamiento cuadrático, tal como fue analizado teóricamente.

5. Conclusiones

A través de un enfoque de programación dinámica, se pudo encontrar un algoritmo para obtener la secuencia óptima de entrenamientos y descansos que permiten a los jugadores alcanzar la mayor ganancia posible, teniendo en cuenta sus niveles de energía y los esfuerzos requeridos en cada sesión de entrenamiento.

Los resultados demuestran que, si bien el problema de Scaloni parecía tener una relación al problema de "Juan el vago", encontrando una solución óptima secuencialmente cada día sólo con los óptimos de los días anteriores, no se puede encontrar una solución de tal estilo, debido a que no depende de las ganancias que se puedan obtener en cada entrenamiento únicamente, sino que la energía es otra variable que debía ser agregada al subproblema. Esto produjo que se precise analizar los distintos casos posibles con cada energía, para encontrar la solución óptima, llegando a una complejidad $O(n^2)$.

6. Anexo

6.1. Algoritmo para obtener la solución óptima

El código en Python que resuelve la problemática resulta, finalmente:

```
def scaloni_dinamico(n, ganancia, energia):
  res = [[0] * n for _ in range (n)]
  res[0][0] = min ( ganancia[0] , energia[0])
  if n == 1:
        return res
  res[1][0] = min ( ganancia[1] , energia[0])
  res[1][1] = res [0][0] + min ( ganancia[1] , energia[1])
  for dia in range (2 , n ) :
    res[dia][0] = max (res[dia -2]) + min(energia[0], ganancia[dia])
    for e in range (1 , n ) :
      if e > dia:
        break
      res[dia][e] = res[dia-1][e-1] + min(ganancia[dia], energia[e])
  return res

  def obtener_resultado(res, n):
  if n == 1:
        optimo = res[0][0]
        orden = ['Entreno']
        return (optimo, orden)
  optimo = max(res[n - 1])
  orden = ['Descanso'] * n
  dia_actual = n - 1
  energia_actual = res[n-1].index(optimo) }
  while dia_actual >= 0:
    for i in range (energia_actual + 1):
      orden[dia_actual] = 'Entreno'
      if (i != energia_actual):
         dia_actual -= 1
      if (dia_actual < 0):
        break
    dia_actual -= 2
    energia_actual = res[dia_actual].index(max(res[dia_actual]))
  return (optimo, orden)
```

De esta manera, el algoritmo propuesto aborda todos los posibles casos, bajo la condición de que $n \geq 1$, y tanto la ganancia como la energía sean arreglos de tamaño n.

6.2. Seguimiento del algoritmo

Con los siguientes datos:

Ganancias:[55, 67, 44, 26, 58, 16, 34, 67, 38, 25]

Energía:[54, 49, 45, 45, 44, 27, 13, 12, 12, 73]

Se puede representar gráficamente una tabla en la que cada fila corresponde a un día de entrenamiento (ganancia) y cada columna representa la energía disponible. Esta representación puede comenzar en la esquina inferior izquierda de la tabla y proceder a iterar.

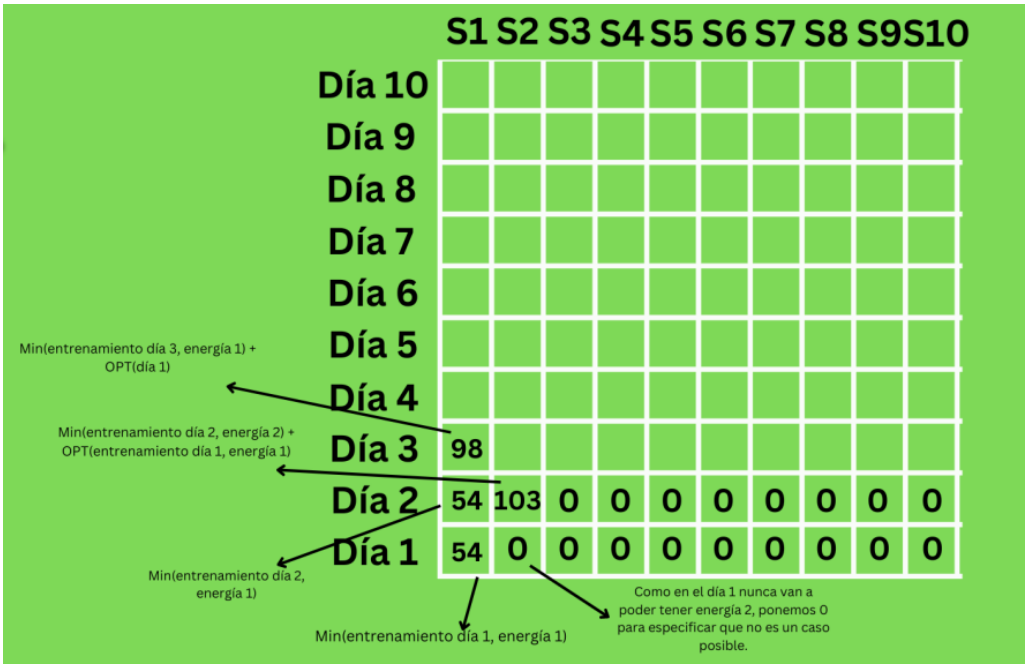


Figura 1: Seguimiento, primeras iteraciones

Siguiendo cada paso tal como se indica en el gráfico, se obtiene la matriz de óptimos para cada ganancia y energía. Luego, para obtener el plan de entrenamiento que lleva a la solución óptima, con las conclusiones expresadas en la sección ***3.1***, gráficamente se podría obtener el plan de entrenamiento de la siguiente forma.

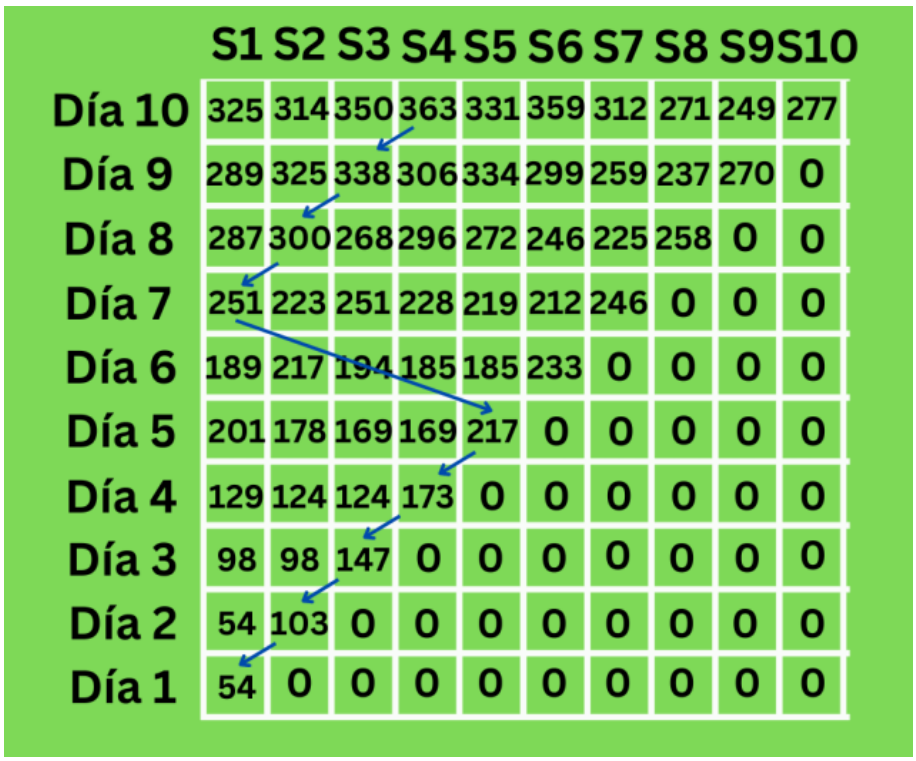


Figura 2: Seguimiento, plan de entrenamiento

La iteración comienza al calcular el óptimo del día que se busca obtener. Durante la iteración, se retrocede una fila y una columna en la tabla correspondiente. Cada retroceso indica un día de entrenamiento. Cuando se alcanza el nivel de energía 1, esto señala que el día anterior se descansó. Se toma entonces un día de descanso y se continúa el proceso iterativo desde el óptimo dos días atrás, repitiendo el mismo algoritmo.

6.3. Mediciones

Para verificar que los distintos valores de esfuerzos no infieren en la complejidad temporal del algoritmo, se plantean distintas variaciones de energía para el problema planteado de la subsección anterior:

Variación 1: [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]

Variación 2: [65, 63, 61, 60, 59, 58, 55, 53, 48, 30]

Variación 3: [60, 20, 18, 16, 15, 14, 12, 10, 6, 5]

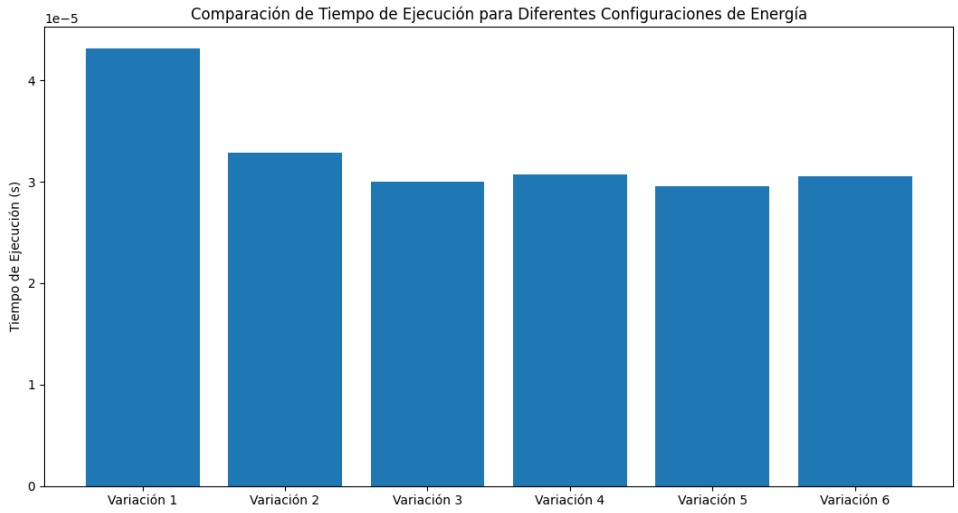
Variación 4: [66, 66, 66, 66, 66, 66, 66, 66, 66, 66]

Variación 5: [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Variación 6:[54, 49, 45, 45, 44, 27, 13, 12, 12, 73]

Ganancias:[55, 67, 44, 26, 58, 16, 34, 67, 38, 25]

Se obtiene el siguiente gráfico con tiempos de ejecución:



Al observar el gráfico comparativo, se puede notar claramente que no hay una variación significativa en el tiempo de ejecución a medida que cambian las configuraciones de energía. Esto sugiere que la complejidad temporal del algoritmo no se ve afectada por las variaciones en la energía.

Referencias

[1] *Repositorio*. URL: `https://github.com/milemarchese/TDA-buchwald`.