(De) Penga

Sheyla Leyva Sánchez - Darío Fragas Gonzalez

Curso 2023

Índice

1. Problema

¿Jenga?

A Marcos le compraron un Jenga por navidad. Jenga es un juego que consiste en armar una torre con palitos, para luego ir retirándolos uno a uno hasta que la torre se desmorone. Marcos no sabe esto. De hecho Marcos no tiene ni idea de qué hacer con el reguero de palitos que tiene sobre la mesa. Como todo un programador, decide utilizar los palitos para un juego que, según él, es más divertido que Jenga, fueran cuales fueran sus reglas.

Primero coloca algunos palitos de manera que se formen n columnas. Cada columna i tiene inicialmente tamaño h_i. Entonces se pueden realizar tres acciones distintas, con distintos costos que se definen al inicio del juego:

- Coloca un palito sobre una columna aumentando en 1 su tamaño (costo C)

- Eliminar el palito más arriba de una columna disminuyendo en 1 su tamaño (costo E)

- Mover el palito más arriba de una columna a la posición más arriba de otra columna (M)

El objetivo del juego es lograr que todas las columnas tengan la misma altura, utilizando acciones que sumen el menor costo posible. Encuentre un algoritmo que reciba una lista de tamaño n con las h_i alturas iniciales junto con los enteros C, E y M de los distintos costos y calcule el costo de la secuencias de acciones más eficiente.

2. Resumen del problema

Dada una lista de alturas iniciales de las columnas (h) y los costos de las acciones (C, E, M), el objetivo es encontrar la secuencia de acciones más eficiente para igualar todas las columnas, minimizando el costo total.

3. Exploración del problema

Como primera aproximación hagamos una modelación abusiva del problema que nos permita obtener la mayor información de la solución.

Para ello definimos un estado del juego S con una tupla (H, P, C, A) donde:

- H: una lista de n enteros representando las columnas - P: el estado padre desde el cual se llegó a S - C: el costo de llegar a este estado - A: acción mediante la cual se llegó de P a S.

Las acciones podemos representarlas por tuplas de 2 y 3 elementos, donde el primer elemento será el tipo de movimiento a realizar:

- C: colocar - E: eliminar - M: mover

Para las acciones C y M el segundo elemento será el índice donde se realiza la acción y el caso M el segundo y tercer elemento corresponden respectivamente con el origen y el destino de mover.

De esta forma se puede construir un grafo G, con estados por vértices y las aristas correspondiente a las acciones que llevan de un estado a otro. Además ponderamos las aristas con el costo de la acción.

Definimos como estados admisibles aquellos estados con columnas de igual altura.

Haciendo una búsqueda del camino de costo mínimo a los estados admisibles en G podemos determinar la solución al problema así como la vía a seguir para llegar a dicha solución.

Para la generación de G debemos tomar criterios de paradas porque sino G podría hacerse infinito.

Para ello tomemos el costo de una solución trivial como igualar todas las alturas a la altura máxima o eliminar todos los elementos hasta obtener la altura mínima.

El algoritmo sería el siguiente:

1. 1- Establecer un costo mínimo C
2. Tomar una cola Q con el estado inicial S_0
3. Mientras haya elementos en Q tomar S primer elemento de Q y desencolarlo. Hacer:
   - Si S es un estado admisible actualizar costo mínimo y guardar S en la lista de estados admisibles
   - Si el costo de S es mayor que el costo mínimo lo ignoramos, de lo contrario:
   - Tomar las acciones que se pueden hacer a partir de S
   - Generar los nuevos estados a partir de las acciones y añadirlos a Q

Para una convergencia más rápida se puede comprobar que un nuevo estado no se añada si un estado con la misma disposición de columnas ya existe con menor costo.

4. Primer acercamiento

Las soluciones que finalizan teniendo todas las columnas a una misma altura satisfacen la condición final del problema, sin embargo, lo que se quiere es de entre todas las soluciones que satisfacen esta restricción, quedarse con la solución optima.

Luego, es fácil pensar que si se tienen los costos de igualar todas las columnas a todas las alturas posibles, se pueden comparar entre sí estos costos y tomar finalmente el mínimo de todos estos.

Proposición 4.1. *Sea* $S = S_{h_0}, S_{h_1}, S_{h_2}...S_{h_n}$ *el conjunto de todas los costos de llevar todas las columnas a una altura* $h_i, 0 \in NU0$ *, la solución óptima del problema es el menor número de este conjunto*

Demostración. Supongamos que el costo óptimo no se encuentra en $S \Rightarrow$ la solución óptima no se obtiene como resultado de igualar todas las columnas a una altura i

Esto es una contradicción pues es requisito del problema que una solución cumpla esto

$\Rightarrow$ La solución óptima se encuentra en S

Supongamos ahora que el óptimo no es el mínimo de $S \Rightarrow$ la solución óptima no es de las que cumplen la condición de S, la de menor costo

Esto es una contradicción pues es requisito del problema que la solución sea la de menor costo

$\Rightarrow$ La solución óptima es el menor número del conjunto S

□

El problema queda entonces resumido a construir el conjunto S. En un principio, el conjunto S es infinito pues la cantidad de alturas posibles es $i \in NU0$ sin embargo, el espacio de las alturas se puede reducir mucho, si se nota que hay muchas que no son soluciones viables, o sea, que siempre van a tener un costo superior al de la solución óptima.

Las alturas que no tienen sentido analizar en este problema son las alturas que son mayor que h_{max} (la mayor de las alturas iniciales) y las que sean menor que h_{min} (la menor de las alturas iniciales).

Proposición 4.2. *Llevar todas las columnas a una altura $h < h_{min}$ o $h > h_{max}$ siempre será más costoso que llevar todas las columnas a una altura H tal que $h_{min} \leq H \leq h_{max}$*

Demostración. caso 1: Si se llevaron todas las columnas a una altura $h < h_{min}$

$\Rightarrow$ Existe una columna i que originalmente tenia altura $h_i \geq h_{min}$ y ahora tiene una altura h tal que $h < h_{min}$

En esta columna se realizó al menos una operación de costo E o una de costo M para llevar de h_{min} a h, pues como las acciones se realizan de una en una, se tiene que llegar primero a h_{min} antes de llegar a $h < h_{min}$

Luego, el costo de llevar la columna i a la altura h será el costo de llevar la columna i a la altura h_{min} más E o M, como $E, M \geq 0$ llevar la columna i a la altura h será igual o más costoso que llevarla a h_min

Lo anterior se debe repetir para cada columna pues ninguna tiene altura h en un principio, por lo que el costo de llevarlas todas a una misma altura h será mayor o igual que llevarlas todas a la altura h_{min}

caso 2: Si se llevaron todas las columnas a una altura $h > h_{max}$

$\Rightarrow$ Existe una columna i que originalmente tenia altura $h_i \leq h_{max}$ y ahora tiene una altura h tal que $h > h_{max}$

En esta columna se realizó al menos una operación de costo C o una de costo M para llevar de h_{max} a h, pues como las acciones se realizan de una en una, se tiene que llegar primero a h_{max} antes de llegar a $h > h_{max}$

Luego, el costo de llevar la columna i a la altura h será el costo de llevar la columna i a la altura h_{max} más C o M, como $E, M \geq 0$ llevar la columna i a la altura h será igual o más costoso que llevarla a h_max

Lo anterior se debe repetir para cada columna pues ninguna tiene altura h en un principio, por lo que el costo de llevarlas todas a una misma altura h será mayor o igual que llevarlas todas a la altura h_{max}

Finalmente, queda demostrado que llevar todas las columnas a una altura $h < h_{min}$ o $h > h_{max}$ siempre será más costoso que llevar todas las columnas a una altura H tal que $h_{min} \leq H \leq h_{max}$

□

Se ha logrado disminuir la cardinalidad de S a $h_{max} - hmin$. Por lo que para encontrar la solución óptima se tendrían que hallar los costos de llevar todas las columnas a la altura h_{min}, $h_{min} + 1, ...h_{max}$ y quedarse con el menor de estos costos

Del problema también se conoce que:

Eliminar de la columna $i \Rightarrow h_i - 1$
Colocar en la columna $j \Rightarrow h_j + 1$
Mover de la columna i a la columna $j \Rightarrow h_i - 1 \ \& \ h_j + 1$

Lo que implica que la acción de mover es equivalente a una acción de eliminar y una de colocar $\Rightarrow$ si $M < C + E$ se deberían cambiar todos los pares colocar-eliminar que se pudieran por acciones de mover; si $M > C + E$, es mejor evitar realizar acciones de mover y realizar las acciones de eliminar-colocar equivalentes. (1)

5. Primera solución

Teniendo en cuenta las ideas anteriores se realizó un algoritmo de programación dinámica basado en que el problema planteado tiene dos soluciones triviales: por cada columna poner todo lo que falte hasta la altura óptima o quitar todo lo posible hasta la altura mínima. Dada solo las acciones de poner o eliminar, se puede construir un algoritmo que haga lo menos costoso de ambas posibles acciones en cada momento. Luego, teniendo en cuenta lo explicado en (1), se sustituyen o no, las acciones de colocar-eliminar por su homólogo mover, tanto sea posible.

5.1. Algoritmo

```
1  def penga(h, C, E, M):
2      """
3      h: lista de enteros
4      C: entero
5      E: entero
6      M: entero
7      """
8
9
10     h_max = max(h)
11     h_min = min(h)
```

```
12
13
14      COSTS = [[(0, 0) for _ in range(h_max + 1)] for _ in range(len(
        h) + 1)]
15
16
17      for i in range(1, len(h) + 1):
18          for j in range(h_min, h_max + 1):
19              if j > h[i - 1]:
20                  c = j - h[i - 1]
21                  t = COSTS[i - 1][j]
22                  COSTS[i][j] = (t[0] + c, t[1])
23              else:
24                  e = h[i - 1] - j
25                  t = COSTS[i - 1][j]
26                  COSTS[i][j] = (t[0], t[1] + e)
27
28
29      min_cost = int("inf")
30      for cost in COSTS[len(h)][h_min:]:
31          if cost[0] < cost[1]:
32              c = cost[0] * C + cost[1] * E
33              min_cost = min(min_cost, c)
34              c = (cost[1] - cost[0]) * E + cost[0] * M
35              min_cost = min(min_cost, c)
36              c = cost[1] * M + (cost[1] - cost[0]) * C
37              min_cost = min(min_cost, c)
38          else:
39              e = cost[0] * C + cost[1] * E
40              min_cost = min(min_cost, e)
41              e = (cost[0] - cost[1]) * C + cost[1] * M
42              min_cost = min(min_cost, e)
43              e = cost[0] * M + (cost[0] - cost[1]) * E
44              min_cost = min(min_cost, e)
45
46
47      return min_cost, COSTS
```

5.2. Correctitud

La solución planteada es una solución mediante programación dinámica, por lo que, para probar la optimalidad y correctitud del algoritmo se deben seguir los siguientes pasos:

Identificar la subestructura óptima: Esto implica demostrar que la solución óptima al problema original puede ser construida a partir de las soluciones óptimas a los subproblemas. En este caso, se construye la matriz $COSTS$ para almacenar la cantidad de movimientos de colocar y eliminar acumulados de los subproblemas, donde $COSTS[i][j]$ representa la cantidad de movimientos optimos para alcanzar la altura j utilizando las primeras i columnas.

El algoritmo toma decisiones óptimas en cada paso, basándose en si j es mayor o menor que el elemento correspondiente en la lista h. Si $j > h[i-1]$, se colocan palitos en la columna, y si $j \leq h[i-1]$, se elimina palitos de la columna.

La función de transición de estados determina cómo se pasa de un subproblema a otro. En este caso, se actualiza la matriz $COSTS$ en función de los valores de h, C, E, y M. La matriz $COSTS$ se actualiza correctamente en función de si se necesita agregar o eliminar un palito de la columna para igualar las alturas.

El algoritmo encuentra la solución óptima al problema, pues se busca el costo mínimo considerando diferentes combinaciones de C, E y M en base a los costos acumulados almacenados en la matriz $COSTS$ y las sustituciones convenientes por acciones de mover (1).

5.3. Complejidad Temporal

La complejidad temporal de este algoritmo es $O(n * |h_max - h_min|)$, donde n es la longitud de la lista h. Esto se debe a que la matriz $COSTS$ tiene dimensiones $(n + 1) * (h_max + 1)$ y, en el peor de los casos, se recorrerán todos los elementos de la matriz en los bucles anidados.

6. Observaciones posteriores

Luego de un análisis mayor del problema, se llegó a la idea de que la función de costos de llevar a las distinta alturas pudiera ser unimodal, o sea, tener un único mínimo, mediante algunos *plots* se pudo ver que muchos casos cumplían la hipótesis planteada, sin embargo, no se logró llegar a una demostración formal.

De igual manera, se decidió asumir como cierta la hipótesis para poder avanzar con una mejor solución basada en que si la función es unimodal se puede aplicar búsqueda binaria o ternaria para encontrar su mínimo siguiendo una idea parecida a esta

Siguiendo la lógica que se ha compartido hasta ahora, se pensó que esto mejoraría mucho el costo temporal del algoritmo pues en lugar de calcular los costos de todas las posibles alturas, solo se deberían calcular los costos necesarios para realizar las comparaciones que exige un algoritmo de búsqueda binaria con predicados

Finalmente, se implementó el siguiente algoritmo donde penga-goal devuelve el costo de llevar todas las columnas a una altura dada:

```
def penga_binary(h, C, E, M):
    """
    O(n log(|h_max-h_min|))
    """
    low = 0
    top = max(h)


    while top - low > 1:
        mid = (top + low) // 2
        mid_cost = penga_goal(h, C, E, M, mid)
        next_cost = penga_goal(h, C, E, M, mid + 1)
        if mid_cost > next_cost:
            low = mid
        else:
```

```
            top = mid

    if top - low == 1:
        top_cost = penga_goal(h, C, E, M, top)
        low_cost = penga_goal(h, C, E, M, low)
        return min(top_cost, low_cost)

    return penga_goal(h, C, E, M, top)
```

Complejidad Temporal

La complejidad temporal de este algoritmo es $O(n * log(k))$, donde n es la longitud de la lista h y $k = h_{max} - h_{min}$. Esto se debe a que saber el costo necesario para llevar las columnas a una altura dada es n y realizar búsqueda binaria en un espacio de tamaño k es $log(k)$

7. Generador de casos de prueba y *tester*

Se implementó un generador de casos de prueba y un tester que se encuentran en el repositorio de GitHub. El algoritmo de exploración es muy lento, por lo que, una vez demostrada la correctitud de la solución dinámica, esta se utilizó para evaluar la correctitud del algoritmo por busqueda binaria.