Informe: MineCraft

Contents

1 Problema

En el juego de MineCraft una de las principales distracciones es la construccion, los mejores jugadores logran hacer monumentos imponentes que sorprenden a todos. Actualmente se esta llevando a cabo un torneo de construcción en el juego, donde la tarea es hacer un muro. Un muro es, como sabemos, una hilera de columnas de bloques de piedra, todos de la misma altura. En MineCraft, para llevar a cabo esta tarea hay 3 movimientos válidos:

- sacar un bloque de piedra del inventario y aumentar la altura de la columna en cuestión en 1 de altura - destruir un bloque de piedra de una columna y disminuir la altura de la columna en cuestión en 1 de altura - mover un bloque de piedra de una columna a otra, aumentando la altura de la 2da columna y disminuyendo la de la 1ra en 1 de altura cada una.

Se sabe que hacer cada movimiento consume c, d y m de energía respectivamente, y que en los inventarios de los jugadores hay suficientes bloques de piedra siempre. Los jugadores comienzan a jugar con un muro a medio hacer aleatorio, o sea, se les da una cantidad n de columnas de bloques de piedra en hilera, de disímiles tamaños y el ganador del torneo será el que construya un muro de largo n utilizando la menor cantidad de energía, no se permite crear columnas nuevas ni dejar huecos de antiguas columnas en el muro por supuesto.

Elabore una estrategia que asegure que para cualquier muro inicial a medio hacer con que comience, usted logrará hacer el muro pedido utilizando la menor cantidad de energía posible.

2 Reformulación del problema

Dado un arreglo `heights` de longitud n, donde cada elemento `heights[i]` representa la altura de la columna i de un muro incompleto, se debe igualar la altura de todas las columnas utilizando los siguientes movimientos, cada uno con un costo energético asociado:

- Agregar un bloque a la columna i (incrementa `heights[i]` en 1) con un costo de c unidades de energía.
- Eliminar un bloque de la columna i (decrementa `heights[i]` en 1) con un costo de d unidades de energía.
- Mover un bloque de la columna i a la columna j (decrementa `heights[i]` en 1 y aumenta `heights[j]` en 1) con un costo de m unidades de energía.

El objetivo es igualar la altura de todas las columnas del muro minimizando el costo total de energía.

2.1 Definir la intención

Una solución óptima del problema tiene una altura h_0. Todas las filas por encima de esa altura estarán vacías y todas las filas por debajo estarán llenas. Para resolver el problema, se determinará, para cada fila, su estado en dicha solución óptima. El estado de cada fila se encuentra mediante el uso de dos lemas clave:

- **Lema 1:** Una fila estará llena en la solución final si el costo de rellenar los bloques hasta esa fila es menor o igual al costo de vaciar solo esa fila.
- **Lema 2:** Una fila estará vacía si el costo de vaciar todos los bloques hasta esa fila es menor o igual al costo de llenarla.

Se demuestra que, en cada iteración, se puede determinar el estado de al menos una fila, lo que garantiza la convergencia hacia una solución óptima.

2.2 Observaciones

- Pueden existir varios muros óptimos con diferentes alturas.
- Las filas se enumeran de abajo hacia arriba, por lo tanto, si $f_0 < f_1$ entonces f_0 tiene menos altura que f_1.
- El término "rellenar" se refiere a construir o mover bloques hacia el lugar, y "vaciar" hace referencia a destruir o mover bloques desde el lugar.

(*1) Consideraciones sobre el movimiento de bloques

Mover bloques hacia arriba o hacia los lados no tiene sentido. Supongamos que queremos mover de la fila f_0 a la fila f_1, siendo $f_0 \leq f_1$.

- **Caso 1:** La altura óptima $h_0 \leq f_0$. En este caso, todos los bloques por encima de h_0, incluidos los de la fila f_1, tendrán que ser destruidos o movidos hacia abajo para alcanzar la altura óptima, por lo que se gasta energía innecesariamente al mover bloques.
- **Caso 2:** La altura óptima $h_0 \geq f_0$. En este caso, todos los bloques por encima, incluidos los de la fila f_0, tendrán que ser construidos para alcanzar la altura óptima, por lo que también se gasta energía innecesariamente al mover bloques.

3 Correctitud

El algoritmo es correcto si, para cualquier configuración de alturas iniciales y costos de energía c, d y m, produce un costo mínimo para nivelar las alturas de las columnas. En cada iteración, evaluando el costo de cada acción, se decide si es más barato disminuir las columnas más altas o aumentar las columnas más bajas. Como esta decisión también es la mejor en una solución final, al repetir hasta que las columnas tengan la misma altura, se puede concluir que se obtiene un costo mínimo para nivelar el muro. La terminación del algoritmo está garantizada porque, como en cada iteración se reduce la diferencia entre la altura máxima $h_{\max}$ y la mínima $h_{\min}$, eventualmente sucederá que $h_{\max} = h_{\min} = h_0$.

3.1 Lemas para determinar el estado de una fila en una solución final

- **Lema 1:** Si el costo de rellenar todos los bloques hasta una fila es menor o igual que el costo de vaciar solo esa fila, entonces la fila estará llena en alguna solución final.

- **Lema 2:** Si el costo de vaciar todos los bloques hasta una fila es menor o igual que el costo de rellenar una fila, entonces la fila estará vacía en alguna solución final.

3.2 Notaciones y definiciones

- $H \rightarrow$ conjunto de alturas de soluciones óptimas.
- $long \rightarrow$ cantidad de columnas del muro.
- $c \rightarrow$ costo de construir un bloque.
- $d \rightarrow$ costo de destruir un bloque.
- $block\_count(f) \rightarrow$ cantidad de bloques de una fila f.
- $refill\_move\_count(f) \rightarrow$ cantidad de bloques que se pueden mover hacia una fila f.
- $clean\_move\_count(f) \rightarrow$ cantidad de bloques que se pueden mover desde una fila f.
- $crf(f) \rightarrow$ costo de rellenar los bloques de una fila f.
- $crtf(f) \rightarrow$ costo de rellenar todos los bloques desde el suelo hasta una fila determinada.
- $cvf(f) \rightarrow$ costo de vaciar los bloques de una fila f.
- $cvtf(f) \rightarrow$ costo de vaciar todos los bloques desde el tope hasta una fila determinada.

Las fórmulas para el costo de rellenar y vaciar una fila f son las siguientes:

$$crf(f) = m \times refill\_move\_count(f) + c \times (long - block\_count(f) - refill\_move\_count(f)) - d \times refill\_move$$

$$cvf(f) = m \times clean\_move\_count(f) + d \times (block\_count(f) - clean\_move\_count(f)) - c \times clean\_move\_count($$

Si $m < c + d$, los costos $crf(f)$ y $cvf(f)$ disminuirán cuando m aumente. Es decir, nos interesa utilizar movimientos siempre que sea posible. Si no, los movimientos no se realizarán, ya que empeorarían el costo.

3.3 Formalización del Lema 1

$$crtf(f) \leq cvf(f) \implies \exists h \in H : h \geq f$$

Supongamos $crtf(f) \leq cvf(f)$ y no existe $h \in H : h \geq f$ para llegar a una contradicción. Si modificamos el muro óptimo de altura h_0, haciendo su altura igual a f, entonces, al rellenarlo hasta la fila f, la diferencia de costo con respecto al muro inicial será la de deshacer la destrucción o movimiento de los bloques que se habían eliminado y la de construir los bloques que no estaban anteriormente. Por tanto, el costo disminuye en $cvf(f)$ y aumenta en $crtf(f)$; como $crtf(f) \leq cvf(f)$, el costo del nuevo muro es menor o igual que el del muro óptimo. Por lo tanto, existe $h = f$ que pertenece a H, lo cual es imposible.

3.4 Formalización del Lema 2

$$cvtf(f) \leq crf(f) \implies \exists h \in H : h < f$$

Análogo al Lema 1: Supongamos $cvtf(f) \leq crf(f)$ y no existe $h \in H : h < f$ para llegar a una contradicción. Si modificamos el muro óptimo de altura h_0, haciendo su altura igual a f, entonces, al vaciarlo hasta la fila $f - 1$, la diferencia de costo con respecto al muro inicial será la de deshacer la construcción o movimiento de los bloques que se habían añadido y la de destruir los bloques que estaban anteriormente. Por tanto, el costo disminuye en $crf(f)$ y aumenta en $cvtf(f)$; como $cvtf(f) \leq crf(f)$, el costo del nuevo muro es menor o igual que el del muro óptimo. Por lo tanto, existe $h = f - 1$ que pertenece a H, lo cual es imposible.

3.5 Existencia de una fila que cumpla alguna premisa de los lemas

piso $\rightarrow$ fila cuya altura es igual a la altura de la columna más baja más uno.
tope $\rightarrow$ fila cuya altura es igual a la altura de la columna más alta.

$$piso = \min(heights) + 1$$

$$tope = \max(heights)$$

Por la naturaleza de las columnas de un muro, si una fila f_0 está por debajo de una fila f_1, la cantidad de bloques en f_0 es mayor o igual que la cantidad de bloques en f_1, porque para que un bloque esté en f_1 necesariamente tiene que estar en f_0. Sin embargo, el hecho de estar en f_0 no implica que esté en f_1. De forma análoga, si falta un bloque en f_0, también faltará en f_1, pero el hecho de que falte en f_1 no implica que falte en f_0.

Lema 3: $f_0 \leq f_1 \iff block\_count(f_0) \geq block\_count(f_1)$

Ahora demostraremos que siempre se cumple la premisa del Lema 1 para la fila *piso* o la del Lema 2 para la fila *tope*:

$$crtf(piso) \leq cvf(piso) \quad \text{o} \quad cvtf(tope) \leq crf(tope)$$

3.6 Vía 1

Demostraremos:

$$\neg\left(crtf(piso) \leq cvf(piso)\right) \Rightarrow cvtf(tope) \leq crf(tope)$$

Es decir:

$$crtf(piso) > cvf(piso) \Rightarrow cvtf(tope) \leq crf(tope)$$

1) $crtf(piso) > cvf(piso)$
Sabemos que:

$$crtf(piso) = crf(piso)$$

por la definición de *piso*, entonces tenemos:

$$crf(piso) > cvf(piso)$$

Además, como:

$$crf(tope) = c \cdot (long - block\_count(tope))$$

ya que no se puede considerar el uso de movimientos debido a (*1).
En el peor de los casos, si no fuera factible usar movimientos para reducir el costo, el costo de $crf(piso)$ sería igual:

$$crf(piso) \leq c \cdot (long - block\_count(piso))$$

Por el Lema 3, sabemos que:

$$block\_count(piso) \geq block\_count(tope)$$

2) Entonces se cumple que:

$$crf(tope) \geq crf(piso)$$

3) De lo anterior obtenemos:

$$crf(tope) \geq crf(piso) > cvf(piso)$$

Sabemos que:

$$cvf(piso) = d \cdot block\_count(piso)$$

ya que no se puede considerar usar movimientos debido a (*1).
En el peor de los casos, si no fuera factible usar movimientos para reducir el costo, el costo de $cvf(tope)$ sería:

$$cvf(tope) \leq d \cdot block\_count(tope)$$

Por el Lema 3, se cumple que:

$$block\_count(piso) \geq block\_count(tope)$$

4) Entonces:

$$cvf(piso) \geq cvf(tope)$$

Finalmente, combinamos los resultados de (3) y (4):

$$crf(tope) \geq crf(piso) > cvf(piso) \geq cvf(tope)$$

Dado que:

$$cvtf(tope) = cvf(tope)$$

por definición de *tope*, tenemos que:

$$crf(tope) \geq crf(piso) > cvf(piso) \geq cvtf(tope)$$

Por lo tanto:

$$crf(tope) > cvtf(tope)$$

y concluimos:

$$cvtf(tope) < crf(tope)$$

Por lo tanto, se cumple que:

$$cvtf(tope) \leq crf(tope)$$

lo que queríamos demostrar (lqqd).

3.7 Vía 2

Demostraremos que siempre se cumple:

$$crf(piso) \leq cvf(tope) \quad \text{o} \quad cvf(tope) \leq crf(piso) \quad (p \quad \text{o} \quad \neg p)$$

3.7.1 Caso 1: $crf(piso) \leq cvf(tope)$

Sabemos que:

$$crf(piso) = crtf(piso)$$

por definición de *piso*, por lo tanto:

$$crtf(piso) \leq cvf(tope)$$

Además, por (4), sabemos que:

$$cvf(tope) \leq cvf(piso)$$

Entonces:

$$crtf(piso) \leq cvf(tope) \leq cvf(piso)$$

Y concluimos que:

$$crtf(piso) \leq cvf(piso)$$

que es la premisa del Lema 1.

3.7.2 Caso 2: $cvf(tope) \leq crf(piso)$

Sabemos que:

$$cvf(tope) = cvtf(tope)$$

por definición de *tope*, por lo tanto:

$$cvtf(tope) \leq crf(piso)$$

Además, por (2), sabemos que:

$$crf(piso) \leq crf(tope)$$

Entonces:

$$cvtf(tope) \leq crf(piso) \leq crf(tope)$$

Y concluimos que:

$$cvtf(tope) \leq crf(tope)$$

que es la premisa del Lema 2.

Por lo tanto, siempre se cumple que:

$$crtf(piso) \leq cvf(piso) \quad \text{o} \quad cvtf(tope) \leq crf(tope) \quad \text{lqqd.}$$

4 Conclusiones

Dado que siempre se cumple la premisa del Lema 1 para la fila *piso*, o la premisa del Lema 2 para la fila *tope*, entonces en cada iteración del problema se puede determinar el estado de una de estas filas en una solución óptima. Esto garantiza la convergencia hacia dicha solución.

5 Implementación de Soluciones

Se han implementado dos soluciones basándose en la demostración:

5.1 Solución a

5.1.1 Código en Python

```
def min_energy_to_build_wall(heights, c, d, m):

destroy_count = 0
construct_count = 0
move_count = 0
move_blocks = m < c + d
max_height = max(heights)
min_height = min(heights)

while max_height != min_height:

ceil_count = heights.count(max_height)
floor_missing_count = heights.count(min_height)

clean_possible_moves = min(ceil_count, construct_count) if move_blocks else 0
ceil_clean_cost = m * clean_possible_moves + d * (ceil_count - clean_possible_moves) +

refill_possible_moves = min(floor_missing_count, destroy_count) if move_blocks else 0
floor_refill_cost = m * refill_possible_moves + c * (floor_missing_count - refill_poss

if ceil_clean_cost <= floor_refill_cost:
# limpiar el techo
heights = [h-1 if h == max_height else h for h in heights]
```

```
max_height -= 1
construct_count -= clean_possible_moves
move_count += clean_possible_moves
destroy_count += ceil_count - clean_possible_moves

else:
# rellenar el suelo
heights = [h+1 if h == min_height else h for h in heights]
min_height += 1
destroy_count -= refill_possible_moves
move_count += refill_possible_moves
construct_count +=  floor_missing_count - refill_possible_moves


cost = destroy_count * d + construct_count * c + move_count * m
print("Move count ", move_count, "Destroy count ", destroy_count, "Construct count ",
return cost
```

5.1.2 Explicación del código

La función `min_energy_to_build_wall` tiene como objetivo igualar las alturas de las columnas de un muro, minimizando el costo energético. A continuación, se explican las principales partes del código:

- La variable `move_blocks` indica si es posible mover bloques en lugar de destruir o construir nuevos.
- El bucle principal se ejecuta mientras haya una diferencia entre la altura máxima y mínima.
- Dentro del bucle, se calculan los costos de limpiar (bajar la altura máxima) y rellenar (aumentar la altura mínima):
- Dependiendo de cuál costo sea menor, se decide limpiar el techo o rellenar el suelo.
- Al final, se calcula y se devuelve el costo total.

5.2 Complejidad Temporal

La complejidad temporal del algoritmo depende de la cantidad de iteraciones del bucle principal y de las operaciones realizadas dentro de él:

1. **Bucle principal:** Este bucle se ejecuta mientras `max_height` no sea igual a `min_height`. En el peor de los casos, esto puede requerir un número de iteraciones igual a la diferencia entre la altura máxima y mínima, que denotaremos como H.
2. **Operaciones dentro del bucle:**
   - Contar elementos con `heights.count(max_height)` y `heights.count(min_height)` toma $O(n)$ en el peor de los casos, donde n es la longitud de `heights`.
   - Las operaciones para actualizar `heights` también son $O(n)$.

Por lo tanto, la complejidad temporal total es $O(H \cdot n)$.

5.3 Complejidad Espacial

La complejidad espacial adicional del algoritmo es $O(1)$ porque no se utilizan estructuras de datos adicionales que escalen con la entrada, ya que la lista `heights` se modifica, pero no se crea ninguna lista nueva que dependa de la entrada. El algoritmo recibe un arreglo para almacenar las alturas de las columnas, lo que requiere $O(n)$ de espacio. La complejidad espacial total es $O(n)$.

5.4 Solución b

5.4.1 Código en Python

```
def min_energy_to_build_wall_optimized(heights, c, d, m):

destroy_count = 0
construct_count = 0
move_count = 0
move_blocks = m < c + d
max_height =  max(heights)
min_height = min(heights)

row_block_count = [0] * (max_height + 1)

for h in heights:
row_block_count[h] += 1

for i in range(max_height, 0, -1):
row_block_count[i-1] += row_block_count[i]

while max_height != min_height:

ceil_count = row_block_count[max_height]
floor_missing_count = len(heights) - row_block_count[min_height + 1]

clean_possible_moves = min(ceil_count, construct_count) if move_blocks else 0
ceil_clean_cost = m * clean_possible_moves + d * (ceil_count - clean_possible_moves) +
refill_possible_moves = min(floor_missing_count, destroy_count) if move_blocks else 0
floor_refill_cost = m * refill_possible_moves + c * (floor_missing_count - refill_poss

if ceil_clean_cost <= floor_refill_cost:
# limpiar el techo
max_height -= 1
construct_count -= clean_possible_moves
move_count += clean_possible_moves
destroy_count += ceil_count - clean_possible_moves

else:
# rellenar el suelo
min_height += 1
```

```
destroy_count -= refill_possible_moves
move_count += refill_possible_moves
construct_count +=  floor_missing_count - refill_possible_moves


cost = destroy_count * d + construct_count * c + move_count * m
print("Move count ", move_count, "Destroy count ", destroy_count, "Construct count ",
return cost
```

5.4.2 Explicación del Código

El código `min_energy_to_build_wall_optimized` optimiza el costo energético de igualar las alturas de las columnas de un muro. Solo se diferencia de la solución a en lo siguiente:

- Se crea un arreglo `row_block_count` que guarda la cantidad de columnas de cada altura.
- Se rellena `row_block_count` para calcular cuántas columnas tienen altura mayor o igual a cada valor, iterando desde la altura máxima hacia abajo.
- `ceil_count = row_block_count[max_height]`: Esta línea calcula cuántas columnas tienen la altura máxima actual. `row_block_count[max_height]` representa el número de columnas cuya altura es mayor o igual a `max_height`.
- `floor_missing_count = len(heights) - row_block_count[min_height + 1]`: Esta línea calcula cuántas columnas están por debajo de la altura mínima + 1 y, por lo tanto, necesitan ser rellenadas (es decir, necesitan que se les construyan más bloques). `row_block_count[min_height + 1]` nos dice cuántas columnas tienen altura mayor o igual a `min_height + 1`. Si restamos este valor del número total de columnas (`len(heights)`), obtenemos el número de columnas cuya altura es menor o igual a `min_height`.

5.5 Complejidad Temporal

- `max_height = max(heights)` y `min_height = min(heights)` tardan $O(n)$, donde n es el número de columnas en `heights`.
- El primer bucle que recorre las alturas tiene una complejidad $O(n)$, ya que recorre cada columna en `heights`.
- El segundo bucle (que ajusta los valores de `row_block_count` desde la altura máxima hacia abajo) tiene una complejidad de $O(h_{\max})$.

Bucle Principal: Cada iteración del bucle reduce `max_height` o incrementa `min_height`, con un número máximo de iteraciones de $h_{\max} - h_{\min}$. Dentro de cada iteración, las operaciones toman tiempo constante $O(1)$.

Por lo tanto, la complejidad del bucle principal es $O(h_{\max} - h_{\min})$.

Complejidad total: La complejidad total es $O(n + h_{\max})$.

5.6 Complejidad Espacial

El espacio para el arreglo `row_block_count`: El espacio adicional más significativo es el arreglo `row_block_count`, que tiene un tamaño $O(h_{\max})$, donde $h_{\max}$ es la altura máxima de las columnas.

Complejidad espacial total: Es $O(h_{\max} + n)$ teniendo en cuenta el espacio de la entrada.

6 Solución por Backtracking

```
def min_energy_to_build_wall_backtrack(heights, c, d, m):
mask = [0] * len(heights)
return min_energy_to_build_wall_backtrackR(heights, c, d, m, 0, mask, float('inf'))

def min_energy_to_build_wall_backtrackR(heights, c, d, m, cost, mask, min_energy):

if cost >= min_energy:
return float('inf')

if len(set(heights)) == 1:
min_energy = min(min_energy, cost)
return cost

max_height = max(heights)
min_height = min(heights)
construct_cost = float('inf')
destruct_cost = float('inf')
move_cost = float('inf')

# aumentar en 1 la altura de una de las columnas más bajas
min_height_index = heights.index(min_height)
if mask[min_height_index] != -1:
last_mask = mask[min_height_index]
mask[min_height_index] = 1
heights[min_height_index] += 1
construct_cost = min_energy_to_build_wall_backtrackR(heights, c, d, m, cost + c, mask
mask[min_height_index] = last_mask
heights[min_height_index] -= 1

# disminuir en 1 la altura de una de las columnas más altas
max_height_index = heights.index(max_height)
if mask[max_height_index] != 1:
last_mask = mask[max_height_index]
mask[max_height_index] = -1
heights[max_height_index] -= 1
destruct_cost = min_energy_to_build_wall_backtrackR(heights, c, d, m, cost + d, mask,
mask[max_height_index] = last_mask
heights[max_height_index] += 1
```

```
# mover un bloque de la columna más alta a la más baja
if mask[max_height_index] != 1 and mask[min_height_index] != -1:
last_mask_max = mask[max_height_index]
last_mask_min = mask[min_height_index]
mask[max_height_index] = -1
mask[min_height_index] = 1
heights[max_height_index] -= 1
heights[min_height_index] += 1
move_cost = min_energy_to_build_wall_backtrackR(heights, c, d, m, cost + m, mask, min_
mask[max_height_index] = last_mask_max
mask[min_height_index] = last_mask_min
heights[max_height_index] += 1
heights[min_height_index] -= 1

return min(construct_cost, destruct_cost, move_cost)
```

6.1 Estructura General del Algoritmo

6.1.1 Base de la Recursión

- Si todos los elementos de `heights` son iguales (es decir, el muro está nivelado), el costo acumulado es la energía mínima actual.
- Si el costo actual supera la energía mínima registrada, el algoritmo corta esa rama de búsqueda, pues no puede producir una solución mejor.

6.1.2 Decisiones Recursivas

El algoritmo busca todas las posibles acciones: construir, destruir, o mover bloques. Evalúa el costo de cada acción en función del costo acumulado y decide en qué columna realizar los cambios:

- Se selecciona la columna de menor altura para agregar un bloque.
- Se selecciona la columna de mayor altura para eliminar un bloque.
- Se selecciona la columna más alta para mover un bloque a la columna más baja.

6.1.3 Máscara (mask)

El arreglo `mask` ayuda a evitar movimientos innecesarios repetidos, limitando las acciones que se pueden tomar en columnas específicas para optimizar el proceso de backtracking.

6.1.4 Detalles Clave

- **Incrementar altura:** Se incrementa la columna de menor altura, se ajusta el costo y se realiza la llamada recursiva.
- **Decrementar altura:** Se decrementa la columna de mayor altura, se ajusta el costo y se realiza la llamada recursiva.

- **Mover bloques:** Se transfiere un bloque de la columna más alta a la más baja y se ajusta el costo.

Finalmente, el algoritmo devuelve el costo mínimo de energía entre las tres operaciones posibles.

6.2 Correctitud

6.2.1 Reducción a Subproblemas

En cada paso recursivo, el algoritmo explora todas las posibles maneras de igualar las alturas: construir, destruir o mover bloques. Dado que explora todas las combinaciones posibles de movimientos mediante backtracking, asegura que, en algún punto, alcanzará una configuración donde todas las columnas sean iguales y retorna el costo acumulado.

7 Optimalidad

El algoritmo almacena y compara el costo mínimo de energía en cada iteración, asegurando que la solución devuelta sea óptima. Corta las ramas donde el costo supera el mínimo conocido, lo que permite minimizar el tiempo de búsqueda sin comprometer la correctitud.

7.1 Complejidad Temporal

El algoritmo de backtracking explora todas las combinaciones posibles de incrementos, decrementos y movimientos de bloques. Esto implica un crecimiento exponencial en el número de posibles configuraciones a explorar.

7.1.1 Número de Posibilidades por Columna

Para cada columna, hay tres posibles acciones: construir, destruir o mover bloques. En el peor de los casos, el algoritmo explora todas estas posibilidades, y dado que hay n columnas y el número de bloques puede cambiar en cada paso, el espacio de búsqueda es aproximadamente del orden de $O(3^n)$.

7.1.2 Complejidad Total

La complejidad temporal aproximada es $O(3^n)$, donde n es el número de columnas. Esto se debe a que en cada paso recursivo se tienen hasta tres decisiones por columna (construir, destruir o mover).

7.2 Complejidad Espacial

El algoritmo utiliza:

- Un arreglo `mask` de tamaño n para almacenar el estado de las columnas.
- La pila de llamadas recursivas tiene una profundidad máxima de $O(n)$, lo que representa el máximo número de decisiones consecutivas antes de llegar a una solución válida o descartar esa rama de búsqueda.

Por lo tanto, la complejidad espacial es $O(n)$, donde n es el número de columnas.

7.3 Conclusión

El algoritmo propuesto utiliza backtracking para encontrar la forma óptima de igualar las alturas de un muro, minimizando el costo de energía. Aunque es correcto y garantiza una solución óptima, su complejidad temporal crece exponencialmente con el número de columnas, lo que puede hacerlo impráctico para grandes valores de n. Sin embargo, la técnica es efectiva para problemas pequeños y asegura la minimización del costo energético a través de la exploración exhaustiva del espacio de soluciones.

7.3.1 Observaciones

1. Si se mueve un bloque desde una columna i con altura menor a $h_{\max}$ hacia abajo, se estaría gastando energía innecesariamente. Digamos que $h_0 \geq h_i$, entonces en esa columna se volverá a poner un bloque. Si $h_0 < h_i$, por las características del algoritmo cuando todas las filas con columnas más altas se hayan vaciado, i será la columna más alta y se moverá el mismo bloque hacia abajo. Da igual si no se pone en la misma posición, el gasto de energía es el mismo.

2. Si se aumenta en 1 la altura de una columna i que no es la más baja y $h_0 \leq h_i$, el bloque que se construyó se va a tener que destruir, incurriendo en un gasto de energía extra. Si $h_0 > h_i$, por las características del algoritmo cuando todas las filas con columnas más bajas se hayan llenado, i será la columna más baja y se agregará un bloque en ella, aunque no sea en el mismo momento, el gasto de energía es el mismo.

3. Si se disminuye en 1 la altura de una columna i que no es la más alta, análogo a 1 se estaría gastando energía innecesariamente, porque si $h_0 \geq h_i$, entonces en esa columna se volverá a poner un bloque. Si $h_0 < h_i$, por las características del algoritmo cuando todas las filas con columnas más altas se hayan vaciado, i será la columna más alta y se destruirá el mismo bloque.