Polígono Mágico

María de Lourdes Choy Fernández
Alejandro Yero Valdéz
Leismael Sosa

Contents

1 Problema **1**

2 Modelación **1**
2.1 Definiciones 1
2.1.1 Triangulación de un polígono convexo 1
2.1.2 Subdivisión ideal 1
2.1.3 Corte ideal 1
2.2 Abstraccíon del problema 2

3 Solución **3**
3.1 Fuerza bruta 3
3.2 Acercamiento por camino de costo mínimo 3
3.3 Greedy 3
3.4 Solución propuesta 4
3.5 Programación Dinámica (DP) 4
3.5.1 Inicialización 5
3.5.2 Transición 5
3.5.3 Condición de Corte Válido 5
3.6 Búsqueda Binaria para Maximizar w 6
3.7 Argumentación de Correctitud 6
3.8 Complejidad Temporal 6
3.8.1 a. Programación Dinámica (DP) 7
3.8.2 b. Búsqueda Binaria 7
3.8.3 c. Complejidad Total 7

1 Problema

En un reino lejano, existía un mago llamado Alejandro que protegía un gran tesoro escondido dentro de un castillo mágico. Este castillo tenía la forma de un polígono perfecto y estrictamente convexo con n torres, cada una conectada por paredes fuertes e inquebrantables.

La reina María que creía firmemente en las ideas de la Revolución (¿Cuál? no sé, la revolución mágica del reino lejano) decidió que era hora de compartir parte del tesoro con el pueblo, pero debía hacerlo con cuidado para no despertar la ira de Alejandro. La reina María llamó a los mejores arquitectos del reino y les pidió que hicieran k cortes en las paredes del castillo, utilizando portales mágicos que conectaran dos torres distintas que no estuvieran directamente adyacentes. Sin embargo, había una advertencia: los portales no podían cruzarse en medio de las paredes, solo en las torres del castillo.

El objetivo de la reina María era repartir las riquezas de tal manera que cada porción de territorio dentro del castillo fuera lo más equitativa posible. Así, pidió a los arquitectos que maximizaran el área de la porción más pequeña que se formara dentro del castillo después de colocar los k portales mágicos.

Tu tarea es ayudar a la reina María a encontrar la mejor manera de cortar el castillo para que la distribución del tesoro sea justa para todos.

(PD: Cuando Alejandro regresó a su castillo y lo vio todo cuarteado, en un ataque de ira convocó a la bestia mágica Yesapin, pero eso es historia para otro momento)

2 Modelación

2.1 Definiciones

2.1.1 Triangulación de un polígono convexo

Por conocimientos de geometría se conoce que la cantidad máxima de triángulos en los que se puede dividir un polígono es $n-2$, donde n es la cantidad de vértices del polígono, esto es el resultado de realizar $n-3$ cortes sin que se crucen y solamnte inciden el los vértices de dicho polígono.

2.1.2 Subdivisión ideal

Se define *subdivisión ideal* a una subdivisión donde todas las areas resultantes tras la división tienen igual área.

2.1.3 Corte ideal

Se define *corte ideal* a aquel corte que genera un área lo más cercana posible modularmente a $\frac{A_p}{k+1}$.

2.2 Abstraccíon del problema

Tenemos como objetivo del problema maximizar el área más pequeña resultante de realizar k cortes. El caso ideal sería poder realizar siempre una subdivisión ideal pero eso casi nunca es posible con las condiciones del problema, por consiguiente enfocaremos nuestro objetivo en aproximarnos lo más posible a dicha subdivisión.

3 Solución

3.1 Fuerza bruta

La primera idea intuitiva fue tomar las combinaciones de pares de tamaño k y computarlas todas, evaluar las áreas resutantes y quedarnos con la mejor solución. Evidentemente esto es altamente costoso, la cantidad de formas posible de distribuir k cortes en n vértices pasa por cuantas triangulaciones distintas se pueden formar en un polígono convexo está dado por $n-2$-ésimo número de Catalán $C_{n-2} = \frac{1}{n-1}\binom{2n-4}{n-2}$ por tanto todas las combinaciones posibles son $\binom{C_{n-2}}{n-(2+k)}$ quedando dicha solución en una complejidad no polinomial.

3.2 Acercamiento por camino de costo mínimo

La idea detrás de esto es ver el polígono convexo como un grafo cíclico C_n ponderar todas las aristas pertenecientes al ciclo con peso 0 y cada corte con peso $A_p - A_d$ donde A_p es el área del polígono y A_d es el área más pequeña que genera ese corte. Al usar la ponderación el camino de costo mínimo tiene peso en todas las aristas del área más pequeña que genera esa subdivisíon, por lo que indiscutiblemente si todos los caminos deberían usar exactamente k aristas ponderadas entonces el resultado sería la subdivisíon óptima. Esto generó un problema extremadamente grande, como ponderar los cortes? Sin discusión para lograr la ponderación deseada se necesitaba tener en cuenta que cortes se habían hecho anteriormente para asignarle el valor correcto. Por este motivo la ejecución de esa solución fue descartada tan pronto como llegó.

Analogamente pensamos en una solución por flujo, dado que el problema de ponderar las aristas es irracional tampoco se pudo obtener nada concreto por esa vía.

3.3 Greedy

Con varias ideas pensadas ya para solucionar el problema, habíamos pasado por deconstruir las triangulaciones posibles y parecía que las deconstrucciones aparentemente dejaban cortes con áreas siempre lo más cercanas posibles a $\frac{A_p}{k+1}$, pero si esto ocurre, por qué no simplemente ir construyedo la solución haciendo *cortes ideales*. La idea es demostrar que al hacer un corte de este tipo el subproblema debajo es de subestructura óptima. Había que demostrar que sea cual fuera el corte que se decidiera hacer con esa premisa pertenecía a una solución del problema (una subdivisión ideal).

Primeramente que sucedería si existen 2 cortes que modularmente están a la misma distancia del área óptima uno de ellos deja un área más pequeña, mientras el otro una más grande. Pero si es cierto que todo corte decidido con la premisa greedy pertenece a una solución del problema entonces da igual que corte escoger porque en ambos casos el criterio de la premisa es válido.

Al no tener conocimiento del resto de las divisiones siguientes a realizar en caso de que $k > 1$ porque solamente se esta teniendo en cuenta un corte por el

criterio greedy no es posible demostrar que es igual.

Si dado el criterio Greedy no se puede concluir en que el resto de la subestructura va en camino a una solución óptima entonces necesitabamos enfocarnos en una forma diferente de solucionar el problema, sin perder de vista que el caso ideal es picar el polígono original en $k+1$ polígonos de tamaño exactamente $\frac{A_p}{k+1}$ por consiguiente optamos por mezclar la idea greedy en un nuevo enfoque.

3.4 Solución propuesta

Para maximizar el área mínima w, utilizaremos una técnica de **búsqueda binaria** sobre el posible rango de w. Para cada valor candidato de w, verificaremos si es posible dividir el polígono en $k+1$ regiones con áreas $\geq w$ usando una programación dinámica eficiente.

Digamos que queremos particionar el polígono en $k+1$ áreas de al menos tamaño w. Sea i,j un par de vértices no adyacentes en el polígono, entonces se puede establecer un corte válido entre i,j. Necesitaríamos saber cuántas regiones de tamaño correcto se pueden obtener usando solamente este intervalo. El área contigua al corte será considerada *resto*.

Dado dos conjuntos de *cortes correctos* es óptimo seleccionar aquel que más cortes tiene, eso deja menos cortes a realizar en el *resto* del intervalo seleccionado. Si tienen la misma cantidad de *cortes correctos* entonces seleccionamos aquel que mayor *resto* tenga.

Este enfoque abre paso a un nuevo tipo de solución por programación dinámica. Definamos $dp_{i,j}$ como una tupla de $(c_{i,j}, r_{i,j})$ donde $c_{i,j}$ es la cantidad de cortes máximos y $r_{i,j}$ el resto máximo en el intervalo (i,j). Para calcular dp iteramos por todos los k con $i < k < j$ y también se considerará k como un vértice del *resto* del corte (i,j). Esto no es más que una transición entre el estado $dp_{i,k}$ y $dp_{k,j}$ es decir un subcorte por k del intervalo (i,j) y un nuevo corte si k formaba parte del resto para decidir que es lo mejor si picar en intervalo (i,j) o realizar un nuevo corte. Después de calcular dp podemos cortar por el intevalo dado si se cumple que el resto es al menos del tamaño deseado, es decir si $r_{i,j} \geq w$.

Dado lo planteado en esta solución si se cumple que $c_{1,n} > k+1$ entonces significa que podemos realizar k *cortes correctos* garantizando área menos al menos w.

La complejidad temporal de este algoritmo es $O(n^3(log(10^{16})))$, la cual explicaremos posteriormente.

3.5 Programación Dinámica (DP)

Definimos una tabla de programación dinámica $dp[i][j]$ que almacenará información sobre el intervalo de vértices (T_i, T_j) del polígono. Específicamente, $dp[i][j]$ será una tupla $(c_{i,j}, r_{i,j})$, donde:

- $c_{i,j}$: Número máximo de cortes válidos que se pueden realizar dentro del intervalo (i,j) tal que cada subregión resultante tenga un área $\geq w$.

- $r_{i,j}$: Área residual máxima restante en el intervalo (i, j) después de realizar los cortes.

3.5.1 Inicialización

Para cada par de vértices adyacentes (i, j), donde $j = i + 1$ (considerando la numeración cíclica del polígono), no se pueden realizar cortes adicionales. Por lo tanto:

$$dp[i][j] = (0, \text{ÁreaTotalPolígono})$$

3.5.2 Transición

Para un intervalo (i, j) con $j > i+1$, consideramos todos los posibles vértices k tales que $i < k < j$ para realizar un corte entre i y j pasando por k. Para cada k:

1. **Cortes en subintervalos:**
   - $dp[i][k]$: Información sobre el intervalo (i, k).
   - $dp[k][j]$: Información sobre el intervalo (k, j).
2. **Cálculo de cortes y residuo:**

$$c_{\text{total}} = c_{i,k} + c_{k,j} + 1 \quad (\text{el } +1 \text{ es por el corte } (i, j) \text{ mismo})$$

$$r_{\text{total}} = \min(r_{i,k}, r_{k,j})$$

3. **Actualización de $dp[i][j]$:**
   - Seleccionamos la opción que maximiza c_{total}. Si hay empate en c_{total}, elegimos la que maximiza r_{total}.

Formalmente:

$$dp[i][j] = \max_{i<k<j} \left\{ \ (c_{i,k} + c_{k,j} + 1, \min(r_{i,k}, r_{k,j})) \ \right\}$$

3.5.3 Condición de Corte Válido

Después de llenar la tabla dp, para el intervalo completo $(1, n)$:

- Si $dp[1][n].c \geq k$ y $dp[1][n].r \geq w$, entonces es posible realizar k cortes que cumplan con el requisito de área mínima w.

3.6 Búsqueda Binaria para Maximizar w

Dado que queremos maximizar w, utilizamos una búsqueda binaria sobre el rango posible de áreas w. El procedimiento es el siguiente:

1. **Determinar el rango de búsqueda:**
   - **Mínimo:** $w_{\min}$ = Área mínima posible (puede ser 0).
   - **Máximo:** $w_{\max} = \frac{\text{Área total del polígono}}{k+1}$.
2. **Búsqueda binaria:**
   - Mientras $w_{\max} - w_{\min}$ sea mayor que una precisión deseada:
     - $w_{\text{mid}} = \frac{w_{\min}+w_{\max}}{2}$.
     - Verificar si es posible realizar k cortes con $w = w_{\text{mid}}$ usando la DP descrita.
     - Si es posible, actualizar $w_{\min} = w_{\text{mid}}$; de lo contrario, actualizar $w_{\max} = w_{\text{mid}}$.
3. **Resultado:**
   - $w_{\min}$ convergerá al valor máximo de w que permite realizar los cortes requeridos.

3.7 Argumentación de Correctitud

- **Divisibilidad:** La programación dinámica asegura que todas las posibles formas de dividir el polígono se consideren, garantizando que se encuentra una solución óptima si existe.
- **Maximización del Área Mínima:** Al utilizar búsqueda binaria para maximizar w y verificar su viabilidad con DP, garantizamos que el w encontrado es el más grande posible que cumple con los requisitos.
- **No Cruce de Diagonales:** La definición de $dp[i][j]$ y la naturaleza recursiva de la DP aseguran que los cortes no se crucen fuera de los vértices, ya que cada subintervalo se maneja de manera independiente y combinada sin superposición indebida.

3.8 Complejidad Temporal

Analicemos la complejidad temporal de cada componente del algoritmo:

3.8.1 a. Programación Dinámica (DP)

- **Estados:** Existen $O(n^2)$ posibles intervalos (i, j) en el polígono. Esto se debe a que cada par de vértices puede ser un punto de inicio y fin de un intervalo, y dado que hay n vértices, la combinación de estos genera $\binom{n}{2}$ intervalos.

- **Transiciones por Estado:** Para cada intervalo (i, j), consideramos todos los vértices k tales que $i < k < j$. Dado que k puede tomar cualquier valor entre i y j, esto implica que hay hasta $O(n)$ transiciones por estado. Así, para cada intervalo, iteramos sobre todos los posibles vértices internos.

- **Cálculo por Transición:** Cada transición involucra un número constante de operaciones, como sumas y cálculos de mínimo. Por lo tanto, el costo de calcular el resultado para una transición es $O(1)$.

- **Total DP:** Combinando los $O(n^2)$ estados y las $O(n)$ transiciones por estado, la complejidad total para la programación dinámica es:

$$O(n^2 \cdot n) = O(n^3).$$

3.8.2 b. Búsqueda Binaria

- Para la búsqueda binaria, la complejidad temporal en el peor de los casos es $O(logn)$, donde n es el número de elementos de la matriz. Esto significa que, en el peor de los casos, el algoritmo solo tiene que comprobar log n elementos en la matriz, donde log n es el logaritmo en base 2 de n.

- **Número de Iteraciones:** Si el área total es A, y suponiendo que queremos una precisión de ϵ, la cantidad de iteraciones necesarias en la búsqueda binaria es $O(\log(\frac{A}{\epsilon}))$. Dado que A puede ser muy grande (por ejemplo, 10^{16}), esto se traduce en:

  $$O(\log(10^{16})),$$

  que es constante en términos de notación asintótica.

3.8.3 c. Complejidad Total

- **Total:** Combinando la complejidad de la programación dinámica con la búsqueda binaria, obtenemos:

  $$O(n^3 \cdot \log(A)),$$

  donde A es el área máxima posible (aquí estimado como 10^{16}).

Por lo tanto, la complejidad temporal del algoritmo es:

$$O(n^3 \cdot \log(10^{16})).$$