Universidad de Buenos Aires
Facultad De Ingeniería
Año 2019 – 2.do Cuatrimestre

(75.29 / 95.06) Teoría de Algoritmos

TRABAJO PRÁCTICO N.º 3
Grupo: "Los Fieles de Cardozo"
Reentrega: *13 de Noviembre de 2019*

Integrante	Padrón	Correo electrónico
Salvador, Yago	# 99 725	yagosalvador@gmail.com
Nenadovit, Emmanuel Angel	# 91 734	emmanuelnenadovit@gmail.com
Untrojb, Kevin	# 97 866	oliverk12@hotmail.com
Bocchio, Guido	# 101 819	guido@bocch.io

1. Resumen

En el presente informe se expone el análisis y resultados de cada consigna. La primera parte presenta un problema de geometría computacional e informática gráfica en el cual se utilizó la estrategia de dividir y conquistar para la resolución del problema. se expone la solución, el análisis del algoritmo utilizado y se programa la solución. La segunda parte consiste en optimizar y minimizar la cantidad de rotaciones de una caja desde un punto de inicio hasta un punto final. Se analizó varios casos del problema y finalmente se utilizó un algoritmo de programación dinámica para poder llegar a la solución óptima. La ultima parte es optativa, consiste en analizar y explicar el *paper "Fast detection of polyhedral intersection"* publicado por David P. Doeskin y David C. Kirkpatrick. Además se proporciona el pseudocódigo del algoritmo y proporcionar dos ejemplos completos de funcionamiento.

2. Rectángulo contenedor

Se desea hallar el rectángulo contenedor de un conjunto de puntos que forman un polígono convexo. Un enfoque directo del problema consiste en hallar cuatro puntos construidos a partir de aquellos puntos cuyas coordenadas x e y sean máximas y mínimas. Para esto se opta por recorrer una vez el arreglo de puntos dado encontrando los valores mínimos y máximos para las coordenadas x e y. Luego, se construye el rectángulo a partir de estos valores. En el Algoritmo 1 se muestra el pseudocódigo de esta solución. Encontrar el mínimo y máximo son operaciones $\mathcal{O}(n)$. De esto se tiene que algoritmo es $\mathcal{O}(n)$.

Además se propone una solución utilizando la estrategia de división y conquista. Esta se vale de que un polígono convexo consiste precisamente en una colección de puntos extremos para toda dirección. En el caso general se busca encontrar el punto que maximiza su componente en una coordenada genérica u (ya sea esta el eje x, el eje x considerado en sentido inverso, el eje y o cualquier vector deseado). La solución consiste en considerar que cada extremo buscado se encuentra en un segmento del polígono. Luego se considera un punto intermedio de ese segmento y se obtienen seis casos. Sean A y B los puntos extremos del segmento a considerar y C el punto intermedio. Se listan en la Figura 1 los casos posibles mencionados anteriormente. En el Algoritmo 2 se muestra el pseudocódigo de esta solución. Se muestra en la Ecuación (1) la recurrencia que modela esta solución. De esta recurrencia, utilizando el teorema maestro, se aprecia que la solución es de orden $\mathcal{O}(\log n)$.

La primera solución utiliza memoria constante para obtener el rectángulo contenedor y únicamente requiere tener acceso a los puntos del polígono. Por esto se obtiene que su complejidad espacial es constante: $\mathcal{O}(1)$. Sin embargo, la segunda solución realiza sucesivas llamadas recursivas. Puesto que divide el problema en partes iguales se concluye que la profundidad del árbol de llamadas es de orden $\mathcal{O}(\log n)$ y la complejidad espacial resulta del mismo orden.

La primer solución tiene una mayor generalidad puesto que no utiliza el hecho de que los puntos representan un polígono convexo funcionando para cualquier entrada.

Se desea analizar para qué cantidad de puntos la segunda solución es preferible. Como se deben obtener cuatro extremos, se tiene $T(n) \sim 4 \log_2(n)$. Para $2 < n < 16$ se observa:

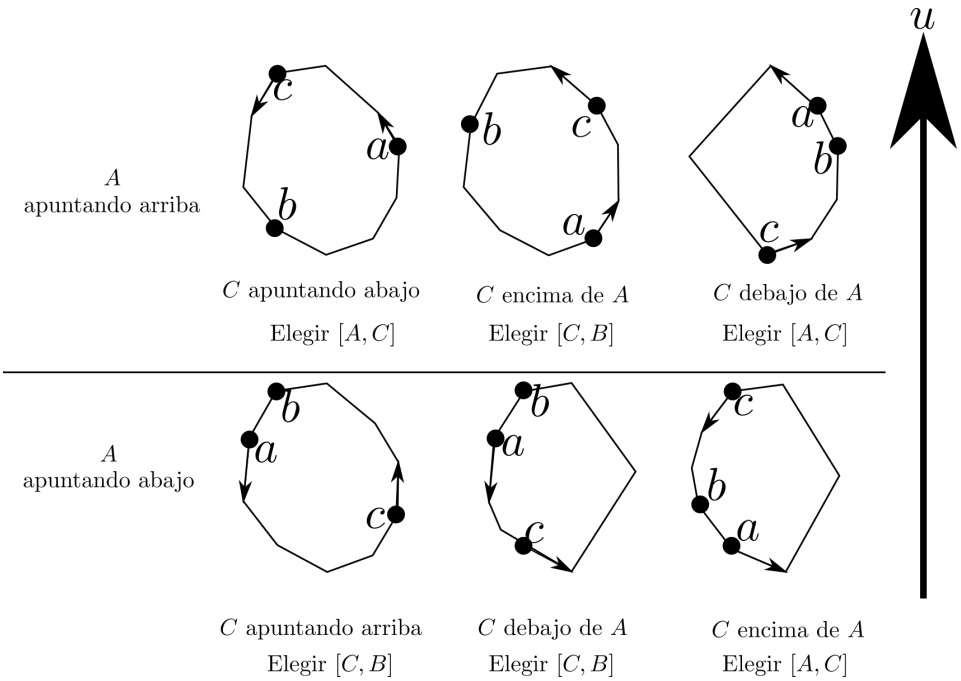

Figura 1: Casos posibles para determinar en que parte del segmento se encuentra el máximo.

$$4 \cdot 1 \leq 4\log_2(n) \leq 4 \cdot 4$$
$$4 \leq 4\log_2(n) \leq 16$$

Como para $n = 2$ se tiene $4\log_2(n) = 4$ y para $n = 16$ se tiene $4\log_2(n) = 16$, siendo ambas funciones monótonas crecientes, se concluye que para $2 < n < 16$, $n < 4\log_2(n)$. De esto, se tiene que cuando se tienen menos de dieciséis puntos, es preferible aplicar el primer algoritmo.

$$T(n) = \begin{cases} 1, & \text{si } n = 1 \text{ o } n = 2. \\ \mathrm{T(n/2)} + 1, & \text{en otro caso.} \end{cases} \tag{1}$$

Algoritmo 1 Solución inicial para obtener el rectángulo contenedor

```
Sea P un conjunto de puntos
x_mín ← −∞
x_máx ← ∞
y_mín ← −∞
y_máx ← ∞
para p en P hacer
    si p.x < x_mín entonces
        x_mín ← p.x
    si no
        si p.x > x_máx entonces
            x_máx ← p.x
    si p.y < y_mín entonces
        y_mín ← p.y
    si no
        si p.y > y_máx entonces
            y_máx ← p.y
rectángulo ← [(x_mín, y_mín), (x_mín, y_máx), (x_máx, y_máx), (x_máx, y_mín)]
devolver rectángulo
```

Algoritmo 2 Solución para obtener el rectángulo contenedor utilizando la estrategia de división y conquista

Sea P un conjunto de n puntos ordenados en sentido antihorario representando un polígono convexo
$x_{mín} \leftarrow$ Extremos(P, 0, n, $(-1, 0)$)
$x_{máx} \leftarrow$ Extremos(P, 0, n, $(1, 0)$)
$y_{mín} \leftarrow$ Extremos(P, 0, n, $(0, -1)$)
$y_{máx} \leftarrow$ Extremos(P, 0, n, $(0, 1)$)
devolver $[(x_{mín}, y_{mín}), (x_{mín}, y_{máx}), (x_{máx}, y_{máx}), (x_{máx}, y_{mín})]$
función EXTREMOS(P, A, B, u)
Sea P un conjunto de puntos ordenados en sentido antihorario representando un polígono convexo
Sean A y B dos puntos de P representando los extremos del segmento donde se encuentra el extremo buscado
Sea u un vector director para el cual se quiere encontrar el punto p en P que maximiza $p \cdot u$
si el segmento $[A, B]$ tiene un punto **entonces**
devolver A
si no si el segmento $[A, B]$ tiene dos puntos
devolver $\text{máx}_u\{A, B\}$
Sea C el punto de P entre A y B que genera dos segmentos $[A, C]$ y $[C, A]$ con la misma cantidad de puntos.
Sean V_A y V_C las aristas, tomadas de forma antihoraria, del polígono con origen en A y C respectivamente
si $V_A \cdot u > 0$ **entonces**
si $V_C \cdot u < 0$ o bien $C \cdot u < A \cdot u$ **entonces**
devolver Extremos(P, A, C, u)
si no
devolver Extremos(P, C, B, u)
si no
si $V_C \cdot u > 0$ o bien $C \cdot u < A \cdot u$ **entonces**
devolver Extremos(P, C, B, u)
si no
devolver Extremos(P, A, C, u)

3. Acomodando cajas

El enunciado pide construir un algoritmo que permite trasladar una caja de almacenamiento de dimensiones estándar de una ubicación inicial a una final determinada teniendo en cuenta que la caja es un prisma rectangular que mide 2 de alto x 1 de ancho x 1 de largo. La caja solo la puedan hacer girar en el piso en cualquiera de sus ejes. Se considera el depósito como una grilla de n filas por m columnas, tanto n como m valores enteros. Cada celda es de 1x1 metro. Algunas de las celdas podrían estar ocupadas por otras cajas que no podemos trasladar. La caja se encuentra inicialmente parada. Para resolver este problema, se puede modelar las celdas como un grafo y la caja se puede modelar con dos estados, parada y acostada.

Para cada estado posible la caja puede girar en cuatro direcciones ilustradas en la Figura 2

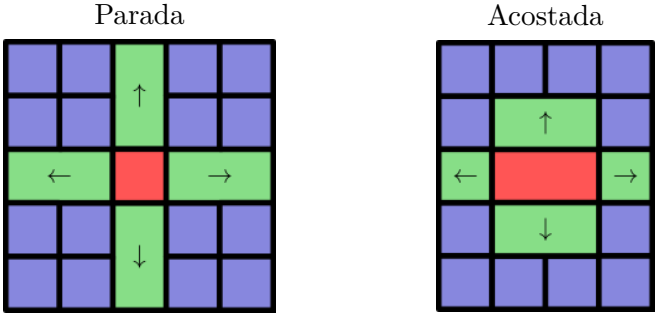

Figura 2: Posiciones en las cuales se puede girar una caja. La posición actual de la caja se indica en rojo. Las posiciones a las que esta puede moverse se indican en verde.

Se puede representar cada posición como un vértice en un grafo. En este grafo un giro es representado por una arista. Como cada giro es un proceso reversible el grafo resultante es no-dirigido. Dado un depósito de $n \cdot m$ celdas libres las posiciones en las que la caja se encuentra parada son, como mucho, $n \cdot m$, debido a que no pueden existir más de $n \cdot m$ celdas libres. Esto puede verse en la Figura 3. Asimismo, es posible listar todas las posiciones en las que la caja puede encontrarse acostada. Es posible ver que no hay más de $n \cdot m$ posiciones en las que la caja puede encontrarse acostada horizontalmente o verticalmente. Para llegar a esta conclusión basta con concluir que es posible asociar cada caja acostada horizontalmente a una única celda (por lo que no hay más de $n \cdot m$ posiciones en las que la caja se encuentra acostada horizontalmente). Esto mismo es válido para cajas acostadas verticalmente. Por lo tanto, hay como mucho, $2 \cdot n \cdot m$ posiciones en las que la caja puede encontrarse acostada. Finalmente las posiciones en las que la caja puede encontrarse son, como mucho, $\mathcal{O}(3(n \cdot m)) = \mathcal{O}(n \cdot m)$.

Se busca construir al grafo como una lista de adyacencia. Para construir el grafo se agregan todas las celdas del depósito (asumiendo que la caja está parada en ellas) como nodos por visitar del grafo a una cola. Por cada posición se lista las cuatro posiciones a las que la caja puede girar (como se ve en la Figura 2). En caso de encontrar un obstáculo en alguna celda de estas posiciones, se ignora dicha posición. En caso de no haber sido visitadas se añaden estas posiciones a la cola de posiciones por visitar. En caso de no existir una arista entre la posición actual y alguna de las posiciones listadas, esta es creada. Esto puede realizarse en $\mathcal{O}(n \cdot m)$ puesto que obtener las posiciones a las que una caja puede girar son constantes y la creación de una arista se realiza en $\mathcal{O}(1)$. Se puede ver en la Figura 4 un ejemplo de los primeros pasos en la creación de dicho grafo. Puede verse en el Algoritmo 3 el pseudocódigo para la creación del grafo.

Algoritmo 3 Algoritmo para la construcción del grafo

```
Sea C el conjunto de todas las celdas del depósito
Sea Q una cola de celdas por visitar
Sea V un conjunto de posiciones visitadas
Sea P el conjunto de posiciones ya agregadas a la cola al menos una vez
para cada celda c de C hacer
    Agregar posición(c, parada) a la cola Q
    Agregar posición(c, parada) a P
mientras la cola Q tenga elementos hacer
    c ← extraer un elemento de la cola Q
    H ← conjunto de posiciones vecinas a las que c puede girar
    para h en H hacer
        si h no está en V entonces
            Conectar c y h
            si h no está en P entonces
                Agregar h a Q
                Agregar h a P
    Agregar c a V
devolver lista(V)
```

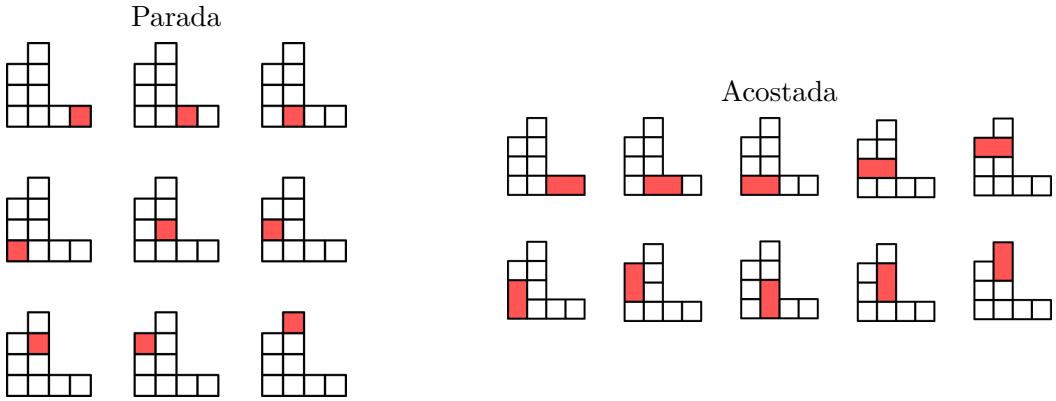

Figura 3: Ejemplo de un depósito y todas las posiciones en las que la caja se puede encontrar acostada en él.

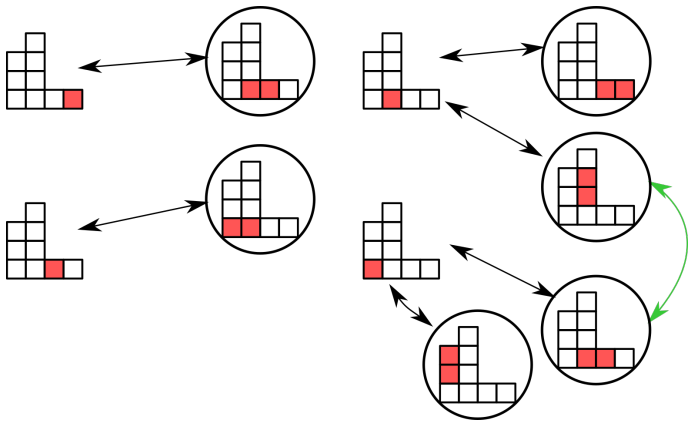

Figura 4: Primeros pasos en la creación de un grafo para el depósito dado. La arista marcada en verde se construirá posteriormente pero existe en el grafo final.

Una vez armado el grafo, se resuelve utilizando programación dinámica. Se utiliza la siguiente ecuación de recurrencia:

$$D(v,w,n) = \begin{cases} 0, & \text{si } v = w. \\ 1, & \text{si } peso(v,w) = 1. \\ \infty, & \text{si } n = 0. \\ \min\{D(v,w,n-1), \min\limits_{u \in U}\{D(v,u,n-1) + peso(u,w)\}\}, & U \text{ vértices alcanzables desde } v. \end{cases}$$

Dado un vértice de inicio y un vértice final el camino más largo no puede constar de más aristas que la cantidad de vértices del grafo. Simplemente se actualiza la distancia de inicio a final siempre que se encuentre una distancia más corta a un vértice intermedio que esté conectado con el final. Esto se debe realizar $|V| - 1$ veces.

Algoritmo 4 Coste mínimo para mover la caja

```
Construir el grafo
Sea peso(v, w) = 1 de existir arista entre v y w o ∞ en otro caso
Sean V las posiciones
n ← |V| − 1
devolver Distancia(posición inicial, posición final, n)

función DISTANCIA(v, w, n)
    si v = w entonces
        devolver 0
    si n = 0 entonces
        devolver ∞
    para cada posición u a la que se puede acceder desde v hacer
        si peso(u, w) < ∞ entonces
            D_u ← Distancia(v, u, n − 1) + peso(u, w)
        si no
            D_u ← ∞
    devolver mín{Distancia(u, w, n − 1), mín_u{D_u}}
```

Referencias

[1] David P. Doeskin & David C. Kirkpatrick *"Fast detection of polyhedral intersection"*. `https://www.cs.jhu.edu/ misha/Spring16/Dobkin83.pdf`

[2] Jon Kleinberg, Éva Tardos *Algorithm Design* . Addison Wesley,2006.