Breves Notas sobre
Aplicaciones de la Computación

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Febrero 2008

Índice general

Prefacio

Las *Breves Notas sobre Aplicaciones de la Computación* presentan en forma simple y sencilla algunas aplicaciones relevantes de la Computación. No tienen la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que, por su relevancia, se consideran esenciales para el conocimiento básico de esta área, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: Simulación, Curvas Spline, Visión por Computadora, Recursión, Escaneo CAT, La Transformada Rápida de Fourier, Almacenamiento de Imágenes y Programación Lineal. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe comprender en las asignaturas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Febrero 2008

Capítulo 1

Simulación
El Método Monte Carlo

En la búsqueda de entender muchos de los sistemas que comprenden el mundo real, cada vez se recurre más a la simulación por computadora. Si el sistema es natural o artificial, frecuentemente uno o más de sus componentes tienen un comportamiento tan complejo que la única aproximación posible a tal comportamiento es suponerlo aleatorio. La técnica obvia es reemplazar el componente complejo por componentes sencillos que siguen una ley estadística. Por ejemplo, si se intenta simular un juego de voleibol entre dos equipos igualmente capaces, se podría hacer un modelo sencillo del juego mediante contínuamente lanzar una moneda: si un equipo "sirve" y si el lado de la moneda está en su contra, entonces el otro equipo sirve; de otra manera, se le concede un punto. La operación de lanzar una moneda al aire recuerda a un procedimiento aleatorio simple, que se conoce con el nombre de "técnica Monte Carlo".

Aquí, se utiliza el ejemplo de un banco como un sistema simple para ilustrar las principales características de una simulación Monte Carlo. Considérese, por ejemplo, un banco con un cajero. Los clientes entran casi de forma aleatoria, y forman una cola enfrente del cajero. La figura 1.1 es un esquema que representa esta situación, como un indicativo de cómo varios elementos de los sistemas del mundo real pueden abstraerse. En una simulación bien diseñada, mucho de los detalles inherentes del sistema original se ignoran con poco o nada de penalización; el comportamiento del modelo abstracto (cuando se implementa en una computadora) es indistinguible del sistema real, respecto a lo que concierne a los parámetros bajo investigación.

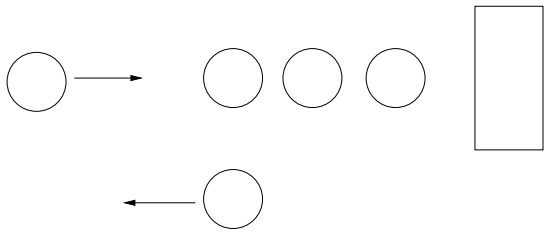


Figura 1.1: Esquema que representa a los clientes formando una cola en el banco.

Por supuesto, cada simulación, ya sea inspirada científica o industrialmente, debe tener un objetivo, y el objetivo determina qué aspectos del sistema que se modela son relevantes. En el caso de la simple simulación de la cola de un banco, podría tratarse simplemente de decidir si "valdría la pena" añadir otro cajero.

Algunas consideraciones teóricas y estudios empíricos sugieren que un sistema real de servicios bancarios, mientras se encuentre en un estado estable, tiende a recibir clientes con un patrón regular de llegada. Dado un tiempo promedio de α segundos entre llegadas, la distribución de tales tiempos siguen una forma exponencial negativa, como se muestra en la figura 1.2.

Como cualquier función de densidad, esta función debe ser interpretada con cuidado. Para cada tiempo posible t entre dos llegadas consecutivas (tiempo entre llegadas), la función f provee lo que se llama "densidad entre llegadas", pero **no** la frecuencia entre llegadas como tal. Se puede sólo tomar un rango de tiempos entre llegadas, por ejemplo entre t y t', y calcular el área bajo la curva. Esto representa el número relativo a las veces que se puede esperar un tiempo entre llegadas en el rango entre t y t'. Así, si se desea "predecir" para 100 clientes consecutivos qué tan frecuentemente un tiempo entre llegadas ocurre en ese rango, simplemente se calcula el área A bajo la curva que se muestra en el diagrama, y se obteniene $A \times 99$, donde 99 es obviamente el número de llegadas.

Pero las consideraciones anteriores no muestran cómo generar llegadas en una computadora. Para ello, se requiere una función de distribución acumulativa F, que simplemente es la integral de f (figura 1.3). Aquí, un tiempo entre llegadas t arroja una frecuencia acumulada $F(t)$, es decir, la porción de llegadas que tienen tiempos entre sí de t o menos. Se puede hacer una

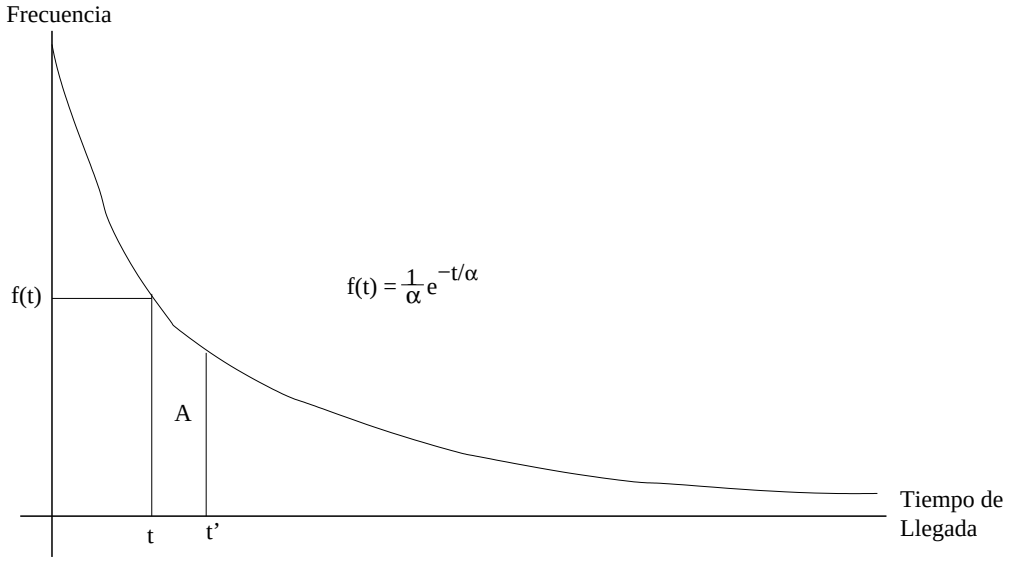


Figura 1.2: La distribución exponencial negativa.

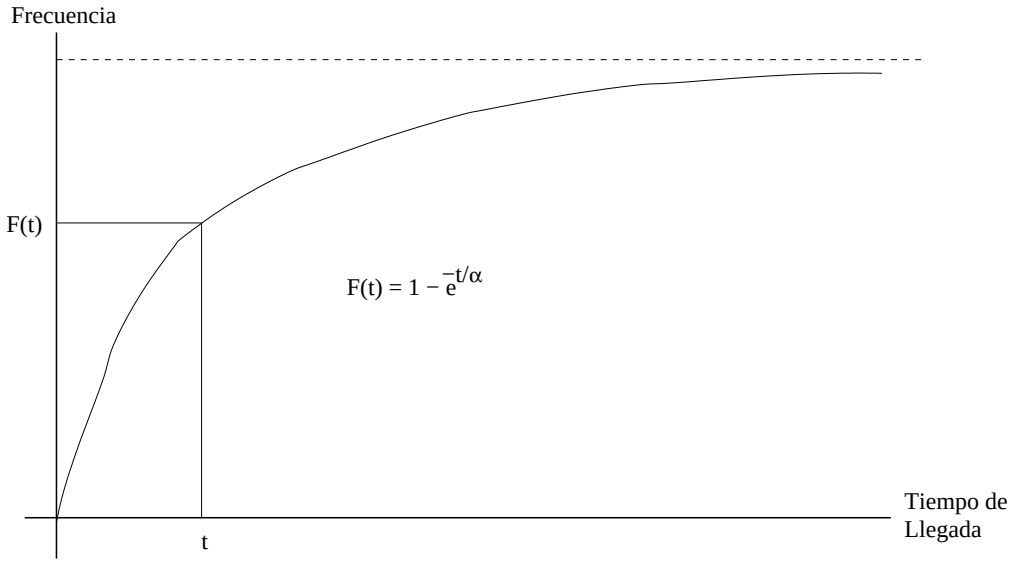


Figura 1.3: Función acumulativa de la función exponencial negativa.

observación más interesante acerca de la función F. Si se seleccionan números aleatorios x entre 0 y 1, y se calcula $F^{-1}(x)$, las cantidades resultantes reproducen la función de densidad original f. En términos prácticos, esto significa que si se seleccionan 1000 números aleatorios en el rango de 0 a 1, y cada uno se opera mediante la inversa de la función F^{-1}, entonces el histograma de las cantidades resultantes tienden a tener la misma forma que la curva de densidad de f.

Esta observación permite generar llegadas aleatorias para la simulación del banco mediante el siguiente algoritmo:

1. Generar el primer "cliente".
2. Seleccionar un número aleatorio x entre 0 y 1.
3. Obtener $F^{-1}(x)$.
4. Permitir que pasen $F^{-1}(x)$ segundos.
5. Generar el siguiente "cliente".
6. Regresar al paso 2.

Los números aleatorios son el problema más difícil aquí. Pueden generarse mediante muchas de las técnicas sugeridas por varios autores. Para la función F que se utiliza, se puede mostrar que:

$$F^{-1}(x) = \alpha \, ln(1 - x)$$

Una de las ventajas de la simulación por computadora es que los experimentos que podrían tomar días o meses de tiempo en los sistemas reales (y suponiendo que se permita entrometerse con ellos) pueden llevarse a cabo en una computadora en segundos, minutos, o (en el peor caso) horas de tiempo real. Esto se debe a que el flujo del tiempo también se simula, en la mayoría de los casos usando un reloj de la simulación, que no es más que una variable (C, por ejemplo), cuyo incremento va llevando la cuenta del tiempo. Así, al inicio de la simulación, en el paso 1, se tiene que $C = 0$. Si el tiempo entre llegadas generado en el paso 3 es de 137 segundos, entonces se añade en el paso 4 el valor 137 a C, actualizando el reloj de la simulación.

El paso 5, que genera el siguiente "cliente", significa simplemente que se debe añadir un nuevo "token", representando al cliente que se pasa a una estructura de datos que representa a la cola frente al cajero. Resulta común

que una simulación tan sencilla como esta no requiere representar la cola por algo tan elaborado como un arreglo o una lista. Es sucifiente que la cola Q meramente registre cuántos clientes "actualmente" ocupan la cola. Así, cuando la computadora se encuentra en el paso 5, nada más añade un 1 a Q.

Los clientes salen de la cola tan pronto como han sido servidos por el cajero. Aun cuando el tiempo de servicio que se encuentra en los bancos realmente no tiene una distribución simple, se supone éste también sigue una cierta distribución exponencial G, en este caso con un valor constante β que representa el tiempo promedio de servicio del cajero por cada cliente.

Se está ahora en posición de escribir el mismo tipo de programa para tiempos de servicio como se hace anteriormente para tiempos de llegada. Ahora, sin embargo, dos tipos de operación deben entretejerse exitosamente para que cada paso se siga en secuencia lógica como se muestra en la figura 1.4. Una característica esencial de la aproximación aquí es obtener el tiempo al siguiente evento (llegada de cliente o terminación de servicio) mediante el uso de dos variables de tiempo TA y TS, respectivamente.

En la figura 1.4, nótese como ambos TA (tiempo a la siguiente llegada) y TS (tiempo al final del servicio actual) actúan como relojes. Si TA es menor que TS, entonces una llegada sucede antes, y se debe incrementar el reloj C por una cantidad de TA, y decrementar TS por la misma cantidad. Después de añadir la nueva llegada a la cola Q, se genera el siguiente tiempo de llegada y se reinicia TA de acuerdo con ello. Del otro lado del diagrama, los papeles de TA y TS obviamente se invierten.

Esta aproximación se conoce como "técnica de evento crítico" (*critical-event technique*). Simplemente se incrementa el reloj de la simulación al tiempo del siguiente cambio en el sistema, y entonces se realiza el cambio indicado. Este es el caso al alterar la longitud de la cola. Otra aproximación para el manejo del tiempo en simulaciones por computadora se conoce como "método de la fracción de tiempo" (*time-slice method*). Este método involucra la selección de un pequeño y fundamental incremento en el tiempo, que continúa actualizando el reloj por su incremento, y barriendo el sistema, obteniendo el nuevo estado en base a su estado actual.

El punto central del método Monte Carlo es la generación de nuevos eventos mediante la técnica de la función inversa. Virtualmente cualquier

distribución puede utilizarse, ya sea teórica, en la forma de una función computable, o empírica, en la forma de un histograma acumulativo. Con una estructura de programa algo más complicada que la anterior, cualquier número de cajeros puede modelarse. De hecho, se pueden simular una multiplicidad de colas (como las colas de banco).

Existen lenguajes de propósito especial como GPSS que pueden utilizarse para modelar sistemas de múltiples colas y varios otros tipos de estructuras. Sin embargo, es muchas veces más fácil escribir programas de simulación en Fortran o Pascal, aunque muchas veces los tiempos de ejecución tienden a ser muy largos.

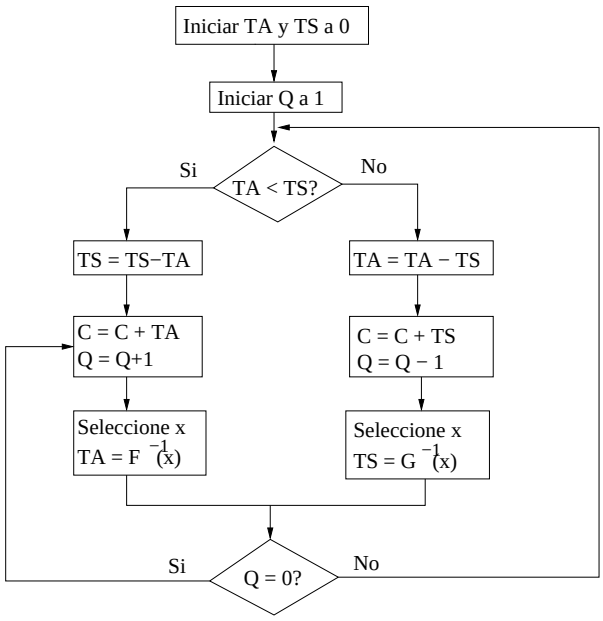


Figura 1.4: Diagrama de bloques de la simulación de un banco.

Capítulo 2

Curvas Spline
Interpolación Suave

Observar un plotter describir una curva spline es entrar en un mundo mágico donde el algebra más sencilla cobra vida (figura 2.1). Varios puntos particulares sobre la superficie de paper del plotter se entrelazan, uno tras otro, mediante una curva sinuosa y aparentemente natural. La pluma del plotter nunca duda. Conforme se aproxima a un punto, se mueve a la izquierda, pareciendo por un momento que va a perder el punto completamente. Pero es entonces cuando se dirige más y más hacia el punto, entrando finalmente por una dirección inesperada. De repente, la razón del errático comportamiento es claro: el siguiente punto se encuentra precisamente hacia la derecha del punto que se ha atravesado.

En graficación por computadora, análisis de datos, y muchas otras aplicaciones, se desea una curva "natural" que conecte un número de puntos. La aplicación gráfica, por ejemplo, puede bien involucrar el dibujo de una figura definida por una colección de puntos. El perfil de una cara puede inicialmente generarse por una computadora como un conjunto de 25 puntos. Un programa de dibujo de curvas spline conecta los puntos, por así decirlo, mediante una curva que no parece en nada una cara. En muchas situaciones, los científicos miden cantidades que varían contínuamente como presión de aire, tiempo de respuesta, o campo magnético, mediante un conjunto relativamente pequeño de mediciones discretas. El valor de las cantidades entre las mediciones es estimado mediante interpolación; una curva spline se dibuja, que (ojalá) coincida con las mediciones no hechas. Así, los datos son expuestos como una curva entera.

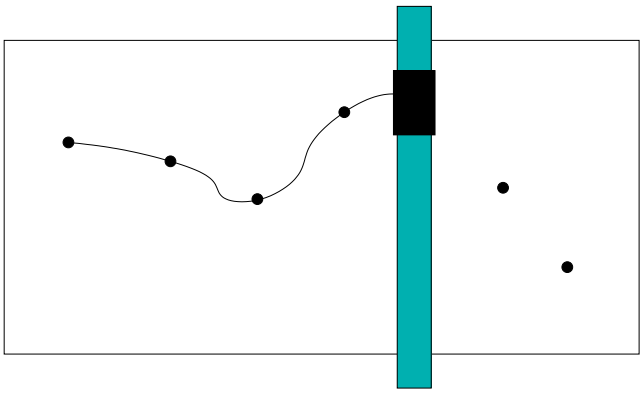


Figura 2.1: Un plotter dibuja una curva spline.

Los puntos que deben ser conectados se conocen como "vértices de control" (*control vertices*). Pueden literalmente estar en cualquier parte del plano, y naturalmente, todos tienen coordenadas:

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$

Un programa spline conecta los puntos en pares consecutivos mediante una función con la forma:

$$f(t) = (x(t), y(t))$$

En otras palabras, la función spline f es en realidad dos funciones por separado $x(t)$ y $y(t)$. La variable independiente t es un parámetro; conforme t va de un valor inicial t_0 a su valor final t_1, la curva se dibuja como una sucesión más fina de puntos $(x(t), y(t))$. Los puntos en la curva se generan al operar cada función para cada valor de t, y entonces dibujar el punto resultante. Ya que las funciones x y y no tienen ninguna relación entre sí, pueden estudiarse por separado. En lo que sigue, por ejemplo, se examina solamente $x(t)$, considerando que la teoría y las técnicas desarrolladas se aplican igualmente a $y(t)$.

De entre las funciones que pueden ser candidatas para ser $x(t)$, los polinomios son los más sencillos y fáciles de operar. También ofrecen cierta flexibilidad, sinuosidad y la elusiva propiedad de "naturalidad". Pero, ¿qué polinomio usar? Obviamente un polinomio lineal es demasiado simple: produce curvas angulares de conexión de puntos. ¿Qué tal polinomios cuadráticos?

Estos son ciertamente capaces de proveer curvatura. Y sin embargo, no son lo suficientemente flexibles. Esto tiene que ver con la forma en que segmentos individuales de la curva spline coinciden con los puntos de control.

El siguiente candidato obvio para la función $x(t)$ es el polinomio cúbico o de tercer grado. Su forma general es:

$$x(t) = a + bt + ct^2 + dt^3$$

donde t va de 0 a 1. Para ser más específico, se supone que x está en el proceso de determinar las primeras coordenadas de puntos de la curva entre los vértices de control (x_i, y_i) y (x_{i+1}, y_{i+1}). La pregunta es: ¿qué valores de a, b, c y d toma la curva del i-ésimo al $(i+1)$-ésimo punto de control? Para encontrar esto, se puede substituir $t = 0$ y $t = 1$ en la fórmula anterior. Ya que $x(0) = x_i$ y $x(1) = x_{i+1}$, surgen inmediatamente dos condiciones:

$$\begin{aligned} x_i &= a \\ x_{i+1} &= a + b + c + d \end{aligned}$$

Obviamente, existe aún un gran número de posibilidades para escoger los coeficientes b, c y d. Cualquier combinación de b, c y d igual a $x_{i+1} - a$ parece adecuada. Sin embargo, los segmentos de la curva deben hacer algo más que meramente conectar los vértices de control; deben hacerlo suavemente. Es decir, la curva no debe parecen "doblarse" en estos vértices. Tiene sentido, entonces, restringir a que los segmentos del spline deben tener la misma pendiente en (x_i, y_i). La aproximación más directa para cumplir esta restricción es especificar la pendiente, dando lugar a otras dos condiciones. Ya que las derivadas de $x(t)$ en $t = 0$ y $t = 1$ son respectivamente s_i y s_{i+1}, se substituyen $t = 0$ y $t = 1$ en la derivada de la función spline para obtener:

$$\begin{aligned} s_i &= b \\ s_{i+1} &= b + 2c + 3d \end{aligned}$$

Considerando las cuatro ecuaciones anteriores, es ahora posible ajustar la función spline a voluntad. Estas condiciones se expresan como cuatro ecuaciones lineales con cuatro incógnitas. Su solución es directa:

$$
\begin{aligned}
a &= x_i \\
b &= s_i \\
c &= 3(x_{i+1} - x_i) - 2s_i - s_{i+1} \\
d &= 2(x_i - x_{i+1}) + s_i + s_{i+1}
\end{aligned}
$$

Dados los valores x_i, x_{i+1}, s_i y s_{i+1}, es posible encontrar los coeficientes de la curva cúbica que comprende el i-ésimo segmento del spline. Tales coeficientes pueden bien ser llamados a_i, b_i, c_i y d_i.

Se podría terminar el tema de curvas spline aquí. Sin embargo, algunas decisiones deben hacerse todavía para utilizar esta técnica: ¿cómo se escogen las pendientes s_i?, ¿para qué valores de t las curvas pasan por los vértices de control?

Primero, las derivadas s_i pueden seleccionarse por una variedad de medios: pueden proponerse por inspección visual de los vértices de control, por ejemplo, pero una técnica más objetiva es ajustar una curva cuadrática sencilla a las tres coordenadas x_{i-1}, x_i y x_{i+1}. Esta curva no se encuentra en el plano de la curva spline, sino en un plano diferente que se conoce como "espacio paramétrico". Los tres puntos (t_{i-1}, x_{i-1}), (t_i, x_i) y (t_{i+1}, x_{i+1}) determinan una parábola. Esta puede obtenerse fácilmente mediante el proceso de substitución y la solución descrita anteriormente; meramente, se inicia con una fórmula cuadrática como $a + bt + ct^2$.

Pero, ¿qué valores de t se supone dan origen a las coordenadas x_i? De nuevo, varias aproximaciones pueden sugerirse. En la técnica llamada "splines cúbicas uniformes" (*uniform cubic splines*), los valores de control de t_i se distribuyen espacialmente uniformemente sobre el intervalo. Supóngase, por ejemplo, que t va de 0 a 1. Si la curva spline tiene n segmentos, entonces el i-ésimo segmento debe dibujarse conforme t varía de $(i-1)/n$ a i/n. Por supuesto, los valores a_i, b_i, c_i y d_i se ven fuertemente afectados por el dominio donde t varía dentro del segmento generado. Así, los coeficientes deben re-obtenerse con los valores $(i-1)/n$ y i/n en lugar de 0 y 1 utilizados anteriormente. Sin embargo, el procedimiento es exactamente el mismo. La aproximación uniforme divide, en cierto sentido, cuando hay grandes espacios entre ciertos pares consecutivos de vértices de control. En tal caso, se puede elegir un espaciado no-uniforme basado en la distancia euclideana entre vértices sucesivos.

Toda la teoría de curvas spline apenas ha sido superficialmente expuesta aquí. Por ejemplo, el análisis de $x(t)$ puede extenderse no sólo a $y(t)$, sino también a una tercera dimensión $z(t)$. De tal modo, una curva spline en un espacio tri-dimensional puede generarse. Otras técnicas puede surgir mediante requerir que tanto la primera como la segunda derivadas deban coincidir en los vértices de control.

Capítulo 3

Visión por Computadora
Escenarios Poliédricos

La forma en que los humanos perciben, analizan y clasifican las imágenes a las que se les confronta en la vida diaria continúa siendo un misterio. Aun cuando la retina del ojo y el primer conjunto de células en la corteza visual del cerebro parecen descomponer una imagen vista en segmentos de línea, bordes, y otros elementos pictóricos simples, no se sabe casi nada sobre cómo el cerebro analiza los escenarios, cómo detecta un camino de entrada o cómo reconoce el rostro de un amigo.

¿Cómo, por ejemplo, hace sentido la imagen de la figura 3.1 para el cerebro humano? Antes de describirlo como un "arco sobre una plataforma", o como un "templo antiguo", o cualquier otra cosa, seguramente se ha reconocido que la estructura representada en la figura 3.1 se compone por cinco elementos por separado: dos placas horizontales, dos columnas verticales y una cubierta superior. Al menos, este es un razonamiento interpretable, y hay razones para suponer que tal análisis forma una parte esencial del proceso de reconocimiento interno humano.

La imagen mostrada representa un ejemplo de lo que se conoce como un escenario poliédrico, es decir, un ensable de sólidos cada uno limitado por superficies planas. Las superficies de estos sólidos se conectan mediante segmentos rectilíneos con una característica geométrica, y mostrando solo un número finito de relaciones donde dos o más se conectan. Por ejemplo, en la figura 3.1, hay esencialmente cinco tipos de relaciones. Estas se muestran en la figura 3.2 de manera esquemática, y dando nombres descriptivos que se utilizan después.

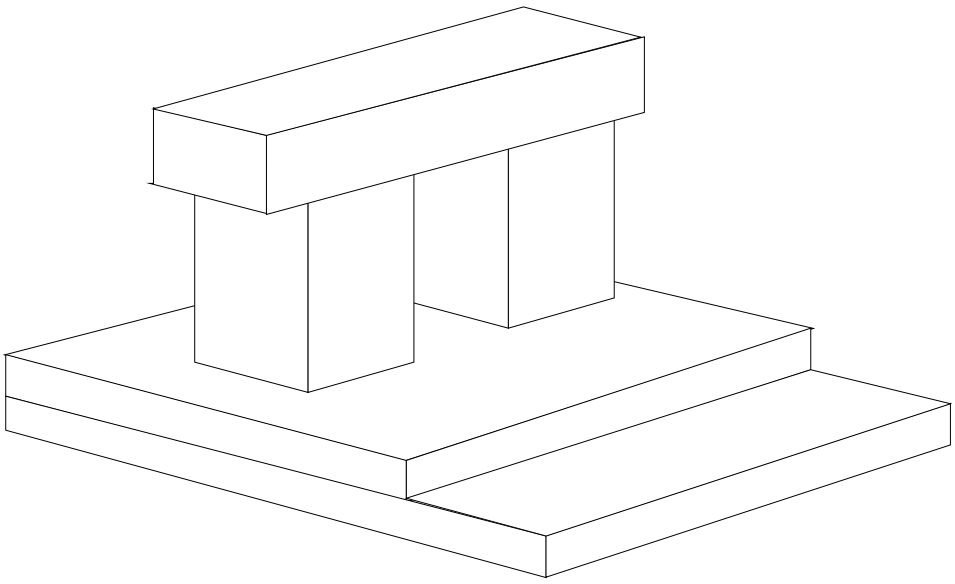


Figura 3.1: Una imagen.

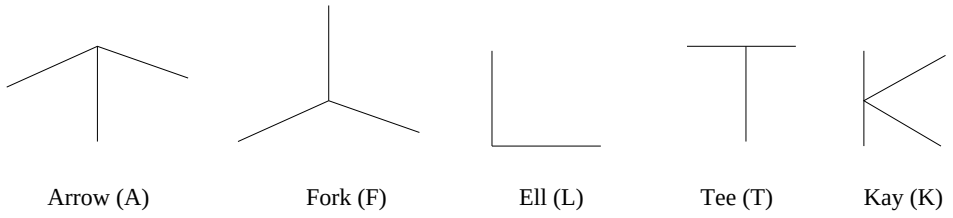
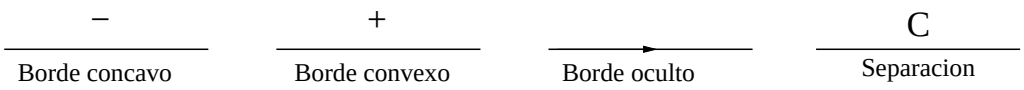


Figura 3.2: Los segmentos rectilíneos se conectan en cinco relaciones.

Es fascinante reconocer que es posible analizar imágenes de escenarios poliédricos mediante una computadora y, basado sólo en información sobre uniones de los tipos que se muestran aquí, llegar a una decisión automática por separado sobre los sólidos que componen el escenario. El trabajo de David Huffman y Maxwell Clowes fue un gran paso en el desarrollo de tal análisis durante los 1970s. Esta labor fue retomada y extendida por David Waltz en el MIT. Inicialmente, se describe a continuación una versión resumida de la aproximación de Waltz, analizando escenarios sin sombras.

Para utilizar exitosamente los tipos de uniones de la figura 3.2 en un

programa de análisis de escenarios, deben etiquetarse de acuerdo con el tipo de bordes o separaciones que dan origen a los segmentos de línea que los componen. En los tipos de escenarios (temporalmente) bajo consideración, sólo cuatro clases de etiquetas pueden asignarse a los segmentos de recta. Cada uno de ellos corresponde a alguna realidad física:



Por ejemplo, un borde cóncavo es una línea recta por la cual dos superficies se encuentran en un ángulo de menos de 180 grados desde el punto de vista relativo del observador.

Usando esta etiquetas, es posible desarrollar una lista extendida de uniones en términos del tipo de bordes y separaciones como se muestra en la figura 3.3. Esta lista no es mucho más larga que la lista original, ya que la física y geometría de estructura posibles ponen severas restricciones sobre qué bordes o separaciones puedan encontrarse en un tipo especial de unión. Por ejemplo, es difícil ver cómo una unión *fork* pudiera surgir a partir de concurrencia de tres bordes ocultos, es decir, bordes cada uno de los cuales representa el límite visual de un sólido particular dentro del escenario.

Esta lista de etiquetas para las uniones está en cierto modo incompleta, pues específicamente excluye ciertas configuraciones degeneradas o poco comunes. Sin embargo, aun con esta lista, el éxito del programa de análisis de Waltz está virtualmente asegurado. La razón de esto recae en la interacción de las uniones que comparten un segmento rectilíneo común: no cambia su identidad física al ir de una unión a otra.

Por ejemplo, si una F comparte un segmento de recta con una T, formando su "esquina superior derecha", hay muchas combinaciones posibles apriori. De estas, sólo cuatro pueden realmente ocurrir respecto a la lista de la figura 3.3:

$$(F2, T1), (F2, T5), (F3, T3), (F3, T4)$$

El programa de Waltz aprovecha este tipo de interacciones. Comenzando con una lista de todas las posibles etiquetas en cada unión, el programa selecciona una específica considerando una unión vecina, y reduciendo la lista de posibilidades para ambas uniones mediante eliminar aquella etiqueta de la unión que no puede coincidir con la otra. Al dirigirse a otra tercera

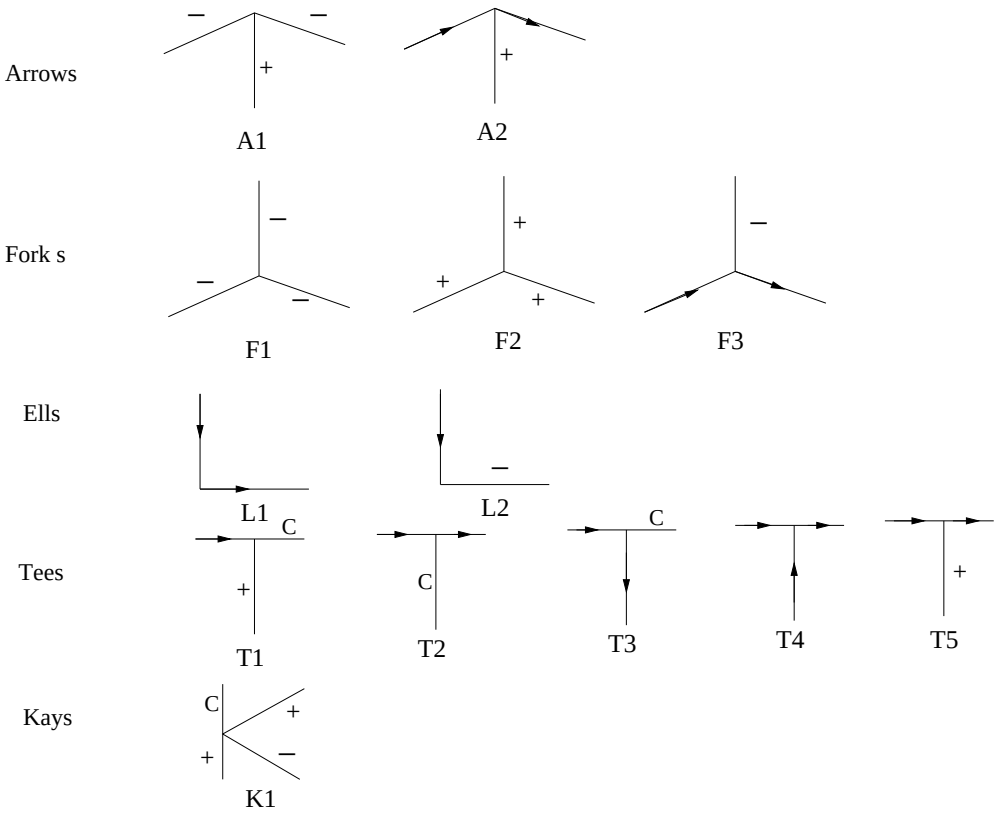


Figura 3.3: Trece tipos de uniones.

unión, este efecto es frecuentemente mejorado por la lista de la segunda unión, que ya ha sido reducida por la interacción con la primera unión.

Como ejemplo de este esquema general, se re-examina la imagen del escenario poliédrico de la figura 3.1, colocando etiquetas A, F, L, T ó K junto a cada una de sus uniones. El análisis comienza en una unión arbitraria, que se marca con una flecha en la figura 3.4.

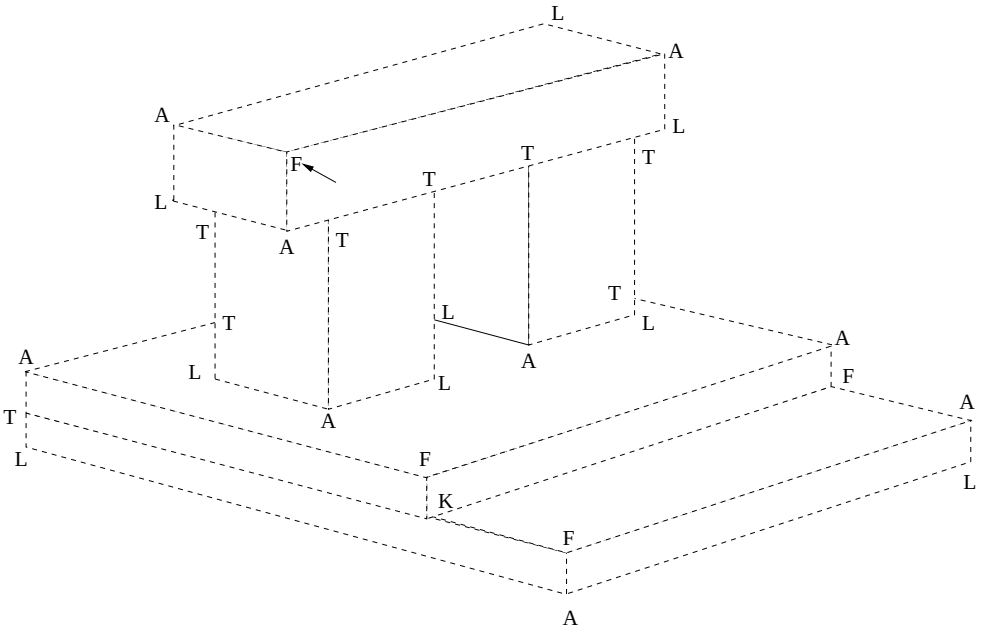


Figura 3.4: Incio del análisis.

Esta únión es una F, e inicialmente tiene tres etiquetas F1, F2 y F3 en su lista. Abajo de la F hay una unión A con etiquetas A1 y A2 en su lista. Cuando se intenta etiquetar el segmento que conecta estas dos uniones, sin embargo, el programa descubre que solo + funciona, lo que resulta en una reducción de la lista de uniones F de dos posibilidades a una sola: F2. Hasta ahora, la lista de uniones A no ha sido reducida: los dos bordes que forman el ángulo pueden ser ocultos o cóncavos, hasta lo que el programa conoce. Sin embargo, al considerar en seguida las uniones T, el programa reconoce rápidamente que no hay forma de hacer coincidir A1, la unión con bordes cóncavos, con ninguna de las uniones T, ya que esta lista no permite bordes cóncavos. Esto forza a la reducción de la lista de A a un solo miembro, A2.

Esta situación actual puede observarse en el detalle mostrado en la figura 3.5.

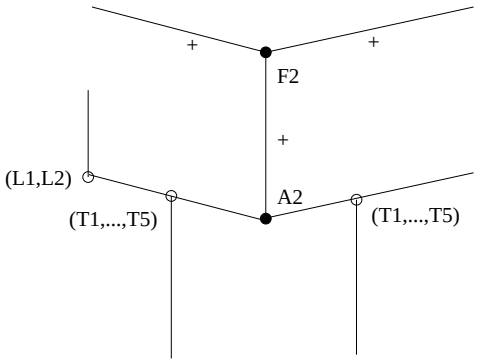


Figura 3.5: El análisis determina las primeras dos uniones.

Cuando el programa llega a examinar la unión T a la izquierda de la unión A2, no es capaz de reducir la lista de T completamente debido a que cada T en esta lista tiene una etiqueta A en uno de sus segmentos. Si, sin embargo, en un punto posterior el programa analiza la unión L adyacente a esta unión T, intentará reducir la lista de L, pero no tiendrá éxito tan solo utilizando la información acerca de la uniones adyacentes. Esta ambigüedad particular se elimina por un dispositivo propuesto por el propio Waltz: reconocer que esta unión en particular recae en el límite visual de la imagen entera. Por lo tanto, no puede ser incidente con ningún borde cóncavo que forme parte de tal límite, lo que elimina a L2 de la lista.

Procediendo de esta forma de unión a unión, el programa eventualmente llega a generar un conjunto de etiquetas para toda la imagen, como se muestra en la figura 3.6.

El programa de Waltz intenta producir una etiqueta para todos los segmentos en la imagen del escenario poliédrico. Daría como resultado el conjunto que se muestra en la figura 3.6. En todos excepto 15 casos, los segmentos han sido etiquetados con una etiqueta única. En el resto de los casos, el programa no puede decidir sobre algunas alternativas. Ciertamente, algunas de estas ambigüedades son perfectamente razonables dado el poco conocimiento del escenario en particular o en escenarios poliédricos en general. De hecho, el programa ignora relaciones de soporte, y en lugar de decidir que el fondo de las columnas son cóncavos, presume que bien pueden ser bordes

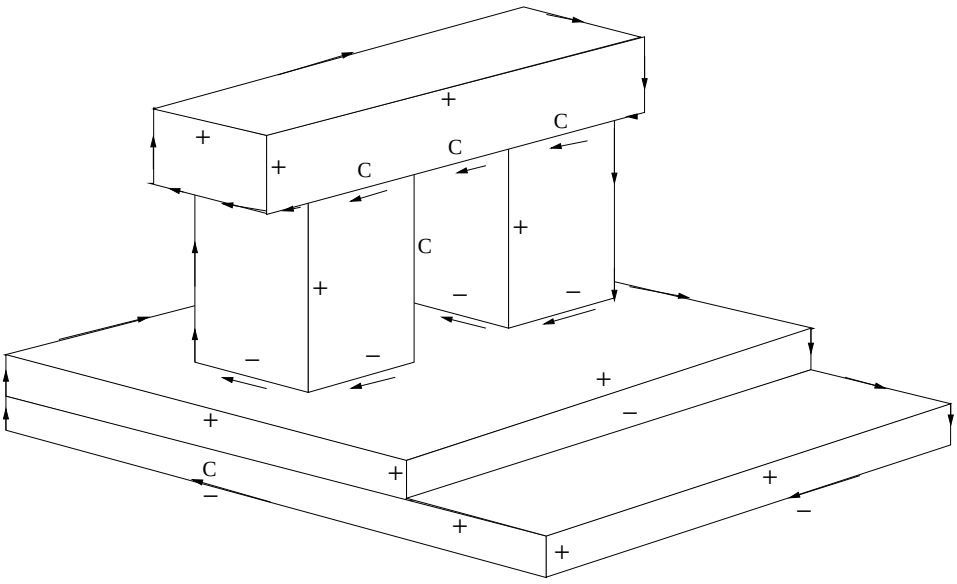
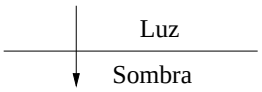


Figura 3.6: El análisis termina excepto por las ambigüedades.

ocultos. ¿Porqué las columnas no pudieran estas flotando algunos centímetros sobre la plataforma? Por otro lado, tampoco se sabe si los bordes más salientes del dintel debieran ser separaciones o bordes ocultos.

Finalmente, el programa de Waltz toma en cuenta el sombreado, operando en un conjunto mucho mayor de uniones (y sus etiquetas) que resulta de considerar bordes de sombra, que se simboliza mediante una flecha que atraviesa el segmento apropiado:



Los trabajos anteriores sobre análisis de escenarios poliédricos supusieron que las sombras eran meramente una irrelevancia confusa. Waltz demuestra su utilidad para eliminar ambigüedades del tipo que se han encontrado en el análisis del ejemplo. La figura 3.7 muestra tres sombras, que dan lugar a nuevas etiquetas de uniones (figura 3.8). Por supuesto, una vez que las sombras se han tomado en cuenta, muchas más etiquetas de uniones son posibles además de estas tres. Sin embargo, aún con una lista expandida,

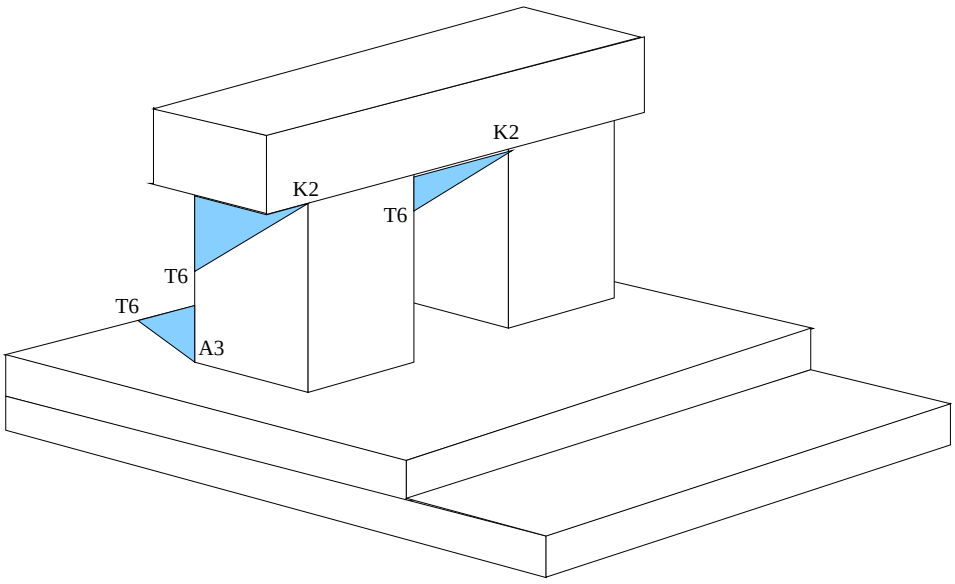


Figura 3.7: Las sombras eliminan ambigüedades.

la presencia de sombras en el escenario poliédrico del ejemplo es suficiente para resolver todas excepto dos de las ambigüedades sobre las etiquetas de los bordes. Nótese que las nuevas etiquetas de uniones se introducen sólo donde una línea con sombra (en oposición de un borde oculto) se encuentra con una unión o crea uno nuevo.

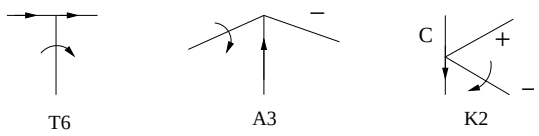


Figura 3.8: Etiquetas de uniones con sombras.

El algoritmo del programa de Waltz se entiende mejor en el contexto de lo que exactamente entra y sale de él. La entrada del programa de Waltz no es la imagen digitalizada de un escenario poliédrico, sino una lista de sus uniones, segmentos y regiones. Aun cuando tal lista no es muy difícil de generar dados:

- una clara y bien iluminada escena poliédrica.

- una cámara digital, interface y computadora.
- un programa efectivo para encontrar líneas.

Waltz supone, para cada uno de los escenarios poliédricos que estudia, la existencia de tales recursos, pero en realidad los simula a mano, nombrando cada unión, creando una lista de ellas, y creando una lista de pares de uniones representando los segmentos entre ellas. Es interesante que tal descripción es enteramente topológica, ya que no se provee ninguna información sobre distancias. Tampoco es muy difícil, bajo las dos primeras condiciones anteriores, escribir un programa que determine las regiones uniformes (en la imagen digitalizada del escenario poliédrico) y su brillantez. Por esta razón, Waltz sintió que era adecuado proveer a su programa con información acerca de qué segmentos de línea limitaban cada región, incluyendo las exteriores que rodean a los objetos representados.

El programa da como salida una etiqueta por cada segmento de entrada, dando a cada segmento una sola o múltiples etiquetas. Cuando tal etiquetado es disponible, es presumiblemente posible para otro programa identificar objetos o partes de objetos en el escenario mediante conjunción de regiones de acuerdo con las etiquetas asignadas a sus bordes limitantes.

El programa de Waltz usa una gran base de datos de uniones etiquetadas y segmentos rectilíneos, así como varias reglas de selección y heurísticas opcionales para ayudar en la eliminación sistemática de combinaciones imposibles de etiquetas asignadas a uniones y segmentos de línea. Como tal, es un programa muy grande, e imposible de describir aquí en todo su detalle. Sin embargo, la parte más interesante e importante del programa, aquélla que elimina etiquetas a las uniones, puede resumirse como sigue.

Para cada unión, el programa realiza tres pasos:

1. Crea una lista de todas las posibles etiquetas para una unión de ese tipo.
2. Examina las uniones adyacentes, usando la lista de etiquetas actual para restringir las posibilidades de esta unión. Etiquetas imposibles se eliminan de la lista.
3. Usando la lista reducida de etiquetas así obtenida, elimina cualquier etiqueta imposible nuevamente a partir de las uniones adyacentes y continúa propagando tales restricciones hacia afuera conforme ocurran.

Waltz mostró que la solución que su programa encuentra es siempre la misma, sin importar con qué unión se comience. El programa está garantizado de terminar después de ejecutar los pasos anteriores para cada unión en la imagen.

Después de mostrar la efectividad de su programa en escenarios poliédricos relativamente sencillos, Waltz pudo extenderlo para manejar ciertas degeneraciones y alineaciones accidentales, aumentando el poder de su programa. Pero ¿qué tan poderoso es finalmente el programa? ¿y qué nos dice acerca de la posibilidad de "computadoras que vean"? Primero, escenarios ordinarios con los que los seres humanos interactúan son muy lejanos de ser poliédricos (con ciertas exceptiones de algunos panoramas urbanos). ¿El programa de Waltz puede ser extendido para tales escenarios? Probablemente no. Sin embargo, algunos escenarios en la industria manufacturera tienen o se les puede hacer tener la suficiente simplicidad geométrica para permitir al programa de Waltz operar efectivamente como parte de un software visualizador de un robot.

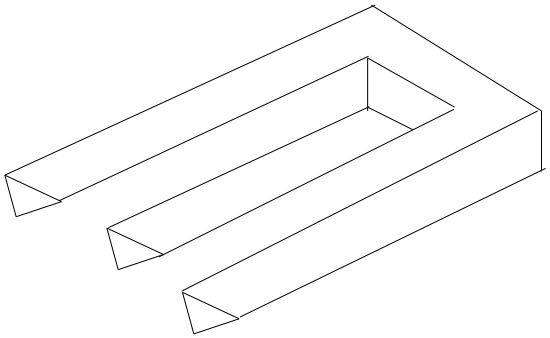


Figura 3.9: Un problema para el programa de Waltz.

Capítulo 4

Recursión
La Curva de Sierpinski

Los temas de recursión y los fractales se consideran siempre complementarios. La recursión es la invocación de un procesamiento dentro de un procesamiento idéntico que se encuentra actualmente en progreso. Los fractales son formas que ocurren dentro de otras formas similares. Para ilustrar la recursión en una forma concreta, se utiliza aquí una curva de Sierpinski.

La curva de Sierpinski es lo que los matemáticos llaman una curva en el límite de una secuencia infinita de curvas numeradas mediante un índice $n = 1, 2, 3, \ldots$. Una propiedad especial de la curva de Sierpinski es que llena el espacio bidimensional. Si cada curva en la secuencia en la figura 4.1 se dibuja a la mitad de la escala de su predecesora, entonces cada punto en la región de la curva se hallará arbitrárіamente cercano a algunos miembros de la secuencia. Es decir, la secuencia de curvas se acerca cada vez más a cada punto dentro de la región. La curva en el límite finalmente las cubre a todas.

Matemáticamente hablando, una función recursiva es aquélla que se usa a sí misma dentro de su propia definición. Se dice que un lenguaje de programación es recursivo si permite la creación de procedimientos que se llaman a sí mismos. Esta última forma de recursión, así como su implementación, son el tema que se trata aquí.

Para dibujar una curva de Sierpinski, se requiren dos procedimientos: ZIG y ZAG. Imagínese un gusano. Cualquier dirección que tome debe hacerlo conforme ZIG y ZAG:

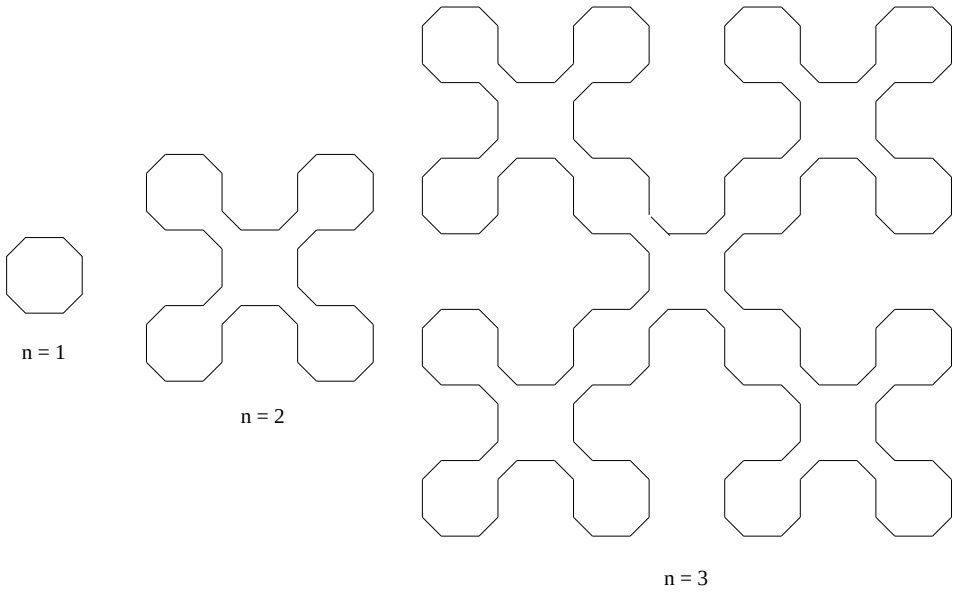


Figura 4.1: Las primeras tres curvas.

- ZIG
  - Gira a la izquierda, avanza una distancia d.
  - Gira a la izquierda (de nuevo), avanza una distancia d.
- ZAG
  - Gira a la derecha, avanza una distancia d.
  - Gira a la derecha (de nuevo), avanza una distancia d.
  - Gira a la izquierda, avanza una distancia d.

La distancia d es variable. Ciertamente, d es el parámetro de funcionamiento del programa.

El argumento n de ZIG y ZAG, como se define enseguida, controla el proceso de recursión mediante especificar cómo termina. Ambos ZIG y ZAG se definen en términos de sí mismos y del otro:

$ZIG(n)$

1. **if** $n = 1$ **then**

a) gira a la izquierda

b) avanza una unidad

c) gira a la izquierda

d) avanza una unidad

2. **else**

a) ZIG($n/2$)

b) ZAG($n/2$)

c) ZIG($n/2$)

d) ZAG($n/2$)

$ZAG(n)$

1. **if** $n = 1$ **then**

a) gira a la derecha

b) avanza una unidad

c) gira a la derecha

d) avanza una unidad

e) gira a la izquierda

f) avanza una unidad

2. **else**

a) ZAG($n/2$)

b) ZAG($n/2$)

c) ZIG($n/2$)

d) ZAG($n/2$)

Habiendo definido ambos procedimientos de manera algorítmica, el "programa principal" se escribe como:

ZIG(8)
ZIG(8)

Este programa consiste de dos llamadas consecutivas al procedimiento ZIG con la distancia de 8 como argumento. De hecho, supóngase que se cuenta con un dispositivo que es capaz de dibujar líneas rectas basadas en las direcciones de giro y avance que se utilizan en ambos procedimientos.

Cuando ZIG inicialmente se llama con un argumento de 8, el procedimiento prueba primero si $n = 1$. Como n no es 1, ZIG se llama a sí mismo con $n = 8/2 = 4$. Esta llamada resulta en otra prueba para ver si n es 1. Ya que esto falla una vez más, la nueva versión de ZIG hace otra llamada a sí misma, esta vez con argumento $n = 2$. Como el argumento no es todavía 1, se hace una llamada más a ZIG; cuando ZIG se ejecuta una vez más, se ejecuta la instrucción de dibujo para producir el primer fragmento de la cuarta curva de Sierpinski (figura 4.2). Cuando se invoca ZIG(2), no hace una sola llamada, sino hace cuatro:

ZIG(1)
ZIG(1)
ZIG(1)
ZIG(1)

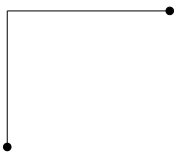


Figura 4.2: Ejecución de ZIG(1).

Tras el primer ZIG(1), el resto del procedimiento se encarga de añadir otros tres segmentos a la curva inicial (figura 4.3).

Figura 4.3: Ejecución de ZIG(2).

El primer ZAG toca el final del segundo ZIG del diagrama. De aquí en adelante, se muestran las operaciones ZIG y ZAG sin esquenas redondeadas, para evitar que la curva se intersecte con sí misma. El progreso del programa

que dibuja curvas puede seguirse mediante un árbol: cada ejecución de ZIG o ZAG resulta en el dibujo de una porción de la curva o resulta en más llamadas a ZIG y ZAG. En el diagrama de árbol de la figura 4.3, ZIG(8) es una llamada a ZIG(4), que llama a ZIG(2), que a la vez llama a ZIG(1). Esta última llamada completa una parte de la curva, lo que termina con la ejecución del primer ZIG(1).

La computadora es capaz de considerar dónde se encuentra durante una recursión. En este caso, se puede considerar que se tiene una representación de la llamada a ZIG(2) como:

ZIG(2):

ZIG(1) ←
ZAG(1)
ZIG(1)
ZAG(1)

La flecha indica que se ha completado la primera llamda ZIG(1). Continúa ejecutando ZAG(1), ZIG(1) y ZAG(1), completando la curva de la figura 4.3. En este momento, la ejecución de ZIG(2) se termina, y el programa retorna a un nivel superior, a ejecutar ZIG(4). Pero sólo la primera llamada dentro de ZIG(4) se completa, ya que la siguiente llamada es a ZIG(2), lo que resulta en cuatro nuevas llamadas a ZIG(1), dibujando otra porción de la curva (figura 4.4).

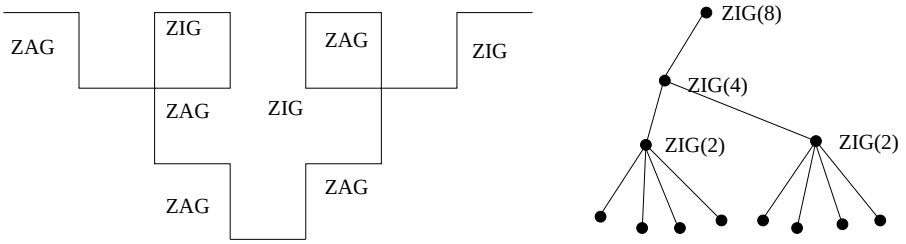


Figura 4.4: Ejecución de la mitad de ZIG(4).

Las reglas para ejecutar ZIG(8) (que, finalmente, es sólo la mitad de la curva) pueden considerarse como las reglas para revisar el árbol. Cada vez que un procedimiento se llama, se desciende un nivel en el árbol a un nuevo nodo. Cada vez que un procedimiento termina, se asciende un nivel al nodo que lo llamó. La curva final puede verse como varias ZIG(1) y ZAG(1), que

se encuentran en las hojas del árbol. Cuando la segunda llamada ZIG(8) se realiza, una tercera curva de Sierpinski aparece como la que se muestra en la figura 4.1.

Cualquiera que escriba un programa para producir una curva de Sierpinski en un lenguaje de programación que tenga recursión puede obtener curvas de my alto nivel. El nivel más alto para una computadora dada depende en la resolución del dispositivo de despliegue. Por ejemplo, con una pantalla de al menos 256 pixeles (en su dimensión menor), una curva de Sierpinski de octavo nivel ($n = 8$) puede dibujarse. Tan solo es necesario ejecutar:

ZIG(256)
ZIG(256)

La recursión se controla en la computadora mediante una pila. Por simplicidad, se supone aquí que el programa, incluyendo las definiciones de ZIG y ZAG, se almacena en direcciones consecutivas de memoria, comenzando con la dirección 1001:

```
        1001  ZIG(8)
        1002  ZAG(8)
ZIG:    1003  if n = 1
        1004  then gira izquierda
        1005  avanza 1
        1006  gira izquierda
        1007  avanza 1
        1008  else ZIG(n/2)
        1009  ZAG(n/2)
        1010  ZIG(n/2)
        1011  ZAG(n/2)
ZAG:    1012  if n = 1
        1013  then gira derecha
        1014  avanza 1
        1015  gira derecha
        1016  avanza 1
        1017  gira izquierda
        1018  avanza 1
        1019  else ZAG(n/2)
        1020  ZAG(n/2)
        1021  ZIG(n/2)
        1022  ZAG(n/2)
```

Por supuesto que el programa se almacenaría en la forma descrita, pero en lenguaje de máquina. La idea esencial, sin embargo, involucra almacenar las instrucciones presentes y pasadas en una pila. Primero, se ejecuta ZIG(8). La dirección de esta instrucción así como su parámetro asociado $n = 8$ se colocan en la pila de recursión. La ejecución del programa pasa automáticamente a la instrucción en la dirección 1003. Entonces (dado que $n \neq 1$) salta a la dirección 1008. Ya que la instrucción en 1008 es otra llamada a ZIG, la dirección de esta instrucción y su parámetro actual $n/2 = 4$ se colocan en la pila, que tiene la apariencia siguiente:

1008	4
1001	8

La llamada a ZIG con parámetro 4 resulta en otra llamada a ZIG con parámetro 2, y finalmente, a una llamada a ZIG con parámetro 1:

1008	1
1008	2
1008	4
1001	8

En este punto, sin embargo, $n = 1$, y las instrucciones en las direcciones 1004 a 1007 se ejecutan, completando ZIG(1). Cuando la recursión regresa un nivel, se saca de la pila:

1008	2
1008	4
1001	8

La ejecución pasa a la siguiente instrucción, en la dirección 1009, de modo que la pila contiene:

1009	1
1008	2
1008	4
1001	8

De esta forma, los contenidos de la pila se meten y sacan alternativamente conforme se ejecutan las instrucciones en las direcciones 1008, 1009, 1010 y 1011 con $n = 1$. Cuando el último de ellos, ZAG(1), termina, la ejecución regresa a ZIG(2) llamado desde la dirección 1008, pasando a ZAG(2) en la siguiente dirección 1009.

No es difícil notar que el número de direcciones que ocupan la pila de recursión en cualquier momento es simplemente la profundidad en el árbol de recursión por el cual el procedimiento desciende. Así es la recursión, al menos implementada.

Parece importante mencionar que la definición de la curva de Sierpinski puede expresarse en términos de una gramática libre de contexto. Si, por ejemplo, se usan letras como a y b para representar respectivamente ZIG y ZAG, entonces el proceso de dibujo de la curva de Sierpinski puede representarse como:

$$
\begin{aligned}
&aa\\
&a \rightarrow abab\\
&b \rightarrow bbab
\end{aligned}
$$

Aquí, aa representa la palabra inicial. Alternativamente reemplazando a por $abab$ y b por $bbab$ resulta en palabras más largas:

$$
\begin{aligned}
&aa\\
&abababab\\
&ababbbababbbbababbbbababbbbab\\
&\vdots
\end{aligned}
$$

En cualquier momento puede detenerse, reemplazando cada letra por su correspondiente segmento de curva, apropiadamente orientado. El resultado para la n-ésima palabra en la secuencia es la n-ésima curva de Sierpinski.

Capítulo 5

Tomografía Axial Computarizada (CAT)

Rayos X Seccionales

En las últimas décadas, la medicina se ha revolucionado por la aparición de la tomografía axial computarizada (*computerized axial tomography*, ó simplemente CAT), una técnica de reconstrucción de imágenes seccionales bidimensionales o tridimensionales de pacientes a partir de una multitud de imágenes de rayos X bidimensionales o uni-dimensionales, respectivamente. Las ventajas de este dispositivo son obvias. En lugar de examinar sombras vagas de una placa tradicional de rayos X, los médicos pueden examinar las características patológicas en la anatomía con casi el mismo grado de claridad como si el paciente estuviera partido en varias partes.

El problema de reconstrucción es fácil de establecer y entender. Aquí, esto se hace inicialmente en términos de una sección esquemática de un cuerpo, imaginando un arreglo rígido de rayos X paralelos y sus respectivos detectores, los cuales pueden rotarse alrededor de un eje que se encuentra cerca de la mitad del cuerpo y perpendicular a la página (figura 5.1).

De hecho, conforme el aparato gira, una serie de tomas (*snapshots*) de los rayos X se leen por el arreglo de detectores D, siendo cada imagen esencialmente una rebanada unidimensional representanto la atenuación de los rayos X paralelos a través de varios tejidos corporales. Como en las placas de rayos X ordinarios, el hueso, con sus elementos más pesados como calcio, absorbe más rayos X que, por ejemplo, tejidos pulmonares con su composición más ligera. Sin embargo, conforme los arreglos generadores y detectores

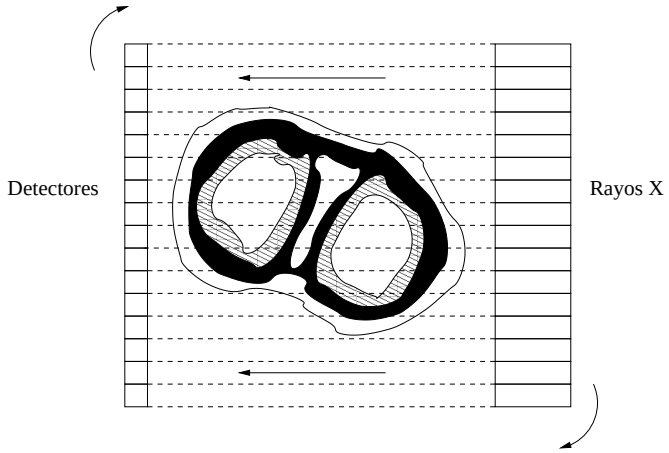


Figura 5.1: Una toma uni-dimensional de rayos X.

giran alrededor del paciente, las tomas contínuamente varían al mismo tiempo conforme la posición relativa de órganos internos y tejidos cambia con respecto a los rayos: cuando los rayos están horizontales, la espina dorsal aparece en el fondo de la imagen; pero cuando los rayos está verticales, la espina dorsal aparece a la mitad de la imagen.

Los detectores realmente no toman una fotografía de los rayos X que recibe. En lugar de esto, generan una señal eléctrica proporcional a la intensidad del rayo X recibido. Esta señal, junto con información sobre la posición del arreglo, se envía a una computadora digital para analizar las tomas sepradas y reconstruir la imagen de la distribución de materiales del cuerpo que causan las variaciones. En general, estos materiales varían de órgano a órgano, y de tejido a tejido. En la reconstrucción, se pueden ver e identificar los pulmones, el corazón, el hígado, las costillas, y demás partes del cuerpo.

Considérese un problema de reconstrucción mucho más sencillo, que involucra un sólido cúbico de densidad uniforme encerrado en una caja rectangular (figura 5.2). Se hacen tres tomas unidimensionales y separadas de rayos X, una por el lado, una por la parte superior, y otra en un águlo de 45 grados. Las vistas del cubo dentro de la caja tomadas desde A, B y C respectivamente, se ven bastante diferentes entre sí, y pueden representarse por tres funciones de una variable (figura 5.3).

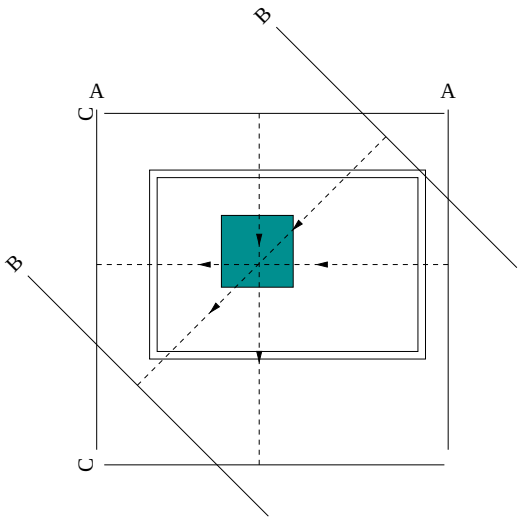


Figura 5.2: Un problema más sencillo.

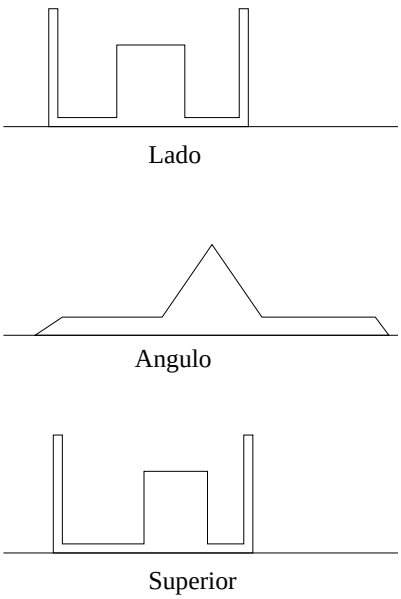


Figura 5.3: Tres vistas de la forma en la caja.

Las vistas unidimensionales de la parte superior y el lado parecen sumamente similares con dos picos altos que representan los lados de la caja y una forma cuadrada entre ellos que representa al cubo. La vista del ángulo, sin embargo, se ve muy diferente. Los dos picos han desaparecido, y el cubo se ha vuelto un triángulo. Esto último es fácilmente explicable: algunos rayos atraviesan al cubo de esquina a esquina, mientras que otros rayos adyacentes van atravesando cada vez partes más delgadas del cubo, lo que da la apariencia de caer linealmente con la distancia. Los rayos que no pasan ni siquiera por el cubo, sin embargo, todavía tienen que atravesar ambos lados de la caja, y lo hacen angularmente. Esto explica porqué la caja se ve "más ancha" en la vista angulada que en las otras vistas.

La técnica más sencilla para reconstruir imágenes de rayos X se conoce como "proyección inversa" (*back projection*). Todas las otras técnicas requieren de versiones sofisticadas del mismo objeto. La proyección inversa es fácil de ilustrar gráficamente, especialmente en el caso del ejemplo del cubo en la caja. Debido a que el número de vistas tomadas de este objeto es severamente limitado, la reconstrucción no resulta especialmente impresionante.

En términos visuales, se sobreponen las tres imágenes entre sí (figura 5.4), siendo cada imagen una serie de bandas paralelas cuya densidad o sobreado representa la altura de las tres funciones de atenuación A, B y C.

Aun cuando no es difícil notar la caja y el cubo dento de la imagen, la figura está llena de trazos: cada densidad proyectada ciertamente contribuye a marcar la densidad del objeto que originalmente lo produjo, pero en ningún otro lado. Para deshacerse de este ruido, se debe dividir todas las densidades entre el número de imágenes utilizadas en la reconstrucción, en este caso 3. Pero para deshacerse del resto, se debe utilizar una técnica de filtrado.

Si se pudiera hacer uso de un "cómputo continuo", se podría de alguna manera procesar cada punto matemático (x, y) dentro de la región R de interés, de modo que una proyección podría especificarse como sigue:

$$P(\theta, t) = \int_R f(x, y)\, \delta(x\, sin\, \theta - y\, cos\, \theta - t) dx\, dy$$

Aquí, $f(x, y)$ representa la imagen de tejido a ser reconstruida, mientras que $P(\theta, t)$ representa la atenuación medida por cada rayo en la posición t y el ángulo θ. La función δ es una función especial de filtrado aplicada al rayo con la ecuación lineal:

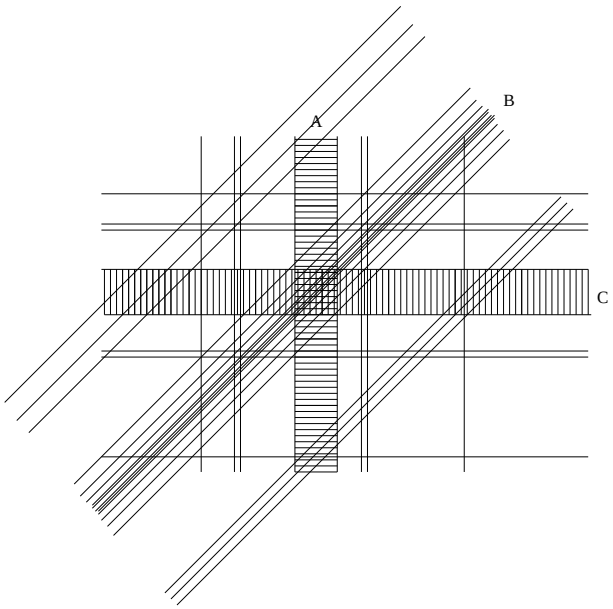


Figura 5.4: Proyectando las tres vistas.

$$x\,sin\,\theta - y\,cos\,\theta = t$$

Este rayo hace un ángulo θ con la horizontal, y pasa dentro de las t unidades desde el origen del sistema de ejes empleado. Ciertamente, t es meramente la posición en el arreglo detectores en el cual este rayo se recibe (figura 5.5).

Si θ se encuentra fija, entonces la función $P(\theta, t)$ provee el grado de atenuación encontrado por cada rayo (valor t) en términos de la densidad $f(x, y)$ de todos los puntos en ella; estos son simplemente los puntos que satisfacen la ecuación $x\,sin\,\theta - y\,cos\,\theta - t = 0$. En este caso continuo, la función de filtrado δ tiene una forma especialmente sencilla: tiene valor de 1 en cualquier momento que su argumento tenga valor 0, de modo que ningún punto fuera del rayo puede contribuir a su atenuación.

Si se pueden tener de alguna manera todos los valores posibles de $P(\theta, t)$, ¿cómo construir $f(x, y)$ a partir de ellos? En términos continuos, simplemente se tiene que invertir la siguiente integral:

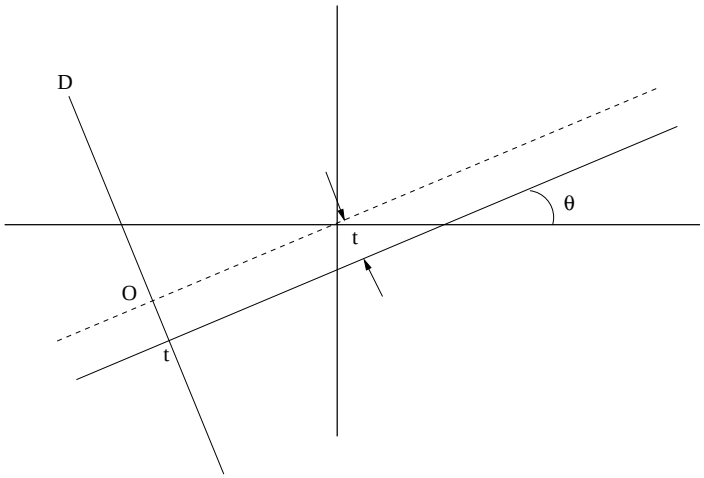


Figura 5.5: El paso de un rayo X.

$$f(x,y) = \frac{1}{2\pi} \int_0^{\pi} \int_{-r}^{r} P(\theta,t)\, \delta(x\, sin\, \theta - y\, cos\, \theta - t)\, dt\, d\theta$$

La integral indica añadir todas las contribuciones al punto (x, y) mediante proyecciones inversas en cada ángulo θ posible desde 0 hasta π. Para un θ dado, sólo interesa el valor de t que satisfaga:

$$x\, sin\, \theta - y\, cos\, \theta - t = 0 \qquad \text{donde} - r \leq t \leq r$$

Este es meramente el rayo proyectado inversamente que pasa por el punto (x, y).

Simplemente, entonces, si hay un número infinito de proyecciones a disposición, la función de densidad del rayo X seccional f (y, por lo tanto, la anatomía del paciente) puede reconstruirse perfectamente.

Sin embargo, las dificultades surgen en el procesamiento de la reconstrucción de la imagen debido a que se requiere trabajar con un número finito de proyecciones. Al intentar re-ensamblar f, se debe utilizar una versión discreta análoga a la ecuación anterior:

$$f(x_i, y_j) = \frac{1}{n} \sum_{k=1}^{n} \sum_{l=1}^{m} P(\theta_k, t_l)\, \delta(x_i\, sin\, \theta_k - y_j\, cos\, \theta_k - t_l)$$

En esta ecuación, (x_i, y_j) es uno de un conjunto finito de puntos en el cuerpo del paciente para quien se intenta recontruir f, mientras que θ_k y t_l son los ángulos y rayos, cada uno obtenido de conjuntos finitos, que contribuyen a la reconstrucción. La fuente precisa de dificultades recae en la no-coincidencia de ningún conjunto candidato de puntos (x_i, y_j) y las intersecciones de ningún sistema finito dado de rayos.

En la práctica, la imagen reconstruida (como toda imagen generada por computadora) se genera con pixels, una malla de pequeños cuadros dentro de los cuales la densidad se supone uniforme. Para cada ángulo de proyección y por cada rayo, los pixeles que se intersectan tienen su contribución alterada por la cantidad de rayos que lo intersectan. Lo mismo ocurre para el rayo proyectado inversamente (figura 5.6).

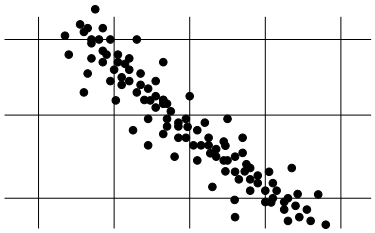


Figura 5.6: Un rayo X intersecta varios pixeles.

En este punto, es necesario utilizar para δ una función de filtrado d que indica a la computadora precisamente cuánto peso dar a la proyección inversa $P(\theta_k, t_l)$ en el pixel (x_i, y_j) dado. Este filtro no debe solo tomar en cuenta la discretización de los rayos, el espaciado entre ellos y la geometría de los pixeles, sino también compensar por el hecho de que los pixeles cerca de los ejes de las proyecciones (su centro de rotación) están más saturados que los pixeles lejos de este punto.

Ciertamente, el estudio de las técnicas tomográficas de reconstrucción es por mucho la investigación de los varios filtros, y en este punto, el tema se vuelve muy técnico para una simple nota. Sin embargo, suponiendo que la función del filtro se denote como $d(x_i, y_j, \theta_k, t_l)$, entonces se puede especificar un algoritmo de reconstrucción que la emplea:

$CATSCAN$

1. **for each** i **and** j

a) $f(x_i, y_j) \leftarrow 0$

2. **for** $k \leftarrow 1$ **to** n

   a) **for** $l \leftarrow 1$ **to** m

      1) **for each** i **and** j

         - $f(x_i, y_j) \leftarrow f(x_i, y_j) + P(\theta_k, t_l)\, d(x_i, y_j, \theta_k, t_l)$

3. **for each** i **and** j

   - $f(x_i, y_j) \leftarrow \frac{1}{nm} f(x_i, y_j)$

Algoritmos de reconstrucción mucho más sofisticados utilizan transformadas de Fourier. Tales algoritmos operan mediante primero realizar la discretización de la ecuación:

$$P(\theta, z) = \int_{-\infty}^{\infty} e^{-jzt} P(\theta, t)\, dt$$

y entonces utilizar la transformada inversa de Fourier:

$$f(x, y) = \frac{1}{4\pi^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} P(u, v) e^{j(ux+vy)}\, du\, dv$$

que en su forma polar es:

$$f(x, y) = \frac{1}{4\pi^2} \int_{0}^{\pi} d\theta \int_{-\infty}^{\infty} P(\theta, z)\, e^{jz(x \sin\theta - y \cos\theta)}\ \mid w \mid\ dw$$

donde j es el número imaginario $\sqrt{-1}$ y $\mid w \mid$ es un determinante especial llamado "jacobiano".

Otras formas de diagnóstico basadas en radiación incluyen tomografía por rayos gama y resonancia magnética. El procesamiento de las imágenes de estas técnicas toma en cuenta el hecho de que la señal se origina dentro del cuerpo del paciente. Sin embargo, todas guardan un gran parecido a la técnica de reconstrucción aquí descrita.

Capítulo 6

La Transformada Rápida de Fourier

Reordenando Imágenes

Cuando se enfrenta un problema computacional que involucra un cierto objeto matemático X, es a veces posible transformar X a otro objeto y resolver un problema computacional (relacionado) en el nuevo objeto. Por ejemplo, cuando X es un número real, se puede tomar su logaritmo antes de encontrar su producto con otro número real Y; en lugar de multiplicar X y Y, simplemente se suman sus logaritmos, obteniendo luego el antilogaritmo del resultado:

$$X \times Y \to log\, X + log\, Y \to antilog(log\, X + log\, Y) \to X \times Y$$

Cuando X es una función valuada en los reales, es útil en ciertos problemas obtener la transformada de Fourier de X antes de proceder. La transformada de Fourier se usa frecuentemente en el procesamiento de imágenes por computadora. En esencia, la transformada de Fourier de una imagen la redistribuye sobre el plano de la misma. Se usa esta propiedad para ver errores o distorsiones que en la imagen original no son visibles.

La transformada de Fourier de una función continua valuada en los reales f es la integral:

$$F(u) = \int_{-\infty}^{\infty} f(x)\, e^{-iux}\, dx$$

Para procesar la transformada de Fourier, se debe utilizar su versión discreta:

$$F(k) = \sum_{j=0}^{n-1} f(j)\,\omega^{kj}$$

donde $f(j)$ es un vector $[f(0), f(1) \ldots f(n-1)]$ y $\omega = e^{2\pi i/n}$ para las aplicaciones que se discuten aquí. Una forma más compacta de escribir la transformada discreta de Fourier es:

$$F = Af^T$$

donde f^T es el vector transpuesto de f y A es la matriz de $n \times n$ de la que el k, j-ésimo elemento es ω^{kj}. La matriz A resulta ser no-singular, de modo que el elemento k, j-ésimo de su inversa A^{-1} es ω^{-kj}/n. Se puede, por lo tanto, definir la inversa de la transformada de Fourier de un vector g como:

$$F^{-1} = A^{-1}g^T$$

De hecho, si se toma la transformada de Fourier de f y se le aplica la transformada inversa de Fourier, obviamente se obtiene de nuevo f:

$$f^T = A^{-1}Af^T$$

Hasta ahora, no se ha hecho mención de que ω es un número complejo, y por lo tanto, la transformada de Fourier de f tiene dos partes, una real y una imaginaria. En muchas aplicaciones, solo se utliza la parte real.

Un programa que directamente obtiene la transformada de Fourier de una función (o vector) f muy probablemente comenzaría con multiplicar cada renglón de A por la columna f^T, utilizando el orden de n multiplicaciones cada vez. Esto lleva a un total de n^2 de tales operaciones para completar la transformada. Cuando n es grande (como es la limitante en muchas aplicaciones), n^2 operaciones resultan una gran carga. Es por ello que a principios de los años 1960s se inicia la búsqueda de métodos más rápidos.

En 1965, J.M. Cooley y J.W. Tukey descubren un método para obtener la transformada discreta de Fourier que requiere del orden de solo $n \log n$ operaciones. Su método utiliza una estrategia de divide-y-conquista, que es muy común en algoritmos veloces.

Una manera de entender cómo la transformada rápida de Fourier de Cooley y Tukey funciona es ver el problema de cómputo:

$$F(k) = \sum_{j=0}^{n-1} f(j)\,\omega^{kj} \qquad \text{en } k = 0, 1, \ldots, n-1$$

como un problema para evaluar el polinomio:

$$P(x) = \sum_{j=0}^{n-1} f(j)\,x^{j} \qquad \text{en } x = \omega^0, \omega^1, \ldots, \omega^{n-1}$$

Pero para obtener $P(\omega^k)$ se requiere solo dividir $P(x)$ entre $x - \omega^k$. La teoría de polinomios dice que el residuo de esta división es $P(\omega^k)$. Se sigue que el problema de obtener P para estos n valores es tan solo encontrar tales n residuos.

La estrategia divide-y-conquista se presenta precisamente en este punto: suponiendo por ahora que n es potencia de 2, por ejemplo 2^m, se forman las cantidades $x - \omega^k$ en dos productos conteniendo $n/2$ factores cada uno. Considerando, por ejemplo, $n = 8$, se divide $P(x)$ por cada uno de estos productos:

$$\frac{P(x)}{(x-\omega^0)(x-\omega^1)(x-\omega^2)(x-\omega^3)}$$
$$\frac{P(x)}{(x-\omega^4)(x-\omega^5)(x-\omega^6)(x-\omega^7)}$$

Las divisiones resultan en dos cocientes $Q_1(x)$ y $Q_2(x)$. Estos pueden ahora dividirse entre los productos conteniendo $n/4$ factores cada uno. Estos factores son diferentes de los que produjeron tales cocientes:

$$\frac{Q_1(x)}{(x-\omega^4)(x-\omega^5)} \qquad \frac{Q_1(x)}{(x-\omega^6)(x-\omega^7)}$$
$$\frac{Q_2(x)}{(x-\omega^0)(x-\omega^1)} \qquad \frac{Q_2(x)}{(x-\omega^2)(x-\omega^3)}$$

Por ejemplo, ya que $x - \omega^0$ es parte de la división que produjo $Q_1(x)$, no participa más en ninguna otra división que involucre este cociente. Una vez más, se dividen los cocientes resultantes $Q_{11}(x)$, $Q_{12}(x)$, $Q_{21}(x)$ y $Q_{22}(x)$ entre los productos, esta vez por aquéllos que contienen $n/8$ factores cada uno:

$$\begin{array}{cccc} \frac{Q_{11}(x)}{(x-\omega^6)} & \frac{Q_{11}(x)}{(x-\omega^7)} & \frac{Q_{12}(x)}{(x-\omega^4)} & \frac{Q_{12}(x)}{(x-\omega^5)} \\ \frac{Q_{21}(x)}{(x-\omega^2)} & \frac{Q_{21}(x)}{(x-\omega^3)} & \frac{Q_{22}(x)}{(x-\omega^0)} & \frac{Q_{22}(x)}{(x-\omega^1)} \end{array}$$

En este ejemplo en particular, donde $n = 2^3$, solo se requieren tres iteraciones de este procedimiento. No es difícil notar, sin embargo, que en el momento en que los cocientes se dividen por expresiones sencillas de la forma $x - \omega^k$, se han realizado m iteraciones y el residuo de cada una de estas n divisiones es en cada caso el mismo de la división:

$$\frac{P(x)}{x-\omega^k} \qquad k = 0, 1, \ldots, n-1$$

Si $P(x)$ hubiera sido dividido por estos n factores uno a uno, habría del orden de n operaciones realizadas por cada división, lo que resulta en el orden de las n^2 operaciones en total. Sin embargo, resulta que cada división en el primer nivel se realiza con $n/2$ operaciones, en el segundo con $n/4$, el siguiente con $n/8$, etc. Así, en cada iteración hay del orden de n operaciones que se realizan, obteniendo una complejidad en tiempo total de:

$$m \cdot n = n \log n$$

Antes de que la transformada rápida de Fourier pueda presentarse algorítmicamente, hay algunos detalles que deben considerarse. Primero, obsérvese que un polinomio $Q(x)$ que tiene grado $2^{k+1} - 1$ puede dividirse por un polinomio:

$$(x - \omega^i) \ldots (x - \omega^j)$$

de grado 2^k en tan solo 2^k pasos elementales. Debido a la especial naturaleza de las potencias de ω, ciertos productos de los factores $x - \omega^j$ pueden escribirse sencillamente.

Específicamente,

$$(x - \omega^{r(j)})(x - \omega^{r(j+1)}) \ldots (x - \omega^{r(j+2^{k-1})}) = (x^{2k} - \omega^{r(j/2^k)})$$

donde r es una función que mapea cada entero a otro entero que tiene los mismos dígitos binarios en su expansión pero en orden inverso. La división

$$\frac{Q(x)}{x^{2k} - \omega^{r(j/2^k)}}$$

puede obviamente realizarse dentro de un múltiplo constante de 2^k pasos.

Anteriormente, se supone que $n = 2^m$, pero ¿qué sucede si n no es potencia de 2? En muchas aplicaciones, especialmente aquéllas que involucran procesamiento de imágenes, n se toma para ser potencia de 2. En otros casos, sin embargo, la función f se aumenta con valores adicionales, por ejemplo ceros, hasta que se convierte en un vector de longitud apropiada.

Un algoritmo que obtiene la transformada rápida de Fourier se presenta a continuación:

FOURIER

1. **for** $j \leftarrow 0$ **to** 2^{k-1}
   - $Q(j) \rightarrow Q_j$
2. **for** $l \leftarrow 0$ **to** $k - 1$
   a) **for** $j \leftarrow 0$ **to** 2^{k-1}
      - $S(j) \leftarrow Q(j)$

   b) **for** $j \leftarrow 0$ **to** 2^{k-1}
      - computar $(d_0 d_1 \dots d_{k-1})$ como la representación binaria en j
      - $Q(j) \leftarrow S(d_0 \dots d_{l-1} d_{l+1} \dots d_{k+1}) + \omega^{d_l d_{l-1} \dots d_0 0 \dots 0} . S(d_0 \dots d_{l-1} d_{l+1} \dots d_{k+1})$
3. **for** $j \leftarrow 0$ **to** 2^{k-1}
   - $b(j) \leftarrow Q(r(j))$

En este algoritmo, Q es un arreglo que almacena los coeficientes del cociente de polinomios descrito anteriormente. En el enunciado 2 se establecen los niveles de iteración desde 0 a $k - 1$. Obviamente, hay $k = log\, n$ de estos. La línea 2.a inicializa un vector temporal S. En la línea 2.b se establece el ciclo interno de n iteraciones, y dentro de este ciclo se realizan las divisiones. De hecho, estas divisiones se realizan tan eficientemente (como resultado de la simplificación matemática) que el j-ésimo coeficiente a la l-ésima iteración es esencialmente una combinación lineal de dos elementos del arreglo S, con ambos índices obtenidos simplemente por una alteración de la representación binaria de j.

Para transformadas de Fourier de imágenes digitalizadas de $n \times n$ se utiliza la fórmula:

$$F(k,l) = \frac{1}{n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f(i,j)\, \omega^{ik+jl/n}$$

donde $f(i,j)$ es el valor de intensidad en la malla de la imagen en la posición i,j-ésima. Esto se obtiene a partir de la fórmula anterior para $F(k)$ y reeemplazando el índice k por índices dobles (k,l), así como los coeficientes y potencias de la fórmula resultante, que se ajustan apropiadamente. El algoritmo para realizar la transformada rápida de Fourier de esta fórmula bi-dimensional es muy similar al algoritmo descrito anteriormente.

Una aplicación específica y fácil de la transformada de Fourier involucra la multiplicación de dos polinomios:

$$a(x) = \sum_{i=0}^{n-1} a_i x^i \qquad \text{y} \qquad b(x) = \sum_{j=0}^{n-1} b_j x^j$$

Brevemente,

$$a(x) \times b(x) = F^{-1}(F(a) \times F(b))$$

La transformada de Fourier F del polinomio a (representado por su vector de coeficientes) no es más que el arreglo de todos los valores a a las n valores $\omega^0, \omega^1, \ldots, \omega^{n-1}$. Los productos de estos valores, $a(\omega^i) \times b(\omega^j)$, dan los valores de $a(x) \times b(x)$ para los mismos valores. Se sigue que la transformada inversa de Fourier F^{-1} traduce esta representación del producto de los polinomios de nuevo a la representación de un vector de coeficientes.

Otras aplicaciones de la transformada rápida de Fourier suceden en procesamiento digital de señales en áreas como ingeniería, medicina, geología, y virtualmente todo dato científico de 1, 2, ó n dimensiones, que puede representarse mediante arreglos de los valores observados.

Capítulo 7

Almacenamiento de Imágenes
Un Gato en el Arbol de Cuadrantes

La llegada de los "árboles de cuadrantes" (*quad trees*) a principios de los 1970s señala un nuevo paso en el progreso de los gráficos por computadora. Muchas operaciones mejoraron, ya sean para el almacenamiento, manipulación o análisis de imágenes por computadora. Nuevas operaciones fueron posibles, todas mediante simplemente subdividir en cuadrantes e identificar los cuadrantes como nodos en un árbol, un árbol de cuadrantes.

Considérese la figura 7.1. Un gato digital se presenta de perfil. El gato puede descomponerse (por así decirlo) en un árbol de cuadrantes. Con esta estructura, se puede almacenar la imagen del gato muy eficientemente, realizar varias transformaciones geométricas en ella, y determinar si hay otros objetos además del gato.

El árbol de cuadrantes para el gato se genera mediante dividir la matriz de la imagen en cuadrantes, y subdividir cada uno de éstos en subcuadrantes, y continuar así hasta alcanzar la última etapa de subdivisión, en la que se alcanza cada *pixel* (*picture element*), que se vuelven cuadrantes. Para tal división de trabajo, el tamaño de la matriz debe ser obviamente una potencia de 2. Pero en otro caso, esto es fácil de compensar.

Cada cuadrante en la subdivisión se representa por un nodo en el corres-

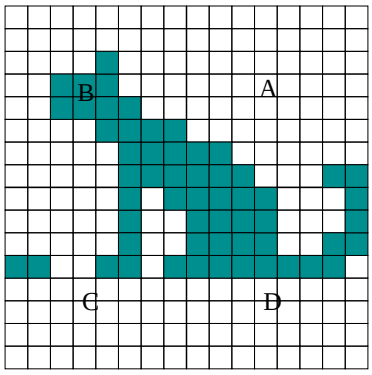


Figura 7.1: Un gato cuadriculado.

pondiente árbol de cuadrantes; si el cuadrante es todo blanco o todo negro, el nodo se etiqueta respectivamente y se le trata como un nodo terminal. No tiene hijos. Pero si el cuadrante no es del todo blanco o negro, el nodo correspondiente es dotado con cuatro hijos, uno por cada subcuadrante. Estos subcuadrantes se toman en el mismo orden como los cuadrantes originales, etiquetados A, B, C y D en la figura.

Cuando se descompone por el proceso, la imagen del gato se representa por un solo nodo de la matriz entera. Los cuatro cuadrantes principales se representan por cuatro nodos etiquetados apropiadamente. El primer nodo (A) representa al cuadrante noreste (NE), el segundo nodo (B) al cuadrante noroeste (NW), el tercer nodo (C) al cuadrante suroeste (SW) y el cuarto cuadrante (D) al cuadrante sureste (SE) (figura 7.2). Cada uno de los cuatro nodos da origen a otro árbol de cuadrantes para ese cuadrante; el nodo B, por ejemplo, presenta el sub-árbol que se muestra en la figura 7.3.

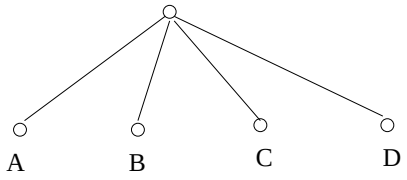


Figura 7.2: Nodos principales del árbol.

El árbol de cuadrantes *es* la imagen del gato cuando se interpreta apropaidamente. Un nodo terminal blanco que se encuentra a k niveles del fondo

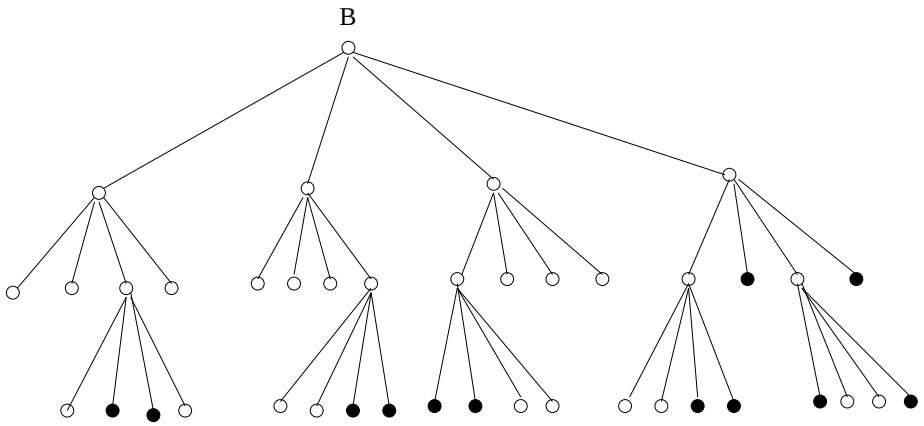


Figura 7.3: El árbol de cuadrantes de B.

del árbol representa un cuadrante de $2^k \times 2^k$ que es enteramente blanco. Los mismo sucede con un nodo negro. El número total de nodos en árbol de cuadrantes proporciona el espacio de almacenamiento que se requiere para la imagen. De tal modo, se dice que el gato ha sido comprimido en 41 nodos. Al mismo tiempo, la imagen del gato ocupa una matrix de 16×16. Por métodos convencionales de almacenamiento, la imagen requeriría el almacenamiento de 256 pixeles. Para la vasta mayoría de imágenes que surgen en varias aplicaciones, este tipo de economía de almacenamiento es común.

Ciertas manipulaciones sobre los árboles de cuadrantes corresponden a manipualciones estándar de imágenes gráficas. Por ejemplo, para rotar la imagen del gato 90 grados en sentido inverso de las manecillas del reloj, se deben rotar los nodos de cada nivel. De tal modo, la raíz del árbol se despliega en un orden diferente (figura 7.4). El cuadrante D ahora ocupa la esquina noreste de la imagen, A la noroeste, etc. El subárbol que depende del cuadrante B toma una apariencia diferente cuando es rotado (figura 7.5).

Algunas otras manipulaciones que pueden llevarse a cabo sobre imágenes representadas como árboles de cuadrantes son los cambios de escala (por un factor de 2) y las traslaciones. Para cambiar la escala de una imagen, se requiere solo remover todos los nodos terminales en el nivel más al fondo del árbol y reinterpretar el nodo raíz como si representara un solo cuadrante. Esto encoge la imagen por un factor de 2. La imagen debe ocupar solo el

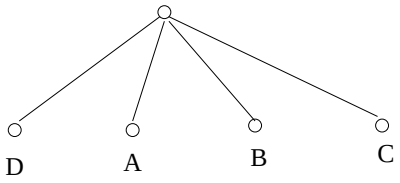


Figura 7.4: Rotando los nodos principales.

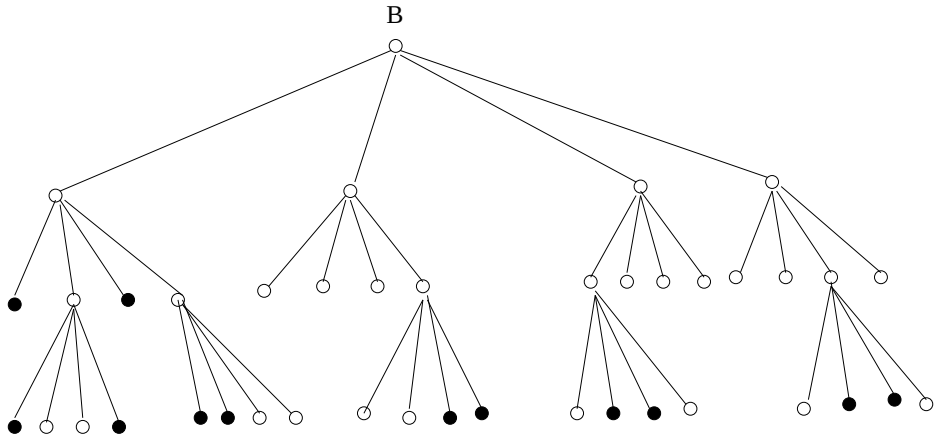


Figura 7.5: Rotando el árbol de cuadrantes de B.

cuadrante principal de modo que se remueve el nodo raíz y los otros tres nodos correspondientes a los otros cuadrantes no utilizados de la imagen.

La operación de disminución se relaciona cercanamente con una técnica útil de aproximación. Si se remueven uno o más niveles del fondo del árbol de cuadrantes, la imagen resultante pierde resolución en un factor de 2 o más. Se convierte en una aproximación. Para algunas aplicaciones como referencias visuales y reconocimiento de patrones, es suficiente trabajar con imágenes de baja resolución, especialmente aquéllas que son fáciles de reproducir.

La operación final que se revisa aquí identifica los componentes de una imagen. En la figura del gato, en realidad existen dos componentes: el gato y dos pixeles adyacentes frente a él. Estos dos pixeles intentan representar un ratón, pero dada la resolución de la imagen, es difícil que éstos sean un ratón convincente. De tal modo, la imagen cuenta con dos componentes. Encontrar los componentes de una imagen es importante para un gran número de aplicaciones, como escaneo automático, o imágenes médicas.

El algoritmo que localiza componentes de una imagen en relidad encuentra un pixel obscuro y busca por pixeles vecinos y obscuros al norte, sur, este y oeste. Busca contínuamente hacia afuera, conjuntando una lista de pixeles obscuros en el mismo componente mientras va añadiendo nuevos pixeles vecinos a la lista. Para identificar un componente de esta forma, un algoritmo de búsqueda de pixeles vecinos es crucial. Tal algoritmo opera sobre el árbol de cuadrantes, por supuesto, de modo que es razonable referirse a los nodos en el árbol y algunos cuadrantes en la imagen indistintamente. El vecino de un cuadrante dado en una dirección dada es el cuadrante más pequeño de al menos el mismo tamaño que comparte una frontera en el lado dado. Esta descripción se convierte en un algoritmo que revisa el árbol de cuadrantes. Supóngase, por ejemplo, que se busca al vecino al oeste de un nodo dado. En el algoritmo que sigue, se utiliza la convención para el etiquetado de los cuadrantes A, B, C y D que se introduce anteriormente. Así, un cuadrante B es un vecino al oeste de un cuadrante A, y un cuadrante C es un vecino al oeste de un cuadrante D. Conforme el algoritmo desciende por el árbol va colocando las etiquetas que encuentra en una pila. Tan pronto como encuentra un nodo a través de una liga etiquetada B o C, el algoritmo desciende, siempre tomando la liga que es horizontalmente opuesta a la misma ascendiendo por el mismo nivel del árbol.

1. **repeat**

   a) asciende una liga
   b) coloca su etiqueta en la pila
2. **until** la etiqueta de la liga es B ó C
3. **repeat**
   a) saca la etiqueta de la pila
   b) desciende la liga con etiqueta opuesta
4. **until** el nodo es terminal y la pila está vacía

Este algoritmo puede generalizarse para tomar cualquier dirección como entrada y encontrar al vecino de un cuadrante dado en esa dirección.

Para utilizar el árbol de cuadrantes para identificar componentes, el árbol debe revisarse en algún orden. Como cada nodo terminal obscuro se visita, se le da una nueva etiqueta (si no ha sido etiquetado), y sus cuatro vecinos se revisan utilizando el algoritmo para encontrar vecinos anterior. Un vecino obscuro (que debe ser un nodo terminal por definición del vecino) recibe la misma etiqueta que el nodo que se revisa:

1. etiqueta ← 1
2. **for each** nodo terminal revisado
   - **if** nodo obscuro y no etiquetado
   - **then** etiqueta el nodo
   - **for each** vecino
     a) **if** vecino obscuro y sin etiqueta
     b) **then** etiqueta al vecino
     c) **else** añade un par de etiquetas a la lista
   - etiqueta ← etiqueta + 1

Evidentemente, este no es el algoritmo completo de búsqueda de componentes, pero es la mayor parte. La lista de pares equivalentes que produce es en realidad un grafo que la siguiente parte del algoritmo de búsqueda de componentes debe revisar. El grafo puede explorarse de forma primero profundidad. Mientras sea posible encontrar nuevos pares para los cuales una etiqueta esté en una clase equivalente y la otra no, el algoritmo continúa. Finalmente, no hay más pares para procesar, y cada clase de equivalencia

recibe ahora una nueva etiqueta. En este punto, el algoritmo puede dar como salida el número de tales etiquetas (dos, en el ejemplo del gato y el ratón).

El tercer paso del algoritmo para buscar componentes simplemente revisa el árbol de cuadrantes una vez más, asignando una nueva etiqueta que es equivalente a cada una de las etiquetas anteriores que se encuentran en la revisión.

Capítulo 8

Programación Lineal
El Método Simplex

Imagínese un plano que atraviesa un sólido prismático poliédrico (figura 8.1). Si el plano se mueve hacia arriba a una posición de reposo final (que se muestra en líneas punteadas), pasa exactamente en un solo punto del sólido, tocándolo. ¿Quién pensaría, sin información del tema, que este sólido poliédrico, el plano, y el punto de contacto, representen un problema importante en economía, planificación, procesos químicos, o embarque de mercancías?

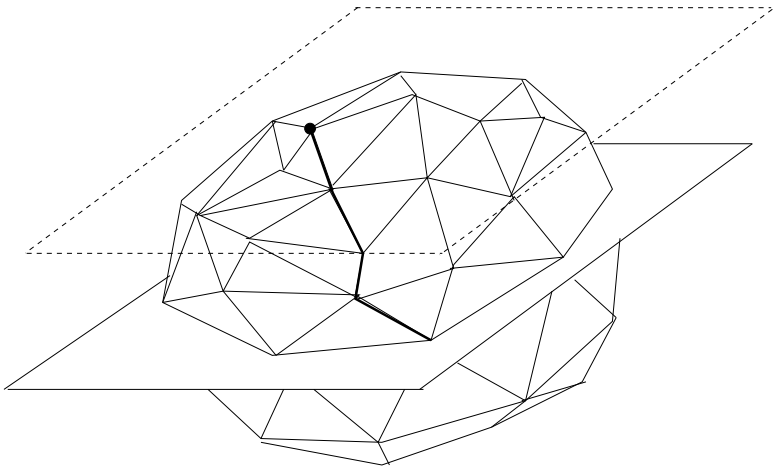


Figura 8.1: Una geometría de restricciones.

El poliedro representa un conjunto de desigualdades lineales que expresan restricciones inherentes a un problema dado. El plano, de hecho la posición del plano, representa el valor de una cierta función lineal que alguien trata de optimizar. El punto representa un valor óptimo de esa función, dadas las restricciones. Con respecto a la orientación del plano, el punto es un vértice externo del sólido.

En la figura 8.1, un trazo (en línea más gruesa) se muestra conduciendo al punto óptimo. Este trazo representa el curso que atraviesa el mejor algoritmo conocido que resuelve problemas de programación lineal: el algoritmo *simplex*. Un conjunto de de desigualdades lineales que se pueden escribir como:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i \qquad \text{for } i = 1, 2, \ldots, m$$

lo que representa m medios espacios, cada uno n-dimensional y cada uno acotado en un lado por un plano. La ecuación de este plano se obtiene mediante reemplazar la desigualdad anterior por una igualdad. El medio espacio representado por la i-ésima desigualdad consiste de todos los puntos que están en un lado del plano. La intersección de todos los m medios espacios, si no está vacía, forma un sólido convexo poliédrico, y cada una de sus facetas se obtiene por la contribución de uno de los planos.

Considérese el siguiente ejemplo bi-dimensional:

$$\begin{aligned} x_1 + 5x_2 &\leq 8 \\ 2x_1 + 3x_2 &\leq 6 \\ 3x_1 + x_2 &\leq 6 \\ -x_1 &\leq 0 \\ -x_2 &\leq 0 \end{aligned}$$

Cada desigualdad define un medio espacio bidimensional, y la intersección de las cinco se muestra como un área convexa sombreada en la figura 8.2. Las "facetas" son segmentos de línea.

El conjunto de desigualdades en la figura 8.2 tienen una cierta forma especial; las dos últimas contienen un coeficiente igual a cero cada una. Tal ocurrencia no es inusual en las desigualdades que surgen en aplicaciones.

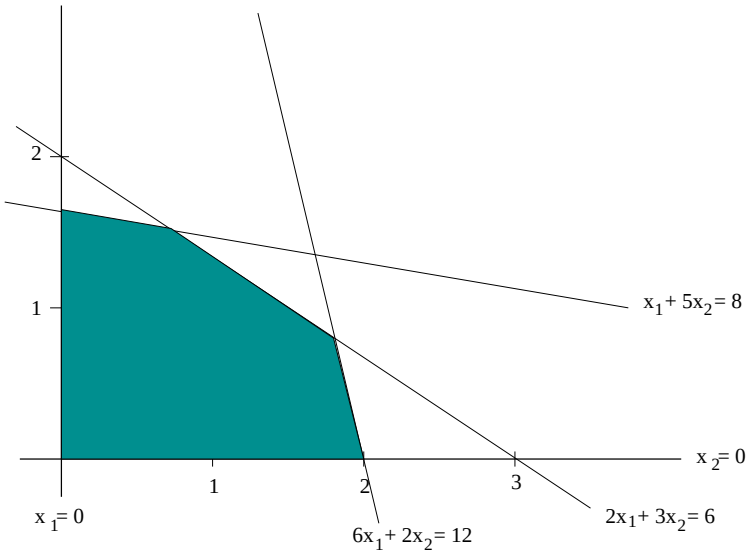


Figura 8.2: Restricciones que surgen de un proceso de manufactura.

Supóngase, por ejemplo, dos productos químicos, unifano y duozeno, que son manufacturados en una planta que utiliza tres substancias químicas en el proceso de producirlos: alfatono, ácido bético y gamina. Si la planta tiene 8 toneladas de alfatono, 6 toneladas de ácido bético y 6 toneladas de gamina, se dese saber cuánto de unifano y duozeno puede producirse de las cantidades en almacén de las substancias primarias.

Sucede que por cada gramo de alfatono utilizado para producir unifano, se requieren 5 gramos para producir duozeno. Si x_1 y x_2 denotan las cantidades actuales de unifano y duozeno producidos respectivamente, entonces se puede escribir:

$$x_1 + 5x_2 \leq 8$$

que es la primera desigualdad del conjunto anterior. Similarmente, las siguientes dos desigualdades surgen de las proporciones relativas de ácido bético y gamina que se utilizan en el proceso de producción. Las dos últimas desigualdades simplemente expresan el hecho que se tienen que producir cantidades no negativas de unifano y duozeno.

Cada punto dentro del área convexa sombreada de la figura 8.2 representa una combinación posible de las cantidades de x_1 y x_2 que se producen.

Las cantidades, naturalmente, se expresan en toneladas. Cualquier punto fuera de la región posible (el área convexa sombreada) representa un par de cantidades que simplemente no pueden producirse a partir de las substancias primarias en almacén.

La compañía química, sin embargo, se interesa no sólo en cuánto unifano y duozeno podría producir, sino también cuánto debe producirse. En este punto, las fuerzas del mercado hacen que el valor de los dos productos se use para asignar un valor a cada par de cantidades de producción (x_1, x_2) en la región de factibilidad. Si el unifano se vende por \$ 2 millones por tonelada mientras que el duozeno vale \$ 1 millón la tonelada, entonces los valores de las cantidades x_1 y x_2 deben dar otro valor z, que:

$$z = 2x_1 + x_2$$

Para cada valor posible de z, esta función de costo puede representarse mediante una línea recta en el diagrama. Todos los puntos posibles sobre esta línea representan cantidades posibles x_1 y x_2, cuyo valor de mercado combinado es z. La compañía química desea maximizar z, y por tanto, maximar su ganancia. Conforme la línea se mueve más y más lejos del origen, z se incrementa (figura 8.3) Finalmente, si se va demasiado lejos, llega a un punto en que no intersecta a la región factible; en tal caso, el valor de z correspondiente no puede ser producido por la compañía con las substancias que tiene en el almacén. Lo mejor que puede hacer se representa por el punto extremo al cual la recta de costo se detiene, por así decirlo.

¿Cuál es el mejor valor de z que se puede lograr? Esto depende de las coordenadas del punto extremo en cuestión. El punto se obtiene simplemente mediante resolver las ecuaciones lineales:

$$\begin{aligned} 2x_1 + 3x_2 &= 6 \\ 3x_1 + x_2 &= 6 \end{aligned}$$

La solución es $x_1 = 12/7$ y $x_2 = 6/7$. Por lo tanto, la compañía debe producir 1.71 toneladas de unifano, y 0.86 toneladas de duozeno. El valor de esta producción es de $z =$ \$ 4.29 millones, lo mejor que se puede obtener dadas las circunstancias.

La solución anterior se obtiene en mucho por medios visuales: se ve qué punto extremo representa el valor máximo de z y meramente se en-

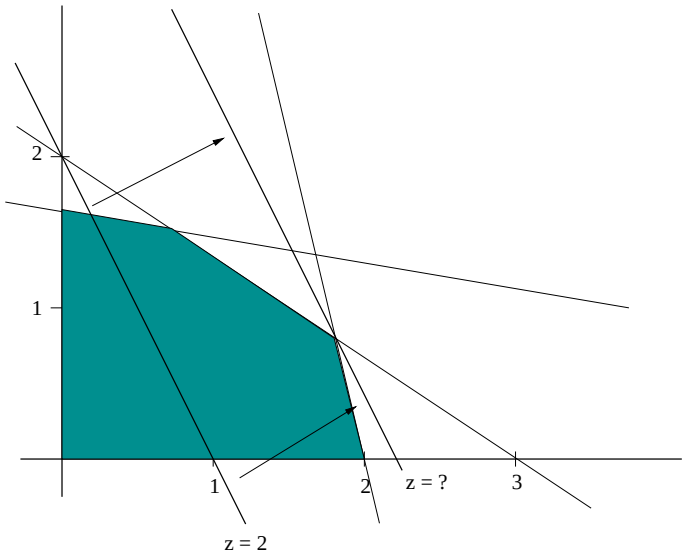


Figura 8.3: Los valores de mayor ganancia de la producción factible.

cuentran sus coordenadas mediante resolver las ecuaciones de las líneas que se encuentran en ese punto. Si el problema tuviera muchas más que dos variables, sería fácil perderse hasta al intentar concebir el plano de intersección que produce el punto, más allá de poder realmente visualizarlo. De tal modo, se requieren métodos automáticos de resolución.

Uno de esos métodos, en uso constante desde su descubrimiento en 1951 por George B. Danzig, se conoce como el método *simplex*. No trabaja propiamente con desigualdades, sino con ecuaciones. Para convertir el sistema anterior a ecuaciones, se introducen variables extra, como s_1, s_2 y s_3. Estas variables no tienen un significado dentro el problema químico; sirve solamente para convertir las desigualdades en igualdades mediante considerar una variación, por así decirlo. Si se añade la función objetivo a las ecuaciones así obtenidas, se tiene el siguiente sistema:

$$\begin{aligned} z - 2x_1 - x_2 &= 0 \\ x_1 + 5x_2 + s_1 &= 8 \\ 2x_1 + 3x_2 + s_2 &= 6 \\ 3x_1 + x_2 + s_3 &= 6 \end{aligned}$$

El sistema resultante tiene cuatro ecuaciones y seis incógnitas, y una solución inicial puede obtenerse fácilmente mediante diagonalizar el sistema. Esto se hace normalmente mediante eliminación gaussiana, pero en el caso anterior, el sistema está casi en esta forma:

$$\begin{aligned} -z + 2x_1 + x_2 &= 0 \\ s_1 + x_1 + 5x_2 &= 8 \\ s_2 + 2x_1 + 3x_2 &= 6 \\ s_3 + 3x_1 + x_2 &= 6 \end{aligned}$$

Se obtiene lo que llama "solución inicial factible" de las ecuaciones de esta forma mediante dar los valores $x_1 = x_2 = 0$.

El método simplex opera sobre sistemas de ecuaciones de esta forma, contínuamente aislando un conjunto de variables después de otro. Las soluciones así obtenidas representan puntos en la región convexa factible original, y cada punto arroja un valor z que mejora al valor obtenido en el paso anterior, hasta que se alcanza el valor óptimo de z. Por ahora, es notorio que $z = 0$ para la solución inicial, lejos del valor óptimo conocido 4.29.

A cada paso en el método simplex, se examinan los coeficientes de la ecuación objetivo, y el más grande se selecciona como base para el siguiente paso. Este, llamado "pivote", involucra intercambiar una variable aislada con una no-aislada. Inicialmente en el ejemplo, las variables aisladas o básicas son z, s_1, s_2 y s_3. La nueva variable básica es x_1, ya que tiene el mayor coeficiente en la función objetivo, es decir, 2. Para averiguar qué variable básica reemplaza x_1, se examinan las razones:

$$\frac{a_{i1}}{bi}$$

y se selecciona la ecuación con la mayor de tales valores. En el ejemplo, estas razones son $\frac{1}{8}$, $\frac{2}{6}$ y $\frac{3}{6}$, respectivamente. La última razón es la mayor. Aparece a partir de la última ecuación, de tal modo que es la variable s_3 la que se reemplaza por x_1.

El reemplazo o "pivoteo" se hace mediante el proceso de eliminación gaussiana: se usa la última ecuación para eliminar la variable x_1 de todas las otras ecuaciones, mediante añadir los múltiplos apropiados de la última

ecuación a las otras. El nuevo sistema es:

$$\begin{aligned} -z - \frac{2}{3}s_3 + \frac{1}{3}x_2 &= -4 \\ s_1 - \frac{1}{3}s_3 + \frac{14}{3}x_2 &= 6 \\ s_2 - \frac{2}{3}s_3 + \frac{7}{3}x_2 &= 2 \\ x_1 + \frac{1}{3}s_3 + \frac{1}{3}x_2 &= 2 \end{aligned}$$

De nuevo, se obtiene una solución básica al dar valor de 0 a las variables no-básicas. Así, $s_3 = x_2 = 0$, obteniéndose:

$$z = 4 \qquad s_1 = 6 \qquad s_2 = 2 \qquad \text{y} \qquad x_1 = 2$$

Observando con respecto a los valores iniciales, es notorio que los valores de x_1 y x_2 arrojan el punto $(2, 0)$ en el cual el valor de la función objetivo es 4. Esta es una gran mejora respecto al valor anterior (0), y sin embargo, todavía se tiene una distancia para encontrar el valor máximo absoluto de 4.29. Al menos, se requiere un pivoteo más.

Para seleccionar la variable pivote, se examinan los coeficientes en la función objetivo, notándose que el mayor coeficiente corresponde a la variable x_2. La más grande razón coeficiente/constante:

$$\frac{a_{i2}}{b_i}$$

es $\frac{14}{18}$, obtenida de la segunda ecuación. Así, la nueva variable básica es x_2 que reemplaza a s_1, la variable básica de la segunda ecuación. Cuando se realiza la eliminación gaussiana por segunda vez con este pivote, se obtiene la solución óptima de $x_1 = \frac{12}{7}$ y $x_2 = \frac{6}{7}$ con el valor máximo de $z = \frac{30}{7}$, aproximadamente 4.29.

El algoritmo simplex termina cuando no hay más coeficientes positivos en la función objetivo. Cuando se programa el algoritmo en una computadora, las listas de los coeficientes, constantes y variables básicas se mantienen en arreglos llamados *tableux*.

La teoría de programación lineal va más allá del método simplex y su soporte teórico. Hay programas lineales en los que las funciones objetivo se

debe minimizar en lugar de maximizarse, de modo que todas las desigualdades tienen la dirección opuesta.

En las últimas décadas, ha habido muchos análisis alrededor del método simplex. Y se ha encontrado, por ejemplo, que su desempeño en el peor de los casos puede ser bastante inadecuado, aunque muy poco probable: en algunos casos el número de pivoteos requiere un número exponencial de variables. Para cuando algunos investigadores comenzaron a sospechar que algunos problemas de programación lineal podían ser computacionalmente inextricables, un nuevo algoritmo fue descubierto por L.G. Kachian, un matemático armenio. El algoritmo de Kachian siempre se ejecuta en un número polinomial de pasos. Brevemente, rodea la región de factibilidad con un elipsoide n-dimensional que sistemáticamente se reduce hasta intersectar la región de factibilidad. Finalmente, intersecta la región de factibilidad en un punto extremo.

Bibliografía

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] B.A. Artwick. *Microcomputer Displays, Graphics, and Animation.* Prentice-Hall, 1985.

[3] N. Deo. *System Dimulation with Digital Computer.* Prentice-Hall, 1979.

[4] G.T. Herman. *Image Reconstruction from Projections.* Springer-Verlag, 1980.

[5] F.S. HIllier and G.J. Lieberman. *Operation Research.* Holden-Day, 1974.

[6] T.C. Hu. *Integer Programming and Network Flows.* Addison-Wesley, 1970.

[7] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipies: The Art of Scientific Computing.* Cambridge University Press, 1986.

[8] K. Maly and A.R. Hausar. *Fundamentals of the Computing Sciences.* Prentice-Hall, 1978.

[9] P. Pavladis. *Algorithms for Graphics and Image Processing.* Computer Science Press, 1982.

[10] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1989.

[11] H. Samet. *Fundamentals of Spatial Data Structures.* Addison-Wesley, 1989.

[12] P.H. Winston. *The Psychology of Computer Vision.* McGraw-Hill, 1975.

[13] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, 1977.

[14] N.Wirth. *Algorithms + Datastructures = Programming.* Prentice-Hall, 1976.