UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONAL
Y DE POSGRADO DEL C.C.H.

INSTITUTO DE INVESTIGACIONES EN MATEMATICAS
APLICADAS Y EN SISTEMAS

"OPTIMIZACION DE LA MEDIDA DE SEMEJANZA PARA OBJETOS TRIDIMENSIONALES USANDO TRANSFORMACIONES PROGRESIVAS"

T E S I S

QUE PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS DE LA COMPUTACION
P R E S E N T A :
FIS. HERMILO SANCHEZ CRUZ

DIRECTOR: DR. ERNESTO BRIBIESCA CORREA

MEXICO, D. F. FEBRERO DE 1998

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

AGRADECIMIENTOS

Agradezco a mi director de tesis, el Dr. Ernesto Bribiesca, por la asesoría y amistad creada durante el tiempo en que realicé este trabajo. A mis sinodales, la M. en C. María Garza, la Mat. Ana Luisa Solís, el Dr. Fabían García y el Dr. Manrique Mata, por sus importes comentarios y sugerencias sobre el contenido y estructura de la tesis.

A mis padres, Hermilo y Magdalena, por el apoyo y el amor tan grandes que me siguen brindado en forma incondicional.

Muy especialmente a la Universidad Nacional Autónoma de México, por el apoyo que me brindó durante el tiempo en que realice mis estudios de maestría así como el de terminación de tesis, sin el cual no hubiese podido concluir mis estudios de manera exitosa.

A mis hermanos, Juan Carlos, Araceli, Paco y Rosy, con quienes sigo jugando de vez en cuando.

A todos los profesores y compañeros de la maestría, de quienes he aprendido mucho.

A las mujeres encargadas de la administración interna en la maestría: Lulú, Viole y Juanita.

ÍNDICE

• RESUMEN

En éste trabajo se utilizan los *invariantes* y la teoría de gráficas para llevar a cabo la transformación de objetos y medir su semejanza. Se deducen dos nuevos invariantes para figuras tridimensionales. Como aportación importante de este trabajo se presenta un algoritmo, llamado *algoritmo Húngaro*, cuya implementación logra optimizar la medida de semejanza de figuras tridimensionales. Se lleva a cabo la *voxelización* de la Cuenca de México con el fin de aplicar, en un trabajo posterior, los invariantes y el algoritmo implementado en este trabajo. Se discute por qué el algoritmo implementado no pudo aplicarse a objetos extraídos de la Cuenca de México, y de cómo el método planteado por uno de los autores aquí citados no es consistente para llevar a cabo dicha implementación, lo que constituye otra aportación de éste trabajo.

•INTRODUCCIÓN

El objetivo del presente trabajo es desarrollar una técnica para obtener una medida de semejanza de objetos o figuras tridimensionales y decir qué tan parecidos son dos objetos. Dicha técnica debe ser la óptima, es decir, el gasto o trabajo realizado en medir la semejanza de las figuras debe ser el menor. Al mismo tiempo, como aplicación específica, desarrollar un software que voxelice el modelo digital de terreno de la Cuenca de México, con la finalidad de que en un trabajo posterior se aplique dicha técnica al terreno y hacer más eficiente el reconocimiento de patrones o relieves diferentes, como pueden ser montañas, cerros, rocas, etc.

Estudiar a través de la computadora y reconocer visualmente patrones u objetos, independientemente de su posición, tamaño y orientación en un campo visual dado, ha sido una de las metas de muchos investigadores de la actualidad. Para lograr esto se han elaborado diferentes técnicas y métodos, principalmente para aplicaciones a la industria de, por ejemplo, máquinas de lectura de caracteres, inspección automática de circuitos, partes de máquina, y en el análisis computacional de material científico como cierto tipo de fotografías.

Aunque en esta tesis se medirá el parecido de figuras en 3D considerando que la *forma* del objeto ha sido identificada, existen muchas teorías de descripción de la *forma*, esto se debe a que es posible conceptualizarla de muchas maneras.

Un análisis de la *forma* puede verse en [30], [31], [32] y [33]. En estos artículos se hace un estudio del sistema visual humano al considerar que la información que llega a una persona consiste de características espaciales como *tamaño*, *forma*, *distancia*, *posición relativa* y *textura*, las cuales son estructuradas por la mente para representar escenas visuales.

Para Zusne [35] los conceptos de *forma*, *figura* o *perfil* son sinónimos, tal como supondremos en éste trabajo. Una colección de contornos, regiones y aristas, se usan para modelar figuras en el espacio bidimensional así como en el espacio tridimensional, el cual se considera como *dominio de la escena.*

Acerca de cómo el cerebro reconoce las *formas* o figuras, ver [37] en el que se explican las habilidades del ser humano para procesar la información.

La mayor parte de las investigaciones que aparecen en la literatura actual está encaminada al reconocimiento de imágenes en dos dimensiones (ver por ejemplo [42], [43] y [44]), una de las contribuciones de este trabajo será estudiar y mejorar técnicas de reconocimiento de figuras en tres dimensiones.

Actualmente el reconocimiento de *objetos*[*] o *figuras* tridimensionales ha sido un campo de interés muy importante para muchos investigadores en lo concerniente al área de Visión por Computadora. Para facilitar el reconocimiento de los objetos (indistintamente hablaremos de objetos o figuras tridimensionales) se han empleado métodos que permiten, primeramente, obtener en forma abstracta algunas características del objeto real que sean de importancia o interés para su estudio, tales como su tamaño, forma, centro de masa, etc. y luego representar al objeto real, utilizando alguna técnica matemática, como el uso de figuras poliédricas, por ejemplo. En nuestro caso lo que se hará es emplear cubos para

[*] Las palabras subrayadas y en cursiva aparecen en el apéndice.

mapear cuerpos rígidos. A esto es lo que llamaremos *voxelizar*. Utilizar cubos y no otro tipo de figuras poliédricas nos facilita mejorar su construcción al querer visualizar los objetos en la pantalla de la computadora, pues se trata de la figura poliédrica más sencilla.

Sobre el reconocimiento de la forma de objetos tridimensionales, hasta ahora han existido algunos autores que han aplicado diferentes técnicas para lograr su propósito, tales como Freeman [38], Besel & Jain [36], Boyse [46], Brooks [40] y Dickinson, Pentland & Rosenfeld [45], entre otros. Todos ellos han realizado el reconocimiento de los objetos basados cada uno en diferentes métodos pero siempre tratando de reconocer figuras regulares o geométricas. Dichos cuerpos siempre están compuestos por planos, líneas rectas, huecos, etc. Por ejemplo Dickinson, Pentland & Rosenfeld [39] basan su método en el uso de *primitivas*, es decir ciertas figuras geométricas que conforman parte de los cuerpos; éstas pueden ser conos, conos truncados, cilindros, elipsoides, etc.

Sin embargo el método planteado en este trabajo se basa en las ideas iniciadas por Bribiesca [2], quien es uno de los primeros investigadores en plantear algoritmos consistentes para reconocer la forma de objetos, no solamente regulares como se había hecho, sino irregulares (o "deformes") y medir su grado de parecido. De esta manera, al alcanzar los objetivos principales de este trabajo, se pretende mejorar el artículo publicado por Bribiesca [2] en el sentido de optimizar la forma de medir el grado de parecido que dicho autor diseñó.

El presente trabajo está estructurado de la siguiente forma: en el capítulo I se hace un estudio de los *invariantes* en dos dimensiones. Tal estudio será muy importante en la realización del capítulo III. Podemos ver a los invariantes como cantidades que nunca cambian a pesar de que cierta imagen o figura cambie de escala (o de tamaño), o bien de que se le haga una rotación o una translación.

Como resultado del estudio de los invariantes en dos dimensiones, se deducen dos nuevos invariantes para figuras tridimensionales, lo cual constituye un aporte de este trabajo.

En el capítulo II se hace un estudio sobre teoría de gráficas, en la parte de asignación óptima; se presenta además un algoritmo, llamado *algoritmo Húngaro*. La implementación de tal algoritmo será muy importante en este trabajo, pues nos permitirá optimizar la transformación de una figura a otra. Como resultado de la implementación de este algoritmo se obtiene un diagrama de flujo, el cual constituye una aportación de este trabajo. Las transformaciones se lograrán al mover los *voxels* de un objeto a otro, lo cual se lleva a cabo en el capítulo III, una vez que se aplican los invariantes encontrados en el capítulo I. Los *voxels* que se hayan movido generarán un cierto trabajo realizado desde el punto de vista de la física clásica, el cual se medirá y se obtendrá un valor para la medida de semejanza de los objetos transformados. Con ésto se completará la información para saber si dos cuerpos son semejantes y, como aportación importante de este trabajo, se logrará optimizar la medida de semejanza de figuras tridimensionales.

En el capítulo IV se explica cómo se forma una imagen, cómo se representan matemáticamente los objetos y finalmente, en el Capítulo V se crea una matriz binaria a partir de los datos de las elevaciones de la Cuenca de México. Posteriormente, como parte de una aportación más, se lleva a cabo la *voxelización* de la Cuenca de México, con el fin de aplicar en un trabajo posterior, los invariantes y el algoritmo implementado en este trabajo. Se discute por qué el algoritmo implementado no pudo aplicarse a objetos extraídos de la Cuenca de México.

Posteriormente se dan las conclusiones, en las que se exponen los logros alcanzados en éste trabajo, y de cómo el método planteado por uno de los autores aquí citados no es consistente para llevar a cabo la implementación del algoritmo Húngaro, que nos permita trabajar con una matriz de cualquier tamaño. Detectar dicha inconsistencia representa una aportación más de este trabajo.

• CAPÍTULO I

TEORÍA DE LOS INVARIANTES

En este capítulo, presentaremos una parte de la herramienta matemática que nos ayudará a entender el problema de invariancia de los objetos bajo transformaciones afines. Para ello se darán a conocer los llamados *invariantes*, basados en el concepto de *momentos*. Los invariantes, junto con las asignaciones que se harán entre diferentes objetos en el próximo capítulo, nos permitirán decir qué tan semejantes son los objetos.

1.1 ECUACIONES DE RECUPERACIÓN 3D

Supongamos que un modelo de objeto está especificado por los parámetros $\alpha_1,...,\alpha_m$, los cuales pueden ser las coordenadas de puntos particulares que describen la escena del objeto, o los coeficientes de las ecuaciones que definen las superficies, las aristas y las caras. Si el objeto está en movimiento, estos parámetros pueden contener las velocidades de traslación y rotación. Sean $c_1,...,c_n$ los parámetros que caracterizan a la imagen, los cuales pueden reflejar los niveles de intensidad, por ejemplo, o bien la textura de la superficie, el relieve o la reflectancia luminosa. Puesto que el modelo del objeto está parametrizado, las características de la imagen a observarse pueden derivarse teóricamente de una geometría de imágenes de proyección de perspectivas [10], resultando ecuaciones como:

$$c_i = F_i(\alpha_1,...,\alpha_m), \quad i = 1,...,n.$$

Estas se llaman ecuaciones de recuperación en 3D, donde los valores de los parámetros $\alpha_1,...,\alpha_m$ pueden determinarse al resolver dicha ecuación después de sustituir los valores observados $c_1,...,c_n$.

Sin embargo, para casi todos los problemas, las ecuaciones de recuperación en 3D son no lineales y difíciles de resolver analíticamente. Una forma de resolver estas ecuaciones

podría ser al intentar con todos los posibles valores de $\alpha_1,...,\alpha_m$, ir calculando $c_i=F_i(\alpha_1,...,\alpha_m)$, y después ir checando si los valores de $c_1,...,c_n$ coinciden con los valores observados. Este proceso, aunque pudiese ser relativamente rápido de resolver, sería impráctico, pues requeriría de mucha memoria.

De lo anterior, sería mejor obtener la solución de una forma cerrada. En este capítulo se presentará una herramienta poderosa, que nos permite hacer esto, encontrando soluciones analíticas al considerar que las ecuaciones de recuperación en 3D tienen una "estructura" que refleja la geometría (de la imagen) de proyección de perspectiva.

Lo que haremos será enfocarnos en las *propiedades de invariancia* de un objeto y en las características de la imagen. El sistema de coordenadas de la imagen no es inherente a la imagen ni al objeto, solo juega un papel auxiliar para representar al objeto y dar su posición espacial, así que, cualquier sistema coordenado obtenido por rotaciones se puede usar en forma equivalente. Se verá que las ecuaciones de recuperación en 3D se pueden expresar en términos de *invariantes*, y las soluciones obtenerse en forma cerrada. El "significado" geométrico del los parámetros del objeto se hace muy claro si se expresan en términos de invariantes.

1.2 INTERPRETACIÓN DE LOS INVARIANTES

Considérese un objeto geométrico S en el espacio X. Se supone un grupo de transformaciones admisibles G que actúa en el espacio X. Un *invariante escalar* de un objeto S es una cantidad que no cambia su valor cuando el objeto S sufre cualquiera de las transformaciones admisibles.

Supongamos que el objeto S tiene invariantes escalares $I_1,I_2,...,I_n$. Consideremos otro objeto S' para el cual pueden estar definidos los mismos invariantes escalares. Si el objeto S' se obtiene al transformar apropiadamente al objeto S usando transformaciones admisibles, los valores de estos invariantes escalares deben ser idénticos. Sin embargo, en general, lo

contrario no es cierto. Los valores de ambos invariantes escalares pueden ser los mismos para diferentes objetos. Por lo tanto, podemos definir una *base invariante*.

Definición: se dice que un conjunto de invariantes escalares es una base invariante *si la coincidencia de valores para dos objetos implica la existencia de una transformación admisible que mapea uno al otro* [10].

1.3 MOMENTOS E INVARIANTES ALGEBRAICOS

Los momentos bidimensionales de orden (p + q) de una función de distribución $\rho(x,y) \geq 0$ están definidos en términos de la integral [29]:

$$m_{pq} = \iint_{\Re} x^p y^q \rho(x, y)\, dxdy. \qquad (1)$$

donde $\rho(x,y)$ es una función real definida en una región finita $\Re$.

El Teorema de unicidad en la teoría de los invariantes dice que la sucesión de momentos $\{m_{pq}\}$ está determinada de manera única por ρ(x ,y); y viceversa, ρ(x ,y) está determinada únicamente por $\{m_{pq}\}$.

La función característica φ(u,v) y la función generadora de momentos M(u,v) de ρ(x,y) se definen como

$$\varphi(u,v) = \iint_{\Re} exp(iux + ivy)\rho(x, y)\, dxdy. \qquad (2a)$$

$$M(u,v) = \iint_{\Re} exp(ux + vy)\rho(x, y)\, dxdy, \qquad (2b)$$

donde u y v son números reales. Si existen los momentos de todos los ordenes, entonces las dos funciones anteriores pueden expandirse en series de potencias en términos de m_{pq}:

$$\varphi(u,v) = \sum_{p=0}^{\infty}\sum_{q=0}^{\infty} m_{pq}(iu)^p(iv)^q/p!q!\,, \qquad (3)$$

$$M(u,v) = \sum_{p=0}^{\infty}\sum_{q=0}^{\infty} m_{pq}u^pv^q/p!q!\,. \qquad (4)$$

Si se conoce la función característica $\varphi(u,v)$, se puede obtener $\rho(x,y)$ de la transformada inversa de Fourier:

$$\rho(x,y) = 1/2\pi \iint_{\Re} exp(-iux - ivy)\, \varphi(u,v)\, dudv. \qquad (5)$$

1.4 MOMENTOS CENTRALES

Los momentos centrales μ_{pq} se definen como [29]:

$$\mu_{pq} = \iint_{\Re} (x-\xi)^p(y-\eta)^q\rho(x, y)\, dxdy, \qquad p.q \in N. \qquad (6)$$

donde $\xi = m_{10}/m_{00}$, $\eta = m_{01}/m_{00}$.

Teorema: los momentos centrales son invariantes bajo *translación* (ver apéndice), en la que se tienen las siguientes transformaciones de coordenadas:

$$x' = x + \alpha$$
$$y' = y + \beta \qquad \text{con } \alpha \text{ y } \beta \text{ constantes.} \tag{7}$$

De la relación para los momentos centrales, se pueden obtener los momentos ordinarios siguientes [29]:

$$\begin{aligned}
&\mu_{00} = m_{00} \equiv \mu, \quad \mu_{10} = \mu_{01} = 0, \\
&\mu_{20} = m_{20} - \mu \xi^2, \quad \mu_{11} = m_{11} - \mu \xi\eta, \\
&\mu_{02} = m_{02} - \mu\eta^2, \\
&\mu_{30} = m_{30} - 3 m_{20} \xi + 2\mu\xi^3, \\
&\mu_{21} = m_{21} - m_{20}\eta - 2\mu m_{11}\xi + 2\mu \xi^2\eta, \\
&\mu_{12} = m_{12} - m_{02}\xi - 2 m_{11}\eta + 2\mu \xi \eta^2, \\
&\mu_{03} = m_{03} - 3 m_{02}\eta + 2\mu \eta^3.
\end{aligned} \tag{8}$$

Por simplicidad, μ_{pq} puede expresarse como

$$\mu_{pq} = \iint_{\Re} x^p y^q \rho(x, y)\, dxdy, \qquad p,q \in N. \tag{9}$$

1.5 FORMAS ALGEBRAICAS E INVARIANTES.

Se define un polinomio homogéneo como

$$f = a_{p,0}u^p + \binom{p}{1} a_{p-1,1} u^{p-1}v + \binom{p}{2} a_{p-2,2} u^{p-2}v^2 + \ldots + \binom{p}{p-1} a_{1,p-1} uv^{p-1} + a_{0,p}v^p, \tag{10}$$

en su forma binaria algebraica. Usando una forma compacta lo anterior se puede escribir como:

$$f = (a_{p,0}; a_{p-1,1}; \ldots ; a_{1,p-1}; a_{0,p})(u,v)^p. \qquad (11)$$

Un polinomio homogéneo *I(a)* es un invariante algebraico de peso ω si:

$$I(a'_{p,0}; a'_{p-1,1}; \ldots ; a'_{1,p-1}; a'_{0,p}) = \Delta^{\omega} I(a_{p,0}; a_{p-1,1}; \ldots ; a_{1,p-1}; a_{0,p}), \qquad (12)$$

donde $a'_{p,0}; a'_{p-1,1}; \ldots ; a'_{1,p-1}; a'_{0,p}$ son los coeficientes obtenidos de las transformaciones lineales aplicadas al polinomio original y Δ es el determinante de la transformación. Dichas transformaciones están dadas por la forma general:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \alpha & \gamma \\ \beta & \delta \end{bmatrix} \begin{bmatrix} u' \\ v' \end{bmatrix} \qquad \Delta = \begin{bmatrix} \alpha & \gamma \\ \beta & \delta \end{bmatrix} \neq 0 \qquad (13)$$

Si $\omega = 0$ el invariante es *absoluto*, y si $\omega \neq 0$ el invariante es *relativo*

1.6 MOMENTOS INVARIANTES DE ESCALA

Para el cambio de la escala, o de tamaño, se tiene la siguiente transformación:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (14)$$

donde α es una constante

1.7 MOMENTOS INVARIANTES ORTOGONALES:

Bajo transformaciones ortogonales o de *rotación* (ver apéndice) se tiene:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & sen\theta \\ -sen\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{15}$$

El *teorema fundamental de los momentos invariantes* [11] nos dice que, si la forma algebraica de orden p tiene un invariante algebraico

$$I(a'_{p,0};\ a'_{p-1,1};\ \ldots\ ;\ a'_{1,p-1};\ a'_{0,p}) = \Delta^{\omega} I(a_{p,0};\ a_{p-1,1};\ \ldots\ ;\ a_{1,p-1};\ a_{0,p}), \tag{16}$$

entonces los momentos de orden p tienen el mismo invariante pero con un factor adicional J :

$$I(\mu'_{p0};\ \mu'_{p-1,1};\ \ldots\ ;\ \mu'_{1,p-1};\ \mu'_{0p}) = |J|\Delta^{\omega} I(\mu_{p0};\ \mu_{p-1,1};\ \ldots\ ;\ \mu_{1,p-1};\ \mu_{0p}). \tag{17}$$

Como $|J| = 1$ para las transformaciones ortogonales, entonces los momentos invariantes son exactamente los mismos que los invariantes algebraicos. Consideremos los momentos como coeficientes de una forma algebraica

$$(\mu_{p,0};\ \mu_{p-1,1};\ \ldots\ ;\ \mu_{1,p-1};\ \mu_{0,p})\ (u,v)^p, \tag{18}$$

y usemos la siguiente transformación ortogonal

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} cos\theta & sen\theta \\ sen\theta & -cos\theta \end{bmatrix} \begin{bmatrix} u' \\ v' \end{bmatrix} \tag{19}$$

Sustituyamos u,v,u' y v' en la siguiente transformación [10]:

$$\begin{bmatrix} U \\ V \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}, \quad \begin{bmatrix} U' \\ V' \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix}\begin{bmatrix} u' \\ v' \end{bmatrix}$$

entonces, tenemos que

$$U' = Ue^{-i\theta}, \quad V' = V e^{i\theta}. \tag{20}$$

Al sustituir en los momentos (los coeficientes de una forma algebraica anterior) se tiene que

$$\begin{aligned}(I_{p,0}, \ldots, I_{0,p})(U,V)^p &= (\mu_{p,0}; \mu_{p-1,1}; \ldots; \mu_{1,p-1}; \mu_{0,p})\,(U,V)^p, \\ &= (\mu'_{p,0}; \mu'_{p-1,1}; \ldots; \mu'_{1,p-1}; \mu'_{0,p})\,(u',v')^p, \\ &= (I'_{p,0}, \ldots, I'_{0,p})(U e^{-i\theta}, V e^{i\theta})^p,\end{aligned} \tag{21}$$

donde $I_{p,0}, \ldots, I_{0,p}$ y $I'_{p,0}, \ldots, I'_{0,p}$ son los coeficientes que se obtienen después de las substituciones. De lo anterior se llega a que

$$I'_{p,0} = e^{ip\theta} I_{p0}; \; I'_{p-1,1} = e^{i(p-2)\theta} I_{p-1,1}; \; \ldots; \; I'_{1,p-1} = e^{-i(p-2)\theta} I_{1,p-1}; \; I'_{0,p} = e^{-ip\theta} I_{0,p}; \tag{22}$$

Las cuales conforman (p+1) momentos invariantes linealmente independientes, bajo transformaciones ortogonales, en la que $\Delta = e^{i\theta}$.

De (21) se tiene que $I_{r,p-r}$ es el complejo conjugado de $I_{p-r,r}$, entonces:

$$I_{p,0} = \mu_{p0}u^p - i\binom{p}{1}\mu_{p-1,1} - \binom{p}{2}\mu_{p-2,2} + \binom{p}{3}\mu_{p-3,3} + \ldots + (-i)^p\mu_{p,0}, \qquad (23)$$

$$I_{p-1,1} = (\mu_{p,0} + \mu_{p-2,2}) - i(p-2)(\mu_{p-1,1} + \mu_{p-3,3}) + \cdots + (-i)^{p-2}(\mu_{2,p-2} + \mu_{0,p})$$

$$I_{p-2,2} = (\mu_{p,0} + 2\mu_{p-2,2} + \mu_{p-4,4}) - i(p-4)(\mu_{p-1,1} + 2\mu_{p-3,3} + \mu_{p-5,5}) + \ldots + (-i)^{p-4}(\mu_{4,p-4} + 2\mu_{2,p-2} + \mu_{0,p})$$

... ...

$$I_{p-r,r} = \{(\mu_{p,0}; \mu_{p-2,2}; \cdots; \mu_{p-2r,2r})(1,1)^r; (\mu_{p-1,1}; \mu_{p-3,3}; \cdots; \mu_{p-2r-1,2r+1})(1,1)^r; \cdots; (\mu_{2r,p-2r}; \mu_{2r+2,p-2r-2}; \cdots; \mu_{0,p})(1,1)^r\}(1,-i)^{p-2r}, \qquad p - 2r > 0.$$

...

y

$$I_{p/2,p/2} = \mu_{p,0} + i\binom{p/2}{1}\mu_{p-2,2} + \binom{p/2}{2}\mu_{p-4,4} + \cdots \mu_{0,p}, \qquad \text{con p un número par.}$$

Estas (p+1) ecuaciones son linealmente independientes en I y en μ.

Por otro lado, utilizando la siguiente transformación ortogonal impropia (de reflexión):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & sen\theta \\ sen\theta & -cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad (19)$$

Tenemos, de forma similar al caso de la rotación:

$$U' = Ve^{i\theta}, \quad V' = Ue^{-i\theta}. \tag{24}$$

y

$$I'_{p,0} = e^{-ip\theta} I_{p,0}; \; I'_{p-1,1} = e^{-i(p-2)\theta} I_{1,p-1}; \; \ldots \; ; I'_{1,p-1} = e^{i(p-2)\theta} I_{p-1,1}; \; I'_{0,p} = e^{ip\theta} I_{p,0}; \tag{25}$$

Utilizando (8), (22), (23) y (25) y con un poco de álgebra, se llega a que

$$\begin{aligned}
&I'_{1,1} I'_{1,1} = I_{1,1} I_{1,1}\,, \\
&I'_{2,0} I'_{0,2} = I_{2,0} I_{0,2}\,, \\
&I'_{3,0} I'_{0,3} = I_{3,0} I_{0,3}\,, \\
&I'_{2,1} I'_{1,2} = I_{2,1} I_{1,2}\,, \\
&I'^{3}_{3,0} I'^{3}_{1,2} + I'^{3}_{0,3} I'^{3}_{2,1} = I^{3}_{3,0} I^{3}_{1,2} + I^{3}_{0,3} I^{3}_{2,1}\,, \\
&(1/i)(I'^{3}_{3,0} I'^{3}_{1,2} - I'^{3}_{0,3} I'^{3}_{2,1}) = (1/i)(I^{3}_{3,0} I^{3}_{1,2} - I^{3}_{0,3} I^{3}_{2,1}). \\
&I'^{2}_{2,0} I'^{2}_{1,2} + I'^{2}_{0,2} I'^{2}_{2,1} = I^{2}_{2,0} I^{2}_{1,2} + I^{2}_{0,2} I^{2}_{2,1}\,.
\end{aligned} \tag{26}$$

Utilizando lo anterior se llega a los siguientes 7 invariantes absolutos:

$$\begin{aligned}
&\phi_1 = \mu_{20} + \mu_{02}\,, \\
&\phi_2 = (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2\,, \\
&\phi_3 = (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{03})^2, \\
&\phi_4 = (\mu_{30} + \mu_{12})^2 + (\mu_{21} + \mu_{03})^2, \\
&\phi_5 = (\mu_{30} - 3\mu_{12})(\mu_{30} + \mu_{12})[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] + (3\mu_{21} - \mu_{03})(\mu_{21} - \mu_{03})[(\mu_{30} \\
&\quad + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2]\,, \\
&\phi_6 = (\mu_{20} - \mu_{02})[(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2] + 4\mu_{11}(\mu_{30} + \mu_{12})(\mu_{21} + \mu_{03}). \\
&\phi_7 = (3\mu_{21} - \mu_{03})(\mu_{30} + \mu_{12})[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] \\
&\quad - (\mu_{30} - 3\mu_{12})(\mu_{21} + \mu_{03})[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2],
\end{aligned} \tag{27}$$

donde hemos calculado

$\phi_1 = I'_{1,1}I'_{1,1} = I_{1,1}I_{1,1}$, $\phi_2 = I'_{2,0}I'_{0,2} = I_{2,0}I_{0,2}$, ... etc.

Los momentos invariantes ortogonales absolutos, dados por (27) pueden utilizarse directamente para identificar un patrón independientemente de la orientación, posición y cambio de escala [11].

1.8 ANÁLISIS.

El emplear los invariantes dados por (27) ha permitido el reconocimiento de patrones o imágenes en dos dimensiones. El problema se complica cuando queremos reconocer patrones o figuras tridimensionales, ya que en este caso, deberíamos definir los momentos centrales en forma análoga a (6) como:

$$\mu_{pqr} = \iiint_{\Re} (x-\xi)^p (y-\eta)^q (z-\zeta)^r \rho(x,y,z)\, dxdydz, \qquad p,q,r \in N. \tag{28}$$

Para efectos de nuestro trabajo, haremos $\rho(x,y,z) = 1$, pues no nos importarán cualidades de la figura, tal como sus diferencias de luminosidad, es decir, sólo nos interesa la forma del objeto. Así que la ecuación (28) se reduce a

$$\mu_{pqr} = \iiint_{\Re} (x-\xi)^p (y-\eta)^q (z-\zeta)^r\, dxdydz, \qquad p,q,r \in N. \tag{29}$$

La cual es muy fácil probar que es invariante bajo traslación, al considerar la siguiente transformación:

$$x' = x + \alpha$$
$$y' = y + \beta \qquad \text{con } \alpha, \beta \text{ y } \gamma \text{ constantes.} \qquad (30)$$
$$z' = z + \gamma$$

Y nuevamente se cumple que los momentos centrales dados por (29) son invariantes bajo traslación.

Ahora, para deducir los invariantes bajo cambio de escala, utilicemos la siguiente transformación:

$$x' = \alpha x$$
$$y' = \alpha y \qquad \text{con } \alpha \text{ constante.} \qquad (31)$$
$$z' = \alpha z$$

Al considerar esta transformación de coordenadas y utilizarlas en 29), se llega a que

$$\mu'_{pqr} = \alpha^{p+q+r+3} \mu_{pqr}\,, \qquad (32)$$

Si $p = q = r = 0$, tenemos que $\mu'_{000} = \alpha^3 \mu_{000}$. (33)

Sea $\mu_{000} = \mu$, entonces, de (32) y (33) se tiene el siguiente invariante normalizado:

$$\frac{\mu'_{pqr}}{\mu'^{(p+q+r)/3+1}} = \frac{\mu_{pqr}}{\mu^{(p+q+r)/3+1}}, \qquad \text{con } p + q + r = 3,4, \ldots \qquad (34)$$

El cual es invariante bajo cambios de escala.

Sin embargo, al tratar de encontrar los momentos invariantes bajo rotación en 3-D, como los equivalentes a los encontrados por Hu [11] para imágenes en 2D, el tratamiento

matemático se hace muy complicado. De hecho ningún autor hasta ahora ha deducido invariantes en 3-D utilizando el método de Hu. Las matrices de rotación como la dada por (15) aumenta solo en una columna y un renglón, pero, al seguir desarrollando y querer llegar a ecuaciones equivalentes a 3-D como las de (23) las matemáticas se complican demasiado.

Pero nosotros creemos que no es necesario deducir tales invariantes, pues para ello proponemos en este trabajo utilizar (29) y (34) además de los ejes mayores de los objetos, así como sus centros de masa, con estos dos últimos obtendremos invariancia bajo rotación. Lo anterior nos proporcionará 4 invariantes que serán robustos y suficientes para el reconocimiento de figuras en 3-D.

• CAPÍTULO II

GRÁFICAS BIPARTITAS Y ASIGNACIÓN ÓPTIMA

En este capítulo veremos que el estudio de graficación, en la parte de asignación óptima, es necesario para entender el problema y la solución de encontrar un algoritmo de optimización que nos permita obtener el *trabajo mínimo* posible para transformar un objeto en otro. Cabe notar que la implementación y uso de este algoritmo para un propósito como el de este trabajo no se había realizado con anterioridad [2].

2.1 TEORÍA DE GRÁFICAS

De la teoría de gráficas (véase por ejemplo [13]) tenemos que una *gráfica* es un conjunto finito, no vacío, compuesto de un conjunto de *vértices* V junto con una relación irreflexiva R en V (es decir, si $(u,v) \in R$, entonces $(v,u) \in R$). Denotemos por E al conjunto de parejas en R. De esta manera una gráfica puede definirse simplemente a través de los conjuntos V y E como $G = G\langle V,E\rangle$. donde E incluye elementos de la forma $\{u,v\}$.

En una gráfica $G = G\langle V,E\rangle$, donde V es el conjunto de vértices, cada elemento de V es un *vértice*. Al número de vértices que hay en V: $|V|$ se le llama *orden* de la gráfica G. Cada elemento de E se llama *arista*. E es el conjunto de aristas de G. El número de aristas que hay en E se llama *el tamaño* de G, denotado como $|E|$

2.2 PESOS EN LAS ARISTAS

A cada arista de una gráfica G, asociemos un número real w(e), al cual llamaremos su *peso*. De esta manera G, junto con los pesos en sus aristas, es llamada *gráfica pesada*.

Las gráficas pesadas aparecen constantemente en aplicaciones de la teoría de gráficas. Por ejemplo, en una gráfica que represente líneas de comunicación, los pesos podrían representar los costos de construcción o de mantenimiento de las varias líneas de comunicación. Otro ejemplo de aplicación es considerar los vértices como ciudades o poblados conectados por diferentes caminos (aristas de la gráfica), los pesos indican las distancias entre los poblados (las cuales pueden estar dadas en kilómetros, por ejemplo). El problema a resolver en este caso podría ser: pavimentar estos caminos con el objeto de conectar a todos los poblados y gastar la menor cantidad de material posible.

Si H es una subgráfica de la gráfica pesada, el peso $w(H)$ de H es la suma de los pesos de las aristas de H: $\sum w(e)$, donde $e \in E(H)$. Muchos problemas de optimización intentan buscar, en una gráfica pesada, una subgráfica de cierto tipo con un peso mínimo (o máximo).

Cuando se desea que la suma de los pesos de las aristas que conforman una subgráfica H sea la menor, esta subgráfica debe ser *conectada* (es decir que exista una trayectoria o camino entre cualquiera dos de sus vértices) y *acíclica* (cuando no hay trayectorias cerradas y en consecuencia H es un *árbol*) entonces se obtiene lo que se llama un *spanning tree de peso mínimo* (toda gráfica conectada G contiene una subgráfica que es un árbol llamada *spanning tree*).

El peso mínimo de una gráfica es aquel tal que el peso $w(H) = \sum w(e)$ sea el menor de todas las subgráficas conectadas.

En la gráfica de la Fig. 2.1 las aristas remarcadas representan una subgráfica de *peso mínimo* (*spanning tree* de peso mínimo), es decir, la suma de los pesos en esas aristas que conectan a todos los poblados es el menor de todos los casos posibles.

Existe un problema en el que encontrar dicha subgráfica se considera como el problema de encontrar la *trayectoria más corta*: dada una red que conecta a varios poblados, determinar la ruta más corta entre dos poblados específicos dentro de la red. En este caso, se debe encontrar una trayectoria de peso mínimo, que conecte dos vértices específicos u_o y v_o; los pesos representan las distancias entre dos poblados diferentes, y por

tanto los pesos deben ser siempre positivos. La trayectoria indicada en la Fig. 2.2 es una trayectoria, que va de u_o a v_o, de peso mínimo.

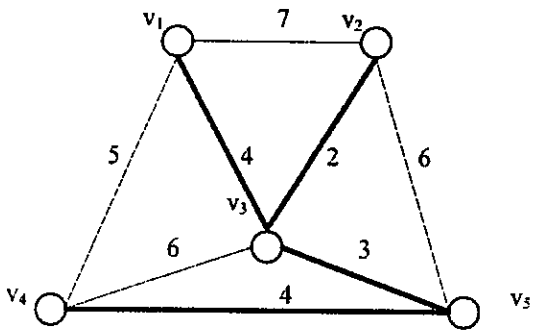


Fig. 2.1. *Spanning tree* cuyo peso es el mínimo de todos los *spanning trees* que se pueden obtener de la gráfica original.

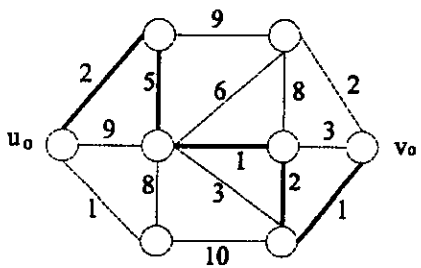


Fig. 2.2. Trayectoria en negro que va de u_o a v_o de peso mínimo.

Para resolver este tipo de problemas existen diferentes algoritmos, tal como el algoritmo BFS, o el algoritmo de Dijkstra, los cuales pueden encontrarse en [14] ó [15]. Sin embargo, el tipo de problema que a nosotros nos interesa tiene que ver en particular con las *gráficas bipartitas*, que a continuación se definen y, posteriormente, daremos a conocer un

algoritmo cuya implementación forma parte de este trabajo, pues nos permitirá encontrar el peso mínimo en dicho tipo de gráficas.

2.3 GRÁFICAS BIPARTITAS

Una gráfica $G(V,E)$ es *bipartita* si existe una partición de los vértices en dos conjuntos U y V, tales que si (u,v) está en E implica que $u \in U$ y $v \in V$ (o al revés).

En la Fig. 2.3 se presenta un ejemplo de gráfica bipartita con 8 vértices en total.

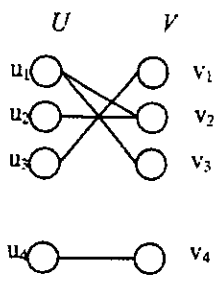


Fig. 2.3 Gráfica bipartita.

2.4 *MATCHING*

Sea $M \subseteq E$ un subconjunto de aristas de la gráfica G, el cual es llamado *matching* en G si cada uno de sus vértices son adyacentes a, a lo más, una arista de M; los dos extremos de una arista en M se dicen que son *matched* bajo M. Se dice que un *matching M satura* a un vértice v, que sea extremo de una arista en M, y entonces v es M-saturado si alguna arista de M es adyacente a v, de otra forma, v es *M-no saturado*. Si todos los vértices de G son M-saturados, el *matching M* es *perfecto* (Fig. 2.4 y 2.5).

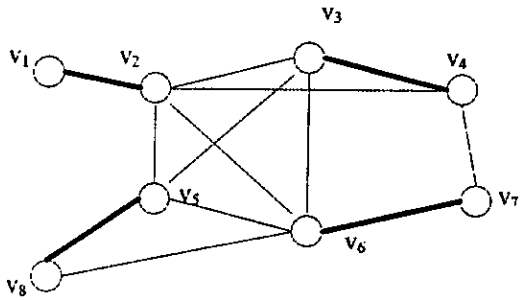


Fig. 2.4 Un *matching* perfecto.

En el caso de una gráfica bipartita, la Fig. 2.5 muestra que todos los vértices están saturados, pues cada uno de ellos es adyacente a una arista en M (las aristas en M están marcadas en línea gruesa) y por lo tanto se trata de un *matching* perfecto.

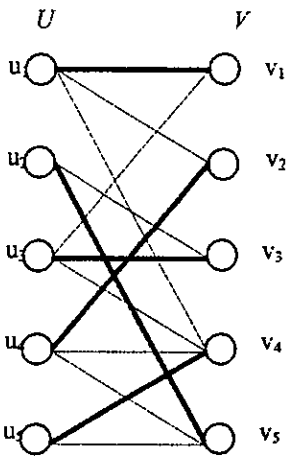


Fig. 2.5. Gráfica bipartita con un *matching* perfecto.

En este último ejemplo, tenemos que esta gráfica bipartita es de la forma $G = G\langle V,E\rangle$, donde

$V = \{u_1,u_2,u_3,u_4,u_5,v_1,v_2,v_3,v_4,v_5\}$,

$E=\{\{u_1,v_1\},\{u_1,v_2\},\{u_1,v_4\},\{u_2,v_3\},\{u_2,v_5\},\{u_3,v_3\},\{u_3,v_4\},\{u_4,v_2\},\{u_4,v_4\},\{u_4,v_5\},\{u_5,v_4\},\{u_5,v_5\}\}$, y

$M = \{\{u_1,v_1\},\{u_2,v_5\},\{u_3,v_3\},\{u_4,v_2\},\{u_5,v_4\}\}$.

Sea M un *matching* en G, un *camino alternante* en G es un camino cuyas aristas están alternativamente en $E\backslash M$ y en M. Un *augmenting path* es un camino alternante cuyo inicio y final no están saturados. Por ejemplo, en la Fig. 2.6 el camino v_4,v_3,v_1,v_2,v_6,v_5, muestra un *augmenting path.*

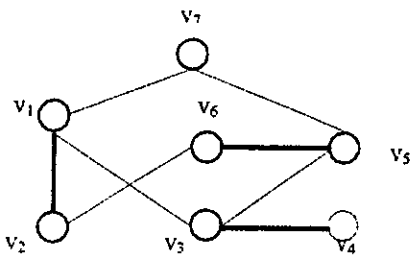


Fig. 2.6. El camino v_4,v_3,v_1,v_2,v_6,v_5 , es un *augmenting path.*

2.5 PROBLEMA DE ASIGNACIÓN ÓPTIMA

Consideremos los siguientes conjuntos de vértices: $X = \{x_1, x_2, \ldots, x_n\}$ y $Y = \{y_1,y_2, \ldots, y_n\}$. El primer conjunto podría representar, por ejemplo, a n trabajadores, y el segundo a n puestos de trabajo, los cuales deben ser ocupados por los n trabajadores. Algunos de estos trabajadores tienen la capacidad de ocupar ciertos puestos de trabajo; generándose así una gráfica bipartita como la siguiente:

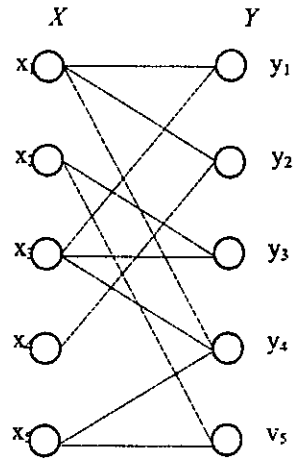


Fig. 2.7. Cinco trabajadores a los que se les puede asignar 5 puestos.

Existe un método llamado *método Húngaro* [14] (que es el método clásico), que puede encontrar una forma eficiente de asignar a cada trabajador un puesto de trabajo, generándose un *matching* perfecto, y llegar a una asignación como la dada por la Fig. 2.8. en la que se ha dado un único puesto de trabajo a cada trabajador.

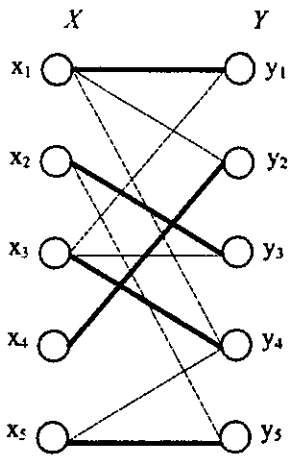


Fig. 2.8. El método *Húngaro*, permite encontrar *matchings* perfectos.

Sin embargo, se podría querer tomar en cuenta la efectividad de los trabajadores en sus diferentes trabajos, en términos de, por ejemplo, las ganancias de la compañía. En este caso uno podría estar interesado en una asignación que maximice la efectividad total de los trabajadores.

En el caso del problema que nos atañe en este trabajo, se trata de tomar en cuenta las distancias que los *voxels* que conforman a un objeto tienen que recorrer, para transformarlos en otro objeto, tal que la suma de esas distancias sea la menor de todas las posibles. El problema de encontrar tal tipo de asignaciones se conoce como el *problema de asignación óptima* [14].

Consideremos una gráfica bipartita pesada completa con bipartición (X,Y), donde $X=\{x_1, x_2, \ldots, x_n\}$, $Y=\{y_1, y_2, \ldots, y_n\}$. El conjunto X puede representar al conjunto de *voxels* de un objeto a transformar y el conjunto Y a los *voxels* del objeto transformado. Las aristas x_iy_j tienen un peso $w_{ij} = w(x_iy_j)$, los cuales representan la distancias que deben recorrer los *voxels* para transformar los objetos (o en el caso de los trabajadores, representan la efectividad del trabajador x_i en el puesto y_j).

El problema de asignación óptima que nos incumbe, en esta gráfica pesada, es equivalente a encontrar un *matching* perfecto de peso mínimo. Nos referiremos a este *matching* como un *matching óptimo*. La Fig. 2.9 muestra un ejemplo de encontrar un *matching* óptimo.

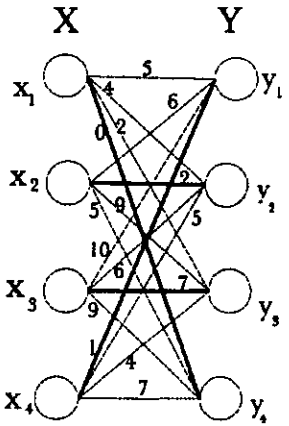


Fig. 2.9. Un *matching* óptimo, dado por las aristas remarcadas en negro.

El problema de asignación óptima consiste en encontrar un *matching* perfecto con las siguiente condición:

- La suma de los pesos de las aristas en el *matching* perfecto debe ser la más pequeña de todos los posibles *matchings* perfectos que pudiesen existir en la gráfica, es decir queremos encontrar el *peso mínimo* de una gráfica bipartita.

Para resolver el problema de asignación óptima se podrían enumerar todos los n! *matchings* perfectos y encontrar uno que sea el óptimo entre ellos. Otra forma mejor, sería el que realizó Bribiesca [2] cuya complejidad es $O(n^2)$. El algoritmo dado por Bribiesca es consistente para realizar la transformación entre los objetos. Sin embargo su método propuesto para mover los *voxels* no minimiza el trabajo desarrollado al llevar a cabo tal transformación.

Por ejemplo, Bondy [14] y Tarjan [42] presentan un algoritmo que resuelve el problema de asignación al encontrar un *matching* óptimo de *peso máximo*. En nuestro caso nos basaremos en Gould [14] para encontrar el *matching* óptimo de peso mínimo el cual se basa en el llamado *algoritmo Húngaro*.

Lo mismo que Gould [14], Bondy [14] se basa en el método Húngaro para encontrar un *matching* óptimo de *peso máximo*.

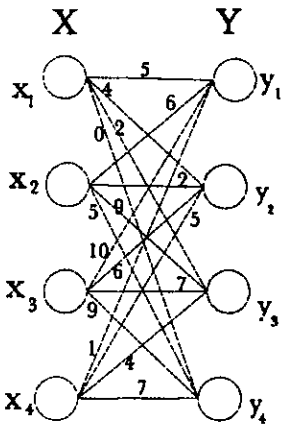
Para Tarjan [42] por otro lado, una gráfica bipartita puede verse como un caso especial de un problema de *red de flujo* y presenta un algoritmo para resolver el problema del *matching* óptimo de peso máximo usando un tiempo de $O(nm\log_{(2+m/n)}n)$, donde n es el número de vértices y m el de aristas. Sin embargo, este algoritmo es equivalente al algoritmo Húngaro, tal como lo explica Tarjan en [42].

A continuación se presenta el algoritmo Húngaro para encontrar un *matching* óptimo, en una gráfica bipartita pesada y que minimiza el trabajo para transformar a los objetos.

2.6 ALGORITMO HÚNGARO

Definamos el peso de un *matching* M como $W(M) = \sum w(e)$, $e \in M$. Así, una solución óptima es un *matching* perfecto con $W(M)$ como mínimo.

1. Comencemos con representar nuestra gráfica bipartita en forma de matriz de tamaño n x n, $U = [w_{i,j}]$ donde $w_{i,j}$ es el peso de la arista que va de x_i a y_j, esto es, la distancia que debe recorrer un *voxel* de la posición x_i a la posición y_j al transformar dos objetos. La Fig. 2.10 muestra un ejemplo.



a)

U	y_1	y_2	y_3	y_4
x_1	5	4	2	0
x_2	6	2	9	5
x_3	10	6	7	9
x_4	1	5	4	7

b)

Fig. 2.10. a) Una gráfica bipartita b) con representación matricial.

2. *Minimizar* la matriz de la siguiente manera:

- Elegir el valor más pequeño de cada renglón y restarlo a ese renglón.
- Elegir el valor más pequeño de cada columna y restarlo a esa columna.

Es importante hacer notar que al realizar estos dos cálculos la solución final no cambia, pues al seleccionar sólo una entrada de un renglón o columna el valor de W(M) para un *matching* M será reducido por la misma cantidad.

Nuestra matriz ejemplo queda como

U	y_1	y_2	y_3	y_4
x_1	5	4	2	0
x_2	6	2	9	5
x_3	10	6	7	9
x_4	1	5	4	7

a)

U	y_1	y_2	y_3	y_4
x_1	0	0	9	2
x_2	4	0	7	3
x_3	4	0	1	3
x_4	0	4	3	6

b)

Fig. 2.11. a) matriz original. b) matriz minimizada.

3. Ahora, se seleccionan los números de la solución en la matriz. Estos números deben ser independientes, es decir, no debe haber dos números en el mismo renglón o columna, tal que la suma de estos números sea la más pequeña de todas las posibles. Puesto que todas las entradas son positivas (o iguales a cero) la suma más pequeña que puede existir es cero. Así, al ser la matriz de nxn, sí se pueden encontrar n *ceros independientes*, tal que no haya o no se tomen en cuenta dos ceros o más en el mismo renglón o columna, se obtendrá una solución a nuestro problema de asignación óptima, es decir, se encontrará una solución óptima.

En nuestro ejemplo anterior, es fácil elegir los números más pequeños (ceros) que cumplen con las condiciones anteriores. Estos números están marcados con asteriscos en la Fig. 2.12.

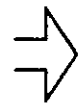
U	y_1	y_2	y_3	y_4
x_1	5	4	1	0*
x_2	4	0*	6	3
x_3	4	0	0*	3
x_4	0*	4	3	6

Fig. 2.12. Ceros independientes, mostrados con asterisco.

Sin embargo, la gráfica inicial puede ser tal que al minimizarla por primera vez no se puedan encontrar n ceros independientes. Ver Fig. 2.13.

U	y_1	y_2	y_3	y_4
x_1	3	4	12	5
x_2	8	6	4	20
x_3	15	4	1	0
x_4	2	7	9	1

a)



U	y_1	y_2	y_3	y_4
x_1	0	0	9	2
x_2	4	1	0	16
x_3	15	3	1	0
x_4	1	5	8	0

b)

Fig. 2.13 a) matriz original y b) matriz modificada una vez que se aplican los pasos **1** y **2**.

La función *inspecciona( )*, que aparece en el apéndice, se encarga de checar si la matriz minimizada tiene n ceros independientes o no.

Esto quiere decir que no siempre se asegurará tener suficientes ceros en esta etapa del algoritmo, de manera de poder elegir n ceros independientes y representar un *matching* perfecto de nuestra gráfica.

En tal caso debemos continuar con el siguiente paso del algoritmo.

4. *Marcar* los renglones y/o columnas que contengan ceros. Lo cual se puede hacer al meter los renglones o columnas que contienen ceros a arreglos bidimensionales o a archivos diferentes. La idea es "cubrir" todos los ceros.

En el apéndice puede verse que estos arreglos son renglón[][] y columna[][]. La forma en que se guardaron (marcaron) fue visitar cada cero, a través de la función *visit_ceros( )*, la cual, y ver cuántos ceros más había en el renglón o columna de ese cero visitado. Si hay más ceros en la columna entonces todos los números de esta columna se guardan en el arreglo columna[][]; y si, por el contrario, existen igual o mayor número de ceros en el renglón, entonces se guarda en al arreglo renglón[][]. Siguiendo con el ejemplo anterior, la Fig. 2.14 muestra cómo quedan cubiertos todos los ceros de la matriz, al guardarlos (marcarlos) en los arreglos correspondientes ya mencionados.

U	y_1	y_2	y_3	y_4
x_1	~~0~~	~~0~~	~~9~~	~~2~~
x_2	~~4~~	~~1~~	~~0~~	~~16~~
x_3	15	3	1	0
x_4	1	5	8	0

Fig. 2.14. Todos los ceros de la matriz deben quedar cubiertos al marcar renglones o columnas completos.

Al marcar de esta manera las columnas y renglones queda una submatriz, a partir de la cual se deben buscar nuevos ceros, sabiendo que las columnas y renglones marcados ya tienen ceros y no debemos "tocarlos", pues de ellos saldrán las aristas que nos lleven a encontrar el *matching* perfecto de peso mínimo, o sea, un *matching* óptimo. Mientras que, de la submatriz debemos encontrar los números tal que la suma sea la menor, y nos lleve a completar el *matching* óptimo.

5. Encontrar el número más pequeño, **min**, que esté en la submatriz, es decir que no esté en ninguno de los renglones o columas marcados.

6.

a) Restar **min** a las columnas y renglones de la submatriz.

b) Sumar **min** a los pesos que están doblemente marcados, es decir, a los que están en guardados en renglón[][] y en columa[][] al mismo tiempo. Este último paso nos permite generar más ceros, con los cuales encontraremos n ceros independientes, después de repetir este paso un número finito de veces si es necesario.

Del ejemplo anterior la matriz nos queda como

U	y_1	y_2	y_3	y_4
x_1	0	0*	9	3
x_2	4	1	0*	16
x_3	14	2	0	0*
x_4	0*	5	8	0

Fig. 2.15. Los ceros independientes se muestran en asterisco.

7. Enseguida *inspecciona( )* checa si se pueden encontrar n ceros independientes. Si es así, el algoritmo termina satisfactoriamente y se llega a un *matching* óptimo. Si no, se va otra vez al paso **4.**

En el ejemplo que estamos analizando, los ceros en asterisco de la Fig. 2.14 son ceros independientes, que corresponden a las posiciones en que se encontraban los números 4,4,0,2 de la matriz original.

8. Se suman los valores que estaban en la matriz original en la posición de los ceros independientes de la última matriz modificada, obteniendo el valor buscado W(M). ☐

En nuestro ejemplo, el peso mínimo da 0 + 2 + 7 + 1 = 10 = W(M). Y el *matching* óptimo es como el de la siguiente figura.

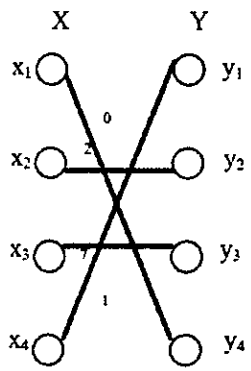


Fig. 2.16. *Matching* óptimo después de encontrar los ceros independientes.

El algoritmo presentado en este capítulo y que soluciona nuestro problema de encontrar el peso mínimo en la transformación de los objetos se conoce como algoritmo Húngaro, y es debido a König y Egerváry, [15] y [16].

La implementación de este algoritmo, como parte del objetivo de este trabajo, se muestra en el apéndice.

2.7 RESULTADOS DE LA PRIMERA IMPLEMENTACIÓN

Para lograr nuestro objetivo del paso 4. en la implementación del algoritmo descrito para encontrar "n ceros independientes", todos los ceros se "etiquetaron" con los números de ceros que había en la columna y en el renglón. Para extraer los ceros independientes, se implementó la función *ext_cers()*, como se muestra en el apéndice. Se visitó cada cero nuevamente en orden de etiqueta menor a mayor, cancelando el resto de ceros que aparecían en renglón y columna en cada visita. En el ejemplo anterior, para extraer los ceros independientes todos los ceros se etiquetaron como se muestra en la siguiente figura:

U	y_1	y_2	y_3	y_4
x_1	0^3	0^2	9	3
x_2	4	1	0^2	16
x_3	14	2	0^3	0^3
x_4	0^3	5	8	0^3

Fig. 2.17. Etiquetación hecha a los ceros de la matriz. A cada cero se le etiquetó con la suma de los ceros que hay en su columna y renglón.

Así, el primer cero visitado es el que se encuentra en la posición (x_1,y_2), pues es el primero que tiene la etiqueta más pequeña, el 2. Por tanto se cancela al cero de la posición (x_1,y_1). Después, se sigue con el cero de la posición (x_1,y_3) cancelando al cero de la posición (x_3,y_3), etc. Al terminar de visitar los ceros no cancelados y cancelar el resto, la matriz queda como en la Fig. 2.18.

U	y_1	y_2	y_3	y_4
x_1	0^2	0^2	9	3
x_2	4	1	0^2	16
x_3	14	2	0^2	0^3
x_4	0^3	5	8	0^2

Fig. 2.18. Se eliminan los ceros que hay en columna-renglón al visitar los ceros con la etiqueta de menor a mayor.

Quedando solamente los ceros que en la Fig. 2.12 se marcaron con asterisco.

Este método de extraer los ceros independientes, construido de una forma empírica, aparentemente es muy bueno para un buen número de matrices, pero falla para matrices en donde hay muchos ceros con la misma etiqueta y se cancelan los ceros de todo un renglón. Al querer hacer más ceros, todos los renglones o columnas se marcan, lo que evita visitar a los ceros. El siguiente ejemplo es un caso típico, de que al usar la implementación realizada hasta este momento, no se llega a encontrar el trabajo mínimo:

U	y_1	y_2	y_3	y_4
x_1	3	8	5	9
x_2	2	1	4	3
x_3	7	10	3	6
x_4	15	8	7	12

U	y_1	y_2	y_3	y_4
x_1	0	5	2	4
x_2	1	0	3	0
x_3	4	7	0	1
x_4	8	1	0	3

U	y_1	y_2	y_3	y_4
x_1	0^1	5	3	4
x_2	1	0^3	4	0^3
x_3	3	6	0^3	0^3
x_4	7	0^3	0^3	2

Fig. 2.19. Se presentan los pasos del algoritmo implementado.

Es claro que en este tipo de matrices, y en general de cualquier matriz, es susceptible de encontrar su peso mínimo. Así que, para lograr obtener el peso mínimo de cualquier matriz, se desechó la idea de etiquetar los ceros como se hizo en la implementación del algoritmo. En vez de ello se optó por construir el llamado *árbol Húngaro* [14] y que permite encontrar los ceros independientes sin necesidad de etiquetar los ceros.

2.8 ÁRBOL HÚNGARO

Una vez que se aplica la función *visit_ceros()*, queda una matriz con posibles ceros independientes, para buscarlos, se procede a construir al llamado árbol húngaro, y posteriormente se busca el llamado *camino húngaro* dado por un *augmenting path.*

1.- Para construir el árbol húngaro se visitan los ceros (con la función *visit_ceros()*) de la matriz y se elige un *matching* cualquiera.

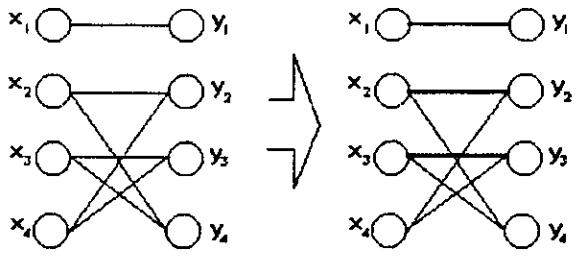
En el último ejemplo, la matriz, a la que ya se le aplicó (dos veces) la función *visit_ceros()* es la siguiente:

U	y_1	y_2	y_3	y_4
x_1	0	5	3	4
x_2	1	0	4	0
x_3	3	6	0	0
x_4	7	0	0	2

Fig. 2.20. Matriz que resulta de minimizar la matriz original, y aplicarle la función *visit_ceros()*.

La Fig. 2.21 a) muestra el *matching* obtenido a partir de los ceros encontrados por la función *visit_ceros()* y la Fig. 2.21.b la elección de un *matching* arbitrario.

2.- Una vez que se ha elegido el *matching* arbitrario, se procede a tomar un vértice x de la gráfica bipartita, obtenida en el paso anterior, que no esté saturado como raíz del árbol húngaro.



a) b)

Fig. 2.21. a) Gráfica bipartita con vértices adyacentes que representan los ceros de la matriz. b) Gráfica con un *matching* arbitrario elegido.

A partir de este vértice raíz se van buscando todos los caminos alternantes posibles:

3.- Se construye un camino alternante con **x** y el primer vértice **y** saturado que se encuentre. Luego, con el siguiente vértice **y** saturado, y así sucesivamente hasta terminar con todas las y´s.

4.- Se repite el paso anterior con los vértices **x**´s para los que las y´s estaban saturadas.

5.- Si alguna **y** no encuentra su respectiva **x** para ser saturada, entonces se ha encontrado un *augmenting path.*

6.- Si hay otra x que no esté saturada (raiz) regresar a 3. De lo contrario se detiene el programa y se construye un nuevo *matching* **M** al usar el *augmenting path*, en el que se intercambian las **x** adyacentes a las **y**´s que no estaban en el *matching* M anterior por un *matching*, y viceversa, se sacan del *matching* anterior la x-y que se encontraban bajo M. De esta manera se ha encontrado un *matching* perfecto, es decir un *matching* óptimo.

7.- Si nunca se encontró el *augmenting path*, regresar al paso 1.

En el ejemplo anterior el árbol húngaro queda como lo muestra la Fig. 2.22. En esta figura se muestra que el camino dado por x_4,y_2 ,x_2,y_4 es el *augmenting path* buscado. En el *augmenting path*, la única arista que estaba en el *matching* M de la gráfica dada por la Fig. 2.21 b) es y_2-x_2; y tanto la arista x_4-y_2 como la arista x_2-y_4 no estaban en M se construye un nuevo *matching* **M** con x_4-y_2, x_2-y_4 y el resto de las aristas que estaban en M. Y puesto que

ya no hay ningún otro vértice x no saturado, el programa termina y se encuentra un *matching* perfecto, que es el óptimo.

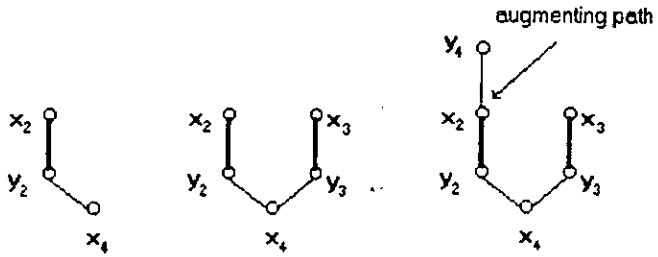


Fig. 2.22. A partir de la raíz dada por x_4 (pues es el primer vértice no saturado encontrado en la gráfica bipartita) se construye el árbol húngaro.

Así, el nuevo *matching* queda como

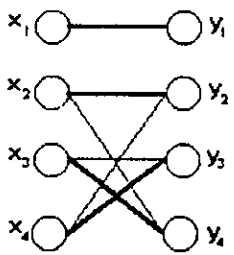


Fig. 2.23. *Matching* final obtenido al aplicar el algoritmo húngaro.

En términos de la matriz inicial se encuentra que los ceros independientes corresponden a los pesos de la gráfica inicial dada por los asteriscos de la siguiente figura.

U	y_1	y_2	y_3	y_4
x_1	0*	5	3	4
x_2	1	0	4	0*
x_3	3	6	0*	0
x_4	7	0*	0	2

U	y_1	y_2	y_3	y_4
x_1	3*	8	5	9
x_2	2	1	4	3*
x_3	7	10	3*	6
x_4	15	8*	7	12

Fig. 2.24. a) Ceros independientes que corresponden al *matching* de la Fig. 2.23 marcados en asterisco, b) valores de los ceros independientes al localizar la posición de la matriz original.

Cuyo peso mínimo da W(M) = 3+3+8+3 = 17.

Adaptando éste método de encontrar los ceros independientes a la implementación se logra encontrar el peso mínimo de la gráfica.

2.9 IMPLEMENTACIÓN DEL ALGORITMO

Para implementar las ideas anteriores en un programa (en C) esto es, construir el árbol húngaro principalmente, es mejor pensar en la gráfica bipartita y sus pesos como una matriz cuyos elementos son los pesos de la gráfica, sus renglones representan al conjunto de elementos o *voxels* del objeto a ser transformado y las columna al conjunto de elementos del objeto con el que se transformará.

La simulación del árbol húngaro se realizó de la siguiente manera:

1.- Se minimiza la matriz. Se encuentra el valor mínimo de cada renglón y columna restándolo a cada elemento (peso) respectivamente para encontrar los ceros de la matriz.

2.- Se realiza un *matching* arbitrario M. Para ello se marca (digamos con una **m**) el primer cero encontrado en cada renglón.

3.- Se visitan los renglones. Aquel que no tenga un cero marcado se considera como la raíz del árbol a construir. Si todos los renglones tienen ceros marcados, se detiene el programa.

4.- Visitar los ceros no marcados del renglón, se busca la marca **m** que hay en la misma columna remarcándola como **m'** y marcando la columna para indicar que ya fue utilizada y visitada. Se hace lo mismo para cada cero que haya en el mismo renglón. Esto simula las ramas del árbol.

5.- Una vez que se obtienen las marcas **m'**, visitar el renglón de cada **m'**. Para cada cero que hay en el renglón de **m'** regresar a 4. Si existe algún cero que no encuentre una **m** en la columna el programa termina y se ha encontrado el *augmenting path*. Se construye un nuevo *matching* M', al intercambiar un cero por una **m'** del *augmenting path*.

6.- Si todos los ceros encuentran marcas repetir 5 hasta que se visiten todas las **m'** (doblemente marcadas).

7.- Si no se encontró el *augmenting path*, regresar a 1.

A continuación se presenta el diagrama de flujo que muestra la implementación del algoritmo en forma general dado en la Fig. 2.25.

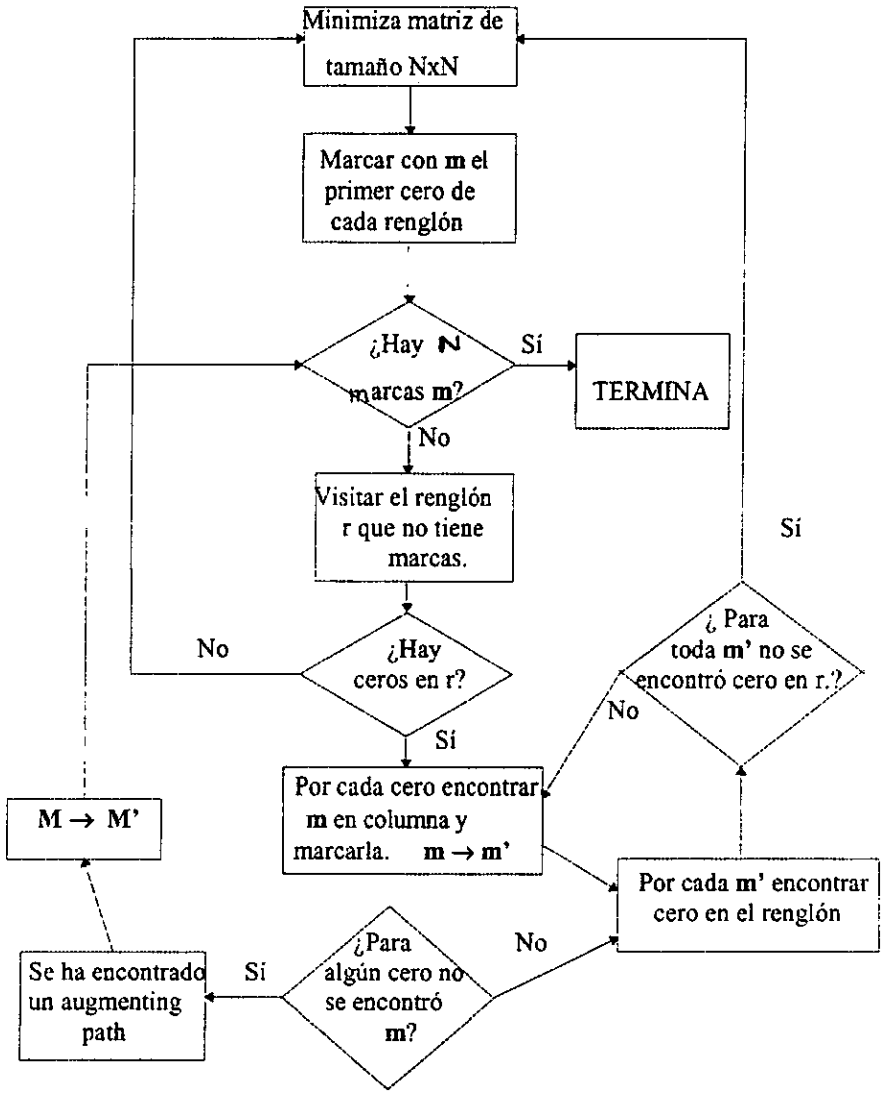


Fig. 2.25. Diagrama de flujo que describe la implementación en términos de matrices, del algoritmo húngaro y de cómo encontrar un *matching* óptimo.

2.10 COMPLEJIDAD DEL ALGORITMO HÚNGARO IMPLEMENTADO.

Como puede verse en la implementación del algoritmo Húngaro que aparece en el Apéndice, casi todas las rutinas requieren un tiempo de orden $O(n^2)$, puesto que se está recorriendo siempre una matriz de tamaño nxn. Sin embargo, como lo muestra la función *visit_ceros()*, cuando se visita cada cero de la matriz (lo cual puede hacerse en un tiempo $O(n^2)$), para ya sea marcar un renglón o una columna, como se explica en el paso 4 de la sección 2.6, se debe calcular donde existen más ceros, el hacer esto por cada cero que aparece en la matriz lleva un tiempo de orden $O(n)$. Por tanto la rutina total se llevará un tiempo de a lo más $O(n^3)$. Esto quiere decir que marcar los renglones de la matriz "cuesta" un orden de magnitud más a la complejidad del algoritmo respecto al elaborado por Bribiesca en [2]. En todas las demás funciones la complejidad será menor. Por lo tanto podemos decir que el algoritmo Húngaro implementado tiene una complejidad de orden $O(n^3)$.

• CAPÍTULO III

TRANSFORMACIÓN DE UN OBJETO A OTRO

Como parte del objetivo de este trabajo, en el Capítulo II se ha encontrado e implementado un algoritmo, el llamado "algoritmo húngaro", que nos permite encontrar el peso mínimo para transformar un objeto en otro. En éste capítulo se relacionarán los pesos de la transformación entre los objetos con el trabajo (en términos de la física clásica) que se realizará para transformar a los objetos. Con ello se definirá la *semejanza* entre los objetos, la cual nos da una medida de qué tan semejantes son dos objetos; se aplicará el algoritmo húngaro a algunos objetos muestra y se dará a conocer el trabajo realizado en las transformaciones así como la semejanza entre los objetos.

Para poder encontrar la semejanza entre dos o más objetos en 3-D, debemos antes que nada, hacerlos invariantes (desde el punto de vista dado por el capítulo I) bajo traslación, rotación y escala.

Existen varios trabajos como los de Bracewell, Brigham y Wechsler [18], [19] y [20], respectivamente, en los que utilizan la transformada de Fourier para producir invarianza en rotación y escala. Sin embargo, como nos hemos basado en el trabajo de Hu usaremos los momentos obtenidos en el capítulo I.

3.1 TRABAJO REQUERIDO PARA TRANSFORMAR A LOS OBJETOS.

De la física clásica el trabajo dW se define como

$$dW = Fds$$

donde F es la fuerza aplicada a un objeto o partícula que tiene un desplazamiento diferencial *ds*. En este trabajo, puesto que se moverán *voxels* (los cuales se pueden considerar como partículas) para hacer las transformaciones entre los objetos, estos tendrán que recorrer una distancia debido a una fuerza aplicada sobre ellos, que por simplicidad consideraremos la constante e igual a uno, para que, de esta manera, el trabajo realizado sea función de la distancia euclidiana al mover un *voxel* de la posición $X(x_1,x_2,x_3)$ a la posición $Y(y_1,y_2,y_3)$. Así, se puede construir un métrica para transformar un objeto A en un objeto B ($W(A \rightarrow B)$) [2]:

$$W(A \rightarrow B) = W(B \rightarrow A),$$

$$W(A \rightarrow B) \geq 0,$$

$$W(A \rightarrow B) = 0 \Leftrightarrow A = B, \text{ y}$$

$$W(A \rightarrow C) \leq W(A \rightarrow B) + W(B \rightarrow C).$$

Para encontrar el trabajo al transformar dos objetos A y B, se procede realizar los siguientes pasos:

1.- Se superponen los dos objetos A y B, esto es, una vez que los objetos son invariantes en rotación, translación, escala, y en ejes mayores, se procede a superponerlos, haciendo que coincidan sus ejes mayores y el centro de masas. Al llevar a cabo la superposición, existen *voxels* que son comunes a los dos objetos, entonces estos *voxels* se quedarán en su posición original, no se moverán. Formalmente, debemos considerar a los conjuntos $I_a(r,c,k)$ e $I_b(r,c,k)$ que representan la imagen binaria 3-D de los cuerpos A y B respectivamente. En dichos conjuntos r se refiere a los renglones de la matriz, dada en forma binaria, que representa tridimensionalmente a los objetos, c representa a las columnas y k a las capas. De esta manera, la superposición de los objetos, se define formalmente como la imagen binaria dada por el conjunto

$$Is(c,r,k) = I_A(r,c,k) \cap I_B(r,c,k)$$

2.- Enseguida se mueven los *voxels* del objeto A que no ocupan la posición respectiva en el objeto B, los *voxels* que se muevan se llaman *voxels positivos* y la posición hacia las que se

moverán se llaman *voxels negativos*. En términos de la imagen binaria 3-D se mueve el conjunto *voxels* dado por

$$I_P(c,r,k) = I_A(r,c,k) \setminus I_B(r,c,k)$$

Así $I_P(c,r,k)$ representa a los *voxels positivos*.

Los *voxels* negativos están dados por

$$I_M(c,r,k) = I_B(r,c,k) \setminus I_A(r,c,k)$$

3.- Se aplica el algoritmo húngaro para mover los *voxels* de manera que el trabajo realizado en ello sea el mínimo.

Cabe hacer notar que si n es el número de *voxels* que contiene un objeto, existen n! formas diferentes de moverlos de I_P a I_N. El método usado en éste trabajo, considera una gráfica G bipartita con bipartición (I_P,I_N) donde $I_P = \{i_{P1},i_{P2},...,i_{Pn}\}$ e $I_N = \{i_{N1},i_{N2},...,i_{Nn}\}$.

Como parte importante de este trabajo, se hace uso del algoritmo húngaro. Con él el trabajo requerido para mover los *voxels* positivos es el mínimo. De la gráfica G se obtiene un *matching* M, el cual se va modificando, como ya se explicó en el capítulo II, hasta encontrar la solución óptima que minimice el trabajo para mover los *voxels positivos*.

3.2 MEDIDA DE SEMEJANZA

Si los objetos son similares o *semejantes*, es claro que prácticamente no se moverán *voxels* en la transformación de los objetos, por lo tanto el trabajo total realizado en la transformación será pequeño. Por otro lado, entre más diferentes sean los objetos, se consumirá mayor cantidad de trabajo y entonces podemos decir que los objetos serán menos semejantes o bien serán más *desemejantes*.

Obviamente al mover los *voxels* de un objeto a otro ellos recorrerán una distancia *d*, de modo que el i-ésimo *voxel* recorrerá una distancia $d(A_i, B_i)$ al moverse del objeto A al objeto B. La suma total de todas las distancias recorridas por los *voxels* a mover (*voxels*

positivos) nos dará la medida de *semejanza*: entre mayor sea la distancia total recorrida por los *voxels* entonces el valor de D dado por la relación (Medida de Semejanza):

$$D = \sum_{ij}^{n} d(A_i, B_j)$$

quiere decir que los objetos serán más *desemejantes*, y si D es pequeño los objetos serán más parecidos o *semejantes*.

Ahora, puesto que el trabajo lo hemos definido de manera que la fuerza la hemos hecho igual a uno, entonces el trabajo total al transformar los objetos coincidirá numéricamente con la desemejanza de los objetos.

3.3 APLICACIÓN DEL ALGORITMO HÚNGARO EN LA TRANSFORMACIÓN DE OBJETOS

Una vez que se han encontrado los invariantes en rotación, translación, volumen, centro de masa y en los ejes mayores, procederemos a transformar a los objetos.

La figuras 3.1, 3.2 y 3.3 muestran la transformación de diferentes objetos A en bjetos B.

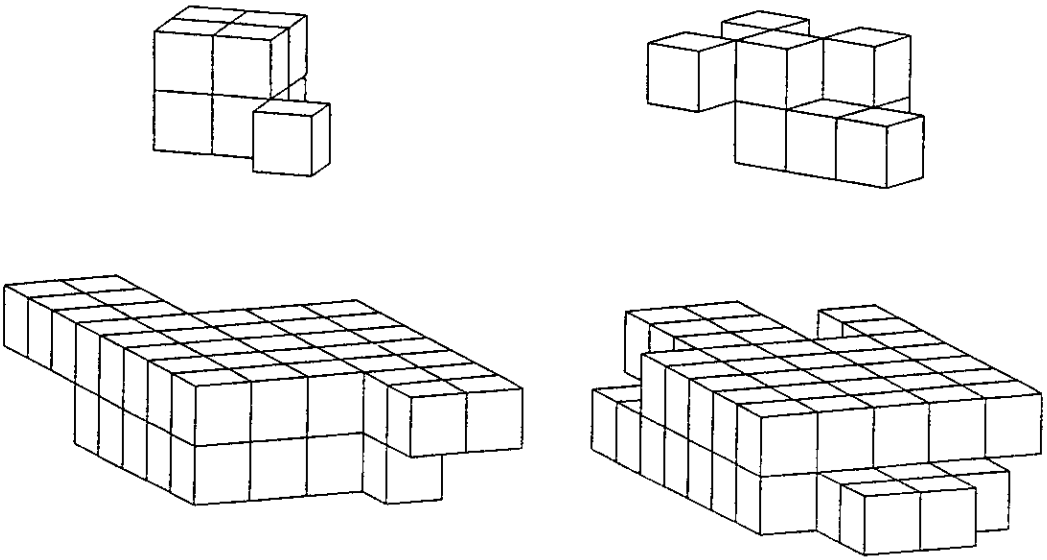


Fig. 3.1. Primera y segunda Transformaciones.

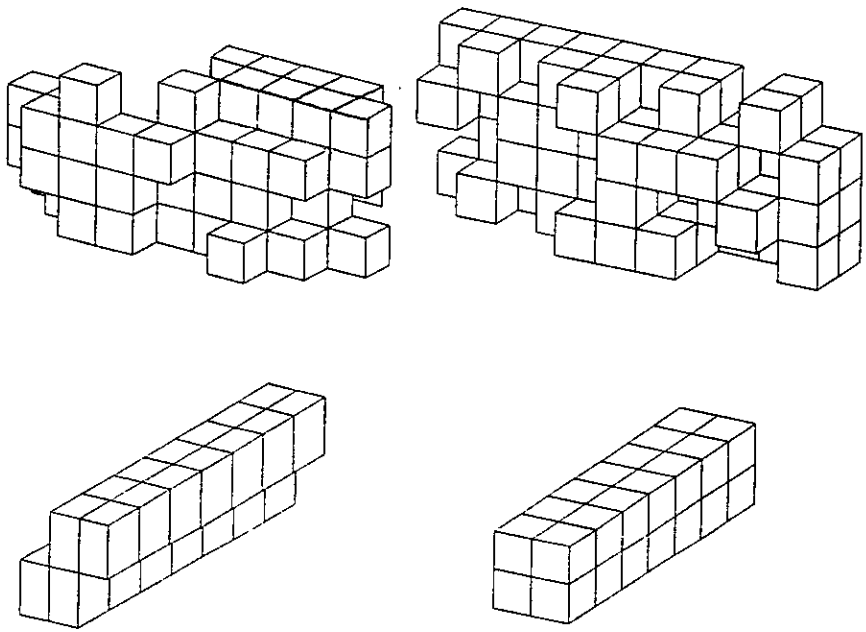


Fig. 3.2. Tercera y cuarta Transformaciones.

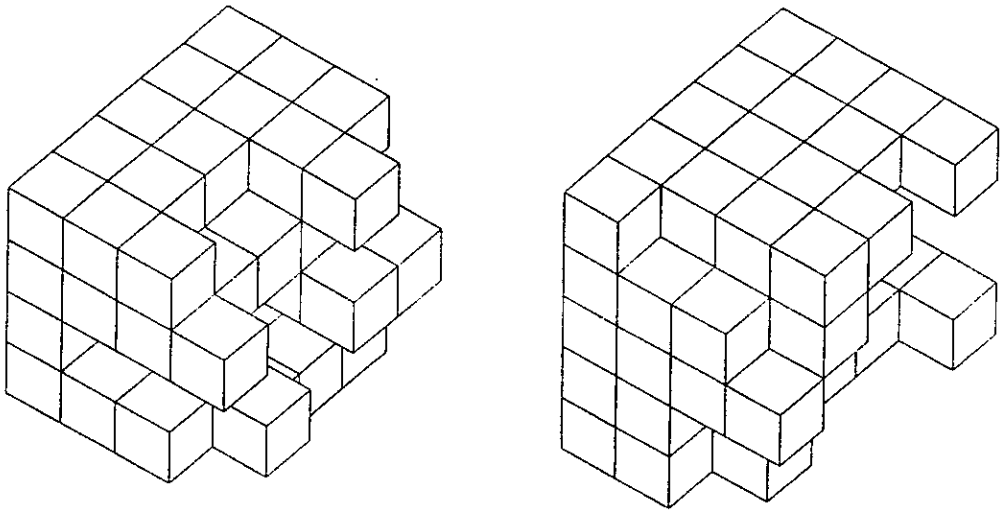


Fig. 3.3. Quinta Transformación

3.4 TRABAJO REALIZADO AL TRANSFORMAR LOS OBJETOS Y MEDIDA DE SEMEJANZA ENTRE OBJETOS

Las siguientes tablas nos dan los resultados de haber realizado las transformaciones entre los objetos, en cada uno de los diferentes casos:

Transformación	*Voxels* comunes	*Voxels* a mover
1	6	3
2	49	11
3	52	20
4	8	20
5	44	20

Tabla 3.1. Número de *voxels* comunes a ambos objetos y *voxels* positivos a mover en cada transformación.

Transformación	Medida de semejanza según Bribiesca	Medida de semejanza según el método húngaro
1	5.58	5.06
2	22.71	22.33
3	41.72	36.61
4	59.95	55.93
5	33.12	30.07

Tabla 3.2. Comparación en la medida de semejanza según el método aplicado para transformar a los objetos.

• CAPÍTULO IV

REPRESENTACIÓN Y FORMACIÓN DE LOS OBJETOS.

En éste capítulo se explica como se obtienen y se definen las imágenes así como algunas técnicas para la representación de los objetos.

4.1 FORMACIÓN DE LA IMAGEN

Para formar una imagen digital es necesario un sensor que registre la radiación reflejada por el objeto físico en cuestión. Con la información registrada es posible construir modelos matemáticos de imágenes. Dichos modelos tienen diferentes componentes:

1.- Una función que representa una abstracción de la imagen.
2.- Un modelo geométrico que describe cómo las tres dimensiones del objeto se proyectan en dos dimensiones.
3.- Un modelo radiométrico que muestra cómo afecta la medida de la radiación al sensor debido a la geometría de la imagen, a las fuentes de luz y a las propiedades de reflectancia.
4.- Un modelo de frecuencia espacial que describe cómo las variaciones espaciales de la imagen pueden caracterizarse en un dominio de transformación.
5.- Un modelo de color que describa las diferentes medidas espectrales y su relación con los colores de la imagen.
6.- Un modelo digitalizado que describe el proceso de obtener muestras simples discretizadas.

4.2 DEFINICIÓN DE LAS IMÁGENES

El primer paso para el reconocimiento de patrones en tres dimensiones es definir una imagen [1]. Las imágenes pueden obtenerse a partir de varias técnicas. Por ejemplo, casi todos los sistemas que utilizan la cámara de televisión convierten la intensidad de la luz reflejada en una señal electrónica que posteriormente es digitalizada. Otros sistemas usan, para tal efecto, otros tipos de radiaciones, como rayos x, luz láser, ultrasonido o calor.

El sistema de visión para detectar la imagen puede ser totalmente pasivo. Tal es el caso, por ejemplo, de tomar como entrada una imagen digitalizada debido a un sensor de microondas o radiación infrarroja, de un escudriñador satelital o una prueba planetaria. Por otro lado puede tratarse de una imagen activa; el sistema automático de imágenes activas puede controlar la resolución y la dirección de sensores o regular y direccionar sus propias fuentes de luz. Estas mismas fuentes de luz pueden tener propiedades especiales para obtener un reconocimiento de la estructura del mundo tridimensional; un ejemplo de ello es el iluminar una escena donde se encuentran los objetos opacos a analizar. El rango de los datos de la escena puede obtenerse por triangulación. Un simple dispositivo de hardware puede transmitir información sobre la reflectividad multiespectral.

Una *imagen generalizada* es un conjunto de entidades relacionadas de la escena. Este conjunto puede incluir los resultados de extraer *imágenes intrínsecas*, donde por imagen intrínseca queremos decir el arreglo de representaciones de cantidades y cualidades físicas importantes. Entre las cantidades están la orientación de la superficie, la inclusión de contornos, la velocidad del objeto, etc. El color del objeto es una cualidad física. Las cualidades físicas son muy importantes y muy útiles pues pueden dar mejor idea de los objetos físicos que los valores de las cantidades, las cuales dan a conocer los valores físicos para reconocer al objeto sólo indirectamente. El tener una imagen intrínseca es una buena forma de entender una escena y representa cálculos computacionales de interés.

La información necesaria para calcular u obtener una imagen intrínseca está en los datos de entrada de la imagen (que es la información obtenida de los sensores de luz, por ejemplo y de otras técnicas a la vez), para después ser procesada.

4.3 REPRESENTACIÓN DE LOS OBJETOS

Existen tres clases generales para representar a los sólidos rígidos [1] :

1.- Por superficie

2.- Usando cilindros generalizados

3.- Por volumen (en general geometría constructiva de sólidos)

Aunque la representación semántica de un sólido es intuitivamente clara, algunas veces es matemáticamente difícil, y cada una de las representaciones tiene propiedades computacionales diferentes.

Una representación de superficie puede describir cómo se ve un objeto; una versión de volumen expresa al sólido como una combinación de subpartes de otros sólidos, y puede no contener información explícita acerca de la superficie. Sin embargo ésta representación puede ser mejor para asignación, siempre y cuando pueda ser estructurada para reflejar partes funcionales.

4.3.1 Representaciones de Superficie:

La superficie envolvente de una figura tridimensional debe especificar al objeto sin ambigüedad. Debido a que las superficies son la parte observable del objeto, estas representaciones son importantes para el reconocimiento de imágenes por medio de la computadora.

En general, una superficie se hace de caras, las cuales a su vez se representan mediante superficies matemáticas y mediante la información acerca de sus propias superficies envolventes, como sucede en el presente trabajo, en el cual se emplearán secciones cúbicas. Bribiesca [2] encontró una manera de eliminar las caras de contacto de los voxels vecinos por medio del concepto de superficies de contacto, para sólidos compuestos por poliedros. Aunque existen otros tipos de representaciones tales como el uso de esferas o superficies armónicas esféricas, así como el uso de cilindros, hexágonos, dodecaedros, etc. en el presente trabajo se utilizarán cubos (hexaedros), por ser los

poliedros más sencillos y menos difíciles de implementar para su visualización en la computadora.

El dibujo de líneas ha sido hasta la fecha otro importante mecanismo para representar las superficies envolventes de los objetos, utilizándose muchas veces como el principal medio de comunicación entre las personas para poder entender algunos de los aspectos cuantitativos de figuras tridimensionales.

El uso de las líneas fue un buen intento inicial en lo concerniente a visión por computadora, por las siguientes razones:

1.- Están relacionadas fuertemente con características superficiales de objetos poliédricos.

2.- Pueden representarse de manera exacta, es decir, el ruido que puede haber afectado a la figura es posible eliminarlo por completo.

Incluso esta técnica de las líneas sirvió para construir un robot [6].

Sin embargo el uso de las líneas es frecuentemente ambiguo [1] y puede requerir de conocimientos incluso físicos para poder interpretarlas. Es decir, el uso de las líneas no es suficiente para hacer representaciones tridimensionales si se quieren reconocer objetos irregulares, como los que pretendemos en este trabajo.

4.3.2 Representaciones Volumétricas:

El primer autor en desarrollar un programa con representación en 3-D fue Roberts [4], en 1963. En el sistema que él desarrolló, se aceptaron imágenes digitalizadas de figuras poliédricas, y se utilizó la técnica del dibujo de las líneas. Éste trabajo dio como consecuencia el desarrollo de imágenes geométricas, la representación de objetos, asignación, la graficación por computadora, etc.

Posteriormente aparecieron varios autores que desarrollaron programas con resultados similares a los de Roberts [7],[8],[9].

Casi todos los objetos del mundo son sólidos, aunque solamente sus superficies sean visibles. Para representar a los objetos es frecuente usar sólidos más primitivos, los cuales pueden tener propiedades fáciles de manejar para así comprender al cuerpo o sólido "real".

Así, para poder reconocer la forma del objeto se utilizará un arreglo de celdas o *voxels* que se supondrán rellenos de material. De esta manera, los volúmenes se representan como arreglos que pueden requerir de mayor capacidad si la resolución es alta, puesto que los requerimientos espaciales incrementan al cubo de resolución lineal.

Los arreglos espaciales son comunes. Es usual convertir una representación exacta en una representación espacial aproximada, ya que se pueden producir con relativa facilidad secciones de cuerpos geométricos. Aunque el arreglo espacial de los *voxels* puede codificarse de tamaños diferentes, en este trabajo se hará codificando todos los *voxels* del mismo tamaño (invariancia en escala).

Con la disminución en el costo de la memoria, el arreglo espacial explícito podría ser más común. El mejoramiento del hardware para computación paralela [3] podría ser una ventaja y apoyar el desarrollo de algoritmos paralelos para calcular propiedades de sólidos bajo estas representaciones. Por ejemplo, las propiedades de masa podrían calcularse a partir de los componentes del cuerpo y luego sumarse. Objetos inhomogéneos o irregulares, como la anatomía humana o como una roca deforme, pueden representarse fácilmente con *voxels*.

• CAPÍTULO V

VOXELIZACIÓN DE LA CUENCA DE MÉXICO

A partir de la obtención de las elevaciones del terreno de la Cuenca de México, en este capítulo se crea una matriz binaria de ceros y unos y se *voxeliza* la Cuenca con el objeto de estraer algunos cerros y medir su semejanza. Se discute porque no fue posible utilizar el algoritmo húngaro implementado en el capítulo II.

Para el reconocimiento de los objetos de la Cuenca de México, tales como rocas, cerros, montañas o volcanes, en el presente trabajo se utilizarán las elevaciones de toda la Cuenca, las cuales fueron proporcionadas por el INEGI [5]. En [5] se presenta un método para obtener y manipular información topográfica al desarrollar un software que permite transformar los datos del INEGI a un formato estándar llamado DXF (Data eXchange File) el cual es compatible con paquetes gráficos como el AUTOCAD. Dicho paquete ha sido utilizado en el presente trabajo para visualizar el mapeo en cubos que se hace de la Cuenca de México y de cualquier objeto que se mencione aquí para su estudio.

Éste paquete se utiliza a lo largo de éste capítulo por ser de uso común, de bajo costo y con pocas exigencias en el hardware.

Una forma de obtener la información de la superficie de un terreno es utilizando curvas de nivel, indicando su altura sobre el nivel del mar.

Otra técnica, la cual se utiliza aquí, es utilizando una representación dada por el Modelo Digital de Terreno (MDT). Este modelo consiste de una red hecha de cuadrículas, en la que cada cruce de las líneas representa la altura al nivel del mar. De esta manera, toda la información está digitalizada y presentada en un grado de longitud por un grado de latitud, que, a la altura de la Cuenca de México, un segundo de arco es de aproximadamente 29.1 metros, lo que hace que toda la Cuenca cubra aproximadamente 11,000km^2.

Para obtener el modelo tridimensional se puede utilizar un par de fotografías estereoscópicas, o sea, dos fotografías del mismo objeto observado desde dos lugares diferentes en el espacio.

5.1. ALGUNAS CARACTERÍSTICAS Y VENTAJAS DEL AUTOCAD

- Despliegue en pantalla con diferentes resoluciones.
- Diferentes vistas del MDT y selección de ventanas.
- Acercamiento y alejamiento de áreas.
- Pueden obtenerse diferentes perspectivas del objeto desde cualquier dirección, ya sea desde afuera o desde dentro de él.
- Diferentes graduaciones de lente de cámara en la perspectiva.
- Se pueden ocultar superficies o líneas internas, tales como aristas.
- Generación de diapositivas para presentaciones o animación
- Diferentes tipos de letras o fuentes.
- Flexibilidad en la edición para la incorporación de otros elementos gráficos al dibujo.
- Facilidades de programación en AutoLISP.
- Incorporación de otros archivos gráficos al MDT.
- Susceptible de poderse leer coordenadas, distancias ángulos y áreas dentro del dibujo.
- Salidas por impresoras, pantallas y graficadores.

5.2 GENERACIÓN DEL ARCHIVO DXF

Se escoge un objeto (un cerro, por ejemplo) representado en el MDT. El formato DXF es un formato estándar de transferencia de información, una vez obtenido este formato se utiliza el paquete AUTOCAD para desplegar la información gráfica.

Asi, el elemento u objeto es apropiado para la representación del MDT, pues almacena una matriz de hasta 256x256 elementos, o sea, queda un elemento superficial compuesto por planos. Para algunas aplicaciones este elemento, que en realidad es una superficie, puede transformarse en planos independientes definidos por elementos independientes.

Para incorporar el MDT al archivo DXF se utiliza el lenguaje "C", lo que produce un archivo DXF vacío sin información geográfica. Después se procede a grabar el elemento gráfico 3D con formato ya definido para ese elemento en el archivo DXF.

La parte de información del MDT leída y transformada corresponde a la ventana de información que ha sido seleccionada por el usuario. Esta ventana se lee en un mapa topográfico a escala de 1:250,000 o 1:50,000, indicando la (x_{min}, y_{min}) y (x_{max}, y_{max}) en coordenadas llamadas UTM, que no son más que coordenadas cartesianas. Existe una variable que permite seleccionar uniformemente los incrementos entre puntos en x y y, a múltiplos del equivalente a 3 segundos.

Una vez que se ha incorporado el MDT al archivo DXF, el AUTOCAD lo invoca con la instrucción DXFIN para luego formar un dibujo DWG.

5.3 CREACIÓN DE UNA MATRIZ BINARIA

Una vez conseguidas las alturas de la Cuenca (todas dadas en metros y concentradas en un archivo MDT) se procedió a construir un programa (en C) que *voxeliza* el Valle de México. Para ello

a) Se calculó un valor máximo y un valor mínimo que corresponden a la máxima y mínima elevación de la Cuenca de México respectivamente, al abrir el archivo MDE y leer todas las alturas.

b) Se construyó una matriz binaria, con un programa que tenía como

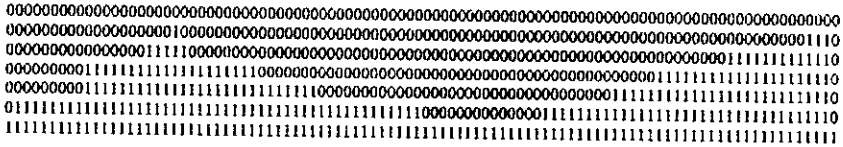
entrada: un archivo que contiene las alturas de la Cuenca de México dadas en metros (el archivo MDT).

salida: una matriz binaria, en la que los 1´s representan la parte sólida de la Cuenca y los 0´s la parte de la superficie hasta el valor máximo, con lo cual se generó un archivo 3DI.

Este último archivo será de gran utilidad para el presente trabajo, pues con éste se harán las transformaciones del objeto que deseemos.

La idea es manejar archivos de 0's y 1's para cada uno de los objetos, pues esto nos permitirá leerlos cuando queramos y de esta manera tener la representación del objeto y aplicarle transformaciones de rotación, traslación, etc.

a)



b)

Fig. 5.1 Representación de una capa o corte transversal de la Cuenca de México en archivo a) MDE. b) 3DI.

c) Con un programa se llevó a cabo la voxelización final:
entrada: el archivo que contiene la matriz binaria (el archivo 3DI).
salida: el archivo (cuya extensión es .DXF) que puede visualizarse en AUTOCAD.

Posteriormente veremos que los archivos DXF pueden representar un mismo objeto pero con transformaciones aplicadas a él.

La Fig. 5.2 ilustra, en síntesis, lo que se pretende realizar con los diferentes programas que realicemos en C, al aplicarlos sobre estos archivos que ya hemos mencionado.

La Fig. 5.3 muestra un despliegue de la Cuenca de México extraído del archivo MDT. La Fig. 5.4 muestra la misma vista de Valle pero ya *voxelizado*. Y la Fig. 5.5 muestra una parte de la Cuenca al extraer un cuerpo de él, primero tomando los datos del archivo MDT y después *voxelizando* como ya se explicó y como se ha ilustrado en la Fig. 5.2.

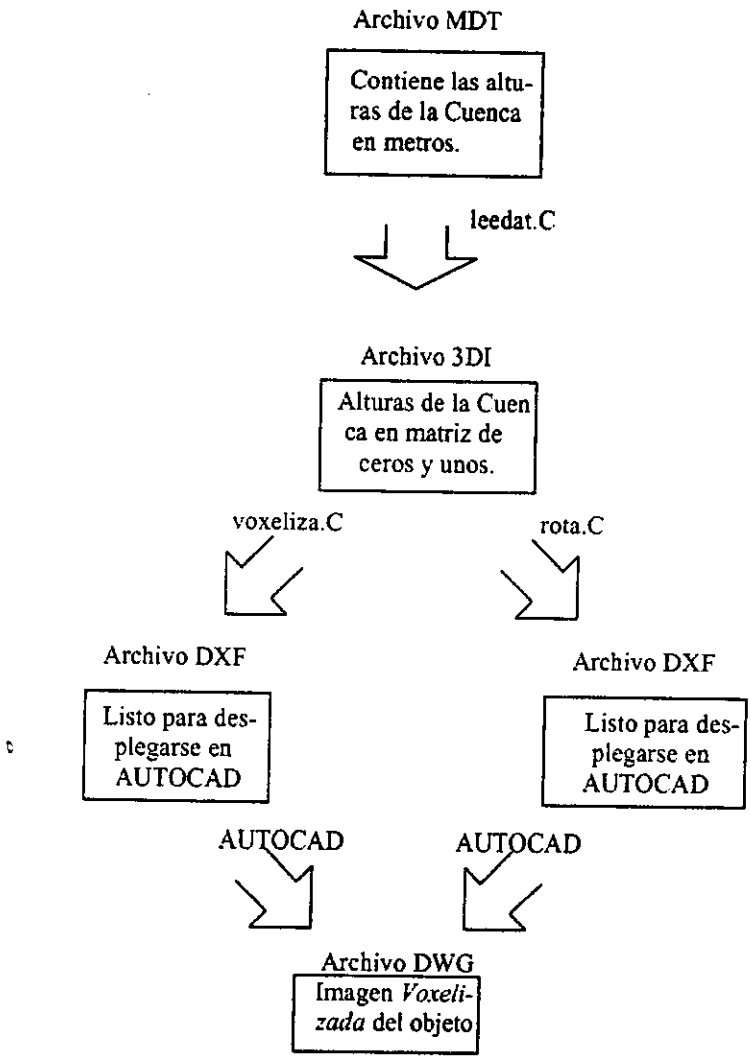


Fig. 5.2. Esquema que muestra las diferentes etapas para llegar a la imagen de un objeto *voxelizado*, comenzando por los datos contenidos en un archivo MDE.

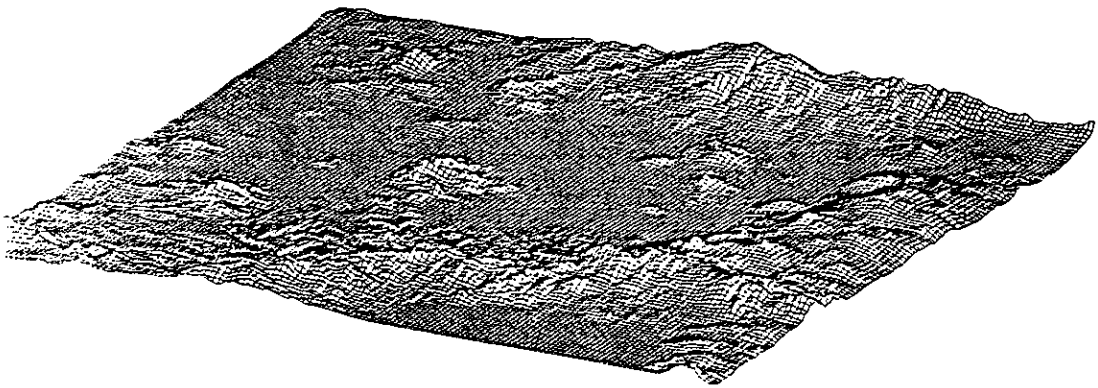


Fig. 5.3. Vista de la Cuenca de México al usar el archivo MDT.

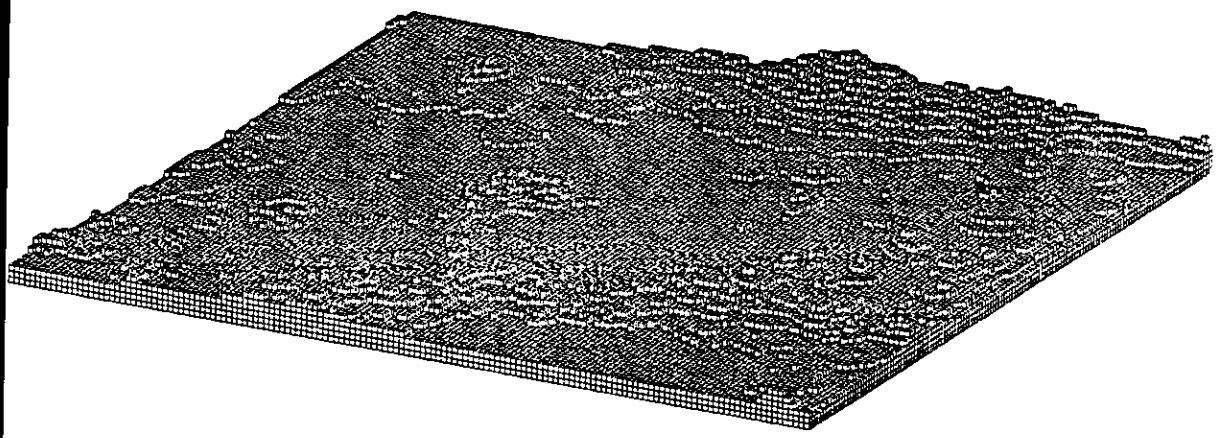


Fig. 5.4. Vista de la Cuenca de México ya *voxelizado*.

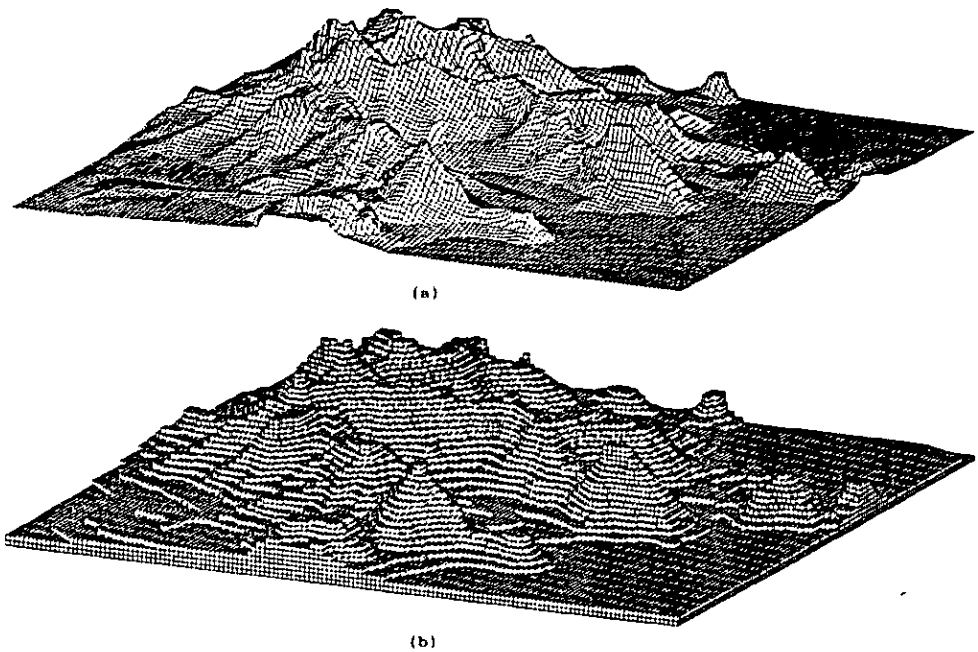


Fig. 5.5. Vistas de cuerpos extraídos de la Cuenca de México. a) Tomados del archivo MDT. b) Ya *voxelizados*.

5.4 DISCUSIÓN

Un estudio de la forma de como trabaja el algoritmo implementado, permite responder a la pregunta de por qué no se pudo aplicar el método a transformaciones en las que el número de *voxels* a mover excede a 20, como fue el caso de los cerros de la Cuenca de México. Se ha llegado a los siguientes resultados y razones:

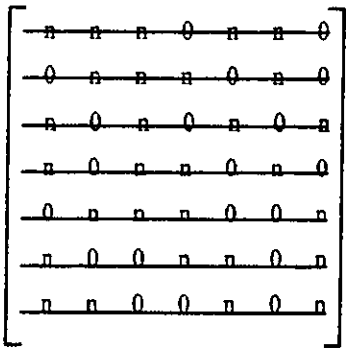
- El método planteado e implementado para el reconocimiento de objetos es consistente con la teoría y los métodos propuestos por la literatura, tal como puede verse en [14] y [41].
- No se tiene ningún problema con los *invariantes* de translación, rotación y volumen, pues la aplicación de estos puede hacerse con un número relativamente grande de voxels (más de 3000).

El problema aparece cuando se aplica el algoritmo húngaro. En una parte del algoritmo, Gould [41] propone marcar los renglones y columnas necesarios para cubrir todos los ceros, tal como se explicó en el capítulo II, al aplicar la función *visit_ceros()*. Sin embargo, aunque el ejemplo que da en su libro y en muchos otros ejemplos encontrados al aplicar su método directamente, da buenos resultados y se logra hacer una transformación sin problemas, cuando se tratan matrices de más de 20 X 20, que representen el número de voxels a mover, se encuentran ejemplos en los que el algoritmo húngaro implementado ya no funciona.. Ello sucede cuando, en el transcurso de la búsqueda de los ceros independientes, se llega a una etapa de la matriz como la presentada en la siguiente figura:

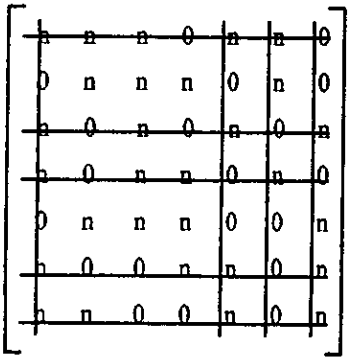
$$\begin{bmatrix} n & n & n & 0 & n & n & 0 \\ 0 & n & n & n & 0 & n & 0 \\ n & 0 & n & 0 & n & 0 & n \\ n & 0 & n & n & 0 & n & 0 \\ 0 & n & n & n & 0 & 0 & n \\ n & 0 & 0 & n & n & 0 & n \\ n & n & 0 & 0 & n & 0 & n \end{bmatrix}$$

Donde n es cualquier número.

A través del método empleado aquí para encontrar los ceros independientes, usando las ideas de Gould [41], los renglones y columnas a marcar quedan como se muestra a continuación:



Vemos que, aunque realmente sí se cubren todos los ceros, también se cubren todos los renglones. Sin embargo, al hacerlo personalmente usando un cierto criterio, uno cubriría los ceros de la siguiente manera:



Esta última forma de marcar los renglones y columnas para cubrir todos ceros, a diferencia de la forma anterior, permite seguir haciendo ceros que nos lleve a encontrar un matching óptimo.

Al continuar con el algoritmo, ya se vio en el capítulo II que el árbol húngaro, necesario para encontrar los ceros independientes, es simulado perfectamente por la implementación realizada en lenguaje C.

• CONCLUSIONES

Durante éste trabajo se lograron los siguientes puntos:

- Se vio que son suficientes y adecuados sólo 4 invariantes en la aplicación a objetos tridimensionales a diferencia de los siete que deduce Hu[11]. Con ello se tiene la primera etapa en el reconocimiento de figura.
- Enseguida se implementó el algoritmo húngaro, lo que nos permitió hacer la transformación entre los objetos.
- Se aplicó el algoritmo húngaro a algunos ejemplos (los cuales aparecen las figuras 3.1,3.2 y 3.3) que fueron diseñados para dar una apariencia de "objetos irregulares".
- Se *voxelizó* la Cuenca de México intentándose aplicar el algoritmo húngaro implementado.

Recordemos, del capítulo I, que una Base Invariante contiene a los invariantes más una transformación admisible que mapee un objeto a otro. Estas dos herramientas nos permiten decir qué tan semejantes en forma son dos objetos. De esta manera tenemos completada la teoría:

Con teoría de INVARIANTES y con ALGORITMO HÚNGARO podemos encontrar SEMEJANZA ENTRE OBJETOS.

- Se ha resuelto un problema que Bribiesca plantea en [2] para optimizar una medida de similitud o semejanza de figuras tridimensionales. En realidad tal reconocimiento se hizo para objetos pequeños, como los planteados en el capítulo IV.
- En el capítulo IV, se muestra que el trabajo realizado en la transformación de los objetos a través del método empleado en este trabajo, siempre es menor que el utilizado por Bribiesca en [2].
- Hasta ahora, el algoritmo implementado puede resolver en forma óptima casos de reconocimiento en donde se deba mover hasta 20 voxels. A través de nuestra implementación no se encontró una transformación en la que se moviesen más de 20

voxels debido a la discusión dada en el capítulo V. Con ello se concluye que el método planteado por Gould [41] no es consistente, y que el problema de elaborar un algoritmo que resuelva encontrar un matching óptimo para encontrar el trabajo mínimo sigue abierto.

- Un trabajo posterior debe atacar el problema de marcar correctamente los ceros de la matriz. Tal trabajo debe encontrar un algoritmo que "sepa decidir" cómo marcar todos los ceros sin llegar a marcar todos los renglones. O bien seguir con otra línea de investigación (ver Freeman [38]) basándose en los llamados *octrices*; es decir, mapear nuestros objetos con sólo ocho voxels y transformarlos para medir la semejanza. Si ésta es grande (es decir, si el trabajo en transformarlos es pequeño), llevar los objetos a una resolución mayor de 64 voxels, y hacer los mismo para ver si la semejanza sigue siendo grande, etc. (la resolución irá como 8^t, donde t es el número de veces que se hace la resolución), con lo que se obtendrá un valor que tienda a cierto trabajo realizado y decir qué tan semejantes son los objetos. De lo contrario, si desde el principio se muestra que la semejanza es muy pequeña, tal vez no se tenga que llevar a cabo mayor resolución, pues desde ese momento se podría decir que los objetos son "muy diferentes".

• APÉNDICE

Se presentan a continuación, algunas expresiones usadas en las tesis que aparecen en letra cursiva y subrayadas.

Figura:

Una *figura* F o una *subimagen* F en una imagen continua I o digital es cualquier función F cuyo dominio es algún subconjunto A de un conjunto de coordenadas espaciales, cuyo rango es el conjunto G de intensidades de imagen, y que esta definida por $F(r,c) = I(r,c)$ para alguna $(r,c) \in A$

Flujo:

Un *flujo* en una red R es un mapeo del conjunto de aristas E a los números reales tal que:

1. (Limitación en la Capacidad) $f(e) \leq c(e)$ para toda $e \in E$
2. (Conservación de flujo) para cada vértice v diferente de s y t en R,

$$\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) = 0.$$

Aquí, $in(v)$ y $out(v)$ denotan el conjunto de aristas que entran o salen del vértice v, respectivamente.

Imagen:

Es una representación espacial de un objeto, de una escena bidimensional o tridimensional, o de otra imagen. En términos de la óptica [17], esta imagen puede ser real o virtual. Matemáticamente puede verse como una función continua I de dos variables definida en una región en el plano, usualmente rectangular. El valor de la imagen obtenida

por las coordenadas espaciales *(r,c)* se denota por *I(r,c)*. Para los sensores ópticos o fotográficos, *I(r,c)* es proporcional a la energía radiante recibida en la banda electromagnética en la que el sensor es sensible.

Invariancia en centro de masa:

Considerando que la posición de cada *voxel* está dada con las coordenadas (renglón, columna, capa) o (r,c,k), el centro de masa de un objeto se calculará a través de las siguientes relaciones [22]:

$$\mathbf{r} = \frac{1}{\#V} \Sigma r, \qquad \mathbf{c} = \frac{1}{\#V} \Sigma c, \qquad \mathbf{k} = \frac{1}{\#V} \Sigma k$$

Invariancia en rotación:

La rotación de un objeto, que supondremos sólido y rígido, es una transformación en $\Re^3$, en la que un *voxel* cualquiera se mueve en un círculo cuyo centro está sobre una línea recta, llamada eje de rotación [23]. Para hacer la rotación de los objetos, existen muchos métodos, uno de ellos es usar una representación por medio de cuaternios, lo que simplifica problemas de orientación absoluta. Tal método se puede encontrar en [2], [24], [25], [26], [27], [28].

Invariancia en translación:

La translación de un objeto implica una transformación lineal en $\Re^3$. Para ello se mueven todos los *voxels* del objeto a transformar, en la misma dirección y con la misma distancia [21], esto es, si los *voxels positivos* $v_1, v_2, \ldots, v_n$ se mueven hacia las posiciones de los *voxels negativos* $u_1, u_2, \ldots, u_n$ respectivamente, entonces las distancias v_1v_2 y u_1u_2 son iguales, lo mismo que v_2v_3 y u_2u_3, etc., de manera que se cumplen las siguientes igualdades:
$v_1v_2 = u_1u_2$, $v_2v_3 = u_2u_3$, ... , $v_{n-1}v_n = u_{n-1}u_n$

Objeto:

Se consideran como sólidos de densidad constante y es representado por un arreglo tridimensional de unos y ceros.

Red:

Una *red R* es una quíntupla *R = (V, E, s, t, c)* donde *(V, E)* es una gráfica dirigida con vértices distinguidos *s* y *t* llamados *fuente* y *origen*, respectivamente. A cada arista *e* de *R* se le asigna un número real no negativo c(e), llamado la *capacidad* de la arista.

Región:

Una *región R* de una imagen es un subconjunto de celdas de resolución en el dominio espacial de la imagen.

Voxel:

Es un elemento pequeño de volumen, el cual se representa con un par ordenado formado de las siguientes partes: la primera es una triada (renglón, columna, capa) que permite representar un volumen rectangular del tipo de un paralelepípedo, y la segunda es el vector de propiedades de dicho volumen.

```
/*
Primera Implementación del Algoritmo Húngaro


Este programa (en lenguaje C y basado en Gould [41]) encuentra
el peso mínimo de una matriz y el trabajo realizado al transformar
dos objetos. Se construye un algoritmo para encontrar los ceros
independientes de una matriz .

*/
#include<stdio.h>
#define TAM 11  // Tamaño de la matriz
int mat_orig[TAM][TAM]; // Matriz original
int mat[TAM][TAM]; // Matriz modificada
void main()
{


void minimiza(int matriz[TAM][TAM]);
void visit_ceros(int matriz[TAM][TAM]);
void ext_cers(int matriz[TAM][TAM]); // Se extraen los ceros independientes
int inspecciona(int matriz[TAM][TAM],int mataux[TAM][TAM]);
int insp;
int item, i,j;
int var;
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
  {
  mat_orig[i][j] = 0;
  mat[i][j] = 0;
  }// for


 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
  {
  printf("Dame renglón %d, columna %d ",i,j);
  scanf("%d",&item);
  mat_orig[i][j] = item; // Primero guarda los valores en la matriz original
  mat[i][j] = item; // Al mismo tiempo los guarda en la matriz que sera modi-
                    // ficada
  }// for

 printf("La matriz que me diste es:\n");
 for(i = 0; i < TAM; i++)
 {
  for(j = 0; j < TAM; j++)
  printf(" %d ",mat_orig[i][j]);
  printf("\n");
 } // for

 minimiza(mat); //encuentra el valor mínimo en renglón y columna, y resta

printf("La matriz  minimizada es:\n");
```

```
  for(i = 0; i < TAM; i++)
  {
   for(j = 0; j < TAM; j++)
   printf(" %d ",mat[i][j]);
   printf("\n");
   } // for
   insp = 0;
   while(insp == 0){
  // Ahora inspecciona si los ceros que quedaron son independientes:
   insp = inspecciona(mat,mat_orig);
   printf("\nEl valor de inspecciona() es: %d",insp);

   if(insp == 0) { // si no hay ceros independientes,entonces
    visit_ceros(mat); // visita cada cero y produce mas ceros
    ext_cers(mat); // anula ceros para que queden solo ceros independientes
   } // if
   if(insp == 1)
   scanf("%d",&var);
   } // while
  } //main

/*-------------------------------------------------------------------------------
 void minimiza(int mat[TAM][TAM])
  {
 int i,j,k=0;
 int min;

// Busca el menor de cada renglón y lo resta

 for(i = 0; i < TAM; i++) {
 min = mat[i][k];
  for(k = 0; k < TAM; k++) {
   if(mat[i][k] < min)
     min = mat[i][k];
   }
   for(j = 0; j < TAM; j++)
    mat[i][j] = mat[i][j] - min;
  k = 0;
  } // for

// busca el menor de cada columna y lo resta

 for(j = 0; j < TAM; j++) {
 min = mat[k][j];
  for(k = 0; k < TAM; k++) {
   if(mat[k][j] < min)
     min = mat[k][j];
   }
   for(i = 0; i < TAM; i++)
    mat[i][j] = mat[i][j] - min;
  k = 0;
  } // for
 } // minimiza
/*-------------------------------------------------------------
```

```
 La función visit_ceros()
 - visita cada cero de la matriz.
 - recorre el renglón y la columna de cada cero.
 - "Marca" las columnas y renglones necesarias para cubrir a todos los
     ceros.
 ----------------------------------------------------------------------*/
 void visit_ceros(int mat[TAM][TAM])
 {
 int renglon[TAM][TAM], columna[TAM][TAM];
 int ind_i[TAM], ind_j[TAM];
 int i,j,r,c,k;

// "Relleno los arreglos, renglón y columna, de -1's
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
    renglon[i][j] = -1;

 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
    columna[i][j] = -1;

 int recorre_ren(int mat[TAM][TAM],int var); // función que da el numero de
                                             // ceros en el renglón
 int recorre_col(int mat[TAM][TAM],int var); // función que da el numero de
                                             // ceros en la columna
 int busca_en_col(int mat[TAM][TAM], int array[TAM]); // Busca si los elemen-
                                          // tos de la columna estan "marcados"
 void sum_max_min(int renglon[TAM][TAM],int columna[TAM][TAM],int mat[TAM][TAM]);
 int busca_en_ren(int mat[TAM][TAM]);
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
  {
   if(mat[i][j] == 0)
    {
//   if (busca_en_col(mat[i][j]) = 0 && busca_en_ren(mat[i][j]) = 0)
    r = recorre_ren(mat,i);
    printf("\nEl n£mero de ceros en el renglón %d es: %d",i,r);
//    printf("\nEl valor de j es: %d",j);
    c = recorre_col(mat,j);
    printf("\nEl n£mero de ceros en la columna %d es: %d",j,c);
  // Si hay m s ceros en el renglón que en la columna, "marca" un renglón me-
  // ti ndolo a un arreglo:
   if (r > c && r > 1)
    {
     for(k = 0; k < TAM; k++)
      renglon[i][k] = mat[i][k]; // "Marca" todo el renglón i
                                 // lleva en la cuenta el n£mero de renglones marcados
     printf(" El renglón %d marcado es ",i);
     for(k = 0; k < TAM; k++)
     printf(" %d ",renglon[i][k]);
   } // if
// Si hay m s ceros en la columna que en el renglón, "marca" una columna me-
// ti ndola a un arreglo:
  if ((r < c) && (c > 1))
```

```
    {
      for(k = 0; k < TAM; k++)
      if (columna[j][k] != mat[k][j]) // Si no hab¡a sido marcada,
       columna[j][k] = mat[k][j]; // "Marca" toda la columna j.
    printf(" La columna %d marcada es ",j);
    for(k = 0; k < TAM; k++)
     printf(" %d ",columna[j][k]);
     } // if
     .
    if (r == c) // Si existe el mismo n£mero de ceros en renglón y columna
     {
       for(k = 0; k < TAM; k++)
       if (renglon[i][k] != mat[i][k]) // Si no hab¡a sido marcado,
             renglon[i][k] = mat[i][k];   // "marca" todo el renglón i
    printf(" el renglón %d marcado es ",i);
    for(k = 0; k < TAM; k++)
     printf(" %d ",renglon[i][k]);
     }
    } // if(mat(i,j) = 0)
   } // for (j)
   sum_max_min(renglon,columna,mat);
  } // visit_ceros()

//-------------------------------------------------------------------------
 void sum_max_min(int renglon[TAM][TAM],int columna[TAM][TAM],int mat[TAM][TAM])
 {
 int i,j,min;
 int bc,br;
 int busca_en_ren(int mat[TAM][TAM],int n,int m,int ren[TAM][TAM]);
 int busca_en_col(int mat[TAM][TAM],int n,int m,int col[TAM][TAM]);
 min = mat[0][0];
  for(i = 0; i < TAM; i++)
   for(j = 0; j < TAM; j++)
if(busca_en_col(mat,i,j,columna) == 0) && (busca_en_ren(mat,i,j,renglon) == 0))
    if(mat[i][j] < min)
          min = mat[i][j];

  for(i = 0; i < TAM; i++)
   for(j = 0; j < TAM; j++){
if(busca_en_col(mat,i,j,columna) == 0) && (busca_en_ren(mat,i,j,renglon) == 0) )
    mat[i][j] = mat[i][j] - min; // resta el minimo a los elementos que no se
                                 // encuentran en ningun renglón "marcado"
//   printf("El m¡nimo es: %d",min);
if(busca_en_col(mat,i,j,columna) == 1) && (busca_en_ren(mat,i,j,renglon) == 1))
    mat[i][j] = mat[i][j] + min; // suma el minimo a los elementos que se
                                 // encuentran doblemente "marcados"
 }
} // sum_max_min

//-------------------------------------------------------------------------
 int busca_en_col(int mat[TAM][TAM],int i,int j,int columna[TAM][TAM])
 {
 int k;
   printf("\nLa columna %d marcada es ",j);
```

```
  for(k = 0; k < TAM; k++)
   printf(" %d ",columna[j][k]);

 for(k = 0; k < TAM; k++)
  if(mat[i][j] == columna[j][k])
    return 1;
 return 0;

} // busca_en_col()

int busca_en_ren(int mat[TAM][TAM],int i,int j,int renglon[TAM][TAM])
{
 int k;
 printf("\nEll renglón %d marcado es ",i);
  for(k = 0; k < TAM; k++)
   printf(" %d ",renglon[i][k]);

 for(k = 0; k < TAM; k++)
  if(mat[i][j] == renglon[i][k])
    return 1;
 return 0;

} // busca_en_ren()

//-------------------------------------------------------------------------
 void ext_cers(int mat[TAM][TAM])
{
int recorre_ren(int matriz[TAM][TAM],int var);
int recorre_col(int matriz[TAM][TAM],int var);
int i,j,p,q;
int c,r,cont;
int num_cers[TAM][TAM];
int ind_i[TAM],ind_j[TAM];
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
   num_cers[TAM][TAM] = -1;

/* El siguiente ciclo es para recorrer la matriz (con "suficientes" ceros) y
"etiquetar" cada "cero candidato" con el n£mero de ceros que tiene cada can-
didato con la columna-renglón                                          */
 printf("\nN£mero de ceros en cada columna-renglón:");
 for(i = 0; i < TAM; i++)
 for(j = 0; j < TAM; j++)
 {
  if(mat[i][j] == 0)
  {
  r = recorre_ren(mat,i);
  c = recorre_col(mat,j);
  num_cers[i][j] = r+c-1;
  printf("\n%d",num_cers[i][j]);
 } // if (mat(i,j) == 0)
} // for (j)  Etiquetación completada

cont = 1;
```

```
  while(cont < 2*TAM - 1)
  {
   for(i = 0; i < TAM; i++)
   for(j = 0; j < TAM; j++) {
    if(num_cers[i][j] == cont)
    {
    for(q = 0; q < TAM; q++) // recorre renglón del 0 candidato
    {
         if((q != j) && (mat[i][q] == 0) && (mat[i][j] == 0)){
         printf("Los valores de q y j son: %d y %d",q,j);
          mat[i][q] = -1; // cancela ceros en renglón
         }
         } // for
    for(p = 0; p < TAM; p++) // recorre renglón del 0 candidato
         if((p != i) && (mat[p][j] == 0) && (mat[i][j] == 0))
         mat[p][j] = -1; // cancela ceros en columna
    } // if(num_cers[i][j] == cont)
   } // for (j)
  cont++;
  } // while(cont < 2*TAM - 1
  printf("\nLa matriz con -1's es:\n");
  for(i = 0; i < TAM; i++){
   for(j = 0; j < TAM; j++)
    printf("%d ",mat[i][j]);
    printf("\n");
    }
} // ext_cers(mat);

//-------------------------------------------------------------------------
int inspecciona(int mat[TAM][TAM],int mat_orig[TAM][TAM])
{
int recorre_ren(int mat[TAM][TAM],int var);
int recorre_col(int mat[TAM][TAM],int var);
int W=0,cont=0;
int i,j,r,c,p,q;

/*int ind_i[TAM],ind_j[TAM]
 for(i = 0; i < TAM; i++){  ind_i[i] = -1;  ind_j[i] = -1; }
 */
 for(i = 0; i < TAM; i++){
 for(j = 0; j < TAM; j++)
 {
 if(mat[i][j] == 0)
 {
 r = recorre_ren(mat,i);
 c = recorre_col(mat,j);
 if ((r == 1) && (c == 1))  // Si solo hay un cero en la columna-renglón
 // se trata de un cero independiente:
 {
 cont++; // contador que lleva el n£mero de ceros independientes
 printf("\nLa posición [%d,%d] contiene el n£mero %d",i,j,mat[i][j]);
 printf("\nEl n£mero %d es un cero independiente",mat_orig[i][j]);
  W = W + mat_orig[i][j];
       if(cont == TAM) // Todos los ceros son independientes
```

```
    {
        printf("\nEl peso m;nimo es: %d\n",W);
        return 1;
    } // if(cont == TAM)
    } // if(r,c)
  } // if(mat(i,j) = 0)
 } // for (j)
 } // for (i)
  return 0;
} // inspecciona()

//-----------------------------------------------------------------------------
 int recorre_ren(int mat[TAM][TAM],int iaux)
 {
 int cont=0,k;
  for(k = 0; k < TAM; k++){
   if (mat[iaux][k] == 0)
    cont++;     // Cuenta el n£mero de ceros en el renglón
  }
  return cont;
} // recorre_ren();

int recorre_col(int mat[TAM][TAM], int jaux)
{
 int cont=0,k;
  for(k = 0; k < TAM; k++){
   if (mat[k][jaux] == 0)
    cont++;   // Cuenta el n£mero de ceros en la columna
    }
  return cont;

} // recorre_col();
```

```
/* Segunda implementación del algoritmo usando el árbol Húngaro.

 Este programa (en lenguaje C y basado en los algoritmos dados en [14]
y en [41]) encuentra el peso mínimo de una matriz y el trabajo realizado
al transformar dos objetos. Para encontrar los ceros independientes de uuna
matriz se construye el árbol húngaro.

*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define OTAM 25
#define TAM 20        /* Tamaño de la matriz*/

#define CAPS 10
#define RENS 10
#define COLS 10

int mat_min[OTAM][OTAM]; /* Matriz minimizada */
int mat[OTAM][OTAM]; /* Matriz modificada */
int col[OTAM];
/*int mat_A[TAM][TAM][TAM],mat_B[TAM][TAM][TAM];*/
FILE *fpa,*fpb,*fpm,*fpos,*fneg,*fmat_1,*fmat_2,*fmat_A,*fmat_B; /* archivo que sera la matriz original
*/
int digN,digP;
void main()
{
/**/
void mueve_vox(int h,int tempor[TAM*TAM],int i,int j,int k);
int tama = 0;
int n = 0;
int tempor[TAM*TAM];

randomize();
int kn;
int insp,apoc;
int item;
int i,j,k,l;
int var,sat;

char obj_A[13],obj_B[13];
int digA,digB;
int t1,t2,t3,p,q;
double dou;
float dis;
int disp;
int h=0;
/**/
void minimiza();
void matching();
int inspecciona();
void visit_ceros();
int saturated();
```

```
void lmp_mat();
void print_mat();

 for(i = 0; i < TAM*TAM; i++)
    tempor[i] = 0;
/**/
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
  {
  mat_min[i][j] = 0;
  mat[i][j] = 0;
  }/* for */

  printf("\nIndique el archivo del primer objeto: ");
  scanf("%s",obj_A);
  fpa=fopen(obj_A,"r");

  printf("\nIndique el archivo del segundo objeto: ");
  scanf("%s",obj_B);
  fpb=fopen(obj_B,"r");

  fpos = fopen("pos.txt","w+"); /* abrimos los archivos donde estaran */
  fneg = fopen("neg.txt","w+"); /* los voxels positivos y negativos */

  for(k = 0; k < CAPS; k++)
   for(i = 0; i < RENS; i++){
     for(j = 0; j < COLS; j++)
   {
    fscanf(fpb,"%1d",&digB);
    fscanf(fpa,"%1d",&digA);
    printf(" digA,digB = %d,%d",digA,digB);
     if(digA == 0 && digB == 1){
      fprintf(fneg,"%1d",1);
      fprintf(fpos,"%1d",0);
      }
     if(digA == 1 && digB == 0){
      fprintf(fpos,"%d",1);
      fprintf(fneg,"%d",0);
      }
     if(digA == digB){
      fprintf(fpos,"%d",0);
      fprintf(fneg,"%d",0);
     } /* if */
  }  /* for j */
     fprintf(fpos,"%c\n",' ');
     fprintf(fneg,"%c\n",' ');
 } /* for i */
fclose(fpa);
fclose(fpb);
fclose(fpos);
fclose(fneg);
printf("Las matrices de los objetos A y B son:\n");
fpa=fopen(obj_A,"r");
fpb=fopen(obj_B,"r");
```

```
    fpos = fopen("pos.txt","r"); /* abrimos los archivos donde estaran */
    fneg = fopen("neg.txt","r"); /* los voxels positivos y negativos */

    for(k = 0; k < CAPS; k++)
     for(i = 0; i < RENS; i++) {
       for(j = 0; j < COLS; j++){
        fscanf(fpa,"%1d",&digA);
        printf("%d",digA);
        }
       printf("\n");
     }
    printf("\n");
    for(k = 0; k < CAPS; k++)
     for(i = 0; i < RENS; i++){
       for(j = 0; j < COLS; j++){
        fscanf(fpb,"%1d",&digB);
        printf("%d",digB);
        }
       printf("\n");
      }

   printf("Las matrices de los voxels positivos y negativos son:\n");

    for(k = 0; k < CAPS; k++)
     for(i = 0; i < RENS; i++) {
      for(j = 0; j < COLS; j++){
       fscanf(fpos,"%1d",&digA);
       printf("%d",digA);
       }
       printf("\n");
      }
    printf("\n");
    for(k = 0; k < CAPS; k++)
     for(i = 0; i < RENS; i++){
      for(j = 0; j < COLS; j++){
       fscanf(fneg,"%1d",&digB);
       printf("%d",digB);
        }
       printf("\n");
     }
      printf("\n");

fclose(fpa);
fclose(fpb);
fclose(fpos);
fclose(fneg);

fpa = fopen(obj_A,"r");
fpb = fopen(obj_B,"r");
fpos = fopen("pos.txt","r"); /* abrimos los archivos donde estaran */
/* los voxels positivos y negativos */
fmat_1 = fopen("mat_1.txt","w+");
fmat_2 = fopen("mat_2.txt","w+");
printf("Las distancias son:");
```

```
// Calculo distancias:
  for(k = 0; k < CAPS; k++)//Los siguientes tres ciclos barren la matriz pos
   for(i = 0; i < RENS; i++)
    for(j = 0; j < COLS; j++){
     fscanf(fpos,"%1d",&digP);
      if(digP == 1){
          fneg = fopen("neg.txt","r");
            for(l = 0; l < CAPS; l++) //Los siguientes tres ciclos barren la
             for(p = 0; p < RENS; p++) //matriz neg
              for(q = 0; q < COLS; q++){
               fscanf(fneg,"%1d",&digN);
                    if(digN == 1) {
                     t1 = i - p;
                     t2 = j - q;
                     t3 = k - l;
                     dou = t1*t1 + t2*t2 + t3*t3;
                     dis = sqrt(dou);
                     disp = dis*100.00;
                     fprintf(fmat_1,"%d",disp);
                     fprintf(fmat_2,"%d",disp);
                     printf("\nn,disp= %d,%d ",n,disp);
                     tempor[n] = disp;
                     n++;
                     }
            } /* for q */
          } /* if digP */
         fclose(fneg);
    } /* for j */
  fclose(fpa);
  fclose(fpb);
  fclose(fpos);
  fclose(fmat_1);
  fclose(fmat_2);

  tama = sqrt(n); /* Tamaño de la matriz a utilizar en el m,todo hungariano*/
  n = 0;
 for(i = 0; i < tama; i++){
 for(j = 0; j < tama; j++)
 {
 mat_min[i][j] = tempor[n];
n++;
 }/* for */
 printf("\n");
 }

fpm=fopen("mat_orig","w");
printf("La matriz de distancias es:\n");
for(i = 0; i < tama; i++){
 for(j = 0; j < tama; j++)
 {
 fprintf(fpm,"%d ",mat_min[i][j]);
 printf(" %d ",mat_min[i][j]);
}/* for*/
fprintf(fpm,"%c\n",' ');
```

```
    printf("\n");
    }
    fclose(fpm);

    minimiza(); //encuentra el valor minimo en renglon y columna, y resta
    matching();
    sat = 0;
    while(sat != 1){
     sat = saturated();  /* checa si las x's estan saturadas */
     lmp_mat(); /* limpia la matriz por si quedo marcada */
      if(sat != 1){
       matching();
       insp = inspecciona();
       if(insp == 1)
        sat = 1;
      }
     }

   // Recorro tempor[n]:
     fmat_1 = fopen("mat_1.txt","r");
     while (h < tama) {
     printf("El valor de h es %d",h);
      for(k = 0; k < CAPS; k++)/* Los siguientes tres ciclos barren la matriz pos*/
       for(i = 0; i < RENS; i++)
        for(j = 0; j < COLS; j++){
              fscanf(fmat_1,"%1d",&digA);
              if(digA == tempor[h])
               mueve_vox(h,tempor,i,j,k);
       } /* for */
              h++;
     }
     fclose(fmat_1);
     fpb = fopen(obj_B,"r");
     printf("voxels movidos:\n");
      for(l = 0; l < CAPS; l++)
       for(p = 0; p < RENS; p++) {
        for(q = 0; q < COLS; q++){
         fscanf(fpb,"%1d",&digB);
              printf("%d",digB);
              }
              printf("\n");
        }
     fclose(fpb);
    } /*main*/
   /* ------------------------------------------------------------ */
    void minimiza()
    {
    int i,j,k=0;
    int min;
   /* Busca el menor de cada renglén y lo resta */
    for(i = 0; i < TAM; i++) {
   min = mat_min[i][k];
    for(k = 0; k < TAM; k++) {
     if(mat_min[i][k] < min)
```

```
        min = mat_min[i][k];
       }
      for(j = 0; j < TAM; j++)
       mat_min[i][j] = mat_min[i][j] - min;
      k = 0;
     } /* for*/
    /* busca el menor de cada columna y lo resta */
     for(j = 0; j < TAM; j++) {
     min = mat_min[k][j];
      for(k = 0; k < TAM; k++) {
       if(mat_min[k][j] < min)
         min = mat_min[k][j];
       }
      for(i = 0; i < TAM; i++)
       mat_min[i][j] = mat_min[i][j] - min;
      k = 0;
     } /* for*/
    } /* minimiza*/
   /* ---------------------------------------------------------------- */
    void matching()
    {
     int i,j = 0,k,l;
     int marcado;
     int rec_colM(int a);
     void print_mat();

      for(i = 0; i < TAM; i++){
       marcado = 0;
       while(marcado == 0 && j < TAM)
       {
       if(mat_min[i][j] == 0 && rec_colM(j) == 0){
        mat_min[i][j] = -3; /* marca el primer cero encontrado en el renglón */
        marcado = 1;
        }
       j++;
      } /* while */
      j = 0;
     } /* for */
    print_mat();
   } /* matching() */
  /* ---------------------------------------------------------------- */
  int saturated()
  {
   int var,ncer,rm=0,rc=0;
   int marcado,cont=0;
   int i,j,insp=0;
   int hngrian(int a);
   void lmp_mat();
   void visit_ceros();
   void sum_max_min();
   int inspecciona();
   int hngrmod(int a,int b);
   int rec_cers(int a);
   int rec_marks();
```

```
  void desmark_col();
   for(i = 0; i < TAM; i++){
    marcado = 0; j = 0; ncer = 0;
     while(marcado == 0 && j < TAM){ /* mientras no se encuentre una marca */
      if(mat_min[i][j] == -3 || mat_min[i][j] == -5){ /* si se encuentra una marca.. */
        marcado = 1; /* x esta saturado */
        cont++; /* cuenta las x's saturadas */
       } /*if */
      j++;
     } /* while */
    if(marcado == 0) { /* si x_i no esta saturado */
    j = 0;
     while(j < TAM){
      if(mat_min[i][j] == 0.0)
       ncer++; /* checa si hay ceros en el renglon */
     j++;
     } /* while j<TAM*/
      if(ncer == 0){
       limp_mat();
       visit_ceros();
       return(0);
       }
    rc = rec_cers(i); /* recorre o visita cada cero del renglon i (no saturado) */
     while(rm != 1){ /*tiene que dejar de encontrar marca en la columna del 0*/
      rm = rec_marks();
     } /* while rm */
      insp = inspecciona();/* e inspecciona si tenemos ceros independientes */
      desmark_col();
      if(insp == 0){ /* si no ha encontrado el "augmenting path" */
       limp_mat(); /*limpia la matriz de marcas */
       visit_ceros(); /* hace m s ceros */
      matching(); /* hace un nuevo matcheo */
      insp = inspecciona();/* e inspecciona si tenemos ceros independientes */
       if(insp == 0) return(0);
            }
     } /* if(marcado == 0) */
    } /* for */
    if(cont == TAM) insp = inspecciona(); /* Si todas las x's estan saturadas.. */
     if(insp == 0) return(0);
     return(1);
  } /* saturated() */
/* ------------------------------------------------------------------- */
 int rec_marks() /* recorre marcas */
 {
 int var;
 int cont=0;
 int i,j,rc;
 int rec_cers(int a);
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
   if(mat_min[i][j] == -3 && mat[i][j] == -7){ /*si esta doblemente marcado*/
    mat[i][j] = -3; /* una vez que ya no se utiliza la marca, la desmarca */
    cont++;
    rc = rec_cers(i);
```

```
    if(rc == 0) /* si no encontro marca, sale */
     return(1);
    }
    if(cont == 0) /* si no encontro ningun cero doblemente marcado.. */
     return(1); /* sale */
   return(0); /* todos los ceros encontraron marca */
   } /* rec_marks */
  /* ---------------------------------------------------------- */
   int rec_cers(int i)
   {
   int var,m,j,bm;
   int busc_mark(int a);
   int hngrmod(int a,int b);
   void lmp_col();
   for(j = 0; j < TAM; j++){
    if(mat_min[i][j] == 0 && col[j] != -4){
     mat[i][j] = -7;
     bm = busc_mark(j); /* busca marca y la remarca en mat[][] */
     col[j] = -4;       /* Marca la columna j*/
      if (bm == 0) { /* si el cero visitado no encuentra marca.. */
            hngrmod(i,j); /* hay un "augmenting path" */
            return(0);
       }
    } /* if */
   } /* for */
   return(1);
  } /* rec_cers() */
 /* ---------------------------------------------------------- */
  int hngrmod(int mp,int np)
  {
   int var,vm,i,j,hnmod,vc;
   int bm=0;
   int path=0,encer,marcado;
   void marca_col(int ren);
   void desmark_col();
   void lmp_mat();
   void print_mat();
   int busc_cer(int *,int *);
   int va_mark(int a,int *);
   int va_cer(int *,int b);
   void lmp_col();

   lmp_col();
 /*  printf(" visita mat_min[%d][%d] = %d",mp,np,mat_min[mp][np]);
   scanf("%d",&var);*/
    mat_min[mp][np] = -5; /* marca al primer cero */
    col[np] = -4;
  while(path == 0){
  vm = va_mark(mp,&np); /* busca la marca en el renglon */
  if(vm == 1){ /* si la encontro..*/
   mat_min[mp][np] = -6; /* la hace cero */
   vc = va_cer(&mp,np);
  if(vc == 0) path = 1; /* si no encuentra cero termina la trayectoria */
  else if(vc == 1)
```

```
      mat_min[mp][np] = -5;
      col[np] = -4; /* como el cero se convierte en marca, no debe ir por
                       el mismo */
     }/* if (vm == 1)*/
     else if(vm == 0){
      path = 1;
     } /* else if(vm == 0)*/
    } /* while path */
   if(bm == 1) /* si el cero tiene una marca */
    return(0);
     return(1);
  } /* hngrmod() */
  /*-----------------------------------------------------------------
   La funciŽn visit_ceros()
   - visita cada cero de la matriz.
   - recorre el renglŽn y la columna de cada cero.
   - "Marca" las columnas y renglones necesarias para cubrir a todos los
      ceros.
   -----------------------------------------------------------------*/
  void visit_ceros()
  {
  int a,b;
  int i,j,r,c,k;

  int recorre_ren(int var); /* funciŽn que da el numero de
                                        ceros en el renglŽn */
  int recorre_col(int var); /* funciŽn que da el numero de
                                        ceros en la columna  */
  void sum_max_min();
  void print_mat();

   for(i = 0; i < TAM; i++)
    for(j = 0; j < TAM; j++)
     mat[i][j] = 0;
  /* A continuaciŽn se calcula el numero de ceros en columna-renglŽn  */
  /* por c/cero y se etiqueta cada cero */
   for(i = 0; i < TAM; i++)
   for(j = 0; j < TAM; j++)
   {
   if(mat_min[i][j] == 0 && mat[i][j] != -1)
    {
  r = recorre_ren(i);
    c = recorre_col(j);
    printf("\nEl n£mero de ceros en la columna %d es: %d",j,c);
  /* Si hay m s ceros en el renglon que en la columna, "marca" un renglon me-
    ti.ndo -1's: */
   if (r > c || r == c)
   {
     for(k = 0; k < TAM; k++){
     if(mat[i][k] == -1)
           mat[i][k] = -2;
           else if(mat[i][k] != -1)
            mat[i][k] = -1; /* "Marca" todo el renglon i */
          }
```

```
    } /* if */
  /* Si hay m s ceros en la columna que en el renglφn, "marca" una columna me-
    ti.ndola a un arreglo: */
    if (r < c)
    {
      for(k = 0; k < TAM; k++) {
           if(mat[k][j] == -1)
            mat[k][j] = -2;
            else if(mat[k][j] != -1)
             mat[k][j] = -1; /* "Marca" toda la columna j. */
           }
    } /* if */
    } /* if(mat(i,j) = 0)*/
  } /* for (j)*/
  sum_max_min();
  } /* visit_ceros() */
 /*-------------------------------------------------------------------------*/
  void sum_max_min()
  {
  int i,j;
  int min;
  int bc,br;
  int busca_en_ren(int n,int m);
  int busca_en_col(int n,int m);
  min = 32000.;
   for(i = 0; i < TAM; i++)
    for(j = 0; j < TAM; j++)
      if((mat[i][j] != -1) && (mat[i][j] != -2) && (mat_min[i][j] < min))
           min = mat_min[i][j];

   for(i = 0; i < TAM; i++)
    for(j = 0; j < TAM; j++){
     if(mat[i][j] != -1 && mat[i][j] != -2)
       mat_min[i][j] = mat_min[i][j] - min; /* resta el minimo a los elementos que no se
                                               encuentran en ningun renglon "marcado"*/
    else if(mat[i][j] == -2)
       mat_min[i][j] = mat_min[i][j] + min; /* suma el minimo a los elementos que se
                                               encuentran "doblemente marcados" */
   }
  } /* sum_max_min*/
 /*-------------------------------------------------------------------------*/
 int inspecciona()
 {
 int recorre_ren(int var);
 int recorre_col(int var);
 int W=0;
 int cont=0;
 int i,j,r,c,p,q;
 int num;

  fpm = fopen("mat_orig","r");
  for(i = 0; i < TAM; i++){
   for(j = 0; j < TAM; j++)
   {
```

```
    fscanf(fpm,"%d",&num);
    if(mat_min[i][j] == -3 || mat_min[i][j] == -5) /* si el cero esta marcado */
    {
    // se trata de un cero independiente:
     cont++; // contador que lleva el n£mero de ceros independientes
      W = W + num;
           if(cont == TAM) // Todos los ceros son independientes
      {
           printf("\nEl peso mínimo es: %d\n",W);
           fclose(fpm);
           return 1;
      } /* if(cont == TAM)*/

    } /* if(mat_min(i,j) = 0)*/
   } /* for (j)*/
  } /* for (i)  */
  fclose(fpm);
   return 0;
  } /* inspecciona()*/
/* ---------------------------------------------------------------- */
 void desmark_col()
 {
  int i=0;
    for(i = 0; i<TAM; i++){
     if(col[i] == -4)
          col[i] = 0;
    }
 }
/* ---------------------------------------------------------------- */
void lmp_col()
{
int m;
 for(m = 0; m < TAM; m++)
   col[m] = 0;
}

void lmp_mat()
{
 int i,j;
 for(i = 0; i < TAM; i++)
  for(j = 0; j < TAM; j++)
   {
   if(mat_min[i][j] == -3. || mat_min[i][j] == -6. || mat_min[i][j] == -5.) /* si el cero de mat esta marcado */
    mat_min[i][j] = 0.; /* le quito la marca */
   }
}
/* ---------------------------------------------------------------- */
int recorre_ren(int iaux)
{
int cont=0,k;
 for(k = 0; k < TAM; k++){
  if (mat_min[iaux][k] == 0)
   cont++;     // Cuenta el n£mero de ceros en el renglón
 }
```

```
    return cont;
   } // recorre_ren();

   int recorre_col(int jaux)
   {
    int cont=0,k;
    for(k = 0; k < TAM; k++){
     if (mat_min[k][jaux] == 0)
      cont++;   // Cuenta el n£mero de ceros en la columna
      }
    return cont;
   } // recorre_col();
  /* ---------------------------------------------------------------- */
   int rec_colM(int jaux)
   {
    int cont=0,k;
    for(k = 0; k < TAM; k++){
     if (mat_min[k][jaux] == -3 || mat_min[k][jaux] == -5)
      cont++;   /* Cuenta el n£mero de ceros marcados en la columna */
      }
    return cont;
   } /* rec_colM(); */
  /* ---------------------------------------------------------------- */
  int busc_mark(int np)
  {
   int var,tmp;
   tmp = 0;
   while(tmp < TAM)
   {
    if(mat_min[tmp][np] == -3 || mat_min[tmp][np] == -5) {
     mat[tmp][np] = -7; /* remarca el cero de mat_min al usar mat[][] */
     return(1);
    }
   tmp++;
   } /* while */
   tmp--;
   return(0);
  } /* busc_mark() */
 /* ---------------------------------------------------------------- */
 int busc_cer(int *mp,int *np)
 {
  int var,tnp=0;
  while(tnp < TAM)
  {
   if((mat_min[*mp][tnp] == 0 || mat_min[*mp][tnp] == -6) && col[tnp] != -4) {
    *np = tnp;
    return(1);
   }
  tnp++;
  } /* while */
  tnp--;
  *np = tnp;
  return(0);
 } /* busc_mark() */
```

```
/* ---------------------------------------------------------------- */
int va_mark(int mp,int *np)
{
 int var,tnp;
  tnp = 0;
  while(tnp < TAM)
  {
  if((mat_min[mp][tnp] == -3 || mat_min[mp][tnp] == -5) && col[tnp] != -4) {
   *np = tnp;
   return(1);
  }
 tnp++;
 } /* while */
 tnp--;
 *np = tnp;
 return(0);
} /* busc_mark() */
/* ---------------------------------------------------------------- */
int va_cer(int *mp,int np)
{
 int var,tmp=0;
  while(tmp < TAM)
  {
  if((mat_min[tmp][np] == 0 || mat[tmp][np] == -6) && mat[tmp][np] == -7) {
   *mp = tmp;
   return(1);
  }
 tmp++;
 } /* while */
 tmp--;
 *mp = tmp;
 return(0);
} /* va_cer() */
/* ---------------------------------------------------------------- */
void print_mat() /* Imprime matriz resultante */
{
 int i,j;
 printf("\n");
  for(i = 0; i < TAM; i++)
  {
  for(j = 0; j < TAM; j++)
  printf(" %d ",mat_min[i][j]);
  printf("\n");
  } /* for */
}
void mueve_vox(int h,int dis_h[OTAM],int m,int n,int o)
 {
int l,p,q,i,j,k,dig2;
  fmat_2 = fopen("mat_2.txt","r");
  fmat_B = fopen("mat_B.txt","w+"); /* se hara el movimiento de voxels
                                       hacia mat_B.txt */
  for(l = 0; l < CAPS; l++) //Los siguientes tres ciclos barren la
  for(p = 0; p < RENS; p++) /*matriz neg*/
```

```
    for(q = 0; q < COLS; q++){
      fscanf(fmat_2,"%1d",&dig2);

   for(i = 0; i < CAPS; i++) /*Los siguientes tres ciclos barren la*/
    for(j = 0; j < RENS; j++) /*matriz neg*/
     for(k = 0; k < COLS; k++){
      fscanf(fmat_2,"%1d",&dig2);
      if(i == m && j == n && k == o && dig2 == dis_h[h]){
            fprintf(fmat_A,"%d",0);
            fprintf(fmat_B,"%d",1); /* muevo voxel */
            }
          }
     }
fclose(fmat_B);
 fclose(fmat_2);
 } /* mueve_vox(tempor[h],i,j,k)*/
```

• BIBLIOGRAFÍA

[1] 'Ballard, Brown, *Computer Vision*. Prentice Hall, Ennglewood Cliffs, New Jersey (1982).

[2] E. Bribiesca, *Measuring 3-D Shape Similarity Using Progresive Transformations*. Pattern Recognition, vol 29, pp. 13.

[3] Kwo-Liang Chung. *Finding shape numbers in parallel.* Pattern Recognition Letters. Vol 16, pp 699-702.

[4] L.G. Roberts, *Machine Perception of Three Dimensional Solids*, Optical and Electro-Optical Information Processing. J. P. Tippett et al., eds. MIT Press. Cambridge Massachutts.

[5] E. Bribiesca, *La Topografía del Valle de México Representada en forma Digital*, vol 3, No. 7. 1993. Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, UNAM.

[6] Ejiri, M., T. Uno, H. Yoda, T. Goto, and K. Takeyasu. *An intelligent robot whith cognition and decision-making ability.* Proc., 2nd IJCAI, September 1971,350-358.

[7] Falk, G. *Interpretation of important line data as a three-dimensional scene.* Artificial intelligence 3,1, Spring 1972,77,100.

[8] Shirai, Y. *Analyzing intensity arraysusing knowledge about scenes.* In PCV, 1975.

[9] Turner, K. J. *Computer perception of curved objects using a television camera.* PhD dissertation, Univ. Edinburg, 1974.

[10] K, Kanatani, *Group of Theoretical Methods in Image Understanding*, 1990, Springer - Verlag.

[11] M. K. Hu, *Visual Pattern Recognition by Moment Invariants*. IRE, vol 8, pp 179- 187, Feb 1962.

[12] Chartrand, Gary. *Introductory Graph Teory*, Dover Publications.

[13] Even, Shimon. *Graph Algorithms*. Addison-Wesley.

[14] Bondy, J. A. and Murty, U.S.R. *Graph Theory with Applications*. Department of Combinatorics and Optimization, University of Waterloo, Ontario, Canada.

[15] König, D., *Graphs and Matrices (Hungarian)*. Mat. Fig. Lapok, vol 38 (1931), 26-30.

[16] Egerváry, J., *Matrixok Kombinatorikus Tulajdonságairól*. Mathematika és Fizikai Lápok, vol 38(1931), pp 16-28.

[17] Hecht, E & Zajac, A. *Óptica*. Addison-Wesley Iberoamericana. 1986.

[18] R. N. Bracewell. *The Fourier Transform and Its Applications*, Electrical and Electronics Engineering Series, McGraw-Hill Book Company(1978).

[19] E. Brigham, *The Fast Fourier Transform and Its Applications*, Prentice-Hall, (1988).

[20] H. Wechsler, *Invariance in Pattern Recognition*, In P.W. Hawkes, editor, Advances in Electronics and Electron Physics, **69**, pp. 262-322. Academic Press, (1987)

[21] W.Karush, *Webster's New World Dictionary of Mathematics*, Simon & Schuter, Inc., New York, 1989.

[22] R. L. Haralick & L.G. Shapiro, *Glosary of Computer Vision Terms*, Pattern Recognition. **24**, 1991, 69-93.

[23] R. Resnick & D. Holliday. *Física*, Parte I. CECSA. 1986.

[24] G. H. Schut, *On Exact Equations for the Computation of the Rotational Elements of Absolute Orientation*, Photogrammetry. **17**, 1992-93, 34-37.

[25] A. Pope, An Advantageous Alternative Parametrization of Rotation for Analytic Photogrammetry, *Symposium on Computational Photogrammetry of the American Society of Photogrammetry*, Alexandria, VA. 1970.

[26] L. Hinsken, A singularity-free Algorithm for Spatial Orientation of Bundles, *International Archives of Photogrammetry and Remote Sensing*, Vol. 27, Kyoto, Japan, 1988.

[27] B. K. P. Horn, *Closed-form Solution of Absolute Orientation using Unit Quaternions*, Journal of the Optical Society of America A., **4**, 1987, 629-642.

[28] R. L. Haralick & L.G. Shapiro, *Computer & Robot Vision*, Volume II, Adison-Wesley Publishing Company, 1993.

[29] K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall Information and System Sciences Series. Thomas Kailath, series Editor. pp 569.

[30] Dodwell, P. C., *Visual Pattern Recognition*, Holt, Rinehart & Wintson, New York,1970.

[31] Neisser, Y, *Cognitive Psychology*, Appleton Century Crofts, New York, 1967.

[32] Tamura, H., *A Comparison of Line Thinning Algorithms from Digital Geometry Viewpoint*, Proceedings of the 4th International Conference on Pattern Recognition, Kyoto, Japan, Nov. 7-10, 1978, pp 715-719.

[33] Haber, R. N., and Wilkinson, L., *Perceptual Components of Computer Displays*, IEEE Computer Graphics and Applications, vol. 2, no. 3, May 1982, pp. 23-34.

[34] Marr, D., *Early Processing of Visual Information*, Philosophical Transactions of the Royal Society, London, ser. B, vol. 275, 1976, pp. 483-524.

[35] Zusne, L., *Visual Perception of Form*, Academic, New York, 1970.

[36] P. Besl and R. Jain. *Three-dimensional Object Recognition*, ACM Comput. Surv. 17(1). 75-145 (1985).

[37] Attneave, F., *Some Informational Aspects of Visual Perception, Phsycological Review*, vol. 61, no. 3, 1954, pp. 183-193.

[38] Freeman, H., *On the Classification of Line-Drawing Data*, in Dunn, W. (de.), Models for the Perception of Speech and Visual Form, MIT Press,

[39] Rosenfeld, A., and Weszka, J. S., *An Improved Method of Angle Detection on Digital Curves*, IEEE Transactions of Computers, vol. C-24, no. 9, September 1975, pp. 940-941.

[40] R. Brooks, *Model Based 3-D Interpretations of 2-D Images*, IEEE Trans. Pattern Anal. Mach. Intell. 5(2), 140-150(1983).

[41] Gould, R., *Graph Theory*, Emory University, The Benjamin/Cummings Publishing Company, Inc. 1988.

[42] E. Bribiesca, D. A. Rosenblueth and M. Garza-Jinich. *Definit-Clause Grammars for 2D Analysis*. Computers Math. Applic. Vol. 30, No. 8, pp 95-103, 1995.

[43] E. Bribiesca and Richard G. Wilson. A measure 2D Shape of Object Dissimilarity.

[44] E. Bribiesca, *A geometric structure for two-dimensional shapes and thre dimensional surfaces*, Pattern Recognition, vol 25, 483-496 (1992).

[45] S. J. Dickinson, A. P. Pentland and A. Rosenfeld, *From Volumes to Views: an aproach to 3-D Object Recognition*. CVGIP: Image Understanding 55, 130-154(1992).

[46] J. W. Boyse. *Data Structure for Solid Modeller*, NFS Workshop on the Representation on Three-Dimensional Objects, University of Pennsylvania (1979).