Centro de Investigación y de Estudios Avanzados del I.P.N.

Departamento de Ingeniería Eléctrica
Sección Computación

Construcción de un editor de gráficas de propósito general

Tesis que presenta
Lic. David Cruz Rojas

Para obtener el grado de Maestro en Ciencias en la especialidad de Ingeniería Eléctrica

Directores de Tesis
Dr. Harold V. McIntosh
Dr. Sergio V. Chapa Vergara

1 de septiembre de 1999

Agradecimientos.

Primero quiero agradecer a mamá y papá por su amor y todo lo que pasaron, para que llegara a este punto. A mis hermanos por su cariño y solidaridad, a mi hermano Octavio por su interés y apoyo en este trabajo. A Maru por ser mi mejor amiga y apoyarme en todo momento.

A la gente de CINVESTAV, a José Manuel Gómez Soto por sus valiosas sugerencias en el proyecto, al Dr.McIntosh por su gran disposición y generosidad, a Sofía Reza por sacarme de apuros en muchos momentos, a la amistad con el Dr. Chapa (no es barba) y a todas las amistades que he hecho en este trayecto; Abdiel, Rafa, Neli, Hugo, Susana, Eduardo, Noé; y otras que seguramente se me olvidan (Ups!) pero de antemano, les doy las gracias.

Introducción

En diferentes ramas del conocimiento se usan gráficas. Una gráfica es un conjunto de vértices o puntos y líneas que los unen. En una gráfica, podemos representar arbitrariamente un concepto con el vértice; y con la línea o relación, otro. De este modo, podemos hablar del conjunto de conceptos asociado a los vértices y del conjunto de conceptos asociados a la relación.

Por ejemplo, en un habitat, a los distintos tipos de animales les podemos asociar el conjunto de los vértices y si un tipo de animal es susceptible de ser comido por otro lo podemos unir mediante una línea. Así con la gráfica resultante podemos visualizar las cadenas alimenticias en ese ecosistema. A fin de cuentas una gráfica es un dibujo y si queremos manipular este dibujo fácilmente, que mejor que la computadora. A la acción de manipular una gráfica la conoceremos como la edición de una gráfica o diagrama.

Si extendemos el ejemplo descrito, a los animales y plantas de todo el planeta, la gráfica crecería bastante. El problema que tiene una gráfica grande es que no podemos percibirla en su totalidad. Hay tal cantidad de nodos en la pantalla que no podemos apreciar ninguna estructura, no podemos percibir la parte que nos interesa de la gráfica. Es por esta razón que lo que queremos crear es una herramienta que nos permita colorear secciones de una gráfica, esconder las partes que no nos interesan y sobretodo visualizar la estructura interna de una gráfica, lo que nos ahorraría tiempo y esfuerzo.

La herramienta que describimos en este documento aborda la problemática de edición, manipulación y análisis de gráficas que en general son complejas.

La utilidad de primera mano de la herramienta diseñada, es para

el análisis del comportamiento de ciertas abstracciones matemáticas, llamadas autómatas celulares, mediante la manipulación de cierta clase de diagramas. Sin embargo, esto no significa que sólo pueden ser usados en ese universo.

Los diagramas que aparecen en el contexto de los autómatas celulares son difíciles de editar, debido a que son demasiado grandes y es tediosa su manipulación, y porque en ocasiones nos interesa visualizar solo ciertas secciones de la gráfica y perdemos más tiempo operando la gráfica que en su explotación real. Usaremos los diagramas mencionados para ejemplificar el uso del graficador.

Con respecto a la estructura interna, el graficador está constituido por una serie de clases en los que se ha buscado que el diseño sea lo más reutilizable posible. Con este fin se han usado patrones de diseño, es decir configuraciones de objetos en lo que la reutilización ya está probada.

El documento elaborado tiene como finalidad, por un lado, exponer los problemas de editar una gráfica grande y solucionarlos en la medida de lo posible con una herramienta específica de edición y un lenguaje de manipulación y edición de gráficas. Por otro lado, dar a conocer un panorama general de cómo está construida la aplicación y cómo se pueden extender aspectos que varían de un editor de gráficas a otro.

El trabajo está dividido en dos partes, la primera parte está dedicada a las problemáticas de edición y la segunda parte a la estructura interna del sistema.

En el capítulo 1 de este trabajo, veremos las generalidades del concepto de gráfica; conceptos principales y las representaciones más usadas; con el fin de que el lector se familiarice con los conceptos básicos de gráficas. A partir de lo anterior, en el capítulo 2, describimos gráficas específicas donde se presentan problemas típicos de edición y resumiremos este tipo de problemas. En el capítulo 3 estudiaremos el proceso general de edición, veremos; las operaciones básicas del editor y una propuesta de un lenguaje gráfico de manipulación de gráficas; necesarios para la resolución de problemas de edición. Al final de la primera parte, en el capítulo 4 veremos en un caso práctico, cómo podemos utilizar este editor. Ya en la segunda parte, en el capítulo 5 veremos cuál es la arquitectura general de nuestro sistema y finalmente en el capítulo 6, describiremos el catálogo de clases de lo construido, con el fin de su

reuso en otros contextos.

Este trabajo ha tenido como meta final, mostrar un enfoque para facilitar el manejo de gráficas grandes así como la descripción de un editor que adopta este enfoque, con el fin de que se puedan desarrollar en el editor nuevas operaciones de edición y así, hacer la herramienta cada vez más versátil.

Contenido

Parte I

Edición de gráficas

Capítulo 1

Teoría de gráficas

1.1 Introducción.

Este capítulo tiene por objetivo proporcionar; la terminología y los conceptos básicos en lo referente a gráficas. En este capítulo se darán respuesta a interrogantes tales como, ¿qué es una gráfica?, ¿para qué sirve?, ¿cómo se nombran a las partes que la componen? , ¿qué criterios sirven para comparar una gráfica contra otras?, ¿por qué se dice que una gráfica es equivalente a otra?, ¿cómo se representa?, ¿cómo se descompone una gráfica en sus partes?, ¿cómo se puede construir una gráfica en función de partes más pequeñas?

1.2 Conceptos básicos.

La idea de gráfica es simple. Por ejemplo, si tomamos al conjunto de miembros de una familia, y a la relación de parentesco que existe entre ellas tendremos una gráfica. Cada miembro de la familia es un nodo o vértice y cada relación particular de parentesco entre una persona y otra es la relación o liga. Así el concepto de gráfica aparecerá cada vez que tengamos un conjunto y relaciones entre sus elementos. De este modo un diagrama de flujo, una red de carreteras, un circuito eléctrico, una molécula pueden ser vistos como una gráfica.

Una gráfica se puede representar esquemáticamente. Si se requiere representar a los estados y a las colindancias entre sí, la gráfica es-

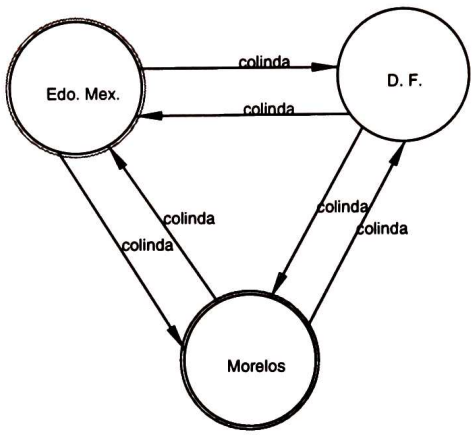

Figura 1.1: Colindancias de la ciudad mediante una gráfica.

quematizada tendrá la apariencia de la figura 1.1.

1.2.1 Definiciones.

Una vez introducido el concepto intuitivo de gráfica, podemos dar una definición un poco más formal.

Definición. Si tenemos un conjunto $A, B, C...$ de puntos ([Kon36] y [Sal79]) los cuales están conectados por una o más líneas, la configuración resultante es llamada *gráfica.* L os puntos $A, B, C...$ son llamados *vértices o nodos* de una gráfica, finalmente las líneas $AB, CD, AD, DF, ...$ involucradas son llamadas *aristas o arcos.*

La forma usual de simbolizar una gráfica es $G = (V, E)$
donde
$V = \{v_1, v_2, v_3, ...\}$ son los vértices o nodos
$E = \{e_1, e_2, e_3, ...\}$ son los arcos
A una gráfica, a la que se le han eliminado rizos y ramas paralelas

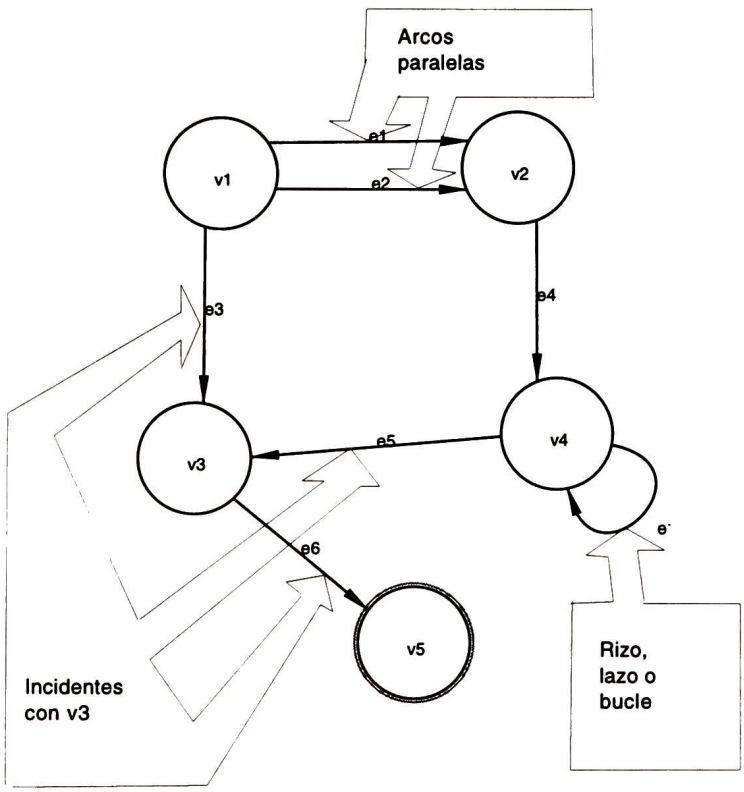

Figura 1.2: Partes de una gráfica.

(ver figura 1.2), se le conoce como *simple.* Al número de ramas que concurren en un nodo se le conoce como el *grado de un vértice.*

Propiedades relacionadas con la gráfica son :

1. La suma de grados de los vértices de toda la gráfica es igual a dos veces el número de arcos.

$$\sum_{i=1}^{n} g(v_i) = 2|E|$$

2. El número de nodos de grado impar es siempre par.

Digráfica o gráfica dirigida.

Es aquella gráfica que tiene un número finito de vértices y las ramas que los unen establecen una relación. (Pareja ordenada de vértices).
$D = (V, E) \ni V = \{v_1, v_2, ..., v_n\}$
y E es una relación binaria en V : $v_i E v_j$.

Subgráficas.

Sea G un grafo $< V, E >$; si un grafo G_S consiste de $< V_S, E_S >$ tal que $V_S \subseteq V$ y $E_S \subseteq E$, entonces decimos que G_S es un subgrafo de G. (Ver fig 1.3).

Vértice aislado y vértice extremo.

Un vértice aislado es aquel que no tiene conexión con otro vértice. Y un vértice extremo es aquel que sólo tiene una conexión con otro vértice. (Vease fig 1.4).

Gráfica nula.

Es la gráfica que no tiene arcos, sólo vértices aislados. (Vease fig 1.5).
$E = \emptyset$
$V \neq \emptyset$
Si $E = V = \emptyset$ no se define una gráfica.

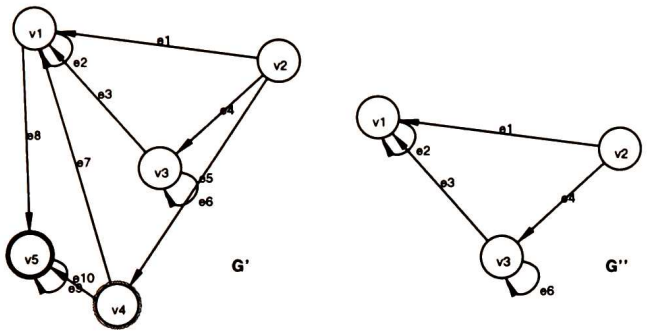

Figura 1.3: Subgráficas.

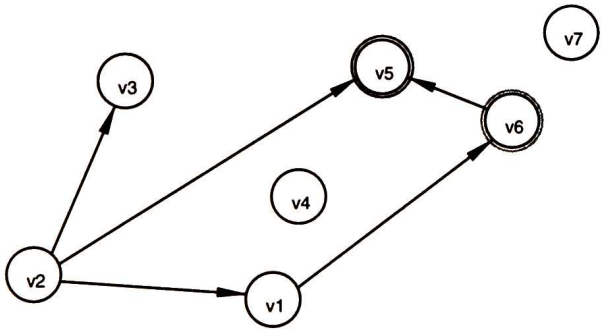

Figura 1.4: Vértices aislados(v4 y v7) y extremos(v3).

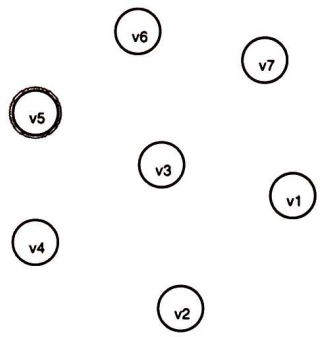

Figura 1.5: Gráfica nula.

1.2.2 Mapeos entre gráficas.

Isomorfismo en gráficas.

Dos gráficas son isomórfas cuando existe una correspondencia uno a uno entre arcos, vértices y grados de cada vértice, (ver fig 1.6). Esto significa que tenemos un mapeo

$\Phi : V(G_1) \to V(G_2)$ tal que

$$\forall x, y \in V(G_1),\ xy \in E(G_1) \longleftrightarrow \Phi(x)\Phi(y) \in E(G_2)$$

Homomorfismo en gráficas.

Cuando entre dos gráficas existe un mapeo

$\Phi : V(G_1) \to V(G_2)$ tal que

$$\forall x, y \in V(G_1),\ xy \in E(G_1) \longrightarrow \Phi(x)\Phi(y) \in E(G_2)$$

Un homomorfismo es un concepto muy útil porque permite simplificar una gráfica en otra, conservando varias de sus propiedades.
Un isomorfismo siempre es un homomorfismo, pero no lo es, en el sentido opuesto.

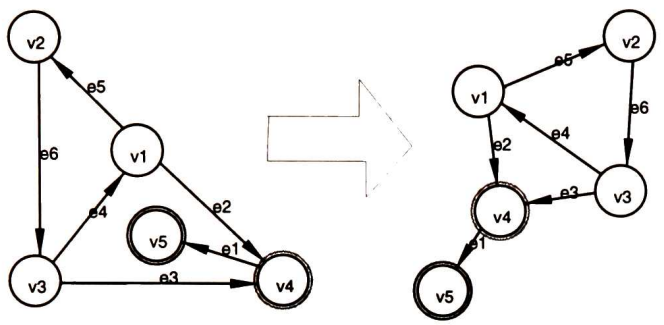

Figura 1.6: Isomorfismo de gráficas.

1.2.3 Clasificación de gráficas conexos.

Una gráfica es conexa cuando partiendo de un vértice v_i se puede ir a otro vértice cualquiera v_j a través de las ramas de la gráfica.

- *Paseo, tren o cadena.* Es una secuencia de ramas sin retrazo. Ejemplo: en la figura 1.6, un paseo es: v_1,v_2,v_7,v_4.

- *Paseo cerrado* es un paseo que inicia y termina en el mismo vértice. Cuando no tienen intersecciones nodales intermedias, recibe el nombre de **circuito**. En la figura 1.7 v_1,v_2,v_7 ,v_6 y v_1 es un circuito. Una gráfica que no tiene ciclos en él, se le llama acíclico.

- *Paseo abierto.* Es el que se inicia y termina en distinto vértice. Cuando no tiene intersecciones nodales intermedias, se le llama **trayectoria o ruta**.

Las gráficas que trataremos en este trabajo siempre serán dirigidas. Por otro lado, a las gráficas que no tienen ciclos y que además están conectadas se les conoce como *árboles.*

A un árbol que pasa por todos los nodos de una gráfica se le conoce como *árbol de cobertura.*

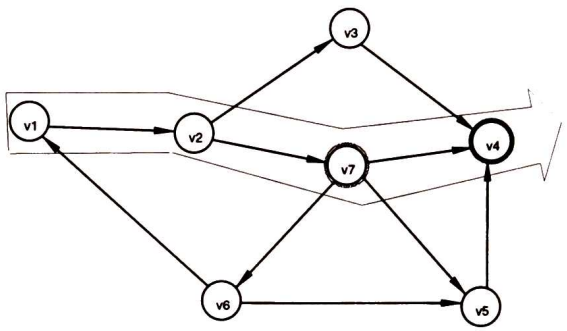

Figura 1.7: Un paseo.

1.3 Representaciones.

Una gráfica puede ser representada de múltiples maneras, cada representación hará énfasis en cierta característica de la gráfica. Analizaremos algunas representaciones útiles [McI91]:

- Definición matemática. Una gráfica formalmente descrito es un conjunto de vértices V y un conjunto de relaciones $R \subseteq V \times V$.

- Forma gráfica. La representación gráfica es la forma esquemática en que hemos venido mostrando la mayoría de los conceptos. Es decir que a cada nodo en la representación, se le asocia un punto o círculo en el dibujo y a cada arco una línea. A menudo se etiqueta al nodo como un miembro del conjunto V.

- Matriz topológica. A la matriz más comúnmente usada se le conoce como matriz topológica y se define así:

$$M_{ij} = \begin{cases} 1 & \text{si } iRj \\ 0 & \text{si } i\not Rj \end{cases}$$

  Una de las aplicaciones más importantes de esta representación es que sus potencias describen veredas entre nodos.

- Ecuaciones simbólicas. Al describir una gráfica podemos representar cada nodo con las letras X, Y, Z y bajo este esquema podemos representar las *ecuaciones de entrada* como sigue:

$$X = Ya + Zb + \lambda$$

La ecuación significa que, a partir del nodo Y mediante la liga a se llega a X, o bien de Z con b se llega a X, la λ es usada si X es un nodo terminal. El conjunto de todas las ecuaciones de entrada o de salida (se trata de lo mismo, solo que ahora en lugar de entrar a X se sale de ella) puede ser representado mediante una forma matricial. Es más, si intentamos solucionar este sistema de ecuaciones, obtendremos como solución, un conjunto de expresiones regulares, que representarán la secuencia de aristas que hay que seguir para llegar a un cierto nodo. Esto último, nos da a pensar, que podríamos representar cada nodo con su expresión regular a partir de cierto nodo, sin embargo, el inconveniente de esta posible representación es que a dos nodos, podría correponderle la misma expresión regular.

1.4 Operaciones en gráficas.

En vista de que $G = (V, E)$ está definida en términos de los conjuntos V y E es posible definir algunas operaciones sobre G empleando las propiedades de los conjuntos.

1.4.1 Unión de dos gráficas.

La *unión* $G_3 = G_1 \cup G_2$ de dos gráficas $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ es otra gráfica $G_3 = (V_3, E_3)$ en la cual

$V_3 = V_1 \cup V_2$ y $E_3 = E_1 \cup E_2$

1.4.2 Intersección de dos gráficas.

La *intersección* $G_3 = G_1 \cap G_2$ de dos gráficas $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ es otra gráfica $G_3 = (V_3, E_3)$ en la cual

$V_3 = V_1 \cap V_2$ y $E_3 = E_1 \cap E_2$

1.4.3 Suma-Anillo de dos gráficas.

La *intersección* $G_3 = G_1 \oplus G_2$ de dos gráficas $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ es otra gráfica $G_3 = (V_3, E_3)$ en la cual

$V_3 = V_1 \cup V_2$
y si $e_i \in E_3$, entonces
$e_i \in (E_1 \cup E_2)$ y $e_1 \in (E_1 \cap E_2)$

1.4.4 Propiedades de las operaciones anteriores.

A partir de las propiedades de los conjuntos es fácil establecer:

Conmutatividad.

- $G_1 \cup G_2 = G_2 \cup G_1$
- $G_1 \cap G_2 = G_2 \cap G_1$
- $G_1 \oplus G_2 = G_2 \oplus G_1$

Idempotencia.

Para cualquier gráfica G:
$G \cup G = G \cap G = G$
Sin embargo
$G \oplus G$ =gráfica nula. La suma-anillo no cumple esta propiedad.

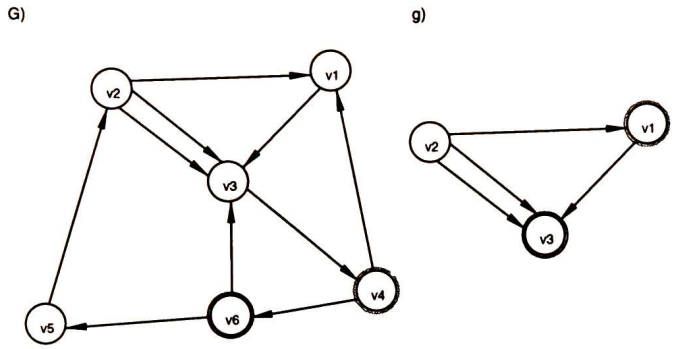

Figura 1.8: G y g.

Complemento de un subgráfica.

Supóngase la gráfica y subgráfica de la figura 1.8, entonces efectuando la suma anillo

$G \oplus g$

obtendremos la figura 1.9

$G' = G \oplus g$

o sea que G' es el subgráfica G , que resulta de suprimir de G todas las ramas de g. Resultando finalmente el complemento relativo de G.

Descomposición de gráficas.

Sean dos subgráficas de G: g_1 y g_2, entonces se dice que G se ha *descompuesto* en las dos subgráficas g_1 y g_2 si

$g_1 \cup g_2 = G$

$g_1 \cap g_2$ =gráfica nula.

Supresión.

Si en una gráfica se suprime el vértice v_1, y consecuentemente todas las ramas que en él inciden, se obtiene $G - v_1$ que es una gráfica de G. Si en una gráfica G se suprime una de sus ramas e_j, lo cual no implica la supresión de los vértices correspondientes, se obtiene $G - e_j$.

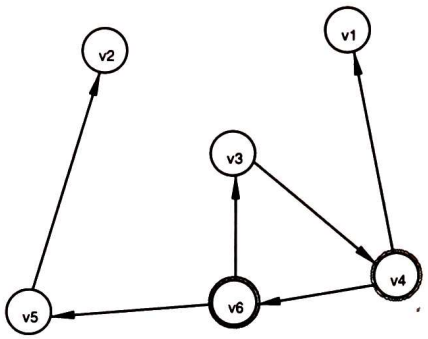

Figura 1.9: Gráfica complemento.

Fusión.

Un par de vértices v_i,v_j se fusionan si v_i y v_j se sustituyen por un único vértice $v_{i,j}$ tal que todas las ramas que incidían en v_i o en v_j inciden ahora en $v_{i,j}$.

1.4.5 Operaciones relacionadas con matrices.

Por otro lado, al definir la matriz topológica, podemos definir varias operaciones [Kol86, pag 73].

Junta.

Si tenemos dos matrices A y B de $n \times n$. Se puede definir la operación $A \vee B = [c_{ij}]$, conocida como la *junta* de A y B:

$$c_{ij} = a_{ij} \vee b_{ij}$$

Reunión.

Del mismo modo se define la operación $A \wedge B = [c_{ij}]$,como la *reunión* de A y B, como:

$$c_{ij} = a_{ij} \wedge b_{ij}$$

Finalmente se puede definir un producto booleano $A \odot B$, como:

$$c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \ldots \vee (a_{ip} \wedge b_{pj})$$

Estas operaciones son útiles para modificar la gráfica. Por ejemplo, si tuviéramos la matriz

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Y deseáramos "desconectar" los arcos que salen del nodo 3, basta con multiplicar la matriz con otra que sea como la identidad pero con toda la columna 3 en ceros

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Obteniendo

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Sin embargo estas operaciones son algorítmicamente caras, es decir del orden $|V|^2$.

1.5 Tipos de gráficas.

Las propiedades de las relaciones que definen la conexión entre los nodos de una gráfica, dan lugar a variedades de gráficas que describiremos a continuación.

1.5.1 Gráficas de redes de orden parcial.

Una gráfica de orden parcial es aquella que tiene las siguientes propiedades:

- Reflexividad. Un nodo esta relacionado consigo mismo
- Antisimetría. Si un nodo A está relacionado con un nodo B y viceversa, ambos nodos coinciden
- Transitividad. Si un nodo A está relacionado con un nodo B y éste a su vez con uno C, entonces A está relacionado con C.

se observa que estas propiedades aseguran que no hay ciclos en este tipo de gráfica.

1.5.2 Gráficas de equivalencia.

Una gráfica de equivalencia tiene las siguientes propiedades :

- Reflexividad. Un nodo esta relacionado consigo mismo
- Simetría. Si un nodo A está relacionado con un nodo B entonces B está relacionado con A.
- Transitividad. Si un nodo A está relacionado con un nodo B y éste a su vez con uno C, entonces A está relacionado con C.

las propiedades aseguran que cada uno de los vértices del grafo estára relacionado con cualquier otro, teniendo una especie de "equivalencia".

1.6 Sumario.

En este capítulo hemos dado un vistazo general de algunos aspectos sobresalientes de la teoría de gráficas, los conceptos más básicos y las representaciones y operaciones más típicas. Eso nos ayudará en los próximos capítulos, a describir tanto al editor como el contexto que lo rodea.

Capítulo 2

Problemas en la edición de gráficas

2.1 Introducción

Despueś de ver los fundamentos en el capítulo anterior; los tipos diversos, las representaciones y las operaciones sobre gráficas; cada aspecto implicará cierta clase de problemas inherentes. Esta clase de problemas dependerá del modo en que se defina la relación de la gráfica y las propiedades que aplican a la misma. Cada tipo de gráfica tendrá problemas bien específicos en su edición, aunque ciertamente existirán problemas comunes en varias gráficas. En este capítulo describiremos, en la medida de lo posible, estos problemas, de manera particular para cada gráfica, finalizando con un resumen de las problemáticas.

2.2 Gráficas con problemas de edición.

A pesar que no todas las gráficas tienen problemas al manipularse, existe un grupo importante de gráficas relacionadas con contextos específicos, que vale la pena analizar con un poco más de cuidado. Dentro de los contextos que dan pie a gráficas con problemas de edición están los autómatas celulares, que estudiaremos a continuación.

2.2.1 Automátas Celulares.

Un ecosistema está formado por varios seres y desde un punto de vista determinista, la situación futura del ecosistema dependerá exclusivamente de su estado actual, así, sabiendo el estado de cada ser, y el de los que interactúan con él, podemos saber el estado del ser en el futuro inmediato. Realizando la analogía con los autómatas celulares; cada uno de los seres del ecosistema es una *célula*; al conjunto de seres que rodean a cada ser y que con él determinan el estado de este último se denomina *vecindad* de la célula; la relación entre el estado de un individuo y el de su vecindad se denomina *regla de evolución*; finalmente si suponemos que en el ecosistema los cambios se dan en forma discreta y al mismo instante de tiempo podemos llamar a cada cambio como una *evolución.*

Los autómatas celulares nacen como un modelo de autoorganización en el cual se puede modelar la reproducción. Esta abstracción matemática consiste [McI87] en la definición de un conjunto definido de entes llamados células y una función conocida como la regla de evolución. Las células pueden estar dispuestas de muchas formas, la más simple es la lineal, es decir una célula tras otra, como se muestra:

1	1	2	0	1

Figura 2.1: Células consecutivas de un autómata

Cada célula tiene asociado un estado (un número entero). En cada transformación o *evolución*, todos los estados de cada célula transitan a otro estado. Este estado viene determinado en función del estado anterior de la célula y los de las células que la rodean. Al conjunto formado por la célula y las que la rodean se le conoce como *vecindad.* El tamaño de la vecindad se especifica por el *radio de la vecindad*, es decir, el número de células que hay entre la célula y la célula en el límite de la vecindad.

La *regla de evolución* es la función que determina el estado de cada célula en cada evolución. La regla de evolución depende de la vecindad correspondiente a la célula.

Definamos un autómata de ocho células, linealmente dispuestas con solo dos posibilidades por cada estado (0 ó 1). La disposición inicial de estados de la células es arbitario. Podemos definir la siguiente disposición:

10110111

Y definir una regla de evolución de radio uno; es decir, que el estado de cada célula depende de ella misma y del estado de una célula de cada lado; la regla tiene la siguiente forma:

Estado anterior de la vecindad			Estado actual
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabla 2.1: Regla de evolución

Definida la regla de evolución y la configuración inicial podemos obtener la siguiente evolución como se muestra en la figura 2.2. Al ver este ejemplo, podemos notar que no es claro cómo definimos los estados de las células de las orillas, dado que no tienen células adyacentes. Para solucionar este problema suponemos que las células de los extremos se unen formando un anillo de células.

Del mismo modo en que la configuración original se transformó en otra mediante la regla de evolución. Podemos aplicar esta última, una y otra vez sobre la configuración resultante *ad infinitum.*

A pesar que en un autómata se conoce el comportamiento local (es decir de una evolución a otra) , no se sabe como es su comportamiento

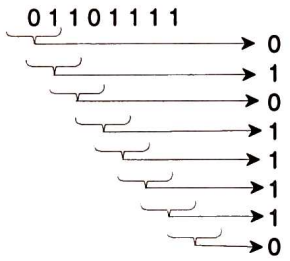

Figura 2.2: Evolución de un autómata.

a largo plazo. Es por esta razón que es útil cualquier herramienta que nos auxilie en el análisis del comportamiento a largo plazo. Gráficas que sirven para lograr ese fin, son:

- Diagramas de de Bruijin.
- Diagramas de subconjuntos.

2.2.2 Diagramas de de Bruijin.

En específico, para encontrar secuencias de células que se repiten al paso de las evoluciones, y como las secuencias de células involucran varias vecindades consecutivas, nos vemos en la necesidad de estudiar los traslapes entre vecindades. Los traslapes (figura 2.3) son las regiones comunes entre dos vecindades consecutivas.

Si formamos una gráfica; en la cual cada posible traslape es un nodo y cada vecindad, formada por dos traslapes consecutivos, es cada uno de los arcos; tendremos lo que se conoce como un diagrama de de Bruijin (ver figura 2.4).

Un diagrama de de Bruijin nos permite observar rápidamente qué vecindades están adjuntas en el autómata. A modo de ejemplo, de la figura 2.4, tomemos la ruta que empieza en el traslape 20 y termina en el 21. Esta ruta muestra que las vecindades 202, 022, 221 además de ser adyacentes, determinan la vecindad que se generará precisamente debajo de ellas. Con el fin de observar en el diagrama de De Bruijin, los estados que se generarán en la siguiente evolución, podemos extender

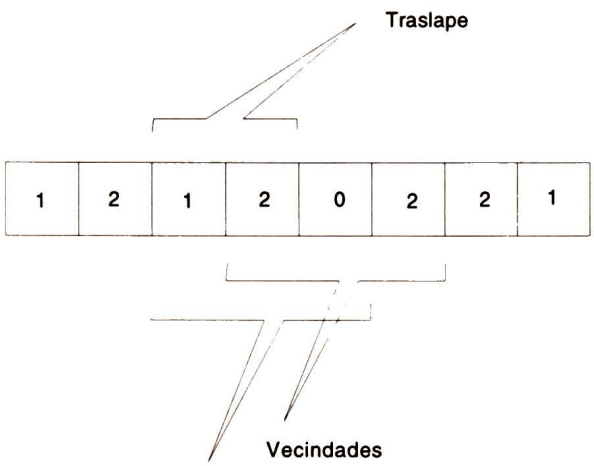

Figura 2.3: Traslape entre vecindades.

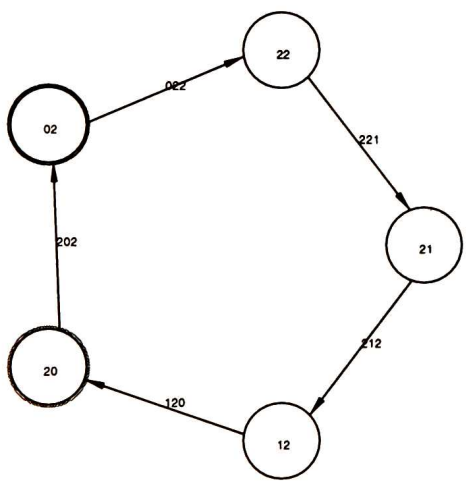

Figura 2.4: Diagrama de Bruijin de una vecindad.

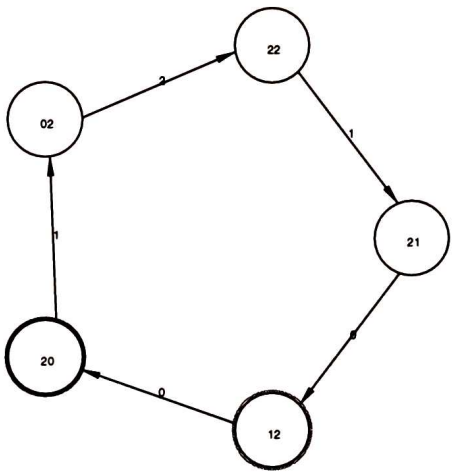

Figura 2.5: Diagrama de Bruijin con transición para cada vecindad.

el modelo inicial. La extensión consiste en asociar a cada vecindad el estado hacia el cual transita. Por ejemplo, si la regla de evolución indica que la célula 1 dentro de la vecindad 212 se convierte en 0 en la siguiente evolución, en la etiqueta del diagrama quedará 0 en lugar de la etiqueta 212 , como se muestra en la figura 2.5.

Esta modificación tiene algunos aspectos interesantes, permite conocer si una secuencia dada de células puede aparecer en la evolución del autómata, únicamente verificando si a través de un camino en la gráfica, la secuencia aparece. En efecto, en el diagrama de la figura 2.5 podemos visualizar que el camino 1-2-1 , que comienza en el traslape 20, aparece en la evolución del automáta, dado que lo generan las secuencia 20221 formada por las vecindades 202, 022 y 221.

2.2.3 Diagramas de subconjuntos.

Al describir los diagramas de de Bruijin comentábamos que una de sus ventajas radicaba en poder servir como un medio para comprobar la aparición de una secuencia de estados en alguna evolución. Sin em-

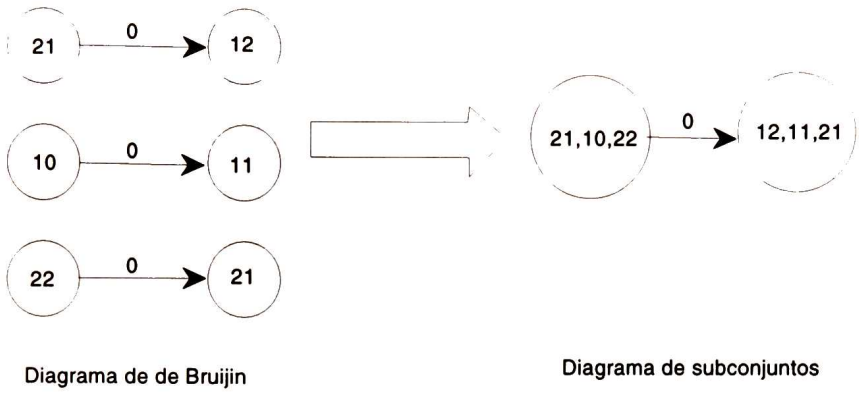

Figura 2.6: Relación entre diagrama de de Bruijin y subconjuntos.

bargo esta comprobación es poco práctica, porque no podemos saber de antemano en qué nodo comenzar a verificar. El diagrama de subconjuntos nos resuelve este problema. Un diagrama de subconjuntos es una gráfica cuyo nodos son todos los subconjuntos posibles de los traslapes (nodos del diagrama de de Bruijin). En este tipo de diagrama, cada subconjunto (es decir cada nodo) estará ligado con aquel subconjunto formado por los traslapes hacia los cuales transitan los traslapes del subconjunto inicial. Por ejemplo en la figura 2.6 las ligas del diagrama de de Bruijin se ubican a la izquierda y la liga resultante en el diagrama de subconjuntos se localiza a la derecha. Un diagrama de subconjuntos es una transformación del diagrama de de Bruijin que ha convertido cada vereda en algo único. Un diagrama de subconjuntos típico se muestra en la figura 2.7, en el diagrama; A, B, C y D son los traslapes entre vecindades, los mismos que aparecen en el diagrama de de Bruijin.

En un diagrama de subconjuntos cada ruta empieza en el subconjunto mayor, y va conectándose a nuevos subconjuntos, eventualmente si la ruta llega al conjunto vacío, significa que la vecindad formada por la ruta, no podrá surgir de ninguna concatenación de traslapes, es decir que solo podría aparecer como configuración inicial para nunca más reaparecer. A este tipo de configuración se le conoce, por esta propiedad,

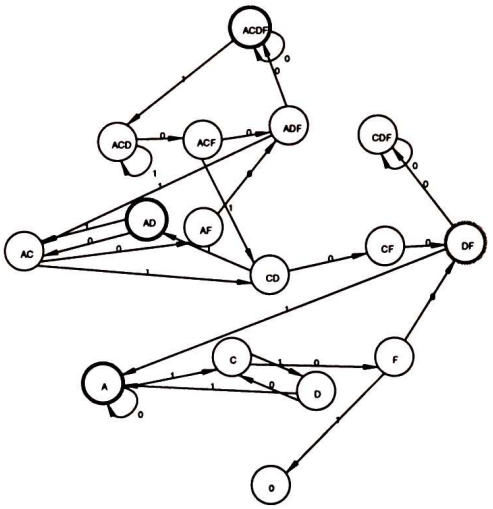

Figura 2.7: Diagrama de subconjuntos

como un jardín del Edén.

Un diagrama con tres estados posibles y una vecindad de radio 1 es un diagrama muy grande. Con el único fin de hacer evidente este hecho, hemos creado una gráfica con la mitad de nodos que el diagrama real, y con un número de arcos aproximadamente igual, éste se muestra en la figura 2.8. Sale sobrando decir que la gráfica no es demasiado manejable, el acomodo de la gráfica es tardado y visualizar secciones que nos interesan es prácticamente imposible.

2.3 Problemas de edición y manipulación.

En esta sección se discutirán diferentes problemas que surgen en el contexto de la edición y manipulación de gráficas, proporcionando un panorama de su diversidad.

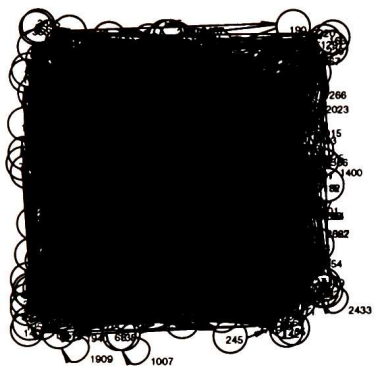

Figura 2.8: Gráfica muy grande.

Destacando orden y equivalencia en la representación visual.

La estética de la gráfica a primera vista pudiera parecer trivial, pero si una gráfica no está esquematizada de un modo adecuado no podremos localizar patrones en la gráfica. Entonces la visualización de la gráfica, requiere de un reacomodo de la gráfica a distintos niveles de detalle, en el ámbito de toda la gráfica o de un subgráfica. Para acomodar una gráfica es necesario saber que en la misma existen varias características a destacar. De ahí que lo que nos interesa sea investigar, cómo vamos a destacar la característica en cuestión. Igualmente, es importante evitar el amontonamiento de nodos así como el cruce de líneas.

El acomodo de una gráfica es un proceso complejo. Por ejemplo si tenemos una gráfica de redes de orden parcial, entonces el acomodo de esta gráfica tiene que poner especial cuidado en qué nodos aparecerán más arriba o más abajo con respecto a otros. Primero deberán aparecer aquellos nodos que son menores que otros, o bien mayores.

En la figura 2.9, se puede observar la relación con cierta claridad, A es menor que B y éste a su vez menor que C. Por otro lado, si se trata de una relación de equivalencia, lo que realmente importa, es observar cada clase de equivalencia de manera independiente. En forma práctica esta relación puede mostrarse en varias ventanas. Como todos los nodos dentro de una clase de equivalencia están relacionados con otros nodos, lo mejor es ordenarlos, dentro de un polígono regular

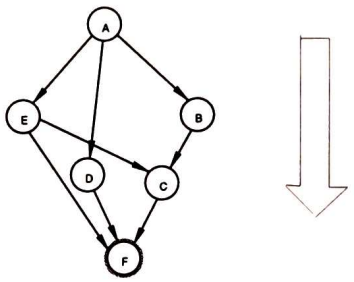

Figura 2.9: Las relaciones van de abajo hacia arriba.

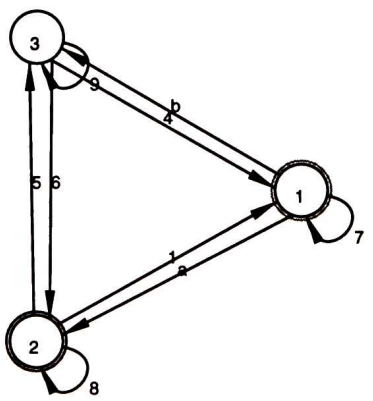

Figura 2.10: Clase de equivalencia.

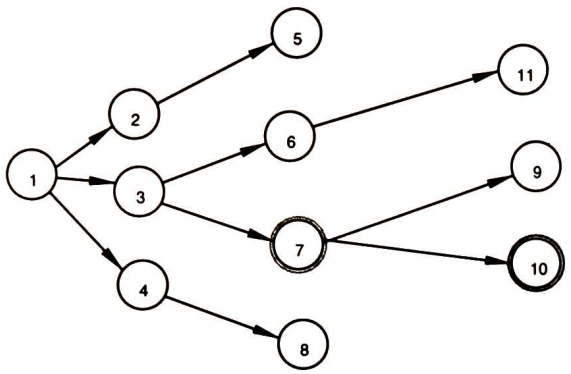

Figura 2.11: Abanico

(ver figura 2.10), con el fin de poder observar mejor los valores de las conexiones. Sobretodo cuando se trata de gráficas pequeños.

El editor debe reflejar las propiedades que aplican a las relaciones de la gráfica que se está editando con el fin de auxiliar al usuario del sistema a decidir que visualización le conviene ver.

Algunas posibilidades para destacar el orden y las relaciones de equivalencia entre gráficas desde un punto de vista geométrico, son:

- *Abanico.* Podemos dividir la gráfica en estudio, en varios árboles que los cubren y estos árboles pueden servir de "esqueletos" de nuestra gráfica. Ahora bien cada árbol es susceptible a un acomodo en abanico, como se puede apreciar en la figura 2.11.

- *Candelero.* En este acomodo el ciclo mayor de la gráfica tiene forma de un círculo en el centro de la gráfica, permitiendo visualizar la estructura mayor de la gráfica y a los alrededores del círculo los subgráficas menores, (ver 2.12).

- *Polígono.* En este acomodo todos los nodos son dispuestos en un polígono regular, (ver fig. 2.13).

- *Retícula.* En este acomodo todos los nodos son dispuestos en "filas" , como si se tratara de un red de pescar, (ver fig. 2.14).

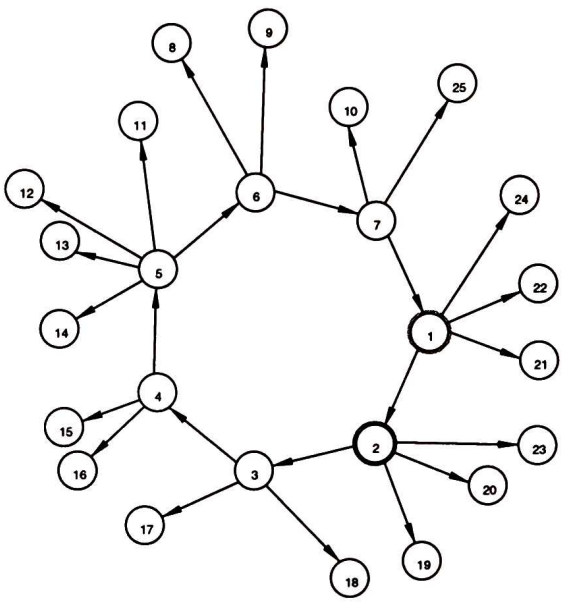

Figura 2.12: Candelero.

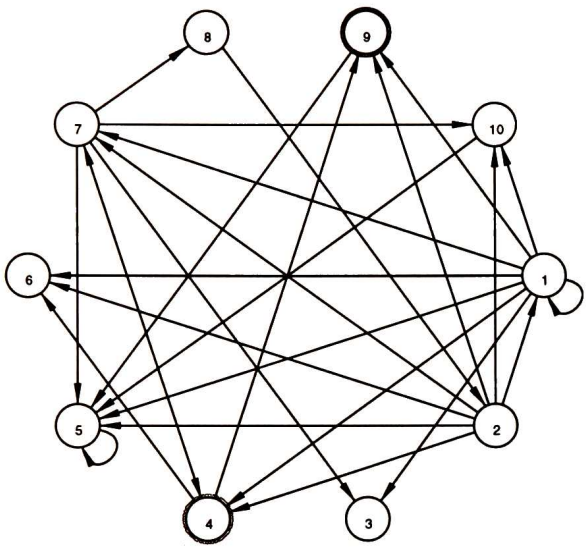

Figura 2.13: Polígono.

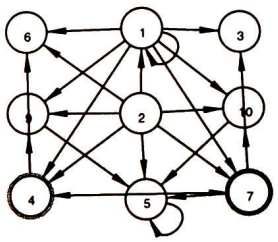

Figura 2.14: Retícula.

Exceso de nodos.

Cuando en una gráfica tenemos una cantidad enorme de nodos (vgr. 5000), se complica la visibilidad. Por decirlo de algún modo "los árboles no nos permiten ver el bosque", dificultando cierta clase de operaciones como:

- *Hacer seguimientos de rutas a partir de un nodo.* Como se aprecia en los diagramas de subconjuntos y de de Bruijin. El hallazgo de una ruta larga o la comprobación de la existencia de una ruta en una gráfica se dificultan bastante.

- *Localizar un nodo en particular.* Para ubicar o referirse a un nodo, no contamos con la ubicación fisica del nodo, en su lugar se puede utilizar la etiqueta del nodo o las relaciones de ese nodo con sus vecinos.

- *Operaciones en grupo.* Así como no se puede localizar fácilmente un nodo, la ubicación de una subgráfica tiene un reto mayor, porque no es posible o es muy difícil enumerar cada uno de los nodos que forman la subgráfica. Por lo tanto, la forma de ubicación debe requerir de un patrón, o bien ubicar la subgráfica paulatinamente.

- *Conocer la estructura interna de la gráfica.* Si vemos un libro como una secuencia de letras, es difícil percibir una estructura, por lo que es necesario elevar el nivel de abstracción. En las gráficas grandes hablamos exactamente del mismo problema. Enfoques para subir el nivel de abstracción son:

  - Englobar o sustituir una subgráfica por un nodo que las represente. Este enfoque permite ver una estructura jerárquica de la gráfica, pero tiene el inconveniente de no poder observar las relaciones "horizontales" entre los elementos de la gráfica.
  - Acercamientos y desplazamientos (scrolls). Como los biólogos que visualizan bajo el microscopio un tejido, solo ven una parte del tejido y analizan los tipos de componentes internos. Inversamente a lo que sucede en el punto anterior aquí

se observa las relaciones al detalle, pero se pierden las relaciones globales.

Elementos importantes en la implantación de herramientas que manejen gráficas grandes es la eficiencia algorítmica, ya que después de localizar una gráfica, generalmente se realiza una operación con toda la gráfica, y si ésta es grande, cobra gran importancia la eficiencia de los algorítmos. Cada algoritmo debe minimizar la cantidad de recorridos sobre la gráfica.

2.4 Sumario.

La cantidad de nodos, la densidad de arcos por nodo, la dirección de cada flecha y las propiedades de las relaciones de la gráfica, implican conflictos en la edición de una gráfica. Por otro lado, la persona que estudia una gráfica, quiere destacar un aspecto de la misma, para verla mejor. En este capítulo se ha propuesto una clasificación de los problemas de manipulación y editado de gráficas, del mismo modo, los aspectos a destacar en una gráfica, con el fin de conocer el contexto en que se dará la edición de gráficas.

Capítulo 3

Edición de gráficas

3.1 Introducción.

Editar una gráfica significa poder seleccionar secciones de la misma, para posteriormente realizar copiados, pegados, acomodos y mezclas, entre otras operaciones. En general, editar una gráfica conlleva varios retos:

- La versatilidad en las operaciones de modificación y visualización de gráficas.
- Debe hacer de la edición, un proceso rápido y sencillo.
- Minimizar el número de pasos para manipular una gráfica.

Sin embargo, como se mostró en el capítulo 2, para algunas gráficas no es fácil, seleccionar un nodo, referirse a una ruta en particular o bien acomodar la gráfica en la disposición que se desea. Es por esta razón que este capítulo está dedicado a mostrar el proceso de edición por medio de la definición de un lenguaje gráfico, dando solución a diversas categorías de problemas de edición expuestos en el capítulo anterior.

3.2 LGRAF: un lenguaje de consulta para gráficas.

Editar una gráfica significa definir con exactitud cada operación y su encadenamiento con otras operaciones. En la medida en que la definición es sencilla, la edición es más flexible. Vale la pena entonces, definir de manera precisa un lenguaje en el que se puedan expresar operaciones de edición y consulta.

Este lenguaje lo hemos llamado LGRAF. LGRAF tiene como expectativas los siguientes puntos:

- Debe ser sencillo.
- Debe contar con una gramática que permita expresar un conjunto rico de consultas y ediciones sobre una gráfica, que pueden ser desde muy detalladas hasta muy complejas.
- La grámatica debe ser sencilla de escribir y entender.
- Debe ser lo suficientemente formal como para poder realizar un intérprete con él.
- Debe permitir manejar diferentes niveles de abstracción.
- Debe ser fundamentalmente gráfico.

3.2.1 Representaciónes básicas.

Representando tipos básicos.

LGRAF tiene como elementos principales:

- Variables. Una *variable* es un tipo de elemento de gráfica (vgr. nodo) o bien un tipo de magnitud (vgr. flotante), la cual puede tomar diversos valores concretos. LGRAF consta de varios tipos de variables y símbolos que los representan, ambos se describen en la tabla 3.1.

Símbolo	*Tipo de variable*
G	Gráfica
A	Arco
N	Nodo
R	Ruta
#	Número
X	Alfanumérico
C	Color

Tabla 3.1: Símbolos para los distintos tipos de gráficas.

- Procesadores. Definimos como *procesador*, un ente que puede transformar un conjunto de variables en otro, con la capacidad de generar *eventos* y ser controlado por medio de *acciones.*

  Un procesador es representado por un rectángulo, con el nombre del procesador anotado en la parte superior, ya sea dentro o fuera del rectángulo. El lado izquierdo del rectángulo recibe las *variables de entrada* y el lado derecho genera *las variables de salida.* En el lado izquierdo se escriben los tipos de las variables de entrada y en el lado derecho los tipos de variables de salida.

- Conexiones. Las *conexiones* son las flechas que salen y entran, de y hacia los procesadores, las cuales representan la salida y entrada de argumentos, respectivamente. En el caso de entrada o salida de variables booleanas, las conexiones se representarán con líneas punteadas.

  Las conexiones sirven para establecer cuál es el intercambio de variables entre procesadores. Opcionalmente, se puede asociar a cada conexión el nombre de la variable que se está mandando o recibiendo. En ocasiones, la salida de una variable, entra a dos o más procesadores, en este caso la línea se puede subdividir.

Para ejemplicar la notación, en la figura 3.1 se representa la selección de un nodo por medio de su etiqueta. En esta selección se recibe una gráfica (g) , una etiqueta (etiqueta1), y se genera un nodo(n).

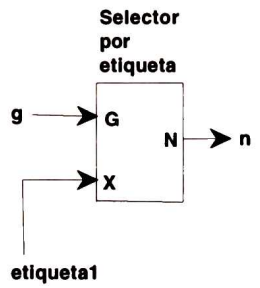

Figura 3.1: Símbolo de selección de un nodo de una gráfica por su etiqueta.

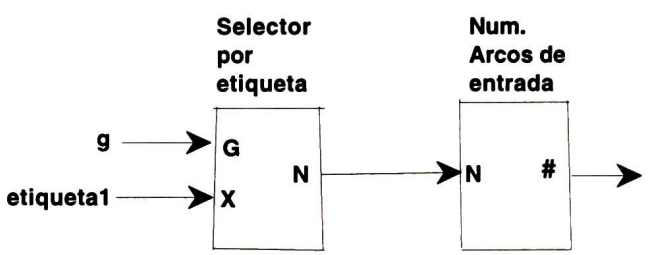

Figura 3.2: Símbología de un procesador encadenado.

Conectando procesadores.

Un procesador puede conectarse con otro con el fin de realizar algo más elaborado. Por ejemplo, una vez que se ha seleccionado un nodo mediante un procesador, se desea saber cuántos arcos tiene de entrada, en cuyo caso, la salida del primero se conecta con el segundo, como se observa en la figura 3.2. La flecha que conecta a los dos procesadores, simboliza el paso de variables, entre un procesador y otro. Es importante notar que las variables deben ser del mismo tipo. Esta propiedad siempre se va a cumplir entre procesadores cualesquiera de una consulta o edición.

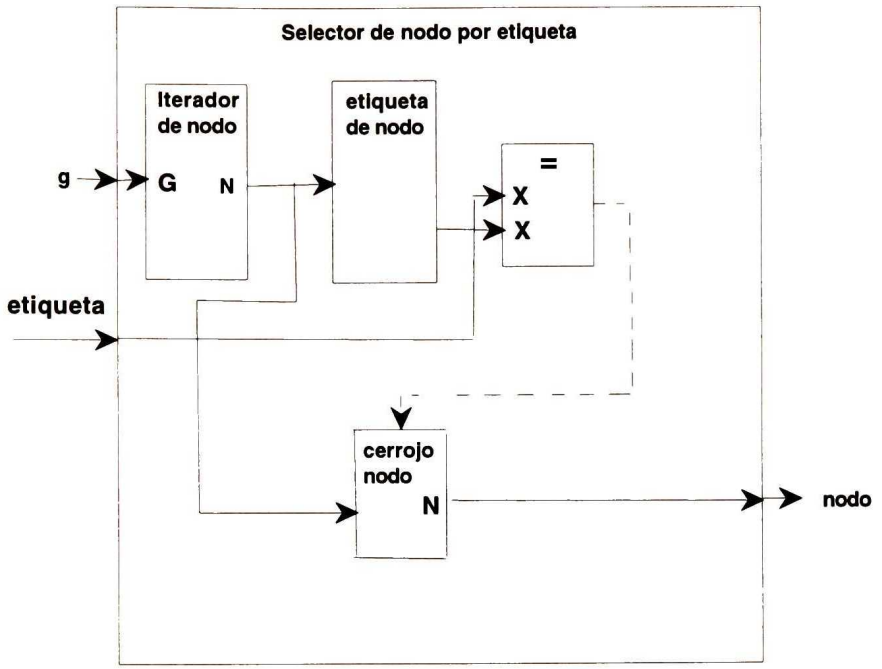

Figura 3.3: Símbología de un procesador compuesto.

Construyendo procesadores compuestos.

Además de poder conectar procesadores uno tras otro, se pueden definir macroprocesadores, es decir, procesadores que están compuestos de otros procesadores más simples, como se puede apreciar en la figura 3.3. Esta propiedad del lenguaje permite ampliar la variedad de procesadores y el refinamiento de las capacidades de los mismos.

Sincronizando procesadores.

En este lenguaje se puede especificar control, es decir el orden en que se necesita que los procesadores ejecuten sus acciones. Definiremos una *señal* como un tipo especial de conexión que indica mediante un valor booleano, la realización (1) o no realización (0) de un evento en un procesador, la realización dispara a su vez, una acción en otro

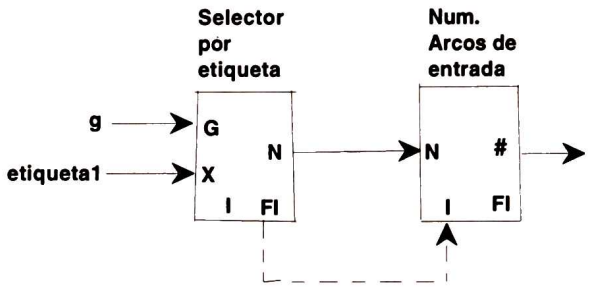

Figura 3.4: Operaciónes en secuencia.

procesador, así la señal es la pareja *(evento, acción)*. Cada procesador tendrá definido, al menos, la acción *iniciar (I)*. Si un procesador tiene una acción con nombre *accionX*; al finalizar la misma, generará un evento con nombre *FaccionX*. Por ejemplo, para la acción I, el evento de finalización es FI.

Los símbolos para un evento los anotaremos preferentemente arriba o abajo del borde del rectángulo.

De este modo dos procesadores pueden ser ejecutados secuencialmente, si hay una señal (FI ,I) del primer procesador al segundo. Una señal se representa como una línea punteada que parte del símbolo del evento en un procesador, al símbolo de acción de otro, como se puede observar en la figura 3.4.

Ahora bien, el inicio de algunas procesadores requieren de la finalización de un conjunto de procesadores o de una combinación de otras señales. Para resolver esta cuestión usaremos la notación gráfica para los operadores lógicos AND, OR y NOT de la figura 3.5.

Un ejemplo del uso de las compuertas se muestra en la figura 3.6. Ahí, estamos condicionando la ejecución de *Iterador de ruta* a la finalización de los otros dos procesadores.

Definiendo constantes.

A menudo es necesario establecer de antemano una magnitud fija para las conexiones, esto se expresa en la notación gráfica anteponiendo a la

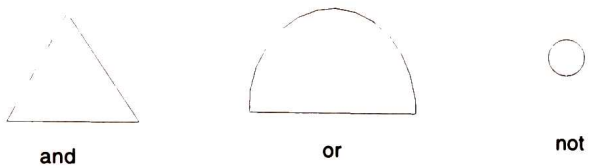

Figura 3.5: Compuertas lógicas.

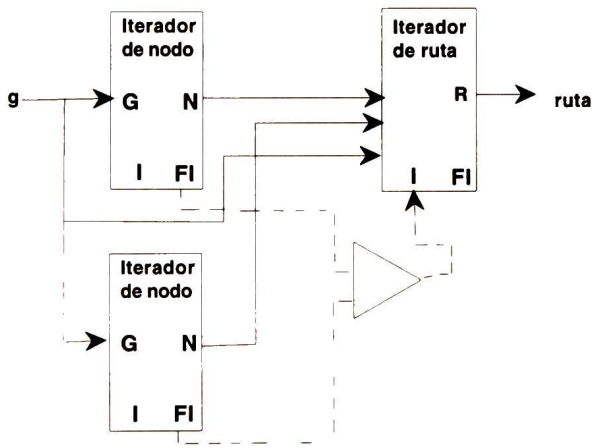

Figura 3.6: La compuerta AND para condicionar la ejecución.

flecha que simboliza a la conexión, la magnitud. Ciertamente, la flecha no puede partir en este caso de un procesador.

Definiendo condiciones.

Frecuentemente se desean hacer transferencias de control condicionales. Una condición es un procesador que genera un valor booleano. Las condiciones con que cuenta este lenguaje gráfico son :

- Predicados. Los predicados son procesadores que generan un valor booleano, pero que reciben elementos de la gráfica o magnitudes. Ejemplos de predicados son :
  - *EsAdyacente* muestra si un nodo es adyacente a otro.
  - *PerteneceAlGrafo* muestra si un nodo pertenece a una gráfica.
  - *PerteneceAlGrafo* muestra si un arco pertenece a una gráfica.
  - *EsNodoOrigen* muestra si el nodo es el origen de una ruta.
  - *EsNodoDestino* muestra si el nodo es el destino de una ruta.
  - *EsRuta* muestra si la gráfica es una ruta o no.
- Condiciones u operadores relacionales.
  - $>$ Mayor que.
  - $<$ Menor que.
  - $=$ Igual a
- Condiciones u operadores booleanos.
  - *and* Y.
  - *or* O.
  - *not* Negado.

Los operadores booleanos y relacionales sólo están definidos entre dos elementos o magnitudes del mismo tipo. Un ejemplo de la notación gráfica para las condiciones, se muestra en la figura 3.7.

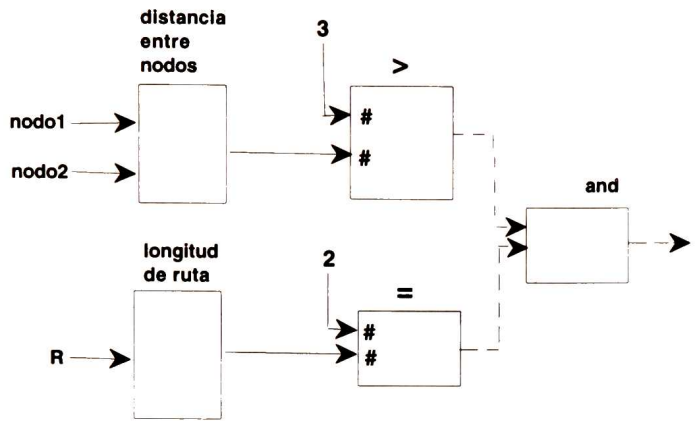

Figura 3.7: Ejemplo de condiciones.
Si la distancia entre el nodo 1 y el nodo 2 es mayor que 3, y la longitud de la ruta 1 es igual a 2.

Medidores.

A menudo se necesita obtener alguna magnitud numérica, alfanumérica de algún elemento de la gráfica o de la gráfica misma. Un medidor es un procesador que realiza esta acción. Medidores comunes son:

- *etiquetaNodo* devuelve la etiqueta de un nodo.
- *etiquetaArco* devuelve la etiqueta de un arco.
- *longitud* devuelve la longitud de una ruta.
- *numNodos* devuelve el número de nodos que contiene una gráfica en particular.

Las notaciones de estos medidores se encuentran en la figura 3.8.

3.2.2 Familias de procesadores.

Con el fin de tener el conjunto mínimo de elementos para la edición y manipulación de gráficas, se han clasificado los tipos de procesadores:

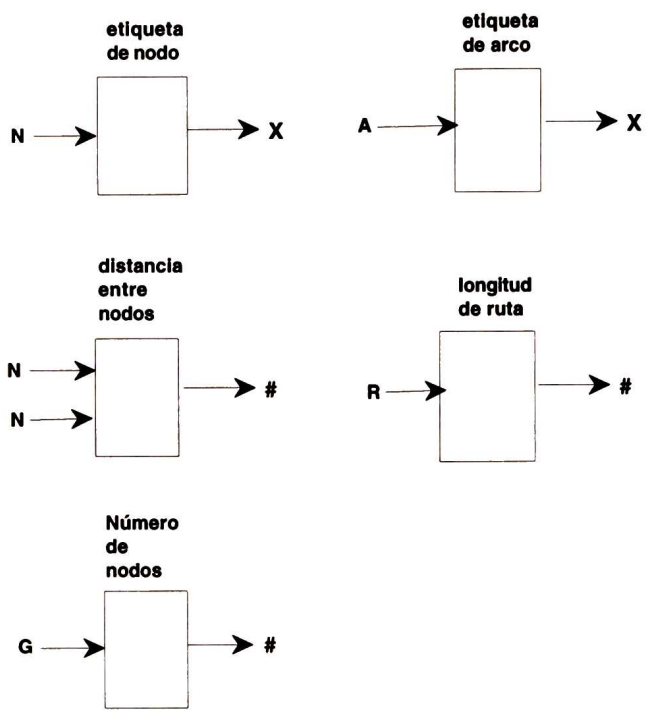

Figura 3.8: Representación de medidores.

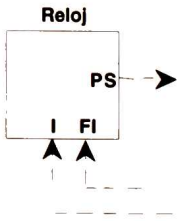

Figura 3.9: Representación del reloj.

- Procesadores de control.
- Procesadores con estado.
- Procesadores para creación.

los cuales describiremos a continuación.

3.2.3 Procesadores para control.

Esta serie de procesadores sirve para seleccionar un conjunto de variables dentro de otro conjunto, o bien seleccionar variables que cumplen o no cumplen una condición. Los tipos de procesadores que entran en esta categoría son:

- Relojes.
- Iteradores.
- Cerrojos.

Reloj.

Un reloj es un procesador que genera señales continuamente. Su objetivo es provocar que otros procesadores funcionen continuamente. Un reloj no se detendrá, a menos que se le mande a ejecutar la acción *apagar (OFF)*. El reloj emitirá una señal *pulso (PS)* continuamente. La representación gráfica del reloj se muestra en la figura 3.9.

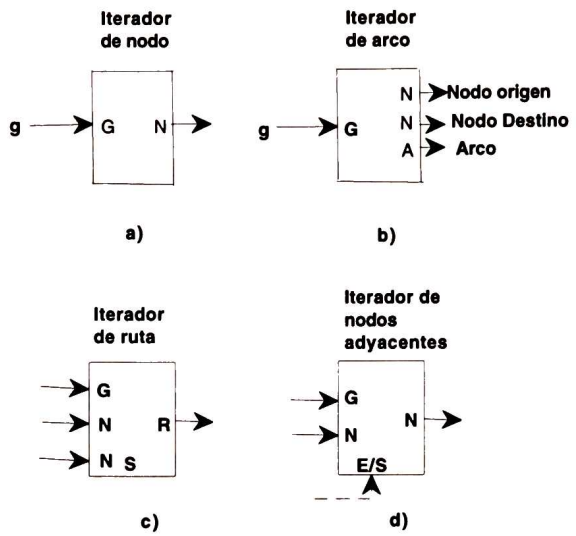

Figura 3.10: Representación de iteradores.

Iteradores en gráficas.

Este tipo de operación sirve para referirse nodo tras nodo o arco tras arco de una gráfica. Un iterador lo definimos como un procesador que genera valores sucesivos de un elemento en particular. Se le solicita al iterador que genere un valor, mediante la acción *Siguiente (Sg)*. Cuando el iterador ya no puede generar valores dispara un evento *Vacío (Vc)*. Distinguimos los siguientes iteradores :

- *Iterador de nodos.* Genera sucesivamente nodo tras nodo de una gráfica. En la figura 3.10-a se muestra un iterador de nodos.
- *Iterador de arcos.* Genera sucesivamente arco tras nodo, junto con su nodo origen y destino respectivos (vease figura 3.10-b).
- *Iterador de rutas.* Genera sucesivamente ruta tras ruta entre dos nodos, un nodo origen y un nodo destino (vease figura 3.10-c).
- *Iterador de nodos adyacentes.* Devuelve uno a uno, cada nodo adyacente a uno de los nodos (vease figura 3.10-d). Si el valor de

E/S es 1, se trata de nodos adyacentes de entrada, 0 si son nodos adyacentes de salida.

Es común anteponer a un iterador la señal de un reloj, la cual disparará la acción Sg en el iterador.

Cerrojos lógicos.

Si requerimos seleccionar un elemento de gráfica dependiendo de la certeza o falsedad de alguna condición requerimos de un cerrojo lógico. Un cerrojo lógico es un procesador que recibe dos argumentos, el primero es un valor booleano y el segundo es un elemento de una gráfica. El cerrojo dejará pasar el elemento de la gráfica sólo si el valor booleano es verdadero.

3.2.4 Procesadores con estado.

Este tipo de procesadores tienen la particularidad de poder recordar su "estado", es decir tienen memoria. Con el fin hacer más sencilla su comprensión, se han clasificado de acuerdo a las siguientes categorías:

- Memorias.
- Acumuladores.
- Diccionarios.

Memorias.

Es necesario en una variedad de consultas, recordar los valores de algunas variables y recuperarlos posteriomente. Una memoria es un procesador que cuenta con dos acciones:

- Guardar elemento de gráfica(W).
- Recuperar elemento de gráfica(R).

Acumuladores.

Un acumulador es un procesador que almacena variables de un mismo tipo y que se le pueden aplicar las siguientes acciones:

- Agregar nueva variable (Ac).
- Recuperar el mayor (My).
- Recuperar el menor (Mn).

Diccionarios.

Un diccionario asocia dos valores, uno de tipo numérico o alfanumérico y el otro, un elemento de gráfica. Con el diccionario podemos recuperar valores asociados previamente por medio de su llave. Las acciones básicas de un diccionario son:

- Guardar par llave-valor (Gp).
- Recuperar valor (Rv).

3.2.5 Procesadores para creación.

La serie de procesadores para creación, sirven para crear elementos de gráfica. Las notaciones para cada procesador se ubican en la figura 3.11. Describiremos las siguientes categorías :

- Creador de nodos.
- Creador de arcos.
- Creador de supernodos.
- Extractor de gráfica.
- Incorporador de nodos.
- Incorporador de ligas.

Todos los procesadores de creación tienen en común la acción *crear (I).*

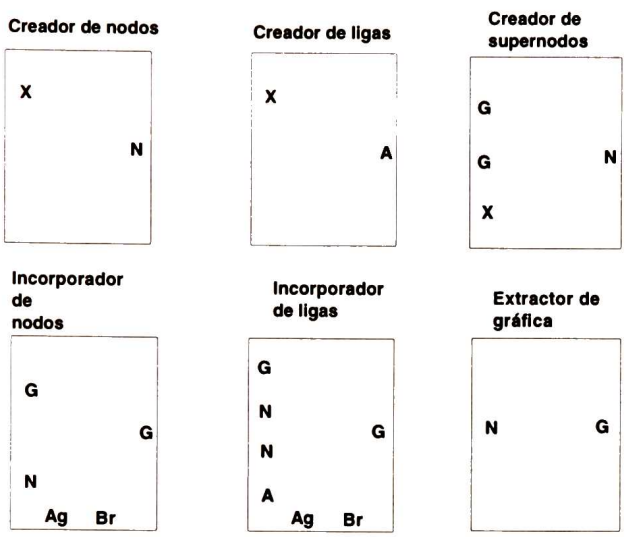

Figura 3.11: Representación de creadores.

Creador de nodos.

Un creador de nodos hace posible la fabricación de un nodo, por medio de su etiqueta.

Creador de arcos.

Un creador de arcos fabrica arcos por medio de la etiqueta del nuevo arco.

Creador de supernodos.

Los supernodos [Lin98] son nodos que agrupan subgráficas con el fin de hacer más operable la gráfica. Se fabrican en función de una subgráfica y una etiqueta.

Extractor de gráfica.

Con este procesador se puede recuperar la gráfica que agrupaba previamente un supernodo.

Incorporador de nodos.

Este procesador incorpora o desincorpora un nodo a o de un grafo, respectivamente. Tiene dos acciones *Agregar (Ag)* y *Borrar (Br)*, dependiendo de lo que se desee realizar.

Incorporador de ligas.

Este procesador incorpora o desincorpora una liga a o de un grafo, respectivamente. Tiene dos acciones *Agregar (Ag)* y *Borrar (Br)*.

3.3 Tipos de operaciones para la edición.

Las operaciones que se efectúan para editar una gráfica pueden caer en tres categorías:

- Selección de secciones. Por selección vamos a entender, el acto de hacer referencia a una subgráfica interesante para la manipulación, quizás porque sus nodos se encuentran en un área de despliegue común, porque son adyacentes a un nodo en particular o cualquier otro criterio.

- Manipulación de la gráfica. Al realizar la selección de una subgráfica; se puede, cambiar su ubicación, distribuir los nodos que forman la gráfica en cierto sentido, ocultar a la misma, desconectar una parte de la gráfica de otra; el cambio es la manipulación de la gráfica.

- Visualización de la Gráfica. Entendemos la visualización como la selección de la representación visual para la gráfica.

Las operaciones para la edición de la gráfica no solo se realizan en el orden Selección-Manipulación-Visualización. Porque bien podemos seleccionar, manipular y volver a seleccionar o bien hacer una secuencia

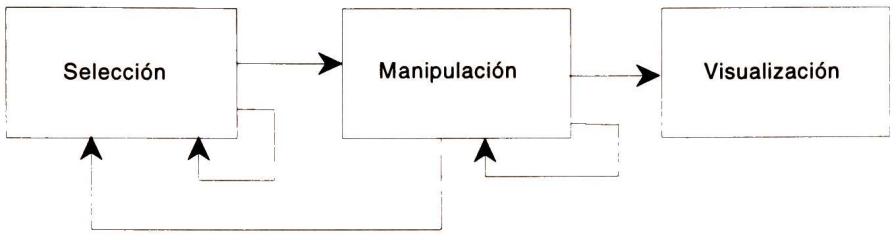

Figura 3.12: Interrelación entre los procesos de edición.
Las flechas indican la secuencia en que se pueden aplicar las operaciones.

más compleja. Las maneras en que se pueden combinar las operaciones vienen descritas en la figura 3.12.

3.3.1 Selección de gráficas.

Primero se describirá qué es la selección de gráficas y la manera como podemos expresar una selección en el lenguaje gráfico.
La selección o la consulta es cualquier operación que permite referirse a zonas específicas de una gráfica. Ejemplos de selecciones son las siguientes:

- ¿Cuál es la subgráfica con mayor número de ligas en relación a sus nodos?
- ¿Qué nodos están conectados con el nodo 25?

Una selección está determinada por dos elementos básicos:

1. El tipo de sección o elemento de la gráfica, que se desea recuperar de la gráfica, el cual puede pertenecer a una de las siguientes categorías:
   - Un nodo.
   - Un arco dado
   - Un subgráfica
   - Una ruta o vereda.

2. El criterio de selección. El criterio de selección puede ser muy variado y en el mismo participan entre otras variables:

   - Etiqueta
   - Posición visual
   - Conexidad con otros nodos.

   Vale la pena destacar que podemos hablar de dos criterios de selección:

   (a) La selección relativa que permite referirnos a alguna sección de la gráfica en función de otra sección, que puede ser obtenida mediante una selección previa.

   (b) La selección absoluta escoge una sección de la gráfica de una manera aislada.

Para realizar una selección requerimos en la mayoría de las ocasiones de la interconexión entre varios procesadores. Por ejemplo, si se deseara seleccionar todas aquellas rutas que parten del nodo1 y que tienen longitud 3, se tendría la disposición de procesadores de la figura 3.13. En esta figura, el reloj le envía al iterador de nodos un señal para que continuamente genere un nodo, este nodo generado junto con el nodo1, es utilizado para obtener cada ruta entre los dos nodos (iterador de rutas). Posteriormente se obtiene la longitud y se compara con el 3, en función de esta comparación se selecciona la ruta (cerrojo de ruta). El proceso termina cuando el iterador de nodos no tiene nada que generar y apaga el reloj.

3.3.2 Manipulación de gráficas.

Dentro de la manipulación distinguimos tres tipos básicos:

- Agregado y borrado de nodos o ligas dentro de una gráfica.
- Coloración.
- Acomodo.

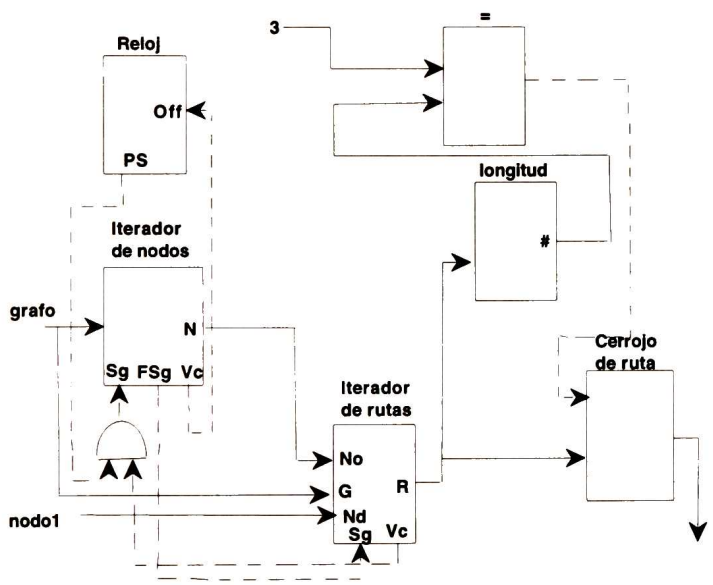

Figura 3.13: Ejemplo de selección con el uso de procesadores.

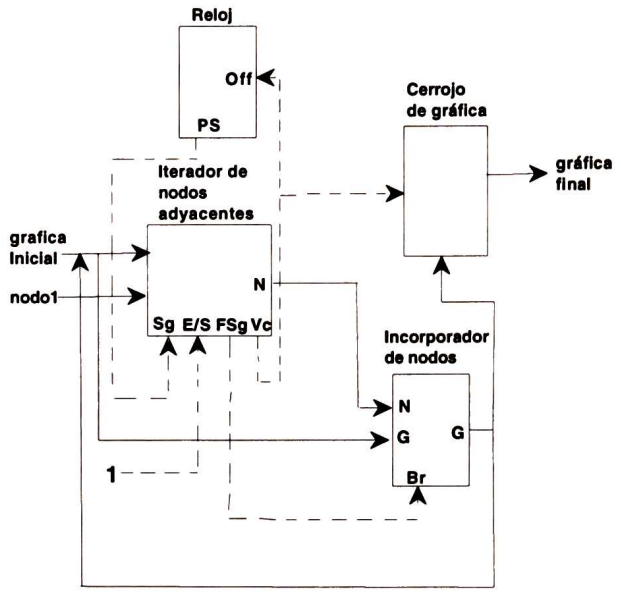

Figura 3.14: Borrado de todos los nodos adyacentes de entrada a nodo1.

El agregado y borrado de nodos están contemplados con los procesadores de creación. Un ejemplo de LGRAF para el borrado de todos los nodos adyacentes de entrada a un nodo, se observa en la figura 3.14. En la figura, el reloj le ordena al iterador que genere cada uno de los nodos adyacentes a nodo1. Conforme va obteniendo cada uno de los nodos adyacentes, los va borrando de la gráfica original. Cuando el iterador no tiene más nodos que generar, apaga el reloj y libera la gráfica final.

Procesadores de coloración.

La coloración permitió a los biólogos distinguir tejidos y organismos a través del microscopio. La coloración es importante porque la podemos utilizar como una dimensión extra. Si tenemos que graficar en un plano, el color nos da el volumen. Para solucionar el problema de coloración, tenemos los siguientes procesadores (vease 3.15):

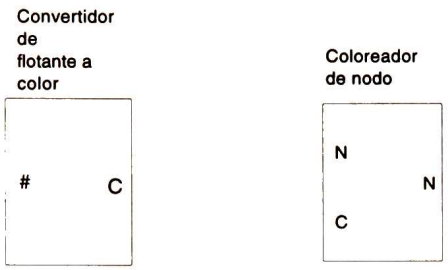

Figura 3.15: Procesadores para color.

- Convertidor de flotante a un color. Si queremos graduar el color en función de una magnitud, requerimos un mapeador de los números reales a un color. Esta es la función precisamente de este procesador.
- Coloreador de un elemento de gráfica. La acción de asignar un color a un elemento de la gráfica es la razón de ser de este procesador.

Procesadores de acomodo.

Una vez que se ha seleccionado un nodo o una gráfica de otra, es común acomodar el elemento seleccionado la gráfica en ciertas formas. Ésta es la tarea de los siguientes procesadores:

- Desplazador de gráfica. Desplaza a la posición *pos* de cada uno de los elementos de la gráfica G. Si el valor del *tipo de desplazamiento (ABS/REL)* es 1, el desplazamiento es relativo y 0 es absoluto.
- Acomodador en polígono. Acomoda la gráfica G en forma de un polígono con centro *centro* y radio *radio.*
- Acomodador en abanico. Acomoda la gráfica G en forma de un abanico con un ángulo de expansión *ángulo* centro *centro* y radio *radio.*
- Acomodador en retícula. Acomoda la gráfica G en forma de una retícula empezando en *pos.*

3.3.3 Visualización de gráficas.

La visualización de una gráfica tiene como objetivo seleccionar el tipo de representación que se quiere. Distinguimos tres tipos principales de representaciones:

- Explorador (browser). En esta representación tenemos una estructura de árbol, como la de un directorio, en la cual podemos ver las etiquetas de la gráfica y si hay gráficas colapsadas en un supernodo, su contenido respectivo.
- Matriz de conectividad. Esta representación, solo muestra si hay conexión entre un nodo y otro.
- Visual. En esta representación se muestran esquemáticamente cada nodo y arco, es la representación más típica de una gráfica.

Las procesadores para despliegue correspondientes a los tres tipos de representaciones son las siguientes:

- Explorador. Visualiza una gráfica G con supernodos en un browser, que permite ver subgráficas dentro de subgráficas.
- Visor-Matriz. Visualiza una gráfica G en su forma matricial.
- Visor-Ventana. Visualiza en una ventana, la gráfica G.

Supongamos que en el ejemplo de la figura 3.14 lo hemos encapsulado en una macrooperación llamada *Borrador de nodos adyacentes* y posteriormente se desea visualizarla en forma de una matriz de adyacencia, la representación resultante se observa en la figura 3.16.

3.4 Editor de gráficas.

Las diferentes primitivas para el soporte de gráficas con problemas de edición, se han implantado en el editor de gráficas de propósito general. Este editor tiene en su código las operaciones que pueden servir de base para la construcción de un ambiente que soporte LGRAF. En este editor de gráficas las operaciones se acceden por medio de menús y paneles de herramientas, lo que lo hace fácil de utilizar. Describiremos a continuación las operaciones principales del editor de gráficas:

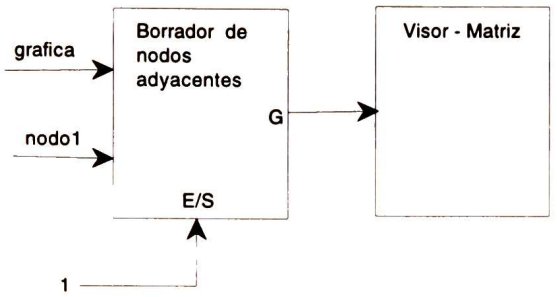

Figura 3.16: Visualización de una matriz.

3.4.1 Selección.

La versión visual de esta operación viene conformada por:

- **Selección manual.** En este tipo de selección el usuario elige mediante el ratón el arco o nodo con el que quiere trabajar.

- **Selección por extensión.** Eventualmente despueś que se ha realizado una selección, se desea extenderla, esta operación puede realizarse mediante la **selección por adyacencia**. Esto significa ampliar la selección a aquellos nodos que son adyacentes a la selección previa.

- **Selección por expresión.** Esta selección se utiliza para seleccionar veredas, de las que se conocen las etiquetas de los arcos que la forman. Se auxilia de lo que hemos denominado una *secuencia abreviada*, ésta consiste de una expresión como la siguiente:

  3(4(0,1,2),2(2,1)).

  Esta expresión significa seguir la secuencia de arcos 0,1,2; cuatro veces, luego seguir la secuencia 2,1; dos veces y todo el proceso 3 veces.

  La selección por expresión, toma el nodo actual y selecciona la rutas cuyos etiquetas de los arcos coincidan con la secuencia.

- **Selección por área**. Es un tipo de selección manual que escoge los nodos y arcos que están dentro de un rectángulo definido mediante el ratón.
- **Selección por etiqueta**. Elige a aquel nodo que coincide con cierta etiqueta.
- **Deselección**. Cancela cualquier selección realizada con anterioridad.

3.4.2 Manipulación.

La manipulación propuesta tiene la siguientes procesadores :

- **Acomodo en abanico**. Distribuye la selección en forma de un abanico.
- **Acomodo circular.** Distribuye la posición de los nodos alrededor de un círculo.
- **Coloración.** Permite por medio de un panel escoger el color con que se desea colorar la selección.
- **Compresión.** Sustituye la selección con un supernodo, que la contendrá.
- **Descompresión.** Hace la operación inversa a la anterior.
- **Desplazamiento manual.** Permite cambiar la posición relativa de una selección previa.
- **Escalar y trasladar.** Cambia el tamaño y ubicación de una gráfica.
- **Generación de gráficas disjuntas.** Si la gráfica tiene varias subgráficas disjuntas entre sí, las separa en distintas ventanas.
- **Generación de árbol de cobertura.** Genera el árbol que tiene como raíz al nodo actual. y cuyas ramas coinciden con los arcos de la gráfica.

3.4.3 Visualización.

Con el fin de observar al gráfica en distintas formas contamos con:

- **Representación gráfica.** Visualiza en una ventana la gráfica que se está editando.
- **Navegador.** Visualiza en una jerarquía, cómo esta estructurado la gráfica, recordemos que algunos nodos contienen selecciones previas que a su vez podrían contener otros nodos.
- **Forma matricial**. Si bien en el programa no se puede observar la gráfica en su forma matricial, permite la exportación e importación en ese formato.

Ciertamente el conjunto de operaciones que soporta el editor no contempla todos los procesadores que se han propuesto en la sección anterior, sin embargo, podemos decir que el conjunto es aunque precario, versátil.

3.5 Sumario.

En este capítulo hemos descrito los elementos más importantes de LGRAF, un lenguaje de consulta para la manipulación de gráficas. LGRAF puede describir una cantidad muy grande de consultas, que se puede aplicar en la construcción de un ambiente de consulta y manipulación de gráficas en diversas plataformas. El ambiente de consulta no necesita partir de cero, se cuenta con un grupo amplio de clases que son la base del editor de gráficas de esta tesis. La descripción de las clases en detalle se analiza en el capítulo 6.

Capítulo 4

Casos de estudio

4.1 Introducción.

En este capítulo utilizaremos el editor en cuestión para llevar a cabo operaciones que son típicas en algunos diagramas y que describimos a continuación:

1. Comprobar si una secuencia de estados existe en un diagrama de subconjuntos (Vease capítulo 2).

2. Acomodo de una gráfica grande.

4.2 Caso I.

4.2.1 Importación.

Primero que nada, tendremos que generar un diagrama de de Bruijin para un autómata de 3 estados. Esto lo podemos lograr alimentando al sistema mediante una matriz de 2^9 nodos esto es 512 nodos, en un archivo con extensión *eti*. En este archivo cada dato viene separado por un espacio o un salto de línea, el archivo contendrá (en ese orden):

1. Número de nodos. En esta sección, para el caso específico del problema, es el número 512.

2. Etiquetas de los nodos. Aquí se describen las etiquetas de los nodos, la primera etiqueta para el primer nodo, la segunda etiqueta para el segundo y así sucesivamente. Cada dato viene separado por un espacio, tabulador o retorno de carro.

   001 101 101 201 ...

3. Etiquetas de los arcos. En esta sección describiremos las etiquetas de las ligas en 512 renglones con 512 columnas, a cada una de ellas le corresponderá una liga con nodo origen de acuerdo al número de columna; y nodo destino, de acuerdo al número de fila.

   1 0 2 1 ...
   0 2 2 1 ...
   1 1 2 2 ...
   2 1 1 1 ...
   ... ...

Una vez que tenemos los archivos hacemos el siguiente procedimiento :

1. Seleccionar la opción del menú *Documento/Nuevo.*

2. Seleccionar la opción del menú *Documento/Cargar* y elegir el archivo recién creado con la extensión *eti.* El sistema cargará automáticamente los archivos de etiquetas y se visualizará la gráfica en ese momento.

4.2.2 Selección de rutas.

Ya que tenemos el diagrama de de Bruijin, seguimos los siguientes pasos.

1. Dentro del diagrama de de Bruijin, tenemos que ubicar el nodo que preside a todos. Para ésto, desplegaremos el panel de navegación mediante el menu *Herramientas/Herramientas de navegación.* y escribimos en el campo de etiqueta *Etiqueta de nodo actual*, la etiqueta del nodo que deseamos ubicar, en ese momento se resaltará el nodo en cuestión.

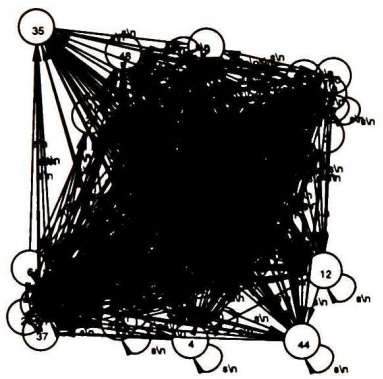

Figura 4.1: Gráfica inicial.

2. Ahora elegiremos una ruta mediante una *secuencia abreviada* (Vease capítulo 3) por medio del comando del menu *Selección/Selección por ruta.*

3. Si al oprimir la opción *Seleccionar*, la aplicación envía un mensaje de *selección completa.* Sabremos si la secuencia existe o no, en el diagrama.

4.3 Caso II.

En este ejemplo tendremos una gráfica de 50 nodos como en la figura 4.1. Los pasos que seguiremos para acomodar esta gráfica serán los siguientes:

1. Seleccionamos del panel de edición la herramienta de selección por área  , la cual nos permitirá seleccionar toda la gráfica.

2. Ahora la acomodaremos en forma de red (veáse tipos de acomodos, en el capítulo anterior), por medio de la herramienta de acomodo en red  . Obteniendo así una gráfica como la de la figura 4.2.

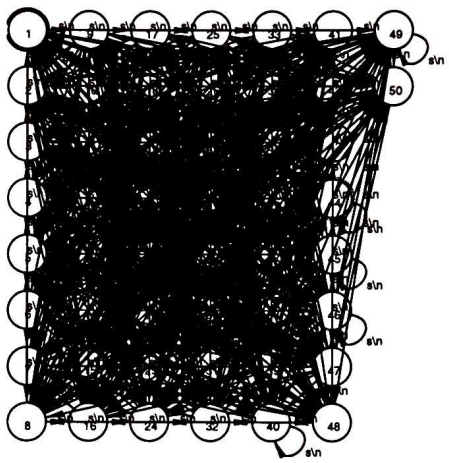

Figura 4.2: Gráfica acomodada en red.

3. Probablemente con la gráfica generada no sea suficiente para mostrar algún tipo de estructura, por lo que procederemos con una simplificación del grafo en estudio. Utilizaremos un árbol que tendrá como raíz el nodo actual. El modo en que generaremos la gráfica es mediante el siguiente procedimiento :

   (a) Situar el nodo actual en aquel nodo cuya etiqueta sea la raíz del árbol. Esto se logra, en el caso de que tengamos la etiqueta del nodo, escribiendo la etiqueta en el campo de etiqueta del nodo en el panel de navegación; por otro lado si se desea situarse en un nodo en determinada parte de la ventana, se debe elegir la herramienta de edición de nodo  en el panel de edición.

   (b) Generar árbol. La generación del árbol se realiza mediante el menú *Generación/Generación de árboles.* El resultado se muestra en la figura 4.3.

4. Acomodo del árbol generado. El árbol que se ha generado puede acomodarse en forma de abanico, siguiendo los siguientes pasos:

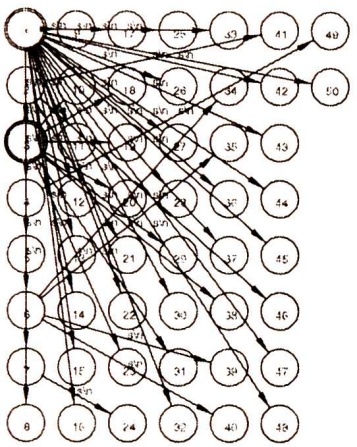

Figura 4.3: Árbol generado.

(a) Seleccionar todo el grafo.

(b) Seleccionar el ícono  para acomodo en forma de abanico del panel de edición.

(c) Marcando en la ventana por medio de una línea, la dirección aproximada que tendrá el árbol, finalmente se visualizará una gráfica, la cual aparece en la figura 4.4

4.4 Sumario.

En este capítulo se han mostrado ejemplos concretos para sacar provecho de la herramienta diseñada en esta tesis. La variedad de operaciones que tenemos a la mano no solo son las que aparecen en este capítulo, la gama completa se ha descrito en el capítulo 3 y las mismas pueden combinarse de la manera que más convenga al lector.

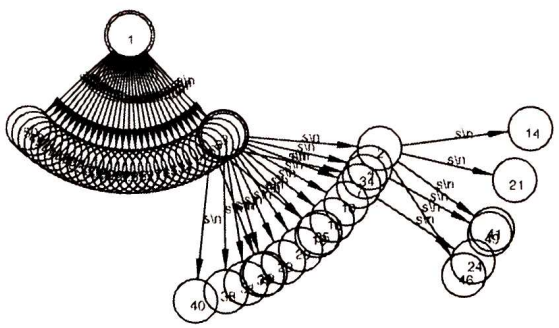

Figura 4.4: Árbol acomodado.

Parte II

Descripción del framework

Capítulo 5

Arquitectura

5.1 Introducción.

En el capítulo 3 se propone un lenguaje para la edición de una gráfica, pero no tomamos en cuenta ningun detalle de implantación. En este capítulo buscaremos dar un panorama general de la arquitectura de la aplicación y la manera en qué se puede extender. Si quisiéramos estudiar la constitución interna de un edificio, empezaríamos estudiando de lo más general hacia lo más particular. Del mismo modo procederemos estudiando los niveles de abstracción de nuestra aplicación y terminando con las características principales de las clases. Empezaremos con conceptos generales de diseño, posteriormente describiremos la arquitectura de la aplicación y finamente características físicas de la aplicación.

5.2 Conceptos básicos de diseño.

5.2.1 Orientación a objetos.

El sistema se ha elaborado bajo el paradigma de la programación orientada a objetos(POO). La razón es que este paradigma ha proporcionado resultados interesantes sobre todo en dos vertientes. Por un lado la comprensión del código, cuestión básica para su posible extensión, y segundo, el polimorfismo, que proporciona flexibilidad al cambio de re-

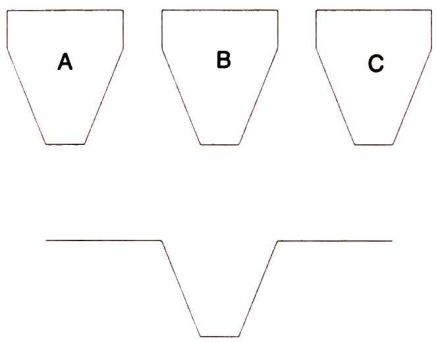

Figura 5.1: Separación entre interfaz e implantación

querimientos o añadiduras al sistema. El polimorfismo consiste en la construcción de un mecanismo que permite a distintos entes (llamados objetos en este paradigma) satisfacer una misma requisición. Este principio ha sido usado en la electrónica [Cox92] con gran éxito.

Por ejemplo en la figura 5.1 A,B y C representan secciones de código que deben satisfacer un mismo grupo de requerimientos, éste último representado por un orificio. A, B o C pueden encajar en el orificio, pero quizás A sea mejor que B o C. Por el momento podríamos poner C y en el futuro sustituirlo por B. Esta idea nos da una flexibilidad al cambio y una vía de extensión por medio de herencia.

5.2.2 Reutilización.

La reutilización de código probado requiere por un lado tener presente el tipo de software que se quiere construir y respetar algunos principios que facilitan la reutilización. Los tipos de software y sus necesidades asociadas son [Gam94, pag. 26]:

- **Programas de aplicación.** Un programa como una hoja de cálculo o un editor de documentos, tiene como prioridades principales el reuso interno. Así como la facilidad de su mantenimiento y extendibilidad. El reuso interno asegura que no se va a diseñar mas que lo que se necesita.

- **Toolkits.** A menudo una aplicación incorpora clases de otras librerías de clases predefinidas, estas últimas son llamadas toolkits. Los toolkits no imponen un diseño particular, ellos solo dan funcionalidad que pueda ayudar al trabajo de la aplicación. El problema con los toolkits es que deben ser generales, deben evitar dependencias y suposiciones entre clases, las cuales pudieran limitar la flexibilidad de la aplicación.

- **Frameworks.** Un framework, armazón o esqueleto es un conjunto de clases cooperativas que configuran un diseño reutilizable para una clase específica de software. Un framework es bastante útil porque permite tener un cimiento para construir algo más grande. Sin embargo el diseño de un framework tiene sus complejidades. Esto se debe a que hay que cuidar que todas las clases cooperen sin acoplarse demasiado. Porque de lo contrario provocará que el sistema sea poco comprensible y extendible.

Principios de reutilización.

Independientemente del tipo de aplicación que se construye, prácticas, que se deben evitar en lo posible, son [Gam94, pp 24]:

1. *Crear un objeto especificando explícitamente una clase.* Al especificar el nombre de una clase cuando se crea un objeto, se obliga al programador a realizar una implantación en particular, en lugar de una interfaz particular, lo que provocará dificultad para los cambios futuros. La manera de evitar este efecto es crear los objetos indirectamente.

2. *Dependencia en operaciones específicas.* Cuando se codifica una operación particular, se especifica que solo se podrá realizar de una manera. Evitando codificar operaciones de manera particular tiene como efecto que sea más fácil el cambio, hacia requerimientos que sean satisfechos en tiempo de compilación y ejecución.

3. *Dependencia de una plataforma de hardware y software específica.* Las interfaces del sistema operativo e interfaces de programación de aplicaciones (APIs. en inglés) difieren según la plataforma de

hardware y software. El software que depende de una plataforma particular será difícil de portar a otras plataformas. Lo que es más, será difícil mantenerla al día en su plataforma nativa. Es importante por lo tanto diseñar el software para limitar sus dependencias de una plataforma.

4. *Dependencias de representaciones e implantaciones.* Los clientes que saben como un objeto es representado, guardado, localizado o implantado, tendrán la necesidad de modificarse si el objeto cambia. El ocultamiento de la información hacia los clientes, los aisla de la cascada de modificaciones.

5. *Dependencias algorítmicas.* Los algoritmos son a menudo extendidos, optimizados y reemplazados durante el desarrollo y reuso. Los objetos que dependen de un algoritmo, cambiarán cuando el objeto cambie. Así los algoritmos que cambien deben aislarse.

6. *Acoplamiento fuerte.* Las clases que están fuertemente acopladas son difíciles de usar de una manera aislada, porque dependen demasiado entre ellas. Un acoplamiento fuerte conduce a un sistema monolítico, donde no se puede cambiar una clase sin analizar y cambiar muchas otras clases. El sistema llega a ser una masa densa, que es difícil de aprender, portar y mantener. El acoplamiento flojo o libre, incrementa la probabilidad de que una clase pueda ser reutilizada por sí misma y además el sistema puede ser aprendido, portado, modificado y extendido más fácilmente.

7. *Extensión de funcionalidad mediante subclases.* Personalizar un objeto por medio de subclases a menudo no es fácil. Cada clase tiene una implantación fija. Definir una subclase requiere un conocimiento profundo de la clase padre.

8. *Incapacidad para alterar clases adecuadamente.* Algunas veces se tiene que modificar una clase pero tal modificación no está a nuestro alcance. Quizás se tiene la necesidad de un código fuente y no se tiene o cambiarlo significaría montones de cambios en las clases existentes.

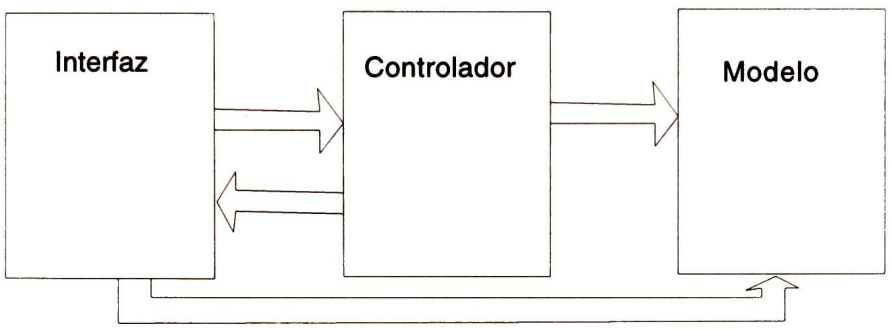

Figura 5.2: Paradigma MVC

Del mismo modo en que se deben evitar ciertas prácticas, una buena guía es el uso de principios probados, como lo es el paradigma MVC(Model-View-Controller).

El paradigma MVC(Model-View-Controller).

Este paradigma apareció por primera vez en SMALTALK, y sirve para aislar el modelo de una interfaz gráfica en particular. La separación de una sección de código de otra, hace más flexible el diseño porque disminuye el acoplamiento. La idea básica es expresar las clases de la aplicación en tres categorías distintas:

- Modelo. Esta categoría identifica a todas aquellas clases que reflejan solamente el dominio del problema.
- Interfaz. Esta categoría asocia a todas las clases que representan un objeto visual en pantalla como son menúes, paneles, ventanas, diálogos etc.
- Controlador. Se encarga de atender cada una de las entradas del usuario y hacérselas llegar al modelo y a la interfaz.

Cada flecha en la figura 5.2 representa que la clase, de donde empieza la flecha, conoce y puede enviar mensajes a donde llega la flecha, pero no al revés [Jac95].

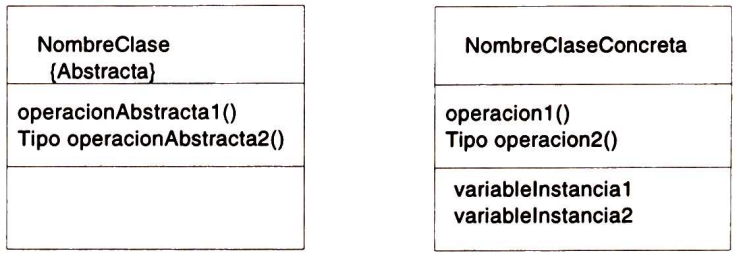

Figura 5.3: Representación de clases abstractas y concretas.

5.3 UML y diagramas de clases.

La notación para clases utilizada en este documento está expresada en UML(Unified Modeling Languaje). UML [Ric97] surge de la unificación de distintos notaciones para representar modelos de información en el paradigma de orientación a objetos, se proyecta como un estándar *de facto* para la notación en las distintas fases de desarrollo de aplicaciones. La sección utilizada de UML, se refiere al modelo de clases. En la figura 5.3 se muestran las representaciones para clases abstractas y clases concretas. Una clase se indica mediante una caja con un título en negrita al tope. Las operaciones principales se ubican debajo del nombre de la clase, finalmente las variables de instancia se ubican debajo de estas operaciones. Respecto a las relaciones entre objetos, la notación en UML(figura 5.4) para denotar herencia es un triángulo con una de las puntas hacia la superclase y la base hacia las clases derivadas. El hecho de que un objeto forme parte de otro se indica mediante una flecha, con un rombo en la base. La flecha apunta a la clase que es agregada. Una flecha sin el rombo en la base, denota una relación no estructural (por ejemplo Cuadrado tiene una referencia a un objeto Color, el cual podrían tener otras figuras). El nombre para una referencia (figuras en este caso) podría aparecer en la base para distinguirla de otras referencias. Hay dos extensiones que se han agregado a la notación de UML, la primera extensión es la indicación de que clases instancian a otras, esto lo indicamos mediante una línea punteada con una flecha hacia el objeto creado . La segunda es una forma de indicar

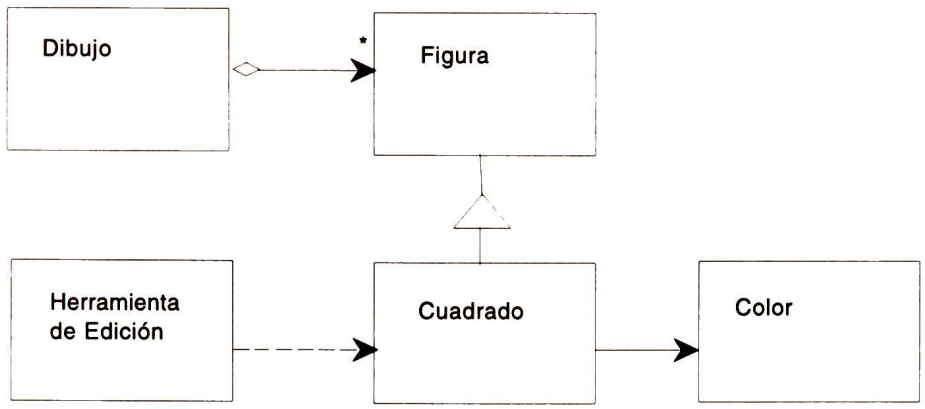

Figura 5.4: Herencia, relaciones simples y de composición.

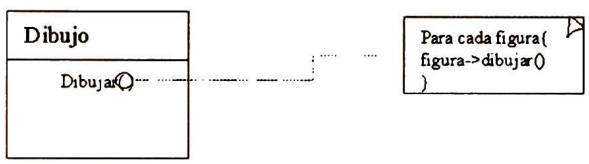

Figura 5.5: Notación para pseudocódigo

anotaciones de pseudocódigo (figura 5.5) mediante una línea punteada.

5.4 Estructura del sistema.

La estructura del sistema la podemos ver desde el punto de vista de la interfaz visual y la estructura de código. Empezaremos desde la parte visual.

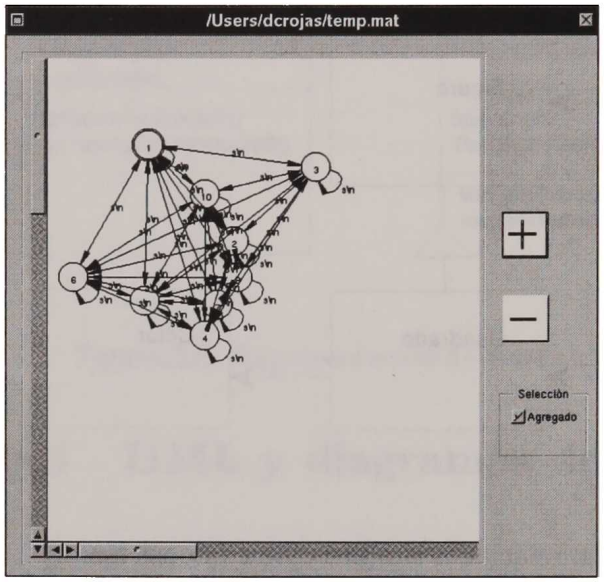

Figura 5.6: Ventanas de visualización de gráficas

5.4.1 Elementos de la interfaz gráfica.

La interfaz gráfica tiene como elementos principales:

- *Ventanas por documento.* El editor puede tener una serie ventanas con distintas gráficas y puede hacer operaciones de copiado entre cada uno de estos documentos, (vease figura 5.6).

- *Panel de edición básicas.* El panel permite accesar a una serie de operaciones al nivel de nodo/liga y subgráficas, (vease la figura 5.7).

- *Explorador(Browser) de la jerarquía dentro de una gráfica.* La gráfica está estructurada en un árbol que la permite ver en distintos niveles.

- *Menú con operaciones de salvado/guardado, copiado y pegado, visualización de paneles, generación de gráficas complemento y componentes.*, (vease 5.8).

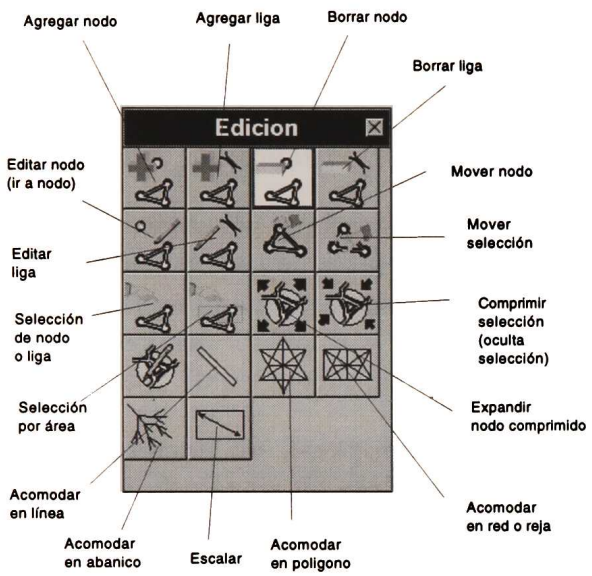

Figura 5.7: Panel de edición.

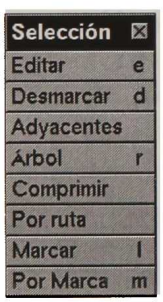

Figura 5.8: Menu de selección.

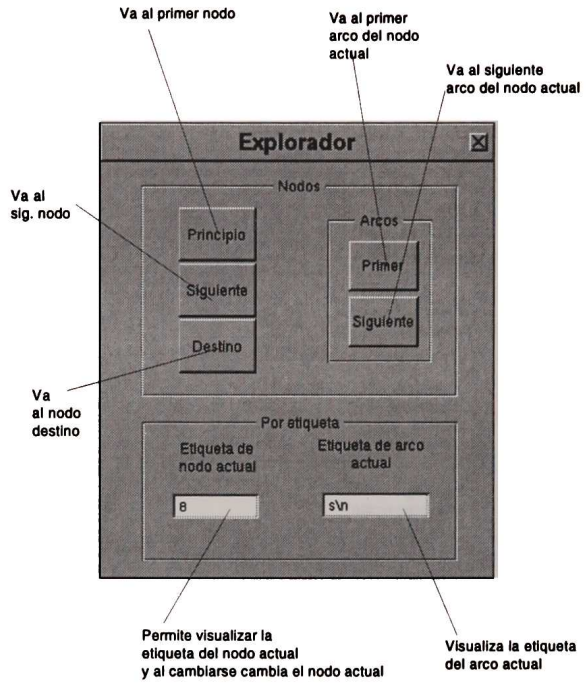

Figura 5.9: Panel de navegación.

- *Panel para navegar por la gráfica.* Por medio de este panel podemos ir de un nodo a otro o cambiar de un arco a otro arco. Esta operación tiene sentido porque varias de las operaciones de edición dependen del nodo que está enfocado, (vease figura 5.9).

5.4.2 Estructura del código

Podemos ver a nuestro sistema como un paquete (en el sentido más amplio del término) que soluciona grupos de servicios:

- **Servicios Básicos.** Satisface las necesidades básicas de implantación de una gráfica, independiente de su representación, permitiendo las operaciones básicas de adición, borrado, búsquedas básicas y enumeración.

Clases : ImpGrafo, ImpGrafoLista, Grafo, Nodo, Liga y Secuenciador.

- **Servicios de Selección de Gráfica.** Ofrece una variedad de selecciones de una gráfica, incluyendo modificaciones a una selección previa.
  Clases: Diagrama.

- **Servicios de gráfica concreta.** Contiene todas las operaciones de manejo de una gráfica pero tomando en cuenta que la gráfica ya contiene una representación en dos dimensiones y una simbología específica.
  Clases: RepNodoSimple, RepGrafo, RepNodoComp, RepLiga.

- **Servicios de Jerarquía de gráficas.** Este módulo es básico para el manejo de gráficas grandes, es decir con muchos nodos, permite la estructuración de una gráfica en una jerarquía, de tal modo que a cada nodo le corresponde una subgráfica. El módulo contiene operaciones de navegado dentro de la jerarquía.
  Clases: NodoComp, GrafoDep, Navegador.

- **Servicios de exportación e importación.** En un momento dado tenemos que guardar y recuperar las gráficas desde distintos formatos.
  Clases: CtrlGrafoGeom, ArchGrafo, ArchGrafoGeom, ArchGrafoMat, ArchGrafoSimple

- **Servicios para operar sobre una gráfica.** Esta gama de servicios tiene como utilidad poder hacer operaciones en profundidad, a lo largo de todo la gráfica.
  Clases: Visitante, Acomodador, RecolectorComp.

- **Servicios de hacer y deshacer** Estos servicios sirven para deshacer un conjunto de operaciones realizadas sobre un diagrama.
  Clases: PilaOper, Operación, OperConcreta.

- **Servicios de búsqueda y reconocimiento.** En algunos momentos de la edición de gráficas, buscamos subgráficas con ciertas

características y deseamos buscarlos en una gráfica.

- **Servicios de exploración en gráfica.** Estas clases permiter referirse a la gráfica sin alterarla en absoluto.
  Clases: IterGrafo.

Ahora bien hay una serie de clases que sirven como puente entre la interfaz gráfica y el modelo éstas son: CtrlGrafoGeom y sus delegados CtrlPrefGrf, CtrlDibujo, PrefDibujoDiag, OperMouse, OperMouseConcreta

El diagrama de la figura 5.10 no tiene todas las relaciones entre las clases, con el fin de apreciar las secciones verdaderamente estructurales del sistema. Siendo estrictos, en UML, sólo es válida una línea entre las clases, así pues las líneas que parecen subdividirse son líneas independientes.

5.5 Características físicas de la herramienta.

Hay un cierto conjunto de características que son dependientes de la implantación y que deben de tomarse en cuenta para la explotación de las librerías. Este conjunto de características las describiremos en seguida.

5.5.1 Plataforma

La arquitectura según el diccionario busca la planeación y distribución del espacio en una construcción. Al igual que en los edificios la construcción del espacio debe ser sólida y por esta razón, este editor de gráficas fue construido en OPENSTEP. OPENSTEP es una plataforma que cuenta con una amplia gama de herramientas para la construcción de programas. La facilidad con que se pueden crear aplicaciones mediante la reutilización de código previamente construido y probado (estructurado en clases) hacen de la construcción algo bastante rápido y práctico. Por ejemplo si queremos programar una lista o alguna estructura de datos, contamos con lo que se conoce como el Foundation Kit, una familia de clases que contiene diccionarios, arreglos dinámicos y

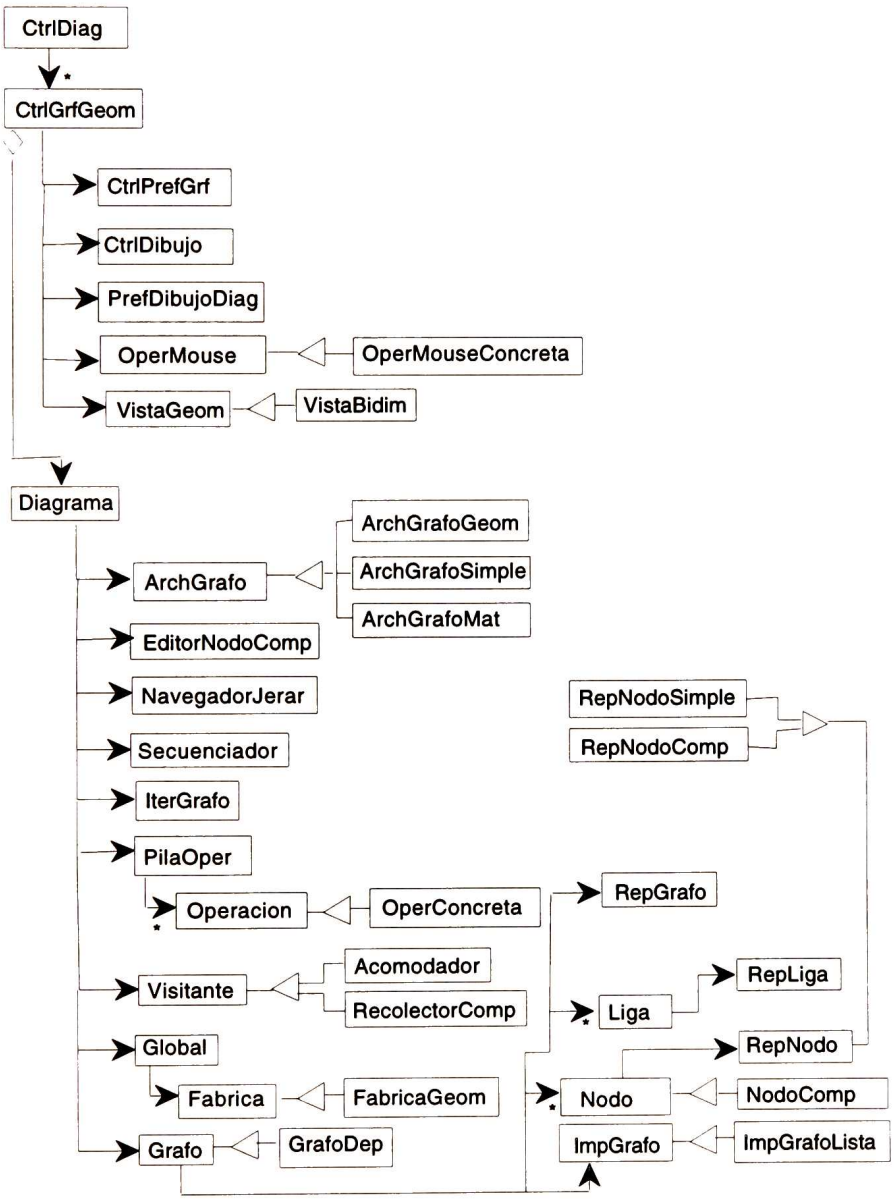

Figura 5.10: Diagrama general del sistema

conjuntos entre otros. Otra facilidad en la programación la constituye la interfaz gráfica. Normalmente en muchas plataformas la interfaz gráfica se diseña como si estuviera totalmente aislada de la aplicación. OPENSTEP así como su antecesor NeXTSTEP cuenta con una herramienta para construir la interfaz gráfica que permite ligar rutinas a eventos que se pueden dar en la interfaz. Un evento en la interfaz es un suceso como hacer una pulsación sobre el ratón o la entrada de texto por el teclado.

5.5.2 Lenguaje

La selección del lenguaje es como la elección de los ladrillos de una construcción, el material con el que modelamos. El editor de gráficas se programó en C Objetivo que si bien es el lenguaje, por así decirlo nativo de OPENSTEP , maneja el paradigma de POO de una manera sencilla sin una sintaxis muy complicada, C objetivo es parecido a SMALTALK pero puede coexistir con rutinas programadas en C. C Objetivo no es tan tipificado como C++, lo cual, aunque lo hace propenso a cierto tipo de errores, lo hace flexible.

5.6 Sumario.

En este capítulo se han expuesto los principios de reutilización de código bajo el paradigma de orientación a objetos, con el fin de facilitar la comprensión de la filosofía de construcción de las clases. Posteriormente describimos la arquitectura por medio de grupos de clases con finalidades comunes. Por último, se dan aspectos físicos, y por lo tanto particulares, de la aplicación por si se desea migrar la aplicación o midificarla.

Capítulo 6

Catálogo de clases

6.1 Introducción.

Este capítulo tiene por objetivo mostrar una descripción de las clases del framework al más puro estilo de las documentaciones de clases en los frameworks como el AppKit de OPENSTEP. Se busca que éste capítulo sea usado como un manual de consulta. Para cada clase se definen:

1. **Descripción general de la clase**. Aquí se muestran las características principales de cada clases, en algunas ocasiones se mostrarán algunos ejemplos de uso de la clase.

2. **Tipos de métodos**. Son las interfaces de los métodos utilizados por las clases, clasificados por categorías.

3. **Métodos de instancia**. Son las descripciones de cada método que aplica a los objetos de la clase.

4. **Métodos de clase**. Son las descripciones de cada método que aplica a las clases específicamente.

6.2 ArchGrafo.

Hereda de : Ninguno
Declarada en : CtrlDiag

Descripción de la clase.

La clase abstracta ArchGrafo almacena y recupera grafos en distintos formatos. ArchGrafo no se puede utilizar directamente, se necesita especializar en una clase concreta. Para su utilización en la recuperación de grafos, requiere la escritura de los métodos **extraerNodo**, **extraerLiga** que se encargarán de fabricar un nodo y una liga respectivamente. Para poder lograr este objetivo deber utilizar **miFp**, el puntero de un archivo asociado a ArchGrafo. Del mismo modo para el guardado de grafos se deben reescribir los métodos **introducirNodo:** e **introducirLiga:** .

Usando formato de matriz.

Para leer una matriz, se deberá utilizar la clase concreta **ArchGrafoGeom**. El formato tiene la siguiente forma:
<numNodos>
<secuencia de unos o ceros en forma de un matriz cuadrada separada por espacios>
ya en esta clase están escritos los métodos **extraerNodo** y **extraer liga**.

Tipos de Métodos.

Recuperación	- extraerNodo extraerLiga extraerGrafo: extraerIdNodo
Salvado	introducirNodo: introducirLiga: introducirGrafo: imprimir:
Inicialización	initDesde:

Métodos de instancia.

extraerGrafo:

-(void)**extraerGrafo:** (Grafo *)grafo Obtiene un *grafo* del flujo de ArchGrafo. Este método se puede reeescribir si se desea controlar el orden y el formato en que se recuperarán nodos y ligas en el flujo. **Ver también:** extraerNodo, extraerLiga

extraerIdNodo

-(int)**extraerIdNodo**
Obtiene del flujo *miFp* el id del nodo, este método se utiliza si se desea hacer una implantación propia de extraerNodo.
Ver también: Nodo.h

extraerLiga

-(id)**extraerLiga**
Extrae una liga del flujo *miFp*. Este método se debe sobrescribir para recuperar según un formato elegido.
Ver también: Liga y extraerNodo

extraerNodo

-(id)**extraerNodo**
Extrae un nodo del flujo *miFp*. Este método se debe sobrescribir para recuperar según un formato elegido.
Ver también: Nodo y extraerLiga

imprimir:

(void)**imprimir:**(NSString *)buffer
Introduce *buffer* hacia el flujo que se está almacenando.

initDesde:

(void)**initDesde:**(FILE *)fp
Inicializa el estado del objeto, tomando a *fp* al flujo actual. Si sobreescribe este método esté seguro de invocar a la implantación de *super*.

introducirGrafo:

(void)introducirGrafo: (Grafo *)grafo
Salva un *grafo* en el flujo de ArchGrafo. Este método se puede reeescribir si se desea controlar el orden y el formato en que se grabarán los nodos y ligas en el flujo. **Ver también:** introducirNodo:, introducirLiga.

introducirLiga:

-(void)**introducirLiga:** (Liga *)liga
Guarda un nodo en el flujo *miFp* en algún formato específico. Si se desea salvar un nodo en un formato específico se debe sobrescribir este método.
Ver también: extraerLiga

introducirNodo:

-(void)**introducirNodo:** (Nodo *)nodo
Guarda un nodo en el flujo *miFp* en algún formato específico. Si se desea salvar un nodo en un formato específico se debe sobrescribir este método.
Ver también: extraerNodo

6.3 Diagrama.

Hereda de : Ninguno
Declarada en : Diagrama.h

Descripción de la clase.

Diagrama es una clase que permite la construcción y edición de un grafo en su forma gráfica, cuenta con:

- Un grafo de edición.
- Una vista donde se despliega el grafo que se está editando. La vista puede ser asignable.
- Un grafo para seleccionar algunas secciones del grafo. El color en que se muestra es configurable.
- Un mecanismo, una pila de operaciones, para hacer y deshacer las operaciones realizadas.
- Un iterador o cursor que resalta el nodo en que se encuentra , así como los arcos que salen del nodo.
- Un diccionario donde se puede establecer las configuraciones de colores, para el desplegado, contiene la llaves :
  - "colorPorOmision". El color en que se dibuja por omisión.
  - "colorNodoActual". El color que se dibuja el nodo actual del iterador.
  - "colorNodoDestino". El color del nodo que se encuentra adyacente al nodo actual del iterador.
  - "colorSel". El color en que se marcará la selección.

Diagrama provee una interfaz común a varias clases, en patrones [Gam94] , esto se le conoce como una fachada (facade). La utilidad de esta clase radica en poder tratar un conjunto de clases que cooperan como una sola clase.

Tipos de Métodos.

Consultando el diagrama	diccDibujo grafo

	grafoSel
	pilaOper
Inicializando y terminando	init
	initConGrafo:
	terminar
Manipulando visualmente	acomodarGrafo:
	desplazarNodo: en:
	formarPoligGrafo:
Comprimiendo y descomprimiendo	comprimirGrafo:
	- descomponerNodoEnPunto:
	- obtenerNodoComp:
Manipulando vistas	actualizar:
	- dibujar:
	- pintarCirculoNodo: en:
	- pintarLinea:
	- pintarRect: y:
	- vistaActual
	- vistaActual:
Editando grafo	agregarLigaEntre: y:
	agregarNodo:
	agregarSubGrafo:
	borrarLiga:
	borrarNodo:
	borrarLigasNodo:
	borrarSubgrafo:
Trabajando con la seleccion	borrarSel
	- colorearSel
	- comprimirSel
	desplazarSeleccion:
	- expandirSelAdyacentes
	- generarSeleccion
	limpiarSel
	seleccionarAreaEntre:y:
	seleccionarLiga:
	seleccionarLigaEnPunto:
	seleccionarLigasInter

	seleccionarNodo:
	seleccionarNodoEnPunto:
Haciendo y deshaciendo	deshacer
	rehacer
Operaciones entre diagramas	integrarDiag:
Iterando	irNodoDestino
	irPrimerArco
	irPrimerNodo
	irSigArco
	irSigNodo
	nodoActual

Métodos de instancia.

acomodarGrafo:

-(void)**acomodarGrafo**: (NSPoint)punto
Reacomoda la posición de cada nodo con el fin de que la orientación de cada arco vaya preferentemente de izquierda a derecha.

actualizar

-(void)**actualizar**
Actualiza la vista con la información que contiene el diagrama.

agregarLigaEntre: y:

-(void)**agregarLigaEntre**: (NSPoint)origen **y:** (NSPoint)destino
Agregar una nuevo liga al diagrama, únicamente proporcionando los puntos *origen* y *destino* de la vista.

agregarNodo:

-(void)**agregarNodo:** (NSPoint *)miPunto
Agrega un nuevo nodo al diagrama, ubicando el punto *miPunto* en donde se desea insertar el nuevo nodo.

agregarSubGrafo:

-(void)**agregarSubGrafo:** (Grafo *)grafo
Agrega un grafo previamente construido al diagrama actual, asignando nuevas secuencias a los nodos que se están agregando.

borrarLiga:

-(void)**borrarLiga:** (NSPoint *)miPunto
Borra la liga más cercana a *miPunto.*

borrarNodo:

-(void)**borrarNodo:** (NSPoint *)miPunto
Borra el nodo que está en la proximidad de *miPunto.*

borrarLigasNodo:

-(void)**borrarLigasNodo:** (Nodo *)unNodo
Borra las ligas que entran y salen de un nodo.

borrarSel

-(void)**borrarSel**
Borra del grafo de edición la selección actual.

borrarSubgrafo:

-(void)**borrarSubgrafo:** (Grafo *)unGrafo
Borra del grafo de edición, el grafo definido como *unGrafo.*

colorearSel

-(void)**colorearSel**
Colorea la sección del grafo de edición, indicado por el grafo de selección.

comprimirGrafo:

-(void)**comprimirGrafo:** (Grafo *)grafoAComp
Convierte en el grafo de edición, grafoAComp, en un nodo, que a partir de entonces almacenará grafoAComp.

comprimirSel

-(void)**comprimirSel**
Comprime el grafo seleccionado.
Ver también: comprimirGrafo:.

descomponerNodoEnPunto:

-(void)**descomponerNodoEnPunto:** (NSPoint)unPunto
Descomprime el nodo más cercano a *unPunto.*
Ver también: comprimirGrafo:.

deshacer

-(void)**deshacer**
Deshace la última operación que se realizó.

desplazarNodo: en:

-(void)**desplazarNodo:** (NSPoint *)pos **en:**(NSPoint *)desp
Mueve un nodo del grafo de edición en la proximidad de *pos* un desplazamiento *desp.*

desplazarSeleccion:

-(void)**desplazarSeleccion:** (NSPoint *)desp
Mueve en un desplazamiento *desp* la selección realizada.

dibujar:

-(void)**dibujar:** sender
Manda a dibujar el grafo de edición, este método es llamado indirectamente por *actualizar.* No es necesario llamarlo directamente.

diccDibujo

-(NSMutableDictionary *)**diccDibujo**
Regresa el diccionario de los colores con que se configura la clase diagrama.

expandirSelAdyacentes

-(void)**expandirSelAdyacentes**
Expande la selección actual para que abarca a los nodos que son adyacentes a la selección, así como sus ligas.

formarPoligGrafo:

-(void)**formarPoligGrafo:** (NSPoint)miPuntoOrigen
Distribuye los nodos del grafo en un círculo con centro en *miPuntoOrigen.*

generarSeleccion

-(Grafo *)**generarSeleccion**
Devuelve una copia de la selección realizada hasta el momento.

grafo

-(Grafo *)**grafo**
Devuelve el grafo de edición.

grafoSel

-(Grafo *)**grafoSel**
Devuelve el grafo seleccionado.

init

-(void)**init**
Inicializa la estructura interna del Diagrama.

initConGrafo:

-(void)**initConGrafo:** (Grafo *)unGrafo
Igual que el método *init* pero tomando como grafo de edición *unGrafo*.

integrarDiag:

-(void)**integrarDiag:** (Diagrama *)diag
Se combina con otro objeto de la clase Diagrama, ejecutando todas las transacciones que habían realizado con *diag* pero ahora con el receptor.

irNodoDestino

-(void)**irNodoDestino**
Mueve el iterador hacia el nodo destino del nodo actual.

irPrimerArco

-(void)**irPrimerArco**
Situa el iterador en el primer arco del nodo actual.

irPrimerNodo

-(void)**irPrimerNodo**
Situa el iterador en el primer nodo del arco.

irSigArco

-(void)**irSigArco**
Situa al iterador en el siguiente arco del nodo actual.

irSigNodo

-(void)**irSigNodo**
Situa al iterador en el siguiente nodo enumerado en el grafo.

limpiarSel

-(void)**limpiarSel**
Limpia la selección actual.

nodoActual

-(void)**nodoActual**
Devuelve el nodo en que está situado el iterador.

obtenerNodoComp:

-(NodoComp *) **obtenerNodoComp:** (NSPoint)miPunto
Devuelve el nodo compuesto mas cercano a *miPunto.*

pilaOper

-(PilaOper *)**pilaOper**

pintarCirculoNodo: en:

-(void)**pintarCirculoNodo:**(NSPoint)posNodo **en:**(NSPoint)desp
Dibuja un círculo similar al de un nodo, a un desplazamiento *desp* del punto *posNodo.*

pintarLinea: y:

-(void)**pintarLinea:** (NSPoint *)origen **y:**(NSPoint *)destino
Pinta una línea desde *origen* y *destino.*

pintarRect: y:

-(void)**pintarRect:** (NSPoint *)origen **y:**(NSPoint *)destino
Pinta un rectángulo con esquinas, *origen* y *destino*

rehacer

-(void)**rehacer**
Rehace una operación previamente deshecha.

seleccionarAreaEntre:y:

-(void)**seleccionarAreaEntre:** (NSPoint)puntoOrigen **y:** (NSPoint)puntoDest
Marca como seleccionado todos los nodos y arcos que están entre el recuadro que se halla entre *puntoOrigen* y *puntoDest.*

seleccionarLiga:

-(void)**seleccionarLiga:** (Liga *)unaLiga
Selecciona del grafo de edición la liga *unaLiga.*

seleccionarLigaEnPunto:

-(BOOL)**seleccionarLigaEnPunto:**(NSPoint *)miPunto
Selecciona la liga más cercana a cierto punto.

seleccionarLigasInter

-(void)**seleccionarLigasInter**
Si la selección realizada tiene nodos aislados, el método selecciona las ligas que existen entre los nodos.

seleccionarNodo:

-(void)**seleccionarNodo:** (Nodo *)unNodo
Selecciona del grafo de edición, unNodo.

seleccionarNodoEnPunto:

-(BOOL)**seleccionarNodoEnPunto:**(NSPoint *)miPunto
Selecciona el nodo en la proximidad de miPunto.

terminar

-(void)**terminar**
Libera los recursos alojados por el diagrama.

vistaActual

vistaActual
Devuelve la vista actual.

vistaActual:

(void)**vistaActual:**unaVista
Asigna *unaVista* como vistaActual.

6.4 Fabrica.

Hereda de : Ninguno
Declarada en : Fabrica.h

Descripción de la clase.

Esta clase abstracta provee de una interfaz para crear distintas partes de un Diagrama, puede ser derivable en clases concretas (como Fabrica-Geom), para distintos tipos de diagramas. Encapsula la complejidad que podría haber en la instanciación de componentes de diagramas.

Tipos de Métodos.

Creando partes
- crearGrafo
- crearLigaEntre:
- crearNodoComp:
  crearNodoEn:
  crearLigaConCuenta:
  generarGrafoDep: conPadre:
  initSoloImp:

Métodos de instancia.

crearGrafo

-(Grafo *)**crearGrafo**
Crea un grafo ya inicializado, si se desea sobreescribir este método se

debe llamar a la implantación de *super*.

crearLigaEntre: y:

-(Liga *)crearLigaEntre: (NSPoint)origen y: (NSPoint)destino
Crea una liga entre dos puntos del un objeto de la clase Diagrama, si se desea sobreescribir este método se debe llamar a la implantación de *super*.

crearNodoComp:

-(NodoComp*)crearNodoComp: (Grafo *)unGrafo
Crea un nodo compuesto, conteniendo *unGrafo*, si se desea sobreescribir este método se debe llamar a la implantación de *super*.

crearNodoEn:

-(Nodo *)crearNodoEn: (NSPoint)unPunto
Crea un nodo sencillo, cualquier reescritura debe llamar a la implantación de *super*.

crearLigaConCuenta

-(Liga *)crearLigaConCuenta
Crea una liga nueva.

generarGrafoDep: conPadre:

-(GrafoDep *)**generarGrafoDep:** (Grafo *)unGrafo **conPadre:** (Grafo *)padre
Crea un grafo que depende de otro.

initSoloImp:

-(void)**initSoloImp:** (Grafo *)unGrafo
Inicializa *unGrafo* pero no inicializa la parte de representación.

6.5 Grafo.

Hereda de : Ninguno
Declarada en : Grafo.h

Descripción de la clase.

Esta es la clase fundamental en el framework, contiene soporte para:

- Cambio de implantación. La clase grafo provee de una interfaz para operar un grafo, independientemente de la implantación. La implantación puede variar, por el momento, se tiene la clase ImpGrafoLista, que soporta las operaciones básicas de un grafo por medio de listas.
- Secuenciador. Cada grafo cuenta con un objeto de la clase Secuenciador que le proporciona la secuencia interna, tanto a sus nodos como a sus arcos.
- Representación. Cada grafo puede tener la representación gráfica que desee, siempre y cuando implante la interfaz RepGrafo.h.
- Visita en profundidad. Permite asociar al grafo un objeto que podrá recorrer el grafo en profundidad. El objeto asociado responderá a la interfaz de la clase abastracta Visitante.

Tipos de Métodos.

Iniciando y terminando	initConImp: terminar
Consulta	imp
Consulta del estado del grafo	- checarConsistencia - esPosibleBorrarNodo: - estaVacio maxNodos
Copiado	clon copiarA: copiarSinRepA:

Llevar la numeracion interna de nodos	sec
	asignarSec:
	renumerar
Editando	agregarNodo:
	agregarLiga:entre:y:
	borrarLiga:entre:y:
	borrarLigaEntre:y:
	borrarLigasNodo:
	borrarArcoNodos
	borrarNodo:
	sustituirNodo:con:
Bu'squeda seg'un orden interior	arcoSaleEn:de:
	buscarLiga:entre:y:
	buscarNodo:
	buscarNumNodo:
	ligaEntre:y:
	nodoDestinoEn:y:
	nodoEstaEn:
	numArcosSalen:
	numArcosSalenNodo:
Consultando entrada y salida de nodos	ligasEntranNodo:
	ligasSalenNodo:
Etiquetando	nodoCuyoId:
Operando a nivel global	hazLigasEjecutar:con:
	hazNodosEjecutar:con:
Generando grafos derivados	generarComplemento
	generarComponentes
	generarDoble
Recorriendo en profundidad	visitarEnProf:
Manejando representacion	asignarRep:
	rep:

Métodos de instancia.

agregarLiga:entre:y:

(void)**agregarLiga:**(Liga *)liga **entre:**(Nodo *)nodo1 **y:**(Nodo *)nodo2
Agrega una liga entre dos nodos dados.

agregarNodo:

(void)**agregarNodo:**(Nodo *)nodo
Agrega un nodo al grafo.

arcoSaleEn:de:

(Liga *)**arcoSaleEn:**(int)numArco **de:** (int)numNodo
Regresa la liga que se ubica en el nodo *numNodo* y en el arco *numArco* dentro del nodo en cuestión.

asignarRep:

-(void)**asignarRep:** (id)rep
Asigna una representación al grafo.

asignarSec:

(void)**signarSec:**(id)unaSec
Asigna un secuenciador a grafo.

borrarArcoNodos

(void)**borrarArcoNodos**
Destruye todos los arcos que tiene un grafo.

borrarLiga:entre:y:

-(void)**borrarLiga:**(Liga *)liga **entre:**(Nodo *)nodoOrigen **y:**(Nodo *)nodoDestino
Borra un liga entre los dos nodos.

borrarLigaEntre:y:

(void)**borrarLigaEntre:**(Nodo *)nodoOrigen **y:**(Nodo *)nodoDestino
Borra la primera liga entre dos nodos.

borrarLigasNodo:

-(void)**borrarLigasNodo:** (Nodo *)unNodo
Borra todas las ligas que entran y salen de un nodo.

borrarNodo:

(void)**borrarNodo:**(Nodo *)nodo
Borra un nodo del grafo.

buscarLiga:entre:y:

-(Liga *)**buscarLiga:** (Liga *)liga **entre:** (Nodo *)nodoOrigen **y:** (Nodo *)nodoDestino
Verifica si una cierta *liga* existe entre *nodoOrigen* y *nodoDestino*, devuelve nil si no existe tal liga y *liga* en otro caso.

buscarNodo:

- **buscarNodo:**(Nodo *)unNodo
Verifica si *unNodo* existe en el grafo, devuelve nil si no lo encuentra y *unNodo* en otro caso.

buscarNumNodo:

(int)**buscarNumNodo:**(Nodo *)nodo
Devuelve el número secuencial en que está acomodado cierto nodo, -1 si no existe.

checarConsistencia

-(BOOL)**checarConsistencia**
Devuelve YES, si el grafo no tiene una inconsistencia interna, NO de

lo contrario.

clon

- **clon**
Devuelve una copia del grafo.

copiarA:

(void)**copiarA:** (Grafo *)nuevoGrafo
Copia el receptor a *nuevoGrafo*.

copiarSinRepA:

-(void)**copiarSinRepA:** (Grafo *)nuevoGrafo
Copia el receptor a *nuevoGrafo* pero sin repetir nodos y arcos con la misma secuencia.

esPosibleBorrarNodo:

-(BOOL)**esPosibleBorrarNodo:** (Nodo *)unNodo
Devuelve YES si *unNodo* no está conectado con ningún otro no, NO en caso contrario.

estaVacio

(BOOL)**estaVacio**
Devuelve YES si *unNodo* no está conectado con ningún otro no, NO en caso contrario.

generarComplemento

(Grafo *)**generarComplemento**
Genera una copia del grafo con los arcos en dirección invertida.

generarComponentes

(NSMutableArray *)**generarComponentes**
Si en el grafo existen varios subgrafos disjuntos, genera un arreglo de todos los subgrafos.

generarDoble

(Grafo *)**generarDoble**
Genera una copia del grafo con cada arco, duplicado, pero con dirección invertida.

hazLigasEjecutar:con:

(void)**hazLigasEjecutar:**(SEL)unSelector **con:**sender
Ejecuta con cada liga del grafo el selector *unSelector*, con *sender* como parámetro.

hazNodosEjecutar:con:

(void)**hazNodosEjecutar:**(SEL)unSelector con:sender
Ejecuta con cada nodo del grafo el selector *unSelector*, con *sender* como parámetro.

initConImp:

(void)**initConImp:** (ImpGrafo *)imp
Asigna una implantación al grafo

imp

-(ImpGrafo *)**imp**
Devuelve la implantación del grafo.

ligasEntranNodo:

-(NSMutableArray *)**ligasEntranNodo:** (Nodo *)unNodo
Devuelve un arreglo conteniendo todas las ligas que entran a *unNodo*

ligaEntre:y:

(Liga *)**ligaEntre:**(Nodo *)nodoOrigen **y:**(Nodo *)nodoDestino
Devuelve la primera liga que hay entre *nodoOrigen* y *nodoDestino*.

ligasSalenNodo:

-(NSMutableArray *)**ligasSalenNodo:** (Nodo *)unNodo
Devuelve un arreglo conteniendo todas las ligas que salen de *unNodo*

maxNodos

(int)**maxNodos**
Devuelve el número de nodos de un grafo.

nodoCuyoId:

(Nodo *)**nodoCuyoId:**(int)valor
Devuelve el nodo cuyo *id* coincida con *valor*

nodoDestinoEn:y:

-(Nodo *)**nodoDestinoEn:** (int)numeroNodo **y:**(int)numeroArco
Devuelve el nodo adyacente al nodo que se ubica en *numNodo*, en dirección del número de arco *numArco*.

nodoEstaEn:

nodoEstaEn:(int)numeroNodo
Devuelve el nodo que está en la posición *numeroNodo*.

numArcosSalen:

(int)**numArcosSalen:** (int)numNodo
Devuelve el número de arcos que salen del nodo en la posición *numNodo*.

numArcosSalenNodo:

-(int)**numArcosSalenNodo:** (Nodo *)nodo
Devuelve el número de arcos que salen del *nodo*.

renumerar

(void)**renumerar**
Recomienza el secuenciador y empieza a asignar una nueva secuencia a los nodos.

rep:

-(id)**rep**
Devuelve la representación gráfica.

sec

(id)**sec**
Devuelve el secuenciador.

sustituirNodo:con:

-(BOOL)**sustituirNodo:** (Nodo *)nodoActual **con:** (Nodo *)nodoNuevo
Sustituye un nodo dado por otro en el grafo.

terminar

(void)**terminar**
Libera la implantación y representación gráfica.

visitarEnProf:

-(void)**visitarEnProf:** (id)unVisit
Recorre el grafo notificando durante su recorrido a *unVisit.*

6.6 GrafoDep.

Hereda de : Grafo
Declarada en : GrafoDep.h

Descripción de la clase.

El objetivo de esta clase es proveer de un grafo que dependa de otro, por ejemplo, si se desea alterar una sección de un grafo y queremos editarla en una ventana aparte, deseamos que no podamos hacer operaciones que serían incongruentes con el grafo original. GrafoDep proporciona este mecanismo, sobreescribiendo el método *esPosibleBorrarNodo:* de Grafo.

Tipos de Métodos.

Inicializando	initConPadre:

Métodos de instancia.

initConPadre:

-(GrafoDep *)*initConPadre:* (Grafo *)unGrafo
Inicializa un grafo tomando como grafo base *unGrafo.*

6.7 ImpGrafo.

Hereda de : Ninguno
Declarada en : ImpGrafo.h

Descripción de la clase.

La clase abstracta ImpGrafo publica una interfaz con operaciones básicas sobre grafos. El objetivo de ImpGrafo consiste en implantar las operaciones básicas de un grafo, es algo así como el *back-end* de Grafo, los métodos que tiene esta clase coinciden con los de Grafo por la misma razón, aunque no al revés. Para consultar en que consiste cada método consulte los métodos en Grafo.

Implantaciones concretas.

ImpGrafo no puede utilizarse directamente por ser una clase abstracta, en su lugar use ImpGrafoLista, esta clase es una implantación por medio de listas, la interfaz es exactamente la misma.

Tipos de Métodos.

Iniciando	* init
Editando	agregarNodo:
	agregarLiga:entre:y:
	borrarLiga:entre:y:
	borrarLigaEntre:y:
	borrarLigasNodo:
	borrarArcoNodos
	borrarNodo:
	sustituirNodo:con:
Bu'squeda seg'un orden interior	arcoSaleEn:de:
	buscarLiga:entre:y:
	buscarNodo:
	- buscarNumNodo:
	- ligaEntre:y:
	nodoDestinoEn:y:
	nodoEstaEn:
	numArcosSalen:
	numArcosSalenNodo:
Etiquetando	nodoCuyoId:
Operando a nivel global	hazLigasEjecutar:con:
	hazNodosEjecutar:con:
Recorriendo en profundidad	* visitarNodoEnProf:visitante:

Métodos de instancia.

En esta sección solo se describen aquellos métodos(*) que son únicos en ImpGrafo.

init:

-(void)**init**
Inicia el objeto.

visitarNodoEnProf:visitante:

-(void)**visitarNodoEnProf:** (Nodo *)unNodo **visitante:**(id)unVisit
Visita en profundidad el nodo *unNodo* invocando en *unVisit* varios métodos.
Ver tambien: Visitante.

6.8 IterGrafo.

Hereda de : Ninguno
Declarada en : IterGrafo.h

Descripción de la clase.

El objetivo de esta clase es explorar un grafo, las ventajas de hacerlo por medio de esta clase, es que podemos realizar varios recorridos en el grafo, al mismo tiempo, además de contar con la ventaja adicional de no depender de una implantación particular del grafo. Adicionalmente el grafo puede crear "instantes" en el tiempo, es decir momentos del recorrido que puede recuperar más tarde, vease Memento(Souvenir o recuerdo) en [Gam94] y LigaInterna dentro de este mismo capítulo.

Tipos de Métodos.

Crear iterador e iniciando	init: + iteradorConGrafo:
Avanzar en el grafo	primerArcoNodo primerNodo sigArcoNodo - sigNodo - sigNodoDestino

Consultar estado del iterador	estaEnfocado
	esUltimoArcoNodo
	esUltimoNodo
Recuperando partes del grafo	arcoActual
	nodoActual
	nodoDestinoActual
Creando instantes	crearLigaInternaActual

Metodos de clase.

iteradorConGrafo:

+(IterGrafo *)**iteradorConGrafo:** (Grafo *)unGrafo
Crea un iterador en función de *unGrafo.*

Metodos de instancia.

arcoActual

(Liga *)**arcoActual**
Recupera la liga en que está situado el iterador.

crearLigaInternaActual

-(LigaInterna *)**crearLigaInternaActual**
Genera un souvenir o recuerdo de la liga, el cual será conocido como LigaInterna dentro del framework.
Vease tambien: LigaInterna.

estaEnfocado

(BOOL)**estaEnfocado**
Verifica si el iterador está sealando a un número de nodo y arco existente, regresa YES si así es, NO, en otro caso.

esUltimoArcoNodo

-(BOOL)**esUltimoArcoNodo**
Devuelve YES si el iterador está sealando al último arco con respecto al nodo actual. No en caso contrario.

esUltimoNodo

-(BOOL)**esUltimoNodo**
Devuelve YES si el iterador está sealando al último nodo en cuestión. No en caso contrario.

init:

(void)**init:**(Grafo *)unGrafo
Inicializa a un iterador con *unGrafo*.

nodoActual

(Nodo *)**nodoActual**
Devuelve el nodo en que está situado el iterador.

nodoDestinoActual

(Nodo *)**nodoDestinoActual**
Devuelve el próximo nodo al nodo en que está situado el iterador, por el arco en que también está situado.

primerArcoNodo

(void)**primerArcoNodo**
Sitúa al iterador en el primer arco de salida de un nodo.

primerNodo

(void)**primerNodo**
Sitúa al iterador en el primer nodo del grafo.

sigArcoNodo

(void)**sigArcoNodo**
Se mueve al siguiente arco del nodo actual.

sigNodo

(void)**sigNodo**
Se mueve al siguiente nodo del grafo actual.

sigNodoDestino

(void)**sigNodoDestino**
Se mueve al nodo al que apunta el arco actual.

6.9 Liga.

Hereda de : Ninguno
Declarada en : Liga.h

Descripción de la clase.

Esta clase sirve para encapsular la información referente a un arco dentro del grafo, pero independientemente de la conexión que pudiera tener con otros objetos. Liga tiene los siguientes elementos:

- Representación gráfica. Cada una de las ligas tiene un delegado que maneja su representación visual.

Tipos de Métodos.

Inicializando y terminando	init
	terminar
Copiando	clonLiga
Etiquetando	id
	id:
Permitiendo acceso	bloquear

	- desbloquear
	estarBloqueada
Seleccionando	seleccionar
	deseleccionar
	estarSeleccionada
Grabando	grabarA:
Asignando representacion grafica	asignarRep:
	rep

Metodos de instancia.

asignarRep:

-(void)**asignarRep:** (id)rep
Asigna una representación gráfica al receptor.

bloquear

(void)**bloquear**
Marca a la liga como de uso exclusivo de alguna entidad.

clonLiga

(Liga *)**clonLiga**
Genera una copia del receptor.

desbloquear

(void)**desbloquear**
Desmarca la liga, a partir de lo cual se lo puede utilizar.

deseleccionar

-(void)**deseleccionar**
Desmarca al receptor como parte de una selección.

estarBloqueada

(BOOL)**estarBloqueada**
Regresa YES, si está marcada como bloqueada, NO en caso contrario.

estarSeleccionada

-(BOOL)**estarSeleccionada**
Regresa YES, si está marcada como seleccioanda, NO en caso contrario.

grabarA:

(void)**grabarA:**unArchivador
Graba el receptor a *unArchivador*.

id

(long int)**id**
Regresa el número de etiqueta del grafo.

id:

(void)**id:**(long int)valor
Asigna *valor* como número de etiqueta.

init

(void)**init**
Inicializar un objeto.

rep

-(id)**rep**
Regresa la representación gráfica de un objeto.

seleccionar

-(void)**seleccionar**
Marca al grafo como seleccionado.

terminar

-(void)**terminar**
Libera los recursos apartados por el receptor.

6.10 Liga Interna.

Hereda de : Ninguno
Declarada en : Liga.h

Descripción de la clase.

Cuando un objeto de Itergrafo está explorando el grafo, puede crear "instantáneas" del recorrido que está haciendo. LigaInterna es esa instantánea y permite recuperar el contexto de cierta parte de un recorrido. Para una explicación más detallada del patrón de una instantánea, vease la sección Memento de [Gam94].

Tipos de Métodos.

Inicializando	initConGrafo:numNodo:numArco:
Copiando	- clonLigaInterna
Recuperando contexto	liga
	nodoDestino
	nodoOrigen

Metodos de instancia.

clonLigaInterna

(LigaInterna *)**clonLigaInterna**
Este método duplica una liga interna.

initConGrafo:numNodo:numArco:

(void)**initConGrafo:** grafo **numNodo:**(long int)unNumNodo **numArco:**(long int)unNumArco
Cada "instantánea" de IterGrafo, queda perfectamente determinada por el número de nodo y arco en el que se quedo el iterador. Así este método inicializa a LigaInterna. Este método generalmente no se debiera llamar solo, ya que es invocado por crearLigaInterna de IterGrafo.

liga

(Liga *)**liga**
Devuelve la liga en que está ubicado el iterador.

nodoDestino

(Nodo *)**nodoDestino**
Devuelve el nodo destino en que está ubicado el iterador.

nodoOrigen

(Nodo *)**nodoOrigen**
Devuelve el nodo origen en que está ubicado el iterador.

6.11 NavegadorJerar.

Hereda de : Ninguno
Declarada en : NavegadorJerar.h

Descripción de la clase.

Esta clase controla la visualización jerárquica del grafo siendo el delegado de NSBrowser (ver NSBrowser en el AppKit de OPENSTEP).
NavegadorJerar tiene un panel de navegación en el que podemos ver el grafo en una forma jerárquica, donde el usuario puede visualizar el nivel de detalle que requiere.

Tipos de Métodos.

Inicializando	cargarJerar:
Implantando a NSBrowser	browser:numberOfRowsInColumn:column:
	browser:willDisplayCell:atRow:

Métodos de instancia.

browser:numberOfRowsInColumn:column:

(int)**browser:**(NSBrowser *)sender **numberOfRowsInColumn:**(int)column
Implanta esta notificación de NSBrowser.

browser:willDisplayCell:atRow:

(void)**browser:**(NSBrowser *)sender **willDisplayCell:**(id)cell **atRow:**(int)row **column:**(int)column
Implanta esta notificación de NSBrowser.

cargarJerar:

(void)**cargarJerar:**(Diagrama *)unDiagrama
Carga el panel de navegación e inicializa el objeto de la clase.

6.12 Nodo.

Hereda de : Ninguno
Declarada en : Nodo.h

Descripción de la clase.

Esta clase tiene como objetivo encapsular todo lo referente al nodo o vértice de un grafo. Un objeto de esta clase tiene:

- Una representación gráfica que debe respetar el protocolo de RepNodo.h.

- Mecanismos para manejar visitas en profundidad.
- Mecanismos para seleccionar e incrementar(expandir) la selección de un grafo.
- Persistencia al almacenar en la clase ArchivadorGrafo.
- Mecanismo de etiquetación de cada nodo, con el fin de dar una identidad única a cada nodo.

Tipos de Métodos.

Iniciando y terminando	init
	terminar
Copiando	clonNodo
	copiarA:
Permitiendo acceso	bloquear
	desbloquear
	estarBloqueado
Seleccionando	deseleccionar
	estabilizar
	estarExpansion
	estarSeleccionado
	- marcarParaExpandir
	- seleccionar
Grabando	grabarA:
Etiquetando	id:
	id
Representando	asignarRep:
	rep
Verificando si esta compuesto	obtenerCompuesto
Recorriendo en profundidad	empezandoVisita
	estarVisitado
	terminandoVisita
	visitandoHijos

Metodos de instancia.

asignarRep:

-(void)**asignarRep:** (id)rep
Asigna la representación *rep* al nodo.

bloquear

-(void)**bloquear**
Marca al nodo para que solo pueda ser accesado por cierto cliente.

clonNodo

(Nodo *)**clonNodo**
Realiza una copia profunda sobre el nodo.

copiarA:

(void)**copiarA:** (Nodo *)unNodo
Copia, de manera profunda, a *unNodo.*

desbloquear

-(void)**desbloquear**
Desmarca al nodo, permitiendo así, el acceso a otros clientes.

deseleccionar

-(void)**deseleccionar**
Desmarca como seleccionado un cierto nodo.

empezandoVisita

-(void)**empezandoVisita**
Marca al nodo, durante un recorrido en profundidad, indicando que se ha pasado por él, pero no así a sus nodos adyacentes.

estabilizar

-(void) **estabilizar**
Desmarca al nodo, indicando que ya no está en expansión.

estarBloqueado

-(BOOL)**estarBloqueado**
Devuelve YES si el nodo está marcado como bloqueado y NO en caso contrario.

estarExpansion

-(BOOL)**estarExpansion**
Devuelve YES si el nodo está marcado como en expansión y NO en caso contario.

estarSeleccionado

-(BOOL)**estarSeleccionado**
Marca al grafo como parte de una selección.

estarVisitado

-(BOOL)**estarVisitado**
Devuelve NO, si el recorrido en profundidad no ha pasado por el nodo, SI en caso contrario.

grabarA:

(void)**grabarA:**unArchivador
Graba en unArchivador el grafo.

id

(long int)**id**
Devuelve la etiqueta del nodo.

id:

(void)**id:**(long int)valor
Fija a *valor* como etiqueta del nodo.

init

(void)**init**
Inicializa un nodo.

marcarParaExpandir

-(void) **marcarParaExpandir**
Marca al nodo, indicando que se puede marcar como seleccionados, los nodos adyacentes a él.

obtenerCompuesto

-(id)**obtenerCompuesto**
Retorna el receptor si se trata de un nodo compuesto.

rep

-(id)**rep**
Regresa la representación gráfica de un nodo.

seleccionar

-(void)**seleccionar**
Marca a un grafo como seleccionado por un cliente.

terminandoVisita

-(void)**terminandoVisita**
Marca al nodo, durante un recorrido en profundidad, indicando que se ha pasado por él y por todos los nodos conexos a él.

terminar

-(void)**terminar**
Libera todos los recursos alojados por el receptor.

visitandoHijos

-(void)**visitandoHijos**
Marca al nodo, durante un recorrido en profundidad, indicando que se ha pasado por él y en que en el momento se están visitando sus nodos conexos.

6.13 NodoComp.

Hereda de : Nodo
Declarada en : NodoComp.h

Descripción de la clase.

Esta clase tiene como objetivo convertir un subgrafo en un solo nodo, con la posibilidad de poder revertir el proceso.

Tipos de Métodos.

Iniciando	initConGrafo:
Conectando con otros nodos	- conectarConGrafo: registrarLigaEntra:conNodoDestino:por: registrarLigaSale:conNodoOrigen:por:
Reintegrando a forma original	integrarAGrafo: integrarAGrafoLigasEntran: integrarAGrafoLigasSalen:
Recuperando grafo	grafo

Métodos de instancia.

conectarConGrafo:

-(void)**conectarConGrafo:** (Grafo *)unGrafo
Conecta al receptor con *unGrafo*, conservando cualquier liga, que entre desde *unGrafo* hacia el grafo del receptor, y que salga hacia *unGrafo* desde el grafo del receptor.

initConGrafo:

-(void)**initConGrafo:** (Grafo *)unGrafo
Inicia el nuevo nodo con el grafo que va a comprimir.

integrarAGrafo:

-(void)**integrarAGrafo:** (Grafo *)unGrafo
Reintegra el grafo comprimido a *unGrafo*. No destruye al receptor.

integrarAGrafoLigasEntran:

-(void)**integrarAGrafoLigasEntran:** (Grafo *)unGrafo
Reintegra solo las ligas que entran a *unGrafo*.

integrarAGrafoLigasSalen:

-(void)**integrarAGrafoLigasSalen:** (Grafo *)unGrafo
Reintegra solo las ligas que salen desde *unGrafo*.

grafo

-(Grafo *)**grafo**
Devuelve el grafo del nodo comprimido.

registrarLigaEntra:conNodoDestino:por:

-(void)**registrarLigaEntra:** (Liga *)liga **conNodoDestino:** (Nodo *)nodoDestino **por:** (Liga *)ligaNueva
Toma a la *liga* que entra al grafo a comprimir y la guarda, asociándola con el nodo al que llegaba originalmente.

registrarLigaSale:conNodoOrigen:por:

-(void)**registrarLigaSale:** (Liga *)liga **conNodoOrigen:** (Nodo *)nodoOrigen **por:** (Liga *)ligaNueva
Toma a la *liga* que sale del grafo a comprimir y la guarda, asociándola con el nodo del que partía originalmente.

6.14 Operación.

Hereda de : Ninguno
Declarada en : Operacion.h

Descripción de la clase.

Esta clase abstracta provee de una interfaz para manejar transacciones que se puedan hacer y deshacer.

Tipos de Métodos.

Copiando	clonOper
Haciendo y deshaciendo	deshacer
	ejecutar
Terminando	terminar

Métodos de instancia.

clonOper

-**clonOper**
Genera una copia profunda del receptor, debe ser reescrito por la subclase derivada.

deshacer

-(void)**deshacer**
Deshacer la operación realizada. Este método debe ser reescrito para

establecer de qué manera se podrá deshacer la operación.

ejecutar

-(void)**ejecutar**
Ejecuta la operación. El método debe ser reescrito para establecer cómo se va a llevar a cabo la operación.

terminar

-(void)**terminar**
Libera los recursos alojados por el receptor.

6.15 OperGrafo y operaciones concretas.

Hereda de : Operacion
Declarada en : Operacion.h

Descripción de la clase.

La clase OperGrafo es una subclase de Operación pero que está asociada a un grafo en particular. Las clases concretas que se derivan de OperGrafo son hasta el momento: *OperAgregarNodo, OperBorrarNodo, OperSustNodo, OperAgregarLiga, OperBorrarLiga, OperDesplazarSel, OperMacro*, las cuales definen sus formas específicas de inicialización.

Tipos de Métodos.

Dado que los métodos que se describen son de inicialización los describiremos para cada clase concreta en particular.

Métodos de instancia.

asignarGrafo:

-(void)**asignarGrafo:** (Grafo *)unGrafo
Asigna *unGrafo* a la operación con el fin de indicarle que el grafo con que se realizará la operación.

init:conGrafo: (OperAgregarNodo)

-(void)**init:** (Nodo *)nodo **conGrafo:** (Grafo *)unGrafo
Inicializa la operación con el nodo a agregar.

init:conGrafo: (OperBorrarNodo)

-(void)**init:** (Nodo *)nodo **conGrafo:** (Grafo *)unGrafo
Inicializa la operación con el nodo a borrar.

init:con:conGrafo: (OperSustNodo)

-(void)**init:** (Nodo *)unNodoActual **con:** (Nodo *)unNodoNuevo **conGrafo:** (Grafo *)unGrafo
Inicializa la operación se sustituir nodo, registrando el *nodoActual* y el *nodoActual.*

init:entre:y:conGrafo: (OperAgregarLiga)

-(void)**init:**(Liga *)unaLiga **entre:** (Nodo *)unNodoOrigen **y:**(Nodo *)unNodoDest **conGrafo:** (Grafo *)unGrafo
Inicializa la operación de agregar unaLiga *unaLiga* al grafo *unGrafo* entre los nodos *unNodoOrigen* y *unNodoDestino.*

init:entre:y:conGrafo: (OperBorrarLiga)

-(void)**init:**(Liga *)unaLiga **entre:** (Nodo *)unNodoOrigen **y:**(Nodo *)unNodoDest **conGrafo:** (Grafo *)unGrafo
Inicializa la operación de borrar unaLiga *unaLiga* del grafo *unGrafo* entre los nodos *unNodoOrigen* y *unNodoDestino.*

initconSel:desp:conGrafo: (OperDesplazarSel)

-(void)**initconSel:** (Grafo *)unGrafoSel **desp:** (NSPoint *)unDesp **conGrafo:** (Grafo *)unGrafo
Inicia la operación de desplazar un grupo de nodos *unGrafoSel* el desplazamiento indicado por *unDesp* en el grafo *unGrafo.*

init:conGrafo: (OperMacro)

-(void)**init:** (PilaOper *)unaPila **conGrafo:**(Grafo *)unGrafo
Inicializa una operación que contiene otras operaciones que vienen empaquetadas en *unaPila* en el grafo *unGrafo*.

6.16 PilaOper.

Hereda de : Ninguno
Declarada en : PilaOper.h

Descripción de la clase.

La clase tiene como objetivo llevar el control de las operaciones que se van a hacer o deshacer, como su nombre lo indica, se trata de una pila, donde la última operación que se realiza es la primera que se deshace.

Tipos de Métodos.

Inicializando y terminando	init
	reempezar
	terminar
Copiando	- clonPila
Almacenando operaciones	- registrarOperacion:
Recuperando operaciones	- operaciones
Haciendo y deshaciendo operaciones	- deshacerTodo
	- deshacerUltimaOper
	- ejecutar
	hayOperacionesDeshacer
	hayOperacionesRehacer
	rehacerUltimaOper

Metodos de instancia.

clonPila

clonPila
Genera una copia profunda de la pila.

deshacerTodo

-(void)**deshacerTodo**
Deshace todas las operaciones de la pila.

deshacerUltimaOper

-(void)**deshacerUltimaOper**
Deshace la última operaci'on que se realizó.

ejecutar

-(void)**ejecutar**
Ejecuta la todas las operaciones deshechas hasta el momento.

hayOperacionesDeshacer

-(BOOL)**hayOperacionesDeshacer**
Devuelve YES si en la pila no hay más operaciones por deshacer. NO, en caso contrario.

hayOperacionesRehacer

-(BOOL)**hayOperacionesRehacer**
Devuelve YES si hay alguna operación deshecha. NO, en caso contrario.

init

-(void)**init**
Inicializa la pila.

operaciones

-(NSMutableArray *)**operaciones**
Devuelve un arreglo conteniendo todas las operaciones que tiene almacenada la pila.

reempezar

-(void)**reempezar**
Reinicia una pila, ignorando cualquier operación que pudiera tener almacenada.

registrarOperacion:

-(void)**registrarOperacion:**(Operacion *)oper
Agrega una operación al tope de la pila.

rehacerUltimaOper

-(void)**rehacerUltimaOper**
Vuelve a rehacer la última operación deshecha.

terminar

-(void)**terminar**
Libera todas la operaciones que almacena la pila.

6.17 RepGrafo.

Hereda de : Ninguno
Declarada en : RepGrafo.h

Descripción de la clase.

La clase RepGrafo tiene como fin, encapsular toda la representación visual del grafo. La Representación gráfica define los siguientes elementos :

- Características generales de los arcos (longitud de la flecha).

- Características generales de los nodos (radio de cada nodo).

Tipos de Métodos.

Inicializando y terminando	asignarGrafo: init terminar
Copiando	clonRepGrafo
Localizando nodos y ligas	ligaInternaEn: ligaInternaMasCercana: nodoEn:
Modificando representacion	cambiarLargoFlecha: cambiarRadio:
Consultando representacion	centro longFlecha radio - rectanguloLimitante
Dibujando	- dibujar:

Metodos de instancia.

asignarGrafo:

(void)**asignarGrafo:** (Grafo *)repres
Asocia a la representación, el grafo a representar.

cambiarLargoFlecha:

(void)**cambiarLargoFlecha:**(float)valor
Fija el largo de la flecha de cada arco de acuerdo al *valor*.

cambiarRadio:

(void)**cambiarRadio:**(float)valor
Fija el tamaño de el nodo por omisión a *valor*.

centro

(NSPoint)**centro**
Devuelve un punto conteniendo las coordenadas del centro promedio de cada nodo.

clonRepGrafo

(RepGrafo *)**clonRepGrafo**
Genera una copia profunda de RepGrafo.

dibujar:

(void)**dibujar:**sender
Dibuja la representación gráfica del grafo.

init

(void)**init**
Inicializa la representación del grafo.

ligaInternaEn:

(LigaInterna *)**ligaInternaEn:** (NSPoint *)punto
Devuelve la liga interna de la liga que está en el *punto*, nil, si no hay liga.

ligaInternaMasCercana:

(LigaInterna *)**ligaInternaMasCercana:** (NSPoint *)punto
Devuelve la liga interna más cercana al *punto*.

longFlecha

(float)**longFlecha**

nodoEn:

(Nodo *)**nodoEn:**(NSPoint)elPunto

radio

(float)**radio**
Devuelve el radio de un nodo por omisión.

rectanguloLimitante

(NSRect)**rectanguloLimitante**
Devuelve el rectángulo delimitado por el grafo en la vista.

terminar

(void)**terminar**
Libera cualquier recurso alojado por el receptor.

6.18 RepLiga.

Hereda de : Ninguno
Declarada en : RepLiga.h

Descripción de la clase.

La clase RepLiga tiene como misión representar gráficamente una liga, es la que determina cómo se va dibujar cualquier arco del grafo. La representación de la liga toma como parte de la representación :

- El punto en que el arco tiene intersección con cada uno de los nodos que une el arco. Para lo cual es necesario determinar dos ángulos de incidencia. Cada ángulo de incidencia se determina por la línea que une los centros de los nodos y la línea del centro del nodo al punto de intersección, los ángulos de incidencia son así utilizados para inicializar la representación.

Tipos de Métodos.

Inicializando y terminando	asignarLiga: init:anguloOrigen:anguloDestino:conColor:
Copiando	clonRepLiga
Dibujando	dibujar:entre:y: dibujarFlecha:a:deTam: imprimirEtiqueta
Consultando representacion	distanciaA:entre:y: - estarPunto:entre: puntoInter:entre:y:a:
Modificando representacion	asignarColor: - cubrirColor: - descubrir longFlecha:

Métodos de instancia.

asignarColor:

(void)**asignarColor:** (NSColor *)unColor
Asigna *unColor* a la representación.

asignarLiga:

(void)**asignarLiga:** (Liga *)repres
Asocia al receptor, la liga representada.

clonRepLiga

(RepLiga *)**clonRepLiga**
Genera una copia profunda del receptor.

cubrirColor:

-(void)**cubrirColor:** (NSColor *)unColor
Asigna un color a la representación, pero sin cambiar el color original.

descubrir

(void)**descubrir**
Reestablece el color original de la representación.

dibujar:entre:y:

(void)**dibujar:**sender **entre:**nodo1 **y:**nodo2
Dibuja la representación entre dos nodos.

dibujarFlecha:a:deTam:

(void)**dibujarFlecha:** (NSPoint *)punto1 **a:** (NSPoint *)punto2 **deTam:** (float)tam
Dibuja una flecha de tamaño entre dos puntos.

distanciaA:entre:y:

-(float)**distanciaA:**(NSPoint *)punto **entre:** (RepNodo *)nodo1 **y:** (RepNodo *)nodo2
Calcula la distancia entre la representación de dos nodos.

estarPunto:entre:

(BOOL)**estarPunto:** (NSPoint *)punto **entre:** (RepNodo *)nodoOrigen **y:** (RepNodo *)nodoDest
Devuelve YEs si *punto* se encuentra en el segmento de recta entre *nodoOrigen* y *nodoDestino*.

imprimirEtiqueta

-(void)**imprimirEtiqueta**
Despliega la etiqueta que tiene la representación, este método es llamado por dibujar.

init:anguloOrigen:anguloDestino:conColor:

- (RepLiga *)**init:**(float)unaLongFlecha **anguloOrigen:** (float)anguloOrigen **anguloDestino:** (float)anguloDestino **conColor:** (NSColor *)unColor

Inicializa la representación con una longitud de la flecha *unaLongFlecha*, los ángulos de incidencia con el nodo origen y con el nodo destino.

longFlecha:

(void)**longFlecha**: (float)valor
Cambia la longitud de la flecha a *valor*.

puntoInter:entre:y:a:

(void)**puntoInter:**(NSPoint *)punto **entre:** (NSPoint *)origen **y:**(NSPoint *)destino **a:**(float)distancia
Deja en el contenido de *punto*, el punto intermedio entre *origen* y *destino*, con una *distancia* del origen.

6.19 RepNodo.

Hereda de : Ninguno
Declarada en : RepNodo.h

Descripción de la clase.

RepNodo es una clase para representar gráficamente un nodo.

Tipos de Métodos.

Inicializando	asignarNodo: init:conRadio:conColor:
Consultando representacion	anguloEntrePunto:y: - centro - distanciaA: - estarPunto: nodo puntoOrillaEntre:y: puntoOrillaHacia:conAngulo: radio

Copiando	clonRepNodo
Modificando posicion	desplazar:
	estaEn:
	moverA:
Modificando color	asignarColor:
	asignarColorContorno:
	cubrirColor:
	descubrir
	desactivarContorno
	ponerContorno
Modificando forma	pedirRadio:
Dibujando	- dibujar:
	imprimirEtiqueta
Grabando	grabarA:

Métodos de instancia.

anguloEntrePunto:y:

(float)**anguloEntrePunto:** (NSPoint)unPuntoOrigen **y:** (NSPoint)unPuntoDest
Calcula el angulo que se forma el ángulo determinado por el centro del nodo, *unPuntoOrigen* y *unPuntoDest.*

asignarColor:

(void)**asignarColor:** (NSColor *)unColor
Cambia el color del nodo a *unColor.*

asignarColorContorno:

-(void)**asignarColorContorno:** (NSColor *)unColor
Cambia el color de un círculo externo al círculo del nodo.

asignarNodo:

(void)**asignarNodo:** (Nodo *)repres
Asigna la representación al nodo *repres.*

centro

(NSPoint)**centro**
Devuelve el centro del nodo.

clonRepNodo

clonRepNodo
Devuelve una copia profunda del receptor.

cubrirColor:

-(void)**cubrirColor:** (NSColor *)unColor
Cambia el nodo a *unColor*, hasta que se reciba el mensaje *descubrir*.

desactivarContorno

(void)**desactivarContorno**
Quita el círculo externo que rodea al nodo.

descubrir

(void)**descubrir**
Regresa el nodo a su color original.

desplazar:

(void)**desplazar:**(NSPoint)puntoDesp
Desplazar el nodo, un desplazamiento *puntoDesp*.

dibujar:

(void)**dibujar:**sender
Dibuja la representación gráfica.

distanciaA:

(float)**distanciaA:**(NSPoint)elPunto
Calcula la distancia desde el centro del nodo a *elPunto* .

estarPunto:

(BOOL)**estarPunto:** (NSPoint *)unPunto
Devuelve YES si unPunto está dentro de la representación del nodo. NO, en otro caso.

grabarA:

(void)**grabarA:**unArchivador
Graba en *unArchivador* la representaqción del nodo.

imprimirEtiqueta

-(void) **imprimirEtiqueta**
Despliega la etiqueta del nodo.

init:conRadio:conColor:

-(id)**init:** (NSPoint)elPunto **conRadio:**(float)unRadio **conColor:** (NS-Color *) unColor
Inicializa la representación del nodo, como centro *elPunto*, con radio *unradio* y color *unColor*.

moverA:

(void)**moverA:**(NSPoint)elPunto
Situa al receptor en *elPunto*.

nodo

(void)**nodo**
Devuelve el nodo que se está representando.

ponerContorno

(void)**ponerContorno**
Activa el contorno o circulo exterior al nodo.

puntoOrillaEntre:y:

- (NSPoint)**puntoOrillaEntre:** (NSPoint)unPuntoOrigen **y:** (NSPoint)unPuntoDest
Localiza el punto de intersección entre el segmento determinado por *unPuntoOrigen* y *unPuntoDest* con el círculo de unPuntoOrigen, este método supone que unPuntoOrigen se encuentra dentro del círculo del nodo.

puntoOrillaHacia:conAngulo:

(NSPoint)**puntoOrillaHacia:** (NSPoint)unPunto **conAngulo:** (float)angulo
Localiza un punto dentro del círculo del nodo formando un ángulo *angulo* con unPunto.

radio

(float)**radio**
Devuelve el radio del nodo.

radio:

(void)**radio:**(float)valor
Fija el radio a *valor.*

6.20 Secuenciador.

Hereda de : Ninguno
Declarada en : Secuenciador.h

Descripción de la clase.

Esta clase tiene como objetivo proporcionar etiquetas tanto para arcos como para nodos. Las etiquetas que devuelve el secuenciador son consecutivas. Cada grafo tiene un secuenciador, aunque solo uno de ellos puede servir de referencia, decimos entonces que se trata del dueño del secuenciador.

Tipos de Métodos.

Inicializando	initConGrafo
	reempezar
Copiado	clonSec
Consultando	esDueno:
	numArco
	numNodo
Cambiando secuencia	asignarNumArco:
	asignarNumNodo:
	sigNumArco
	sigNumNodo

Métodos de instancia.

asignarNumArco:

-(void)**asignarNumArco:** (int)numArco
Cambia la secuencia de arcos, empezando en *numArco.*

asignarNumNodo:

-(void)**asignarNumNodo:** (int)numNodo
Cambia la secuencia de nodos, empezando en *numNodo.*

clonSec

-(Secuenciador *)**clonSec**
Copia el secuenciador.

esDueno:

-(BOOL)**esDueno:** (Grafo *)unGrafo
Devuelve YES si *unGrafo* es dueño del secuenciador, NO en caso contrario.

initConGrafo:

-(void)**initConGrafo:** (Grafo *)unGrafo
Inicializa el secuenciador, estableciendo a *unGrafo* como dueño.

numArco

-(int)**numArco**
Devuelve el numero del arco actual del secuenciador.

numNodo

-(int)**numNodo**
Devuelve el numero del nodo actual del secuenciador.

reempezar

-(void)**reempezar**
Reinicia tanto la secuencia de arcos como de nodos.

sigNumArco

-(void)**sigNumArco**
Avanza la secuencia de arcos en uno.

sigNumNodo

-(void)**sigNumNodo**
Avanza la secuencia de nodos en uno.

6.21 Visitante.

Hereda de : Ninguno
Declarada en : Visitante.h

Descripción de la clase.

Esta clase abstracta permite visitar en profundidad un grafo, "montando" rutinas durante el recorrido, en base a las cuales se pueden realizar clases concretas que redefinan lo que se va a hacer en esas partes del recorrido.

Al recorrer todo el grafo, se van explorando sucesivamente diversos subárboles, cada vez que se cambia de subárbol, se invoca a *cambiandoArbol.* Posteriormente durante el recorrido de un subárbol en particular, y se llega a un nodo por primera vez, se invoca a *visitarNodo*, luego a *empezarVisitaArcos:* y justo antes de visitar a cada uno de los nodos adyacentes en profundidad, se invoca (por cada nodo) a *visitarAntesNodoOrigen:conArco:yDestino:*. Cuando se termina la visita de algun nodo adyacente en profundidad se invoca a *visitarDespuesNodoOrigen:conArco:yDestino:*. Finalmente al terminar de visitar el árbol, se invoca a *terminarVisita.*

Tipos de Métodos.

Visitando el grafo	cambiandoArbol empezarVisitarArcos: - terminarVisita visitarAntesNodoOrigen:conArco:yDestino: visitarDespuesNodoOrigen:conArco:yDestino: - visitarNodo:

Métodos de instancia.

cambiandoArbol

-(void)**cambiandoArbol**
Invocado cuando se cambia de árbol.

empezarVisitarArcos:

-(void)**empezarVisitarArcos:** (int)numArcos
Invocada antes de visitar a cualquiera de los nodos adyacentes, recibe el numero de arcos que tiene el nodo.

terminarVisita

-(void)**terminarVisita**
Invocado al terminar de visitar un árbol.

visitarAntesNodoOrigen:conArco:yDestino:

-(void)**visitarAntesNodoOrigen:** (Nodo *)unNodo **conArco:**(Liga *)liga **yDestino:** (Nodo *)destino
Invocado antes de visitar un nodo adyacente en profundidad, recibe como parámetros el nodo de dónde se ubica actualmente (*unNodo*), el nodo adyacente(*destino*) y la liga que los une (*liga.*

visitarDespuesNodoOrigen:conArco:yDestino:

-(void)**visitarDespuesNodoOrigen:** (Nodo *)unNodo **conArco:**(Liga *)liga **yDestino:** (Nodo *)destino
Invocado despúes de visitar un nodo adyacente en profundidad, recibe como parámetros el nodo de dónde partió (*unNodo*), el nodo adyacente que visitó (*destino*) y la liga que los une (*liga.*

visitarNodo:

-(void)**visitarNodo:** (Nodo *)nodo
Invocado al visitar un nodo durante el recorrido.

Capítulo 7

Conclusiones

La edición de una gráfica grande es un proceso bastante complejo, que nos hace conscientes de problemáticas bien específicas, las cuales se describen a continuación:

- La visión parcial de la gráfica es fundamental, en la mayoría de las ocasiones no nos sirve visualizar toda la gráfica, por el hecho de ser demasiado grande. Por lo que en muchos casos, debemos hacer una selección de la gráfica. Ahora bien el modo de construir esta selección es difícil de generalizar, sobretodo porque una generalización involucra un decremento en la eficiencia de los algoritmos.

- Un acomodo adecuado nos permite percibir las relaciones entre gráficas, en especial el acomodo en forma de árbol y la coloración por distancia entre nodos llega a ser bastante útil.

- La selección anidada, esto es poder seleccionar una gráfica en base a otra previamente seleccionada es muy útil porque permite ir disminuyendo detalles poco a poco.

- La herramienta puede servir de partida para la construcción de nuevas selecciones y acomodos.

Líneas de investigación.

- La construcción de un compilador eficiente para LGRAF, ya que permitiría realizar consultas flexibles.
- La definición explícita de la relación que existe entre el acomodo de gráficas y las características de la gráfica.
- La implantación de REC como un medio para especificar consultas y para la construcción de gráficas.
- La aplicación de reconocimiento de patrones en la búsqueda de subgráficas.
- La especificación de una taxonomía de gráficas, en relación a su estructura (recordemos que hablamos de gráficas grandes) lo cual nos permitiría la comprensión de gráficas bastante grandes.

Bibliografía

[Cox92] Cox, Brad. Introduccion a la OOP Brad. Cox

[Gam94] Gamma, Erich [et al]. Design Pattern: elements of reusable object-oriented software, Ed. Addison Wesley.

[Lin98] Lin Huang Mao, Eades Peter and Wang Junhu. On-line Animated Visualization of Huge Graphs using a Modified Spring Algorithm, Jounal of Visual Languages and Computing, 1998 (9),623-645.

[Kol86] Kolman, Bernard y Busby, Robert C. Estructuras de matemáticas discretas para la computación, Ed. Prentice Hall, 1986.

[Kon36] König, D. Theory of Finite and Infinite Graphs Ed. Springer Verlag, 1986

[McI87] McIntosh Harold V. Linear Cellular Automata, Universidad Autónoma de Puebla, 1987

[McI91] McIntosh Harold V. Linear Cellular Automata via de Bruijin Diagrams, Universidad Autónoma de Puebla, 1991

[Ric97] Richter Charles, Exploring the Unified Modeling Languaje by Example, OOPSLA 97

[Sal79] Salazar Resines Javier, Teoría de graficas, 1979, Ed. Limusa.

[Shu92] Shu, Nan.C Visual Programming 1988, Ed. Van Nostrand Reinhold

Lista de Figuras

Los abajo firmantes, integrantes de jurado para el examen de grado que sustentará el **Ing. David Cruz Rojas,** declaramos que hemos revisado la tesis titulada:

"Construcción de un editor de gráficas de propósito general" consideramos que cumple con los requisitos para obtener el Grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente,

Dr. Sergio V. Chapa Vergara _____

Dr. Shiguang Ju _____

Dr. José Angel Lodegario Ortega Herrera _____