Tesis de Licenciatura
en Ciencias de la Computación

Trazado de grafos mediante métodos dirigidos por fuerzas: revisión del estado del arte y presentación de algoritmos para grafos donde los vértices son regiones geográficas

Autores

Andrés Aiello
aaiello@dc.uba.ar

Rodrigo Ignacio Silveira
rsilveir@dc.uba.ar

Directores

Manuel Abellanas
Gregorio Hernández Peñalver

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Diciembre 2004

Resumen

El problema del trazado de grafos (o *graph drawing*) consiste en dado un grafo, encontrar una representación gráfica que lo presente de una forma "agradable a la vista" en base a determinados criterios estéticos. Los métodos dirigidos por fuerzas, sobre los cuales se enfoca este trabajo, son de los más usados, y en los últimos diez años ha habido una gran producción de algoritmos basados en estas técnicas.

Este trabajo está compuesto por dos partes. En la primera se hace una minuciosa revisión del estado del arte de los métodos dirigidos por fuerzas, comenzando por las técnicas que le dieron origen y haciendo énfasis en las más recientes (que casi no han sido analizadas). Se presenta un análisis de los principales trabajos en sus diversas variantes, clasificados según su objetivo y se señalan puntos en común, diferencias, y tendencias.

La segunda parte se centra en un problema no explorado de trazado de grafos: el trazado de grafos en los que cada vértice representa una región geográfica. Para este problema se definen criterios estéticos apropiados y se proponen varios algoritmos nuevos para responder a estos criterios, basados en los algoritmos clásicos dirigidos por fuerzas. Los algoritmos propuestos permiten obtener trazados para este problema de mucha más calidad que los que se consiguen con los algoritmos hasta ahora existentes.

Abstract

The problem known as *graph drawing* consists in finding a drawing of a given graph that is visually pleasant, based on some aesthetic criteria. Force-directed methods, which are the focus of this work, are among the most commonly used algorithms, and in the last ten years the graph drawing community has witnessed a major explosion in the number of publications related to these techniques.

This dissertation is divided into two parts. In the first one, a fully comprehensive review of the state of the art of force-directed methods is presented, covering from the early techniques that gave birth to the field, to the latest ones, most of which have not been analysed before. The most relevant force-directed algorithms of the literature are introduced, highlighting similarities and differences, as well as giving insight into new trends.

In the second part of this thesis we concentrate on an unexplored graph drawing problem: the one arising when each vertex of the graph represents a geographical region. For this problem we first define new aesthetic criteria. Secondly, several force-directed algorithms for finding layouts that satisfy those specific criteria are proposed, which successfully obtain much better drawings for this problem than the previous algorithms.

Índice general

Capítulo 1. Introducción **8**
1.1. Contribuciones 9
1.2. Organización del trabajo 10

Capítulo 2. Qué es el trazado de grafos **11**
2.1. Criterios estéticos 12
2.2. Algoritmos para trazado de grafos 14
2.2.1. Algoritmos para clases específicas de grafos 15
2.2.2. Algoritmos para grafos generales 17

I Revisión del estado del arte de los métodos dirigidos por fuerzas 21

Capítulo 3. Introducción a los métodos dirigidos por fuerzas **22**
3.1. Ventajas de los métodos dirigidos por fuerzas 23

Capítulo 4. Algoritmos clásicos **26**
4.1. *Spring embedder* 26
4.2. FR 29
4.3. GEM 33
4.4. Modelando con las distancias teóricas: KK 36
4.4.1. Modelo 37
4.5. Funciones generales de energía: DH 41
4.6. Resortes magnéticos: SM 47
4.7. Un algoritmo incremental: Tu 51
4.8. El método baricéntrico: Tutte 53
4.9. Comentarios generales 56
4.9.1. Comparaciones entre los algoritmos 56
4.9.2. Desventajas de los métodos dirigidos por fuerzas 57

Capítulo 5. Aspectos numéricos **61**
5.1. Modelo de fuerzas vs modelo de energía 62
5.2. El método del gradiente 63
5.3. El método de Newton-Raphson 64
5.4. Tunkelang: optimizando con gradiente conjugado 65
5.5. Resolviendo un sistema lineal: Tutte 67
5.6. Métodos más generales de optimización 69
5.6.1. Algoritmos genéticos 69
5.7. Otras 71

Capítulo 6. Trazados 3D **72**
6.1. Adaptar algoritmos 2D a 3D 73
6.1.1. Adaptación de los algoritmos clásicos 73
6.2. Otros algoritmos 3D 77

Capítulo 7. Trazado de grafos dinámicos **78**
7.1. El mapa mental de un trazado 79
7.2. Agregando dinámica a los algoritmos dirigidos por fuerzas 81
7.2.1. Trazado de grafos *online* 82
7.3. *Framework* general 84

Capítulo 8. Trazados con restricciones **86**
8.1. Vértices con forma y tamaño 86
8.1.1. Adaptación de la longitud ideal de las aristas 87
8.1.2. Adaptación de las fuerzas 88
8.1.3. Adaptación de KK 92
8.1.4. Una solución en tres etapas 93
8.1.5. Uso de campos potenciales 93
8.2. Aristas especiales 94
8.2.1. Grafos con pesos 94
8.2.2. Grafos dirigidos 94
8.2.3. Aristas curvas 95
8.3. Restricciones en la posición de los vértices 98
8.3.1. Restricciones usando fuerzas 99
8.3.2. Restricciones usando una función general de energía 99
8.3.3. Otras formas de implementarlas 101

Capítulo 9. Trazado de grafos con clusters **103**
9.1. Agregar vértices atractores 104
9.2. Dividir y vencer 105
9.3. Grafos con clusters dinámicos 108
9.4. Visualizar los clusters del grafo, sin conocerlos 110

Capítulo 10. Trazado inicial **113**
10.1. Trazado inicial al azar 114
10.2. Trazados iniciales elaborados 115

Capítulo 11. Grafos Grandes **119**
11.1. Algoritmos multidimensionales 119
11.1.1. Gajer: un algoritmo multicapa 120
11.1.2. Proyectando altas dimensiones de forma inteligente 122
11.1.3. Emparejamiento multicapa 124
11.2. Simulación de N cuerpos 127
11.2.1. Variante malla de FR 128
11.2.2. Acercamiento numérico con simulación de N cuerpos . . . 129

11.2.3. FM^3: Fast Multipole Multilevel Method 131

Capítulo 12. Comentarios finales **135**
12.1. Otros algoritmos que vale la pena mencionar 135
12.1.1. Algoritmos generales 135
12.1.2. Otros algoritmos 136

II Algoritmos para grafos donde los vértices son regiones geográficas 139

Capítulo 13. Introducción **140**
13.1. Trabajo previo 141
13.2. Organización de la segunda parte 142

Capítulo 14. Criterios estéticos **143**
14.1. Una primera solución 143
14.2. Análisis de criterios estéticos previos 144
14.3. Nuevos criterios estéticos 146
14.3.1. Criterios para cuando las regiones son segmentos 148

Capítulo 15. Algoritmos propuestos **149**
15.1. Longitud ideal de las aristas 150
15.1.1. Longitudes fijas 150
15.1.2. Longitudes variables 151
15.2. Fuerzas hacia el centro 153
15.3. Minimizar número de cruces entre aristas 156
15.4. Evitar vértices muy cercanos a los bordes de las regiones 158
15.5. Los nodos no deben estar cerca de las aristas - SET 159
15.6. SER^2 y DHR^2 161
15.7. Observaciones para regiones que son segmentos 161
15.8. Otros criterios estéticos 163

Capítulo 16. Resultados **164**
16.1. Resultados cualitativos 164
16.2. Tiempos de ejecución 174

Capítulo 17. Conclusiones **177**
17.1. Conclusiones 177
17.2. Contribuciones principales 177
17.3. Trabajo futuro 178

Apéndice A. Implementación de los algoritmos **182**

Agradecimientos

A Gregorio y Manuel, por introducirnos en la geometría computacional, y por aceptar ser nuestros directores a distancia a pesar de no conocernos.

Andrés y Rodrigo

A mi familia que siempre me ayudó, a Paola que estuvo conmigo y a Rodrigo que pese a todo trabajamos juntos hace cinco años siempre obteniendo buenos resultados académicos y forjando cada vez una mayor amistad.

Andrés

A mi familia, que estuvo desde siempre apoyándome, y a Ariela, que me acompañó en cada etapa de esta tesis, escuchando y aconsejándome.

Rodrigo

Capítulo 1

Introducción

El trazado de grafos (más conocido por su nombre en inglés, *graph drawing*) es un campo de las ciencias de la computación que ha experimentado un tremendo despegue en los últimos quince años. Su objetivo puede ser resumido en dos palabras: visualizar grafos.

Los grafos son entidades matemáticas abstractas, y si bien en un principio uno podría pensar que el interés por dibujarlos (o *trazarlos*, que es el término más apropiado) sólo interesa a un pequeño grupo de gente cercana a las matemáticas y la computación, la realidad es que las aplicaciones del trazado de grafos surgen de áreas tan diversas como la biología o la sociología. Esto se debe a que los grafos constituyen la forma más común de modelar información relacional. Aunque probablemente la persona que visualice la información no esté pensando en un grafo, sino en interacciones entre proteínas, o tal vez, en qué ciudades están conectadas por la línea de tren por la que está circulando.

Cualquiera sea el origen de la información, si se trata de objetos y relaciones entre ellos, es altamente probable que pueda ser modelada por un grafo. Entonces visualizar esta información pasa a transformarse en visualizar un grafo.

El área de la computación del trazado de grafos se dedica a buscar algoritmos para automatizar el trazado. Es un tema que no pertenece a una única rama de la computación, sino que involucra a áreas como algoritmos de grafos, teoría de grafos, geometría computacional, topología y visualización de la información, entre otras.

La amplia variedad de familias de grafos ha hecho que los algoritmos de trazado desarrollados se dividan según el tipo de grafos que permiten visualizar. Como se verá en más detalle en el capítulo 2, se han diseñado algoritmos específicos para dibujar árboles, para grafos dirigidos acíclicos, para grafos planares, y, entre varios más, otros para grafos generales (es decir, para cualquier tipo de grafo). Dentro de éstos, los más importantes son una familia de métodos conocidos como "dirigidos por fuerzas".

Los métodos dirigidos por fuerzas son hoy en día los más usados para dibujar grafos generales, ya que dan buenos resultados, son sencillos de implementar y son muy flexibles, por lo que pueden ser fácilmente adaptados a aplicaciones concretas con requerimientos de visualización específicos.

El primero de estos algoritmos fue publicado hace veinte años. Posteriormente, a fines de los ochenta y comienzos de los noventa, fueron desarrollados varios algoritmos más, basados en las mismas ideas (llamaremos a estos primeros algoritmos "clásicos"). Fue a partir de mediados de los noventa que se produjo una explosión de publicaciones y desarrollos de algoritmos de trazado construidos sobre los métodos iniciales. Los algoritmos dirigidos por fuerzas clásicos fueron mejorados, extendidos y adaptados a diversas situaciones y para cientos de usos

específicos.

Este abrupto crecimiento acompañó al crecimiento del trazado de grafos, cuyos dos primeros libros sobre el tema fueron publicados hace menos de cinco años ([DETT99], [KW01]). Cada uno de estos incluye un capítulo completo dedicado a los métodos dirigidos por fuerzas, pero que se limita a presentar los algoritmos clásicos y a comentar algunas extensiones posibles.

La impresionante cantidad de variantes de algoritmos dirigidos por fuerzas producidas ha hecho que exista un gran número de herramientas disponibles para el trazado de grafos generales: los hay que permiten trazar grafos en dos dimensiones o en tres, con restricciones sobre los vértices o con vértices de tamaños variados, con aristas rectas o curvas, para grafos pequeños o de millones de nodos.

Ninguna otra técnica de trazado de grafos cuenta con tantas posibilidades y opciones. Sin embargo, al mismo tiempo que crece el número de algoritmos, crece el desorden y la dificultad para mantenerse al día sobre qué se puede lograr con estos algoritmos. Los libros que de DiBattista [DETT99] y Kaufmann [KW01] sólo dan una introducción a la gran variedad de métodos disponibles. Más aun, muchos otros fueron presentados después de la publicación de estos libros, así que la única forma de llegar a ellos es accediendo a los artículos originales. No existe ningún trabajo que presente una imagen amplia y actualizada de cuál es el estado del arte en los métodos dirigidos por fuerzas. La primer parte de este trabajo pretende contribuir a revertir esta situación.

La segunda parte se centra sobre un problema no explorado del trazado de grafos: trazar grafos donde cada vértice representa a una región geográfica. El problema tiene diversas aplicaciones prácticas, pero no cuenta con algoritmos específicos y, como se verá, tiene varias características que lo hacen un problema de interés.

1.1. Contribuciones

Este trabajo está compuesto por dos partes, siendo cada una un aporte en sí misma.

En la primera hacemos una revisión exhaustiva del estado del arte de los métodos dirigidos por fuerzas, algo muy necesario pero que hasta ahora estaba ausente.

Partiendo de la presentación en detalle de los algoritmos clásicos, sobre los cuales se construirán después todos los otros, avanzamos luego hacia adaptaciones más concretas y menos analizadas en la literatura, llegando hasta las más recientes publicaciones sobre el tema.

Se presentan los algoritmos clasificados según su objetivo pero al mismo tiempo se da una visión global y contextual de cada uno. Además de señalar similitudes y diferencias, se incluyen varias secciones "transversales" que pretenden agregar más luz a la comprensión de estos métodos.

La segunda parte del trabajo se enfoca en un problema concreto y no explorado: cómo visualizar grafos en los que cada vértice representa (y está restringido a) una región geográfica.

Por un lado se plantea (y se responde) la pregunta de cuáles son los criterios estéticos que debe cumplir el trazado de esta clase de grafos, ya que no son los mismos que para grafos generales. Como parte de la respuesta se proponen nuevos criterios estéticos para el problema.

En base a esto se presentan nuevos algoritmos dirigidos por fuerzas que extienden algoritmos existentes de manera de poder responder a estos nuevos criterios. Varias alternativas son propuestas y discutidas.

Además se presenta una implementación de los algoritmos en Java, que permite evaluar empíricamente los resultados, así como comparar los distintos métodos.

1.2. Organización del trabajo

Este trabajo está organizado de la siguiente manera: el próximo capítulo provee una visión global del trazado de grafos, con los distintos tipos de algoritmos que existen, de manera de mostrar el contexto en el que se encuentran los métodos dirigidos por fuerzas.

Los capítulos que siguen constituyen la primer parte del trabajo, y se ocupan exclusivamente de los métodos dirigidos por fuerzas. Primero, en el capítulo 4 se presentan los algoritmos clásicos, seguido de un capítulo sobre los aspectos numéricos de estos algoritmos. Los capítulos siguientes analizan algoritmos para problemas más concretos: trazados en tres dimensiones, grafos dinámicos, trazados con restricciones y con *clusters*. Luego sigue el capítulo 10 dedicado a analizar un factor muy importante de estos métodos que es el trazado inicial. El capítulo 11 se ocupa de algoritmos para el trazado de grafos grandes, con miles o hasta millones de nodos. Por último, algunos otros algoritmos varios son comentados brevemente en el capítulo 12.

La segunda parte del trabajo comienza con una introducción al problema y en el segundo capítulo un análisis de los criterios estéticos que se aplican. En base a éstos, el capítulo siguiente propone varios algoritmos para lograr trazados estéticamente buenos. A continuación se presenta un capítulo con resultados obtenidos con la implementación de los algoritmos.

Finalmente, el capítulo 17 presenta las conclusiones de esta tesis y algunas direcciones de trabajo futuro.

Capítulo 2

Qué es el trazado de grafos

Graph drawing is the best possible field I can think of: it merges aesthetics, mathematical beauty and wonderful algorithms. It therefore provides a harmonic balance between the left and right brain parts.

Donald Knuth, Simposio Graph Drawing '96

En este capítulo presentaremos con más detalle el problema del trazado de grafos. Definiremos con más precisión qué propiedades se buscan en el dibujo de un grafo, y daremos una recorrida rápida por los principales tipos de algoritmos disponibles.

Un grafo G consiste de un conjunto de nodos o vértices V y un conjunto de ejes o aristas E. Usualmente se lo denota $G = (V, E)$. Las aristas son pares de vértices (u, v) con $u, v \in V$. Si $(u, v) \in E$, se dice que u y v están conectados o son adyacentes. Un trazado o dibujo de un grafo $G = (V, E)$ usando segmentos de recta para las aristas es una asignación de una posición en el plano[1] a cada vértice en V. En las versiones más simples, los vértices son dibujados como puntos y las aristas como segmentos entre vértices. De aquí en más supondremos que el trazado usa segmentos para las aristas, excepto cuando se aclare lo contrario. También se supondrá que el grafo es conexo, ya que si el grafo a trazar no lo es, se puede trazar cada componente conexa por separado.

Los grafos sólo contienen información relacional entre los vértices, así que en principio no tienen ninguna forma "natural" de ser dibujados. Sin embargo, de las infinitas formas de trazar un grafo en el plano, algunas son claramente peores que otras, siempre teniendo en cuenta que el objetivo es visualizar el grafo.

En la Figura 2.1, podemos ver dos trazados de un mismo grafo. El de la derecha muestra bien la estructura del grafo, un ciclo de siete vértices más una arista conectando los vértices 2 y 7. En el de la izquierda, en cambio, es mucho más difícil distinguir cómo se relacionan los vértices. Indudablemente, el trazado de la derecha es mejor que el de la izquierda, así que es importante estudiar qué propiedades tiene que lo hace mejor que el otro. Esto lo haremos en la próxima sección sobre criterios estéticos.

Antes debemos aclarar que existen distintas convenciones acerca de cómo trazar un grafo. Las más comunes son:

- **Trazados rectos**. Cada arista es representada como un único segmento de recta. Es la que usaremos aquí a menos que se aclare otra cosa.

[1] Por ahora supondremos trazados en el plano. Más adelante se tratarán los trazados en el espacio.

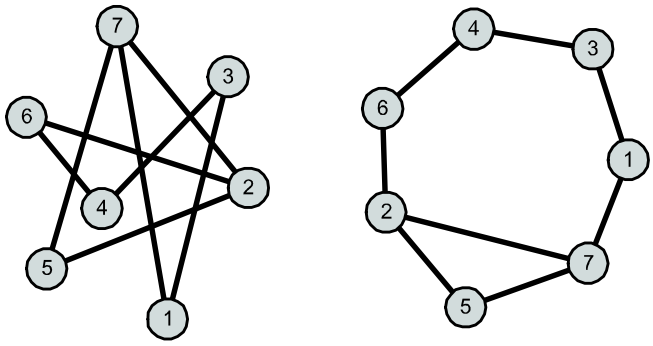


FIGURA 2.1. Dos trazados distintos del mismo grafo

- **Trazados poligonales**. Cada arista se representa por una cadena poligonal formada por uno o varios segmentos. Los extremos de los segmentos que no son vértices del grafo se denominan codos del trazado.
- **Trazados ortogonales**. Cada arista es una cadena de segmentos horizontales y verticales alternados.
- **Trazados planos**. No hay cruces de aristas. Sólo para grafos planares.
- **Trazados hacia arriba (*upward*)**. Para grafos dirigidos. Las aristas dirigidas (arcos) se representan generalmente como segmentos rectos, siempre subiendo en dirección vertical.
- **Trazados de malla**. Todas las posiciones, tanto de los vértices como de los codos, si los hay, tienen coordenadas enteras.

En la Figura 2.2 se ilustran los primeros cinco. Además los trazados pueden estar en el plano o en el espacio. Por ahora supondremos que son en el plano y que los vértices se dibujan como puntos.

2.1. Criterios estéticos

Una vez decidida cuál de las convenciones anteriores se usará, debe decidirse qué propiedades se buscan en el trazado desde el punto de vista de la visualización ¿Qué distingue a un trazado bueno de uno malo? Probablemente dependa de la aplicación. Sin embargo, existen propiedades "buenas" que suelen aplicarse a la mayoría de los grafos y aplicaciones de trazado para "consumo humano". Estas propiedades se conocen como **criterios estéticos**.

Los criterios estéticos más comunes son los siguientes [KW01], [Tun99b]:

- **Minimización de cruces**. Los cruces de aristas hacen más difícil distinguir las conexiones entre vértices, e inclusive a veces dan una falsa sensación de que hay vértices donde no los hay. Unos pocos cruces no suelen ser graves, pero a medida que el número aumenta, el trazado puede tornarse totalmente incomprensible.

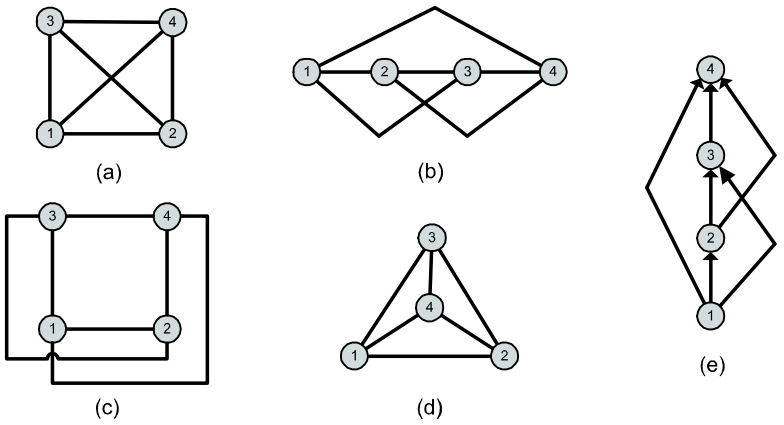


FIGURA 2.2. Distintas convenciones para dibujar grafos: (a) recto, (b) poligonal, (c) ortogonal, (d) plano y (e) hacia arriba.

- **Minimización de codos**. Cuando las aristas están compuestas por varios segmentos los giros o codos en los puntos de conexión las hacen más difíciles de seguir. En los trazados ortogonales este criterio es particularmente importante.
- **Minimización de área**. Los trazados donde los vértices ocupan el espacio de dibujo con una densidad homogénea suelen ser mejores. Otro criterio relacionado es el de **distribuir los vértices uniformemente** en el espacio de dibujo, muy importante en los métodos dirigidos por fuerzas.
- **Maximización de ángulo entre aristas**. Las aristas incidentes a un mismo vértice con ángulos pequeños dificultan la visualización. Lo ideal es que las aristas mantengan el ángulo más grande posible. Este criterio también se conoce como **maximizar la resolución angular**.
- **Simetría**. Mostrar la simetría presente en el grafo es otra propiedad importante de un buen trazado.
- **Longitud uniforme de aristas**. Minimizar la variación entre las longitudes de las aristas también facilita la comprensión de la estructura del grafo. En particular, las aristas largas son difíciles de seguir, siendo generalmente perjudiciales. Criterios relacionados son **minimizar la longitud máxima de las aristas** y **minimizar la suma de sus longitudes**.

Debe aclararse que otros criterios son concebibles. Por ejemplo, en [CSP96] se presenta una lista con diecinueve criterios distintos, que incluye muchos específicos del trazado de árboles. Sin ir más lejos, en la segunda parte de este trabajo se analizan los criterios propios del problema de trazar grafos donde los vértices representan regiones geográficas, y se discuten algunos otros criterios generales (ver capítulo 14).

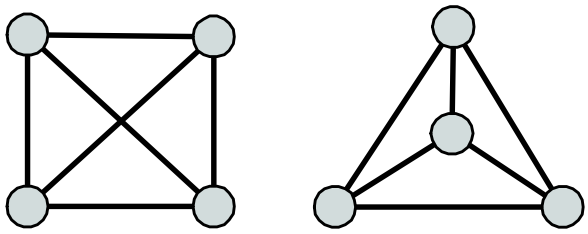


FIGURA 2.3. Dos criterios en conflicto: mostrar simetría (izq.) y evitar cruces de aristas (der.).

No se puede decir que algunos criterios siempre sean más importantes que otros, ya que eso depende mucho de la aplicación. En realidad tampoco ha habido muchos estudios que corroboren la importancia de estos criterios, o que confirmen que no hay otros tan o más importantes. Unas de las pocas pruebas al respecto son las realizadas por Purchase [PCA02] [Pur02], donde se concluyó que minimizar el número de cruces, mostrar la simetría del grafo y minimizar el número de codos son los criterios identificados como más importantes.

Los criterios estéticos son muy importantes en el trazado de grafos porque son los que definen el objetivo de cada algoritmo: qué criterios intenta cumplir. Lamentablemente, hay criterios que son muy difíciles de cumplir con exactitud (encontrar un trazado ortogonal que minimice el número de codos es NP-Hard, al igual que minimizar el número de cruces [Man90]) y muchas veces los criterios son incompatibles entre sí. En la Figura 2.3 se puede ver la incompatibilidad entre mostrar simetría y minimizar el número de cruces. Para este grafo, ambos criterios entran en conflicto. En este caso, un trazado será mejor que otro dependiendo de cuán importante sea para el usuario cada criterio. Para la mayoría de los criterios estéticos el algoritmo de trazado intentará encontrar dibujos que los cumplan "lo mejor posible", a través de algún tipo de heurística.

2.2. Algoritmos para trazado de grafos

En esta sección recorreremos los principales algoritmos para el trazado de grafos, con el doble propósito de presentar las distintas opciones existentes y mostrar el contexto en el que se ubican los algoritmos dirigidos por fuerzas, de los cuales se ocupa este trabajo. Desde ya aclaramos que esto no pretende cubrir todos los tipos de algoritmos existentes, sino simplemente dar una idea de algunos otros enfoques para este problema.

Una clasificación muy usada de métodos es según el tipo de grafo que permiten trazar. En un primer nivel, distinguiremos entre dos grupos: los que trabajan sobre un tipo particular de grafos (árboles, DAGs, etc.) y los que no imponen ninguna restricción sobre el tipo de grafo (ortogonales, dirigidos por fuerzas, etc.).

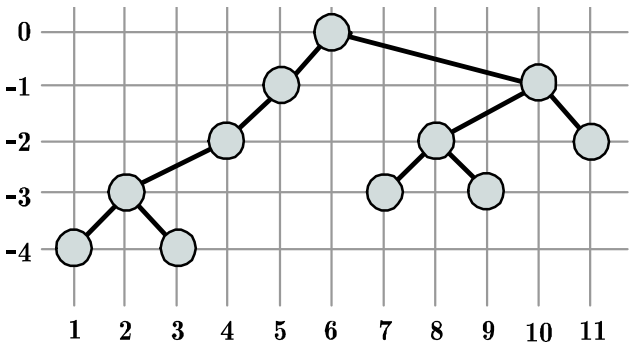


FIGURA 2.4. Resultado de usar un sencillo algoritmo para trazar árboles.

2.2.1. Algoritmos para clases específicas de grafos

Arboles. Los árboles son una de las clases de grafos más usadas, sobre todo en la computación. Además, a diferencia de otros tipos de grafos, hay cierta convención respecto a cómo dibujarlos: la raíz arriba, los hijos debajo, etc. Las principales convenciones para árboles con raíz son [Tun99b]:

1. La distancia entre un vértice y su raíz debe ser proporcional a la distancia teórica[2] entre los dos vértices.
2. Los vértices en un mismo nivel deben mantener una separación mínima para no solaparse.
3. Los vértices padres deben estar centrados respecto de sus hijos.
4. Las aristas no deben cruzarse.
5. Subárboles isomorfos deben trazarse de la misma forma.

Debido a la estructura particular de los árboles, hay muchos algoritmos para trazarlos, que van desde algunos muy sencillos hasta otros bastante complicados.

Uno de los más sencillos, para árboles con raíz, es el siguiente: asignar a cada vértice v la posición $(o_v,-h_v)$, donde o_v es el orden que tiene el vértice en el árbol[3] y h_v es el nivel en el que está v (su distancia a la raíz). En la Figura 2.4 puede verse un ejemplo. La desventaja de este método es que el área del dibujo puede llegar a ser cuadrática si el árbol no es balanceado. Los métodos de trazado de árboles suelen ocuparse de obtener mejores cotas en criterios como el área o la proporción alto/ancho.

Uno de los algoritmos más conocidos para trazado de árboles es el de Reingold y Tilford [RT81], que intenta satisfacer las convenciones al mismo tiempo que

[2]La distancia teórica entre dos vértices de un grafo es la longitud del camino mínimo entre ambos.

[3]Esto puede definirse como la posición relativa, de izquierda a derecha, en la que queda el vértice en el recorrido *in-order* del árbol.

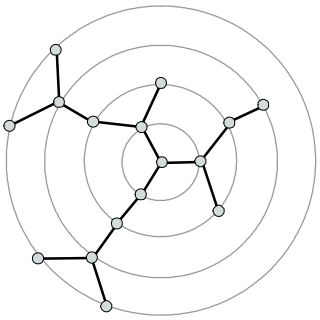


FIGURA 2.5. Trazado radial de un árbol.

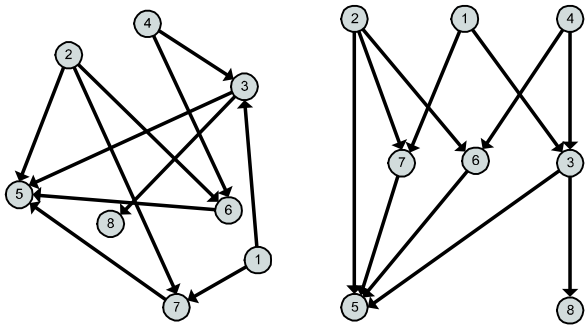


FIGURA 2.6. Un grafo dirigido (izq.) y un posible trazado en capas (der.).

minimizar el ancho del árbol. Tiene una complejidad lineal y obtiene trazados de área cuadrática y proporción alto/ancho lineal.

También existen algoritmos para dibujar árboles libres (árboles sin raíz). Por empezar siempre es posible elegir un vértice como raíz y usar uno de los otros algoritmos. Otra opción para los árboles libres es dibujarlos radialmente, como en el algoritmo de Eades [Ead92]. Primero se toma como raíz un vértice en el centro del árbol y se van colocando los otros vértices en círculos concéntricos alrededor de la raíz. Un ejemplo puede verse en la Figura 2.5.

Grafos dirigidos en capas. La forma más común de trazar grafos dirigidos es en capas. Esto se debe a que los grafos dirigidos que surgen en muchas aplicaciones tienen una dirección natural de flujo, es decir que pueden ser dibujados con (casi) todos los arcos en una misma dirección. Generalmente la convención establece dibujarlos apuntando hacia abajo. Un ejemplo de estos trazados puede verse en la Figura 2.6.

Los algoritmos más comunes para este tipo de trazados trabajan en capas, y usualmente se los organizan en cuatro etapas:

1. **Eliminar ciclos**. Se invierte la dirección de algunas aristas de manera que desaparezcan los ciclos. Esto hace que todos los arcos puedan estar apuntando hacia abajo, necesario para las próximas etapas. Lo ideal es que el número de arcos invertidos sea el menor posible.

2. **Asignar vértices a capas**. El espacio de dibujo se supone atravesado por capas (líneas horizontales) sobre las cuales se ubican los vértices. Cada vértice debe ser ubicado sobre una capa (esto define su coordenada y). Lo importante al hacer esto es que todas las aristas queden apuntando hacia abajo. Esto siempre es posible ya que los ciclos fueron removidos. Algunos algoritmos requieren que ninguna arista cruce por encima de una capa (es decir que no haya aristas entre vértices de capas no contiguas). Si éste llega a ser el caso, se agregan vértices ficticios en las capas intermedias (con respectivas aristas).

3. **Reducir cantidad de cruces**. Para cada capa, un ordenamiento de los vértices es buscado de manera de que el número de cruces de aristas sea lo menor posible. Esto define la posición relativa de los vértices en la capa (notar que la posición exacta en x todavía no fue fijada).

4. **Asignar coordenada x a los vértices**. De manera de que la distancia entre vértices de una misma capa sea adecuada, evitando solapamientos. La posición de cada vértice ya quedó definida. Ya se pueden dibujar las aristas (volviendo a dar vuelta las invertidas en el primer paso).

Este es sólo el esquema general de este tipo de algoritmos. En cada etapa hay muchas cuestiones importantes que se deben decidir, como qué criterio usar para elegir las aristas que deben ser invertidas o cómo elegir el ordenamiento del paso 3. Cada uno es un problema en sí mismo con muchas variantes distintas. El algoritmo más usado que sigue esta línea es el de Sugiyama et al. [STT81]. Para una revisión detallada sobre esta clase de algoritmos recomendamos [KW01].

Grafos planares. El problema de dibujar grafos planares es uno de los que lleva más tiempo siendo estudiado. Obviamente, el objetivo es obtener trazados que no tengan cruces.

Existen muchos algoritmos. Algunos producen trazados convexos (todas las caras son polígonos convexos), para grafos biconexos. (Un algoritmo que produce trazados convexos para grafos triconexos es el de Tutte, descripto en la sección 4.8).

Otra familia de métodos para trazar grafos planares son los basados en ordenamientos canónicos. Básicamente funcionan armando una estructura de datos especial en la que se van insertando los vértices con cierto orden. Dos exponentes muy conocidos de estos algoritmos son el de De Fraysseix et al. [dFPP90] y el algoritmo baricéntrico de Schnyder [Sch90]. Lamentablemente son algoritmos complicados como para explicarlos en esta breve introducción. El lector interesado puede consultar [KW01] para una presentación más detallada sobre el tema.

2.2.2. Algoritmos para grafos generales

Trazado ortogonal. Los trazados ortogonales son aquellos en los que las aristas son dibujadas como cadenas de segmentos verticales y horizontales (alternados). Esta

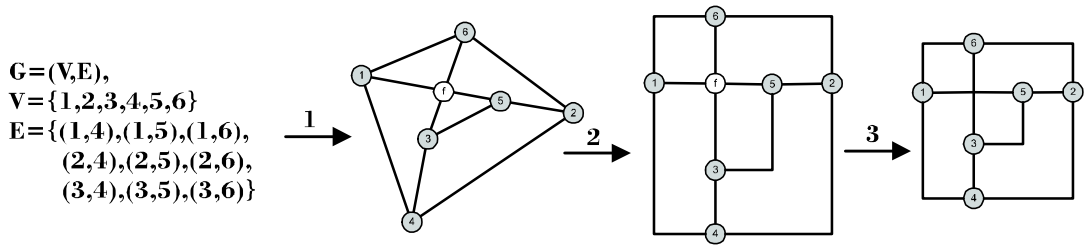


FIGURA 2.7. Trazado de un grafo en tres etapas: (1) topología, (2) forma y (3) compactación.

clase de trazados está íntimamente relacionada con el diseño de circuitos VLSI, para lo cual muchos algoritmos han sido presentados.

Un enfoque en tres etapas presentado en [DETT99] es el llamado *topology-shape-metrics*. El proceso de obtención del trazado ortogonal se divide en tres fases, resumidas a continuación:

1. La primera etapa se encarga de la **topología**. El grafo es hecho planar reemplazando cruces de aristas por vértices ficticios. Una vez hecho esto se calcula un trazado plano de este nuevo grafo (que no va a ser ortogonal).

2. La segunda se encarga de darle la **forma ortogonal**. Los vértices son alineados y se asigna cada arista del trazado anterior a una cadena de segmentos horizontales y verticales, con la intención de minimizar el número de codos.

3. La última etapa es **compactación**. El objetivo es minimizar el área, compactando el trazado todo lo posible, y conservando la forma del paso anterior.

Las tres etapas se ilustran en la Figura 2.7 . Nuevamente se omiten los detalles técnicos de cómo llevar a cabo cada etapa. Otros algoritmos de trazado ortogonal que no siguen este esquema también son posibles (ver [KW01]).

Algoritmos dirigidos por fuerzas. Los algoritmos dirigidos por fuerzas producen trazados para grafos generales con aristas que son segmentos de recta[4]. Reciben su nombre de que muchos de ellos modelan al grafo como un sistema de fuerzas en el que interactúan vértices y aristas. En los ejemplares típicos de estos algoritmos para hallar el trazado se realiza una simulación de este sistema físico, que va moviendo las posiciones de los vértices según las fuerzas que los afectan. Esta simulación se mantiene hasta alcanzar un sistema en equilibrio, y las posiciones con que terminan los vértices en ese equilibrio son usadas para el trazado. Esto es una descripción muy breve y simplificada. Toda la primera parte de este trabajo se dedica a estudiar en profundidad a los métodos dirigidos por fuerzas.

[4]Otros tipos de aristas son posibles, ver capítulo 8.

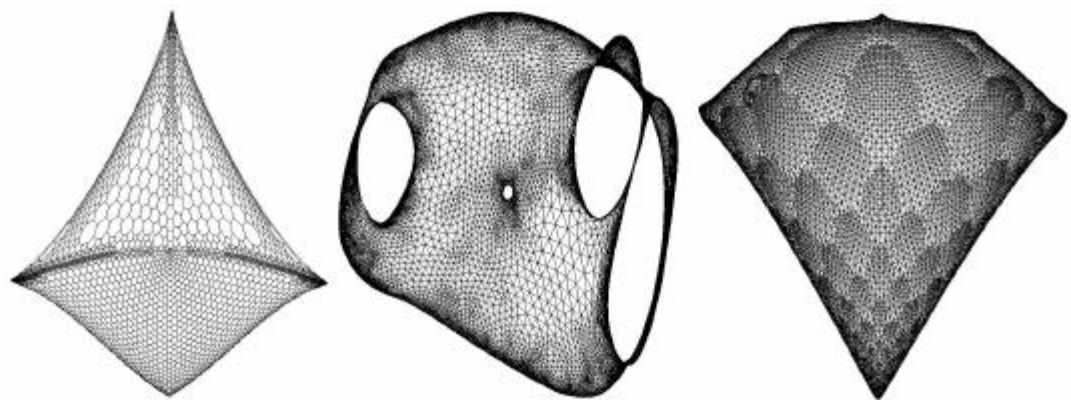


FIGURA 2.8. Ejemplos de trazados de grafos muy grandes (alrededor de 10.000 vértices) realizados con técnicas espectrales [Kor03].

Algoritmos espectrales. Los algoritmos de trazado espectral son un conjunto de técnicas para trazar grafos generales (no dirigidos) usando aristas que sean segmentos de recta.

El trazado del grafo es calculado usando ciertos autovectores de una matriz asociado al grafo (de ahí recibe su nombre). Si bien el enfoque espectral se remonta al trabajo de Hall [Hal70] en la década del setenta, ha pasado casi desapercibido. Los métodos espectrales recién han sido redescubiertos en los últimos años, principalmente gracias a Harel y Koren [KCH01], [Kor03], [KH04].

Estos algoritmos tienen dos características importantes: minimizan una función de energía que permite ser optimizada de forma exacta (en el caso de los otros algoritmos, como los dirigidos por fuerzas, lograr esto es NP-Hard), y segundo, son extremadamente rápidos.

Minimizar la energía de Hall [Hal70] de un trazado produce el efecto de intentar acortar las aristas en proporción al peso de cada una, y muchas veces esto produce trazados con calidad estética aceptable (en general, no tan buena como la obtenida con los métodos dirigidos por fuerzas convencionales). Algunos ejemplos de trazados de grafos grandes producidos minimizando esta energía pueden verse en la Figura 2.8 . De esta manera, el objetivo de un algoritmo que utiliza esta energía es encontrar un trazado de energía mínima.

Explicaremos brevemente cuál es la idea básica de estos algoritmos, en particular el propuesto en [Kor03]. La clave es la relación entre esta energía, y los autovectores de cierta matriz. Para eso son necesarias algunas definiciones.

La matriz de adyacencia de $G = (V, E)$ es una matriz simétrica de $n \times n$ ($n = |V|$) definida como:

$$A_{ij} = \begin{cases} 0 & i = j \\ w_{ij} & i \neq j \end{cases} \qquad i, j = 1, ..., n$$

donde w_{ij} es el peso de la arista $(i, j) \in E$ ó 0 si los vértices no están conectados.

El laplaciano de G es otra matriz simétrica de $n \times n$ definida como

$$L_{ij} = \begin{cases} \deg(i) & i = j \\ -w_{ij} & i \neq j \end{cases} \qquad i, j = 1, ..., n$$

donde $\deg(i)$ es el grado del vértice i.

El laplaciano tiene muchas propiedades interesantes, siendo la más importante en este contexto que se puede obtener un trazado que minimiza la función de energía de Hall calculando los autovectores correspondientes al segundo y tercer autovalor de L (ordenados por módulo en orden creciente).

Por lo tanto la base de los algoritmos espectrales es encontrar dos autovectores del laplaciano del grafo. Obviamente esto es una explicación simplificada, pero la esencia es esa. Lo bueno es que existen métodos muy eficientes de lograr esto, uno de ellos el propuesto en [Kor03].

Si bien no parece que los algoritmos espectrales vayan a reemplazar a los otros métodos de trazado descriptos en este capítulo, sí son candidatos a ser usados para el trazado de grafos grandes, como los algoritmos dirigidos por fuerzas que se describen en el capítulo 11. Como quedará claro luego de profundizar en los algoritmos dirigidos por fuerzas, los métodos espectrales están bastante relacionados con éstos.

Parte I

Revisión del estado del arte de los métodos dirigidos por fuerzas

Capítulo 3

Introducción a los métodos dirigidos por fuerzas

Los métodos de trazado de grafos dirigidos por fuerzas son una familia de algoritmos que usan analogías físicas para dibujar el grafo. Tienen como denominador común las siguientes características:

1. Modelan al grafo como un sistema físico.
2. El trazado del grafo es obtenido buscando un *equilibrio* del sistema físico.

Una tercer característica que se puede agregar, si bien no forma parte de la definición estándar, pero que se verifica en la mayoría de los casos, es que se aplican a grafos generales, es decir, los algoritmos no se basan en ninguna propiedad estructural del grafo (por ej., su planaridad), sino que se aplican a cualquier tipo de grafo.

El criterio para determinar cuándo un método es dirigido por fuerzas es amplio y poco preciso. Como se verá a lo largo de este trabajo, existen muchos algoritmos que no encajan perfecto en alguna de estas condiciones, pero igualmente son considerados dirigidos por fuerzas.

Antes de proseguir, debe aclararse que el nombre "dirigido por fuerzas", por el cual tradicionalmente se conoce a estos algoritmos en el área de *graph drawing*, no es el ideal, ya que la supuesta "física" presente en los algoritmos englobados bajo esta denominación dista mucho de la que estudian los físicos. Un nombre más apropiado para estos algoritmos, utilizado por Brandes en [KW01], es el de "trazado de grafos por analogías físicas".

La mayoría de los algoritmos dirigidos por fuerzas consisten de dos partes:

- Un **modelo** físico del grafo, que representa a los elementos del grafo (vértices y aristas) junto a los criterios estéticos que se desean obtener del dibujo.
- Un **algoritmo** para encontrar un equilibrio del sistema físico, que se corresponderá, en principio, con un trazado del grafo estéticamente agradable.

Los modelos físicos más comunes son los que consisten de un sistema de fuerzas (donde generalmente se definen fuerzas que actúan entre los vértices del grafo), en cuyo caso el objetivo del algoritmo es encontrar un equilibrio para este sistema de fuerzas, es decir, una posición para cada vértice, de manera que el total de la fuerza ejercida en cada vértice sea cero. El mejor ejemplo de este tipo de modelo es el usado en los algoritmos, derivados del *spring embedder* de Eades [Ead84], donde

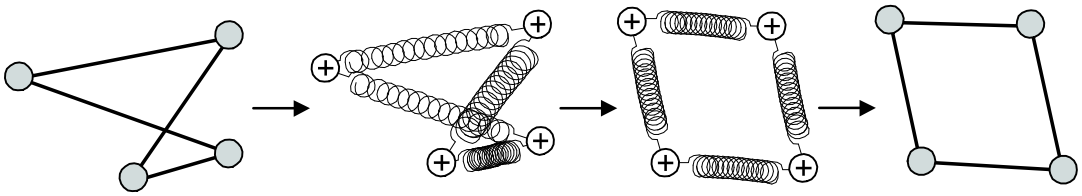


FIGURA 3.1. Un algoritmo basado en resortes.

las fuerzas son resultado de colocar resortes entre cada par de vértices conectados por una arista, junto a fuerzas repulsivas entre todos los vértice que no están conectados. En la Figura 3.1 se ilustra esta analogía.

Otra forma de definir el modelo del grafo, en lugar de usando un sistema de fuerzas, es a través de un sistema de energía. En este caso, el propósito del algoritmo es encontrar una disposición para los vértices del grafo tal que la energía total del sistema sea mínima. El principal exponente de los algoritmos basados en este modelo es el de Kamada y Kawai [KK89].

3.1. Ventajas de los métodos dirigidos por fuerzas

Los métodos dirigidos por fuerzas se han convertido en la técnica más popular para dibujar grafos generales, y por ende constituyen una de las técnicas principales de *graph drawing*. Esto se refleja en el hecho de que casi cualquier *software* de trazado de grafos contiene al menos una implementación de un algoritmo dirigido por fuerzas.

Los principales motivos de esta popularidad son los siguientes: son **intuitivos**, relativamente **sencillos de programar**, dan **buenos resultados** y son muy **flexibles**.

Son intuitivos

Porque se basan en analogías físicas de objetos conocidos (ej. resortes) cuyo comportamiento es fácil de comprender y predecir. Por ejemplo, cualquiera tiene al menos una idea de qué sucede si se estira un resorte y se lo suelta en el aire. Muchos algoritmos dirigidos por fuerzas son poco más que una simulación de estirar muchos resortes, y soltarlos en el aire. A lo que esto apunta es que uno puede, por lo menos para un grafo pequeño, predecir qué resultado dará el algoritmo con sólo conocer la esencia del mismo y el grafo al cual será aplicado. Esto no es lo que usualmente sucede con otros tipos de algoritmos, como por ejemplo los de dibujo ortogonal, donde no es intuitivo qué hará el algoritmo y qué forma tendrá el resultado.

Son relativamente sencillos de programar

Así como el comportamiento de los algoritmos dirigidos por fuerzas es generalmente fácil de entender, también son fáciles de programar. En comparación con la

mayoría de las otras familias de algoritmos de trazado de grafos los algoritmos dirigidos por fuerzas suelen ser más cortos y más sencillos de implementar, al menos en sus versiones más simples.

Un algoritmo dirigido por fuerzas típico tiene la siguiente estructura:

AlgoritmoFD (Grafo G)
1. Asignar una posición inicial a los vértices de G
2. Repetir hasta que el sistema basado en G esté en equilibrio:
    2.1. Seleccionar un vértice v en G
    2.2. Mover el vértice v tal que las fuerzas del sistema disminuyan

Si bien este esquema está muy simplificado, pueden observarse dos puntos importantes. Por un lado, el **modelo** elegido determinará la configuración del sistema "físico" que representa al grafo, y con ello cuáles son las fuerzas del sistema y cómo hacen interactuar a las partículas del mismo (que generalmente son los vértices). También estará implícito en el modelo cuáles son los criterios estéticos que debe satisfacer un buen trazado. Esto es así debido a que en estos algoritmos, lo que se busca siempre es un trazado cuyo sistema asociado esté en equilibrio, así que es una condición elemental para el modelo elegido que trazados con sistemas en equilibrio se correspondan con trazados estéticamente agradables.

Por otro lado, el **algoritmo** elegido determinará la posición inicial de los vértices, el criterio con el cual elegir qué vértice debe ser movido en cada iteración, y a dónde moverlo, siempre con la intención de minimizar las fuerzas del sistema.

Los muchos algoritmos dirigidos por fuerzas que existen se diferencian unos de otros en el modelo o el algoritmo (o ambos). Generalmente, variar (sólo) el modelo implicará variar los criterios estéticos que se intentan obtener, y por ende algoritmos con distintos modelos aspirarán a objetivos diferentes (podríamos decir que el conjunto de trazados óptimos, es distinto). Los cambios en los algoritmos implican distintas formas de llegar a un trazado en equilibrio, pero manteniendo el mismo espacio de búsqueda y los mismos óptimos. Obviamente esto no significa que ambos algoritmos lleguen al mismo resultado. Muchas veces un cambio relativamente menor, como la forma de elegir la posición inicial de los vértices, hace que dos algoritmos idénticos en todo lo demás lleguen a resultados muy distintos. Más adelante veremos algunos ejemplos.

Dan buenos resultados

Los resultados en grafos de tamaño mediano (hasta 50 vértices) suelen ser estéticamente muy buenos. En particular, son buenos logrando los siguientes objetivos estéticos: longitud de las aristas uniforme, vértices equiespaciados y simetría. Este último, como se mencionó previamente, es uno de los que visualmente resulta más importante y es difícil de obtener con los otros métodos de trazado de grafos. De hecho, esto ha hecho que algunos autores se refieran a los algoritmos dirigidos por fuerzas como "algoritmos simétricos".

Son muy flexibles

Una característica muy importante de los métodos dirigidos por fuerzas es su flexibilidad. Esto hace referencia al hecho de que con los ingredientes que conforman el modelo de los algoritmos dirigidos por fuerzas (fuerzas o energías) es posible modelar muchos criterios estéticos distintos, más allá de los tres que usualmente se satisfacen (longitud de aristas uniforme, vértices equiespaciados y simetría).

Esto es de gran importancia, ya que cuando un algoritmo de trazado de grafos es aplicado a un problema real, casi siempre aparecen criterios propios de la información que se quiere visualizar (criterios semánticos) que deben poder ser reflejados en los trazados resultantes. Los algoritmos dirigidos por fuerzas no ponen restricciones sobre el grafo al cual pueden ser aplicados y fácilmente pueden ser adaptados para reflejar estos nuevos criterios: muchas veces alcanza con agregar alguna otra fuerza o redefinir alguna ya existente para poder alterar significativamente los trazados obtenidos. A lo largo del presente trabajo, este último punto aparecerá una y otra vez a medida que se revisen algunas de las muchas adaptaciones que se pueden realizar a estos algoritmos.

Ejemplos de adaptaciones existentes son: trazados tridimensionales, grafos con agrupamiento (clustering) de vértices, grafos con restricciones en la posición de los vértices, grafos dirigidos y trazados con aristas curvas, entre muchos otros que serán cubiertos en los próximos capítulos.

Como es de esperar, no todo es perfecto en esta familia de algoritmos, por lo cual existen algunas desventajas que deben ser mencionadas. Sin embargo, creemos mejor discutirlas luego de presentar a los principales exponentes de estos métodos, por lo que pospondremos esa discusión hasta el próximo capítulo, sección 4.9.2.

Capítulo 4

Algoritmos clásicos

En este capítulo revisaremos los primeros algoritmos de trazado de grafos dirigido por fuerzas que surgieron. Los denominamos "clásicos" a pesar de que en promedio no tienen más de 15 años, porque constituyen los métodos dirigidos por fuerzas tradicionales, sobre los cuales se fueron creando en los últimos años nuevas técnicas para solucionar distintos problemas. La mayoría de los algoritmos que veremos en los próximos capítulos son modificaciones o extensiones de los aquí presentados.

Los métodos dirigidos por fuerzas existen desde antes de que se los comenzara a usar para la visualización de grafos. Uno de los primeros exponentes de este tipo de algoritmos es el algoritmo baricéntrico de Tutte ([Tut60],[Tut63]), que estaba orientado a la matemática más que a la visualización (ver sección 4.8). Muchos otros antecesores de los algoritmos dirigidos por fuerzas para trazado de grafos surgen en la misma época, creados para el diseño de plaquetas y circuitos VLSI ([FCW67],[QJB79],[WWM82]). Sin embargo, el primer trabajo que se podría decir que inauguró el uso de estos métodos para el trazado de grafos es el de Eades ([Ead84]). Debido a que fue el primero y uno de los más usados, comenzaremos por él.

4.1. *Spring embedder*

El primer algoritmo de trazado de grafos dirigido por fuerzas es el *spring embedder* de Eades ([Ead84]), y es el que presentó la analogía de los resortes que usualmente se usa para describir a estos métodos. Fue concebido con la intención de satisfacer dos criterios estéticos: longitud de aristas uniforme y mostrar toda la simetría que sea posible.

El modelo utilizado por Eades consiste en un sistema mecánico donde se reemplazan los vértices por anillos de acero y las aristas por resortes entre los anillos. Además de los resortes se agregan fuerzas repulsivas entre los anillos que no están conectados.

Conceptualmente el algoritmo consiste en colocar los anillos en alguna posición inicial y luego "liberar" el sistema, lo que hace que los anillos comiencen a moverse al ser tirados por los resortes y repelidos por las fuerzas repulsivas. Se espera hasta que el sistema llega a un estado de energía mínima, y cuando eso sucede se toman las posiciones finales de los anillos como posiciones de los vértices asociados. A continuación veremos los detalles del modelo y del algoritmo propiamente dicho.

Modelo

Aclaración sobre notación: de aquí en adelante utilizaremos la siguiente notación:

Dado $G = (V, E)$ un grafo no dirigido, con V el conjunto de vértices ($|V| = n$) y E el conjunto de aristas (u, v) $(u, v \in V, u \neq v)$, con $|E| = m$.

$p_v = (x_v, y_v)$ es la posición del vértice v (en principio supondremos que los vértices están en el plano).

$\overrightarrow{p_u p_v}$ es el vector unitario $\frac{p_u - p_v}{\|p_u - p_v\|_2}$

d_{uv} es la distancia entre las posiciones de los vértices u y v ($d_{uv} = \|\overrightarrow{p_u p_v}\|_2 = \|p_u - p_v\|_2$)

Se definen dos tipos de fuerzas: atractivas y repulsivas.

Fuerzas atractivas

Se definen para todo par de vértices u y v conectados por una arista:

$$f_a(u, v) = C_1 \log(\frac{d_{uv}}{C_2})\overrightarrow{p_v p_u} \tag{4.1}$$

donde C_1 es una constante que controla la fuerza del resorte y C_2 es otra constante que define la longitud ideal o natural de la arista. Estas fuerzas hacen que cuando la longitud sea mayor a C_2, los vértices de los extremos tiendan a acercarse, mientras que cuando la longitud sea menor a C_2, los mismos se repelan. El objetivo es que todas las aristas terminen con la longitud ideal, C_2.

Es importante notar que dado que la analogía física establece que estas fuerzas son originadas por resortes, sería de esperar que las mismas obedecieran a la ley de Hooke, es decir, algo del estilo $f_a(u, v) = K d_{uv}\overrightarrow{p_v p_u}$. Sin embargo, Eades argumenta que en la práctica estas fuerzas lineales son demasiado fuertes cuando los vértices están muy alejados, y decide reemplazarlas por fuerzas logarítmicas (de donde surge que estos resortes (imaginarios) a veces sean llamados *resortes logarítmicos*). Varios autores ([DETT99], [Tun99b]) ponen en duda que este cambio, que incurre en un costo adicional debido a la necesidad del cálculo del logaritmo, realmente produzca mejores resultados. Más aun, Tunkelang ([Tun99b]) dice que el uso de las fuerzas logarítmicas produce mucha variación en la longitud de las aristas, lo cual tiene un efecto perjudicial (en el sentido estético).

Fuerzas repulsivas

Se definen para todo par de vértices u y v distintos no conectados por una arista:

$$f_r(u, v) = \frac{C_3}{d_{uv}^2}\overrightarrow{p_u p_v} \tag{4.2}$$

donde C_3 es una constante de repulsión.

Estas fuerzas, que siguen una ley inversa al cuadrado de la distancia (*inverse square law)*, hacen que los vértices se repelan de manera inversa a su distancia, evitando que se condensen en cierta parte del dibujo, y por ende ayudando a que se distribuyan uniformemente en el plano.

En base a (4.1) y a (4.2), la fuerza $f(u)$ que experimenta un vértice u es

$$f(u) = \sum_{(u,v)\in E} f_a(u,v) + \sum_{v\in V/(u,v)\notin E, u\neq v} f_r(u,v) \tag{4.3}$$

Algoritmo

El algoritmo tiene como objetivo encontrar una posición de los vértices tal que el sistema se encuentre en equilibrio (es decir que la suma de todas las fuerzas sea cero). El algoritmo que propone Eades en [Ead84] es el siguiente:

SpringEmbedder(Grafo G)
1. Asignar una posición al azar a cada vértice de G.
2. Repetir M veces:
    2.1. Para cada v $\in V$
        2.1.1 Calcular f(v) (en base a 4.3)
    2.2. Para cada v $\in V$
        2.2.1 $p_v = p_v + C_4 f(v)$

donde C_4 y M – la cantidad de iteraciones – son constantes. En su artículo original, Eades propone usar los siguientes valores para las constantes: C_1=2,0, C_2=1,0, C_3=1,0, C_4=0,1 y M=100. Respecto a la última, tener en cuenta que estos números fueron probados, según el mismo autor, para grafos de hasta 50 vértices. Grafos más grandes requerirán aumentar la cantidad de iteraciones. En general, pruebas experimentales estiman que $M \approx n$ es apropiado [Beh99].

El algoritmo es sumamente sencillo, a pesar de lo cual los resultados que produce suelen ser muy buenos. Una característica a observar es que en cada iteración principal todos los vértices se mueven al mismo tiempo (en el sentido de que las nuevas posiciones se calculan en base a las fuerzas de la iteración anterior, sin que influyan los movimientos ya realizados en la iteración actual). Como veremos más adelante, no todos los algoritmos dirigidos por fuerzas hacen esto.

Es también destacable lo simple e intuitivo del proceso de optimización (sobre el cual volveremos el próximo capítulo). El mismo se basa en la idea de que para reducir la tensión o el *stress* en un vértice (o sea la magnitud de $f(v)$), uno puede moverlo hacia la dirección en la que le están "tirando" (que corresponde al vector de fuerzas $f(v)$), de manera de reducir la tensión para ese vértice. Este proceso se realiza en "simultáneo" con cada vértice y es repetido M veces. Como veremos en el capítulo 5, esta forma de minimizar la energía es simplemente aplicar el método del gradiente.

Algunas observaciones

Si bien la intención del algoritmo es atacar los criterios estéticos de longitud de aristas uniforme y simetría, sólo el primero es atacado explícitamente a través de las fuerzas atractivas. Además, el uso de las fuerzas repulsivas hace que se cumpla un segundo criterio estético, que es que los vértices estén distribuidos uniformemente en el plano. A pesar de que la simetría no es atacada explícitamente, los trazados resultantes sí presentan simetría, y esto es un denominador común de todos los algoritmos dirigidos por fuerzas. La relación entre estos y la simetría ha sido explorada con más profundidad en [EL00].

Como ya se mencionó, los resultados obtenidos con el *spring embedder* suelen ser muy buenos. En [Ead84] se afirma que en general se satisfacen, en la medida de lo posible, los dos criterios buscados. También se comenta que existen algunos grafos para los cuales los resultados no son buenos, por ejemplo los grafos que contienen subgrafos densos. Esto también es algo común para los algoritmos dirigidos por fuerzas.

Veamos cuál es la complejidad (en tiempo) de este algoritmo. Cada cálculo de $f(v)$ tiene costo $O(n)$. A su vez, la función se calcula $O(Mn)$ veces, dando una complejidad de $O(Mn^2)$. Empíricamente se ha comprobado que $M \approx n$ es suficiente [Beh99], así que bajo esta suposición la complejidad del *spring embedder* es $O(n^3)$. Esto ya deja ver una de las desventajas del método: su alto costo computacional, principalmente debido a que evaluar las fuerzas del sistema tiene costo $O(n^2)$.

Otras desventajas que se le suelen atribuir son la falta de fuerzas repulsivas entre las aristas que están conectadas, que puede ocasionar que varios nodos se concentren en una área pequeña del trazado, la falta de un verdadero criterio de corte para el ciclo principal y el hecho de que la magnitud del desplazamiento de cada vértice está fijo a lo largo de todo el proceso (constante C_4) [Beh99].

Varios de los algoritmos que veremos a continuación son, o pueden ser vistos como, variantes del *spring embedder* de Eades que intentan solucionar alguno de sus puntos débiles o mejorar alguna característica.

4.2. FR

Fruchterman y Reingold presentan en [FR91] un método basado en el *spring embedder* de Eades que usualmente es conocido como el algoritmo FR. El mismo fue creado con dos criterios estéticos como objetivo: que los vértices no estén muy cerca unos de otros y que los vértices que estén conectados por aristas se dibujen cerca. Al igual que el de Eades, su algoritmo está inspirado en una simulación de un sistema físico en el que la realidad física es sacrificada en pos de algoritmos más sencillos y que se apliquen mejor al problema en cuestión, el trazado de grafos.

El método en sí mismo preserva la esencia del *spring embedder*, ya que el modelo sigue siendo anillos que se repelen conectados por resortes, pero cambian las fuerzas atractivas y repulsivas. Además modifican el algoritmo de optimización haciendo que el desplazamiento de cada vértice dependa de un "esquema de enfriamiento",

de manera similar al que se usa en *Simulated Annealing* (y del que hablaremos más en la sección del algoritmo DH).

Modelo

Al igual que en el anterior, existen dos tipos de fuerzas: atractivas y repulsivas.

Fuerzas atractivas
Se definen para todo par de vértices u y v conectados por una arista:

$$f_a(u,v) = \frac{d_{uv}^2}{K}\overrightarrow{p_v p_u} \tag{4.4}$$

donde K es una constante que representa la distancia ideal entre los vértices del grafo para que estén distribuidos uniformemente, y que se define como $K = C\sqrt{\frac{a}{n}}$, donde C es una constante experimental, $n = |V|$ y a es el área disponible para dibujar el grafo. Fruchterman y Reingold observaron que las fuerzas lineales muchas veces no son suficientemente fuertes como para superar los mínimos locales, por lo que propusieron usar cuadráticas en su lugar. Una observación interesante que se hace tanto en [FR91] como en [Tun99b] es que el uso de potencias más grandes que 2 para estas fuerzas no produce mejoras importantes en los resultados, con la contra de que son más costosas de evaluar.

Fuerzas repulsivas
Se definen para todo par de vértices u y v ($u \neq v$):

$$f_r(u,v) = \frac{K^2}{d_{uv}}\overrightarrow{p_u p_v} \tag{4.5}$$

En [FR91] los autores comentan que eligieron estas fuerzas porque se asemejan a las de Hooke y se ajustan a los objetivos buscados, y porque tenían como ventaja sobre otras (como la de Hooke, $f_a(u,v) = K d_{uv}\overrightarrow{p_v p_u}$) el hecho de permitir superar con más facilidad los mínimos locales. Respecto a las fuerzas que usa Eades en su *spring embedder*, dicen que con estas fuerzas obtienen resultados similares, pero con menos costo de cómputo, ya que no requieren usar aritmética de punto flotante.

En base a (4.4) y a (4.5), la fuerza $f(u)$ que experimenta un vértice u es

$$f(u) = \sum_{(u,v)\in E} f_a(u,v) + \sum_{v\in V, u\neq v} f_r(u,v) \tag{4.6}$$

Algoritmo

El algoritmo que usan para minimizar la energía del sistema es similar al de Eades en que todos los vértices se mueven en simultáneo y en la forma de elegir la dirección en la cual moverlo. Sin embargo, FR elige hasta cuánto desplazar cada vértice en base a una temperatura t, y que disminuye a medida que transcurre el tiempo, por ende, el algoritmo se inicia permitiendo movimientos más bien grandes que se van limitando con el correr del tiempo.

FR(Grafo G)
1. Asignar temperatura inicial $t=t_0$
2. Asignar una posición al azar a cada vértice de G
3. Repetir M veces:
    3.1. Para cada $v \in V$
        3.1.1 Calcular $f(v)$ (en base a 4.6)
    3.2. Para cada $v \in V$
        3.2.1 $p_v = p_v + \frac{f(v)}{\|f(v)\|} \min(\|f(v)\|, t)$
        3.2.2 Controlar que p_v no quede fuera del área de dibujo
    3.3 $t = cool(t)$

El algoritmo presentado en [FR91] no especifica qué esquema de enfriamiento usar, es decir, qué temperatura inicial (t_0) y cómo va decreciendo en función del tiempo ($cool()$). Recalcan que los resultados varían mucho de acuerdo a esto, y sostienen que en sus pruebas obtuvieron los mejores resultados con un esquema dividido en dos fases: durante la primera fase la temperatura es inicialmente alta y decrece rápidamente, y en la segunda fase la temperatura es baja y constante. De todas maneras conjeturan que mejores formas de enfriamiento que las que usaron ellos acelerarían mucho su algoritmo.

Nuevamente, la cantidad de iteraciones, M, es una constante fijada por el usuario.

Notar que en este algoritmo aparece el factor del espacio disponible para dibujar el grafo, que en el de Eades no aparecía. De hecho esto es usado para calcular el valor de K y en el paso 3.2.2 para verificar que ningún vértice es ubicado fuera del área del dibujo. Como se verá a lo largo de este trabajo, si bien siempre (o casi siempre) existe esta limitación física del área de dibujo, en general no se la tiene en cuenta y al finalizar el algoritmo se escala el trazado resultante de manera de que entre en el espacio disponible (esto es también lo que hacen muchas de las implementaciones del algoritmo FR, por ejemplo la del programa *Graphlet* [Beh99]).

Algunas observaciones

El algoritmo FR extiende el de Eades en dos aspectos, por un lado usa fuerzas ligeramente distintas que contienen los mismos criterios estéticos pero que son menos costosas de evaluar. Por otro lado, incluyen una mejora en la forma de minimizar la energía del sistema que se basa en una temperatura global (ya que es

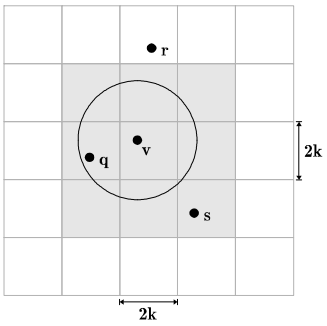


FIGURA 4.1. Cálculo de fuerzas repulsivas en la variante malla de FR.

la misma para todos los vértices) junto a un esquema de enfriamiento, que permite acelerar la convergencia del método. Esta mayor velocidad es la principal ventaja del algoritmo, ya que si bien sigue siendo sencillo, no lo es tanto como el de Eades, y los resultados son semejantes en calidad [FR91].

La complejidad en tiempo del algoritmo es la misma que el de Eades, ya que la estructura de algoritmo sigue siendo la misma, por lo tanto si tomamos $M \approx n$, la misma sería $O(n^3)$.

Para mejorar aun más la velocidad de su algoritmo, Fruchterman y Reingold proponen en su artículo una variante del algoritmo FR, que denominan **variante malla** (*grid variant* en el original).

La variante malla está inspirada en las técnicas usadas en simulación de n-cuerpos (ver capítulo 11) para aproximar las fuerzas repulsivas entre los vértices, ya que éstas son responsables de una parte importante del costo del algoritmo (hace que evaluar todas las fuerzas del sistema sea $O(n^2)$ en lugar de $O(m)$).

La idea es que en lugar de calcular la repulsión entre todo par de vértices, se calcula sólo entre los vértices más cercanos, ya que es de esperar que dada la forma de las fuerzas repulsivas (decrecen con el cuadrado de la distancia), los vértices lejanos no contribuyan mucho a la suma de fuerzas.

Para esto, se divide el área del dibujo en una cuadrícula, y para cada vértice sólo se calculan las fuerzas repulsivas entre v y los vértices de su propio cuadrado y los de sus ocho cuadrados vecinos. Sin embargo, los autores notaron que en la práctica la subdivisión en cuadrados provocaba distorsión en el trazado resultante. Para solucionar esto decidieron calcular las fuerzas repulsivas sólo con los vértices de cuadrados vecinos que estén a distancia menor a un umbral $2k$ (donde $2k$ es el ancho de los cuadrados). En la Figura 4.1, el vértice v es repelido de q, pero no de r ni de s, ya que están demasiado lejos como para ser tenidos en cuenta.

Estas modificaciones hacen que cuando la distribución de los vértices es aproximadamente uniforme, el cálculo de las fuerzas repulsivas baje a $O(n)$, en cuyo caso la complejidad total de la variante malla de FR sería $O(n^2)$. Es importante mencionar que el hecho de ubicar cada vértice en su correspondiente cuadrado en la cuadrícula resulta en un *overhead* que no es despreciable, por lo cual en [FR91] sugieren usarlo sólo para grafos grandes. Esto último, más el hecho de que

el algoritmo se hace más complicado y que la ganancia de velocidad está sujeta a la distribución de los vértices, ha hecho que esta variante no sea muy usada en la práctica [Beh99].

Tunkelang [Tun99b] critica al algoritmo FR argumentando que el proceso de optimización usado es innecesariamente complicado, y que el esquema de enfriamiento es un sustituto pobre de una búsqueda lineal (ver capítulo 5).

Finalmente, se observa que el algoritmo FR también tiene el problema de que el número de iteraciones del ciclo principal es arbitrario.

4.3. GEM

En [FLM95], Frick et al. proponen otro algoritmo que sigue la línea de los dos anteriores, que denominan GEM (por *graph embedder*). GEM está basado en el *spring embedder* de Eades con el agregado de varias ideas novedosas. En primer lugar, se usan temperaturas para acelerar la convergencia, pero a diferencia de las que se usan en FR, éstas son individuales de cada vértice. En segundo lugar agregan una fuerza que atrae los vértices hacia el baricentro del dibujo, con el objetivo de acelerar la convergencia y evitar que parte del grafo se aleje mucho del resto. Finalmente, también para acelerar la convergencia, agregan detección de oscilaciones y rotaciones.

Modelo

Se definen tres tipos de fuerzas: atractivas, repulsivas y gravitacionales.

Fuerzas atractivas
Se definen para todo par de vértices u y v conectados por una arista:

$$f_a(u,v) = \frac{d_{uv}^2}{l^2 \Phi(v)} \overrightarrow{p_v p_u} \tag{4.7}$$

donde l es la longitud ideal de la arista, que en principio sería una constante (igual a 128 en la implementación original), y Φ es una función que crece con el grado del vértice, $\Phi(v) = 1 + \frac{\deg(v)}{2}$ en la implementación original.

Fuerzas repulsivas
Se definen para todo par de vértices u y v ($u \neq v$):

$$f_r(u,v) = \frac{l^2}{d_{uv}^2} \overrightarrow{p_u p_v} \tag{4.8}$$

Fuerzas gravitacionales
Se definen para todo vértice v:

$$f_g(v) = \Phi(v)\gamma\,(\zeta - p_v) \tag{4.9}$$

donde $\zeta = \frac{1}{n}\sum_{w \in V} p_w$ es el baricentro del trazado y γ es una constante (constante gravitacional). Como se puede ver, esta fuerza atrae todo vértice hacia el baricentro del trazado. Su objetivo es doble. Por un lado, evita que partes disconexas del grafo se queden "a la deriva" y se alejen del resto. Por otro lado, acelera la convergencia del método hasta un 30 %, según las pruebas realizadas por los autores del algoritmo.

En base a (4.7), (4.8) y a (4.9), la fuerza $f(u)$ que experimenta un vértice u es

$$f(u) = \sum_{(u,v)\in E} f_a(u,v) + \sum_{v \in V, v \neq u} f_r(u,v) + f_g(u) \tag{4.10}$$

Algoritmo

El algoritmo en sí es parecido al FR en la forma de elegir la dirección en la que se mueve cada vértice y en que limita los movimientos a una temperatura, pero en este caso la temperatura es independiente de cada vértice (t_v es la temperatura del vértice v), y además la forma de calcular la nueva temperatura es bastante más compleja, y tiene en cuenta además del tiempo si el vértice está oscilando o rotando.

GEM(Grafo G)
1. Asignar temperatura inicial a cada vértice, round=0
2. Asignar una posición al azar a cada vértice de G
3. Mientras ($T_{global} > T_{min}$) y (round<M):
    3.1. Encontrar una permutación al azar de los vértices de G
    3.2. Para cada v en la permutación al azar
        3.2.1. Calcular f(v) (en base a 4.10)
        3.2.2. Perturbar al azar f(v)
        3.2.3. $p_v = p_v + \frac{f(v)}{\|f(v)\|} t_v$
        3.2.4. Actualizar temperatura t_v y otros datos accesorios de v
    3.3. round=round+1

La temperatura global T_{global} se define como el promedio de las temperaturas de todos los vértices (t_v), y T_{min} es un umbral ingresado por el usuario.

Notar que en el paso 3.2.2, una pequeña perturbación al azar es introducida en la fuerza de v (o impulso de v, usando el mismo término que los autores del algoritmo) para hacer al algoritmo más robusto frente a mínimos locales pobres.

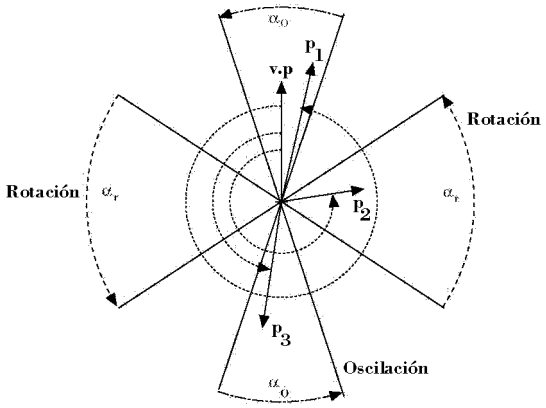


FIGURA 4.2. Detección de rotaciones y oscilaciones en GEM [FLM95].

Notar también que a diferencia de los dos algoritmos anteriores, en GEM los vértices se mueven uno a uno, es decir que el movimiento de un vértice afecta inmediatamente al de todos los que se muevan luego de éste. En Eades y FR, en cambio, todos los vértices eran movidos en simultáneo.

La otra parte importante del algoritmo es la correspondiente al paso 3.2.4, donde se calcula en cada iteración la temperatura del vértice, junto a otros datos accesorios.

En el cálculo de la temperatura de cada vértice t_v intervienen dos factores: la temperatura anterior de v y los dos últimos movimientos que sufrió el vértice; llamaremos $f_r(v)$ al último y $f_{r-1}(v)$ al anteúltimo.

La forma de calcular t_v es la siguiente. Se compara el vector fuerza de los dos últimos movimientos, $f_r(v)$ y $f_{r-1}(v)$. Sea $\alpha = \angle(f_r(v), f_{r-1}(v))$ el ángulo entre las dos últimas direcciones en las que el vértice v fue movido. Si v fue movido en aproximadamente la misma dirección que la vez anterior ($\cos\alpha \approx 1$), t_v aumenta, mientras que si la dirección es opuesta a la anterior ($\cos\alpha \approx -1$), se teme que v esté oscilando y t_v se hace menor. De manera similar se consideran las rotaciones. Si los dos últimos movimientos fueron aproximadamente perpendiculares ($\cos\alpha \approx 0$), se teme una rotación y t_v es disminuida, si además en los últimos movimientos el vértice se viene moviendo en dirección similar (para decidir esto se definen umbrales apropiados), se teme una rotación alrededor de un punto fijo y también se disminuye t_v. En la Figura 4.2 puede verse un esquema de la detección de oscilaciones y rotaciones.

Algunas observaciones

El algoritmo GEM intenta atacar los mismos criterios estéticos que los anteriores, así que no sorprende mucho el hecho de que los resultados sean muy similares a los de Eades o FR [FLM95],[BHR95]. Según los autores, los trazados obtenidos

son ligeramente mejores que los de FR, aunque según [Beh99] los trazados de GEM suelen tener más cruces que los otros.

Al igual que con FR, el principal aporte de este algoritmo es un aumento en la velocidad de convergencia. Gracias a un esquema de enfriamiento elaborado, que detecta oscilaciones y rotaciones, y que mantiene una temperatura independiente para cada vértice, GEM logra con éxito disminuir la cantidad de iteraciones necesarias para converger sobre las que requiere el *spring embedder* [BHR95],[KW01], y en general es considerado uno de los algoritmos más rápidos de los que se cubren en este capítulo [BHR95] (decimos en general, ya que las comparaciones de velocidad entre estos algoritmos dependen mucho de las implementaciones que se usan. Por ej., en [BHR95] se dice que GEM resultó más veloz que FR, mientras que las pruebas presentadas en [Beh99] indican lo contrario). También hay que destacar que, al igual que con FR, todas las operaciones de GEM pueden realizarse evitando usar aritmética de punto flotante.

El costo en tiempo de GEM sigue en el mismo orden que el de Eades y FR (en su variante estándar), ya que si bien la cantidad de iteraciones depende de la temperatura global T_{global}, los mismos autores en su artículo original aclaran que estimaron que el número anda cerca de las n iteraciones, dando un costo total en tiempo de $O(n^3)$. Al respecto del criterio de corte, si bien en [FLM95] dicen que en la mayoría de los casos se lograr alcanzar la temperatura de corte, [Beh99] comenta que en sus pruebas casi siempre se terminó al exceder la cantidad máxima de iteraciones M, no pudiendo alcanzar el umbral T_{min}.

Finalmente, observamos que este algoritmo hace uso del azar más que los otros, perturbando aleatoriamente el vector de fuerzas de cada vértice y eligiendo al azar el orden en el cual mover los vértices. Esto se hace bajo la hipótesis de Frick et al. de que el azar puede ser beneficioso para estos métodos de trazado de grafos. Los mismos autores dicen que en su algoritmo, el hecho de elegir al azar el orden en el cual mover los vértices mejoró los resultados notablemente. También comentan que el trazado inicial no parece influir mucho en el trazado final. Esto es algo que se ha observado en otros algoritmos, por ej. en KK [KK89], aunque no implica que un mejor trazado inicial no pueda acelerar la convergencia y mejorar el resultado (el tema de los trazados iniciales se analiza con más detalle en el capítulo 10).

4.4. Modelando con las distancias teóricas: KK

Los tres algoritmos hasta aquí vistos tienen muchas cosas en común, tanto en el modelo de fuerzas usado como en el algoritmo que usan para encontrar el equilibrio del sistema. Un enfoque bastante diferente para abordar el problema es el tomado por Kamada y Kawai, cuyo algoritmo presentado en [KK89] es referido simplemente como el algoritmo KK.

Kamada y Kawai proponen un modelo en el que hay un único tipo de fuerzas, que pueden ser vistas como resortes, que conectan entre sí a todos los vértices. Los resortes obedecen a la ley de Hooke (fuerzas lineales) y cada resorte tiene

una longitud ideal y una rigidez distinta. Uno de los aportes más importantes y originales de KK es que la distancia ideal de cada resorte está en proporción a la distancia teórica[1] entre los vértices que conecta.

El segundo punto importante donde KK se diferencia en los algoritmos hasta aquí vistos es en que en lugar de trabajar con las fuerzas, y buscar un equilibrio de las mismas, ellos trabajan directamente con la energía del sistema, y buscan un estado de energía mínima (es decir, una posición para los vértices del grafo en la cual la energía sea (localmente) mínima). Además, la manera de llegar a este mínimo local de la energía es mucho más "numérica" que en los algoritmos anteriores, ya que buscan directamente mediante una técnica numérica un trazado donde las derivadas parciales se anulen.

4.4.1. Modelo

Hay un único tipo de fuerzas definido, que son resortes **entre todo par de vértices** u, v. Los mismos tienen una rigidez y longitud ideal determinada y producen el ya comentado efecto de atraer a los vértices cuando el resorte está estirado, y repelerlos cuando está comprimido. La fuerza que usan para modelar el resorte entre u y v ($u \neq v$) es la siguiente:

$$f(u, v) = S_{uv}(d_{uv} - \delta_{uv})$$

donde d_{uv} es la distancia Euclídea entre u y v ($d_{uv} = \|p_u - p_v\|_2$), δ_{uv} es la distancia teórica entre u y v y S_{uv} es un parámetro de rigidez del resorte, que es más fuerte para vértices cuya distancia teórica es menor y decrece a medida que su distancia teórica aumenta, definido como $S_{uv} = \frac{S}{\delta_{uv}^2}$, para S constante.

Como ya se mencionó, el algoritmo no trabaja con las fuerzas directamente, sino con la energía potencial del sistema (que es la integral de estas fuerzas), cuya expresión es:

$$\begin{aligned} E(u, v) &= \int f(u, v) d(d_{uv} - \delta_{uv}) \\ &= \frac{1}{2} S_{uv}(d_{uv} - \delta_{uv})^2 \end{aligned}$$

Así que la energía total del sistema, o equivalentemente, el nivel de "desequilibrio" del sistema, que es la suma de la energía de cada uno de los $\binom{n}{2} = \frac{n(n-1)}{2}$ resortes, queda expresada como:

$$E = \sum_{u,v \in V, u \neq v} E(u, v) = \sum_{u,v \in V, u \neq v} \frac{1}{2} S_{uv}(d_{uv} - \delta_{uv})^2 \tag{4.11}$$

[1] Dado un grafo $G = (V, E)$, la distancia teórica entre los nodos u y v ($u, v \in V$) es la longitud del camino mínimo entre u y v en G.

Algoritmo

El objetivo del algoritmo es encontrar un mínimo de la función de energía E. Recordemos que E es una función de las posiciones de los vértices $p_v = (x_v, y_v)$, es decir que queremos minimizar una función de $2n$ variables:

$$E = \sum_{u,v \in V, u \neq v} \frac{1}{2} S_{uv} (\|p_u - p_v\|_2 - \delta_{uv})^2 \tag{4.12}$$

$$= \sum_{u,v \in V, u \neq v} \frac{1}{2} S_{uv} (\sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} - \delta_{uv})^2 \tag{4.13}$$

Condiciones necesarias para que las posiciones de los vértices estén en un mínimo local[2] de la función E son

$$\frac{\partial E}{\partial x_m} = 0 \text{ y } \frac{\partial E}{\partial y_m} = 0 \text{ para todo } m \in V \tag{4.14}$$

El objetivo del algoritmo KK es encontrar una posición para los vértices donde las derivadas parciales sean nulas, o lo mismo, buscar una raíz de las derivadas parciales:

$$\frac{\partial E}{\partial x_m} = \sum_{v \in V, v \neq m} S_{uv} \left\{ (x_m - x_v) - \frac{\delta_{uv}(x_m - x_v)}{\sqrt{(x_m - x_v)^2 + (y_m - y_v)^2}} \right\}$$

$$\frac{\partial E}{\partial y_m} = \sum_{v \in V, v \neq m} S_{uv} \left\{ (y_m - y_v) - \frac{\delta_{uv}(y_m - y_v)}{\sqrt{(y_m - x_v)^2 + (y_m - y_v)^2}} \right\}$$

Para esto utilizan una variante de la técnica Newton-Raphson. Primero notan que si bien hay $2n$ ecuaciones en (4.14), no pueden usar directamente Newton-Raphson para $2n$ variables porque las variables no son independientes entre sí. Entonces deciden adoptar la siguiente estrategia: toman un vértice v_m por vez y suponen los otros fijos, y buscan una raíz de las derivadas (4.14) considerando todas las otras posiciones constantes. De esta manera, el problema pasa a ser de 2 variables (x_m, y_m), sobre el cual usan Newton-Raphson para 2 variables.

Recordemos que el método de Newton-Raphson para 2 variables ([BF98]) consiste en una iteración de la forma

$$x_{n+1} = x_n - J(x)^{-1} F(x) \tag{4.15}$$

donde x_n y $x_{n+1} \in \mathbb{R}^2$, $F : \mathbb{R}^2 \to \mathbb{R}^2$ es la función cuya raíz se quiere encontrar, y $J(x) \in \mathbb{R}^{2\times 2}$ es la matriz Jacobiano de F. En nuestro caso, tenemos

[2]En realidad ésta también es condición para un máximo, pero en nuestro contexto E no tiene ningún máximo, ya que es intuitivo que siempre puedo mover algún vértice "hacia afuera" y aumentar el valor de E.

$$F(x) = \begin{pmatrix} \frac{\partial E}{\partial x_m} \\ \frac{\partial E}{\partial y_m} \end{pmatrix} \text{ y } J(x) = \begin{pmatrix} \frac{\partial^2 E}{\partial x_m^2} & \frac{\partial^2 E}{\partial y_m \partial x_m} \\ \frac{\partial^2 E}{\partial x_m \partial y_m} & \frac{\partial^2 E}{\partial y_m^2} \end{pmatrix}$$

En la práctica, para calcular el desplazamiento $(\delta x, \delta y)$ del vértice en cada paso de la iteración 4.15 (donde $(\delta x, \delta y)^t = -J(x)^{-1}F(x)$) puede evitarse el cálculo de la inversa del Jacobiano resolviendo un sistema de ecuaciones de 2×2, que es lo que se hace en el algoritmo original. Los detalles, que aquí omitimos ya que no hacen a la esencia del algoritmo, pueden encontrarse en [KK89].

En síntesis, en cada iteración, el algoritmo KK va a tomar un vértice v_m y va a buscar mediante Newton-Raphson una posición que será un mínimo de la función de energía E, donde las posiciones de todos los otros vértices han sido congeladas. Luego se repetirá este mismo proceso cambiando el vértice v_m. El pseudo-código del algoritmo es el siguiente:

KK(Grafo G)
1. Calcular distancias teóricas
2. Asignar una posición inicial a cada vértice de G
3. Mientras ($\max_i \Delta_i > \epsilon$):
    3.1. Tomar v_m tal que $\Delta_m = \max_i \Delta_i$
    3.2. Mientras ($\Delta_m > \epsilon$)
        3.2.1. Calcular una nueva posición de v_m en base a 4.15 y recalcular Δ_m

donde $\Delta_m = \sqrt{(\frac{\partial E}{\partial x_m})^2 + (\frac{\partial E}{\partial y_m})^2}$ es la norma del gradiente de la energía del vértice v_m, o visto de otra forma, la magnitud de las fuerzas que actúan sobre v_m.

Notar que el vértice que se elige para ser movido en cada iteración del ciclo principal es el que está en mayor "desequilibrio", o el que está siendo afectado por la mayor fuerza.

Notar también que el ciclo 3.2 es exactamente la aplicación del método iterativo de Newton-Raphson, usando como criterio de corte un umbral en las derivadas (que es la función cuya raíz se quiere encontrar, así que es de esperar que Δ_m tienda a cero).

Algunas observaciones

El algoritmo KK se centra en un único criterio estético, distinto a los de los algoritmos anteriores: que las distancias en el trazado final sean iguales a las distancias teóricas. Es interesante ver que este criterio implica de cierta manera los de longitud de aristas uniforme (esto es inmediato) y de vértices distribuidos uniformemente. Este último se puede entender pensando en que se evitan concentraciones de vértices, ya que si los mismos no están conectados y su distancia teórica es grande, se verán forzados a alejarse, y en caso de que estén conectados entonces está bien que estén cerca unos de otros, debido al criterio de las aristas de longitud uniforme. Esto explica que los resultados obtenidos con KK sean muy

buenos y muy similares a los obtenidos con otros algoritmos que se ocupan de los otros criterios estéticos como FR y GEM [BHR95].

Al igual que en GEM, y a diferencia del *spring embedder* y FR, en KK los vértices se mueve de a uno por vez. Por otro lado, a diferencia de los tres algoritmos anteriores, aquí no se usa el método del gradiente para elegir la dirección en la cual mover el vértice, sino que se va moviendo el vértice de acuerdo al método de Newton-Raphson. Al respecto, en [SSL00] se argumenta que un defecto de KK es que como Newton-Raphson no siempre converge, puede ocurrir que para algún grafo el método no converja, en cuyo caso no se llegaría a un punto de energía mínimo. Sugieren para enmendar este problema usar una temperatura que vaya limitando los movimientos de los nodos hasta congelarlos, en caso de que Newton diverja. Sin embargo, no dan ningún ejemplo de un grafo en el que ello ocurra y tampoco explican si hay alguna evidencia de que esto pueda llegar a ocurrir (ya que es sabido que Newton-Raphson tiene casos en los cuales no converge, y casos en los cuales converge para cualquier valor inicial). Ningún otro trabajo de los analizados comenta acerca de este supuesto problema.

Una ventaja de KK que merece ser destacada es que el algoritmo no requiere ninguna modificación para ser aplicado a grafos con pesos. Esto es así debido a que los pesos de las aristas se ven reflejados automáticamente en los cálculos de las distancia teóricas, y por ende ninguna alteración al algoritmo original es necesaria.

La complejidad temporal del algoritmo está dominada por el cálculo de las distancias teóricas (paso 1), que en la implementación de los autores es $O(n^3)$ (ya que usan el algoritmo de Floyd, pero como ellos mismos aclaran, podría mejorarse a $O(nm \log n)$ si se usara en su lugar el algoritmo de camino mínimo de Dijkstra [CLRS01], lo cual sería una mejora para grafos esparsos, es decir, con $m \in O(n)$). Por otro lado se encuentran las iteraciones de los dos ciclos anidados. El cálculo de máx Δ_i requiere $O(n)$ pasos, mientras que el paso 3.2.1 puede hacerse en $O(1)$, por lo tanto si el ciclo interior se ejecuta T veces, cada iteración del exterior tendría costo $O(T + n)$. El valor de T y la cantidad de veces que se ejecuta el ciclo exterior no se conocen de manera teórica. En algunas implementaciones se usan $O(n)$ iteraciones en ambos ciclos [Beh99], en cuyo caso la complejidad sería $O(n^3)$ (ya que está dominada por el cálculo de las distancias).

Varios autores [Beh99],[Tun99b] coinciden en que el principal punto débil del algoritmo de Kamada y Kawai es su alto costo computacional, sin embargo, la comparación realizada en [BHR95] dio como resultado que KK, junto a GEM, son los algoritmos más rápidos de entre los comparados (más detalles sobre este estudio en la sección 4.9). Una observación interesante que se hace en [Tun99b], es que al requerirse resortes entre todo par de vértices, no se pueden usar técnicas de simulación de n-cuerpos para aproximar las fuerzas, cosa que sí se puede hacer con la mayoría de los métodos que tienen por separado fuerzas atractivas y repulsivas.

4.5. Funciones generales de energía: DH

Otro enfoque basado en energía – pero muy distinto al de KK – es el que proponen Davidson y Harel [DH96], cuyo algoritmo, usualmente conocido como DH, modela el problema con una función general de energía compuesta por varios términos independientes que cubren distintos criterios estéticos deseables en el trazado de un grafo. DH usa una función de energía general que considera varios criterios estéticos al mismo tiempo, y que permite mucha flexibilidad a la hora de elegirlos. De hecho, DH es uno de los pocos algoritmos que intentan evitar de manera explícita el cruce de aristas.

El algoritmo DH se basa en un conocido método de optimización llamado *Simulated Annealing* [KGJV83]. El método usa también una analogía física, tomada de la termodinámica. Se basa en la forma en la que los metales líquidos se enfrían. Cuando las moléculas del metal líquido están sujetas a altas temperaturas se mueven libremente entre sí. A medida que el metal se enfría la movilidad va decreciendo. Si este enfriamiento se realiza suficientemente despacio, las moléculas pueden ordenarse y se termina formando un cristal puro con energía mínima. Pero si el enfriamiento se realiza muy rápido, termina en estado sin forma con mayor energía.

En el contexto de minimizar una función, este comportamiento de los metales líquidos se aplica simulando este proceso de enfriamiento a través de movimientos discretos en el espacio de soluciones del problema. A partir de un estado inicial, y su correspondiente energía, se van haciendo movimientos aleatorios según una distribución basada en la de Boltzmann ($p(E) = e^{\frac{-E}{kT}}$), que hacen que la energía del sistema vaya cambiando. Las movidas que disminuyen la energía son siempre aceptadas, mientras que las que la aumentan (llamadas *uphill moves*, movidas cuesta arriba) son aceptadas aleatoriamente según una distribución que considera la diferencia de energía y la temperatura actual. La idea es que a temperaturas altas, es más probable aceptar movidas que aumentan la energía, pero a medida que la temperatura disminuye, es cada vez menos probable aceptarlas.

En el problema de trazado de grafos, el espacio de soluciones está formado por los posibles trazados del grafo. Los movimientos posibles (la forma de pasar de una solución a otra) son los cambios en la posición de alguno de los vértices del grafo. El ingrediente más importante bajo este enfoque es la elección de la función de energía que se quiere minimizar, que va a constituir el modelo de qué representa un grafo estéticamente bueno, seguido de la elección del esquema de enfriamiento, que determina cómo va cambiando la temperatura a medida que transcurre el tiempo.

Modelo

Davidson y Harel proponen considerar cinco criterios estéticos, y para cada uno definen un potencial que lo representa en la función de energía. Los mismos se detallan a continuación.

Distribución de nodos

Los nodos deben estar distribuidos uniformemente en el espacio disponible para el trazado. Para esto definen un potencial para evitar que los nodos estén muy cerca unos de otros (lo cual es análogo a tener a una fuerza repulsiva entre todo par de nodos). El potencial de repulsión se define para todo par de nodos (u, v) $(u \neq v)$ como:

$$U_r(u, v) = \frac{1}{d_{uv}^2} \tag{4.16}$$

Los autores comentan haber probado con varios potenciales distintos, y haber obtenido los mejores resultados con éste, que es el mismo que se usa en el *spring embedder.*

Espacio de dibujo

Los nodos no pueden salirse del espacio disponible para el trazado, que se asume es rectangular. Existen varias formas de incorporar esta restricción a la función objetivo. Una opción es incorporar vértices ficticios "clavados" sobre los bordes que repelan a todos los otros vértices. Sin embargo el número de vértices ficticios crece con la cantidad de vértices del grafo, así que esta idea fue desechada por los autores del algoritmo. Con lo que obtuvieron mejores resultados es con el agregado de un nuevo potencial para repeler a los nodos de los bordes. El mismo considera las distancias entre cada vértice y los cuatro bordes, y se define como

$$U_b(u) = \frac{1}{r_u^2} + \frac{1}{l_u^2} + \frac{1}{i_u^2} + \frac{1}{s_u^2} \tag{4.17}$$

donde r_u, l_u, i_u y s_u, son las distancias entre la posición del vértice u y el borde derecho, izquierdo, inferior y superior, respectivamente. El uso de este potencial puede evitarse si el trazado final es escalado para que entre en el espacio de dibujo, aunque es posible que el resultado no sea el mismo, ya que la función de energía es distinta y por ende el proceso de optimización puede resultar diferente.

Longitud de aristas

Como ya se ha visto, uno de los criterios estéticos más importantes es el de la longitud de aristas uniforme. Sin embargo, en la función de energía de DH no hay ningún potencial que exija una longitud fija de arista. Lo que sí se define es un potencial que penaliza las aristas de longitud grande. El mismo es extremadamente sencillo, y se define para cada par de nodos (u, v) conectados por una arista:

$$U_e(u, v) = d_{uv}^2 \tag{4.18}$$

Este potencial puede ser visto como la fuerza de un resorte de longitud cero, como los que se usan en el algoritmo de Tutte (ver sección 4.8). Lo que evita que los resultados obtenidos con DH estén concentrados en un único punto (o sea que todas las aristas tengan longitud cero) es la interacción con los otros potenciales, principalmente con los de repulsión. Más adelante volveremos sobre esto y calcularemos la longitud ideal de las aristas en este modelo.

Cruce de aristas

Cuando se discutieron los principales criterios estéticos se vio que empíricamente se ha comprobado que evitar los cruces de aristas es uno de los más importantes. Sin embargo, si bien los algoritmos vistos hasta ahora suelen dar resultados con pocos cruces de aristas, ninguno de ellos se encargó de evitar este problema de forma explícita. El motivo de esto es que evitar cruces es algo difícil de lograr, principalmente porque la noción de cantidad de cruces es discreta, y por lo tanto no puede ser usada en los métodos que requieren que la función de energía sea continua y diferenciable (como los anteriores, que usan el método del gradiente y el de Newton-Raphson). *Simulated Annealing*, en cambio, no requiere continuidad (aunque que la superficie de la función sea suave ayuda a mejorar la búsqueda) y mucho menos diferenciabilidad, ya que siempre se trabaja a nivel de la función, así que puede manejar la inclusión de un término discreto como la cantidad de cruces. El potencial que se agrega es uno solo, y no tiene más que la cantidad de cruces c:

$$U_c = c \tag{4.19}$$

Distancias nodo-arista

Si bien el término anterior penaliza cruces, aristas que "casi" se cruzan también son perjudiciales para la estética del grafo. Para evitar esto Davidson y Harel proponen un quinto potencial que generaliza la noción de cruce de aristas, y tiene la gran ventaja de ser continua. El potencial propuesto considera la distancia entre cada arista y cada vértice (definida como la distancia mínima entre el nodo y cualquier punto en la arista). Para cada arista $e = (u, v) \in E$ y nodo $w \in V$ donde w es distinto a u y a v, se define:

$$U_{en}(e, w) = \frac{1}{g_{ew}^2} \tag{4.20}$$

donde g_{ew} es la distancia mínima entre la posición de w y cualquier punto sobre la arista e.

En base a los cinco potenciales anteriores, (4.16)-(4.20), la función de energía de DH se define como

$$\begin{aligned} E &= c_r \sum_{u,v \in V, u \neq v} U_r(u, v) + c_b \sum_{u \in V} U_b(u) + c_e \sum_{(u,v) \in E} U_e(u, v) \\ &\quad + c_c U_c + c_{en} \sum_{e=(u,v) \in E, w \in V} U_{en}(e, w) \end{aligned} \tag{4.21}$$

donde cada potencial está multiplicado por una constante (c_r, c_b, c_e, c_c,c_{en}) que regula el peso relativo de cada término dentro de la función de energía. Ajustando

estos parámetros se pueden cambiar sustancialmente los trazados resultantes. Por ejemplo, aumentar los términos c_c y c_{en} priorizarán evitar los cruces resignando distribución uniforme y longitud de aristas corta (aunque es oportuno advertir que aumentar mucho el peso de los cruces puede disminuir demasiado la calidad general del trazado [BHR95]). También en relación a los pesos, en [BHR95] sugieren normalizar estos parámetros (por ejemplo entre 0 y 10) para facilitar el ajuste, ya que como estos números suelen tener distintos órdenes de magnitud, al usuario se le hace difícil equilibrarlos.

Algoritmo

DH(Grafo G)
1. Asignar temperatura inicial t=t_0.
2. Asignar una posición al azar a cada vértice de G
3. Repetir M veces:
    3.1. Repetir T veces:
        3.1.1. Para cada v $\in$ V
            3.1.1.1. p_{ant}= p_v
            3.1.1.2. p_v=p_v+Δ_{rand}
            3.1.1.3. Si E(p_{ant}) < E(p_v)
                3.1.1.3.1 Con probabilidad 1-$e^{\frac{E(p_{ant})-E(p_v)}{t}}$, volver a p_v= p_{ant}
    3.2 Reducir temperatura t

donde M y t_0 son constantes y E(p) es la energía que resulta al mover el vértice v a la posición p. T es la cantidad de iteraciones con una misma temperatura, que depende de $n = |V|$.

En el paso 3.1.1.2 se crea una nueva posición para v al sumarle Δ_{rand}. Δ_{rand} es un vector que determina cuánto y en qué dirección mover al vértice. En [DH96] este vector se elige al azar entre todos los que están en la circunferencia de radio r, donde r va decreciendo en proporción a la temperatura. La idea es que al comienzo los movimientos son grandes, y a medida que pasa el tiempo, y el trazado se acerca a una configuración de energía mínima, los movimientos se van haciendo cada vez más cortos.

Una cuestión muy importante de la que todavía no hemos hablado es el esquema de enfriamiento, es decir, qué temperatura inicial usar y cómo reducir la temperatura con el correr del tiempo. Respecto a la temperatura inicial, en [DH96] sugieren usar un número bien alto debido a que como el trazado inicial se elige al azar, no se puede saber cuánto valdrá la energía inicial, y el objetivo es que en las primeras etapas se permita casi cualquier movida. En caso de que se sepa que el trazado inicial está cerca de una configuración de energía mínima se puede elegir un t_0 más chico para obtener un resultado cercano al inicial.

Para la reducción de la temperatura hay dos cuestiones a decidir: cada cuánto disminuir la temperatura, y cuánto disminuirla. En muchas aplicaciones de *Simulated Annealing* la temperatura se mantiene fija por una cantidad de iteraciones

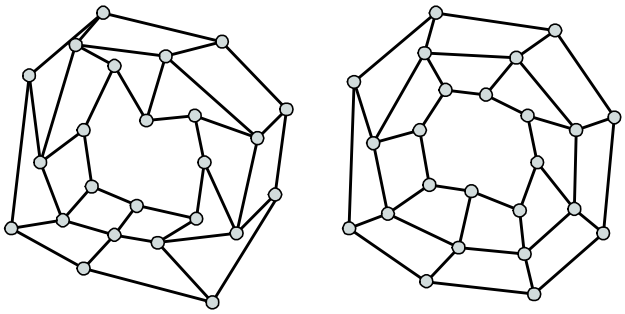


FIGURA 4.3. El resultado de aplicar DH sin (izq.) y con (der.) *fine tuning* [DH96].

polinomial en el tamaño de la entrada. Sin embargo, Davidson y Harel reportan que obtuvieron buenos resultados usando un número fijo de $T = 30n$ ($n = |V|$) iteraciones. Para la segunda cuestión, reducen la temperatura de manera geométrica:

$$t_{n+1} = \gamma t_n$$

donde γ es una constante entre $0,6$ y $0,95$ ($\gamma = 0,75$ parece dar buenos resultados en la mayoría de los casos).

Fine tuning

El algoritmo como hasta aquí fue presentado suele alcanzar trazados que son buenos topológicamente (es decir la ubicación relativa de los nodos es buena, se refleja bien la estructura del grafo), pero no estéticamente buenos. El motivo de esto es que es difícil estabilizarse en un buen trazado cuando hay tanto azar en juego. Es por esto que los autores introducen una etapa que denominan de *fine tuning* en la que prohíben las movidas cuesta arriba y sólo permiten movimientos de unos pocos pixels. La etapa de *fine tuning* se lleva a cabo luego de las iteraciones normales descriptas en los párrafos anteriores. La diferencia entre el trazado sin y con *fine tuning* es asombrosa. Si bien la estructura del trazado se conserva, visualmente es una mejora enorme. Un ejemplo puede verse en la Figura 4.3.

Algunas observaciones

El algoritmo DH es uno de los algoritmos más flexibles que hay para trazado de grafos. Es sustancialmente distinto a los anteriores, excepto por el hecho de que también usa criterios de atracción y de repulsión para ubicar los vértices. Con el que más parecidos encuentra es con el algoritmo KK, ya que ambos proveen una visión energética del problema, en lugar de una basada en fuerzas, y en menor medida con los de FR y GEM por el uso de temperaturas como parte del proceso de optimización (en realidad los otros dos algoritmos toman esta idea de *Simulated Annealing*).

El uso de una función general es el principal aporte de este algoritmo, ya que permite agregar cualquier criterio estético que pueda ser traducido en un potencial. En la práctica, los resultados obtenidos con DH – con los potenciales presentados aquí – son de muy buena calidad, comparables a los de los otros hasta aquí vistos [BHR95] y hasta tal vez mejores. Además el hecho de poder modificar sustancialmente los resultados cambiando sólo los pesos le da una ventaja sobre los otros algoritmos. Aunque hay que aclarar que esto también puede jugar en contra, ya que requiere dedicar un tiempo de ajuste con los parámetros hasta encontrar una combinación que se ajuste a las necesidades de uno. Otra ventaja de DH sobre los otros algoritmos es que considera de manera explícita los cruces de aristas. Obviamente, el algoritmo ideal es el que no requiere ningún parámetro y da siempre el resultado que uno espera. Lamentablemente esto suele ser difícil de lograr, aunque ha habido intentos de reducir la cantidad de parámetros del algoritmo DH, por ejemplo [Tun94].

La gran desventaja que tiene DH es su alto costo computacional, que es común a los algoritmos que usan *Simulated Annealing*, y que supera al costo de los otros algoritmos de esta sección. De hecho, en la prueba realizada en [BHR95] sugieren que la mejor opción es DH, siempre que el tiempo no sea un problema.

Veamos cuál es la complejidad del algoritmo DH. El ciclo externo se ejecuta M veces, con M constante –independiente del tamaño de la entrada–, así que el costo está dominado por cuánto lleva cada iteración de ese ciclo. A su vez, el ciclo 3.1 se ejecuta T_n veces, que usando la sugerencia de los autores estaría en $O(n)$. Actualizar la energía luego de un movimiento, que es necesario en el paso 3.1.1.3 tiene costo $O(nm)$, por lo que la complejidad total queda $O(n^2m)$.

Longitud ideal de las aristas

Si bien el algoritmo requiere cinco parámetros, ninguno es la longitud ideal de las aristas, que suele ser un parámetro común y muy importante. Sin embargo, la longitud ideal de las aristas, llamémosla l, está determinada por la interacción entre los potenciales de repulsión y los de aristas (U_r y U_e), y por ende depende de sus pesos, c_r y c_e. La misma está dada por $l = \sqrt[4]{\frac{c_r}{c_e}}$, lo cual puede verse de la siguiente manera.

Supóngase un grafo con sólo dos vértices conectados por una arista (u, v), cuya longitud es d_{uv}. Si omitimos los potenciales de los bordes, entonces la función de energía está dada por

$$\begin{aligned} E &= c_r U_r(u,v) + c_e U_e(u,v) \\ &= c_r \frac{1}{d_{uv}^2} + c_e d_{uv}^2 \end{aligned}$$

La configuración de energía mínima cumple que la derivada de E respecto de d_{uv} es cero, así que si igualamos a cero la derivada, tenemos

$$-2c_r \frac{1}{d_{uv}^3} + 2c_e d_{uv} = 0$$

que despejando d_{uv} queda

$$\begin{aligned} -2c_r + 2c_e d_{uv}^4 &= 0 \\ d_{uv} &= \sqrt[4]{\frac{c_r}{c_e}} \end{aligned}$$

lo cual nos confirma que la longitud de la arista para que la energía sea mínima debe ser $\sqrt[4]{\frac{c_r}{c_e}}$.

Nota: este mismo procedimiento se puede aplicar a cualquiera de los modelos de fuerzas de los algoritmos anteriores para verificar cuál es la longitud ideal de las aristas. Sólo hay que verificar que las fuerzas se anulan cuando d_{uv} es igual a la longitud ideal.

4.6. Resortes magnéticos: SM

El algoritmo que presentamos a continuación extiende el modelo del *spring embedder* con la idea de que algunos de los resortes están magnetizados y hay además un campo magnético global que actúa sobre los resortes magnetizados. Si bien este algoritmo es más una extensión que un nuevo algoritmo, lo incluimos de todas formas en esta sección porque fue de las primeras adaptaciones y porque numerosos trabajos posteriores se han basado y han extendido este algoritmo para ajustarlo a aplicaciones específicas (un ejemplo reciente es [HMN04]).

La motivación de este algoritmo, que llamaremos SM y fue presentado por Sugiyama y Misue en [SM95], es poder darle a las aristas algún tipo de orientación prefijada, por ejemplo, hacer que las aristas sean verticales u horizontales. Una posible aplicación de esto es para obtener resultados similares a los de los algoritmos específicos para grafos dirigidos o árboles.

La manera en que se logra este efecto es conceptualmente sencilla. Supongamos que queremos que algunas aristas sean verticales. En ese caso basta con definir un campo magnético vertical, que hace que los resortes que estén magnetizados tiendan por influencia del campo magnético a alinearse verticalmente. Lo interesante de este modelo, es que en lo único en lo que difiere con el modelo del *spring embedder* tradicional es en que existe un tipo de fuerza más, que es la que hace que los resortes magnetizados se "peguen" al campo magnético.

Modelo

Campos magnéticos

Tres tipos de campos magnéticos, que pueden ser combinados, son definidos (ilustrados en la Figura 4.4). A su vez, cada uno está definido por un vector orientación $m(x, y)$ que indica la dirección del campo magnético.

- Paralelos: todas las fuerzas magnéticas operan en la misma dirección.

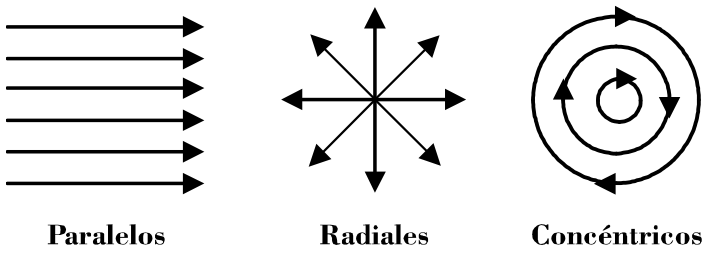


FIGURA 4.4. Tipos de campos magnéticos

El vector orientación es $m(x, y) \in \{e_1, -e_1, e_2, -e_2\}$, para e_i i-ésimo vector canónico de $\mathbb{R}^2$, dependiendo de si el campo es horizontal o vertical y dirección norte-sur o sur-norte .

- Radiales: las fuerzas operan radialmente a partir de un punto.

El vector orientación es $m(x, y) = (x, y)/\left\|(x, y)\right\|$.

- Concéntricos: las fuerzas operan en círculos concéntricos.

El vector orientación en este caso es $m(x, y) = s(y, -x)/\left\|(x, y)\right\|$, con $s \in \{1, -1\}$, dependiendo de si el campo es en sentido horario o en sentido antihorario.

Sugiyama y Misue dan un paso más y definen dos tipos de resortes o aristas magnetizadas: los unidireccionales y los bidireccionales. Los unidireccionales tienden a alinearse con la dirección del campo magnético ($m(x, y)$), mientras que los bidireccionales tienden a alinearse en cualquiera de los dos sentidos del campo magnético ($m(x, y)$ y $-m(x, y)$)

Fuerzas

Se definen tres tipos de fuerzas: atractivas, repulsivas y rotativas. Las dos primeras son las mismas que las usadas en el *spring embedder*. Las terceras son las que hacen que los campos magnéticos influyan en las aristas magnetizados.

Fuerzas atractivas
Se definen para todo par de vértices u y v conectados por una arista:

$$f_a(u, v) = C_1 \log(\frac{d_{uv}}{C_2})\overrightarrow{p_v p_u} \tag{4.22}$$

donde C_1 es una constante que controla la fuerza del resorte y C_2 es otra constante que define la longitud ideal del resorte. En sus experimentos los autores usan $C_1 = 2$ y $C_2 = 1$.

Fuerzas repulsivas

Se definen para todo par de vértices u y v distintos no conectados por una arista:

$$f_r(u,v) = \frac{C_3}{d_{uv}^2}\overrightarrow{p_u p_v} \tag{4.23}$$

donde C_3 es una constante de repulsión (en sus experimentos los autores usan $C_3 = 1$).

Fuerzas rotativas

Se definen para toda arista magnetizada $(u,v) \in E_{Mag}$ (denotaremos con E_{Mag} al conjunto de aristas magnetizadas).

$$f_m(u,v) = C_m b d_{uv}^{\alpha} \theta^{\beta} \overrightarrow{p_d} \tag{4.24}$$

donde:

b es la fuerza del campo magnético.

$m(x,y)$ es el vector de orientación del campo magnético[3].

θ ($-\pi < \theta < \pi$) es el ángulo en radianes entre la arista y la orientación del campo magnético[4].

α, β, y C_m son constantes para ajustar los resultados (en sus experimentos los autores usan $\alpha = \beta = C_m = 1$ y $0 \leq b \leq 16$).

El vector $\overrightarrow{p_d}$ es el que da la dirección del movimiento y se define como $\overrightarrow{p_d} = m(x,y) - \overrightarrow{p_e}$, donde $\overrightarrow{p_e}$ es el vector que va desde el medio del resorte hacia su extremo (el mismo que se usa para calcular el ángulo θ).

En base a (4.22), (4.23) y a (4.24) , la fuerza $f(u)$ que experimenta un vértice u es

$$f(u) = \sum_{(u,v)\in E} f_a(u,v) + \sum_{v\in V/(u,v)\notin E, u\neq v} f_r(u,v) + \sum_{v\in V/(u,v)\in E_{Mag}} f_m(u,v) \tag{4.25}$$

[3]El punto (x,y) en $m(x,y)$ correspondiente a una arista (u,v) es el punto medio w del segmento que conecta los dos vértices, $w = (p_v + p_u)/2$.

[4]Si el eje $e = (u,v)$ es unidireccional entonces tendrá un único extremo, digamos v, y definirá un único vector $\overrightarrow{p_e} = w - p_v$, donde $w = (p_v + p_u)/2$ es el vector medio del eje, en cuyo caso $\theta = \angle(\overrightarrow{p_e}, m(x,y))$. En cambio, si el eje es bidireccional, definirá dos vectores, $\overrightarrow{p_e}$ y $-\overrightarrow{p_e}$ y por ende dos ángulos. En ese caso el ángulo θ que se usa al calcular $f_m(u,v)$ en 4.24 es el que tiene menor valor absoluto.

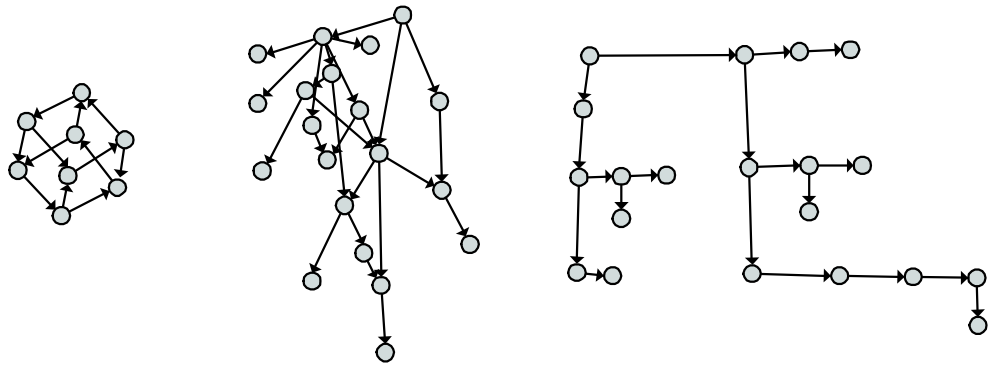


FIGURA 4.5. Trazados obtenidos con: (de izq. a der.) campos concéntricos, campos verticales y la combinación de campos horizontales y verticales.

Algoritmo

El algoritmo es muy parecido al del *spring embedder*, lo único que varía es el trazado inicial y, por supuesto, el cálculo de las fuerzas.

SM(Grafo G)
1. Asignar una posición a cada vértice de G colocándolos al azar sobre una circunferencia de radio $C_2 n/2\pi$
2. Repetir n veces:
   2.1. Para cada $v \in V$
      2.1.1 Calcular f(v) (en base a 4.25)
   2.2. Para cada $v \in V$
      2.2.1 $p_v = p_v + C_4 f(v)$

Para C_4 constante y $n=|V|$. En sus experimentos los autores usan $0,005 \leq C_4 \leq 0,1$. Notar que en lugar de situar los vértices al azar en cualquier parte del plano, el algoritmo los ubica sobre una circunferencia.

Algunas observaciones

Es interesante observar los resultados que se pueden obtener incluyendo estas fuerzas rotacionales. En la Figura 4.5 se muestran varios ejemplos.

La complejidad del algoritmo es igual a la del *spring embedder*, $O(n^3)$.

A diferencia de los métodos anteriores, SM puede usarse para dibujar grafos dirigidos, por ejemplo con las aristas apuntando para abajo. Esto se logra fácilmente agregando un campo magnético vertical en dirección norte-sur y usando aristas magnetizados unidireccionales.

También se puede usar para dibujar grafos ortogonales utilizando una combinación de un campo magnético horizontal y uno vertical, aunque obviamente

por las características del método, es difícil que el resultado sea exactamente un grafo ortogonal, pero tal vez sí uno "casi" ortogonal (es decir, con aristas "casi" ortogonales). Un ejemplo es el trazado de más a la derecha de la Figura 4.5.

4.7. Un algoritmo incremental: Tu

Tunkelang [Tun94] propuso un algoritmo incremental, al cual nos referiremos como Tu, que usa la función de energía de DH y se vale de varias heurísticas para ubicar a cada uno de los nodos. Incluimos este algoritmo en esta sección porque además de haber surgido tempranamente, es un buen ejemplo de cómo se pueden reutilizar parte de los modelos de los algoritmos anteriores.

El algoritmo Tu puede resumirse en tres etapas. La primera consiste en determinar el orden en el cual los nodos serán ubicados. A diferencia de los algoritmos anteriores, Tu es un algoritmo incremental en el que un nodo se va ubicando a la vez (en lugar de partir de un trazado inicial donde todos los nodos ya están ubicados). La segunda etapa consiste en ubicar a cada nodo, en el orden antes determinado. Para cada nodo que se debe ubicar, se toma una muestra de posibles ubicaciones, que dependen de los vecinos del nodo, y se elige la mejor de éstas (usando una función de costo basada en la de DH). Cada vez que un nodo es ubicado, se analizan de nuevo las ubicaciones de sus vecinos para ver si pueden mejorarse. Finalmente, la tercer etapa consiste en un *fine-tuning* final de todos los nodos, que consiste en ir uno a uno viendo si su situación puede mejorarse.

A continuación analizamos el algoritmo con más detalle.

Modelo

La función de costo que utiliza Tu es un subconjunto de la función de energía de DH (4.21), donde se usan sólo tres de los cinco potenciales definidos para DH: el de distribución de nodos (4.16), el de longitud de aristas (4.18) y el de la cantidad de cruces de aristas (4.19).

Utilizando la notación de DH, la función de costo de Tu es

$$F = c_r \sum_{u,v \in V, u \neq v} U_r(u,v) + c_e \sum_{(u,v) \in E} U_e(u,v) + c_c U_c \tag{4.26}$$

donde c_r, c_e y c_c son constantes que regulan el peso de cada término.

Notar que excepto por el término que corresponde a la cantidad de cruces, las fuerzas que usa Tu son muy parecidas a las usadas en FR.

Algoritmo

El algoritmo tiene la siguiente estructura:

Tu(Grafo G)

1. Calcular un ordenamiento de los vértices en base al árbol generador de G de altura mínima
2. Para cada vértice v ∈ V, siguiendo el ordenamiento del paso 1:
    2.1. Analizar posibles ubicaciones para v
    2.2. Ubicar v en la ubicación con menor costo
    2.3. Realizar optimización local en v y en sus vecinos
3. Realizar optimización local en todos los nodos (*fine-tuning*)

Lo primero que debe explicarse es cómo calcular el ordenamiento del paso 1. La idea de Tunkelang es dibujar el grafo desde un vértice del centro hacia afuera. Los vértices centrales del grafo son los nodos $c \in V$ que minimizan $\max_{v \in V} \delta_{cv}$, donde δ_{cv} es la distancia teórica entre c y v. Para calcular un nodo del centro y elegir un ordenamiento adecuado, el algoritmo recorre el grafo a lo ancho, haciendo un BFS (*breadth-first traversal*) desde cada nodo. Con cada BFS se obtiene un árbol, y se elige de todos ellos uno de altura mínima. El ordenamiento de los nodos comienza por la raíz r de ese árbol (que será un vértice del centro) y sigue a lo ancho (es decir, primero se ubica r, luego los nodos que están a distancia 1 de r, luego los que están a distancia 2, etc).

Pasemos ahora al paso 2.1, en el que se evalúan posibles ubicaciones para un nodo. A la hora de ubicar un nodo, se consideran un número finito de ubicaciones prefijadas. El espacio de dibujo se subdivide en celdas (que pueden llegar a ser cada uno de los pixels de la pantalla), donde cada una puede contener a lo sumo a un nodo. Cuando el vértice v debe ser ubicado, se analiza un conjunto de celdas que depende de los vecinos de v, para cada una se calcula la función de costo y se elige una que tenga costo mínimo.

El vértice correspondiente al centro del grafo se ubica siempre en el centro del área de dibujo. Cuando otro nodo va a ser ubicado, se itera a través de los vecinos de ese nodo y por cada vecino se consideran: i) las celdas contiguas al vecino, ii) las celdas a distancia l del vecino[5]. En total son 16 las celdas consideradas: 8 contiguas y 8 a distancia l. Además de para los vecinos, estas 16 posiciones también se consideran para el baricentro[6] de los vecinos. Finalmente, también se consideran las cuatro esquinas del espacio de dibujo.

La optimización local es otro factor importante del algoritmo. La optimización local en un nodo v consiste en analizar las celdas vecinas a las del nodo (nuevamente i) y ii)) y si alguna de ellas tiene costo menor al actual, mover el nodo a esa nueva celda. Además, cada vez que un nodo es movido como producto de la optimización local, recursivamente se hace optimización local en todos los vecinos del nodo movido.

[5]Denotaremos con l, usando la misma notación que en [Tun94], a la longitud ideal de las aristas.

[6]El baricentro del vértice v, con vecinos $N(v)$ (los ya ubicados en el trazado) se define como $\frac{1}{|N(v)|} \sum_{w \in N(v)} p_w$.

Algunas observaciones

En [Tun94] se proponen varias técnicas para acelerar el cálculo de la función de costo, ya que debido a la forma de proceder del algoritmo, es uno de los principales responsables del costo total del mismo. Una de las técnicas propuestas es el uso de una malla más gruesa, que denominan "malla uniforme", que permite calcular el número de cruces entre aristas de manera más eficiente, evitando recalcular todos los cruces cada vez que un nodo es movido. Además esta malla puede aprovecharse para incorporar el cálculo de las fuerzas repulsivas de forma similar a la variante malla del algoritmo FR (ver sección 4.2).

Es interesante que éste es el primer algoritmo – de los presentados en este capítulo – que se basa sólo en heurísticas para optimizar la función de costo. Los otros métodos usan el método del gradiente (por ej. el *spring embedder)*, Newton-Raphson (KK) o *Simulated Annealing* (DH), los tres son métodos generales para optimizar funciones. Sin embargo, Tu hace toda la optimización en base a una heurística propia.

Es también interesante el enfoque incremental del algoritmo, algo muy poco común dentro de los algoritmos dirigidos por fuerzas. El mismo Tunkelang comenta en [Tun94] que la forma en que se van agregando los nodos, desde el centro hacia afuera, es una especie de generalización de lo que se hace en los algoritmos para grafos dirigidos, como el de Sugiyama et al. [STT81], donde los nodos se van ubicando en distintas capas.

Los resultados que se obtienen con este algoritmo son distintos de los que se obtienen con los anteriores [BHR95], específicamente se ha observado que no capturan tanto la simetría como los otros, y que suele dar buenos resultados en mallas donde los otros producen distorsiones. También se observa que los resultados obtenidos con este algoritmo son más difíciles de predecir que con los otros.

El cálculo de la complejidad de este algoritmo es más complicado que en los anteriores por la naturaleza recursiva de la optimización local. El cálculo del ordenamiento consiste en hacer n veces BFS, lo cual da en total un costo $O(nm)$. La función de costo, luego de mover un nodo, puede ser recalculada en tiempo $O(dm)$, donde d es el grado del nodo movido. Lo que es difícil de estimar es el costo de la optimización local, ya que al ser recursiva depende de cuánto se puede mejorar el trazado del grafo actual. Lamentablemente, en [Tun94] no se profundiza el cálculo de este costo. Experimentalmente, de las pruebas hechas en [BHR95] se puede deducir que el tiempo de ejecución de Tu resultó intermedio, entre los de KK y GEM (más rápidos) y DH (más lento).

Finalmente, hay que destacar que Tu es uno de los pocos algoritmos, junto con DH, que contemplan el cruce de aristas de manera explícita.

4.8. El método baricéntrico: Tutte

El llamado método baricéntrico, propuesto por Tutte ([Tut60],[Tut63]) hace más de 40 años, es uno de los principales antecesores de los métodos dirigidos por fuerzas que se usan para trazado de grafos. Como se mencionó al comienzo de este

capítulo, su motivación era más matemática que de visualización, y debe advertirse que los resultados que se obtienen con este método son inferiores, en calidad, a los de los otros métodos presentados en este capítulo. De todas formas creemos que merece ser mencionado, primero porque es una de las principales influencias de los otros métodos, y segundo porque sus ideas son usadas en otra familia de algoritmos más recientes, los métodos espectrales (por ej. [Kor03]).

El método recibe su nombre porque su manera de optimizar consiste, a grandes rasgos, en ubicar a cada nodo en el baricentro de sus vecinos. Por otro lado, puede ser visto casi como un caso particular del *spring embedder* donde no hay fuerzas repulsivas (o usando la notación del *spring embedder*, $C_3 = 0$), la rigidez de cada resorte es 1 ($C_1 = 1$) y las fuerzas atractivas usan resortes lineales con longitud ideal cero. Una gran ventaja de este modelo de fuerzas es que es mucho más simple que los otros:

Modelo

Sólo se definen fuerzas atractivas entre los vértices conectados por una arista.

Fuerzas atractivas

Se definen para todo par de vértices u y v conectados por una arista:

$$f_a(u, v) = p_u - p_v$$

Donde p_u y p_v son los vectores posición de los vértices u y v.
La fuerza $f(u)$ que experimenta un vértice u está dada por

$$f(u) = \sum_{v \in V/(u,v) \in E} f_a(u, v) = \sum_{v \in V/(u,v) \in E} (p_u - p_v) \tag{4.27}$$

La función de energía que definen de manera implícita estas fuerzas está íntimamente relacionada con la energía de Hall [Hal70]. La misma juega un papel importante en los algoritmos espectrales, mencionados en el capítulo 2.

Hay un detalle muy importante acerca de estas fuerzas y es que una solución trivial donde estas fuerzas se minimizan es cuando todos los vértices están ubicados en el mismo punto (ya que todas las sumas de 4.27 dan trivialmente cero). Claramente esta no es una de las soluciones buscadas, por lo que para evitarla algunos vértices – al menos tres – son fijados, usualmente en los vértices de un polígono convexo. La optimización se realiza para los otros vértices, cuyas posiciones están "libres".

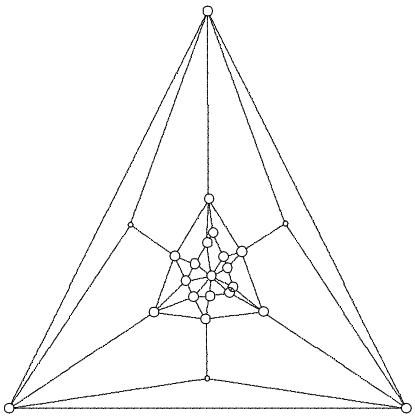


FIGURA 4.6. Trazado obtenido con el algoritmo baricéntrico de Tutte.

Algoritmo

El algoritmo consiste en ubicar cada vértice de los libre en el baricentro de sus vecinos.

Baricentro(Grafo G)
1. Fijar al menos tres vértices en los vértices de un polígono convexo.
2. Ubicar los otros vértices, vértices libres, en el origen.
3. Repetir hasta que las posiciones de los vértices libres no varían más que un ε
   3.1 Para cada vértice libre v $\in$ V:
      3.1.1 Ubicar v en $p_v = \frac{1}{\deg(v)} \sum_{(u,v)\in E} p_u$

Donde deg(v) es el grado del vértice v.

Algunas observaciones

En comparación con los métodos anteriores, el método baricéntrico no produce resultados de gran calidad, principalmente por la ausencia de fuerzas repulsivas que resultan en trazados de poca resolución. Por lo tanto, no es un método que se use mucho en las aplicaciones de trazado de grafos. Ver por ejemplo la Figura 4.6. Sin embargo, tiene varias propiedades buenas que valen la pena ser mencionadas.

Una de ellas es que si G es un grafo planar y triconexo, el resultado de aplicar este algoritmo es plano y convexo (es decir que cada cara del trazado es un polígono convexo). La demostración puede encontrarse en [Tut60].

La otra propiedad buena es que encontrar un equilibrio del sistema de fuerzas definido por (4.27) es equivalente a resolver un sistema de ecuaciones lineales esparso, por lo que existen métodos muy eficientes de calcularlo. De hecho, como se observa en [DETT99], el método de optimización que usa este algoritmo consiste simplemente en resolver iterativamente un sistema lineal (más detalles sobre esto en el capítulo 5). Esto es uno de los motivos por los que algunos de los métodos

que se usan para dibujar grafos grandes (ver capítulo 11) usen técnicas similares a esta.

4.9. Comentarios generales

4.9.1. Comparaciones entre los algoritmos

Dado el gran número de algoritmos "clásicos" que existen, los cuales fueron presentados en este capítulo, a la hora de elegir uno para implementar es lógico preguntarse cuál da resultados mejores que otros, cuál tarda menos o cuál tarda más. Algunos de estos puntos ya fueron cubiertos individualmente para cada algoritmo. En esta sección repasaremos los resultados disponibles en la literatura que comparan estos aspectos de los algoritmos presentados.

Lamentablemente, no existen muchos estudios empíricos que analicen estos factores que no sean los que cada autor presenta cuando introduce su propio algoritmo. El trabajo más elaborado al respecto es el de Brandenburg et al. [BHR95], donde **se comparan los siguientes cinco algoritmos: FR, KK, DH, Tu y GEM**. Sus principales conclusiones se resumen a continuación:

- Los cinco algoritmos obtienen grafos estéticamente agradables, y en líneas generales, el comportamiento es el reportado por sus respectivos autores.
- Los algoritmos son estables frente a grafos armados al azar, ya que casi siempre convergen hacia uno de los pocos trazados estables del grafo.
- Los trazados producidos por KK, FR, GEM y DH sin la optimización de cruces suelen tener apariencia similar.
- El algoritmo Tu muchas veces produce trazados distintos a los otros. En particular no captura la simetría como los otros, y su comportamiento es difícil de predecir.
- El más flexible es DH, pero también el más costoso en tiempo. Además, esta flexibilidad tiene el costo de que no es fácil ajustar los parámetros con los pesos de los componentes para obtener el resultado deseado. Notan también que valores altos en los términos relacionados con el cruce de aristas son incompatibles con los otros criterios y terminan destruyendo la simetría y la longitud uniforme de aristas.
- En lo que respecta a velocidad, GEM y KK requieren tiempos similares y son los más rápidos, mientras que DH es el más lento de todos. FR parece ser rápido para grafos de hasta 60 nodos, pero su performance decrece cuando son más grandes.

La conclusión final del trabajo es que no hay un ganador universal, y recomiendan probar con varios métodos para ver cuál da mejores resultados para el problema específico. El orden en que recomiendan probar los algoritmos es: GEM o KK

primero (o FR si el grafo es pequeño), y luego Tu y DH. Si el tiempo no es un inconveniente recomiendan jugar con los parámetros de Tu o DH hasta obtener resultados adecuados.

Los parámetros que se usaron para realizar las comparaciones de calidad fueron: la relación entre la arista de mayor longitud y la de menor longitud, la desviación estándar normalizada de la longitud de las aristas y el número de cruces. Los grafos usados fueron grafos generados aleatoriamente y 59 grafos varios tomados de los que se presentan en los artículos del algoritmo DH [DH96] y el de FR [FR91].

Otra comparación más reciente es la que presenta Behzadi [Beh99] con el propósito de comparar tiempo y calidad de algunos algoritmos tradicionales contra los de su propio algoritmo, *CostSpring* (del cual hablaremos en el Capítulo 12). En lo que respecta a la **comparación entre FR y GEM** sobre un conjunto de 34 grafos tomados principalmente de los usados por Frick et al. [FLM95] para probar el algoritmo GEM, los principales hallazgos de Behzadi son:

- Para los grafos planares más grandes (más de 100 nodos), tanto GEM como FR producen muchos cruces de aristas.
- La longitud de aristas suele ser similar, aunque GEM en muchos casos obtiene diferencias más pequeñas, lo cual es atribuido al uso de las fuerzas gravitacionales y las perturbaciones que se aplican a cada nodo al moverlo.
- En lo que discrepan estos experimentos con los de [BHR95] es en la performance de FR y GEM para grafos grandes. En sus pruebas, Behzadi encontró que FR resulta más rápido que GEM a medida que los grafos aumentan de tamaño, lo cual es contrario a los resultados de los experimentos de [BHR95] y los de los autores del GEM que se presentan en [FLM95]. Las diferencias con estos últimos son las más llamativas ya que supuestamente las pruebas fueron hechas usando la misma implementación de los algoritmos y computadoras muy semejantes.

4.9.2. Desventajas de los métodos dirigidos por fuerzas

Ahora que hemos presentado cuáles son los principales algoritmos de trazado de grafos dirigidos por fuerzas, podemos analizar cuáles son sus principales desventajas (sus ventajas más importantes ya han sido mencionadas en la sección 3.1).

Los algoritmos dirigidos por fuerzas tienen principalmente tres puntos débiles. Ellos son, en orden decreciente de importancia: su alto costo computacional (en tiempo), los mínimos locales pobres (que afectan a la calidad de los resultados obtenidos) y la falta de fundamentos teóricos sólidos que los respalden. A continuación analizaremos cada uno con más profundidad.

Alto costo computacional (en tiempo). Este es, desde nuestro punto de vista, el principal punto débil de los algoritmos dirigidos por fuerzas. Una complejidad de $O(n^3)$, como es la de casi todos los algoritmos que se vieron en esta sección, hace que estos métodos – al menos tal cual fueron presentados aquí – no sean escalables

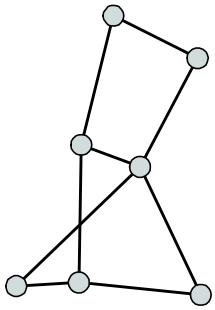


FIGURA 4.7. Mínimo local pobre obtenido con el *spring embedder.*

a grafos de más de unos pocos cientos de vértices, y menos aun si se los quiere para aplicaciones interactivas, donde los límites de tiempo son mayores.

Sin embargo, no todo está perdido en este aspecto. Como veremos más adelante, si uno está dispuesto a sacrificar algo de la calidad del resultado, y por sobre todo la simplicidad de los algoritmos (que recordemos que es una de las principales ventajas de estos métodos), entonces es posible trabajar con grafos de miles de vértices. Algoritmos que se basan en los que vimos en esta sección, específicamente diseñados para trabajar con grafos grandes, serán presentados en los próximos capítulos.

Mínimos locales pobres. Los métodos de optimización que usan los algoritmos dirigidos por fuerzas encuentran un mínimo local de la función de energía, ya que encontrar un óptimo global es un problema NP-Hard ([EMW86],[MO85]).

Más aun, la mayoría de los algoritmos usan métodos que sólo permiten movidas que minimizan el valor de la energía, por lo que el mínimo al que pueden llegar está predeterminado por la solución desde la cual comienza la búsqueda, que es el trazado inicial. Esto hace que dependiendo del trazado inicial, que muchas veces se elige al azar, se termine convergiendo a un mínimo local o a otro. El problema que surge de esto es que hay grafos que tienen mínimos locales que no son buenos y que pueden ser encontrados por el algoritmo. Lamentablemente, en la mayoría de los casos por la forma de realizar las movidas (siempre "en bajada"), una vez que el algoritmo queda atrapado en ese mínimo, no hay forma de escapar de él. Esto resulta en que hay veces que los resultados no son satisfactorios. Afortunadamente, excepto algunos grafos particulares, en la mayoría de los casos esto no ocurre.

Un ejemplo de mínimo local pobre puede verse en la Figura 4.7 donde el *spring embedder* llegó a un mínimo local malo que no pudo superar. En ese caso puede verse que un nodo ha quedado mal ubicado. Las mismas fuerzas repulsivas hacen que no haya forma de saltar la "barrera" que lo detiene sin aumentar la energía. Este problema de las barreras que evitan que un nodo ocupe una posición mejor es típico de los métodos dirigidos por fuerzas, y es una clara manifestación de los mínimos locales pobres.

Una de las formas de intentar evitar caer en un mínimo pobre es eligiendo cuidadosamente el trazado inicial. El capítulo 10 está dedicado enteramente a este

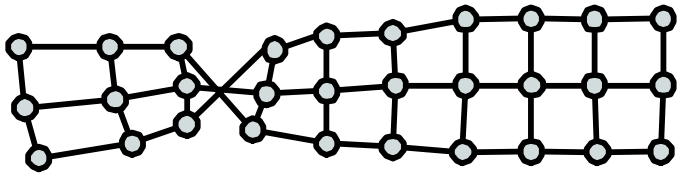


FIGURA 4.8. Mínimo local pobre de una malla.

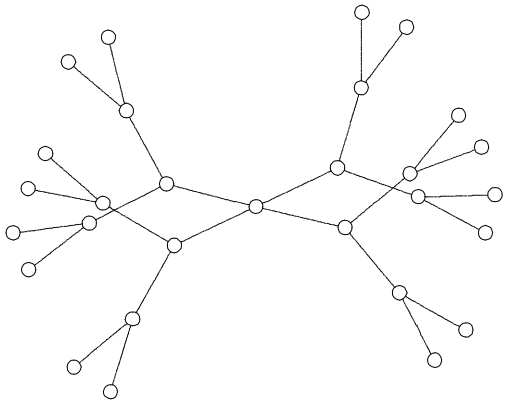


FIGURA 4.9. Mínimo local pobre para un árbol.

tema. Otra forma de intentar solucionar este problema es usando una función de energía que sea sensible a estas situaciones (como la de la barrera del ejemplo anterior) y haga que los que son mínimos en las funciones de energía convencionales, dejen de serlo. Obviamente eliminar todos los mínimos locales pobres es imposible, al menos si se mantiene la esencia de las funciones de energía tradicionales, pero pueden evitarse algunos mínimos malos que sean reconocibles de alguna manera. Un ejemplo de esto es el algoritmo *CostSpring* [Beh99], que usa una función de costo que considera, entre otras cosas, la distancia entre un vértice y todos sus vecinos, y permite evitar algunos mínimos locales como el de la Figura 4.8 (más detalles sobre este algoritmo se presentan en el capítulo 12).

Un ejemplo más general de las consecuencias de los mínimos locales es lo que ocurre con los árboles. Los algoritmos dirigidos por fuerzas producen resultados bastante pobres para árboles, en comparación con los que se pueden obtener para otros tipos de grafos. Un posible motivo de esto es que tienen una grandísima cantidad de mínimos locales malos (resultado de intercambiar dos subárboles de manera de que queden cruzados), como el de la Figura 4.9.Además, esto se combina con otro problema, y es que son grafos muy esparsos y al haber pocas aristas la mayoría del trazado queda determinado por las fuerzas repulsivas. Según [Tun99b], al ser estas menos suaves que las fuerzas atractivas, los métodos de optimización de primer orden, como el del gradiente, se ven lentificados.

Falta de fundamentos teóricos. Finalmente, existe un gran vacío de fundamentos teóricos alrededor de los algoritmos dirigidos por fuerzas. Ninguno de los algorit-

mos aquí presentados puede dar garantías de convergencia, ni tiene cotas conocidas en la cantidad de iteraciones que requieren. De ahí que en ningún caso se puede saber cuántas iteraciones del ciclo principal son necesarias para estar seguro de llegar a un mínimo (excepto en el caso del algoritmo baricéntrico, cuyos resultados son muy inferiores a los de los otros algoritmos).

Tampoco se conoce la relación entre los resultados que se obtienen y el óptimo al que se aspira, por lo que nunca se sabe de manera objetiva cuán cerca está el trazado obtenido del mejor.

Probablemente este vacío de teoría se deba a que tampoco existe casi teoría para el problema general del trazado de grafos. Citando a Tunkelang [Tun94] "es necesario entender mejor la teoría del trazado de grafos para poder diseñar heurísticas más específicas".

Uno de los pocos aspectos de los algoritmos dirigidos por fuerzas que ha sido estudiado desde la teoría es la relación entre estos algoritmos y la simetría que logran capturar. Los detalles pueden encontrarse en [EL00].

Capítulo 5

Aspectos numéricos de los métodos dirigidos por fuerzas

Como ya se ha mencionado, los algoritmos de trazado de grafos dirigidos por fuerzas tienen dos partes bien diferenciadas: el modelo de fuerzas o energía, y el algoritmo que busca un trazado bueno respecto a un conjunto de criterios estéticos, y que lo que en realidad hace es minimizar cierta función de energía o función de costo (o equivalentemente, buscar el equilibrio de un sistema de fuerzas).

El problema de minimizar –numéricamente– una función es un problema muy estudiado y que excede ampliamente el área del trazado de grafos. Por esto mismo, existe mucho conocimiento y muchas herramientas disponibles para este problema. Todos los algoritmos que vimos, y todos los algoritmos dirigidos por fuerzas que veremos en este trabajo, de alguna forma se encargan de minimizar una función.

Se puede decir que una vez definido el modelo de fuerzas, el algoritmo en sí mismo consiste en un algoritmo para optimizar numéricamente una función determinada. Lo interesante, que destaca Tunkelang en [Tun99b], es que en casi ninguno de los artículos que presentan los distintos algoritmos se reconoce el hecho de que el centro del algoritmo es una técnica para minimizar una función. La forma de optimizar la función, que además muchas veces no aparece de manera explícita, queda escondida detrás de algún tipo de operación con las posiciones de los vértices, sin revelar que en la mayoría de los casos el proceso que se aplica es una técnica general de optimización como el método del gradiente o Newton-Raphson.

Conocer lo que se está haciendo en realidad, desde el punto de vista numérico, tiene varias ventajas. Por un lado, aporta claridad al funcionamiento del algoritmo y al porqué de la convergencia de los mismos. Por otro lado, permite aprovechar la gran cantidad de herramientas que existen en el área de los métodos numéricos para poder encarar el problema desde nuevas perspectivas.

El objetivo de este capítulo es enfocarnos en la parte numérica de los algoritmos dirigidos por fuerzas. Revisaremos los descriptos en el capítulo anterior para destacar qué es en realidad lo que hacen desde este punto de vista. Mostraremos de manera explícita los algoritmos numéricos que se usan – algo que muchas veces es omitido por los mismos autores de los algoritmos – y comentaremos decisiones tomadas en otros algoritmos dirigidos por fuerzas de manera de presentar un panorama lo más amplio posible de las distintas técnicas que se usan para optimizar esta función, que aunque no siempre está a la vista, está presente en todos los algoritmos de esta clase.

5.1. Modelo de fuerzas vs modelo de energía

Los algoritmos que hemos visto hasta aquí dejan ver que hay dos grandes formas de expresar un algoritmo de trazado de grafos dirigido por fuerzas:

1. Como algoritmos que buscan encontrar un equilibrio en un sistema de fuerzas (*spring embedder*, FR, GEM, SM). Es decir que lo que se busca es un trazado donde los vértices estén en equilibrio (que la fuerza en cada vértice sea cero).

2. Como algoritmos que buscan minimizar una función de energía (KK, DH, Tu). Donde lo que se busca es un trazado de energía mínima.

Es importante tener en claro que estas formas de expresarlos son equivalentes. Esto se debe a que la función de energía y las fuerzas están íntimamente relacionadas, ya que las fuerzas constituyen el gradiente[1] negativo de la función de energía. Esto puede expresarse, con un poco de abuso de notación, de la siguiente manera:

$$\frac{\partial E}{\partial p_v} = -f(v)$$

donde $v \in V$, $p_v = (x_v, y_v)$ es la posición del vértice v en el trazado, E es la función de energía y $f(v)$ es el vector de $\mathbb{R}^2$ con la fuerza total que actúa sobre el nodo v. Más precisamente, si $f(v) = (f_x(v), f_y(v))$ y tomando como variables a los (x_v, y_v), tenemos

$$\begin{aligned} \frac{\partial E}{\partial x_v} &= -f_x(v) \\ \frac{\partial E}{\partial y_v} &= -f_y(v) \end{aligned}$$

De esto surge que en la práctica sea equivalente minimizar la energía que buscar un equilibrio de las fuerzas. Cuando la energía está en un mínimo, sus derivadas parciales son nulas. A su vez, las derivadas parciales de la energía son las fuerzas que actúan sobre cada vértice, por lo que un estado de energía mínima implica fuerzas nulas. En el otro sentido, si las fuerzas son nulas, entonces las derivadas parciales en ese trazado valen cero, y dadas las características de la función, el trazado debe ser un mínimo (o un punto estacionario, pero en la práctica esto no constituye un problema).

Hecha esta aclaración, de aquí en adelante diremos siempre que el objetivo de los algoritmos es minimizar la función de energía (que variará según el algoritmo), lo que nos permitirá reconocer más fácilmente los factores numéricos que entran en juego.

---

[1] El gradiente de una función $f : \mathbb{R}^n \to \mathbb{R}$ se define como $\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1} \quad \dots \quad \frac{\partial f(x)}{\partial x_n}\right)^t$.

En nuestro caso la f es la función de energía E, $E : \mathbb{R}^{2n} \to \mathbb{R}$, (con $n = |V|$) donde cada $x \in \mathbb{R}^{2n}$ define un trazado, especificando la posición (x, y) de los n vértices. Para alivianar la notación omitiremos cuando sea claro el x en $E(x)$.

5.2. El método del gradiente

Muchos de los algoritmos de trazado de grafos vistos hasta ahora, y de los que veremos a lo largo de este trabajo, utilizan el llamado método del gradiente. Este método, al igual que todos los métodos que mencionaremos en esta sección es un método clásico de optimización que puede encontrarse en cualquier libro sobre el tema (por ej. en [Fri94]).

El método del gradiente (también conocido como método de Euler para ecuaciones diferenciales) es uno de los llamados algoritmos de búsqueda direccional, en los que a partir de un punto en el dominio de la función, se elige una dirección en la cual avanzar. En los "métodos de descenso" se elige la dirección de forma tal que la función disminuya si se avanza por ella. Una vez determinada la dirección de descenso debe determinarse cuánto avanzar (comúnmente llamado "paso"). Este subproblema, a su vez, es conocido como "búsqueda lineal", y en principio es independiente de la forma en la que se elige la dirección.

El método del gradiente consiste simplemente en tomar como dirección el vector $-\nabla f(x)$, donde f es la función que se busca minimizar. Se puede demostrar que ésta siempre es una dirección de descenso.

Llevemos este método a nuestro problema de trazado de grafos. La función que queremos minimizar es la función de energía E. El dominio de esta función son los posibles trazados de nuestro grafo, o lo mismo, $\mathbb{R}^{2n}$, por los $2n$ valores que definen la posición en el plano de cada uno de los n vértices.

Ahora quedará claro que tanto el *spring embedder,* como la mayoría de los métodos que en él se basan (FR, SM, etc.), utilizan este método, aunque no sea aparente a simple vista. Recordemos que el gradiente de E lo conforman $2n$ componentes, 2 por cada vértice, siendo las correspondientes a v el par $(\frac{\partial E}{\partial x_v}, \frac{\partial E}{\partial y_v})$.

Vayamos ahora al algoritmo de la sección (4.1). En cada iteración del ciclo principal del *spring embedder* se hace:

2.1. Para cada v $\in V$
  2.1.1 Calcular f(v)
2.2. Para cada v $\in V$
  2.2.1 $p_v = p_v + C_4 f(v)$

El resultado final de ejecutar 2.1 y 2.2 es que cada vértice v es movido en dirección $f(v) = (\frac{\partial E}{\partial x_v}, \frac{\partial E}{\partial y_v})^t$. Si notamos con $t_k \in \mathbb{R}^{2n}$ al vector que tiene el trazado actual (tiene las posiciones actuales de todos los vértices), entonces lo que se está haciendo en cada iteración es tomar como nuevo trazado a $t_{k+1} = t_k - C_4 \nabla E(t_k)$. Y esto es justamente elegir siempre la dirección contraria al gradiente actual.

Es importante notar que el paso que se usa en el *spring embedder* es $\lambda = C_4 \|\nabla E(t_k)\|$. Esto hace que la elección equivocada de C_4 (demasiado grande) pueda hacer que la función E termine valiendo más en t_{k+1} que en t_k, lo cual es contrario a lo buscado, y además puede causar oscilaciones, haciendo que el algoritmo nunca converja. En la práctica, usar C_4 en proporción al que sugiere Eades

($\approx 0,1$) suele ser una buena elección. De todas formas, una elección más inteligente del paso podría permitir acelerar la convergencia. Fruchterman y Reingold hicieron esto para su algoritmo FR.

En lo que refiere a la forma de optimizar la energía, el algoritmo FR, presentado en la sección (4.2), es idéntico al de Eades, excepto en que el paso está restringido por una temperatura global. Inicialmente la temperatura es alta, permitiendo movimientos grandes (de hasta $\|\nabla E(t_k)\|$) y luego va decreciendo, restringiendo los movimientos cada vez más. La idea obedece al hecho de que en las primeras iteraciones los trazados necesitan movimientos grandes para tomar la forma topológicamente adecuada. A medida que esta forma se va definiendo, el paso disminuye para que sólo se hagan movimientos menores que no alteren mucho el aspecto del trazado.

El método del gradiente tiene una propiedad importante: los movimientos que produce son suaves. No hay saltos abruptos en las posiciones de los vértices (como ocurre con otros métodos, como los que usan KK o DH). Esta suavidad de los movimientos a veces es muy importante, por ejemplo cuando el algoritmo es parte de un sistema de trazado dinámico (ver capítulo 7), donde se quiere que el usuario pueda ir siguiendo con la vista la animación del movimiento de los vértices. A propósito, en [HCE98] puede encontrarse una versión más "dinámica" del método del gradiente, basada en la segunda ley de Newton.

Mirando al método del gradiente desde la perspectiva de la resolución de sistemas de ecuaciones diferenciales, Ostry [Ost96] observó que los sistemas que surgen en el problema del trazado de grafos dirigido por fuerzas pueden (y es bastante probable) que presenten una propiedad llamada rigidez (*stiffness*). La presencia de esta propiedad, que aparentemente surgiría principalmente cuando la solución es cercana a un mínimo, puede causar demoras en la convergencia e inclusive oscilaciones. Para solucionar este problema, Ostry sugiere usar técnicas específicas para lidiar con este tipo de sistemas, que según sus pruebas, aceleran la convergencia notablemente (aunque en [Ost96] no se presentan detalles cuantitativos de los resultados obtenidos).

5.3. El método de Newton-Raphson

El algoritmo KK (sección 4.4) utiliza un método de optimización distinto al método del gradiente de la sección anterior. Primero, su forma de encarar el problema numérico se diferencia de los otros por presentar explícitamente la función de energía E

$$E = \sum_{u,v \in V, u \neq v} \frac{1}{2} S_{uv} (d_{uv} - \delta_{uv})^2$$

En la sección (4.4) se vio que esta función surge de colocar un resorte lineal entre todo par de vértices y que la expresión de E se obtiene integrando las fuerzas producidas por los resortes. Sin embargo, vale la pena notar que esta función de energía no es usada de manera explícita en ninguna parte del algoritmo, ya

que siempre se trabaja con sus derivadas (que son las fuerzas), por lo que el mismo modelo podría haber sido formulado, al igual que los de la sección anterior, exclusivamente desde el punto de vista de las fuerzas, sin explicitar E.

El algoritmo KK se dedica a buscar un trazado donde las derivadas parciales de E se anulan. La segunda diferencia es que en lugar de usar el método del gradiente para esto – método para minimizar funciones – utilizan una variante del conocido método de Newton-Raphson, que en realidad es un método para buscar raíces de funciones.

Las diferencias son aun mayores, porque en lugar de usar Newton-Raphson para resolver el sistema de $2n$ ecuaciones que resulta de igualar a cero todas las derivadas parciales, y mover todos los vértices al mismo tiempo, Kamada y Kawai trabajan con un vértice a la vez, suponiendo que los otros están fijos. De esta manera reducen el problema a uno de dos variables, sobre el cual aplican el método iterativo de Newton-Raphson (y pasan de tener un problema de $2n$ variables a tener n problemas de 2 variables, lo cual es mejor desde el punto de vista de la cantidad de operaciones que se realizan).

Como se mencionó en la sección (4.4), Newton-Raphson para varias variables es un método iterativo de punto fijo con iteraciones de la forma

$$x_{n+1} = G(x_n) = x_n - J(x)^{-1}F(x) \tag{5.1}$$

donde si $F : \mathbb{R}^2 \to \mathbb{R}^2$ es la función cuya raíz se quiere encontrar, x_n y $x_{n+1} \in \mathbb{R}^2$ y $J(x) \in \mathbb{R}^{2\times 2}$ es la matriz Jacobiano de F. En el caso de KK, tenemos

$$F(x) = \begin{pmatrix} \frac{\partial E}{\partial x_m} \\ \frac{\partial E}{\partial y_m} \end{pmatrix} \text{ y } J(x) = \begin{pmatrix} \frac{\partial^2 E}{\partial x_m^2} & \frac{\partial^2 E}{\partial y_m \partial x_m} \\ \frac{\partial^2 E}{\partial x_m \partial y_m} & \frac{\partial^2 E}{\partial y_m^2} \end{pmatrix}$$

Todo el ciclo principal del algoritmo KK se dedica a seleccionar un vértice (el que tenga gradiente con norma mayor –considerando que los otros nodos están fijos–) e iterar con la fórmula (5.1) hasta que la derivadas son suficientemente pequeñas. Algunos comentarios más sobre aspectos numéricos de KK se hacen en la sección (4.4).

5.4. Tunkelang: optimizando con gradiente conjugado

La independencia entre el modelo de fuerzas y el algoritmo que usa para minimizarla hace que casi cualquier método para minimizar funciones o buscar raíces de funciones pueda ser aplicado al problema de trazado de grafos dirigido por fuerzas. Para aportar un nuevo ejemplo a los ya vistos, presentamos en esta sección un algoritmo de Tunkelang, presentado en [Tun99b] donde se usa el método de gradiente conjugado para la fase de optimización. (Aclaración: el algoritmo que presentamos aquí, si bien es del mismo autor, es distinto al algoritmo Tu de la sección 4.7).

Si bien en este capítulo nos concentramos en la fase de optimización, comentaremos brevemente el resto del algoritmo ya que constituye un algoritmo más que el lector puede encontrar útil, y que tiene algunas características interesantes.

Modelo y cálculo de las fuerzas

Las fuerzas que usa este algoritmo son casi las mismas que las de FR (sección 4.2). Las fuerzas atractivas son las mismas. En lo que difiere con FR es en las fuerzas repulsivas.

Las fuerzas repulsivas tienen dos usos: por un lado evitan que dos vértices se solapen o queden muy cerca. Por otro lado, hacen que los vértices se distribuyan uniformemente en el espacio de dibujo. Tunkelang sostiene que fuerzas como las de FR, inversas a la distancia, son demasiado fuertes para el segundo propósito. Por esto divide las fuerzas repulsivas en dos, una para cuando los vértices están cerca y otra, menos fuerte, para cuando están lejos:

$$f_r(u,v) = \begin{cases} \dfrac{K^2}{d_{uv}}\overrightarrow{p_u p_v} & d_{uv} \leq K \\ \dfrac{K^3}{d_{uv}^2}\overrightarrow{p_u p_v} & d_{uv} > K \end{cases}$$

donde K es la longitud ideal de la arista entre u y v.

Otra característica importante del algoritmo de [Tun99b] – además de que usa gradiente conjugado, sobre lo cual detallaremos a continuación – es que el cálculo de las fuerzas repulsivas se realiza de manera más eficiente, tratando de evitar el $O(n^2)$ que implica el cálculo directo. Con el mismo espíritu que la variante malla de FR, Tunkelang aproxima las fuerzas repulsivas usando un árbol de Barnes-Hut, técnica tomada de la simulación de N-cuerpos de la física. Los detalles los dejaremos para el capítulo 11.

Optimización basada en gradiente conjugado

El propósito de usar una técnica distinta al método del gradiente es obtener una convergencia más rápida. En particular, la velocidad del método del gradiente disminuye cuando el trazado se acerca mucho a un mínimo local [Tun99b], [Ost96]. El método del gradiente conjugado permite una convergencia más rápida y que no sufre de este problema. La clave es la forma en la que se eligen las direcciones de descenso.

A grandes rasgos, el método del gradiente conjugado que usa Tunkelang consiste en elegir direcciones p_k

$$p_k = -g_k + \left(\frac{\|g_k\|}{\|g_{k-1}\|}\right)^2 p_{k-1}$$

donde $g_k = \nabla E(x_k)$ es el gradiente evaluado en la solución x_k que se tiene en el paso k. Si la función que se intenta minimizar es cuadrática y su matriz Hessiano

definida positiva, y además el paso en cada iteración se calcula de forma exacta, este método cumple una serie de propiedades teóricas que garantizan la convergencia en tantos pasos como variables tenga la función (ver detalles en [GMW81]). Sin embargo, en nuestro caso no tenemos ni función objetivo cuadrática (E) ni Hessiano definido positivo, así que no hay garantías de que el método converja. Más aun, el cálculo del paso – la búsqueda lineal – es demasiado costoso como para que valga la pena realizarlo de forma exacta.

Teniendo en cuenta esto, Tunkelang modifica el método de gradiente conjugado de manera de que cada vez que una dirección deja de ser de descenso, se resetea el método (se vuelve a tomar como dirección $-\nabla E(x_k)$). La búsqueda lineal se realiza con un método "adaptativo" bastante sencillo basado en bisección. Los detalles se pueden consultar en [Tun99b]. Es interesante destacar que en sus experimentos Tunkelang encontró que el uso de procedimientos más precisos (y costosos) para elegir el paso no traían grandes beneficios en el desempeño del algoritmo. Respecto al criterio de corte del algoritmo, el mismo es muy conservador: se itera hasta que todo vértice se haya movido menos de medio pixel en la última iteración.

Estas modificaciones al método del gradiente conjugado, y el hecho de que la función no sea cuadrática con Hessiano definido positivo, dejan sin efecto las propiedades teóricas del método. Sin embargo, en la práctica Tunkelang reporta que la ventaja en velocidad sobre el método del gradiente es significativa.

Los resultados que se obtienen con este método de optimización, reportados por Tunkelang en [Tun99b], muestran que el método del gradiente conjugado consistentemente supera al método del gradiente, sobre todo cuando la cantidad de vértices aumenta.

Esto, sumado al método con el cual se aproximan las fuerzas repulsivas, hacen que este algoritmo sea bastante más eficiente que los algoritmos "clásicos", aunque también más difícil de implementar.

5.5. Resolviendo un sistema lineal: Tutte

De manera similar a lo que ocurre en el *spring embedder*, el método numérico usado en el algoritmo baricéntrico de Tutte (sección 4.8) está "escondido" detrás de un método mucho más intuitivo, que para Tutte es el de ubicar a cada vértice en el baricentro de sus vecinos.

Comencemos viendo cuáles son las fuerzas que intervienen. La fuerza que afecta a un vértice u está dada por

$$f(u) = \sum_{(u,v)\in E} (p_u - p_v) \tag{5.2}$$

que reescrito para $p_u = (x_u, y_u)$ nos da las derivadas parciales de la función de energía:

$$\begin{aligned}\frac{\partial E}{\partial x_u} &= \sum_{(u,v)\in E}(x_u - x_v)\\ \frac{\partial E}{\partial y_u} &= \sum_{(u,v)\in E}(y_u - y_v)\end{aligned}$$

El objetivo del algoritmo es hacer que todas estas derivadas parciales sean cero, por lo que la $F : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ cuya raíz se quiere encontrar es:

$$F\begin{pmatrix} x_{u_1}\\ \vdots\\ x_{u_n}\\ y_{u_1}\\ \vdots\\ y_{u_n}\end{pmatrix} = \begin{pmatrix} \frac{\partial E}{\partial x_{u_1}}\\ \vdots\\ \frac{\partial E}{\partial x_{u_n}}\\ \frac{\partial E}{\partial y_{u_1}}\\ \vdots\\ \frac{\partial E}{\partial y_{u_n}}\end{pmatrix} = \begin{pmatrix} \sum_{(u_1,v)\in E}(x_{u_1} - x_v)\\ \vdots\\ \sum_{(u_n,v)\in E}(x_{u_n} - x_v)\\ \sum_{(u_n,v)\in E}(y_{u_1} - y_v)\\ \vdots\\ \sum_{(u_n,v)\in E}(y_{u_n} - y_v)\end{pmatrix} \tag{5.3}$$

donde el grafo es $G = (V, E)$, con $V = \{u_1, ..., u_n\}$.

A diferencia de lo que ocurre con las fuerzas de los otros algoritmos vistos, esta F es una función lineal, por lo que cualquier técnica para resolver sistemas lineales puede ser usada.

En particular, un método iterativo muy conocido es el de Gauss-Seidel (ver [BF98]). El método consiste en despejar el valor de una variable en función de las otras, comenzando con una solución inicial, haciendo esto con cada variable y repitiendo el proceso hasta la convergencia.

Para nuestra función F, cuando en el paso k se debe actualizar el valor de x_{u_i}, se usa la i-ésima ecuación de F para despejar x_{u_i}, de la siguiente manera:

$$\sum_{(u_i,v)\in E}(x_{u_i} - x_v) = 0 \Leftrightarrow \deg(u_i)x_{u_i} - \sum_{(u_i,v)\in E} x_v = 0 \Leftrightarrow x_{u_i} = \frac{1}{\deg(u_i)}\sum_{(u_i,v)\in E} x_v$$

Con $\deg(u_i)$ igual al grado del vértice u_i. Por lo tanto el valor asignado a $x_{u_i}^{(k)}$es

$$x_{u_i}^{(k)} = \frac{1}{\deg(u_i)}\sum_{(u_i,v)\in E} x_v^{(h)} \tag{5.4}$$

donde $h = k$ ó $h = k - 1$, dependiendo de cuál esté disponible (siempre se intenta usar el más nuevo, así que la disponibilidad depende del orden en el que se van actualizando las variables x_u [2]).

Notar que 5.4 es exactamente lo que se hace en cada iteración del algoritmo baricéntrico de Tutte. Por lo tanto, el método de optimización que está detrás de este intuitivo y sencillo algoritmo es el de Gauss-Seidel.

También es importante observar que la matriz resultante es esparsa, por lo tanto puede resolverse usando herramientas específicas en tiempo $O(n^{1,5})$ [DETT99].

5.6. Métodos más generales de optimización

En el capítulo 4 se vio que existen algoritmos que utilizan métodos aun más generales que los anteriores para la etapa de optimización, siendo el mejor exponente el algoritmo DH que usa *Simulated Annealing*. Decimos que es todavía más general que los anteriores porque utiliza sólo información de la función E, mientras que los otros se basan en información de primer orden (gradiente). Usar sólo la función además permite que la misma tenga discontinuidades, lo cual permite introducir términos discretos como la cantidad de cruces en el trazado. La desventaja evidente de no usar las derivadas es que hay pocos elementos para guiar la búsqueda.

La mayor ventaja que presenta el uso de estos métodos generales, u otras *metaheurísticas*, es que son extremadamente flexibles. El precio a pagar suele ser convergencia lenta. (En el caso de *Simulated Annealing*, se estima que en promedio requieren 10 veces más tiempo que los basados en el *spring embedder* [KW01]).

5.6.1. Algoritmos genéticos

Otra metaheurística que ha sido aplicada con relativo éxito al problema de trazado de grafos son los algoritmos genéticos. A modo de ejemplo, describiremos el algoritmo de Branke et al. [BBS96].

Los algoritmos genéticos (ver por ej. [Gol89]) son un método estocástico de búsqueda global que ha tenido éxito en muchos problemas de optimización. Trabajan con una población de candidatos a soluciones e intentando optimizarlas a través de tres operaciones: selección, recombinación y mutación. En un algoritmo típico, la población inicial de soluciones se produce al azar. Luego, en cada generación subsecuente una nueva solución candidato es producida eligiendo dos candidatos con alguna probabilidad determinada. Los dos candidatos son recombinados para formar uno nuevo (llamado cría). Luego esta cría es sometida a una mutación (que es algún tipo de perturbación). La nueva cría se agrega a la población, y se descarta el "peor" candidato.

En el contexto de trazado de grafos presentado en [BBS96], los candidatos son los posibles trazados del grafo, representados por la posición de cada vértice. En

[2] Esto diferencia al método de Gauss-Seidel del método de Jacobi, donde siempre se usa $h = k - 1$.

este punto es bueno observar que otras representaciones son posibles. Branke et al. comentan que probaron otras, como por ejemplo incluir la longitud de las aristas, pero surgían algunos problemas que los llevaron a preferir la primera.

La inicialización, es decir, la generación de los individuos de la población inicial es realizada al azar (esto es equivalente a elegir un trazado inicial al azar).

De manera similar que en *Simulated Annealing*, la función de evaluación es un componente crucial del algoritmo. La función puede evaluar y asignar un peso a cualquier tipo de criterio estético. Los criterios con los que probaron Branke et al. son: número de cruces de aristas, promedio de fuerzas en cada nodo, variación en la longitud de las aristas, distancia mínima y máxima entre vértices, longitud mínima y máxima de aristas, distancia mínima entre un nodo y una arista, y número de ángulos diferentes en el trazado. Para la mayoría de sus pruebas consideraron los tres primeros más la distancia mínima entre un nodo y una arista.

La siguiente decisión que se debe tomar es cómo seleccionar los candidatos que generarán la cría. La opción elegida en [BBS96] es establecer probabilidades para cada candidato en función a cuán buena es la solución en relación con las otras de la población. Las soluciones mejores tendrán más probabilidad de ser elegidas que las soluciones pobres. En cada iteración se genera una cría, que es agregada, y se descarta la solución peor.

Luego debe definirse la recombinación, encargada de producir una cría en base a dos soluciones candidatas. La recombinación es llevada a cabo intercambiando en las dos soluciones (cada una un trazado) el trazado de un subgrafo elegido al azar. Las posiciones de todos los nodos del subgrafo son intercambiadas. Sin embargo, esto sufre de problemas cuando el mismo subgrafo aparece rotado, invertido o desplazado en un trazado respecto al otro, porque intercambiarlos directamente resulta en trazados pobres. Para solucionar este problema definen un operador que antes de realizar el intercambio desplaza y rota los trazados para alinearlos de manera aproximada.

El último factor a determinar es la etapa de mutación. En el algoritmo propuesto en [BBS96] la mutación se implementó agregando un valor al azar con distribución normal a cada uno de los individuos.

Además de lo anterior, que es propio de todo algoritmo genético, en [BBS96] se agrega al final de cada mutación una etapa de *fine-tuning* en la que un algoritmo basado en el *spring-embedder* se aplica al trazado, para mejorar aun más la calidad de la solución. Los autores observaron que esto produce un incremento en la velocidad de convergencia en las primeras iteraciones.

En lo que respecta a calidad, los resultados que se obtienen con este enfoque son similares a los obtenidos con DH. El algoritmo de Branke et al. tiene como supuesta ventaja adicional sobre el de DH el uso del *spring embedder* para mejorar las soluciones intermedias. Sin embargo, no hay suficientes pruebas empíricas que indiquen si esto logra producir resultados mejores a los de DH.

Como es de esperar, la gran desventaja de este método basado en algoritmos genéticos es su alto costo computacional. Además en [BBS96] se menciona que el algoritmo tal como fue expuesto puede requerir algunos ajustes antes de poder ser

aplicado a un problema real.

No son muchos los trabajos como el de Branke et al. en los que aplican algoritmos genéticos al trazado de grafos. Otros un poco más recientes son [RO98] y [BB00]. Probablemente su poco uso se deba a que si bien los resultados obtenidos con estos algoritmos son similares en calidad a los obtenidos con DH, los algoritmos son bastante más difíciles de implementar. De todas formas vale la pena tenerlos en cuenta como una opción más a la hora de buscar un algoritmo de trazado de grafos.

5.7. Otras

Si bien la mayoría de los algoritmos dirigidos por fuerzas suelen usar una de las técnicas de optimización anteriores (o similares), también es posible realizar la optimización basándose sólo en heurísticas *ad-hoc*. Un ejemplo que ya se vio en el capítulo anterior es el método que usa el algoritmo Tu (sección 4.7), donde la optimización usa una heurística basada en evaluar la función de energía en un conjunto prefijado de posiciones para cada vértice.

Capítulo 6

Trazados 3D

Hasta ahora hemos estado considerando sólo trazados donde los vértices se ubican en el plano. Sin embargo, representar el grafo en el espacio puede tener sus ventajas.

Desde el punto de vista del algoritmo que debe encontrar el trazado, hay ventajas y desventajas. Por un lado, el tener una dimensión más agrega un grado de libertad para ubicar a los vértices, lo cual da mayor flexibilidad y es más fácil cumplir algunos criterios estéticos. Los cruces de aristas, por ejemplo, tan problemáticos en el plano, siempre son evitables en 3D. Por otro lado, el trabajar con una dimensión más trae algunas nuevas complicaciones. Por empezar que la visualización del grafo en última instancia será en 2D, por lo cual se debe decidir cómo proyectar el trazado 3D a 2D. A su vez, estas proyecciones pueden hacer que se pierdan algunas de las propiedades estéticas que el trazado cumplía en 3D. Por ejemplo, pueden aparecer cruces de aristas. Para compensar esto las aplicaciones que usan trazado tridimensional de grafos suelen proveer distintos ángulos desde los cuales ver el trazado y otras operaciones que permitan "navegar" por el mismo.

Tanto o más importante es el punto de vista del usuario ¿tiene alguna ventaja visualizar el grafo en 3D? ¿Se puede transmitir más información con una dimensión más? Ha habido bastantes discusiones sobre estos puntos, y la conclusión ha sido que en general es beneficioso: hay pruebas empíricas que indican que se puede mostrar mejor la estructura de los datos y se puede percibir una mayor cantidad de información [Ost96],[Dwy00].

Ya sea para la visualización de *software* [Dwy00] o para visualizar interacciones entre proteínas [HB04], los trazados en tres dimensiones encuentran cada día más usos. Es por esto que ha surgido la necesidad de algoritmos adecuados para trazar grafos en 3D.

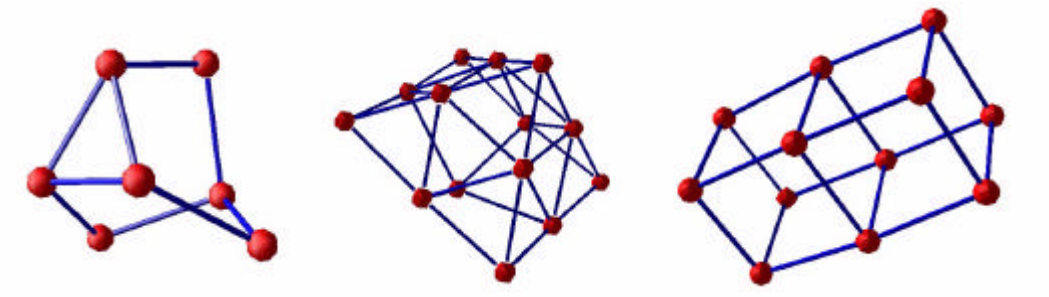


Figura 6.1. Trazados 3D producidos con el programa *WilmaScope*.

6.1. Adaptar algoritmos 2D a 3D

Dado que los objetivos del trazado de grafos siguen siendo los mismos en 3D que en 2D, la primer idea que surge es cómo adaptar los algoritmos existentes de trazado de grafos en 2D para que soporten una dimensión más.

De las tres familias principales de algoritmos de trazado de grafos: por capas, ortogonales y dirigidos por fuerzas, no todas se adaptan a 3D tan fácil como otras.

Los algoritmos que se basan en separar en capas pueden ser adaptados a 3D con relativamente poco trabajo, aunque algunas modificaciones son necesarias (como para resolver las colisiones).

Los algoritmos que producen trazados ortogonales, en cambio, requieren prácticamente algoritmos totalmente nuevos. Además los resultados no suelen ser buenos a medida que los grafos crecen en cantidad de nodos [KW01].

Una vez más, los algoritmos dirigidos por fuerzas demuestran su flexibilidad al ser llevados a tres dimensiones. Veamos qué cambios son necesarios en un algoritmo típico. El modelo básico compuesto de fuerzas repulsivas y fuerzas atractivas no requiere ninguna adaptación (excepto las obvias, hacer todas las operaciones en $\mathbb{R}^3$ en lugar de en $\mathbb{R}^2$). Las fuerzas, en sus exponentes más sencillos, sólo se calculan en base a distancias y longitudes ideales, así que nada de eso es específico del plano. El algoritmo, que busca un mínimo de la energía, en la mayoría de los casos es exactamente el mismo, considerando una coordenada más en las posiciones de los vértices.

Conclusión: la mayoría de los algoritmos dirigidos por fuerzas se extienden de manera natural a dimensiones mayores, ya que no hacen ninguna asunción sobre el número de dimensiones.

De todas formas hay adaptaciones que merecen ser comentadas, ya que si bien los algoritmos más sencillos (como el *spring-embedder*) se adaptan casi sin ningún cambio, algunos que son un poco más complicados pueden requerir adaptaciones no tan triviales (ej. GEM).

6.1.1. Adaptación de los algoritmos clásicos

Debido a la facilidad con que los algoritmos más sencillos pueden adaptarse, las publicaciones con algoritmos dirigidos por fuerza para 3D no tardaron en aparecer.

Tanto el *spring embedder* como FR no requieren ningún tipo de adaptación para 3D (excepto las obvias, que no las consideraremos de aquí en más), puesto que los términos de las fuerzas no usan nada propio de 2D, y tampoco los algoritmos. Se ve aquí una ventaja importante de que el método de optimización sea equivalente a algo intuitivo – e independiente de la dimensión – como mover el vértice en la dirección de la fuerza total que lo afecta, ya que hace que no se requiera adaptación alguna. De la misma forma, el algoritmo baricéntrico de Tutte (sección 4.8) se generaliza sin problemas, ya que la noción de baricentro no cambia para puntos en el espacio.

El primero de los algoritmos clásicos que presenta alguna dificultad es GEM.

GEM El algoritmo GEM (sección 4.3) fue adaptado a 3D por Bruß y Frick en [BF95], dando lugar al algoritmo GEM-3D.

El modelo es casi el mismo. La fuerza total $f(u)$ que experimenta un vértice u es definida como (manteniendo la notación de la sección 4.3):

$$f(u) = \sum_{(u,v)\in E} f_a(u,v) + \sum_{(u,v)\in V\times V, u\neq v} f_r(u,v) + f_g(u) + \rho \tag{6.1}$$

que es igual a la original con el único agregado de un vector ρ, que los autores llaman "componente de movimiento Browniano" y consiste simplemente en un pequeño vector al azar con esperanza $\mathbf{0}$.

Otra de las modificaciones incluye disminuir la constante correspondiente a las fuerzas gravitacionales. Según se indica en [BF95], en el caso tridimensional el uso de las mismas constantes que en 2D para estas fuerzas hace que el trazado quede muy concentrado.

La parte más delicada para adaptar es la de los cálculos de la temperatura local a cada vértice. Recordemos que el mismo tomaba en cuenta oscilaciones y rotaciones. Estas últimas no pueden ser llevadas a 3D directamente, ya que no es posible controlar todos los posibles planos de rotación. Para suplantar esto Bruß y Frick consideran tres opciones:

1. Proyectar sobre tres planos (y-z, x-z y x-y) y detectar rotaciones dentro de esos planos.

2. Detectar sólo rotaciones con ángulos de $\approx 90^o$.

3. No considerar rotaciones y agregar un esquema de enfriamiento local.

Lamentablemente, en [BF95] no queda claro cuál de los tres da mejores resultados.

Acerca de la calidad de los trazados obtenidos, comentan que se logra espaciar los vértices y capturar la simetría sin problemas, y que en general la topología 3D de los grafos suele aparecer como resultado. Por ejemplo, comentan que logran obtener un trazado del grafo de Petersen del estilo de los de libro de texto (Figura 6.2), lo cual, al menos al momento de la publicación del artículo, no se había podido obtener con ningún algoritmo de trazado 2D.

KK En lo que hace al modelo, adaptar KK a 3D no requiere ningún cambio. Sin embargo, el método de optimización requiere ser adaptado para una nueva dimensión. Recordemos que KK congelaba todos los vértices menos uno y resolvía un problema de 2 variables usando Newton-Raphson. En 3D el problema pasa a ser de 3 variables, y es necesario recalcular la expresión de Newton-Raphson para este problema tridimensional. Comentaremos brevemente las adaptaciones necesarias, siempre con la notación de la sección 4.4.

La función de energía se extiende a 3D simplemente como:

$$E = \sum_{u,v\in V, u\neq v} \frac{1}{2} S_{uv} (\sqrt{(x_u - x_v)^2 + (y_u - y_v)^2 + (z_u - z_v)^2} - \delta_{uv})^2$$

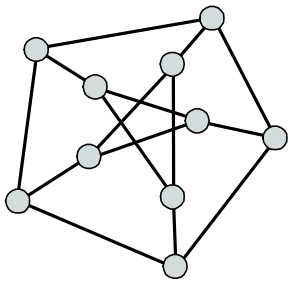


FIGURA 6.2. Trazado del grafo de Petersen obtenido con GEM-3D [BF95].

El objetivo del algoritmo KK era encontrar un cero de las derivadas parciales, que ahora en tres dimensiones, tienen la forma:

$$\begin{aligned}
\frac{\partial E}{\partial x_m} &= \sum_{v\in V, v\neq m} S_{uv}\left\{(x_m - x_v) - \frac{\delta_{uv}(x_m - x_v)}{\sqrt{(x_m - x_v)^2 + (y_m - y_v)^2 + (z_m - z_v)^2}}\right\} \\
\frac{\partial E}{\partial y_m} &= \sum_{v\in V, v\neq m} S_{uv}\left\{(y_m - y_v) - \frac{\delta_{uv}(y_m - y_v)}{\sqrt{(x_m - x_v)^2 + (y_m - y_v)^2 + (z_m - z_v)^2}}\right\} \\
\frac{\partial E}{\partial z_m} &= \sum_{v\in V, v\neq m} S_{uv}\left\{(z_m - z_v) - \frac{\delta_{uv}(z_m - z_v)}{\sqrt{(x_m - x_v)^2 + (y_m - y_v)^2 + (z_m - z_v)^2}}\right\}
\end{aligned}$$

Recordemos que el método de Newton-Raphson ([BF98]) consiste en una iteración de la forma

$$x_{n+1} = x_n - J(x)^{-1}F(x) \tag{6.2}$$

que ahora en 3 variables tiene x_n y $x_{n+1} \in \mathbb{R}^3$, $F : \mathbb{R}^3 \to \mathbb{R}^3$ (función cuya raíz se quiere encontrar) y $J(x) \in \mathbb{R}^{3\times 3}$, matriz Jacobiano de F. En nuestro caso,

$$F(x) = \begin{pmatrix} \frac{\partial E}{\partial x_m} \\ \frac{\partial E}{\partial y_m} \\ \frac{\partial E}{\partial z_m} \end{pmatrix} \text{ y } J(x) = \begin{pmatrix} \frac{\partial^2 E}{\partial x_m^2} & \frac{\partial^2 E}{\partial y_m x_m} & \frac{\partial^2 E}{\partial z_m x_m} \\ \frac{\partial^2 E}{\partial x_m y_m} & \frac{\partial^2 E}{\partial y_m^2} & \frac{\partial^2 E}{\partial z_m y_m} \\ \frac{\partial^2 E}{\partial x_m z_m} & \frac{\partial^2 E}{\partial y_m z_m} & \frac{\partial^2 E}{\partial z_m^2} \end{pmatrix}$$

Al igual que en el caso bidimensional, el desplazamiento $(\delta x, \delta y, \delta z)$ del vértice en cada paso de la iteración 6.2 (donde $(\delta x, \delta y, \delta z)^t = -J(x)^{-1}F(x)$) puede calcularse evitando el cómputo de la inversa del Jacobiano, resolviendo un sistema de ecuaciones de 3×3, que es lo que usualmente se hace al implementar el algoritmo. Los detalles pueden encontrarse en [SSL00] y en [HB04].

DH El algoritmo DH cuenta con varias adaptaciones de 3D, siendo una de las primeras en la literatura la de [CT95]. Adaptar el algoritmo DH es adaptar sus potenciales. Todos se pueden extender directamente excepto el que corresponde a la cantidad de cruces, que en 3D deja de tener valor. En [CT95] en su lugar utilizan un término de repulsión arista-arista.

En [MRS95], Monien et al. presentan un algoritmo de trazado 3D que usa *Simulated Annealing* pero que usa tres potenciales distintos a los de DH. Si bien no tienen nada que no se pueda aplicar a 2D, vale la pena mencionarlos como potenciales alternativos que se han usado con éxito. Los mismos son:

1. Un potencial de ángulos, que penaliza ángulos pequeños entre aristas incidentes a un mismo nodo.

2. Un potencial de longitud de aristas, que a diferencia del usado en DH, combina una función hiperbólica (para cuando el eje mide menos que su longitud ideal) y una lineal (para cuando mide más). Además toman en cuenta el grado de los vértices extremos de la arista (si alguno de los vértices tiene grado grande, la fuerza tiende a disminuir, mientras que si ambos tienen grado pequeño, la fuerza aumenta).

3. Un potencial de "pseudo longitud de aristas" que obliga a que los vértices no adyacentes (en realidad los vértices con distancia mayor que 2) estén lejos unos de los otros.

Las fórmulas de los potenciales y los detalles del esquema de enfriamiento (que es muy distinto del usado en DH) pueden encontrarse en [MRS95].

Otra característica importante del algoritmo que proponen es que es paralelizable. En la implementación de Monien et al., los procesadores comienzan explorando soluciones distintas. En un momento dado uno puede ver que otro procesador tiene una solución mucho mejor que la suya, y puede elegir comenzar a trabajar con la solución del otro. Así varios procesadores pueden estar al mismo tiempo explorando una misma región del espacio de búsqueda, mientras que se invierte poco esfuerzo en explorar áreas con soluciones pobres.

SM El algoritmo de los resortes magnéticos (sección 4.6) también ha sido llevado a 3D con éxito (por ej. [Dwy00],[DE02]). El modelo es lo único que requiere ser adaptado, ya que la optimización se realiza – al igual que en FR – con el método del gradiente. A su vez, lo único en lo que difiere el modelo es en las fuerzas de los campos magnéticos (fuerzas rotativas). Específicamente, los únicos que deben adaptarse son los campos magnéticos.

El tipo de campo magnético más usado, el paralelo, se generaliza directamente a 3D, al igual que los radiales. Para el caso de los campos concéntricos hay varias opciones, dependiendo del uso que se le quiera dar: por ejemplo se pueden usar esferas o cilindros.

Tu El algoritmo Tu (sección 4.7) no tiene una adaptación sencilla debido a que usa heurísticas para optimizar la función y para el trazado inicial. Teóricamente se podrían extender las heurísticas para la optimización a 3D, pero no parece ser obvio cómo hacerlo sin elevar al cuadrado la cantidad de posiciones que se analizan en cada etapa de optimización local, lo cual no sería bueno desde el punto de vista de la velocidad. No se han encontrado en la literatura extensiones de este algoritmo de ninguna clase.

6.2. Otros algoritmos 3D

Son muchos los algoritmos dirigidos por fuerzas para trazado 3D que se han presentado. De hecho, en los últimos años han sido tantos o más que para 2D. Sin embargo, los algoritmos utilizados casi siempre se basan en alguno de los algoritmos clásicos aquí revisados, que a su vez son versiones extendidas de los algoritmos clásicos del capítulo 4. Los cambios que se introducen son generalmente cambios en el modelo, para contemplar restricciones propias del dominio de aplicación, y mejoras (en general mínimas) en el proceso de optimización, pero por detrás siempre está uno de los algoritmos clásicos.

Por ejemplo, en [SSL00] se usa KK en 3D para dibujar grafos densos relacionados con ingeniería de *software*. En [HB04] también se usa KK en 3D para la visualización de redes de interacción entre proteínas. Dwyer [Dwy00] combina FR y resortes magnéticos para visualizar modelos UML en tres dimensiones.

Un algoritmo pensado directamente para 3D (aunque en realidad no tiene mucho propio de 3D) es el presentado por Ostry en [Ost96]. Básicamente es una adaptación a 3D de un algoritmo que usa el modelo de fuerzas del *spring embedder* pero para optimizar usa un *solver* de ecuaciones diferenciales. Como se mencionó en el capítulo 5, el método del gradiente puede también ser visto como un método para resolver sistemas de ecuaciones diferenciales. La principal ventaja de usar este *solver* es que incluye técnicas para lidiar con una propiedad de este tipo de ecuaciones llamada *rigidez*, que puede causar retrasos en la convergencia y oscilaciones (ver capítulo 5).

Capítulo 7

Trazado de grafos dinámicos

Los grafos de los que nos hemos ocupado hasta ahora han sido siempre grafos "estáticos", en el sentido de que el algoritmo de trazado se encarga de un único grafo que no cambia – es estable – durante todo el cálculo del trazado, y una vez que termina con el trazado, se "olvida" del grafo.

Supongamos ahora que luego de calcular el trazado de un grafo G_0, con alguno de los algoritmos vistos, se requiere calcular el trazado de G_1, grafo igual a G_0 pero con un vértice y una arista más. No nos quedará otra alternativa que volver a aplicar nuestro algoritmo, pero ahora sobre G_1. Si al rato de terminar con G_1, nos surge la necesidad de hacer lo mismo con G_2, que es casi igual a G_1 con un par de aristas menos, por tercera vez habrá que calcular todo.

Si bien esta puede parecer una situación artificial, ocurre muy seguido en algunas aplicaciones. Imaginemos que una empresa tiene un gran sitio web, que actualiza varias veces por día, y que quiere mantener el trazado del grafo con la estructura del sitio[1] para guiar a los diseñadores y ayudar a los empleados a encontrar las distintas páginas que necesitan. Cada vez que una página se agrega o se elimina o un nuevo enlace se agrega/elimina entre dos, el grafo cambia, y es necesario recalcular el trazado. Sin embargo el nuevo grafo va a ser muy parecido al anterior, con la excepción de unos pocos vértices/aristas.

Un grafo como el de la empresa del ejemplo es conocido como **grafo dinámico**. Puede ser visto como un grafo inicial G_0 que va sufriendo modificaciones generando nuevos grafos G_1, G_2, ... con la característica de que la diferencia entre G_i y G_{i+1} es muy poca: se agregan y/o eliminan unos pocos vértices y aristas.

Otra situación común donde aparecen los grafos dinámicos es en aplicaciones interactivas que utilizan (y visualizan) grafos sobre los cuales el usuario puede hacer modificaciones: agregar vértices, aristas, etc. Es de esperar que cuando el usuario agregue una arista al grafo que está en su pantalla y se recalcule el trazado el resultado sea muy parecido a lo que tenía antes. Debido a que ésta es una situación muy frecuente, los trazados de grafos dinámicos también son conocidos como trazados interactivos.

La solución más sencilla – y la única al alcance de los algoritmos hasta ahora vistos – es tratar a cada grafo G_i como uno distinto y realizar para cada uno un trazado desde cero.

Esto tiene dos grandes desventajas:

1. **Eficiencia**. Dado que los grafos consecutivos casi no varían, la mayoría del trazado anterior podría ser reutilizado para el segundo trazado, ahorrando gran parte del esfuerzo del cálculo del trazado.

[1] En el supuesto grafo los vértices serían las páginas web y las aristas representarían los enlaces entre las mismas.

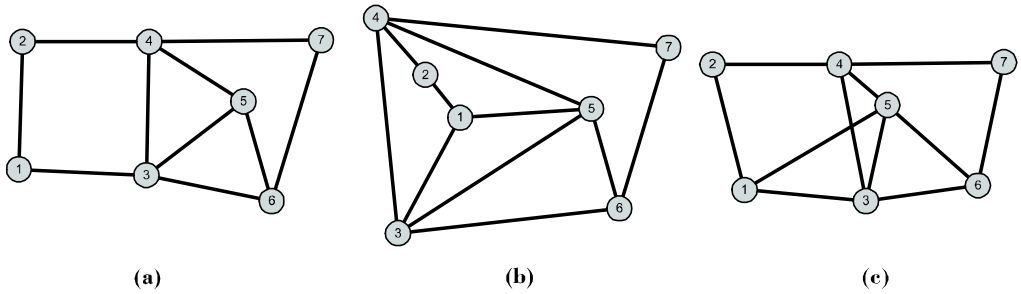


FIGURA 7.1. Dos trazados del grafo resultante de agregarle al grafo (a) la arista (1,5).

2. **Mapa mental**. Existe la posibilidad de que los trazado de G_i y el de G_{i+1}, si son tratados como problemas independientes, sean muy distintos entre sí. Recordemos que muchos de los algoritmos dirigidos por fuerzas tienen componentes azarosos (muchos inclusive parten de un trazado elegido al azar). Si una persona ha consultado varias veces el trazado y ya tiene en mente la ubicación de algunas páginas, no le agradará en su próxima consulta encontrar que le han movido de lugar muchas de las páginas cuya ubicación ya conocía. Esta idea de dónde están las cosas en el trazado ha recibido el nombre de "mapa mental". Este concepto de "mapa mental" es el elemento más importante en el trazado de grafos dinámicos, así que le dedicaremos la próxima sección.

7.1. El mapa mental de un trazado

El principal objetivo del trazado de grafos dinámicos es mantener el mapa mental del usuario. El usuario es la persona que va a "utilizar" el trazado, el que va a visualizar la información representada por el grafo. El mapa mental [ELMS91] es lo que el usuario aprendió sobre la estructura del trazado, al visualizarlo y navegarlo.

Cada vez que el trazado del grafo es modificado, el usuario debe reconstruir su mapa mental, es por eso que se pretende modificar lo menos posible el trazado, pero al mismo tiempo se quiere respetar los criterios estéticos generales del trazado de grafos. En la Figura 7.1 vemos un ejemplo de un grafo inicial (a), y en el medio y derecha (b y c) dos trazados del grafo resultante de agregar al de la izquierda una arista entre los vértices 1 y 5. Tanto el trazado del medio como el de la derecha corresponden al nuevo grafo, pero el del medio claramente es muy distinto al inicial (no preserva el mapa mental), mientras que el de la derecha se asemeja mucho y es además es estéticamente bueno pese a no ser plano.

Las principales medidas que se adoptan para intentar preservar el mapa mental son dos [KW01]:

1. Animar los movimientos de los vértices entre dos grafos consecutivos, de manera de que el usuario pueda ir siguiendo los cambios y adaptando su

mapa mental al mismo tiempo.

2. Minimizar los cambios entre grafos consecutivos, pero a la vez intentando cumplir los criterios estéticos del trazado de grafos. Estos dos objetivos suelen entrar en conflicto, por lo que se busca una solución de compromiso.

En la mayoría de los casos (1) y (2) se aplican juntas: a partir de G_{i-1} y G_i se calcula el trazado i-ésimo, y para mostrarlo se hace una animación entre ambos trazados.

Asociada al segundo punto surge la pregunta de qué significa minimizar los cambios entre dos grafos. Dos opciones, que también son combinables, son las más comunes: 1) restringir los cambios a un subconjunto del grafo y 2) usar una métrica para medir el cambio, tratando de minimizar la distancia entre los dos trazados.

La primera opción tiene varios puntos a favor. Es fácil de implementar y puede disminuir el costo de los trazados consecutivos[2]. Es necesario decidir qué vértices pueden moverse y cuáles no. Un extremo es que ningún vértice de G_i que también esté en G_{i-1} pueda moverse. Lo malo de esto es que es probable que la calidad del trazado resultante sea muy mala (por ej. que aparezcan muchos cruces de aristas). Existen muchas opciones intermedias, como que sólo se modifiquen los vértices "directamente afectados" por los cambios (que podrían ser, por ejemplo, sólo los vértices nuevos y los vértices con nuevos vecinos).

La segunda opción permite más flexibilidad, pero requiere definir una métrica adecuada. Muchas métricas han sido propuestas y analizadas (por ej. [BT98], [BT01]). Entre las más comunes, que se aplican a grafos generales, se encuentran:

- Promedio, suma de distancias entre vértices y suma de distancias relativas. Esta son de las más simples y más usadas.
- Distancia de Hausdorff. Métrica para medir distancia entre conjuntos de puntos.
- Métricas de proximidad. Sobre la idea de que si dos vértices están próximos en G_{i-1}, deben estarlo también en G_i. Hay varias métricas para medir esto, generalmente son medidas que capturan los *clusters* del grafo.
- Topología. Métricas para medir la preservación del orden de las aristas alrededor de un vértice.
- Ángulos. Métricas para medir la diferencia de ángulo entre las aristas que están en ambos trazados.

Los detalles de todas estas métricas, junto a otras para trazado ortogonal, pueden encontrarse en [BT98].

[2]Aunque esto depende mucho del algoritmo que se use para el trazado estático y de cómo se lo implemente.

7.2. Agregando dinámica a los algoritmos dirigidos por fuerzas

Las dos opciones que se mencionaron en la sección anterior para lidiar con grafos dinámicos son fácilmente combinables con un algoritmo dirigido por fuerzas.

Si se elige limitar por completo el movimiento de algunos vértices, lo único necesario para "fijarlos" es no moverlos (por más evidente que parezca). Mientras que si se prefiere trabajar en función de una métrica, también es fácil la integración. Si se usa un método como DH que minimiza una función de energía arbitraria, alcanza con agregar un nuevo potencial con la métrica deseada. Si se usa un algoritmo del estilo del *spring embedder* dependerá de qué métrica se use. La más fácil de incluir es una que penalice que los vértices se alejen de sus posiciones en el trazado anterior, lo cual se puede implementar agregando una nueva fuerza al modelo (que atraiga a los vértices hacia sus posiciones anteriores, como resortes de longitud ideal cero).

A continuación explicaremos con más detalle varias de las adaptaciones posibles.

La opción más sencilla y que ha mostrado dar buenos resultados es animar los movimientos entre trazados consecutivos y minimizar los cambios en las posiciones usando una métrica basada en las distancias de los vértices. Un caso concreto donde se aplicó esto es en [BKL+00], donde Brandes et al. presentan un método para trazar grafos dinámicos tomados de la *World Wide Web*. Mencionaremos aquí los aportes de este trabajo que hacen al trazado dinámico.

El modelo elegido para el algoritmo de trazado estático fue una función de energía basada en potenciales muy similar a la de DH (sección 4.5), con potenciales de repulsión, distancia, cruces entre aristas y vértices, atracción hacia el centro del dibujo y rotación (para favorecer ejes mirando hacia abajo, ya que el método es para grafos dirigidos).

Para que los vértices no se alejen mucho de sus posiciones en el trazado anterior, definen un algoritmo de trazado dinámico que es igual al estático, pero agregando un sexto potencial de estabilidad para todo vértice v, con la forma

$$U_s(v) = \sigma dist(p_v^i, p_v^{i-1})^2$$

donde σ es una constante y $dist(p_v^i, p_v^{i-1})$ denota la diferencia en la posición del vértice v en el trazado anterior y el actual.

Para poder animar los trazados, en lugar de pasar directamente del trazado de G_i al de G_{i+1}, usan dos trazados intermedios que facilitan la animación. Sea T_i el trazado de G_i. En total, pasar de T_i a T_{i+1} requiere la creación de T_{i+1}^0, T_{i+1}^1 y $T_{i+1}^2 = T_{i+1}$. Los tres trazados intermedios se crean de la siguiente manera:

- T_{i+1}^0 es obtenido aplicando el algoritmo de trazado estático a G_{i+1} pero fijando las posiciones de todos los vértices de G_{i+1} que también están en G_i. Es decir que los vértices nuevos (que están en G_{i+1} pero no en G_i) son introducidos de manera óptima en T_i

- T^1_{i+1} es obtenido aplicando el algoritmo de trazado dinámico con $\sigma = 0,75$.
- T^2_{i+1} es obtenido aplicando el algoritmo de trazado dinámico con $\sigma = 1,50$.

Una vez que se tienen las tres posiciones, se interpolan las tres con una curva *spline* que define la trayectoria del vértice durante la animación. Para calcular los tres trazados usan *Simulated Annealing*.

Otros trabajos que usan esta misma idea de agregar una fuerza hacia la posición en el trazado anterior son [EHK+04b] y [EHK+04a]. Un enfoque ligeramente distinto es el presentado por Lyons et al. en [LMR98]. Los algoritmos que se proponen tienen dos objetivos principales: mejorar la distribución de los nodos en base a una métrica de distribución y minimizar los movimientos de los vértices. Para esto último utilizan también el agregado de una fuerza que atrae a cada vértice a su posición anterior, pero a diferencia con [BKL+00], usan fuerzas lineales.

7.2.1. Trazado de grafos *online*

El problema de los trazados *online* puede ser visto como un caso particular de grafos dinámicos. El mismo consiste en visualizar subgrafos de grafos muy grandes (que pueden no ser conocidos en su totalidad). La parte dinámica surge de que el usuario va explorando este grafo grande "moviéndose" entre subgrafos "cercanos" (y similares).

Huang et al. proponen en [HCE98] un modelo y un algoritmo de trazado *online*. El modelo establece una forma de navegar, es decir de generar a partir de un subgrafo actual un nuevo subgrafo y su correspondiente trazado. La secuencia de trazados producida contempla tanto la conservación del mapa mental como los criterios estéticos usuales. Ejemplos donde este tipo de navegación puede ser útil es en la visualización de software y en la visualización de grafos de la *World Wide Web*.

La forma de navegar a través de este grafo está basada en *nodos de foco*. Supongamos que en un momento dado el usuario está visualizando cierta parte del grafo. Diremos que el trazado actual está centrado en un conjunto de nodos (que son los que interesan al usuario en este momento) que llamaremos *nodos de foco*.

El trazado actual mostrará a todos los nodos de foco, y para cada uno de éstos se mostrarán también sus vecinos. El resto del grafo será omitido, por lo que esto es equivalente a estar visualizando un subgrafo G_i, conformado por el subgrafo inducido por los nodos de foco más sus vecinos.

En algún momento, el usuario puede querer saber más sobre las conexiones de alguno de los nodos que no son de foco que aparecen en el trazado. Entonces se agregará ese nodo a los nodos de foco, creándose un nuevo subgrafo G_{i+1} que deberá ser visualizado. Reconocemos aquí al problema del trazado dinámico de grafos, ya que G_{i+1} va a diferir de G_i en sólo unos pocos vértices y aristas.

A continuación explicaremos el modelo más formalmente (usaremos la misma notación que en [HCE98]).

Modelo de grafo dinámico

Sea $G = (V, E)$ el grafo que se desea explorar. La exploración de G se lleva a cabo a través de una serie de *cuadros lógicos* F_1,F_2,...,F_i,.... Cada cuadro lógico $F_i = (G_i,\ Q_i)$ consiste de un subgrafo conectado $G_i = (V_i, E_i)$ y una cola Q_i de *nodos de foco*.

Sea $N(v)$ el subgrafo de G inducido por los nodos cuya distancia (teórica) a v es a lo sumo 1, es decir, el conjunto compuesto por v y sus vecinos. El subgrafo G_i asociado a $Q_i = \{v_1, v_2, ..., v_s\}$ es el subgrafo inducido por $N(v_1) \cup N(v_2) \cup ... \cup N(v_s)$. Los nodos $\{v_1, v_2, ..., v_s\}$ son los *nodos de foco* del cuadro lógico $(G_i,\ Q_i)$.

El cuadro lógico F_{i+1} se obtiene a partir de F_i agregando un nodo de foco u y su vecindario $N(u)$, y quitando a lo sumo un nodo de foco y su vecindario. Esto hace que los subgrafos G_{i+1} y G_i difieran en muy pocos elementos. La idea es que es el usuario el que elige enfocarse en un nuevo nodo, seleccionado cuál es el nodo u que quiere agregar. Como consecuencia de esto, si la cola Q_i está llena será necesario quitar un nodo para que u pueda entrar. En su implementación, Huang et al. usan una política *FIFO* (sale el que está en la cabeza de la cola).

Modelo de fuerzas y algoritmo

El algoritmo que se propone en [HCE98] usa tanto animación como minimización de cambios en las posiciones. Detallaremos brevemente el modelo de fuerzas que usan, que tiene como particularidad un tercer tipo de fuerza que evita solapamiento entre vecindarios, que puede ser de utilidad para otras aplicaciones de trazado dinámico.

El modelo está compuesto por tres tipos de fuerzas: las atractivas y repulsivas usuales más una fuerza repulsiva f_g cuyo objetivo es que los vecindarios $N(v)$ no se solapen.

Las fuerzas atractivas f_a siguen la ley de Hooke (son lineales), mientras que las repulsivas f_r y f_g son de la forma $\frac{K}{d_{uv}^3}\overrightarrow{p_u p_v}$. En base a esto, y suponiendo que el cuadro lógico actual es $F_i = (G_i, Q_i)$, la fuerza total $f(u)$ que experimenta un vértice u es

$$f(u) = \sum_{v \in N(u)} f_a(u, v) + \sum_{v \in V_i} f_r(u, v) + \sum_{v \in Q_i} f_g(u, v)$$

Notar que la combinación de las fuerzas f_a y f_g hace que los vértices de un mismo vecindario tiendan a estar juntos y apartados de los otros vecindarios, lo que reduce el solapamiento.

El método que usan para minimizar la energía es el método del gradiente, ya que tiene la ventaja de que produce movimientos suaves de los vértices, apropiados para la animación y para preservar el mapa mental. Además para mejorar la

animación agregan un tope a la distancia que puede moverse un vértice. Para mejorar el tiempo de cómputo ignoran ambas fuerzas repulsivas (f_r y f_g) cuando la distancia entre los vértices es mayor a 100 y 400, respectivamente.

En sus pruebas, Huang et al. observan que las fuerzas f_g tienen otro efecto beneficioso: tienden a alinear los nodos de foco en una recta, lo cual permite identificar fácilmente la dirección en la que se está explorando el grafo.

Una observación final importante es que el modelo de fuerzas usado hace que este algoritmo produzca buenos trazados de árboles, ya que el agregado de las terceras fuerzas elimina el solapamiento que es común al aplicar métodos dirigidos por fuerzas a este tipo de grafos. Recordemos que los árboles son uno de los grafos para los cuales se obtienen peores resultados con estas técnicas, por lo que esta adaptación merece ser considerada para las aplicaciones que necesiten realizar trazados de esta clase de grafos.

En la sección 9.3 presentaremos un algoritmo relacionado, pensado para grafos dinámicos y con *clusters*.

7.3. *Framework* general

Algunos autores han sugerido usar un *framework* general para tratar el problema del trazado de grafos dinámicos. Uno de los más importantes, y que es aplicable a los métodos dirigidos por fuerzas, es el enfoque bayesiano propuesto por Brandes y Wagner ([BW97]). Si bien el framework es independiente del tipo de algoritmo usado, se adapta particularmente bien a los dirigidos por fuerzas.

Si X es el nuevo trazado e Y es el trazado anterior, el objetivo es encontrar un trazado X que maximice

$$P(X = x|Y = y) = \frac{P(Y = y|X = x)P(X = x)}{P(Y = y)}$$

donde $P(X = x)$ es equivalente a la energía de X usando la función de energía estática usual (por ejemplo la energía de DH), y $P(Y = y|X = x)$ representa la diferencia (en energía) entre los trazados X e Y.

Para el caso de un algoritmo del estilo del *spring embedder*, donde hay fuerzas atractivas y repulsivas, el modelo puede formularse de la siguiente manera (usando la notación de [BW97]):

$$P(X = x) = \frac{1}{Z} e^{-\sum_{u \neq v \in V} U_{uv}(x)}$$

donde $U_{uv}(x)$ es un potencial que combina las fuerzas atractivas y repulsivas que afectan al par de vértices (u, v), por ejemplo, usando potenciales similares a los del algoritmo DH:

$$U_{uv}(x) = \begin{cases} \frac{c_1}{d_{uv}^2} + c_2 d_{uv}^2 & \text{si } (u, v) \in E \\ \frac{c_1}{d_{uv}^2} & \text{en caso contrario} \end{cases}$$

Con c_1, c_2 y Z constantes, y d_{uv} la distancia entre u y v en el trazado en cuestión. Notar que $P(X = x)$ tiende a 1 a medida que la energía del trazado X disminuye. Bajo este modelo, trazados de mejor calidad tendrán mayor probabilidad que trazados de mala calidad.

Por otro lado, $P(Y = y|X = x)$ es el componente que modela la estabilidad. Si se usa, como es habitual, la diferencia en las posiciones entre X e Y como medida de la misma, se puede definir:

$$P(Y = y|X = x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\sum_{u \neq v \in V_{XY}} \frac{\left\| p_v^{(Y)} - p_v^{(X)} \right\|_2}{2\sigma^2}}$$

donde V_{XY} son los vértices comunes a los grafos asociados a X e Y, $p_v^{(Y)}$ es la posición del vértice v en el trazado Y y σ es una constante que controla la amplitud de la desviación. Esta expresión supone una distribución normal alrededor de la posición anterior de los vértices. Notar que la expresión $\left\| p_v^{(Y)} - p_v^{(X)} \right\|_2$ es muy semejante a utilizar un resorte de longitud cero que atraiga a cada vértice hacia su posición anterior (en Y).

Notar que este framework separa por completo los potenciales relacionados con la calidad del trazado estático ($P(X = x)$) de los relacionados con la preservación del mapa mental ($P(Y = y|X = x)$).

La ventaja de un formalismo como éste es que permite definir de manera precisa las nociones del trazado dinámico (grafo dinámico, estabilidad, etc.), y al ser suficientemente general, permite llevar distintos algoritmos, aparentemente sin relación, a un mismo nivel teórico donde se los puede comparar y analizar. Por ejemplo, en [BW97] también se muestra cómo aplicar el framework a un algoritmo de trazado ortogonal.

Capítulo 8

Trazados con restricciones

En la mayoría de las aplicaciones donde se usa trazado de grafos existen restricciones que van más allá de los criterios estéticos con los que se quiere mostrar el grafo. Son restricciones inherentes a la semántica del grafo y a información adicional que debe ser visualizada. Vértices que no son puntos en el plano, ejes que deben tener orientaciones particulares y otras restricciones propias de la información representada aparecen a diario cuando se intenta llevar un algoritmo de trazado de grafos a un problema de visualización concreto. Si los algoritmos no son capaces de ser adaptados a estas situaciones de la vida real, su uso se restringiría mucho.

En este capítulo hemos reunido las principales restricciones adicionales con las que los algoritmos de trazado de grafos – dirigidos por fuerzas – deben convivir cuando son llevados a problemas del mundo real.

Para simplificar la presentación, hemos organizado este capítulo según tres tipos de restricciones: en la forma y tamaño de los vértices, en el tipo de las aristas y restricciones generales sobre la posición de los vértices. Al igual que en los otros capítulos, en cada caso presentaremos los problemas más frecuentes y las soluciones más importantes encontradas en la literatura, de manera de que el lector tenga a su alcance un completo equipo de herramientas para abordar, si lo necesita, sus propios problemas de trazado de grafos.

8.1. Vértices con forma y tamaño

El primer caso que consideraremos es el de los vértices con forma y tamaño. Hasta ahora los vértices habían sido considerados puntos en el plano o en el espacio. Si bien esto es muy cómodo para el diseño de los algoritmos, no es lo que suele ocurrir en la práctica, ya que en la mayoría de los casos información adicional, como etiquetas o figuras geométricas son usadas para identificar a los vértices.

Una alternativa poco satisfactoria pero que es encontrada frecuentemente en los paquetes de *software* de visualización de grafos es ignorar el problema por completo. Esto es: suponer que los vértices son puntos, calcular el trazado así y luego reemplazar los puntos por vértices más grandes. Por ejemplo, el grafo de la izquierda de la Figura 8.1 al ser calculado con vértices como puntos obtiene un trazado bastante bueno. Sin embargo, cuando se reemplazan los puntos por la verdadera forma de los vértices (derecha), el resultado empeora notablemente, ya que aparecen dos nuevas situaciones no deseadas:

- Vértices que se solapan.

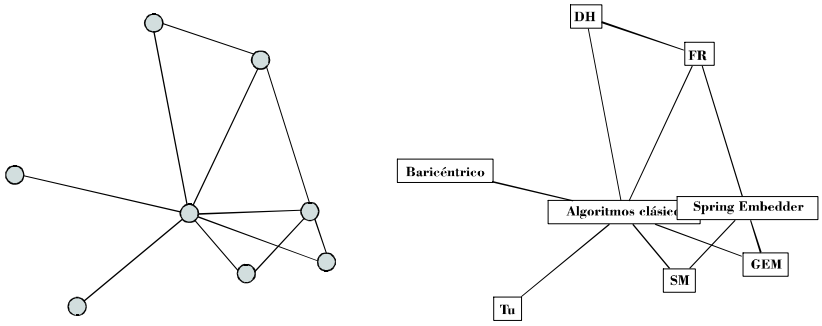


FIGURA 8.1. Vértices con tamaño y forma.

- Aristas que atraviesan vértices.

Para trabajar adecuadamente con vértices con forma y tamaño, es necesario contemplar esto en el modelo y/o en los algoritmos. Debido a su flexibilidad, casi todos los algoritmos dirigidos por fuerzas pueden ser adaptados de alguna manera para considerar este factor.

Como veremos a continuación, todas las soluciones propuestas se ocupan explícitamente del primer problema (solapamiento), pero no del segundo. De manera similar a lo que ocurre con los cruces de aristas, en general al atacar el primer problema, el segundo también se soluciona.

8.1.1. Adaptación de la longitud ideal de las aristas

Es posible considerar los tamaños de los vértices adaptando la longitud ideal de las aristas que los conectan. Lo más fácil es mantener una única longitud ideal de arista K. Si se desea evitar los solapamientos entre todos los vértices, K debe contemplar el tamaño del vértice más grande, por ejemplo $K = 2r_{max} + k$, donde r_{max} es la máxima distancia entre el centro de un vértice y un punto de su borde (a esto lo llamaremos *radio* del vértice), y k es la distancia que se querría mantener entre los vértices si fueran puntos. La distancia entre los vértices se define como la distancia entre sus centros.

Esto funciona bastante bien si todos los vértices tienen aproximadamente igual radio y sus formas son cercanas a discos o esferas. Sin embargo, cuando estos factores cambian el resultado será un trazado con vértices demasiado separados. En la Figura 8.2 se puede ver un posible trazado de una malla con vértices de tamaños no uniformes. El resultado no aprovecha bien el espacio, ubicando los vértices más pequeños a distancias demasiado grandes.

Otra opción menos "extremista" es la adoptada en [BF95], donde para que el algoritmo GEM-3D soporte vértices de distinta forma y tamaño, usan un parámetro de separación mínima entre vértices definido como el tamaño promedio de los mismos (t_{prom}). Este a su vez permite definir la longitud ideal de las aristas como $K = t_{prom}.\deg_{prom}$, donde $\deg_{prom}$ es el grado promedio de los vértices. Al igual que en el caso anterior, su efectividad es muy limitada ya que sigue uniformizando

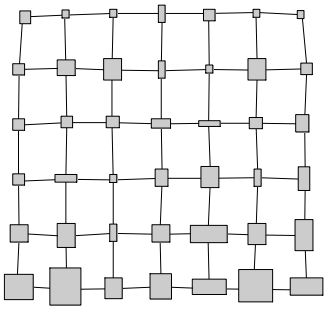


FIGURA 8.2. Calcular la longitud ideal en base al tamaño del vértice más grande.

la longitud de las aristas, lo cual no da buenos resultados si los tamaños de los vértices no son uniformes.

Para cuando los vértices no son uniformes es recomendable utilizar las técnicas que se describen a continuación, que usan una longitud ideal que varía según cada arista, adaptan las fuerzas de repulsión entre vértices y algunas agregan fuerzas específicas para evitar el solapamiento.

8.1.2. Adaptación de las fuerzas

Para poder manejar mejor vértices no uniformes es necesario introducir cambios más elaborados en las fuerzas. Por un lado, la longitud ideal de las aristas debe ser específica a cada una y considerar la forma y tamaño de los dos nodos involucrados. Por otro lado, los cálculos de las distancias entre los vértices – definida como la distancia entre los centros – debe considerar por separado la parte dentro del vértice y la parte afuera. La notación que usaremos de aquí en adelante se ilustra en la Figura 8.3. Cada arista (u, v) tiene tres partes: la parte dentro del vértice u (r_u), la parte dentro del vértice v (r_v) y la parte que propiamente conecta a ambos.

Siguiendo esta línea, Tunkelang [Tun99b] (sección 5.4) propone unas simples modificaciones a su modelo de fuerzas para poder trabajar con vértices convexos de tamaño variable. A continuación mostramos cómo se modifica la magnitud de las fuerzas (la dirección no varía). En esta sección u y v denotarán siempre vértices de G y d_{uv} es la distancia entre los centros de los vértices u y v.

Las fuerzas atractivas pasan de $\frac{d_{uv}^2}{K}$ (con K la longitud ideal de la arista) a

$$\|f_a(u,v)\| = \frac{(d_{uv} - (r_u + r_v))^2}{K + (r_u + r_v)}$$

Es importante señalar que además, si $d_{uv} < (r_u + r_v)$, es decir si hay solapamiento, el sentido de la fuerza f_a es invertido, de manera de repeler a los vértices en lugar de atraerlos.

Las fuerzas repulsivas pasan de $\frac{K^2}{d_{uv}}$ a

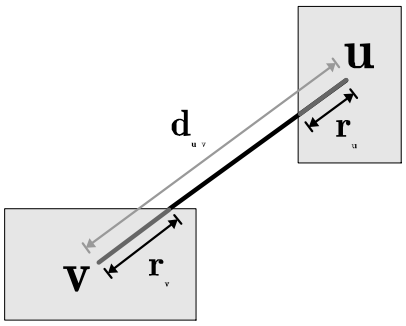


FIGURA 8.3. Radio de los vértices.

$$\|f_r(u,v)\| = \frac{(K+(r_u+r_v))^2}{d_{uv}}.$$

Además también se presenta la adaptación de unas fuerzas repulsivas vértice-arista que pasan de $c_{ve}(\frac{K^2}{d_{ve}} - K)$ a $c_{ve}(\frac{(K+r_v)^2}{d_{ve}} - (K + r_v))$. Donde c_{ve} es una constante, y d_{ve} es la distancia entre el vértice v y la arista e.

Como se puede observar, los cambios son intuitivos y obedecen a las adaptaciones comentadas al comienzo de esta sección. Los vértices pueden tener cualquier forma siempre y cuando los r_v sean calculables de manera eficiente, y la forma sea convexa. Si esto último no se cumple pueden surgir solapamientos, por lo que en caso de estar obligado a usar vértices cóncavos, Tunkelang sugiere calcular los r_v en base a la envolvente convexa del vértice.

Una solución semejante a la de Tunkelang es la propuesta, varios años antes, por Wang y Miyamoto [WM95]. Las fuerzas atractivas y repulsivas de FR (sección 4.2) son adaptadas de siguiente manera:

$$\|f_a(u,v)\| = \begin{cases} 0 & \text{si } u \text{ y } v \text{ se solapan} \\ \frac{(d_{uv}-(r_u+r_v))^2}{K+(r_u+r_v)} & \text{en caso contrario} \end{cases} \tag{8.1}$$

$$\|f_r(u,v)\| = \begin{cases} C\frac{K^2}{d_{uv}} & \text{si } u \text{ y } v \text{ se solapan} \\ \frac{K^2}{d_{uv}} & \text{en caso contrario} \end{cases}$$

Las fuerzas repulsivas incluyen una constante de penalización C que aumenta la fuerza cuando los vértice se solapan, a la vez que dejan de atraerse. Según Wang y Miyamoto esto es suficiente para eliminar el solapamiento. Lo que no comentan es si la introducción de esta posible discontinuidad en la fuerza f_r no lentifica la convergencia del algoritmo de optimización (método del gradiente).

Notar también que las fuerzas atractivas difieren con las de Tunkelang sólo en el caso de un solapamiento. En este caso, Tunkelang es más "agresivo", invirtiendo el signo de la fuerza, mientras que Wang y Miyamoto simplemente anulan la atracción entre vértices. El trazado de la Figura 8.4 fue producido con este algoritmo.

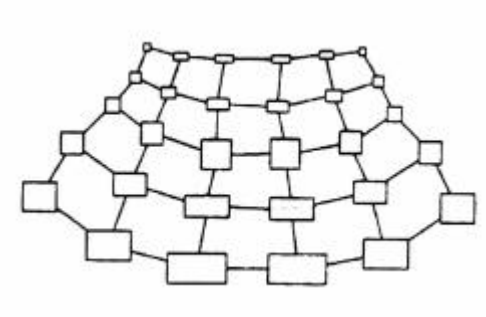


FIGURA 8.4. Resultado de aplicar el algoritmo de Wang y Miyamoto a un grafo con vértices no uniformes.

Una observación final es que las fuerzas 8.1 se anulan cuando $d_{uv} = K+(r_u+r_v)$, lo que constituye la longitud ideal de la arista (u, v).

Otra alternativa usada para evitar los solapamientos es agregar una fuerza de repulsión entre vértices para evitar específicamente este problema. Esto es lo que se hace en [KKR95] para vértices rectangulares e isotéticos, donde se agrega un potencial repulsivo U_s entre todo par de vértices u y v:

$$U_s(u, v) = c_s\alpha_x(u, v)\alpha_y(u, v)$$

donde c_s es una constante, y $\alpha_x(u, v)$, $\alpha_y(u, v)$ miden el solapamiento en x y en y. Si el vértice u es el rectángulo definido por las rectas $x = x_u^I$, $x = x_u^D$, $y = y_u^I$, $y = y_u^D$, $\alpha_x(u, v)$ se define como

$$\alpha_x(u, v) = \begin{cases} \text{máx}\{x_u^D - x_u^I, 0\} & \text{si} \quad x_u^D + x_u^I \leq x_v^D + x_v^I \\ \text{máx}\{x_v^D - x_v^I, 0\} & \text{en caso contrario} \end{cases}$$

La expresión de $\alpha_y(u, v)$ es análoga. En base al potencial U_s calculan la fuerza f_s que usan para la optimización como $f_s(u, v) = (\frac{\partial U_s(u,v)}{\partial x_u}, \frac{\partial U_s(u,v)}{\partial y_u})^t$. Notar que $U_s(u, v)$ es al menos tan grande como el área del solapamiento entre u y v, y 0 si no hay superposición.

La ventaja de este enfoque es que permite mantener intactas las fuerzas atractivas y repulsivas, y sólo requiere agregar las fuerzas f_s, aunque como siempre, para obtener buenos resultados los pesos relativos de cada fuerza deberán ser ajustados adecuadamente.

Spring Embedder elíptico. Si los vértices del grafo pueden ser aproximados por elipses (por ejemplo si son rectángulos) entonces puede usarse el algoritmo ESM (por *elliptic spring method*) de Harel y Koren [HK02a]. El método es una generalización del modelo de fuerzas de FR, pero para vértices elípticos.

Las fuerzas atractivas y repulsivas se explican a continuación. En ambos casos los vértices se consideran elipses, con centro $p_u = (x_u, y_u)$ y radios (r_u^x, r_u^y) para todo $u \in V$. K es la longitud ideal de las aristas.

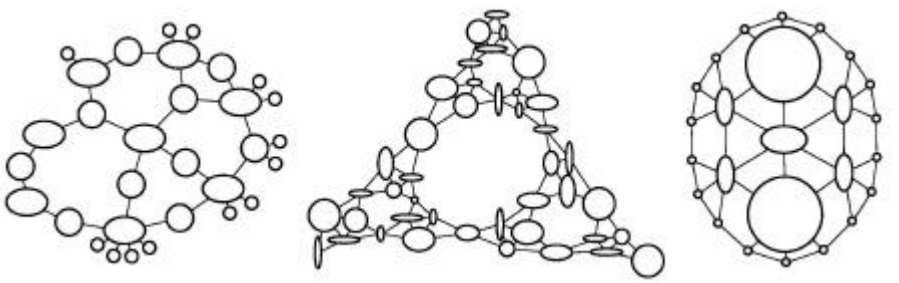


FIGURA 8.5. Resultados obtenidos con el *spring embedder* elíptico [HK02a].

Magnitud de las fuerzas atractivas (para $(u, v) \in E$):

$$\|f_a(u,v)\| = \frac{(x_u - x_v)^2}{(r_u^x + r_v^x + K)^2} + \frac{(y_u - y_v)^2}{(r_u^y + r_v^y + K)^2}$$

Este término puede interpretarse intuitivamente como que se quiere evitar que el centro de v (x_v, y_v) esté dentro de la elipse con igual centro que u, (x_v, y_v), y con radios $(r_u^x + r_v^x + K, r_u^y + r_v^y + K)$.

Magnitud de las fuerzas repulsivas (para todo $u, v \in V$):

$$\|f_r(u,v)\| = \omega \left(\frac{(x_u - x_v)^2}{(r_u^x + r_v^x + K)^2} + \frac{(y_u - y_v)^2}{(r_u^y + r_v^y + K)^2} \right)^{-1}$$

donde ω es una constante (los autores usan $\omega = 1$).

El efecto de estas fuerzas es que si dos elipses se solapan, f_r va a ser muy grande mientras que f_a será chica. Si los vértices están alejados, el efecto será el contrario.

Lo que hacen estas fuerzas es contemplar la forma de las elipses en el cálculo de las distancias (y en la longitud ideal de las aristas). Es la misma idea de los algoritmos anteriores, pero especializada en elipses. La ventaja de usar elipses es que permiten aproximar bastante bien muchas formas comunes para vértices, como rectángulos (¡y elipses!), y al mismo tiempo mantiene los cálculos sencillos. La figura 8.5 muestra algunos resultados obtenidos con este algoritmo.

Observación sobre la velocidad de convergencia. Los autores del algoritmo ESM presentan en [HK02a] otro algoritmo, también basado en el *Spring Embedder* – llamado MSM – que se asemeja mucho a los de Wang y Tunkelang. Harel y Koren notaron que la velocidad de convergencia de sus dos algoritmos (el ESM y el MSM) es muy lenta, y lo atribuyen a la inclusión de términos que evitan el solapamiento de vértices.

La observación, razonable y al mismo tiempo de gran importancia, es que cuando los vértices ocupan un porcentaje importante del espacio disponible, hay menos espacio para maniobrar en busca de un buen trazado. Esto repercute de dos formas: por un lado decrece la velocidad de convergencia, y por el otro crece

la probabilidad de caer en un mínimo local pobre, ya que debido a su tamaño, es mucho más fácil que los vértices queden bloqueados sin poder mejorar su situación.

Para sobrellevar este inconveniente sugieren aumentar gradualmente el peso de las fuerzas que evitan los solapamientos. De esta manera permiten que la estructura general del grafo se obtenga con mayor facilidad, durante las primeras iteraciones.

8.1.3. Adaptación de KK

Hasta ahora hemos visto sólo adaptaciones de métodos derivados del *spring embedder*. El primer trabajo encontrado que adapta otro algoritmo clásico es [HK02a], donde Harel y Koren proponen una elegante generalización de KK que soporta vértices no uniformes.

Recordemos que KK (sección 4.4) se basa en las distancias teóricas entre los vértices para establecer las longitudes ideales. El método IKKM (*Iterative Kamada-Kawai Method*) de Harel y Koren define en base al grafo original $G(V, E)$ un nuevo grafo con pesos $G^X(V, E, w)$, con función de peso $w(u, v) = K + r_u + r_v$ (mantenemos la notación del comienzo de esta sección). A su vez definen sobre esto una métrica de distancia entre vértices $d^X(u, v)$ definida como la distancia teórica entre u y v en el grafo G^X.

El objetivo de IKKM es minimizar la siguiente función de energía:

$$E = \sum_{u,v \in V} k_{uv}^X (d_{uv} - d^X(u, v))^2 \tag{8.2}$$

donde d_{uv} es la distancia Euclídea entre p_u y p_v, y $k_{uv}^X \approx d^X(u, v)^{-2}$ es un factor de normalización.

La función de energía (8.2) es muy parecida a la de KK (4.11), utilizando la función de distancia d^X, que no es otra cosa que una generalización de la distancia teórica para contemplar que los vértices tienen tamaño no nulo.

El método de optimización propuesto es el mismo de KK, Newton-Raphson de a un vértice por vez. La convergencia es muy rápida, según [HK02a] un par de iteraciones son suficientes. La calidad de los trazados resultantes parece ser buena excepto en los solapamientos, ya que el IKKM parece ser más propenso a ocasionarlos que el ESM o el MSM. Esto es atribuido a que las fuerzas repulsivas son más bien débiles (son lineales) y a que IKKM hace más énfasis en el aspecto global del trazado que los basados en el *spring embedder*, descuidando los aspectos más locales como el solapamiento de vértices. Para subsanar esto proponen un algoritmo que llaman "combinado", que primero aplica IKKM y luego realiza una etapa de *fine-tuning* para eliminar el solapamiento. Para este segundo paso puede usarse alguno de los otros métodos que presentan, como el ESM o el MSM.

En [HK02a], Harel y Koren llevan esta idea de generalizar un paso más adelante, y muestran que con la misma idea que usan para adaptar KK, todo problema de trazado con vértices no uniformes puede llevarse al trazado de un grafo con vértices con pesos (que es lo mismo que tener vértices que son círculos de radio variable). Más detalles pueden encontrarse en [HK02a].

8.1.4. Una solución en tres etapas

Una forma muy distinta a las anteriores de encarar el problema es la que presentan Gasner y North en [GN98]. Su solución consta de tres etapas independientes:

1. Calcular el trazado de G con algún algoritmo ignorando el tamaño de los vértices (en su implementación, Gasner y North usan KK).

2. Arreglar el solapamiento de vértices.

3. Dibujar las aristas como curvas.

El primer paso es independiente del algoritmo que se use, cualquiera es válido.

La segunda etapa es la más interesante: cómo ordenar el trazado del paso 1) para eliminar todos los solapamientos que sea posible. El objetivo es mover gradualmente los vértices hasta que haya un espacio adecuado que los separe.

Para esto usan una variante de la técnica de Lyons para distribuir vértices uniformemente ([LMR98]). El método consiste en construir repetidamente un diagrama de Voronoi en una ventana fija usando como sitios los centros de los vértices y moviendo cada vértice al centro de su celda en el diagrama. Eventualmente la ventana quedará chica y deberá ser agrandada para que los vértices puedan seguir moviéndose. Este proceso se repite hasta que todos los solapamientos son reducidos.

Sobre cómo realizar la tercer etapa, que dibuja curvas en lugar de ejes rectos, hablaremos más adelante en este mismo capítulo.

Lo más original de este método es que plantea una forma distinta de encarar el problema de los vértices no uniformes: como una etapa de *postproceso* al cómputo del trazado.

Como se observa en [HK02a], este método basado en el diagrama de Voronoi no tiene en cuenta la forma de los vértices, así que no suele obtener trazados tan compactos como los que sí la consideran. Los resultados que se obtienen con los otros métodos, que incluyen el tamaño de los vértices en el modelo mismo, generalmente son mejores, pero igualmente este enfoque merece ser tenido en cuenta.

8.1.5. Uso de campos potenciales

Recientemente se propuso una técnica de trazado de grafos para vértices no uniformes basada en campos potenciales. Los campos potenciales ya han sido aplicados a otros problemas indirectamente relacionados, como la planificación de trayectorias libres de obstáculos (ver por ej. [Kha86]). La técnica propuesta por Chuang et al. en [CLY04] usa fuerzas atractivas logarítmicas para vértices conectados por aristas, como las del *spring embedder*, y en lugar de usar fuerzas repulsivas utiliza campos potenciales.

La idea consiste en que los vértices, representados por polígonos, tienen en sus bordes una carga que repele a los otros vértices, lo que conforma el campo potencial del vértice. De esta manera, las fuerzas que afectan a un vértice dado

son dos: las atractivas, producto de sus aristas, y las repulsivas, causadas por los campos potenciales de los otros nodos.

En la propuesta de Chuang et al., cada lado del polígono correspondiente a un vértice ejerce una fuerza que repele a los otros lados de los otros vértices. Además de ser desplazados por estos campos potenciales, los vértices también pueden rotar como producto del efecto de estas fuerzas.

Si bien en [CLY04] no se dan muchos detalles de cómo implementar eficientemente este algoritmo, ni de los resultados que se pueden obtener, el enfoque parece prometedor, sobre todo por el hecho de combinar métodos de trazado de grafos con técnicas de planificación de trayectorias.

8.2. Aristas especiales

También las aristas pueden estar sujetas a restricciones. En esta sección nos ocuparemos de tres situaciones que pueden surgir en relación con las aristas. En la primera las aristas del grafo tienen pesos y se quiere que esos pesos se vean reflejados en el trazado. En la segunda el grafo es dirigido y el trazado debe mostrar los arcos con alguna orientación determinada. Finalmente, en la tercera las aristas son curvas en lugar de ser segmentos de recta. Si bien esta no suele ser una restricción sino más bien una decisión de diseño, la incluimos en esta sección.

8.2.1. Grafos con pesos

En muchas aplicaciones el grafo que se desea trazar tiene pesos asignados a las aristas. Por ejemplo, para visualizar temas en común entre artículos científicos, en [EHK+04b] se usa un grafo en el que los vértices son artículos y las aristas conectan artículos con temas en común, y el peso de cada arista es función de la cantidad de temas que comparten. La intención es que cuantos más temas comparten, más cerca se dibujen los vértices.

Las adaptaciones necesarias para contemplar los pesos de los ejes son mínimas. El objetivo es que cuanto mayor sea el peso de la arista, mayor sea su longitud en el trazado.

El algoritmo KK no requiere adaptaciones ya que las distancias teóricas contemplan por definición los pesos de los ejes. Los algoritmos derivados del *spring embedder* sólo necesitan mantener una longitud ideal de arista por separado, definida en función del peso de la misma.

8.2.2. Grafos dirigidos

Muchos de los grafos que aparecen en las aplicaciones son dirigidos. Sin ir más lejos, los grafos que modelan la estructura de sitios web, que hemos mencionado en varias ocasiones, son dirigidos porque los enlaces entre páginas web tienen una única dirección.

Cuando los grafos son dirigidos es frecuente que se prefiera dibujar todos los arcos en una misma dirección, por ejemplo con todos los arcos apuntando hacia

abajo, debido a que esto ayuda a la comprensión de la información. Esto requiere adaptaciones a los algoritmos de trazado.

Como se comentó en la sección 4.6, el método de los resortes magnéticos SM de Sugiyama y Misue se ajusta perfecto a este objetivo si se usa un campo magnético vertical y aristas magnetizadas unidireccionalmente. A cualquier otro método que use fuerzas, como el *spring embedder*, se le puede agregar fuerzas similares a las rotativas de SM para lograr este efecto. Una solución de ese estilo es usada con buenos resultados en [Tun99a]. El único problema que el autor dice haber tenido es con los grafos con ciclos dirigidos; pero una simple etapa de pre-proceso, que invierte el sentido de uno de los arcos de cada ciclo dirigido, permite superar el inconveniente.

Los algoritmos que usen una función general de energía, como DH, deberán agregar un potencial que penalice los ejes que no están alineados correctamente. Uno posible, usado en [BKL+00] para que aristas en el espacio (3D) apunten hacia abajo, es

$$U_{rot}(u,v) = c_{rot} K \left(\frac{\arccos \frac{z_u - z_v}{d_{uv}}}{\frac{\pi}{2}} \right)^2 \tag{8.3}$$

Con c_{rot} constante, K la longitud ideal de la arista y z_u las coordenada z de p_u.

Este potencial se acerca a cero únicamente cuando p_u y p_v son casi verticales, que es lo que se quiere promover. Si los vértices no están en el espacio sino en el plano, alcanza con cambiar en 8.3 $z_u - z_v$ por $y_u - y_v$.

Estas simples medidas son suficientes, en la mayoría de los casos, para obtener buenos resultados para grafos dirigidos. Por supuesto será necesario ajustar las constantes con cuidado para darle el peso adecuado a este nuevo criterio estético. El uso de restricciones en las posiciones de los vértices (del cual nos ocuparemos más adelante) también puede ayudar a trazar grafos dirigidos.

8.2.3. Aristas curvas

En una amplia mayoría de los trabajos sobre trazado de grafos dirigido por fuerzas se supone que las aristas del grafo son dibujadas como segmentos, por lo que el problema del trazado de grafos se reduce a encontrar una posición para cada vértice.

El uso de curvas para dibujar las aristas permite tener más flexibilidad a la hora de buscar un trazado estéticamente bueno. Si bien en el campo del trazado ortogonal se comprobó que curvas con muchos codos no ayudan a la visualización [PCA02], curvas sencillas pueden ser de gran ayuda. Permiten evitar que las aristas pasen por encima de vértices y aumentan la resolución angular (criterio estético muy importante). En la Figura 8.6 podemos ver un ejemplo.

Las ventajas de usar curvas para las aristas no han sido exploradas en profundidad, y tampoco existen muchos trabajos al respecto. Uno de los primeros en proponer usar curvas para grafos generales es [GN98], en un intento de hacer más claro el trazado de grafos con vértices no uniformes (ver sección 8.1.4), sin embargo

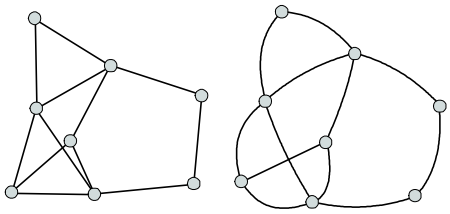


FIGURA 8.6. Dos trazados del mismo grafo, uno con aristas rectas (izq.) y el otro usando curvas (der.).

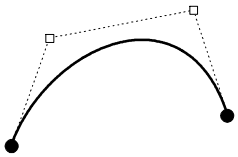


FIGURA 8.7. Las curvas cúbicas de Bézier quedan definidas por cuatro puntos.

esto se hacía en una etapa posterior al cómputo del trazado, independientemente del mismo.

Es posible integrar un método dirigido por fuerzas para que el trazado se calcule directamente usando curvas. La clave para esto es trabajar con los puntos de control que definen las curvas.

En pocas palabras, el método es el siguiente. Supongamos que las aristas de G se quieren dibujar como curvas cúbicas de Bézier. Las mismas están definidas por cuatro puntos: los dos extremos (que serían los dos vértices de la arista) más otros dos intermedios (Figura 8.7)[1]. Así, para la arista (u, v) se usará una curva definida por cuatro puntos $\{p_u, p_v, p^1_{uv}, p^2_{uv}\}$ (p^1_{uv} y p^2_{uv} son los puntos intermedios).

En base a G se arma un grafo auxiliar G' donde cada arista (u, v) es reemplazada por tres aristas (u,i^1_{uv}) $(i^1_{uv}, i^2_{uv},)$ y (i^2_{uv}, v) (donde i^1_{uv} y i^2_{uv} son nuevos vértices, con posiciones p^1_{uv} y p^2_{uv}). Con algún algoritmo dirigido por fuerzas (con algunas modificaciones) se busca un trazado de G'. Finalmente se dibuja cada curva en base a los cuatro puntos que la definen.

En síntesis, la idea general es reducir el problema de dibujar con aristas curvas G a dibujar con aristas rectas un grafo auxiliar G'. Detalles de cómo implementar esto se explican a continuación.

En [BW00], Brandes y Wagner usan curvas cúbicas de Bézier para dibujar conexiones entre estaciones de tren. La esencia es la ya descripta, daremos ahora algunos detalles más sobre G' y las fuerzas que deben agregarse.

Los dos puntos de control de la curva correspondiente a (u, v) se ubican inicialmente en la recta que une p_u y p_v, a igual distancia de los extremos y del otro

[1]Muchos tipos de curvas son posibles. Las de Bézier tienen varias propiedades que las hacen adecuadas para dibujar aristas. Para más detalles sobre este tipo de curvas, recomendamos [FvDFH96].

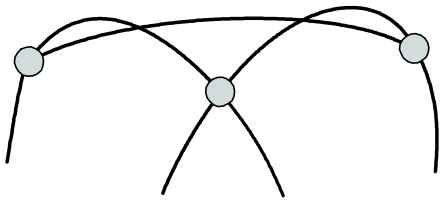


FIGURA 8.8. Un problema propio de usar curvas: cruces entre aristas incidentes a un mismo vértice [Fin03].

punto de control.

La distancia ideal entre los puntos de control de una misma curva y éstos con los extremos se define como $\lambda(d_{uv}/3)$ (para λ constante). Las fuerzas repulsivas para los puntos de control se calculan sólo entre los vértices cercanos. Además, debido a requisitos propios de la información que visualizan, también agregan algunas restricciones respecto a las curvas que salen de un mismo extremo. Más detalles se encuentran en [BW00].

Finkel [Fin03] también propone extensiones similares a las de [BW00] para trabajar con curvas en métodos dirigidos por fuerzas. A diferencia de [BW00], Finkel también usa curvas de Bézier cuadráticas, que quedan definidas por los dos extremos y un punto de control. Por lo tanto, en ese caso, el grafo auxiliar G' tendrá un nuevo vértice y dos nuevas aristas por cada arista de G.

Una observación importante hecha en [Fin03] es que la aplicación directa del algoritmo de trazado en este grafo puede ocasionar muchos cruces entre aristas con un vértice en común – algo propio de usar aristas curvas –, como se puede ver en la Figura 8.8. Esto ocurriría tanto con curvas cuadráticas como cúbicas.

Afortunadamente, este tipo de cruces es solucionable. En [Fin03] se propone una heurística que parece funcionar muy bien para curvas cúbicas. Consiste en una vez obtenido el trazado del grafo, que puede presentar este tipo de cruces, para cada vértice v (en G), conectar entre sí los puntos de control más cercanos alrededor de v, siguiendo el orden radial que deberían tener si las aristas fueran segmentos. Esto formará un "anillo" de aristas alrededor de v. Luego se vuelve a aplicar el algoritmo de trazado, partiendo de las posiciones anteriores pero con estas nuevas aristas, que es de esperar hagan que los cruces desaparezcan.

Las extensiones aquí propuestas para trabajar con aristas curvas dan, en general, buenos resultados en los algoritmos derivados del *spring embedder* y, es de esperar, también en DH y similares. Sin embargo, una adaptación de KK requerirá más trabajo, ya que al usar las distancias teóricas del grafo, la adición de vértices para los puntos de control repercute en las distancias y hace que los resultados no sean buenos [Fin03].

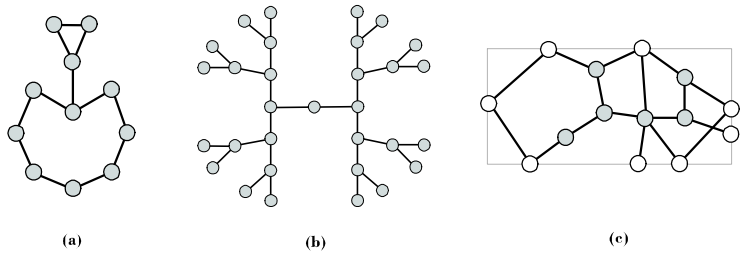


FIGURA 8.9. Trazados obtenidos restringiendo la posición de los vértices [HM98].

8.3. Restricciones en la posición de los vértices

Otro tipo de restricción que suele ser necesario en las aplicaciones son las que conciernen a la posición de los vértices. Vértices que deben permanecer fijos, grupos que deben moverse juntos y vértices que deben mantenerse en una región como una línea o un círculo están entre las más usadas.

La Figura 8.9 muestra tres ejemplos de trazados obtenidos restringiendo las posiciones de los vértices, tomados de [HM98]. En el (a) se especificó un mínimo para la distancia vertical entre los vértices. En el (b) posiciones absolutas para alinear sobre una recta vertical los vértices centrales. Finalmente, el (c) restringe la posición de los vértices blancos para que estén sobre un rectángulo.

En términos generales, es sencillo incluir este tipo de restricciones debido a que muchas veces es suficiente considerarlas sólo al mover los vértices. Por ejemplo, supongamos que usamos una técnica que optimiza con el método del gradiente. Cuando en una iteración es el turno de mover el vértice v, podemos ver si v está sujeto a algún tipo de restricción:

- Si v está fijo, no debe ser movido.
- Si v debe estar dentro de una región, podemos moverlo siempre que la nueva posición esté también dentro.
- Si v debe mantener una distancia fija con otros vértices, podemos mover v junto a los otros vértices.

Estos son sólo algunos ejemplos muy simples, pero ilustran la idea. A continuación repasaremos con más detalle las distintas soluciones encontradas en la literatura. Algunas ya fueron vistas indirectamente. En particular, el algoritmo de resortes magnéticos SM (sección 4.6) sirve para imponer restricciones para alinear los vértices a rectas, círculos y otros campos magnéticos que se puedan construir.

Es importante distinguir entre los algoritmos que garantizan que se cumplan las restricciones y los que sólo intentan cumplirlas. Los que se basan en fuerzas, como SM, entran en este segundo grupo. Por ejemplo, usando SM los vértices van a quedar "casi" alineados, pero no exactamente alineados. Otros métodos, en cambio, como la simple idea de no mover un vértice si se sale de la región

en la que debe permanecer, garantizan que en todo momento las restricciones se cumplen. Cuál de los dos usar depende de la aplicación. A veces cumplimientos "aproximados" de las restricciones son aceptables, y otras veces no.

8.3.1. Restricciones usando fuerzas

Uno de los primeros trabajos en tratar la inclusión de varios tipos de restricciones en algoritmos dirigidos por fuerzas fue [KKR95].

Kamps et al. presentan un algoritmo que soporta cuatro tipos de restricciones:

- **Vértices fijos**. Ambas coordenadas de la posición de los vértices pueden ser fijas, al mismo tiempo o por separado. La forma de implementar esto es la directa, ya comentada.

- **Distancias fijas**. La distancia entre ciertos pares de nodos puede estar fija, tanto en la coordenada x, como en la y (o en ambas). En la práctica no usan distancias fijas sino desplazamientos fijos (con signo, por ej. una restricción podría ser $x_u = x_v - 10$), y se implementan de la manera directa ya comentada.

- **Posición relativa**. Un vértice puede estar restringido a estar debajo o arriba de otro, o a su izquierda o derecha (por ej. $x_u \leq x_v$). En [KKR95] no queda claro cómo implementan esto. La opción más sencilla es mover v siempre que no viole ninguna restricción, y si un movimiento viola, por ej. $x_u \leq x_v$, mover también u de manera de que la desigualdad se siga manteniendo.

- **Orientación**. Dos vértices (u, v) pueden estar forzados a tener la misma orientación horizontal o vertical. Para intentar cumplir esto agregan un potencial que es igual al ángulo entre la recta formada por (p_u, p_v) y el eje x o el eje y (según si la orientación es horizontal o vertical).

Es interesante mencionar que la optimización, que se hace con el método del gradiente, incluye una temperatura local que de manera similar al algoritmo GEM (sección 4.3) disminuye al detectar oscilaciones, aumentando la velocidad de convergencia.

Otros tipos de fuerzas, que restringen los vértices a distintas áreas, son fáciles de incorporar. En [ET03], por ejemplo, para detectar intrusiones en redes, se usan varios tipos de fuerzas "gravitatorias" que distribuyen los vértices en distintas áreas del trazado, según su significado.

8.3.2. Restricciones usando una función general de energía

Cuando el método de optimización consiste en minimizar una función arbitraria de energía, potencialmente discreta, como en el caso de DH (sección 4.5) o de los algoritmo genéticos (sección 5.6.1) aparecen nuevas formas de introducir restricciones.

Por empezar, las ideas de la sección anterior pueden seguir siendo útiles. Por ejemplo, en DH se pueden implementar sin problemas. Más aun, se puede hacer algo más teniendo en cuenta que las posiciones en *Simulated Annealing* se eligen al azar: es posible elegir al azar de entre las posiciones factibles (que no violan ninguna restricción). Esto es lo que hacen Brandes et al. [BKW03] cuando quieren visualizar un tipo de red social donde los vértices deben estar ubicados sobre círculos prefijados. Como su método de optimización es *Simulated Annealing*, la inclusión de este tipo de restricciones sólo requiere hacer que en lugar de elegir una posición al azar, se elija un ángulo al azar para moverse sobre el círculo que contiene al vértice. De esta manera, "por construcción" todas las posiciones candidatas son factibles, y no es necesario ninguna otra extensión.

No siempre será posible o sencillo restringir desde el comienzo los movimientos. Si se usan algoritmos genéticos, por ejemplo, no siempre es tan evidente cómo incorporar las restricciones. Los métodos que se mueven en base a una función arbitraria de energía pueden incorporar restricciones a través de potenciales que penalicen el no cumplimiento de las mismas.

Cualquier restricción sobre la que se pueda medir cuánto es violada por un trazado, puede ser considerada aumentando la función de energía con un potencial por cada una. De hecho esto ya se ha usado en la sección 8.1.2 para evitar el solapamiento de vértices.

Este enfoque tiene dos problemas. El primero es que puede que no todas las restricciones se cumplan; recordemos que se busca minimizar una función con muchos potenciales, y no siempre los de las restricciones llegarán exactamente a su mínimo. Esto es análogo a lo que ocurre al modelar restricciones con fuerzas.

El segundo, y más importante, es que cada potencial debe ir acompañado por su correspondiente peso. Un peso demasiado alto hará que se descuiden los criterios estéticos y uno demasiado bajo hará que las restricciones no se cumplan.

Este problema de cómo elegir los pesos abarca mucho más que el trazado de grafos, y ha sido estudiado desde diversas áreas. Un algoritmo – el único encontrado – que usa técnicas bastante recientes de optimización global y las aplica al trazado de grafos es el de Hansen et al [HMMS02]. El método busca de manera dinámica los pesos para cada potencial y lo hace al mismo tiempo que busca el trazado.

El método que aplican es *Constrained Simulated Annealing* (CSA), presentado por Wah y Wang en [WW99]. CSA está inspirado en los métodos Lagrangianos. A grandes rasgos, la diferencia con *Simulated Annealing* es que el espacio de búsqueda es extendido con nuevas dimensiones: un vector con los posibles pesos de cada potencial. El método no sólo hace descensos en el espacio de la función objetivo, sino que también intercala ascensos en el espacio de los pesos. Para el lector interesado en los detalles recomendamos ver [HMMS02] y [WW99].

En [HMMS02] no hay muchos resultados que justifiquen el uso de este método, que tiene un costo bastante mayor a los otros aquí descriptos [HMMS02]. De todas maneras es importante su consideración porque brinda una flexibilidad muy grande (todo tipo de restricción puede ser incluida) y se apoya en técnicas sólidas y estudiadas de optimización global.

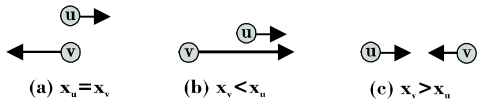


FIGURA 8.10. Ejemplos de barreras que surgen al agregar restricciones.

8.3.3. Otras formas de implementarlas

Otro de los primeros trabajos que combinaron métodos dirigidos por fuerzas con restricciones fue el de Wang y Miyamoto [WM95]. Las restricciones soportadas son similares a las de Kamps [KKR95], permitiendo establecer posiciones absolutas, posiciones relativas (un vértice en relación con otros) y restricciones de *clusters*, que permiten que un conjunto de vértices sea tratado como uno único.

A diferencia de los otros métodos vistos, aquí un algoritmo dirigido por fuerzas basado en FR es alternado con un *solver* de restricciones. Luego de cada iteración principal del algoritmo de trazado, el *solver* mueve los vértices que no cumplen restricciones para que la solución pase a ser factible. El problema que aparece al hacer esto es que la calidad del trazado puede no progresar debido a que un vértice está bloqueado por una restricción que depende de otro. Más específicamente, un vértice u puede ser una "barrera" para v si una restricción entre u y v hace que v no pueda ocupar su posición óptima en el trazado. En la Figura 8.10 se ilustran tres situaciones posibles. La flecha negra indica la dirección en la que los vértices deberían moverse (según las fuerzas), que violan las restricciones mostradas en cada caso.

Para aminorar las consecuencias de esto, definen "varas rígidas" (*rigid sticks*) que fuerzan a los dos vértices involucrados a moverse juntos. Si u pasa a ser una barrera para v (antes de que se viole la restricción), se introduce una de estas varas entre ambos, de manera de que se muevan juntos, como si fueran un objeto rígido. Las fuerzas que afectan a este nuevo objeto son un promedio de las que afectan a cada vértice por separado.

En resumen, cada iteración del algoritmo de Wang y Miyamoto consiste en:

1. Calcular fuerzas.
2. Identificar barreras e introducir varas rígidas.
3. Calcular nuevas posiciones en base a 1) y 2)

Restricciones lineales arbitrarias también pueden ser incorporadas si el problema es abordado con técnicas más elaboradas de optimización no lineal, como en el algoritmo propuesto por He y Marriot [HM98]. El problema de trazado de grafos con restricciones es reducido a minimizar una función objetivo no lineal con restricciones lineales, donde la función objetivo debe capturar los criterios estéticos. En [HM98] se prueban tres funciones distintas: la función de energía de KK (sección 4.4), una aproximación polinomial de la misma y una combinación de ambas. La ventaja de la aproximación es que es más fácil de calcular y, a diferencia de la

primera, no tiene puntos donde se indefinan las derivadas, por lo que el método de optimización converge más fácilmente.

La técnica para resolver el problema de optimización se basa en el método de las restricciones activas, técnica iterativa muy usada y con buenas propiedades de robustez y velocidad. Los lectores interesados en los detalles pueden consultar [HM98] y cualquier libro de optimización no lineal (por ej. [Fri94]).

Según [HM98], el uso de la función objetivo aproximada permite alcanzar trazados de calidad comparable a la de KK y con una velocidad ligeramente menor, lo cual no es poco dado que se está resolviendo el problema con las restricciones. Además de permitir restricciones lineales arbitrarias, también permiten que cada vértice tenga una posición "sugerida". Esto se integra con el algoritmo tomando como trazado inicial el que más cerca a está al sugerido que respeta las restricciones (para esto resuelven previamente un problema cuadrático).

Otro problema relacionado con restricciones, sobre el cual se ocupa la segunda parte de este trabajo, es el que surge de buscar trazados donde cada vértice del grafo representa (y está restringido) a una región distinta. Como se verá en la parte 2, si bien parece un problema fácilmente solucionable con las técnicas descriptas en esta sección, esconde algunas particularidades que deben ser analizadas y resueltas.

Capítulo 9

Trazado de grafos con clusters

Los grafos que surgen en las aplicaciones de visualización crecen cada día más, llegando a tener miles o incluso millones de vértices. Los métodos dirigidos por fuerzas clásicos, como ya se ha mencionado, tienen una complejidad de $O(n^3)$ que hace que no sean aplicables a instancias tan grandes. En este capítulo y en el capítulo 11 veremos técnicas que permiten trabajar con grafos más grandes.

En este capítulo nos ocuparemos de cómo trazar grafos que contienen *clusters* (o "agrupamientos" de vértices). Grupos de nodos relacionados, de alguna manera, pueden ser agrupados en "super-nodos", y las aristas entre esos nodos pasar a ser "super-aristas", obteniendo una simplificación del grafo original con menos vértices y aristas. El objetivo de usar *clusters* para poder lidiar con grafos grandes es simplificar el grafo hasta obtener uno cuyo tamaño esté dentro de lo manejable por los algoritmos disponibles.

En el contexto de un sistema de visualización, el usuario comenzará viendo esta versión resumida del grafo, pero podrá luego seleccionar uno de estos super-nodos para ver en más detalle cómo está conformado. Recién cuando el usuario quiera profundizar en ese super-nodo, se calculará su trazado, y así se irá navegando y trazando el grafo por partes.

Otra consecuencia importante de trabajar con *clusters* es que da lugar a algoritmos de tipo "divide y vencerás", que suelen ser más eficientes que los tradicionales.

Además de proveer una forma para visualizar grafos grandes, los grafos con *clusters* surgen de manera natural en muchas aplicaciones, como lingüística, estadística y visualización de software [KW01],[FT04], y en general, como una herramienta más para la visualización [WB99].

Más formalmente, un grafo con *clusters* se define como un grafo $G = (V, E)$ con una partición $(C_1, ..., C_k)$ del conjunto de vértices V. Cada C_i es llamado *cluster*. Supondremos que todo vértice pertenece a exactamente un *cluster* (que contiene al menos a ese vértice). Como indica esta definición, en principio se asumirá que hay un único nivel de agrupamiento, pero como veremos más adelante, es posible tener toda una jerarquía de agrupamiento de vértices.

Para los algoritmos que veremos a continuación supondremos que el grafo es un grafo con *clusters*, es decir que los mismos ya vienen definidos junto con el grafo. El objetivo será que el trazado refleje lo mejor posible esta estructura: que los nodos estén agrupados correctamente y que se minimicen los posibles solapamientos entre *clusters*. Al final de este capítulo comentaremos algunos algoritmos que permiten visualizar *clusters* sin tenerlos de antemano.

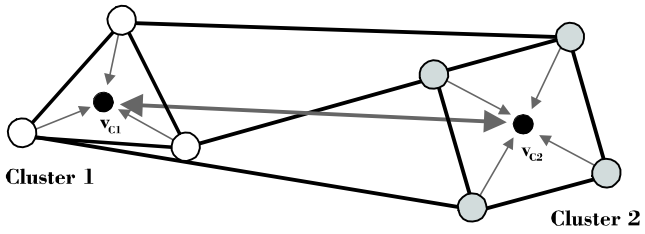


FIGURA 9.1. Agregar vértices atractores para trabajar con *clusters*.

9.1. Agregar vértices atractores

La opción más sencilla para lograr que el trazado muestre la estructura de *clusters* del grafo es agregar para cada *cluster* un vértice atractor. Cada uno de estos vértices representará al *cluster* y además será responsable de mantener juntos a sus vértices. Para lograr esto se agregan fuerzas que involucran a los vértices atractores. La idea se ilustra en la Figura 9.1, y puede resumirse en tres etapas:

1. Por cada cluster C_i, **agregar** un **vértice atractor** v_{C_i}.
2. **Agregar fuerzas atractivas** entre el atractor v_{C_i} y los vértices en el *cluster* C_i.
3. **Agregar fuerzas repulsivas** entre todos los pares de vértices atractores.

Esta sencilla adaptación permite obtener resultados aceptables con muy pocas modificaciones al algoritmo original. Como siempre, los pesos de las fuerzas agregadas deberán ser ajustados para encontrar un equilibrio entre un trazado que ignora los *clusters* (pesos muy bajos) y uno que trata a cada *cluster* como un vértice (pesos muy altos).

Un sistema para navegar grafos con *clusters* que extiende la idea anterior es el sistema DA-TU de Huang y Eades ([EH00],[HE98]). Pensado para el trazado de grafos con *clusters* y dinámicos, este sistema no sólo permite *clusters* de un nivel, sino toda una jerarquía de agrupamientos (además de la parte dedicada al trazado de grafos dinámicos, de lo cual se ocupa el capítulo 7). Más adelante veremos otro algoritmo que combina *clustering* con grafos dinámicos.

En este contexto, un grafo con *clusters* es un grafo $G = (V, E)$ con un árbol T tal que las hojas de T son exactamente los vértices de G. Cada nodo interno (no hoja) c_i de T es un *cluster*, que agrupa a todas las hojas del subárbol de T que tiene como raíz a c_i. De esta forma, T especifica la relación de inclusión entre los *clusters*.

El modelo de fuerzas usado distingue entre cuatro tipos de fuerzas distintas:

- *Fuerzas internas.* Fuerzas atractivas entre pares de nodos hermanos en T.
- *Fuerzas externas.* Fuerzas atractivas entre pares de nodos que no son hermanos en T.

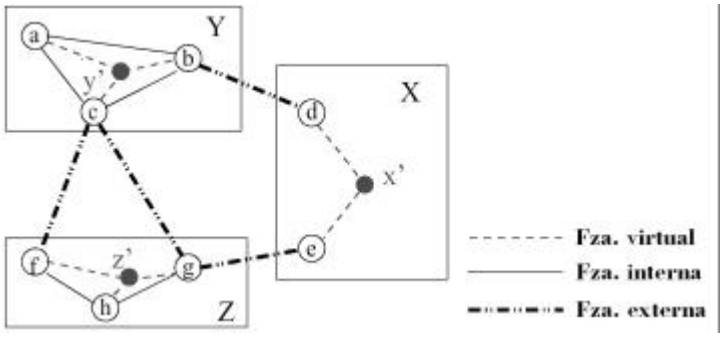


FIGURA 9.2. Tres tipos de fuerzas para trabajar con *clusters* [EH00].

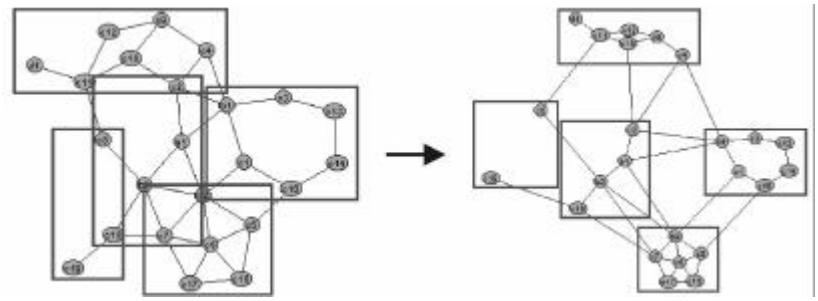


FIGURA 9.3. Un grafo con *clusters* (recuadrados) (izq.) y el resultado de aplicar el algoritmo DA-TU (der.) [EH00]

- *Fuerzas virtuales.* Cada cluster c_i tiene un nodo virtual asociado (equivalente a los atractores de arriba). A su vez, para cada nodo v en G, si el padre de v en T es c_j, entonces se coloca una arista virtual entre c_i y c_j, con una fuerza atractiva virtual asociada.
- *Fuerzas gravitacionales.* Fuerzas repulsivas entre todo par de vértices.

La Figura 9.2 ilustra los tipos de fuerzas para un grafo sencillo.

El algoritmo de trazado utilizado es un derivado del *spring embedder*, ya que permite animar fácilmente los movimientos (recordemos que DA-TU es un sistema de trazado de grafos dinámicos). En la Figura 9.3 puede verse un ejemplo resultado de aplicar el algoritmo, donde se logró eliminar el solapamiento entre *clusters* por completo. Como parte del sistema de navegación provisto, el programa permite "comprimir" *clusters* en nodos, reduciendo la cantidad de vértices en pantalla (y obviamente también se puede descomprimir los comprimidos).

9.2. Dividir y vencer

Como se mencionó en la introducción, los *clusters* dan lugar de manera natural a algoritmos de trazado del tipo "divide y vencerás". Una forma de lograr esto es tratar a cada *cluster* como un super-nodo (vértice con ciertas propiedades). Se

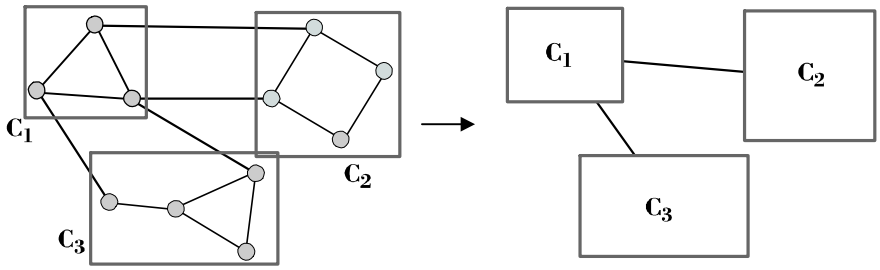


FIGURA 9.4. Un grafo (izq.) y su metagrafo asociado (der.). El tamaño y forma de los vértices del metagrafo depende de los *clusters*.

traza por separado el subgrafo inducido por cada *cluster* y el grafo simplificado que resulta de usar estos super-nodos. Luego en lugar de cada super-nodo, se coloca el trazado del *cluster* correspondiente.

El algoritmo de Wang y Miyamoto [WM95] se basa en esta idea. El grafo simplificado recibe el nombre de "meta-grafo" ($G_{meta} = (V_{meta}, E_{meta})$), y su trazado "meta-trazado".

El meta-grafo tiene tantos vértices como *clusters*. Las aristas son particionadas en dos conjuntos disjuntos:

- *Intra-aristas* (E_{intra}). Aristas que unen dos vértices de un mismo *cluster*.
- *Inter-aristas* (E_{inter}). Aristas que unen vértices de distintos *clusters*.

Notar que $E = E_{intra} \cup E_{inter}$. Cada *cluster* C_i tendrá un meta-vértice (o super-nodo) asociado, v_{C_i} , y estos meta-vértices serán el conjunto de vértices de G_{meta}. Las aristas de G_{meta} serán llamadas meta-aristas, y se formarán colapsando las inter-aristas entre un par de *clusters*; formalmente: $(v_{C_i}, v_{C_j}) \in E_{meta} \Leftrightarrow \exists (u, v) \in E_{inter} : u \in C_i \wedge v \in C_j$. La Figura 9.4 ilustra un grafo con *clusters* y su meta-grafo asociado.

El modelo de fuerzas usado distingue tres categorías de fuerzas: las que afectan a vértices en un mismo *cluster* (intra-fuerzas), las que involucran a vértices en distintos *clusters* (inter-fuerzas) y las que intervienen entre meta-vértices (meta-fuerzas).

Un detalle importante sobre este meta-grafo, que se ve en la Figura 9.4, es que sus vértices van a ser rectángulos con tamaño igual al tamaño que requiera trazar el subgrafo correspondiente. Para trazar el grafo respetando el tamaño de los vértices, Wang y Miyamoto presentan su propio algoritmo, que se explica en el capítulo 8. Allí también se detallan las fuerzas atractivas y repulsivas usadas.

Conceptualmente el algoritmo tiene la siguiente forma:

1. Construir meta-grafo G_{meta}.
2. Trazar el subgrafo asociado a cada cluster.
3. Trazar G_{meta} usando como tamaño de vértices el espacio ocupado por los trazados de 2.

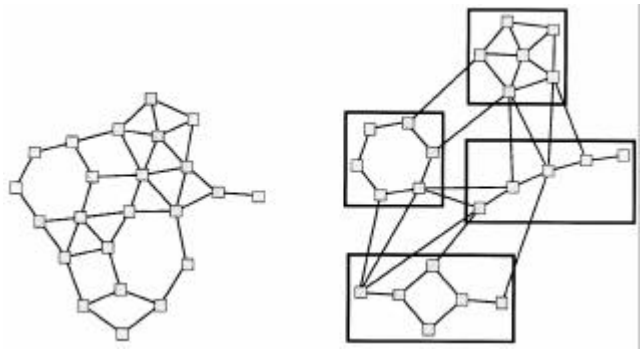


FIGURA 9.5. El resultado de la primer etapa (izq.), y el trazado final (der.) [WM95].

4. Reemplazar cada meta-vértice por el trazado del subgrafo correspondiente y agregar inter-aristas.

Sin embargo, como el paso 2 no tiene en cuenta las inter-aristas, al agregarlas en el paso 4 pueden aparecer muchos cruces, deteriorando el resultado final.

Para evitar esto Wang y Miyamoto integran los pasos 2-4 en único algoritmo de trazado con una variable temporal t, donde las fuerzas en juego van variando a medida que transcurre el tiempo. Las fuerzas que afectan a un vértice $u \in V$ durante el trazado son:

$$F_{comp}(u) = F_{intra}(u) + S(t)F_{inter}(u) + (1 - S(t))F_{meta}(u)$$

Donde F_{intra} son las intra-fuerzas, F_{inter} las inter-fuerzas y F_{meta} las meta-fuerzas que afectan al meta-vértice del *cluster* de u. $S(t) \in [0, 1]$ es una función que depende del tiempo t:

$$S(t) = \begin{cases} 1 & t < t' \quad \text{(etapa 1)} \\ g(t) \in (0,1) & t' \leq t < t'' \quad \text{(etapa 2)} \\ 0 & t'' \leq t \quad \text{(etapa 3)} \end{cases}$$

Con t' y t'' umbrales que fijan la duración de cada etapa y $g(t)$ es una función decreciente que produce una transición entre la primera y la tercer etapa. De esta manera, las etapas "conceptuales" 2-4 del algoritmo van apareciendo gradualmente con el pasar del tiempo. En la primer etapa, $F_{comp}(u) = F_{intra}(u) + F_{inter}(u)$, es decir que se realiza un trazado que ignora los *clusters*. En la tercer etapa, $F_{comp}(u) = F_{intra}(u) + F_{meta}(u)$, por lo que la posición final de los *clusters* es determinada en base a las meta-fuerzas, al mismo tiempo que los vértices de cada *cluster* se mantienen cercanos gracias a las intra-fuerzas.

En la Figura 9.5 se muestra un ejemplo del resultado al final de la primer etapa junto al resultado final, luego de la tercer etapa.

Por último resaltamos el hecho de que este algoritmo se integra además con un sistema para considerar vértices de tamaño no uniforme y restricciones en la posición de los vértices (ambos explicados en el capítulo 8).

9.3. Grafos con clusters dinámicos

En la sección anterior se explicó parte del algoritmo de Huang y Eades ([EH00], [HE98]) para grafos con *clusters* y además dinámicos. En esta sección presentaremos otro algoritmo que también combina ambas características, pero a diferencia de la sección anterior, mencionaremos no sólo lo relacionado con los *clusters* sino también la parte dinámica debido a que son difíciles de separar y creemos que vale la pena mencionarlas. En lo que respecta a la parte dinámica, mantendremos la notación del capítulo 7.

Recientemente, Frishman y Tal presentaron en [FT04] un algoritmo para el trazado de grafos con *clusters* y dinámicos que incluye varias ideas novedosas.

El objetivo primordial del algoritmo es preservar el mapa mental, con especial énfasis en preservar la posición y tamaño de los *clusters*. Denotaremos con G_i y T_i al i-ésimo grafo y a su correspondiente trazado, respectivamente. Para preservar el mapa mental, el algoritmo pretende encontrar un trazado de G_i (T_i) en base a G_i y a T_{i-1} (el trazado anterior).

Estructura de clusters

Para imponer la estructura de los *clusters* sobre G se agrega un vértice atractor v_{C_i} para cada *cluster* C_i, con aristas entre el vértice atractor y los vértices de C_i, junto a aristas entre el atractor y los atractores de otros *clusters* conectados.

Cada arista tiene dos parámetros: longitud y peso. Cuatro longitudes $K_1, .., K_4$ distintas son definidas[1], según el tipo de arista (entre paréntesis se indica la longitud usada por los autores para sus pruebas).

1. Aristas entre el atractor v_{C_i} y vértices de C_i ($K_1 = 2$).
2. Aristas entre dos vértices de un mismo *cluster* C_i ($K_2 = 1,5$).
3. Aristas entre dos vértices de distintos *clusters* ($K_3 = 4$).
4. Aristas entre los atractores de dos *clusters* distintos (K_4 entre 5 y 6).

Las aristas también tienen pesos. Un peso mayor hace que el algoritmo de trazado le de prioridad a conseguir cumplir con la longitud ideal de esa arista (implementado aumentando la constante de la fuerza atractiva). En [FT04] usan pesos de 1 y 2,5 para aristas dentro de un *cluster* y entre *clusters*, respectivamente.

Preservar el mapa mental con vértices invisibles

Para disminuir los cambios visuales se usa una estrategia muy interesante y novedosa basada en vértices invisibles. Todo *cluster* tiene un número – variable – de vértices invisibles, que eventualmente se usan como casillas donde ubicar otros vértices.

[1] En el artículo original, debido a restricciones del dominio de aplicación (visualización de software móvil), tienen un quinto tipo de arista que aquí omitimos.

Cuando un vértice v es quitado de un *cluster*[2], su lugar lo pasa a ocupar un vértice invisible, que se ubica en el nuevo trazado donde antes estaba el vértice v. De manera análoga, cuando un vértice v es agregado, se lo ubica en lugar de algún vértice invisible, y su posición inicial será la que éste tenía.

El uso de estos vértices invisibles tiene un doble propósito. Por un lado, el tamaño del *cluster* se mantiene aproximadamente constante. Por otro lado, sirven para reservar posiciones "conocidas" para nuevos vértices, lo que, se supone, facilita la preservación del mapa mental.

El número de vértices invisibles en cada *cluster* es mantenido entre un umbral mínimo y uno máximo, en proporción a la cantidad de vértices en el agrupamiento. Esto es importante para evitar que el trazado de un *cluster* ocupe mucho lugar debido a que tiene demasiados vértices invisibles.

Algoritmo

El algoritmo en sí mismo también es bastante original, así que lo explicamos aquí brevemente. El pseudocódigo es el siguiente:

AlgoritmoFT(Trazado T_{i-1}, Grafo G_i)
1. T_{inic} = fusión de T_{i-1} y G_i
2. $T^{(1)}$= trazado del grafo G_i a partir de T_{inic}
3. Si $T^{(1)}$ es suficientemente bueno (dens($T^{(1)}$)$<\mu$)
   3.1 $T_i = T^{(1)}$
sino
   3.2 $T^{(2)}$= trazado del grafo G_i permitiendo más movimientos
   3.3 T_i= el mejor entre $T^{(1)}$ y $T^{(2)}$
4. Animar transición entre T_{i-1} y T_i

El paso 1 se encarga de crear un trazado inicial T_{inic} a partir de T_{i-1}, quitando de T_{i-1} los vértices que ya no están en G_i y agregando los vértices nuevos a T_{i-1}. Esto se lleva a cabo por etapas. Primero se copian a T_{inic} todos los vértices que están en ambos (tomando la posición en T_{i-1}). Luego, para cada *cluster*, se forman pares con vértices quitados y agregados, de manera que la posición que ocupaba el vértice quitado la ocupe ahora un vértice nuevo. A continuación los vértices nuevos o quitados que no se hayan podido compensar son agregados o quitados usando vértices invisibles, como se describió arriba. Todos los vértices hasta aquí ubicados en T_{inic} son fijados, es decir que no podrán moverse. Finalmente, los vértices que pertenecen a *clusters* nuevos son agregados.

El paso 2 usa un algoritmo estático dirigido por fuerzas para encontrar un trazado, usando como trazado inicial T_{inic}. El algoritmo contempla las fuerzas ya comentadas y los vértices fijados. En el paso 3 se debe decidir si el trazado obtenido es suficientemente bueno. Para esto se usa únicamente una métrica de densidad de *clusters*, definida como:

[2]Un vértice es quitado (agregado) de un *cluster* cuando está en ese mismo *cluster* en G_{i-1} (G_i) pero no en G_i (G_{i-1}).

$$dens(T_G) = \max_{C_i \in G} \frac{\text{superficie}(C_i)}{|C_i|}$$

Es decir, para cada *cluster* se tiene en cuenta el cociente entre el área que ocupa (para esto se considera el rectángulo más chico que lo contiene) y la cantidad de vértices. El máximo de todos estos valores es el usado (y μ es un umbral prefijado).

Si la densidad del trazado $T^{(1)}$ no es buena, se calcula otro trazado $T^{(2)}$, pero esta vez sin vértices fijos. Finalmente se elige el que sea mejor respecto a la densidad, y se anima la transición entre el trazado anterior y el nuevo.

Los resultados que se presentan en [FT04], si bien no son muchos, son prometedores. En particular, el uso de vértices invisibles, o puesto de otra forma, el uso del "espacio en blanco" como un componente más en el trazado de grafos no ha sido estudiado, y parecería que puede valer la pena hacerlo, no sólo para grafos dinámicos, sino para grafos en general, ya que puede ser un factor más que ayude a mejorar la calidad de los trazados.

9.4. Visualizar los clusters del grafo, sin conocerlos

Los algoritmos hasta aquí vistos reciben como entrada los *clusters* del grafo. Sin embargo, muchas veces los *clusters* no son conocidos de antemano y se quiere obtener un trazado que los muestre. Estamos hablando de *clusters* intrínsecos al grafo, definidos por su estructura: los vértices y sus conexiones. Informalmente, un *cluster* es un conjunto de nodos con muchas aristas que los conectan entre sí.

Los algoritmos dirigidos por fuerzas tienden de manera natural a mantener los vértices conectados cerca (debido al uso de fuerzas atractivas), pero esto no siempre es suficiente cuando el objetivo principal del trazado es exponer los *clusters* del grafo. En particular, para poder separar bien los *clusters* suele ser necesario que algunas aristas sean bastante más largas que otras (las aristas entre *clusters*), pero esto se opone al objetivo de los algoritmos clásicos de obtener aristas de longitud uniforme [Noa04].

Existen numerosos métodos para identificar *clusters* (jerárquicos, particionales, etc.), que exceden ampliamente al trazado de grafos. El lector interesado puede consultar [JD88] para interiorizarse sobre algunas de las técnicas disponibles. En el contexto del trazado de grafos, por ejemplo, Ostry [Ost96] propone una basada en particionar por cliques maximales, pero en realidad tampoco es específico de este tema. En [QE01] para acelerar la velocidad del algoritmo de trazado se usa un método de *clustering* geométrico basado en una descomposición del espacio. Veremos más detalles sobre este algoritmo en el capítulo 11.

El propósito de esta sección no es discutir métodos de análisis de *clusters* sino resaltar el hecho de que algoritmos dirigidos por fuerzas pueden ser usados para destacar los *clusters* presentes en un grafo. Daremos dos ejemplos concretos al respecto.

El primero es un algoritmo ya comentado en el capítulo 10: el preprocesador de [MR02]. Según reportan los autores, Mutton y Rodgers, el preprocesador (sin

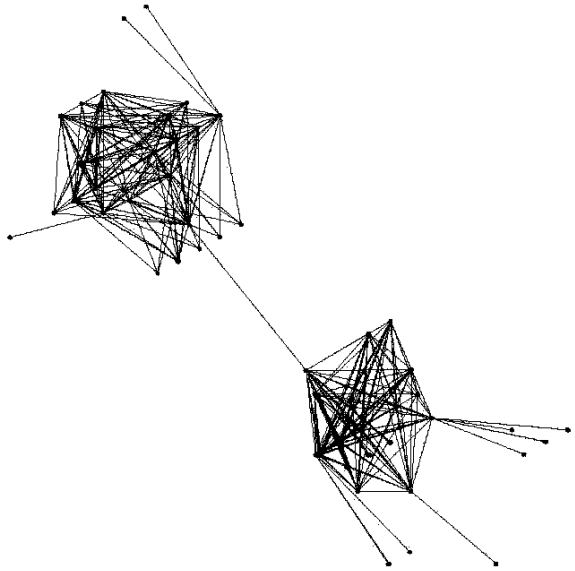


FIGURA 9.6. El preprocesador de Mutton y Rodgers permite identificar los *clusters* del grafo [MR02].

la etapa de mejoramiento) puede ser usado para identificar *clusters*. Si bien esta no es una consecuencia buscada del algoritmo, parecería que en la práctica con muchos grafos lo logra. En la Figura 9.6 se puede ver un ejemplo.

El otro algoritmo que mencionaremos sí fue pensado específicamente para identificar *clusters*. Más que un algoritmo, es un modelo de energía que tiene la propiedad de que los trazados de los mínimos de esta energía revelan los agrupamientos intrínsecos del grafo.

El modelo de energía LinLog fue presentado por Noack en [Noa04]. La energía correspondiente a un trazado T está dada por:

$$U_{LinLog}(T) = \sum_{(u,v)\in E} \|p_u - p_v\| - \sum_{(u,v)\in V\times V, u\neq v} \ln \|p_u - p_v\| \qquad (9.1)$$

El primer término puede interpretarse como el total de las fuerzas atractivas, mientras que el segundo como el de las fuerzas repulsivas entre todo par de vértices.

En [Noa04] se demuestra formalmente que en los trazados de energía LinLog mínima, la distancia entre un *cluster* y el resto del grafo es inversamente proporcional al acoplamiento entre ambos. Esto hace que esta energía provea una forma efectiva de aislar los distintos clusters. Si bien en [Noa04] no hay suficientes detalles como para implementar eficientemente un algoritmo para minimizar esta energía, simples adaptaciones de los algoritmos clásicos deberían servir de punto de partida.

Finalmente, es oportuno notar que la energía LinLog es un caso particular de un modelo general de energías llamadas r-PolyLog (para LinLog, $r = 1$):

$$U_{r-PolyLog}(T) = \frac{1}{r} \sum_{(u,v)\in E} \|p_u - p_v\|^r - \sum_{(u,v)\in V\times V, u\neq v} \ln \|p_u - p_v\| \qquad (9.2)$$

Otra energía conocida es la 3-PolyLog, que derivada en función de $\|p_u - p_v\|$ resulta en

$$U_{3-PolyLog}(T) = \sum_{(u,v)\in E} \|p_u - p_v\|^2 - \sum_{(u,v)\in V\times V, u\neq v} \frac{1}{\|p_u - p_v\|}$$

que no es otra cosa que el sistema de fuerzas de FR (sección 4.2).

Capítulo 10

Trazado inicial

Ya se ha visto que los algoritmos de trazado de grafos dirigidos por fuerzas se dedican a tomar un trazado inicial y mejorarlo, iterativamente, hasta llegar a uno con buenas propiedades estéticas[1]. Esto hace que el trazado desde el cual se empieza a trabajar sea un ingrediente fundamental, que puede tener muchísima repercusión en los resultados.

De hecho, cuanto mejor es el trazado inicial, menos debe hacer el algoritmo de trazado: llevado al extremo, si el mismo fuera ya un trazado óptimo, el algoritmo no tendría que hacer nada. Obviamente partir de un trazado casi óptimo no es posible, pero sí es posible estudiar qué trazados iniciales permiten obtener mejores resultados.

El trazado inicial influye en dos aspectos muy importantes:

- **Velocidad de convergencia**. Dado que la mayoría de los algoritmos que se usan son métodos de optimización local, cuanto más cerca está el trazado inicial de un mínimo, más rápido se converge.

- **Calidad del resultado**. Por lo anterior, el trazado inicial también influye en la calidad del trazado resultante, ya que el mínimo al que el algoritmo de trazado va a converger va a ser uno cercano al trazado inicial. Si este último está cerca de "buenos" mínimos, el resultado final será bueno, pero si, por el contrario, hay mínimos pobres cerca, el algoritmo de trazado puede terminar convergiendo a uno de estos trazados no deseados.

Visto desde otro punto de vista, la influencia del trazado inicial tiene dos consecuencias no deseadas sobre los algoritmos de trazado: (a) el mismo grafo puede tener muchos trazados distintos, y de muchas calidades diferentes; (b) el mismo grafo puede requerir una cantidad muy distinta de iteraciones para converger.

A lo largo de este trabajo hemos ido comentando, al describir los distintos métodos, qué opciones de trazado inicial usa cada uno. En este capítulo nos proponemos analizar con más profundidad cuáles son las alternativas en uso, tanto en los algoritmos ya vistos como en otros que serán presentados aquí. A modo de ejemplo, en la Figura 10.1 se muestran tres trazados del mismo grafo obtenidos con el *spring embedder*, comenzando con distintos trazados iniciales.

[1] En la amplia mayoría de los casos. Hay excepciones, como puede ser el algoritmo Tu (sección 4.7).

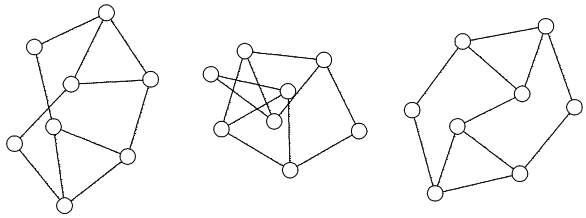


FIGURA 10.1. Tres trazados distintos del mismo grafo, obtenidos variando el trazado inicial.

10.1. Trazado inicial al azar

Los algoritmos basados en el *spring embedder* de Eades (sección 4.1) son los más susceptibles al trazado inicial. Esto se debe a que usan el método del gradiente para alcanzar un mínimo local, lo que hace que una vez fijado el trazado inicial, quede ya determinado el mínimo al cual se va a converger y cuánto tiempo llevará hacerlo.

A pesar de esto, la mayoría de estos algoritmos usa trazados iniciales elegidos al azar, con resultados bastante buenos la mayoría de las veces. Entra aquí en juego un factor importante que es cuáles son los trazados de energía mínima que tiene el grafo. Si todos los mínimos son buenos, no es tan importante a cuál converger, y comenzar con posiciones al azar no trae inconvenientes en este sentido. Pero si existen muchos trazados mínimos pobres, entonces será frecuente que un trazado elegido al azar desemboque en un resultado pobre.

Esto explica que algunos autores sostengan que el trazado inicial no tiene gran influencia en los resultados [FLM95]. Muchos grafos sencillos, generalmente pequeños o medianos, sólo presentan mínimos buenos, y para la mayoría de los trazados iniciales, el resultado de comenzar al azar será de buena calidad (igualmente, como se menciona en [FLM95], la velocidad de convergencia sí se ve afectada).

Es oportuno volver a señalar que existen ciertas clases de grafos, como mallas grandes o árboles, que tienen una gran cantidad de mínimos locales pobres [Tun99b]. En estos últimos, los mínimos pobres son provenientes de intercambiar la posición de dos subgrafos de manera de que un subárbol quede cruzado sobre otro (para un ejemplo ver Figura 4.9). Para estos grafos, un trazado inicial al azar no es lo más conveniente. De todas maneras, el trazado de árboles no es el principal objetivo de estos métodos (para ellos existen otras técnicas especializadas más eficientes), y siempre es posible tener un trazado inicial distinto reservado para cuando el grafo sea un árbol. Esto hace que, a pesar de todo, para los *spring embedders* el trazado al azar de grafos no muy grandes sea una buena opción debido a que funciona bastante bien y no tiene un gran costo.

El algoritmo KK (sección 4.4) no es tan sensible a la posición inicial de los vértices debido a su forma de optimizar. Los mismos Kamada y Kawai confirman con sus pruebas que el trazado inicial no tiene gran influencia en el resultado

final [KK89]. A pesar de esto, no usan en su algoritmo original un trazado al azar sino que ubican los vértices sobre un círculo de diámetro fijo. No está claro si esto trae algún beneficio sobre un trazado al azar (que es la opción usada en muchas implementaciones), además de evitar que todos los vértices queden alineados. En este último caso, el algoritmo KK producirá un trazado que también tiene todos los vértices alineados, algo no deseado en la mayoría de los casos. Si bien Fruchterman y Reingold [FR91] no lo mencionan, el algoritmo FR también sufre de este problema [CSP96].

Los métodos como DH (sección 4.5) son los menos sensibles al trazado inicial, debido a que – al menos en sus versiones originales – todo el proceso de optimización es azaroso (puesto que todas las direcciones de movida se eligen al azar). Recordemos que en las primeras iteraciones de DH los vértices se mueven al azar dando grandes saltos en el espacio de dibujo, por lo que poco quedará del trazado inicial luego de unas cuantas repeticiones. Lamentablemente esta independencia del trazado inicial no significa que los resultados para un grafo dado sean siempre los mismos, sino todo lo contrario: ni siquiera es posible "sugerir" mejorar el resultado con un buen trazado inicial. Esto es una ventaja y una desventaja al mismo tiempo, ya que el uso del azar es el que permitirá muchas veces evitar mínimos locales pobres.

10.2. Trazados iniciales elaborados

En un intento por mejorar la velocidad de convergencia y los resultados finales, varios métodos para generar trazados iniciales han sido propuestos. A diferencia del sencillo trazado al azar, estos métodos puede llegar a ser muy complicados, como se verá a continuación.

Uno de los primeros – y más elaborados – ha sido el propuesto por Harel y Sardas en [HS95]. El mismo fue pensado para ser combinado con el algoritmo DH, aunque cualquier otro de los basados en el *spring embedder* también sirve. Un intrincado algoritmo para encontrar un trazado inicial es propuesto, cuyo principal objetivo es lograr que grafos planares tengan trazados planos (algo que no siempre sucede).

El algoritmo completo está dividido en seis etapas:

A: Realizar prueba de planaridad.
B^-:Extraer subgrafo planar.
B: Calcular trazado del subgrafo planar.
B^+: Reinsertar las aristas removidas.
C: Realizar trazado plano de todo el grafo.
D: Realizar mejoramiento del trazado.

Comentaremos brevemente en qué consiste cada una. La primera etapa consiste en ver si el grafo es planar. Si lo es, B^- y B^+ se omiten. Si no es así, en B^- se busca un subgrafo planar maximal. Una vez que se tiene esto, en B se usa un algoritmo

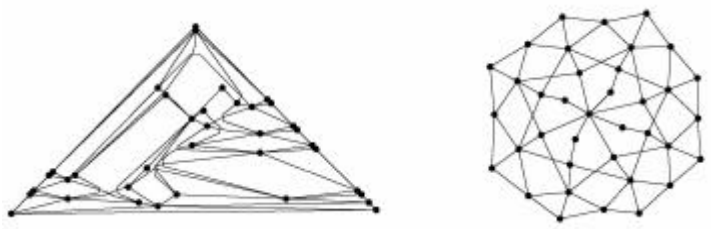


FIGURA 10.2. Resultado intermedio (izq.) y resultado final (der.) de aplicar el complejo algoritmo con preproceso [HS95].

basado en árboles PQ para construir un orden de los vértices que luego (en C) permitirá obtener un trazado plano. En B^+ se reinsertan las aristas removidas en B^-, y en C se realiza un trazado del grafo completo con un algoritmo también basado en árboles PQ[2]. Este algoritmo no es dirigido por fuerzas ni cumple con ninguno de los criterios estéticos usuales, excepto que el trazado está libre de cruces. Por último, la etapa D consiste en aplicar un algoritmo de trazado basado en DH que produce el resultado final.

La idea de todo esto es producir un trazado inicial que no es estéticamente correcto, pero sí topológicamente correcto. En particular, si el grafo es planar, el trazado inicial será plano.

El algoritmo que usan en la etapa D usa la misma función objetivo que DH y el mismo método para mover los vértices, pero suprimen la posibilidad de las movidas cuesta arriba, porque, según experimentaron, partiendo de este trazado topológicamente correcto la mayoría de las movidas son cuesta abajo. Además agregan un chequeo para evitar introducir cruces de aristas al mover un vértice.

Todo este complejo preproceso para obtener un trazado inicial da resultados. Por un lado los grafos planares siempre obtienen trazados planares, y por el otro se logra acelerar la velocidad sustancialmente, ya que lo más costoso, DH, sólo debe mejorar un trazado inicial muy bueno. Sorprendentemente, las etapas de preproceso, A-C, sólo son responsables del 3 % del tiempo de corrida del algoritmo. En la Figura 10.2 se muestra el trazado resultado de la etapa C (izq.), y el mismo luego de la etapa de mejoramiento (der.). Se puede ver que las diferencias son muy importantes.

Behzadi [Beh99] también corroboró con su algoritmo *CostSpring* que algunos tipos de grafos son demasiado sensibles al trazado inicial. Por ejemplo, en sus pruebas con una malla de 3×20, el 80 % de las veces el resultado fue un mínimo local pobre. Para disminuir las consecuencias de esto propone ejecutar las primeras iteraciones de su algoritmo con varios trazados elegidos al azar. A cada uno de estos trazados le aplica una función de costo, y el de menor costo es elegido para continuar.

[2]Los detalles de este algoritmo son bastante engorrosos y no aportan mucho sobre la parte dirigida por fuerzas de la cual se ocupa este trabajo, así que referimos al lector interesado en más detalles a [HS95].

Este recurso puede aplicarse siempre, con casi cualquier algoritmo, y está en la misma línea del enfoque de *multitrazado* de [BMRW98], donde se calculan varios trazados y se deja al usuario que se quede con el mejor.

Si bien esta propuesta de Behzadi ayuda a disminuir la pérdida en calidad por un trazado inicial malo, es bastante costosa, y depende mucho de cuántos trazados iniciales usar. Para que el 100 % de los trazados de la malla de 3×20 sea bueno, en [Beh99] necesitaron usar entre 10 y 15 trazados iniciales, aumentando el tiempo de corrida casi 7 veces.

Más cerca de Harel y Sardas que de Behzadi, en [MR02], Mutton y Rodgers presentan un pre-procesador para *spring embedders 3D*. El método para obtener el trazado inicial tiene dos etapas (nuevamente, K denotará la longitud ideal de las aristas).

La primera etapa busca posiciones donde todas las aristas tengan aproximadamente la misma longitud δ ($\delta > K$). La segunda etapa intenta que los vértices mantengan una distancia de al menos K, ubicándolos sobre un malla entera con celdas de $K \times K$.

El pseudo código es el siguiente:

PreprocesadorMR(Grafo G)
//Parte 1
1. Repetir M veces
  1.1 Para cada v $\in$ V
    1.1.1 $F_v = \{ p_v^u \in \mathbb{R}^3 \ / \ (p_v^u - p_u) = \delta \frac{(p_v - p_u)}{\|p_v - p_u\|} \wedge u \in V \wedge (u,v) \in E\}$
    1.1.2 Ubicar v en $p_v = p_v + \frac{1}{\deg(v)} \sum_{p \in F_v} p$
//Parte 2
2. Para cada v $\in$ V
  2.1 Mover v a la posición libre en la malla más cercana a p_v.

La parte 1 ubica a cada vértice en una posición calculada como sigue: para cada vecino de v, se estima dónde debería estar v si la distancia al vecino fuera la ideal δ (en el pseudocódigo, ese valor sería $p_v + p_v^u$). Luego se ubica v en el promedio de esos puntos. Luego de la etapa 1, el trazado tendrá distancias entre aristas de aproximadamente δ. Se puede ver que esta primera parte muestra cierta semejanza con el algoritmo baricéntrico de Tutte (sección 4.8).

La segunda parte simplemente intenta ubicar a los vértices a distancia K uno del otro, ubicándolos en las celdas de una malla con posiciones enteras. En [MR02] el algoritmo FR es aplicado luego del preproceso.

Las pruebas presentadas en [MR02], todas sobre grafos de sitios de internet, muestran que el uso de este preproceso produce un incremento sustancial en la velocidad de convergencia en grafos de más de 100 vértices. El ahorro en tiempo aumenta con el tamaño del grafo, y para grafos de 300 y 400 vértices se alcanza un mínimo local unas 10 veces antes que sin el preproceso. Habría que estudiar qué efecto produce en otras clases de grafos, pero de todas formas merece ser tenido

muy en cuenta, ya que además de lograr disminuir el tiempo, es muy sencillo de implementar.

Utilizar el mismo algoritmo de trazado de grafos para producir un trazado inicial es una idea prometedora que no ha sido muy explorada. En realidad, a medida que la complejidad del cálculo de los trazados iniciales aumenta, pasa a transformarse en un nuevo algoritmo de trazado – dirigido por fuerzas –, que tiene como objetivo encontrar trazados que sirvan de entrada a otros algoritmos de trazado de grafos (y no necesariamente trazados estéticamente buenos).

En [Cre01], Creek propone un método para encontrar un trazado inicial bueno, denominado *Big Bang*. El nombre está inspirado en que el objetivo del método es expandir los vértices, para establecer, a grandes rasgos, la relación espacial entre los vértices y eliminar posibles situaciones patológicas que lleven a mínimos locales pobres.

La forma de producir esta expansión es aplicando un algoritmo tipo *spring embedder*, con una elección especial de fuerzas: las fuerzas repulsivas son mucho más fuertes que las atractivas, favoreciendo que el grafo se "expanda". Esta etapa de iteraciones con fuerzas especiales se realiza $\approx 2n$ veces, variando según la dificultad que presenta el grafo. El trazado resultante es usado como trazado inicial para el algoritmo de trazado de grafos. Si bien en [Cre01] se usa un algoritmo propio, cualquier otro puede ser usado.

Según Creek, los resultados que se obtienen usando este trazado inicial son muy buenos: son independientes del trazado inicial al azar, reducen la variación en la cantidad de iteraciones necesarias, y los grafos que ya se trazaban "bien" con un trazado inicial al azar, también lo hacen con este trazado inicial más elaborado. Según los resultados de [Cre01], el tiempo que requiere el cálculo del trazado inicial es compensado por una convergencia más rápida.

Otro algoritmo que cuenta con un trazado inicial especial es el algoritmo jerárquico de Gajer et al. [GGK00], pensado para dibujar grafos grandes. Dado que el método para determinar la posición inicial de los vértices está muy ligado al algoritmo, explicaremos ambos en el capítulo 11.

Capítulo 11

Grafos Grandes

Las técnicas de trazado de grafos vistas hasta este momento son eficientes cuando se trata de grafos con una pequeña cantidad de nodos (hasta unos pocos cientos), pero no es así cuando se trabaja con miles o inclusive millones de nodos. Aunque puede parecer difícil encontrar aplicaciones de visualización donde se trabaje con grafos tan grandes, cada día son mayores los volúmenes de información manipulados, y por ende la cantidad de información que debe ser visualizada. El ejemplo canónico al respecto es la visualización de grafos tomados de Internet, que fácilmente pueden superar los cientos de miles de vértices.

Para poder lidiar con este problema han aparecido en los últimos cinco años nuevos tipos de algoritmos que se enfocan exclusivamente en poder dibujar grafos de enorme cantidad de vértices.

Estos algoritmos pueden dividirse en dos ramas:

1. Los multidimensionales
2. Los de simulación de N-cuerpos

En el presente capítulo trataremos cada uno de estos en detalle.

11.1. Algoritmos multidimensionales

Uno de los primeros trabajos sobre algoritmos multidimensionales fue el de Gajer et al. [GGK00], [GK00] en el cual se presenta una nueva forma de pensar los algoritmos de trazado de grafos. Las soluciones vía modelos dirigidos por fuerzas son muy buenas tanto sea en costo como en resultados obtenidos. Como vimos en capítulos previos, el eje principal de una solución dirigida por fuerzas es presentar un trazado inicial y una función de costo y lo que diferencia un método de otro es la función elegida y la forma en que se busca optimizarla. Esta idea se hace impracticable cuando los grafos superan los cientos de nodos visto que la cantidad de distancias entre nodos que se deben calcular (cuadrática en el número de nodos) determina que estos algoritmos se tornen imprácticos.

Es importante notar que en el problema de dibujar grafos grandes la realidad no es exactamente la misma que en el problema de grafos pequeños. Supóngase un ejemplo con un grafo de un millón de nodos. Es probable que si se pudiese esperar el tiempo suficiente para utilizar algún algoritmo clásico, el resultado sea incomprensible visto que sólo se podría ver un enjambre de aristas sin ninguna estructura definida. Por estos motivos es que para el tratamiento de grafos grandes se consideran comúnmente dos opciones.

La primer opción conocida como *fisheye* consiste en ver determinada porción del grafo en detalle pero dejando el resto con menor definición. Este efecto es similar al efecto que provocaría ver con una lupa el grafo.

La segunda opción, no menos utilizada, es la llamada visión multinivel. En este caso lo que se busca es poder separar el grafo en determinados niveles y ver en cada nivel con más detalle lo que se veía en el nivel anterior. Dicho efecto es similar al utilizado cuando se realiza *zoom* en una imagen digital (mientras que la primer opción sería equivalente a ver el grafo a través de una ventana con un *zoom* al 100 %).

11.1.1. Gajer: un algoritmo multicapa

La idea presentada por Gajer et al. consta de tres partes bien diferenciadas, las cuales le permiten dibujar grandes grafos en cuestión de segundos.

La primera es dividir el conjunto de nodos en k conjuntos V_i tales que $V_0 \supset V_1 \supset V_2 \supset ... \supset V_k \supset \emptyset$, siendo V_0 el conjunto de todos los nodos. Una vez definidos los conjuntos, la forma de dibujar el grafo completo es dibujar V_k, lo cual es sencillo porque tiene pocos nodos, entonces una vez dibujado V_k sobre ese dibujo agregar los nodos de V_{k-1}, lo cual también debe ser sencillo porque al estar los primeros nodos ya fijos, los siguientes pasan nuevamente a ser como un dibujo pequeño, y así sucesivamente. Vemos cómo esta idea viene de la par del concepto de la visión multinivel que se le podrá dar al grafo.

Claramente la dificultad del algoritmo radica en poder definir de forma adecuada los conjuntos V, lo cual no es trivial. La estrategia utilizada para este fin fue bautizada como MIS (por *Maximal Independent Set*), la cual se define de la siguiente forma. Supóngase que V_i es un conjunto independiente correspondiente al paso i del algoritmo; el conjunto V_{i+1} se define con el siguiente algoritmo (a partir de i=0):

1. Sea $V_i' = V_i$
2. Mientras $V_i' \neq \emptyset$
   - 1.1. Sea $v \in V_i'$
   - 1.2. $V_i' = V_i' - \{v\}$
   - 1.3. $V_{i+1} = V_{i+1} \cup v$
   - 1.4. Eliminar de V_i' todos los nodos $u \in V_i'$ tales que dist$(u,v) \leq 2^i$

Es claro que para el caso de $i = 0$, es decir cuando se comienza el algoritmo, V_0 es el grafo original y para construir V_1 se eliminan los nodos que se encuentran a distancia menor o igual que uno, es decir los que están conectados al vértice elegido. De esta forma se genera un subconjunto independiente maximal, y de ahí sale el nombre de la estrategia. Lo importante de esta forma de tomar los nodos es que garantiza que se encuentren uniformemente distribuidos y la cantidad de nodos que queda en cada paso es el logaritmo de la cantidad de nodos en el paso anterior. Por esto mismo el valor de k (la cantidad de conjuntos generados) es $k = \log(|V|)$.

La segunda parte importante del algoritmo consiste en definir un buen trazado inicial para los vértices de cada conjunto. Los primeros nodos (es decir los del

conjunto V_k, ya que se empieza por el más pequeño) siempre serán a lo sumo tres por el procedimiento con el cual se formaron los conjuntos, entonces la forma de acomodarlos será ubicándolos de forma tal que formen un triángulo donde cada arista tenga una longitud equivalente a la distancia teórica que separaba a dichos nodos en el grafo, es decir:

$$\begin{cases} d_{uv} = \delta_{uv} \\ d_{vw} = \delta_{vw} \\ d_{wu} = \delta_{wu} \end{cases}$$

donde u,v y w son los tres vértices del conjunto V_k, d_{uv} es la distancia entre u y v en el trazado y δ_{uv} es la distancia teórica entre u y v en el grafo original.

En caso de no tener el último conjunto tres nodos, se modifican los últimos dos conjuntos para que el último resulte de exactamente tres nodos. Luego para cada subconjunto de forma iterativa se agregan los nodos de una forma similar a la presentada. Supóngase que deseamos agregar el nodo $t \in V_{i-1} - V_i$, entonces lo que se hace es considerar los nodos (ya ubicados) u, v, w tales que son los más cercanos a t según las distancias del grafo. Con esta información, a t se lo ubica resolviendo el siguiente sistema:

$$\begin{cases} d_{ut} = \delta_{ut} \\ d_{vt} = \delta_{vt} \\ d_{wt} = \delta_{wt} \end{cases}$$

El problema es que este sistema puede no tener solución, así que se consideran las ecuaciones a pares, y se calculan las soluciones de tres sistemas de ecuaciones. Como las mismas son cuadráticas, se pueden obtener hasta seis resultados distintos. Se seleccionan las tres soluciones más cercanas entre sí y se las promedia, definiendo de esta forma la nueva posición de t. Prosiguiendo de la misma forma se ubican el resto de los vértices.

El algoritmo de Gajer et al. también utiliza temperaturas locales como una forma de limitar el movimiento de cada vértice. Sin embargo, el cálculo de la temperatura en cada iteración no es tan importante visto que, a diferencia de la mayoría de los algoritmos dirigidos por fuerzas, en este caso la posición inicial no es al azar, sino que es pensada de forma tal que el nodo no se desplace mucho una vez ingresado, por lo tanto desde un comienzo la temperatura es pequeña y no cambia mucho a lo largo del algoritmo.

El tercer paso importante e innovador de este algoritmo es que todo lo explicado recién sobre el trazado inicial y como se acomoda tomando una pequeña temperatura es fácil de extender a cualquier dimensión, entonces lo que se hace es hacerlo en altas dimensiones donde se tienen mayores libertades y luego se proyecta a $\mathbb{R}^2$o $\mathbb{R}^3$. El nuevo problema que surge es cómo realizar dicha proyección. La solución propuesta es bastante sencilla: se genera una base ortonormal de $\mathbb{R}^n$ y luego se proyecta sobre el subespacio generado por los dos o tres últimos vectores de esa base (dependiendo si el trazado será en $\mathbb{R}^2$ o $\mathbb{R}^3$). Ambos pasos se pueden hacer linealmente utilizando el algoritmo de Gram-Schmidt y luego simplemente

restando y multiplicando para realizar la proyección. Como veremos más adelante, pueden obtenerse resultados aun mejores refinando la forma de proyectar.

El algoritmo resultante es el siguiente:

Multidimensional(Grafo G)
1. Crear los conjuntos $V_0 \supset V_1 \supset V_2 \supset ... \supset V_k \supset \emptyset$
2. Para $i = k$ hasta 0
    2.1. Para cada $v \in V_i$
        2.1.1 Calcular los vecinos N_j a distancia i, $i-1$,...,0 de v
        2.1.2 Definir la posición inicial de v
    2.2 Repetir hasta que se cumpla condición de corte
        2.2.1 Para cada $v \in V_i$
            2.2.1.1 Calcular la temperatura $h(v)$
            2.2.1.2 $D(v) = h(v)\overrightarrow{F_{N_i}(v)}$
        2.2.2 Para cada $v \in V_i$
            2.2.2.1 $p_v = p_v + D(v)$
3. Agregar las aristas
4. Proyectar sobre los p_v a $\mathbb{R}^2$ o $\mathbb{R}^3$

La complejidad del algoritmo es $O(n \log(n))$, que es lo requerido para poder calcular los conjuntos V_i. El resto de los pasos del algoritmo, incluido el trazado inicial y la proyección, se realizan en $O(n)$.

Esta forma de pensar el trazado de grafos ha abierto una rama hasta ese entonces inexplorada. La cantidad de nodos que debía tener un grafo para poder ser graficado con un método dirigido por fuerzas no podía superar los cientos de vértices, pero con técnicas como esta se pueden graficar grafos con millones de nodos (con técnicas que veremos más adelante se han llegado a procesar grafos de 10^6 nodos en cuestión de segundos). Lo más notorio es cómo todo esto se logró dentro del paradigma de trazado dirigido por fuerzas, cuando una de las críticas más comunes que recibían es que no eran útiles para grafos grandes. Son considerados dentro del paradigma porque pese a realizar una evaluación fraccionada en pequeños conjuntos, la optimización que se realiza es la de cualquier algoritmo dirigido por fuerzas. En el caso recién presentado en cada iteración se utiliza el método FR, aunque en el trabajo original recomiendan también utilizar KK.

En la Figura 11.1 pueden verse dos trazados ($n \approx 1000$) obtenidos con el algoritmo de Gajer et al.

11.1.2. Proyectando altas dimensiones de forma inteligente

En los años siguientes surgieron variantes de este algoritmo, como es el caso de ACE [KCH01], que aplica el mismo concepto cambiando la función de energía por una cuadrática, pero sin cambiar su estructura radicalmente, hasta que en el año 2002 Harel y Koren presentaron en [HK02b] un nuevo algoritmo, que nuevamente reformula la forma de trazar grafos grandes. La idea que plantea es valerse de

FIGURA 11.1. Trazados producidos con el algoritmo GRIP [GK00].

dimensiones altas para poder simplificar el trazado inicial aun más y luego proyectar el grafo a una dimensión menor como $\mathbb{R}^2$ o $\mathbb{R}^3$ pero de una forma inteligente y no azarosa como es el caso de Gajer et al., lo cual era, en cierta forma, una desventaja.

Antes de explicar el funcionamiento del algoritmo planteado por Harel mencionaremos un trabajo previo que realizó con Hadany [HH99] del cual utilizó algunos conceptos para su próximo método. El algoritmo presentado en [HH99] considera el grafo visto desde distintos niveles de abstracción. El concepto importante que aquí se presenta es que el grafo debe ser estéticamente agradable en todos los niveles de abstracción en que se lo mire. Para realizar esto se plantean abstracciones llamadas "grafos gruesos" (*coarse graphs*) en los cuales se simplifica su complejidad pero se intenta mantener la topología del grafo original. La función de energía utilizada es la misma presentada por KK (4.11) en capítulos anteriores. La minimización de dicha función es realizada en estos grafos para de esta forma considerar siempre pocos nodos al mismo tiempo. La forma de minimizar la función consta de tres partes:

1. *Fine-scale:* se realiza la optimización del grafo de forma local dentro de cada *coarse graph.*

2. *Coarse-scale:* se solucionan los problemas que posiblemente introdujo la etapa 1 visto que no posee una visión global del problema.

3. *Fine-scale:* se solucionan los problemas posiblemente agregados por la etapa 2.

En los pasos 1 y 3 realizan una minimización local de la función de energía de KK utilizando descenso por gradiente.

En el paso 2 se juntan nodos adyacentes de forma tal que minimicen la suma de los siguientes tres puntos:

1. Cantidad de *clusters*: cantidad de vértices que forman un nuevo vértice en esta capa de abstracción.

2. Grados de libertad: el grado de los vértices que forman el nuevo vértice.

3. Número homotópico: cantidad de nodos en la visión actual que son adyacentes a ambos extremos de la arista comprimida.

La forma de realizar esta minimización no es exacta sino que se realiza de forma golosa.

Lo interesante es que este proceso se repite una y otra vez pero realizando la optimización de KK sólo sobre una vecindad pequeña. De esta forma, dado que cierto nivel de abstracción ya se encuentra bien dibujado, en el siguiente nivel con menos nodos (los nodos comprimidos en la parte de *coarse-scale*) sólo se debe realizar una optimización local para mantener la topología.

El trazado inicial es simple: se elige una dimensión m y para dibujar el grafo en dicha dimensión lo que se hace es tomar m nodos uniformemente distribuidos como *pivotes* y formar una base con estos nodos. Esto significa que si p_i es el pivote de la coordenada i entonces la coordenada i-ésima de cada nodo será la distancia al pivote que dicho nodo tiene en el grafo.

Para hacer que los nodos estén uniformemente distribuidos lo que se hace es elegir al azar solamente el primer pivote y luego el pivote $i+1$ es aquel que maximiza la distancia a todos los pivotes anteriores. De esta forma se garantiza una uniformidad en la distribución de los pivotes.

El algoritmo para el trazado inicial resulta ser:

TrazadoInicEnAltasDimensiones(Grafo G, Entero m)
1. Elegir p_1 al azar
2. Inicializar cada posición de $d[1, ..., n]$ con ∞
3. Para $i = 1$ hasta m
    3.1 Para cada $v \in V$
        3.1.1 $X^i = d_{p_i v}$ (distancia calculada con BFS)
        3.1.2 $d[v] = \min\{d[v], X^i\}$
    3.2 $p_{i+1} = \arg\max_{v \in V}\{d[v]\}$

La complejidad del trazado es $O(m|E|)$ porque se realiza un BFS por cada dimensión. Las dimensiones que suelen usar los autores son m entre 50 y 100.

El problema que luego se debe resolver es cómo proyectar a una dimensión más baja el grafo resultante, y para esto se utiliza PCA (análisis de componentes principales) que por medio del uso de autovalores se puede realizar en $O(m^2 n)$ (o se puede calcular de otras formas visto que es un área arduamente estudiada).

11.1.3. Emparejamiento multicapa

Otra forma de encarar el problema de grafos grandes es la presentada por [Wal03] en un trabajo muy reciente. Básicamente mantiene la misma línea que [HH99] pero presentando una nueva forma de generar los grafos de cada nivel de abstracción.

El algoritmo general de Walshaw plantea generar distintos niveles de abstracción para luego realizar la optimización sobre estos grafos simplificados. Considérese una sucesión $G_0 \supset G_1 \supset ... \supset G_L$, G_0 es el grafo original, por ende el de mayor tamaño, y a partir de éste se generan los siguientes subgrafos. Luego se realiza la optimización sobre G_L y cada grafo optimiza sobre lo generado en la capa anterior.

Generando los subgrafos. El objetivo es realizar este paso de forma eficiente para poder luego minimizar la función de energía sobre estos grafos. La forma en que se plantea generar los subgrafos debe ser sencilla y no debe agrupar muchos nodos juntos, sino se perderían los beneficios de esta agrupación. Por estos motivos decide utilizar la conocida técnica de emparejamiento (*matching*), es decir agrupar aquellos vértices que compartan una arista. Las mejores soluciones que se conocen para este problema son demasiado lentas para el nuestro, entonces en lugar de utilizar un emparejamiento se utiliza solamente un emparejamiento maximal calculado de forma heurística en un orden menor. La forma de realizar esto es con el algoritmo de Hendrickson y Leland. Su procedimiento consiste en listar los nodos al azar e ir recorriendo esa lista para emparentarlos. Luego los nodos emparentados son fusionados en uno solo. A los nodos formados de esta forma los llamaremos nodos pesados, y definimos el *peso del nodo* como la cantidad de nodos del grafo original que representa. La forma de elegir las parejas es recorrer la lista de nodos y a cada nodo emparentarlo con alguno de sus vecinos que no esté marcado aun, y en caso de no haber ningún vecino que no esté marcado se lo emparenta consigo mismo. El vecino seleccionado es marcado, sacado de la lista y fusionado con su pareja. Este proceso se repetirá con todos los nodos pesados que se originaron por la colisión de otros nodos. En caso de poder seleccionar más de un nodo, se selecciona el vértice de menor peso, para poder mantener de esta forma los pesos lo más parejo posible.

Trazado inicial y optimización. Una vez definidos los grafos $G_0, ..., G_L$ se debe definir la forma de realizar el trazado inicial. Una forma sencilla es generar G_L hasta $|G_L| = 2$, quedando de esta forma sólo 2 nodos. Es sencillo ver que siempre es posible hacer esto manteniendo que el grafo siga estando conectado. Lo bueno de realizarlo es que el trazado inicial de sólo dos nodos puede ser producido al azar. Para los siguientes grafos la forma de definir el trazado inicial es trivial. Supongamos que se desea trazar el grafo G_i partir del G_{i+1}, lo que se hace es colocar cada nodo de G_i donde se encuentra su representante en G_{i+1}. Claramente esto puede llevar a que varios nodos terminen en la misma posición, pero cuando se realice la optimización local se separarán.

Para la etapa de optimización se utiliza un algoritmo dirigido por fuerzas clásico como es FR, pero adaptándolo para que represente mejor la situación. Las fuerzas repulsivas son ponderadas por los pesos de los nodos, haciendo de esta forma que un supernodo que contiene más nodos repela más que uno que posee pocos. Con las

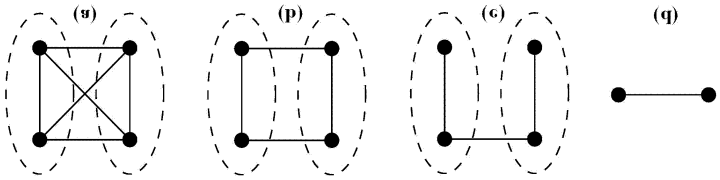


FIGURA 11.2. Ejemplo de porqué dar pesos a las aristas no soluciona el problema de la longitud.

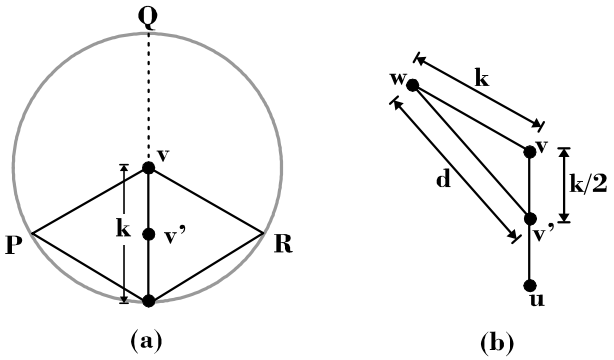


FIGURA 11.3. Cálculo de la longitud de aristas [Wal03].

aristas esto no se realiza porque se puede ver en el siguiente ejemplo que asignarle a la arista un peso proporcional a las aristas que representa no necesariamente da buenos resultados. En el ejemplo de la Figura 11.2 en todos los casos se desearía que diesen iguales, pero los pesos de las aristas son 4, 2 y 1, lo cual daría resultados diferentes.

Lo último que se debe definir es la longitud ideal k, la cual es un poco más compleja de definir visto que debe escalarse de forma que mantenga los criterios de beatitud de un nivel de abstracción al siguiente.

Consideremos que la distancia ideal para determinado nivel del grafo sea k, y que se desea colapsar el nodo v con el nodo u. Si un nodo se encuentra conectado por medio de una arista a v sería de esperarse que se encuentre a distancia k – la distancia ideal para ese nivel – un caso particular de esto debería ser u. Veamos entonces que existe un círculo de centro v y radio k que contiene al nodo vecino a v. Definamos los puntos PQR como indica la Figura 11.3 (a). La distancia de Q a v al colapsar en v' pasa a ser $3k/2$, y $\sqrt{3}k/2$ la distancia de P y R. Considerando el caso promedio supondremos que el nodo w que está conectado a v se encuentra en el medio del arco formado por P y Q, entonces por la regla del coseno obtenemos que se encuentra a distancia $\sqrt{7/4}k$ de v'.

De esta forma podemos definir la distancia ideal de un nivel de abstracción basándonos en el anterior simplemente tomando:

$$k_l = \sqrt{4/7}k_{l+1}$$

FIGURA 11.4. Trazados de grafos grandes obtenidos con el algoritmo de Walshaw [Wal03].

Complejidad. El orden de complejidad resultante es difícil de calcular visto que depende altamente de L y esto depende mucho del grafo y del azar. En el caso ideal se disminuye en un factor de dos en cada paso y en el peor caso sólo se disminuye un nodo por iteración (el caso de una estrella). Con esto podemos decir que $\log_2 n \leq L \leq n$, lo cual indica que el algoritmo se comporta mejor cuando el radio del grafo es mayor.

La etapa de emparejamiento y de generación de los grafos se realiza en $O(V_l + E_l)$ en cada nivel. El trazado de cada capa es en general cercano a $O(V_l + E_l)$. Es importante ver que las pruebas realizadas mostraron que esta estrategia pese a tener igual orden de peor caso en la mayoría de las instancias de entrada es ampliamente superior.

En la Figura 11.4 pueden verse tres trazados obtenidos con este algoritmo. El primero (de izq. a der.) tiene 4.970 vértices. El segundo es un detalle de un trazado del grafo *4elt*, de 15.606 vértices. El tercero es un trazado del grafo *sierpinski10*, que tiene 88.575 vértices.

11.2. Simulación de N cuerpos

El problema de la simulación de N cuerpos es un problema clásico de la física que estudia sistemas en los cuales interactúan varios cuerpos. Tal es el caso de sistemas estelares, moleculares o tantos otros. Estos sistemas suelen caracterizarse por estar compuestos por una gran cantidad de cuerpos y eso es lo que dificulta su análisis, visto que suelen tenerse en cuenta tres tipos de fuerzas, las fuerzas débiles que tienen un alcance reducido, las fuerzas independientes del número de cuerpos y dependientes de la posición de éstos, y las fuerzas de interacción entre los cuerpos. El problema de estos sistemas son estas últimas visto que para calcularlas se tiene un orden $O(n^2)$, lo cual manejándose con números grandes en muchos casos es impracticable. Es importante notar que la simulación de N cuerpos sólo tiene sentido para grafos grandes porque en los casos de grafos chicos no es necesario estimar las fuerzas visto que se pueden calcular directamente. Si se observa bien, el algoritmo clásico de Eades puede ser visto como un sistema de N cuerpos pero que no aproxima las fuerzas lejanas, y por esto mismo es que no se lo puede escalar cuando aumenta el tamaño de N (cantidad de vértices).

Desprendida de esta teoría clásica nace la simulación de N cuerpos, que lo que intenta es aproximar estas fuerzas difíciles de calcular para poder obtener resultados en un tiempo aceptable. Existen diversas estrategias para esto, una de ellas es conocida como PIC (*particle in cell*) y lo que hace es definir una malla en el plano y dentro de cada celda de la malla, todos los nodos que se encuentran en ella contribuyen a la masa de la celda. De esta forma el cálculo de las fuerzas deja de ser contra todos los nodos y pasa a ser contra las celdas.

11.2.1. Variante malla de FR

Un ejemplo de lo comentado anteriormente en [FR91] es su *variante malla* (sección 4.2), que lo que hace es tomar como modelo los sistemas atómicos, con la salvedad de que en estos sistemas se busca un equilibrio dinámico (como un péndulo) y en nuestro caso se busca un equilibrio estático, por lo tanto en lugar de calcular la aceleración en cada instante de tiempo calcula la velocidad. Este es el primer método que usa simulación de N cuerpos propuesto para el problema de trazado de grafos. Sólo difiere del *spring embedder* en el cálculo de las fuerzas. Es importante recordar que cuando se hace una simulación de N cuerpos se intenta no considerar todas las fuerzas exactas, sino considerar como que hay ciertos polos de atracción.

Para el cálculo de las fuerzas repulsivas lo que hace es considerar el espacio dividido en una malla y hacer que las fuerzas repulsivas que afectan al nodo son sólo aquellas que se encuentran en alguna celda vecina a la celda correspondiente al nodo en cuestión. Para mejorar aun más esto, considerando que la longitud de cada celda de la malla sea un valor $2k$ (se considera que todas las celdas son cuadradas), solamente se calculan las fuerzas ejercidas por los nodos que se encuentren a una distancia menor o igual que $2k$. Esto hace que el cómputo sea simple y la búsqueda de estos nodos también visto que sólo se busca dentro de las celdas vecinas y se sabe de antemano qué nodos se encuentran allí (ver Figura 4.1).

Es importante considerar un valor adecuado de k (que representa la separación ideal entre vértices). El sugerido en el trabajo es:

$$k = C\sqrt{\frac{area}{n}}$$

Donde C es una constante calculada de forma experimental. Las funciones que se consideran para las fuerzas atractivas y repulsivas son las presentadas en la sección 4.2.

Es muy interesante ver cómo si la distribución de los nodos es aproximadamente uniforme la complejidad del cálculo de las fuerzas baja de $O(n^2)$ a $O(n)$. La complejidad total del algoritmo es $O(n + m)$, siendo $O(m)$ por las fuerzas atractivas y $O(n)$ por las fuerzas repulsivas.

Es también interesante ver cómo varios conceptos vistos en otros capítulos, como por ejemplo *clusters*, en el fondo pueden ser pensados como otro acercamiento al problema de N cuerpos. Los beneficios que trae el modelo de N cuerpos al

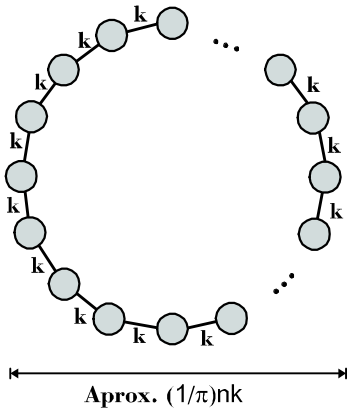


FIGURA 11.5. Ejemplo de que algunos grafos no tienen trazados buenos con una distribución uniforme de vértices.

trazado de grafos se deben básicamente a la cantidad de herramientas que ya se conocen para este problema, visto que en la física ha sido estudiado desde hace años, y aquí contamos con la ventaja de que sólo se busca un resultado estéticamente agradable y no una solución exacta, lo cual permite ciertas libertades que en la física no son admisibles.

11.2.2. Acercamiento numérico con simulación de N cuerpos

Tunkelang [Tun99b] ocho años después del trabajo presentado por Fruchterman realiza un nuevo trabajo considerando la simulación de N cuerpos. Como se dijo en el capítulo 5, lo que Tunkelang plantea es ver el problema de trazado de grafos como un problema numérico de minimización de funciones. Aquí no profundizaremos sobre cómo se optimiza utilizando gradiente conjugado (de eso se ocupa el capítulo 5), sino que veremos otra de las cosas que hace para acelerar la convergencia: modelar el problema como un sistema de N cuerpos.

Optimizando con simulación de N cuerpos. Al igual que con FR, lo que intenta hacer Tunkelang con la optimización de N cuerpos es lograr disminuir la complejidad computacional del cálculo de las fuerzas repulsivas entre los nodos. En el trabajo [Tun99b] dice que no comparte la idea de Fruchterman porque se basa en la uniformidad de los nodos en la malla, lo cual no siempre es cierto y hay casos donde por este motivo el orden es mucho mayor (ver Figura 11.5).

Para su propuesta se basa en el algoritmo Barnes-Hut [BH86] que lo que plantea es utilizar un *quad-tree* en lugar de una malla para poder optimizar los cálculos. El algoritmo consta de dos partes, la primera es la construcción del *quad-tree*, y la segunda el cálculo de las fuerzas.

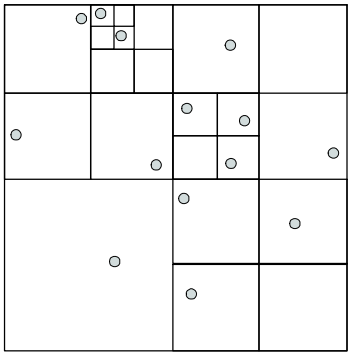


FIGURA 11.6. Partición del espacio de trabajo.

Construcción del quad-tree. Un *quad-tree* es una partición jerárquica del área en cuadrados cada vez más pequeños. La creación de esta estructura se hace de forma sencilla. Se comienza con un *quad-tree* que sólo contiene la raíz. Este árbol representa a toda el área de trabajo con un único cuadrado. El cálculo de dicho cuadrado se puede hacer fácilmente en $O(n)$. La forma de ir agregando nodos es sencilla, sólo se deben considerar tres casos.

El primer caso es cuando la celda se encuentra vacía, en dicho caso simplemente se agrega el nodo a la celda. Un ejemplo de esto es el primer nodo. El segundo caso es cuando la celda está ocupada. En dicho caso se subdivide la celda en cuatro partes iguales partiéndola horizontal y verticalmente y el caso pasa a ser del tercer tipo. El tercer y último caso es cuando la celda es un nodo interno y no una hoja, entonces lo que se hace es colocar el nodo en la subcelda que corresponda. Para entender mejor el funcionamiento ver la Figura 11.6.

Mientras se agregan los nodos no sólo se genera el *quad-tree*, sino que también se calculan los centroides de cada celda. Es importante ver que se llama celda a las hojas del quad-tree y también a los nodos internos. El cálculo del centroide se define en dos casos. En el caso en el cual la celda se encuentra vacía y se va a agregar un nodo, el centroide de dicha celda es la posición de ese nuevo nodo. Si en cambio ya hay algún nodo en la celda, el centroide es el promedio ponderado del centroide anterior y la posición del nuevo nodo.

El siguiente paso importante una vez creado el *quad-tree* es el cálculo de las fuerzas.

Cálculo de fuerzas repulsivas nodo-nodo. La optimización se realiza en esta etapa, en la cual para evitar el cálculo de todas las fuerzas repulsivas nodo-nodo, lo que se hace es considerar a los centroides como representantes de los nodos de la celda, como si el *quad-tree* agrupase por *clusters*. La forma de calcular las fuerzas es la siguiente. Primero se debe encontrar la hoja correspondiente al nodo en cuestión. Una vez encontrada la hoja se consideran sólo tres fuerzas: las de los tres cuadrados que se particionaron junto con la hoja seleccionada. Para calcular estas fuerzas se

consideran los centroides de dichas celdas. El paso siguiente es considerar las tres celdas hermanas de la celda madre de la actual y así sucesivamente.

Claramente la cantidad de fuerzas que se calculan, como siempre se utilizan los centroides, se encuentra en $O(\log n)$.

Pero esta estrategia posee un inconveniente. Cuando los nodos no se encuentran realmente cerca de los centroides, el método no provee una buena aproximación. Por este motivo se agrega una condición más. Se dice que un nodo y una celda están bien separados si $r/D < \theta$, siendo r el lado de la celda, D la distancia del nodo al centroide y θ un valor fijo entre 0 y 1; algunos autores [BH86] recomiendan utilizar valores cercanos a 1. Con esto definido se agrega la siguiente condición. Si los vértices de una celda se encuentran bien separados, entonces se considera la fuerza como si todos los nodos se encontrasen en el centroide, es decir como se dijo previamente. En caso contrario se considera, recursivamente, cada celda interior por separado. Esto permite que si los nodos no se ven bien representados por el centroide igualmente se obtenga una buena aproximación por los centroides más interiores.

Un último agregado que realiza Tunkelang es el de una fuerza centrípeta. Visto que no se podía agregar de forma inmediata, la forma de representarla fue haciendo que si una celda posee n nodos, entonces las fuerzas ejercidas por dichos nodos sean la enésima parte de las fuerzas que realmente deberían ejercer.

Como se vio, la construcción del *quad-tree* se realiza en $\Theta(n \log n)$, mientras que el cálculo de las fuerzas se encuentra en $O(\log n)$, por lo tanto el orden de complejidad final es $\Theta(n \log n)$.

11.2.3. FM^3: Fast Multipole Multilevel Method

Uno de los algoritmos más recientemente presentados para tratar grafos de gran tamaño es el llamado FM^3, presentado en [Hac04]. Este trabajo presenta una nueva variante utilizando muchas de las ideas ya presentadas pero combinándolas de forma inteligente.

El planteo general del algoritmo consiste en dividir el grafo en sucesivos subgrafos e ir optimizando en cada grafo, tal como se vio en el apartado de grafos multidimensionales.

Sistemas solares y planetas. La forma que Hachul presenta para la generación de los grafos es un modelo en el cual se consideran soles, planetas y lunas. Se selecciona un nodo al azar y éste será marcado como sol. Se eliminan los nodos que se encuentren a una distancia menor o igual que 2 de dicho sol y se repite el proceso hasta que no queden nodos. Luego se seleccionan todos los vecinos de cada sol y se los marca como planetas. Un planeta no puede orbitar alrededor de más de un sol, así que iterativamente se van marcando. Al finalizar quedarán nodos que no son ni soles ni planetas, a los cuales llamaremos lunas. A cada luna se le asigna el planeta más cercano. De esta forma un sol con sus planetas y sus respectivas

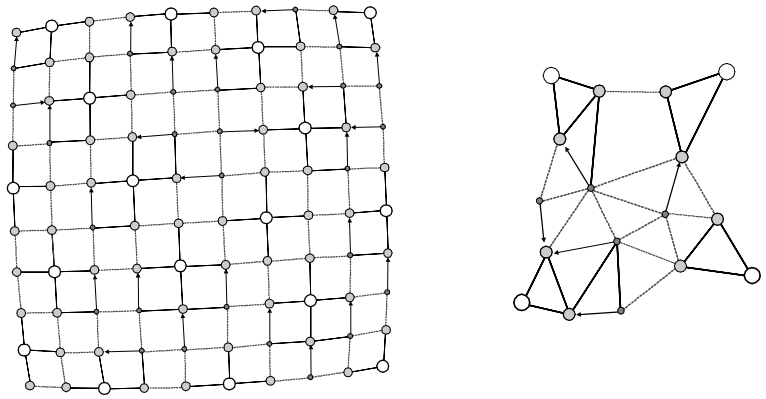


FIGURA 11.7. Dibujos de G_0 (izq.) y G_1 (der.). La malla queda particionada en 17 sistemas solares [Hac04].

lunas forman un sistema solar. La forma de generar el grafo con menor cantidad de nodos es colapsando los nodos de cada sistema solar en un único nodo.

Al igual que en [Wal03] el siguiente problema a resolver es definir la longitud ideal de la arista. Considerando que existe una arista entre los planetas u y v de longitud ideal l_{uv}, y u pertenece al sol U y v al sol V, la longitud ideal cuando colapse todo el sistema solar será $l_{Uu} + l_{uv} + l_{vV}$. Un ejemplo de este proceso se puede apreciar en la Figura 11.7 donde se genera el sistema solar a partir de una malla.

El proceso inverso para generar un grafo mayor a partir de uno menor consiste en colocar los soles donde se ubican en el grafo menor, visto que representan justamente a su sistema solar. Luego para los planetas y las lunas el procedimiento es el siguiente. Siguiendo con el ejemplo anterior supongamos que deseamos expandir v. Para esto lo colocamos sobre la recta que conecta U con V, en la posición $Pos(V) + \frac{l_{Vv}}{l_{Uu}+l_{uv}+l_{vV}}(Pos(U) - Pos(V))$. En el caso de que v pueda ser ubicado en varias posiciones según esta regla – es decir si posee aristas que lo conectan con más de un sistema solar – se considera el baricentro de todas las posibles posiciones. Continuando con nuestro ejemplo anterior podemos ver en la Figura 11.8 cómo a partir de G_2, es decir el sistema solar generado a partir de G_1, podemos volver a obtener G_1. En la figura (a) se muestra G_2. En la figura (b) se realizó el proceso inverso recién descrito obteniendo un trazado inicial para G_1. Luego en (c) se obtiene un grafo extremadamente similar a G_1.

Expansión multipolar. El siguiente paso una vez generados los grafos es el cálculo de las fuerzas para poder aplicar el algoritmo de trazado de grafos. Para hacer esto de manera eficiente lo primero que se realiza es un *quad-tree* similar al de [Tun99b], pero aquí el árbol se construye para luego poder hacer un cálculo de una expansión multipolar. La idea se basa en el siguiente teorema:

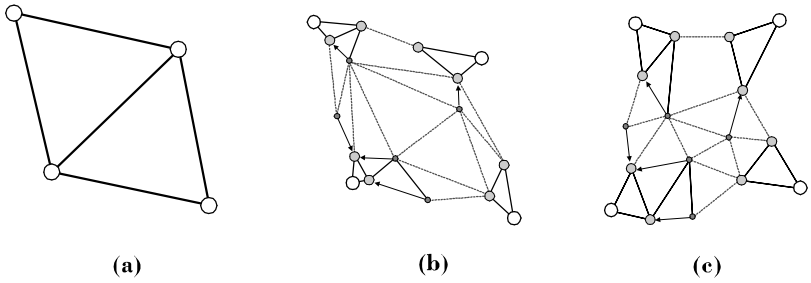


FIGURA 11.8. Continuación del ejemplo de la Figura 11.7, partiendo de G_2 (a) para generar un paso intermedio (b) y optimizando sobre éste para obtener nuevamente G_1 (c).

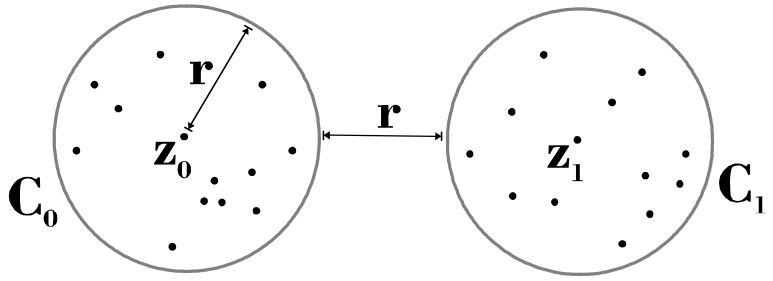


FIGURA 11.9. Uso del teorema de expansión multipolar para aproximar el cálculo de las fuerzas [Hac04].

Teorema de expansión multipolar. Sean m cargas de fuerza q_i, localizadas en un círculo de radio r y centro z_0. Para todo $z \in \mathbb{C}$ tal que $|z - z_0| > r$, la energía ejercida $\varepsilon(z)$ está dada por:

$$\varepsilon(z) = Q \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z-z_0)^k}$$

$$\text{Con } Q = \sum_{i=1}^{m} q_i \text{ y } a_k = \sum_{i=1}^{m} \frac{-q_i(z_i - z_0)^k}{k}$$

Es importante notar que este teorema se encuentra definido para el espacio complejo, por lo tanto debemos poder mapear nuestro sistema de coordenadas a los complejos. La forma de realizar esto es la trivial considerando que un punto $p = (x, y) \in \mathbb{R}^2$ se mapea a $z = x + iy$, con $z \in \mathbb{C}$. Con esto se puede utilizar esta fuerza para cada nodo. El siguiente problema es que se debe resolver la serie infinita, por este motivo se debe truncar en algún valor p. Empíricamente se observó que con $p = 4$ se obtienen buenos resultados.

Para entender la importancia de esta fuerza analicemos el ejemplo de la Figura 11.9. Supongamos m cargas que se encuentran en el círculo C_0 y m cargas en el círculo C_1. Ambos círculos son de radio r y sus centros se encuentran a distancia mayor o igual que $3r$. Con la forma común de evaluar las fuerzas, el cálculo de las

fuerzas entre estas cargas sería $O(m^2)$, pero utilizando el teorema de expansión multipolar es solamente $O(pm)$ por cada círculo, lo cual es en total $O(m)$.

Ahora se debe aplicar esto para el cálculo de las fuerzas. Primero se calcula la expansión multipolar de las hojas del árbol. Luego se recorre el árbol en forma ascendente y se van calculando las expansiones multipolares. Por último se recorre en orden descendente para calcular las fuerzas definitivas.

Lo importante de este método es que tiene un orden $O(n \log n + m)$, al cual logra llegar gracias a estas ideas, que en su mayoría son combinaciones de ideas anteriores.

Capítulo 12

Comentarios finales

Esta primera parte del trabajo ha intentado abarcar los algoritmos dirigidos por fuerzas relevantes de la literatura. Si bien la intención inicial fue hacer una revisión que incluyera a todos los trabajos sobre métodos dirigidos por fuerzas, cuando comenzamos a ver la cantidad de material disponible, y su impresionante tasa de crecimiento, quedó claro que esto no iba a ser posible del todo.

Sin embargo intentamos que el trabajo estuviera lo más cerca posible de este objetivo ideal, y creemos que esta primera parte logra reflejarlo.

Para poder brindar una visión lo más completa posible en todos los temas se presentaron los algoritmos más importantes y tradicionales, pero junto a ellos también los más recientes, que no han sido analizados en ninguna otra revisión publicada hasta ahora. Varios de los algoritmos aquí presentados serán publicados recién el próximo año o fueron publicados en los últimos meses.

Presentar los algoritmos de manera organizada no fue sencillo. Ninguna clasificación permite cubrir a todos los algoritmos existentes, y mucho menos hacer que todos caigan ordenadamente en una única categoría.

Si bien creemos que la estructura usada permitió exponer ordenadamente y con claridad la mayoría de los métodos, algunos algoritmos que nos parecieron dignos de mención no pudieron ajustarse bien a esa clasificación. Los más importantes, que no queremos dejar de mencionar, son comentados a continuación.

12.1. Otros algoritmos que vale la pena mencionar

12.1.1. Algoritmos generales

Otros algoritmos generales[1], con objetivos similares a los clásicos, han sido ideados con la intención de mejorar defectos de los primeros. Aquí comentamos sobre dos de ellos.

El algoritmo AGLO [CSP96] pretende combinar la flexibilidad de DH con la velocidad de FR usando potenciales diferenciables y aprovechando la información del gradiente para guiar la minimización de la energía. También usan temperaturas para acelerar la convergencia. La principal ventaja es que proponen combinar arbitrariamente varios criterios estéticos, permitiendo obtener trazados muy variados, a un costo no tan elevado como con DH.

Para reducir las posibilidades de caer en un trazado local pobre, Behzadi [Beh99] propone el algoritmo *CostSpring*, cuya principal diferencia con los demás

[1] Con esto nos referimos a algoritmos para trazar grafos generales, sin ningún otro objetivo o restricción.

es la función de energía que utiliza. Esta función de energía se detalla a continuación.

Primero algunas cuestiones de notación. $N(v)$ es el conjunto de vecinos de $v \in V$ (es decir, los vértices adyacentes a v). $\overline{N(v)}$ son los vértices no adyacentes a v. Como siempre, d_{uv} es la distancia en el trazado entre los vértices u y v, y además se definen $D_{\text{máx}}(v, U) = \underset{u \in U}{\text{máx}}\, d_{vu}$ y $D_{\text{mín}}(v, U) = \underset{u \in U}{\text{mín}}\, d_{vu}$ (ambos con $U \subseteq V$).

Cada vértice v tiene una energía local $Q(v)$, definida como

$$Q(v) = \frac{D_{\text{máx}}(v, N(v))}{\lambda D_{\text{mín}}(v, N(v)) + (1 - \lambda) D_{\text{mín}}(v, \overline{N(v)})}$$

Con $0 < \lambda < 1$ (la autora sugiere $\lambda = 0,5$). Los algoritmos clásicos dirigidos por fuerzas intentan obtener aristas de longitud uniforme ($D_{\text{máx}}(v, N(v)) \approx D_{\text{mín}}(v, N(v))$) e implícitamente buscan que los vértices adyacentes a v estén más cerca que los no adyacentes ($D_{\text{máx}}(v, N(v)) \leq D_{\text{mín}}(v, \overline{N(v)})$), así que trazados buenos generados con algoritmos clásicos tendrán valores pequeños para $Q(v)$. La función de energía del grafo se define como

$$E = \underset{v \in V}{\text{máx}}\, Q(v)$$

Los resultados presentados en [Beh99] indican que esta función, al menos para grafos de hasta 250 vértices, cumple con varias propiedades buenas: convergencia más rápida que las usadas en los métodos clásicos, independencia de la escala y efectividad para evitar algunos tipos de mínimos locales pobres. Hay que destacar que esta función de energía es bastante distinta a las comúnmente usadas, lo cual recuerda que todavía queda mucho para profundizar en la búsqueda de otras energías adecuadas para este problema.

12.1.2. Otros algoritmos

Se han diseñado muchos algoritmos para cumplir objetivos más específicos, que ponen un énfasis especial en algún criterio estético o facilitan el trazado de ciertos tipos de grafos.

Preservar cruces de aristas. Bertault [Ber00] quiso poder combinar las características de los algoritmos para grafos planares y de los dirigidos por fuerzas (algo parecido a lo hecho por Harel y Sardas, ver capítulo 10). El algoritmo que proponen, llamado PrEd tiene la propiedad de que preserva los cruces de aristas. Más precisamente, dos aristas se cruzarán en el trazado final de PrEd si y sólo si se cruzaban en el trazado inicial. Si bien los detalles de implementación son un poco más complicados, la idea es muy sencilla: en cada iteración del algoritmo dirigido por fuerzas (tipo *spring embedder*) un vértice es movido sólo si ese movimiento no agrega o quita cruces de aristas. Además de las fuerzas atractivas y repulsivas usuales, también aplican fuerzas repulsivas entre vértices y aristas.

Otra técnica interesante propuesta en [Ber00] es una para ir eliminando los cruces de un trazado. Si en medio del algoritmo PrEd se detecta que el movimiento

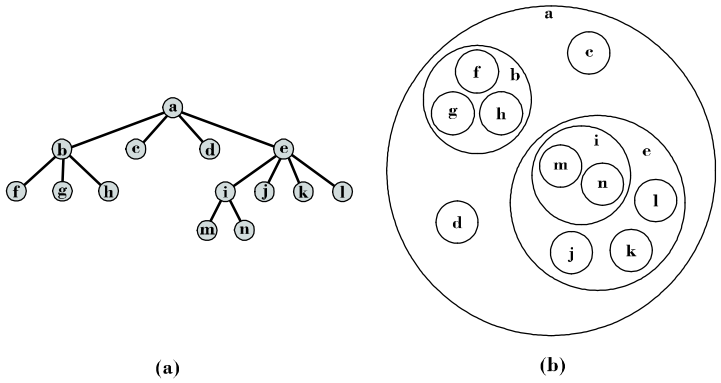


FIGURA 12.1. Un árbol representado de dos formas. De la manera más usual (izq.) y como una relación de contención (der.).

de un vértice puede eliminar un cruce, se lo mueve sólo si el número total de cruces disminuye (para evitar que la eliminación de ese cruce introduzca nuevos). Esta simple heurística es efectiva pero tiene como desventaja un costo computacional muy alto (al menos implementándola directamente).

Dibujar subgrafos isomorfos. Bachl y Brandenburg en [BB02] combinan heurísticas para identificar subgrafos isomorfos con un algoritmo de trazado basado en GEM para poder hacer que dos subgrafos isomorfos se dibujen de manera idéntica. Esto es implementado moviendo los vértices en los dos subgrafos de igual manera y agregando una fuerza repulsiva para separar a los subgrafos isomorfos.

Resaltar la simetría. Que los algoritmos dirigidos por fuerzas tienden a exponer la simetría presente en los grafos ha sido comprobado formalmente [EL00]. Aprovechando esto, Chuang y Yen [CY02] proponen un algoritmo que prioriza aun más este criterio estético. El algoritmo está basado en la noción de "simetría cercana", que mide el grado de simetría de un grafo en función de cuántas operaciones básicas (como la contracción de aristas) se necesitan para llevar un grafo a otro simétrico (se considera sólo simetría axial o rotacional). La esencia del algoritmo propuesto es llevar el grafo a su versión simétrica, trazarlo con un algoritmo dirigido por fuerzas (FR), y agregar a este trazado las aristas quitadas al comienzo.

Dibujar árboles. Como ya se ha mencionado, los árboles son una de las peores entradas para algoritmos dirigidos por fuerzas. Sin embargo, recientemente Sun et al. [SSC03] han propuesto un algoritmo para trazar árboles (con raíz) que da buenos resultados, con el detalle de que los árboles no son trazados de acuerdo a la convención más usual (la descripta en el capítulo 2) sino que son dibujados representados como una relación de contención. La Figura 12.1 muestra un ejemplo. De esta forma cada vértice se dibuja como un disco, y sus hijos como discos disjuntos dentro de él. Luego esto se repite para cada uno de sus hijos.

El algoritmo es dirigido por fuerzas y recursivo. Trata un nivel del árbol por vez. El objetivo es ubicar los discos dentro del disco mayor de una manera estéticamente buena. El diámetro de cada disco está definido por el tamaño del subárbol que

tiene a ese vértice como raíz, así que – como siempre – el algoritmo sólo debe determinar la posición de cada vértice dentro del disco que lo contiene (excepto para la raíz).

El modelo físico está compuesto por dos clases de fuerzas, una para mantener cierta distancia ideal entre los discos y otra para atraerlos hacia el centro. Para modelar el diámetro de los vértices (o de los discos) se usan puntos con masa proporcional al diámetro.

Es interesante mencionar que la fuerza que usan para mantener la distancia ideal entre los vértices se basa en el potencial de Lennard-Jones:

$$\phi_{LJ}(u,v) = 4\varepsilon \left[\left(\frac{\sigma}{d_{uv}} \right)^{12} - \left(\frac{\sigma}{d_{uv}} \right)^{6} \right]$$

donde ε y σ son constantes y d_{uv} es la distancia en el trazado entre u y v. Como siempre, para obtener la fuerza se deriva el potencial en función de d_{uv}. Este potencial tiende a infinito cuando $d_{uv} \to 0$ y tiene su único mínimo en $d_{uv} = \sqrt[6]{2}\sigma$.

Las pruebas presentadas en [SSC03] muestran que el algoritmo obtiene muy buenos resultados y con buena complejidad: levemente peor que lineal.

Parte II

Algoritmos para grafos donde los vértices son regiones geográficas

Capítulo 13

Introducción

Supongamos que se necesita visualizar información relacional acerca de las oficinas de una empresa. Por ejemplo, se desea mostrar qué oficinas de un edificio en particular colaboran entre sí, para estudiar la forma en la que los empleados interactúan. La información puede ser modelada como un grafo, donde cada oficina tiene un vértice asociado y dos vértices están conectados cuando las oficinas colaboran entre sí.

A la hora de visualizar este grafo de colaboraciones lo ideal sería que la posición de los vértices permita asociarlos fácilmente con la oficina a la que corresponden. Una buena forma de lograr esto es dibujando el plano del edificio de fondo y haciendo que cada vértice esté confinado a la oficina que representa. Además esto tiene la ventaja adicional de permitir ver rápidamente factores como si existe relación entre la cantidad de colaboraciones entre oficinas y su cercanía geográfica, que pueden ser de interés.

Desde el punto de vista que aquí nos interesa, el del trazado de grafos, estamos frente a un problema estándar de trazado con dos restricciones adicionales: los vértices deben estar dentro de sus respectivas regiones, y además cada vértice debe representar a su región de la mejor manera posible (llamaremos a este tipo de restricciones, restricciones *geográficas*).

No es difícil encontrar otros ejemplos donde surge este mismo problema. Un área en la que aparece constantemente es en la visualización de redes de datos. Un ejemplo real puede verse en la Figura 13.1, perteneciente a la red IRIS, que interconecta universidades y centros de investigación de España. La figura muestra los enlaces de datos entre cada Comunidad Autónoma española. Claramente los vértices tienen asociados información geográfica, con la que el trazado debe poder ser coherente. Esto agrega a los criterios estéticos usuales del trazado de grafos la restricción de que cada vértice no salga de su comunidad asociada.

En esta segunda parte del trabajo nos ocuparemos de este problema, que puede ser enunciado más formalmente de la siguiente manera:

Enunciado del problema. Dado un grafo $G = (V, E)$ y un conjunto de regiones R, donde $r_v \in R$ es la región (no vacía) del plano *representada* por el vértice $v \in V$, encontrar un trazado de G en el plano que sea estéticamente bueno y donde $p_v \in r_v$ para todo vértice $v \in V$ (p_v es la posición de v en el trazado).

Enfatizamos el hecho de que cada vértice *representa* a una región para marcar que los vértices no sólo deben ubicarse dentro de las regiones, sino que deben representarlas. Recordemos de los ejemplos anteriores que los vértices modelaban las oficinas y las autonomías, es decir que modelaban a las misma regiones, y por

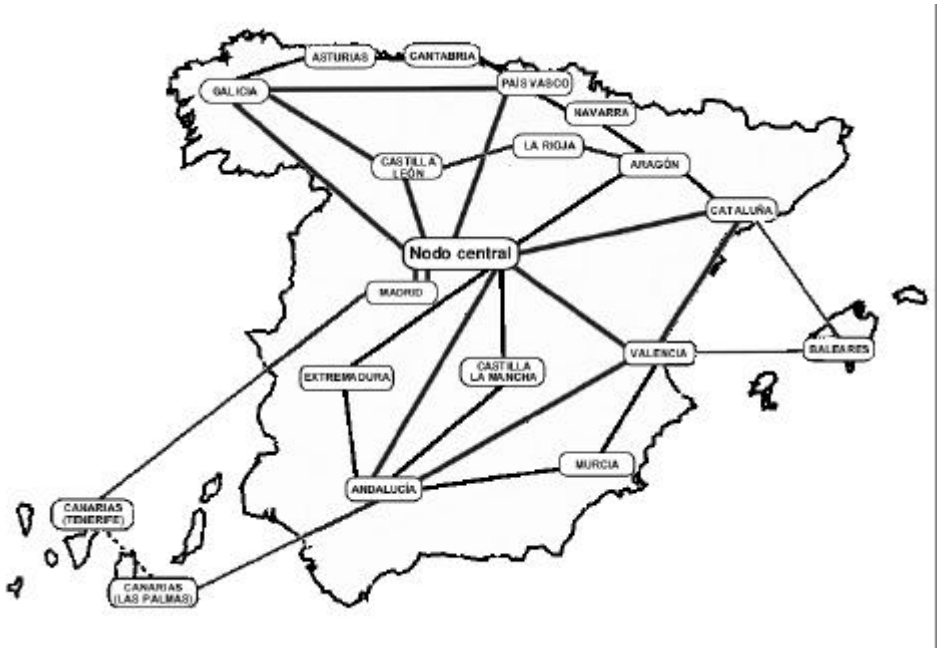


FIGURA 13.1. Una figura ilustrando la Red IRIS, extraída de `www.rediris.es`.

ende, las aristas del trazado deben dar la sensación de ser aristas entre las regiones, más que entre los vértices. Este punto quedará más claro en el próximo capítulo.

Desde otro punto de vista, el problema puede ser visto como uno en el que los vértices tienen formas arbitrarias (de regiones), y están fijos. El problema pasa a ser de qué punto hacer partir las aristas.

A lo largo de este trabajo, teniendo en mente aplicaciones como las descriptas arriba, supondremos que las regiones son disjuntas (aunque haremos algunas consideraciones sobre el caso en el que hay solapamiento). Además, si bien cualquier tipo de región es admisible, la mayoría del tiempo trabajaremos con polígonos, ya que otras regiones pueden ser aproximadas con ellos, y también consideraremos brevemente algunas regiones más particulares como discos y segmentos de recta.

13.1. Trabajo previo

No hemos encontrado en la literatura antecedentes donde se tratara con este tipo específico de restricción en las posiciones, y sobre todo, que analicen este problema de visualización en su conjunto: vértice que representan regiones (que, como veremos, es más que vértices que deben estar confinados al interior de una región).

Como se vio en el capítulo 8, sí existen muchos trabajos que contemplan la inclusión de restricciones de otros tipos en la posición de los vértices.

Los tipos de restricciones más comunes que permiten agregar son restricciones absolutas o relativas (respecto de otros nodos) en las posiciones de los vértices [KKR95], [WM95]. Algunos trabajos permiten incluir restricciones lineales arbi-

trarias [HM98], o inclusive cualquier tipo de restricción no lineal [HMMS02]. Estos últimos se valen de métodos específicos para trabajar con restricciones, como *solvers* especializados o métodos generales de optimización.

Las restricciones con las que nos enfrentamos en este problema no son de posiciones absolutas, y en la mayoría de los casos tampoco pueden ser modeladas con restricciones de posiciones relativas, excepto cuando las regiones son discos. En ese caso sí es posible ubicar un vértice fijo en el centro de cada disco y agregar a cada vértice la restricción de que su distancia al vértice del centro sea menor al radio del disco.

Si las regiones son polígonos, que es el caso que más nos interesa, entonces si cada polígono es convexo, podrá ser formulado como un problema con restricciones lineales (ya que cada región sería un poliedro), y se podría usar un algoritmo como el de [HM98].

Regiones cóncavas sólo podrían ser tratadas con algoritmos generales como el de [HMMS02]. Ostry [Ost96] propone algunas adaptaciones para que los vértices (todos) estén restringidos a algunas superficies tridimensionales, pero a excepción de cuando las regiones son segmentos de recta, no pueden adaptarse fácilmente a nuestro problema.

De todas formas, usar estos algoritmos sería usar herramientas demasiado generales para un problema muy específico y determinado. Nuestras restricciones son muy sencillas y estructuradas: ningún vértice tiene permitido salir de su región. Debido a esto, es posible tomar un camino mucho más sencillo y hacer lo que sugieren DiBattista et al. en [DETT99]: adaptar un algoritmo iterativo para que los movimientos de los vértices en cada iteración estén restringidos al interior de su región. Como se verá en el próximo capítulo, esto no es suficiente para obtener buenos resultados, debido a que el hecho de que cada vértice represente a una región implica restricciones estéticas que van más allá de limitar su posición a un área del plano.

13.2. Organización de la segunda parte

El próximo capítulo analiza los criterios estéticos que surgen en este problema con restricciones, revisando los criterios usuales y los específicos de trabajar con regiones. En el capítulo siguiente se presentan algoritmos para el trazado de grafos con regiones, y en el siguiente los resultados obtenidos al implementarlos. Finalmente, el capítulo 17 provee un cierre a ambas partes del trabajo, señalando además algunas direcciones de trabajo futuro.

Capítulo 14

Criterios estéticos

El problema planteado presenta dos aspectos. Por un lado es necesario que la posición de los vértices esté limitada a su región. Por otro lado, como en toda aplicación de trazado de grafos, se busca que el trazado sea (estéticamente) bueno: esto significa – principalmente – que la información se transmita de la manera más clara posible. Cuando los vértices representan a regiones geográficas, este segundo aspecto cobra especial importancia, aunque en una primera aproximación pueda parecer que los criterios usuales del trazado de grafos son suficientes para obtener resultados de calidad. Veremos en este capítulo que esto no es así. Comenzaremos mostrando por qué es necesario redefinir los criterios estéticos en juego.

14.1. Una primera solución

A simple vista puede parecer que lo único necesario para abordar este problema es implementar el primer aspecto, que los vértices no queden fuera de sus regiones.

Esta es una opción tentadora porque es muy fácil de implementar. Si usamos un algoritmo del tipo del *Spring Embedder*, alcanza con controlar en cada paso que la posición elegida para un vértice dado no salga afuera de su región.

Si bien esto será suficiente para obtener un trazado factible (que cumple con las restricciones) no cuesta mucho ver que el mismo método que antes obtenía trazados estéticamente buenos, ya no lo hace.

Un ejemplo se puede observar en la Figura 14.1, donde un sencillo grafo ha sido trazado usando un algoritmo clásico limitando los movimientos. Si las regiones no estuvieran, el trazado sería bueno. Pero con las regiones, es evidente que no lo es. Principalmente, no está claro qué regiones están conectadas, que es lo que debe transmitir el trazado. El trazado de la Figura 14.2, en cambio, es mucho más efectivo en esto.

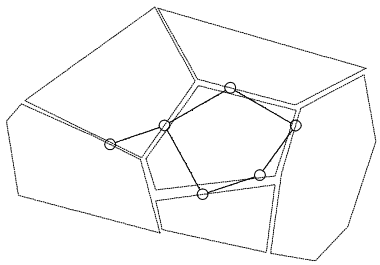


Figura 14.1. Restringir las posiciones de los vértices no es suficiente para obtener buenos trazados.

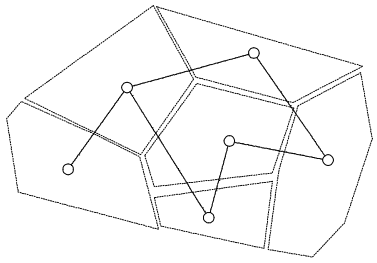


FIGURA 14.2. Un trazado que muestra mejor las relaciones entre las regiones.

Este ejemplo debe ser suficiente para ilustrar que **trazados buenos producidos considerando sólo las restricciones en las posiciones de los vértices no necesariamente son trazados buenos para nuestro problema**.

Esta simple observación es de suma importancia, ya que hace que sea necesario redefinir qué es un trazado bueno en nuestro problema, y al mismo tiempo hace que los algoritmos existentes no puedan ser aplicados directamente. En este capítulo exploraremos cuáles son los criterios estéticos que tienen valor en nuestro contexto.

14.2. Análisis de criterios estéticos previos

Comenzaremos revisando los criterios estéticos que rigen en el problema de trazado de grafos, para evaluar cuáles tienen sentido para nuestro problema[1] (los principales ya fueron comentados en el capítulo 2).

- **Minimizar número de cruces entre aristas**. Este criterio sigue siendo válido, y tiene tanta o más importancia que antes. Será considerado en nuestros algoritmos, presentados en el próximo capítulo.

- **Maximizar el ángulo entre aristas adyacentes**. Este criterio sigue siendo válido, pero no de la misma forma que en el problema general (ver comentarios en el capítulo 17).

- **Maximizar el ángulo entre aristas que se cruzan**. Este criterio equivale a evitar aristas paralelas. La motivación es que si dos aristas van a cruzarse, lo hagan lo más perpendicular posible, de manera de que el cruce sea claro para el usuario. También será tratado posteriormente.

- **Mostrar simetría**. Sigue siendo válido, pero en general la posible simetría a mostrar está muy reducida, debido a que las posiciones de los vértices están muy limitadas por sus regiones. Un factor muy importante de este problema es que la topología del trazado está determinada en gran parte por las mismas regiones, así que de éstas depende, principalmente, que pueda mostrarse simetría.

[1] A menos que se aclare lo contrario, supondremos siempre que las regiones son polígonos disjuntos.

- **Distribución uniforme de vértices**. En principio no es válido, ya que la posición de los vértices se limita a sus respectivas regiones. Puede ser adaptado a algo más débil, como "intentar que los vértices estén distribuidos lo más uniformemente posible", pero nunca tendrá la importancia que tiene en los trazados comunes.

- **Longitud uniforme de aristas**. En principio no es válido, por lo mismo que el anterior. Sin embargo, sí sigue siendo válido el criterio de que **vértices conectados deben dibujarse cerca**, lo cual implica que debe definirse una longitud ideal para las aristas. Pero entonces, ¿cuál es esta longitud ideal? Este es un punto importante que será considerado en detalle más adelante, pero está claro que la distancia entre las regiones de los respectivos vértices debe ser un ingrediente de esta longitud ideal. De todas formas, es claro que no es un tema simple, ya que si dos regiones se encuentran cercanas y dos regiones muy alejadas, en el primer caso la longitud ideal será pequeña mientras que en el segundo será mayor. En el próximo capítulo presentaremos varias soluciones que encontramos e implementamos para este criterio.

- **Vértices no conectados no deben dibujarse muy cerca unos de otros**. Sigue siendo válido, pero podrá ser satisfecho sólo si las regiones son suficientemente grandes como para permitir movilidad de los vértices. Supondremos que para muchos de los vértices esto es así, ya que si no tienen libertad de movimiento no tiene sentido el problema de trazado de grafos.

- **Area del dibujo lo más pequeña posible**. No es válido ya que el área queda determinada de antemano por las regiones. Un criterio alternativo que puede plantearse es que **el área ocupada por los vértices sea lo más pequeña posible**. Esto en principio no es un criterio válido, ya que si se pretende que cada vértice represente a su región, de alguna manera debe estar contemplada la geometría y el tamaño de la misma. Intentar concentrar los vértices lo más posible ignora el tamaño de las regiones, y creemos que esto no es beneficioso. Un ejemplo de esto puede verse en la Figura 14.3.

- **Minimizar la suma de las longitudes de las aristas**. No es válido. En el problema general de trazado de grafos minimizar las longitudes de las aristas está bien, dado que aristas largas son difíciles de seguir con la vista. Sin embargo en el caso de las regiones, aunque sigue siendo cierto que aristas largas no son deseables, las aristas cortas se acercan mucho a los bordes de las regiones y son propensas a confundir. Veremos más sobre esto en las próximas secciones.

- **Minimizar la longitud máxima de las aristas.** No es válido por los mismo motivos que el anterior.

- **Los nodos deben estar cerca del centro del área de trabajo**. No es válida. En el problema general, este criterio hace que los vértices no se alejen mucho y permanezcan cerca, pero en el problema actual el centro

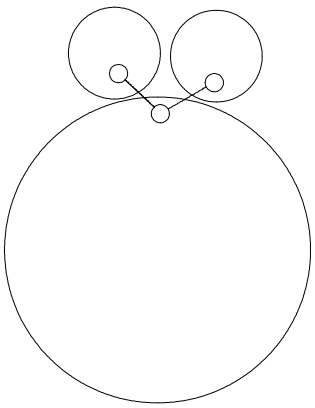


FIGURA 14.3. El área del dibujo queda determinada por las regiones.

del área de trabajo deja de tener importancia, ya que el movimiento de los vértices queda limitado por sus regiones. Un criterio que puede ser visto como relacionado es que cada nodo esté cerca del centro de su región, sobre el cual profundizaremos más abajo.

- **Los nodos no deben estar cerca de las aristas**. Sigue siendo válido. Este criterio está porque si un nodo se encuentra muy cerca de una arista se dificulta verlo, y puede causar la sensación de que hay aristas donde no las hay, lo cual no favorece al entendimiento del trazado.

Es importante destacar que de todos estos criterios válidos para el problema general, son unos pocos los que son atacados explícitamente por los algoritmos de trazado de grafos. Los algoritmos derivados del *spring embedder* buscan distribución uniforme de vértices y longitud de aristas uniforme, al igual que KK (aunque indirectamente). Tu y DH intentan minimizar los cruces de aristas y DH además intenta alejar a los vértices de las aristas. Lo que queremos resaltar con esto es que para nuestro problema también es conveniente concentrar la atención de los algoritmos en unos pocos criterios estéticos, en lugar de intentar abarcarlos todos, porque, como ya se explicó, es muy fácil que entren en conflicto.

14.3. Nuevos criterios estéticos

En la sección anterior se vio que varios de los criterios estéticos del problema general de trazado de grafos siguen siendo válidos para nuestro problema particular. Las pruebas que realizamos para determinar qué criterios eran mejores (sobre las cuales detallaremos más en los próximos capítulos) mostraron que de los criterios anteriores los más determinantes de la calidad del trazado eran el de la longitud de las aristas y el de minimización de cruces. Esto no es sorprendente, ya que estos también son de los criterios más importantes en el problema general.

Sin embargo, notamos que éstos dos no eran suficientes para producir trazados buenos. Un ejemplo puede verse en la Figura 14.4. La longitud de las aristas parece

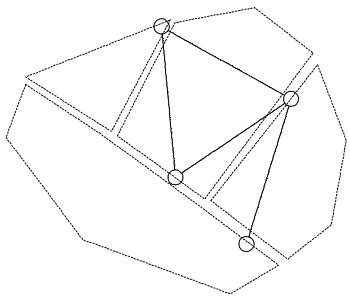


FIGURA 14.4. Son necesarios nuevos criterios estéticos para este problema.

ser buena, inclusive está cerca de uniforme, y no hay cruces. Sin embargo, sigue sin ser un trazado muy atractivo en nuestro contexto.

Resultados como éste motivaron la búsqueda de nuevos criterios que permitieran obtener trazados aun mejores. Los que nos parecieron más importantes fueron dos, y se explican a continuación.

Los vértices deben estar ubicados cerca del centro de sus regiones

El primer criterio que proponemos es que los vértices intenten estar cerca del centro de sus regiones. Son varios los motivos por los que creemos que éste es un criterio importante:

- **Vértices en el centro representan bien a sus regiones**. Ya dijimos que el vértice debe representar a la región en la que está ubicado. Esto significa que la posición del vértice debe permitir identificar lo mejor posible a la región. El centro de la región es, en general, el punto que mejor la representa.
- **Vértices cerca de los bordes se prestan a confusión**. Ya que están muy cerca de varias regiones, y puede ser difícil identificar a cuál pertenece (en relación a esto proponemos también el criterio que sigue).

Además, en muchos casos, cuando la distancia entre el centro y los bordes es aproximadamente uniforme, esto permite que la longitud de las aristas dentro de la región sea también uniforme, lo cual es también beneficioso.

En la Figura 14.2 puede apreciarse que los vértices cerca de los centros ayudan a producir buenos trazados.

Los vértices no deben estar cerca de los bordes

El segundo criterio significativo es que los vértices no estén cerca de los bordes. Si bien el criterio anterior suele implicar a éste, creemos que es un criterio en sí mismo (y puede valer la pena considerarlo sin el anterior).

La motivación de este criterio es que cuando los vértices están muy cerca de los bordes cuesta distinguir a qué región pertenecen y muchas veces las aristas

quedan casi paralelas a los bordes de la región, por lo que se hace complicado distinguirlas. Es un problema similar al criterio que indica que no es bueno que los nodos se encuentren muy cerca de las aristas. En la Figura 14.1 puede verse cómo el hecho de que los vértices estén muy cerca de los bordes dificulta determinar cuál es la región de cada vértice.

Si bien la elección y conveniencia de estos criterios es subjetiva y discutible, como lo es con todos los otros criterios estéticos que aparecen en el trazado de grafos, las pruebas que hemos hecho nos hacen pensar que son acertados. En el capítulo 16 se presentan algunas de ellas.

14.3.1. Criterios para cuando las regiones son segmentos

Los criterios anteriores fueron pensados para cuando las regiones son polígonos. Entre otros casos particulares, es de interés considerar qué sucede cuando las regiones están definidas por segmentos de recta (o cadenas de segmentos), ya que existen algunas aplicaciones donde puede ser útil.

Un ejemplo es si el trazado es parte de un mapa de carreteras y los vértices representan tramos de la carretera, lo que los restringe a estar "a lo largo" de las mismas.

En este trabajo abordaremos sólo el caso más simple en el que cada región es un único segmento de recta.

Los criterios estéticos propios para este caso no son muy claros. Nuevamente el centro del segmento es una posición representativa de la región, pero no parece tener tanta influencia como con los polígonos. El que con seguridad cobra especial valor es que los vértices no estén cerca de los extremos del segmento. Esto puede verse como la versión "unidimensional" del criterio anterior que indica que los vértices no deben estar cerca de los bordes de sus regiones.

Capítulo 15

Algoritmos propuestos

Ahora que han sido definidos los criterios estéticos válidos en este nuevo problema, propondremos algoritmos dirigidos por fuerzas que intentan obtener trazados estéticamente buenos.

Dada la flexibilidad que demuestran tener los algoritmos dirigidos por fuerzas, no parece una idea apropiada crear un nuevo algoritmo desde cero dedicado al problema de las regiones, sino que es conveniente adaptar alguno de los ya existentes a la nueva situación. Por ser de los más representativos, decidimos utilizar DH y el *spring embedder* (SE) para nuestras pruebas, aunque los conceptos son extensibles a otros. En el caso particular de segmentos también utilizaremos KK, ya que nos permite integrar en el modelo la restricción de que los vértices deben estar sobre segmentos.

El primer cambio que se debió realizar para que los algoritmos clásicos se pudieran aplicar a nuestro problema fue que los movimientos de los nodos se restrinjan a sus regiones. Como ya se mencionó, esto es muy fácil de hacer. Nuestra solución fue mantener la estructura del algoritmo y cuando un nodo intenta realizar un movimiento fuera de la región, prohibirlo. La primer forma de prohibir esto fue simplemente no realizar el movimiento en estos casos. Pero esto no da buenos resultados visto que es muy fácil que los nodos se traben al intentar salir de las regiones, haciendo que el algoritmo no pueda avanzar y se mantenga siempre en un mismo lugar.

Es importante también considerar que si bien nos basamos en un algoritmo dirigido por fuerzas, esta solución no respeta la física del problema. En el caso en que una pared diagonal tenga un nodo y una fuerza lo empuje hacia la pared, el nodo no se quedará quieto, sino que tenderá a deslizarse sobre ésta[1].

Lo que se hizo para que el algoritmo modelara mejor la realidad fue en lugar de no hacer el movimiento, realizarlo, pero luego proyectarlo al punto más cercano de la región (Figura 15.1). Esto permitió que los nodos puedan moverse bordeando las regiones para de esta forma buscar un mejor equilibrio.

Implementamos esta estrategia adaptando el *spring embedder* y DH, obteniendo de esta forma una primer aproximación que solucione nuestro problema. Como era de esperar, por lo expuesto en el capítulo anterior, los resultados obtenidos no fueron buenos, pero es importante contar con estos para poder tener algún punto de referencia con quien comparar los resultados generados por las siguientes soluciones.

Una vez obtenida esta primera solución al problema, lo que sigue es incorporar los criterios estéticos adecuados para poder obtener los resultados deseados. Con-

[1]Excepto cuando la dirección de la fuerza es perpendicular a la pared.

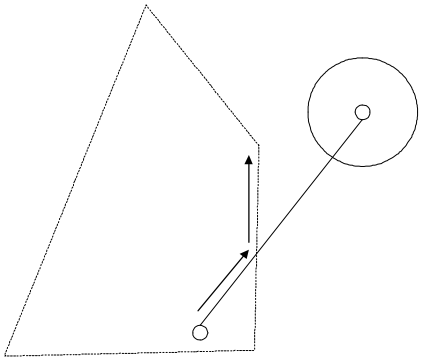


FIGURA 15.1. Proyección de los movimientos sobre la región.

siderando los criterios del capítulo anterior, definimos e implementamos nuestra solución para los que consideramos más relevantes. En las próximas secciones presentaremos los cambios realizados a los algoritmos clásicos y algunos resultados obtenidos. Repasaremos uno a uno los principales criterios que decidimos atacar y qué algoritmos proponemos para cada uno.

15.1. Longitud ideal de las aristas

El primer criterio estético que consideramos fue la longitud de las aristas. Este criterio es uno de los más importantes (y atacados) en el problema general; no hay ningún algoritmo que no lo considere de una forma u otra. Luego de realizar pruebas con el algoritmo simple de la sección anterior, analizamos los resultados obtenidos y fue clara la necesidad de definir una longitud ideal de las aristas que contemple las regiones.

15.1.1. Longitudes fijas

En el problema general de trazado de grafos la longitud de las aristas suele ser una constante, visto que no hay aristas que por algún motivo deban ser más largas que otras, pero las limitaciones de regiones imponen ciertas longitudes mínimas (y máximas) para nuestro problema en particular.

Nuestro primer intento fue considerar una distancia fija, pero que sea acorde a las regiones. De manera similar a lo hecho en [BF95] para vértices con distinto tamaño, definimos la distancia ideal de las aristas como el promedio de las distancias entre los centros de las regiones:

$$K = \frac{1}{|E|} \sum_{(u,v)\in E} \|c_u - c_v\|_2$$

donde $c_u \in \mathbb{R}^2$ es el centro[2] de la región r_u.

[2]La definición precisa de centro se presenta más abajo.

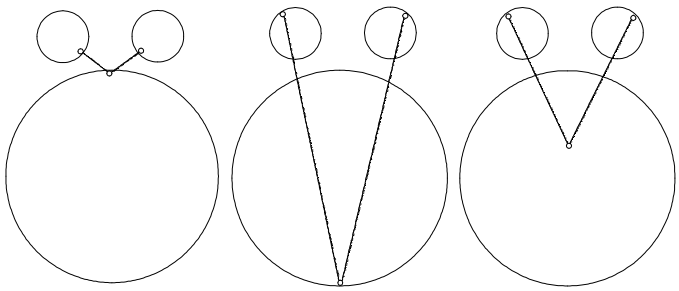


FIGURA 15.2. Trazados utilizando como longitud ideal las distancias mínima, máxima y promedio (de izq. a der.).

Los resultados obtenidos no fueron satisfactorios, ya que a diferencia de lo que ocurre en el problema general, en nuestro caso no es natural considerar que la longitud ideal de todas las aristas sea igual, visto que cada una conecta dos regiones que tienen una sección del plano ya establecida y fija. Pretender uniformizar esta longitud cuando la distancia que separa a las regiones no es uniforme resulta contraproducente.

Otras distancias fijas que fueron probadas fueron la distancia mínima o máxima entre regiones, considerando la distancia mínima como la distancia entre los dos puntos más cercanos de cada región y la máxima lo contrario. Implementamos esta solución, pero los resultados de utilizar estas longitudes no fueron buenos, visto que la topología del grafo era casi ignorada llevando los nodos a las fronteras de las regiones. Esto se debe a que utilizar la distancia mínima es casi equivalente a considerar cero como longitud ideal de las aristas[3]. En el caso de la longitud máxima pasa justamente lo contrario pero produciendo el mismo efecto, al tender a ser de gran tamaño la arista, hay pocas formas de acomodarla y que queden ambos extremos dentro de las regiones, por lo tanto también termina fijando los nodos a las fronteras, aunque en este caso a las fronteras opuestas.

En la Figura 15.2 pueden verse ejemplos obtenidos con las distancias fijas anteriores.

15.1.2. Longitudes variables

Las consideraciones previas nos llevaron a buscar una distancia ideal individual para cada arista. La primer opción fue utilizar como longitud ideal de cada arista (u, v) la distancia entre sus centros:

$$K_{uv} = \|c_u - c_v\|_2$$

En este punto se hace necesario definir qué punto elegir como centro de la región. Hay varias opciones posibles, por ejemplo puede verse que Carl Kimberling

[3]No es exactamente lo mismo pero los resultados finales sí serán iguales.

tiene registrados 1.477 "centros" diferentes para triángulos[4], así que la cantidad que podría definirse para polígonos es aun mayor. El que hemos elegido para nuestras implementaciones es el centro de gravedad o centroide. El mismo es adecuado porque es independiente de la densidad de los vértices del polígono y porque puede ser calculado de manera eficiente. El algoritmo que usamos para su cómputo es el siguiente:

centroRegión(RegiónPoligonal r)
0. Inicializar suma=0
1. Sean $\{p_0,..,p_k\}$ los vértices que definen r (en ese orden)
2. Para i=1 hasta k-1
    2.1. a = área del triángulo definido por $\{p_0, p_i, p_{i+1}\}$
    2.2. $\overline{p}=\frac{1}{3}(p_0 + p_i + p_{i+1})$ //centroide del triángulo
    2.3. suma = suma + a.$\overline{p}$
3. Tomar centro = suma/(área total región)

Se omiten los detalles para cuando la región tiene menos de 3 vértices. Es importante señalar que este centro está pensado para regiones convexas. De todas formas, en polígonos que no son convexos pero tampoco son "muy" cóncavos (es decir cuando la envolvente convexa del polígono no tiene forma muy distinta al polígono mismo) igualmente es útil.

Al introducir esta distancia como longitud ideal de aristas, las mejoras empezaron a hacerse visibles, obteniendo trazados estéticamente agradables y sin tener vértices sujetos a los bordes de las regiones. Implementamos estas modificaciones en el *spring embedder* y en DH obteniendo en ambos buenos resultados. Un ejemplo se puede ver en la Figura 15.3.

Si bien la introducción de esta distancia mejoró notablemente los resultados, notamos que a veces los nodos tendían con demasiada fuerza a ubicarse en los centros de sus regiones o contrariamente, para los nodos de las regiones más exteriores – es decir aquellas que comparten caras con el área de trabajo no utilizada – ésta longitud a veces resultaba demasiado grande. Por este motivo decidimos adoptar una longitud de compromiso, que mantenga las virtudes de la distancia entre centros pero permita también que los nodos se acerquen más cuando sea necesario. Esta longitud ideal fue definida como

$$K_{uv} = \lambda \left\| c_u - c_v \right\|_2 + (1 - \lambda) \left\| r_u - r_v \right\|_2 \tag{15.1}$$

donde $\lambda \in (0, 1)$ y $\|r_u - r_v\|_2$ es la distancia mínima entre las dos regiones. En nuestras pruebas utilizamos mayoritariamente $\lambda = 0,5$, es decir, el promedio entre la distancia entre los centros de las regiones y su distancia mínima. Este parámetro puede variarse para darle más importancia a una de las dos distancias.

[4]Para mayor información referirse a Carl Kimberling: http://mathworld.wolfram.com/KimberlingCenter.html.

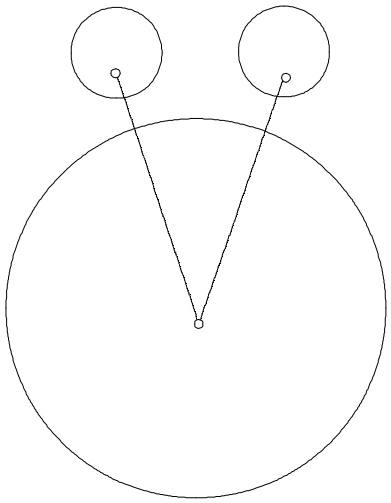


FIGURA 15.3. Trazado obtenido usando como distancia ideal la distancia entre centros.

Esta nueva medida permitió que los nodos se atraigan mutuamente con mayor fuerza pero sin llegar a juntarlos a los bordes ni quitarles tanta libertad de movimiento.

Los resultados fueron ampliamente mejores que los anteriores, marcando en la gran mayoría de los casos una muy buena topología para el trazado resultante. Nuevamente llevamos este cambio a SE y DH dando en ambos casos buenos resultados. Un ejemplo puede verse en la Figura 15.4, donde se logró reducir la longitud de las aristas (en relación a la figura anterior), manteniendo la calidad del trazado. En el capítulo 16 se presentan más resultados.

Por los resultados obtenidos consideramos que para el criterio de longitud de aristas, **la mejor longitud ideal que encontramos fue la distancia de compromiso**.

Habiendo definido ya una longitud ideal para las aristas, el siguiente paso es incorporar a nuestros algoritmos los demás criterios.

15.2. Fuerzas hacia el centro

El siguiente criterio a considerar fue que los vértices deben estar cerca del centro de sus regiones.

Como ya se mencionó, este criterio tiene dos motivaciones: que los vértices estén cerca del centro y que no se encuentren muy cerca de un borde.

Para poder acercar los nodos al centro decidimos agregar al *spring embedder* una fuerza de atracción centrípeta que evitara que los nodos se alejen mucho del centro.

De esta manera, las fuerzas en juego son las atractivas, las repulsivas y las centrípetas, definidas a continuación.

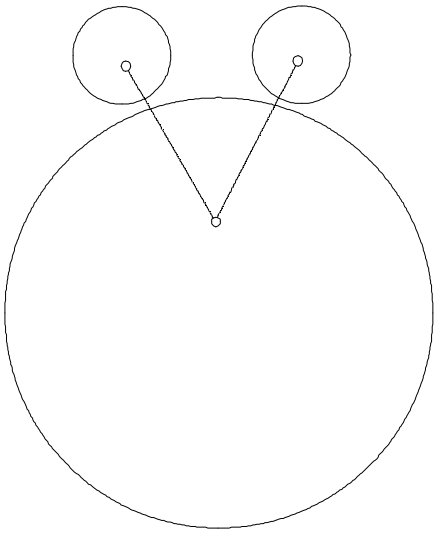


FIGURA 15.4. Trazado usando la distancia de compromiso.

Fuerzas centrípetas
Definidas para cada vértice $v \in V$

$$f_c(v) = C_1 \log(\frac{d_{c_v v}}{\varepsilon})\overrightarrow{p_v c_v} \tag{15.2}$$

donde c_v es el centro de la región de v, $\varepsilon < 1$ y C_1 constantes. Esto puede ser pensado como agregar un nodo ficticio ubicado en el centro de cada región, con una única arista incidente que lo conecta con el nodo perteneciente a dicha región.

La longitud ideal de la arista que conecta dichos nodos es un valor ε muy pequeño (porque idealmente debería ser cero, pero el cociente no lo permite), para que de esta forma el nodo tenga una atracción hacia el centro.

Podría haberse usado una fuerza de otro tipo, por ejemplo, una fuerza lineal, en lugar de una logarítmica, pero en un principio preferimos esta última para que esta arista ficticia esté en igual condiciones que las aristas reales.

Esto mismo motivó la elección del valor de la constante C_1. En una primera instancia consideramos que debería tener su propia constante de fuerza, pero los resultados de esto fueron que cuando el valor de C_1 era mayor que para las demás aristas, los nodos tendían a alejarse muy poco de sus correspondientes centros, mientras que cuando el valor era menor, la fuerza era superada ampliamente lo cual hacía que fuese ignorada y no pueda cumplir con su propósito. El delicado equilibrio buscado lo encontramos recién al hacer que la constante tenga el mismo valor que C_1 para las demás aristas.

Con esta fuerza se obtienen resultado bastante buenos. En la Figura 15.5 se muestran dos situaciones en las cuales la figura y las longitudes resultantes son las mismas pero en una se utilizan fuerzas hacia el centro y en la otra no.

Posteriormente introdujimos una mejora más para intentar que la fuerza sea más intensa cuando el vértice está muy cerca del borde. Esto fue motivado por la

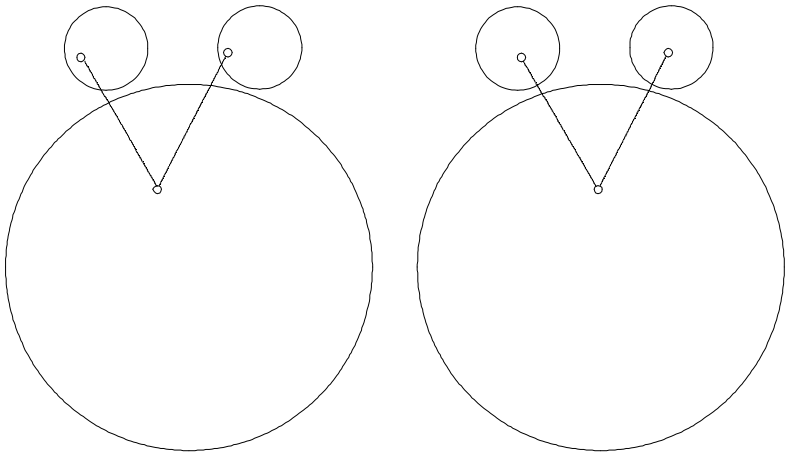


FIGURA 15.5. Ejemplo de mejoras causadas por utilizar fuerza centrípeta.

observación de que el principal efecto que produce esta fuerza es alejar los vértices de los bordes. Para aumentar la velocidad de convergencia modificamos el cálculo de la fuerza hacia el centro elevando al cuadrado dicha fuerza.

Este cambio también requirió cambiar la constante C_1, ya que al estar elevada al cuadrado las fuerzas centrípetas resultaban demasiado fuertes como para tener la misma constante que las otras. Los valores apropiados de C_1 surgen de la experimentación, y estuvieron entre $0,5$ y $0,8$, aunque no se encontró ningún valor que pueda ser considerado ideal.

Además de implementar esto en el *spring embedder*, también lo hicimos en DH. Para esto se agregó en la función de energía un término que penaliza la distancia entre el vértice y el centro de su región. El potencial agregado se basó la fuerza anterior, y es similar al usado para penalizar longitud de aristas largas en DH (ver sección 4.5), pero en lugar de utilizar un término cuadrático usamos uno cúbico, que de manera similar al caso anterior, dio mejor resultado. El potencial se define para todo vértice $v \in V$:

$$U_c(v) = d^3_{c_v v}$$

La constante usada para darle peso fue también menor que uno, siendo 1/3 la usada finalmente en nuestra implementación.

Es importante analizar el impacto que poseen estos cambio en el orden de complejidad del algoritmo. Considerando que los centros de las regiones son calculados previamente, el cálculo de las fuerzas centrípetas no supera en costo al de las atractivas, por lo tanto el orden de complejidad no cambia.

El costo total de calcular los centros con el algoritmo anterior es $O(|r_m|n)$, donde n es la cantidad de vértices y $|r_m|$ la cantidad máxima de vértices en una región, que no depende del grafo. Por lo tanto la complejidad total queda expresada como $O(|r_m|n + n^3)$. Es importante ver cómo para el caso de regiones

circulares o segmentos, el cálculo del centro se puede realizar en $O(1)$, sin afectar la complejidad del algoritmo.

15.3. Minimizar número de cruces entre aristas

El problema de minimizar el número de cruces entre las aristas es uno de los criterios más difíciles de atacar, y son pocos los algoritmos (generales) que contemplan este criterio.

Sin embargo, también es uno de los que más afecta a la estética del grafo resultante, y creemos que en nuestro problema este criterio es tan o más importante que en el caso general, y decidimos atacarlo de forma explícita. Nuestra forma de encarar el problema fue adaptando DH para que considere no sólo la cantidad de cruces entre las aristas, sino que también se pudiese dar cuenta si ese cruce se podría evitar fácilmente o no. Lo primero ya ha sido usado en otros algoritmos (incluido DH). Lo segundo, en cambio, no ha sido muy estudiado y es sobre lo que se concentrará esta sección.

El primer acercamiento a lograr esto fue reutilizar ideas de otros algoritmos de la literatura. Al igual que en DH, a la función de energía se le agregó un término que penaliza los cruces entre aristas, pero en lugar de ser simplemente la cantidad de cruces decidimos utilizar una función inversamente proporcional a la distancia entre los pares de aristas. Esto hace que cuando dos aristas se cruzan, queden a distancia cero, por lo tanto sean muy penalizadas, pero cuando dejan de estarlo pasan a tener una distancia positiva que penaliza mucho menos. Lo importante de esto es que permite un tratamiento continuo para el problema de la cantidad de cruces de aristas.

Consideramos que este nuevo potencial sólo debe ser considerado en la etapa de *fine tuning* visto que cuando se está construyendo la topología del trazado es muy común que las aristas se crucen, y agregar esta fuerza va a dificultar esto, no pudiendo arribar a una buena estructura topológica. Una vez iniciada la fase de *fine tuning*, la estructura del grafo ya está definida, por lo tanto sí es bueno intentar disminuir la cantidad de cruces. Nótese que este problema se debe a que la misma fuerza que se encarga de revertir los cruces entre aristas de forma indirecta también impide que si dos aristas no se cruzan, pasen a hacerlo. El potencial propuesto se define para todo par de aristas $e_1, e_2 \in E$ y tiene la siguiente forma:

$$U_{ee}(e_1, e_2) = \frac{1}{d^2_{e_1 e_2}}$$

donde $d_{e_1 e_2}$ es la distancia entre las aristas e_1 y e_2. Además, al igual que todos los potenciales en DH, va multiplicado por su peso relativo, constante c_{ee}.

Los resultados obtenidos con esta modificación fueron muy buenos pero aun se pueden mejorar considerando otros factores.

Si bien con lo anterior ya contamos con un término continuo que nos permite discriminar en la energía si dos aristas se cruzan o no, la forma en la que está definido hace que cuando no existe el cruce, sea poco probable que un nodo atraviese una arista creando un nuevo cruce, y cuando sí existe, sólo se deshace

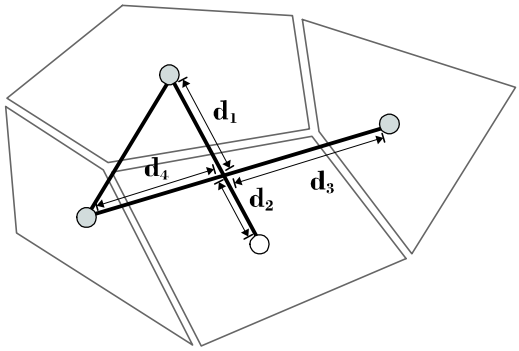


FIGURA 15.6. Las cuatro distancias asociadas con cada cruces. El de distancia menor (d_2), será movido.

si el nodo se encuentra lo suficientemente cerca como para deshacerlo en un solo paso, visto que de lo contrario, la distancia entre aristas continuará siendo cero haciendo infinita la sumatoria.

En una línea similar a la del algoritmo de Bertault, [Ber00] (ver sección 12.1.2), decidimos agregar otro potencial que nos permita eliminar de manera más efectiva ciertos cruces sencillos, pero muy frecuentes.

La intención fue crear un potencial que para cada cruce señalara de alguna forma qué vértice mover para eliminarlo.

Inicialmente pensamos en usar la suma de las cuatro distancias entre los respectivos nodos y el cruce. Este término no generó buenos resultados, visto que toda la figura tiende a acercarse a los centros de los cruces, pero modificando levemente la idea logramos obtener lo deseado.

El potencial que finalmente agregamos fue la distancia mínima entre los vértices que forman el cruce y el cruce mismo. Para dos aristas que se cruzan, $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$, definimos el potencial como:

$$U_{cr}(e_1, e_2) = \min_{v \in \{u_1, v_1, u_2, v_2\}} \|x - v\|^2$$

donde $x \in \mathbb{R}^2$ es el punto de intersección entre las dos aristas.

De esta forma el único nodo que tenderá a moverse será el que se encuentra más cercano al cruce, mientras que el resto no lo hará porque su movimiento en ese sentido no modificará la función objetivo. Lo importante de este potencial es que en muchos casos existe un nodo que de él depende que se deshaga el cruce, y éste suele ser el que se encuentra más cercano a dicho punto. En la Figura 15.6 se ilustra la idea. Es muy importante notar que este potencial, al igual que el anterior, también es continuo.

Debido a la continuidad y a que este potencial contribuye a definir la topología del trazado sin restringir la libertad de movimiento, nos pareció adecuado incluirlo en todo momento y no sólo en la etapa de *fine tuning* y ha mostrado dar muy

buenos resultados. Es importante notar que no hemos encontramos precedente alguno de un tipo de potencial como éste en la literatura.

La función objetivo resultante, incluyendo los dos nuevos potenciales aquí presentados, queda definida como:

$$\begin{aligned} E &= c_r \sum_{u,v \in V, u \neq v} U_r(u,v) + c_e \sum_{(u,v) \in E} U_e(u,v) + c_c U_c \\ &+ c_{en} \sum_{e=(u,v) \in E, w \in V} U_{en}(e,w) + c_{ee} \sum_{(e_1,e_2) \in E} U_{ee}(e_1,e_2) + c_{cr} \sum_{(e_1,e_2) \in Cr_E} U_{cr}(e_1,e_2) \end{aligned} \tag{15.3}$$

donde los primeros cuatro potenciales son los mismos que en DH (sección 4.5), Cr_E es el conjunto de pares de aristas que se cruzan, y c_{ee}, c_{cr} son los pesos relativos de los nuevos potenciales.

Desde el punto de vista de la complejidad, esta estrategia no empeora la complejidad de DH cuando se utiliza la versión que considera la cantidad de cruces de aristas, visto que cuando se calcula la cantidad de cruces, se puede calcular la distancia entre éstos en $O(1)$ y como la cantidad de nodos que posiblemente sean los más cercanos al centro de un cruce son cuatro (los extremos de las aristas), calcular la distancia del mínimo de éstos también se puede realizar en $O(1)$. El problema radica en tener que considerar en cada iteración todos los posibles pares de aristas, lo cual tiene un alto orden de complejidad (hasta $O(n^4)$), mayor al típico $O(n^3)$ de la mayoría de los algoritmos clásicos.

Sin embargo, creemos que en nuestro problema, donde los vértices están restringidos a una región del plano, es razonable pensar que para un gran número de aplicaciones la cantidad de vértices va a ser pequeña (hasta unos pocos cientos), debido a que si todo el grafo debe entrar en el espacio de trabajo, para que tenga sentido aplicar estos algoritmos cada región debe ser suficientemente grande como para que haya libertad acerca de dónde ubicar a los vértices (si no se los podría ubicar siempre en el centro de la región), y el lugar disponible limita el número de vértices.

Por otro lado, si el número de vértices es muy grande creemos que puede ser más fácil que en el caso general aproximar las fuerzas lejanas de manera eficiente, debido a que la posición de los vértices está muy limitada por las regiones. Si bien en este trabajo no se consideran técnicas para lidiar con grafos grandes en este problema con regiones, una breve discusión se presenta en la sección de trabajo futuro.

15.4. Evitar vértices muy cercanos a los bordes de las regiones

Otro criterio estético importante que nace del hecho de que los nodos se encuentren contenidos en regiones es evitar distancias muy chicas entre los nodos y los bordes de sus regiones. Este criterio es similar pero no equivalente al criterio

previamente comentado de que los nodos se encuentren cerca del centro de la región. Analizando ambos problemas en detalle se observa que las características que presentan hacen que sean dos problemas diferentes, aunque no son independientes.

Para que nuestro algoritmo considere este criterio decidimos agregar un potencial a DH que penalice cuando un nodo se encuentra muy cerca del borde de la región.

La primer idea y la más intuitiva es agregar un término de la forma

$$U(v) = c_{distBorde} * distABorde^{-1}$$

con $c_{distBorde}$ cierta constante y $distABorde$ la distancia entre el vértice v y el borde de su región. Implementamos esta modificación en DH pero no pudimos encontrar un valor para la constante $c_{distBorde}$ que se comportara de forma adecuada. En algunos casos cuando era muy pequeña no producía casi efecto alguno, y cuando era aumentada repelía al nodo hasta el centro de la región.

Analizando los resultados obtenidos nos dimos cuenta de que el problema no era el valor de la constante, sino que era un problema intrínseco del tipo de potencial. Es necesaria una función que cuando el nodo se encuentra muy cerca de la frontera lo repela fuertemente, pero que esta fuerza no sea proporcional a la distancia, sino que vaya decreciendo más abruptamente a medida que el nodo se aleja. Luego de experimentar con varios, elegimos el siguiente potencial (definido para cada $v \in V$):

$$U(v) = c_{distBorde} e^{(-2*\beta*distABorde)} \tag{15.4}$$

Esta función tiende a $c_{distBorde}$ cuando la distancia al borde tiende a cero, mientras que cuando el nodo se aleja, la función exponencial hace que todo el potencial tienda a cero. La constante β es el factor de ajuste de cuán lentamente se tenderá a cero.

Con esta fuerza se consigue alejar enérgicamente al nodo de la frontera cuando éste se encuentra cerca, pero a medida que se aleja la fuerza pierde importancia, dejándole libertad para que pueda encontrar su punto de equilibrio. Empíricamente utilizamos $\beta = 0{,}2$ y $c_{distBorde} \approx 10^5$, visto que la intención es que este crecimiento sea marcado. Si se quisiera hacer que el efecto de la fuerza sea más duradero con respecto a la distancia del nodo al borde, bastaría con ajustar el valor de β.

15.5. Los nodos no deben estar cerca de las aristas - SET

Este criterio es uno de los más antiguos pero menos tratados en el área. Pese a que existe una gran variedad de ideas al respecto, no suele tratarse de forma directa en los algoritmos derivados del SE – nótese que DH sí lo considera pero de una forma totalmente diferente a la que presentaremos –, nosotros decidimos atacar de forma explícita este problema.

Nuestro algoritmo, que decidimos llamar SET (por SE Triangulado), agrega una fuerza de repulsión nodo-arista. Esta fuerza se agrega en el cálculo de las

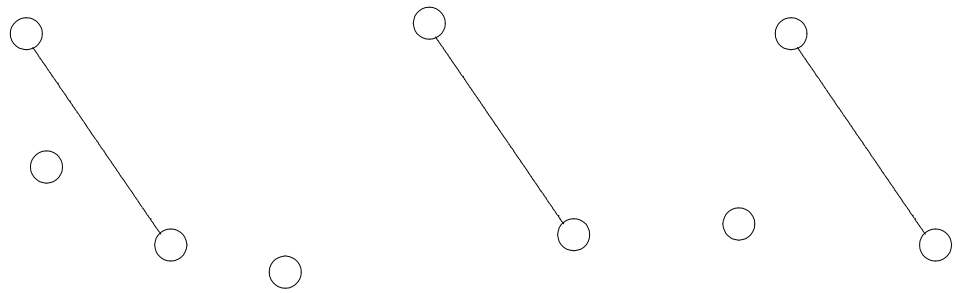


FIGURA 15.7. Ejemplo de cómo se ve un nodo cerca o lejos de una arista, independientemente de su contexto. Consideramos que la forma equilátera (izq.) es la más armoniosa.

fuerzas ejercidas sobre cada nodo. Explicaremos cómo funciona para un nodo y una arista aunque en el algoritmo este proceso se aplica a todos los nodos y todas las aristas.

Para incluir esta fuerza analizamos a qué distancia debería estar un nodo de una arista en caso de no estar conectado y si no interactuase con ningún otro nodo ni arista. Consideremos un ejemplo de un nodo y una arista. Claramente, como se ve en la Figura 15.7, no todas las distancias son equivalentes, hay distancias que quedan mejor – estéticamente – que otras.

Nos preguntamos porqué es que a uno le parece mejor una que otra y llegamos a la conclusión de que se mantiene cierta armonía cuando el nodo está en la posición que formaría un triángulo equilátero con la arista como se muestra en la Figura 15.7. Con este fin consideramos que la distancia ideal de un nodo a una arista es la que tendría que tener si formase dicho triángulo. Es fácil calcular la distancia de un nodo a una arista suponiendo que es un triángulo equilátero: la distancia será $\sqrt{3}/2$ veces la longitud de la arista; entonces penalizamos al nodo por lo que difiera de esta longitud.

Los resultados obtenidos, como se podrá apreciar en el próximo capítulo, son muy buenos pero tiene una falencia y es que en la etapa en que se está definiendo la topología del grafo esta fuerza dificulta que un nodo atraviese una barrera formada por una arista.

El algoritmo resultante tiene otro problema que es su alto costo computacional, pero creemos que eso podría mejorarse en un futuro y no es la idea central de este trabajo presentar implementaciones eficientes sino ideas que puedan luego mejorarse.

La expresión de esta fuerza de repulsión nodo-arista se presenta a continuación. Se calcula para todo par (u, e) con $u \in V$ y $e \in E$.

$$f_{SET}(u, e) = C_1 \log \left(\frac{\|p_v - p\|^2}{(\|e\| \frac{\sqrt{3}}{2})^2} \right)$$

Donde p es el punto más cercano a p_v de la arista e, $\|e\|$ es la longitud de la

arista e en el trazado y C_1 es una constante.

15.6. SER^2 y DHR^2

Con las ideas presentadas anteriormente armamos nuestros dos algoritmos más importantes, SER^2 y DHR^2 (SE y DH Restringidos a Regiones) que permiten probar todo lo presentado en este trabajo sobre restricciones dadas por regiones y cuentan con una implementación realizada en Java (detalles sobre la misma se pueden encontrar en el apéndice). Es importante marcar que SER^2 y DHR^2 no cuentan con la fuerza repulsiva nodo-arista presentada en SET, pero puede ser que en trabajos futuros sea agregada. SER^2 fue diseñado con las fuerzas comunes de SE pero utilizando la distancia ideal definida en la sección 15.1 y agregándole las fuerzas centrípetas de la sección 15.2. DHR^2 cuenta también con casi todas las ideas presentadas en este capítulo y fue definido como se indica en la sección 15.3 agregándole el potencial de distancia a bordes de la sección 15.4.

15.7. Observaciones para regiones que son segmentos

Como se mencionó en el capítulo anterior, es de interés el caso particular en el que las regiones son segmentos de recta. Muchos de los criterios presentados para regiones poligonales siguen siendo válidos, por lo que las adaptaciones propuestas al *spring embedder* y a DH sirven también para este sub-problema. Sin embargo, queremos mencionar que la importancia relativa de cada criterio es levemente distinta para este problema. Las pruebas que hicimos nos hacen pensar que el centro no es tan importante en los segmentos como lo es en los polígonos, mientras que evitar que las aristas sean paralelas a segmentos surge como un nuevo criterio de importancia.

Por otro lado, nos pareció interesante considerar también qué resultados se pueden obtener adaptando el algoritmo KK a cuando los vértices están restringidos *analíticamente* a sus segmentos, aprovechando que la restricción de que un vértice $v \in V$ esté sobre una recta definida por (d_v, h_v) puede incluirse usando que

$$p_v = \lambda d_v + (1 - \lambda) h_v \qquad \lambda \in \mathbb{R} \tag{15.5}$$

De esta forma, introducimos esta expresión en la función de energía de KK:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{i,j} (|p_i - p_j| - l_{i,j})^2 \tag{15.6}$$

Recordemos que para el caso de grafos generales, KK es uno de los mejores algoritmos tanto sea porque converge a los mejores resultados como por la velocidad con que lo hace. Al poder integrar las restricciones de regiones al algoritmo, se puede restringir el espacio de búsqueda a uno muy "parecido" al deseado (no

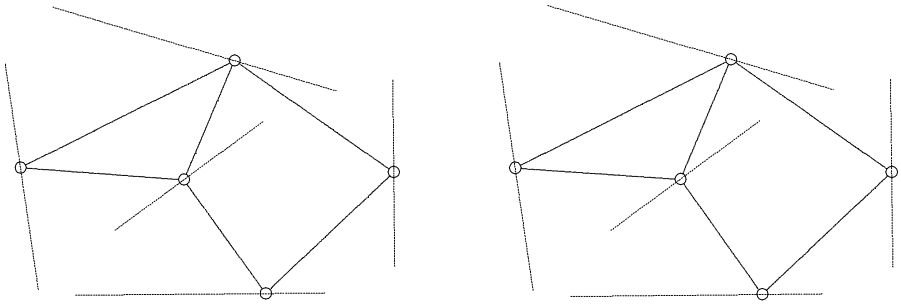


FIGURA 15.8. El trazado de la izq. fue creado con la versión adaptada de KK, mientras que el de la derecha con la versión que aproxima con parábolas. No hay casi diferencia.

es igual porque la ecuación (15.5) obliga a que el vértice esté sobre una recta, en lugar de un segmento).

Se reemplazó en la función de energía de KK cada p_v según (15.5), y se calcularon las derivadas primeras y segundas para poder realizar Newton-Raphson. Las expresiones de las derivadas resultaron extremadamente engorrosas, y no tiene sentido presentarlas aquí.

De todas formas, el hallazgo más notable no está relacionado con las derivadas, sino con la función de energía (15.6), ya que cuando las posiciones de los vértices cumplen la ecuación (15.5) y se fijan todos los vértices excepto uno, la función E, que pasa a depender sólo de λ, tiene una forma muy similar a la de una parábola. Esto hace que pueda ser aproximada con una parábola y aproximar su mínimo con el vértice de la función cuadrática.

Esto nos lleva a proponer un algoritmo extremadamente eficiente y sencillo:

KKSegmentosParabola(Grafo G, Regiones R)
1. Repetir n veces
    1.1. Para cada vértice v $\in$ V
        1.1.1 Fijar las posiciones de todos los vértices distintos de v
        1.1.2. Aproximar E(λ) con una parábola P(λ)
        1.1.3. Calcular el $\lambda_{\text{mín}}$= λ corresp. al vértice de P(λ)
        1.1.4. Hacer $\mathsf{p_v} = \lambda_{\text{mín}}\mathsf{d}_v + (1 - \lambda_{\text{mín}})\mathsf{h}_v$

Las pruebas realizadas muestran que el resultado de este simple algoritmo es muy parecido al obtenido utilizando las expresiones de las derivadas y haciendo Newton-Raphson. En la Figura 15.8 puede observarse para un mismo grafo con segmentos la similitud entre los dos resultados.

La calidad de los trazados que se obtienen con esta adaptación suele ser muy inferior a la obtenida con las adaptaciones presentadas en las secciones anteriores, sin embargo este algoritmo no deja de ser digno de mención debido a su eficiencia. Un posible uso es para generar rápidamente un trazado inicial para otros algoritmos mejores, como el *spring embedder*.

15.8. Otros criterios estéticos

Existen otros criterios estéticos válidos como presentamos al comienzo, pero no serán tratados en este trabajo. Estos criterios son:

- Maximizar el ángulo entre aristas adyacentes.
- Maximizar el ángulo entre aristas que se cruzan.
- Mostrar simetría.

Algunos de estos criterios, tal es el caso de "mostrar simetría", no suelen ser tratados de forma explícita (hemos encontrado en la literatura una única excepción, ver sección 12.1.2), pero los algoritmos utilizados los consideran en cierta medida, por ser dirigidos por fuerzas. Además volvemos a señalar que al estar tan limitada la topología del grafo por las regiones, es mucho menos lo que puede lograrse en lo que respecta a la simetría.

Un conjunto de criterios que, aunque no abordamos, nos parecen muy interesantes para desarrollar son los que consideran el ángulo de las aristas adyacentes. Comentaremos acerca de éstos en la sección de trabajo futuro.

Capítulo 16

Resultados

Habiendo presentado ya nuestros algoritmos, veremos figuras con resultados generados por éstos. Como ya hemos dicho reiteradas veces, es importante notar que pese a que tomamos como base los algoritmos clásicos de trazado de grafos, estos deben ser adaptados para el problema específico porque si no los resultados que generan no son satisfactorios. Por este motivo en los resultados compararemos nuestros algoritmos con los clásicos y mostraremos las bondades y utilidad de cada uno.

16.1. Resultados cualitativos

Como primer ejemplo consideramos un grafo con regiones sencillo como el presentado en la Figura 16.1, donde se aplicó el *spring embedder* (SE) y DH, obteniendo los trazados de más a la derecha.

Como se puede observar los resultados distan de ser estéticamente buenos. Ignoran criterios como la distancia de los nodos a los bordes y las distancias a los centros de las regiones que son muy importantes en nuestro contexto. Las primeras soluciones generadas por nuestros algoritmos son bastante simples, por ejemplo se pueden ver en la Figura 16.2 que muestra los resultados generados por SE considerando las distancias ideales como las distancias mínimas entre regiones (izq.) o SE considerando la longitud ideal de las aristas como la distancia promedio (der.).

Claramente se ve cómo pese a no ser el mejor resultado que podría esperarse, el de la derecha supera a los generados por los algoritmos clásicos. En 16.2 (izq.) se puede apreciar que utilizar la distancia mínima entre regiones trae resultados contraproducentes como juntar todos los nodos en una frontera, y en este ejemplo es peor aun visto que todas las regiones tienen un punto fronterizo muy cercano.

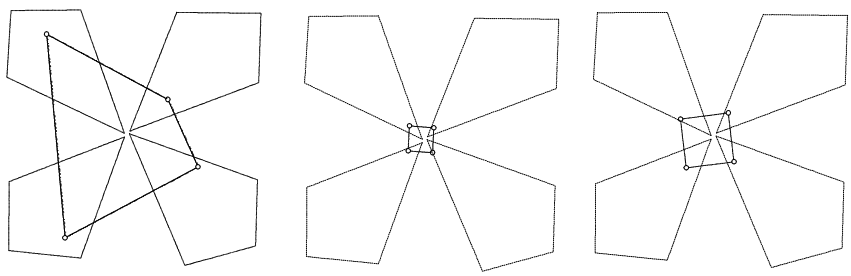


Figura 16.1. SE y DH proyectados a regiones.

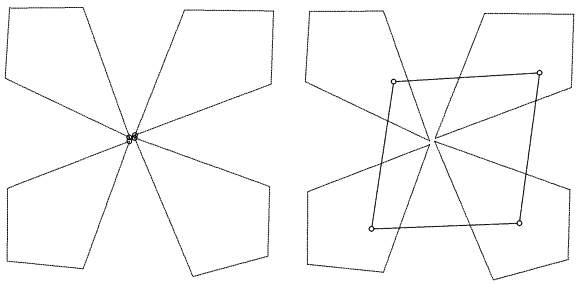
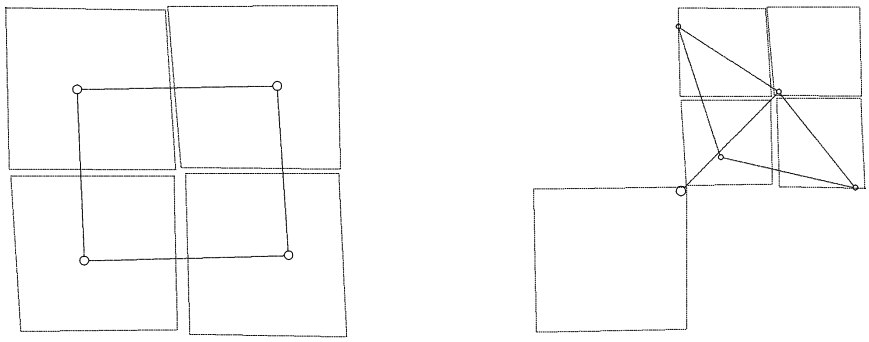


FIGURA 16.2. SE considerando distancia mínima y distancia promedio.



(a) **(b)**

FIGURA 16.3. Un grafo (a) y un trazado del mismo (b) luego de agregar una región más, que ilustran uno de los problemas de usar distancias promedio: la sensibilidad a los valores atípicos.

Se ve como se desperdicia el resto de la región y se produce un resultado en el cual es poco claro qué región es representada por cada nodo o cuántos nodos hay. La Figura 16.2 (der.) muestra resultados aceptables porque al estar todas las regiones a una distancia similar, y ser el grafo simétrico, la distancia promedio da un buen resultado. No es así en el caso de la Figura 16.3, donde se pueden apreciar las carencias de este método. En la Figura 16.3-a se muestra una malla de 2×2 luego de haber utilizado el algoritmo de promedio. Al agregarle un nodo más conectado con el extremo opuesto se obtiene la Figura 16.3-b.

Como suele pasar en los algoritmos que utilizan el promedio, son propensos a ser afectados de forma significativa por los valores alejados (*outliers*). Se ve claramente como toda la estructura del grafo se vio afectada por la incorporación de un solo nodo.

Todos estos problemas aparecen por no considerar una verdadera distancia natural entre las regiones, que son las que determinan la longitud de las aristas.

Las figuras siguientes (Figura 16.4) son las que sí consideran las cualidades propias de las regiones, y se puede apreciar cómo al considerar más criterios se obtienen cada vez mejores trazados. Las primeras consideran sólo modificaciones en

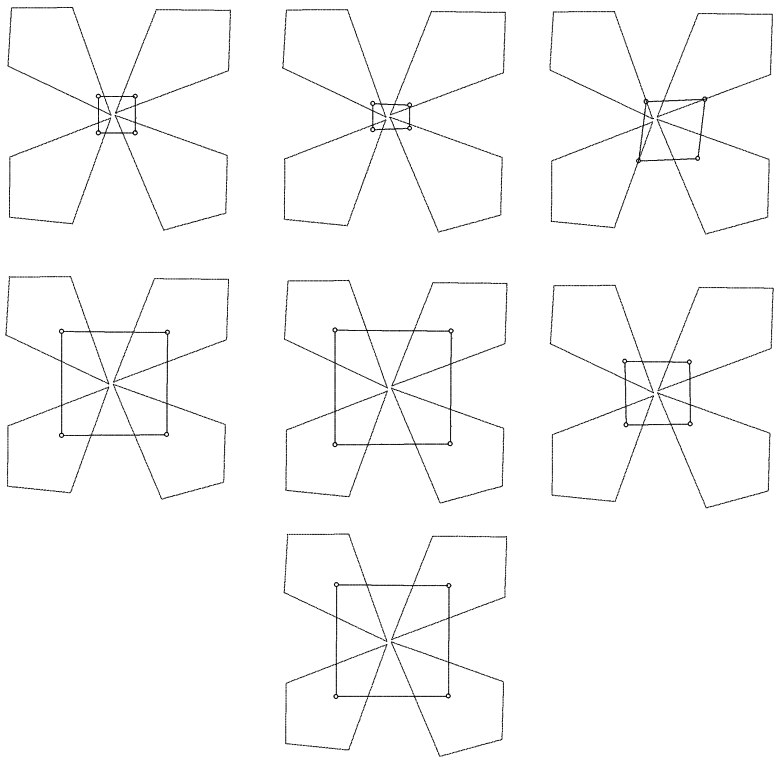


FIGURA 16.4. Resultados de algoritmos adaptados para regiones (SE y DH), agregando distintos criterios estéticos.

las longitudes ideales como el promedio de los centros, la distancia de compromiso definida en 15.1 tanto sea para SE como para DH. Las siguientes también consideran las distancias a los centros, llegando por último a los resultados generados por DHR^2 y SER^2.

El siguiente resultado que mostraremos (Figura 16.5) es el clásico grafo de un camino. Consideraremos un camino formado por 5 nodos y 4 aristas. Nuevamente en este resultado se aprecia la necesidad de un algoritmo que considere los criterios estéticos propios de nuestro dominio. Pese a que SE común o DH obtienen buenos resultados formando una línea casi recta, la hacen sobre uno de los bordes lo cual dificulta la comprensión de qué es lo que se está viendo.

Es interesante entender porqué pasa esto. La ubicación de la recta en sentido horizontal está dada por los equilibrios de fuerzas entre las aristas, pero la ubicación vertical es independiente de esto, por lo tanto aquellos algoritmos que no consideren otro criterio no podrán lograr buenos resultados verticales. A simple vista la ubicación vertical es importante, y esto se debe principalmente a dos motivos: la distancia a los centros y la distancia a los bordes. Se puede apreciar que los algoritmos que consideran estos criterios obtienen mejores resultados (Figura 16.6).

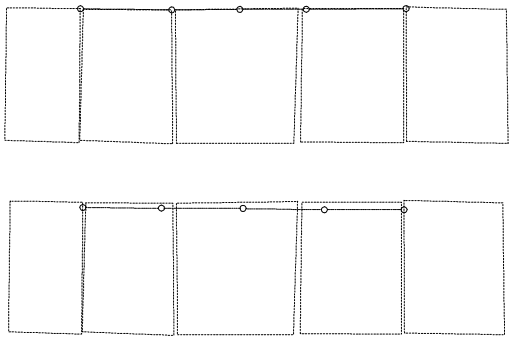


FIGURA 16.5. Grafo de un camino simple producido con SE (arriba) y DH (abajo).

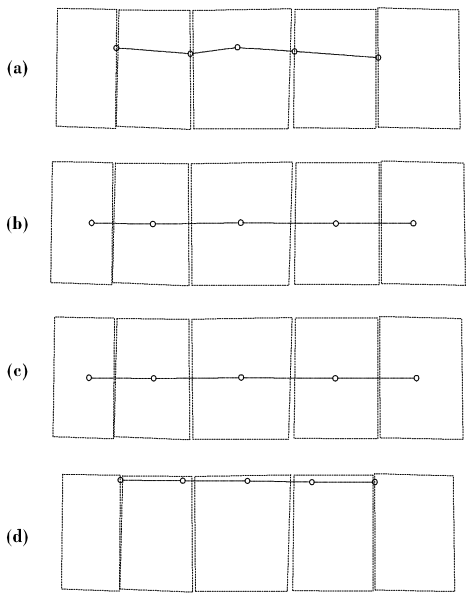


FIGURA 16.6. Camino simple considerando las regiones. SE Repulsión nodo-eje (a), DH-R^2 (b), SE-R^2 (c), SE longitud mínima (d).

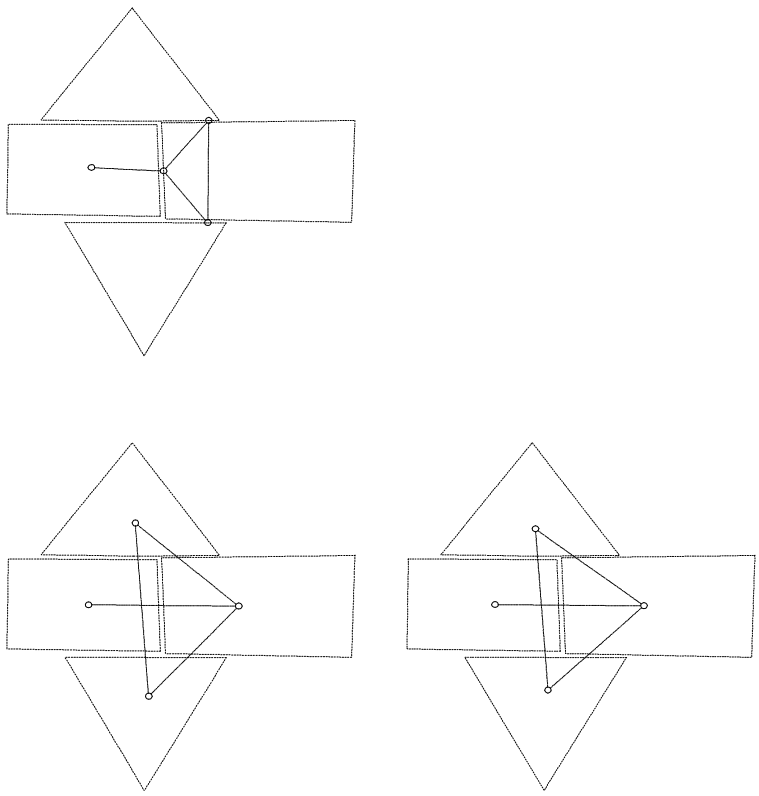


FIGURA 16.7. Nodo encerrado por una arista. DH (arriba), SE-R^2 (abajo izq.) y DH-R^2 (abajo der.).

Otro ejemplo interesante es el que se puede ver a continuación (Figura 16.7), en el cual se comparan varios algoritmos de los propuestos que consideran las regiones.

El primer ejemplo fue generado por DH sin considerar las regiones, y como puede verse resultó en una forma muy particular a la cual ninguno de los otros algoritmos llegó. Dicha figura es poco interesante en nuestro contexto porque da la idea de un triángulo en una región y un punto en otra, con dos regiones vacías arriba y abajo. No se transmite realmente la información de que hay cuatro regiones que están relacionadas. Por este motivo es que los otros algoritmos ignoraron esta solución y mantuvieron una topología semejante.

Otro dato importante para ver en esta figura es que hay una arista que puede ser cruzada o no. En caso de cruzarla se puede acercar el nodo al centro mostrando más claramente su pertenencia a la región, en caso contrario se disminuye la cantidad de cruces de aristas. En la Figura 16.7-a se utilizó el algoritmo SE con repulsión nodo-arista, por lo tanto el nodo intenta mantener distancia de la arista y no rompe dicha frontera. En los casos b y c (SER^2 y DHR^2) la frontera se pasa en pos de obtener una posición más representativa de la región. Una pregunta inmediata es cuál de estas soluciones es mejor. En realidad ambas son buenas y si una es mejor que la otra depende del contexto para el cual se las desee utilizar,

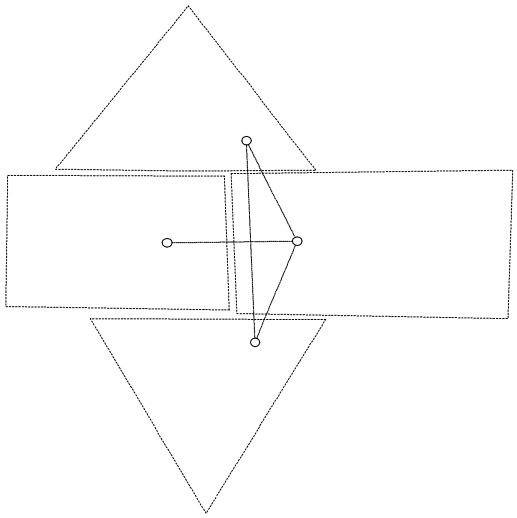


FIGURA 16.8. SE-R^2 disminuyendo la fuerza hacia el centro.

pero en ambos casos se transmite de forma clara la relación entre las regiones participantes. Se puede ver también la Figura 16.8 en la cual se utilizó SE al igual que en 16.7-b, pero en este caso disminuyendo el peso que se le da a la fuerza del centro. En el caso anterior se multiplicó la fuerza hacia el centro por un factor de $0,2$, mientras que en el segundo se usó $0,0002$. Es importante notar que el valor de las constantes no es fijo para todo grafo, y ciertos valores funcionan mejor en algunos casos que en otros. Empíricamente obtuvimos que valores entre $0,2$ y $0,8$ son apropiados para una gran cantidad de grafos, pero este ejemplo muestra que siempre hay excepciones.

En estos ejemplos vimos casos sencillos donde se notan poco las diferencias de rendimiento, pero es importante ver cómo en ejemplos levemente más grandes ya se empieza a notar la gran diferencia entre DH y SE.

En el grafo que se ve en la Figura 16.9 ya no es tan sencillo determinar cuál es la mejor representación gráfica.

Probando con los algoritmos más importantes se obtuvieron los resultados de la Figura 16.10.

Se ve como en la Figura 16.10, b y c son superiores. Esto se debe a que el algoritmo de la Figura 16.10-a no considera estar alejado de los bordes de las regiones como un criterio estético. También es importante ver cómo ya en un grafo de estas dimensiones empieza a notarse la diferencia en tiempo entre DH y SE. Mientras que SE terminó luego de 235 ms., DH tardó 3875 ms. Esta diferencia se hace cada vez más notoria a medida que se consideran grafos de mayor tamaño.

Otra característica importante de nuestros algoritmos es la estrategia implementada en DH que permite evitar cruces. En la Figura 16.11 se presenta una situación en la cual el nodo es atraído en un sentido por su centro y por otro nodo, y en el otro sentido por dos aristas. Las fuerzas son más o menos semejantes, visto que el centro está configurado para pesar como un nodo más, pero

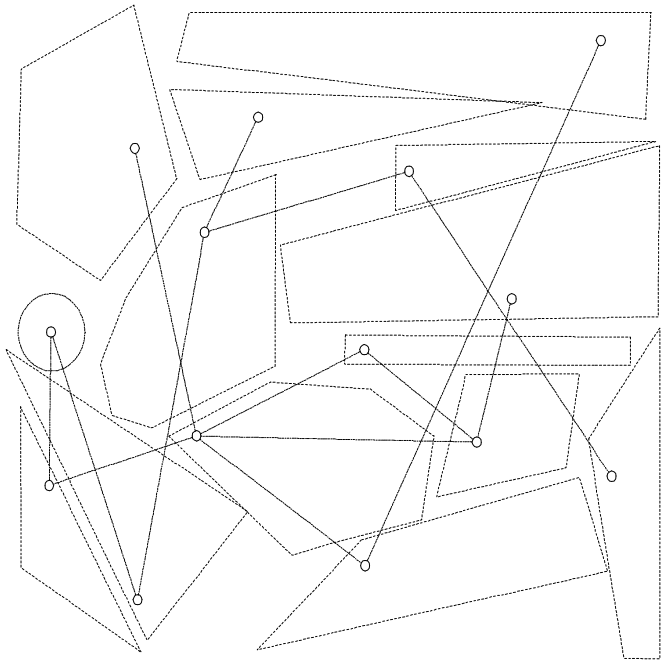


FIGURA 16.9. Ejemplo de grafo donde es difícil determinar buenos criterios estéticos.

lo que marca la diferencia es la existencia de una arista en el medio que según en qué sector del trazado se coloque formará un cruce o dos.

La Figura 16.12 (izq.) muestra el resultado luego de utilizar DH sin considerar la estrategia para detectar cruces de aristas, mientras que en 16.12 (der.) sí se la utiliza.

Hay que destacar que este caso no es equivalente al de la Figura 16.7 porque aquí no se trata simplemente que el nodo no cruce la frontera sino todo lo contrario, que la cruce para de esta forma poder disminuir la cantidad de cruces de aristas.

También probamos diseñar una malla, la cual es una prueba estándar en el área de trazado de grafos a la hora de probar nuevos algoritmos. Los resultados, muy buenos con ambos algoritmos, se muestran en la Figura 16.13. Es importante ver cómo el caso de la malla que para muchos algoritmos es difícil, en nuestro contexto es mucho más fácil, dado que se posee una gran cantidad de información sobre las posiciones de los vértices, debido a sus regiones.

La Figura 16.14 muestra en un mismo escenario la respuesta de SER^2, DHR^2 y DH con fuerzas nodo-arista. Es interesante ver la disposición final. El primer trazado es el inicial, y el segundo y tercero son los producidos con SER^2 y DHR^2, respectivamente. Pese a que estos dos son claramente mejores que el último, éste logra dibujar un triángulo equilátero, pero como no posee ningún sistema de detección de bordes tiende a irse hacia alguno de ellos y no poder solucionarlo.

Aparte de los algoritmos que presentamos para trabajar con regiones que sean

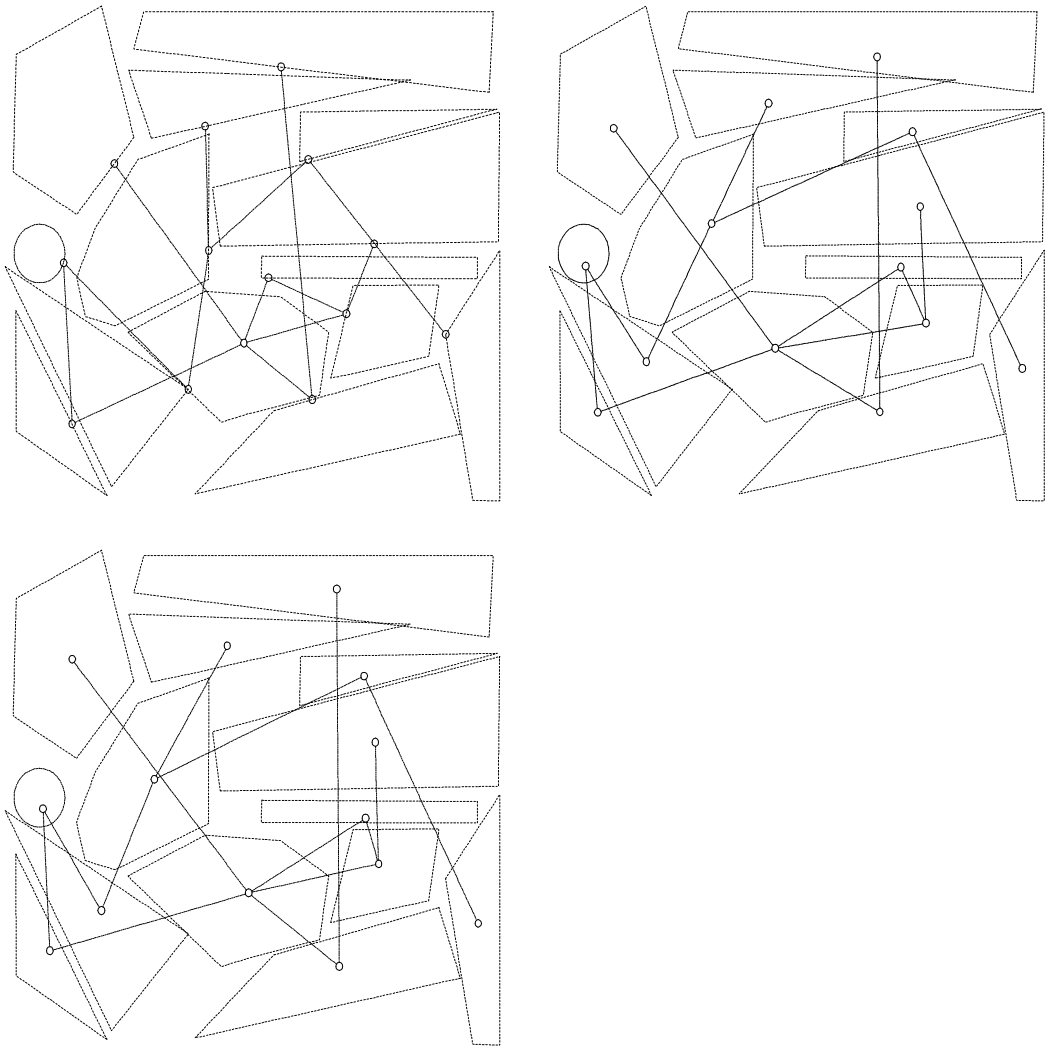


FIGURA 16.10. Soluciones de SE con fuerza nodo-eje (arriba izq.), SER^2 (arriba der.) y DHR^2 (abajo).

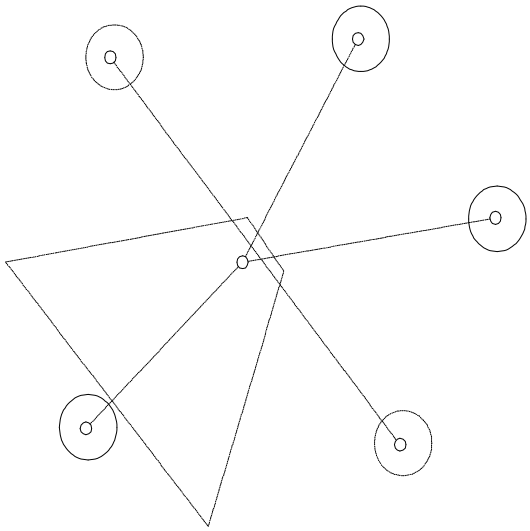


FIGURA 16.11. Nodo atraído por su centro y por aristas opuestas.

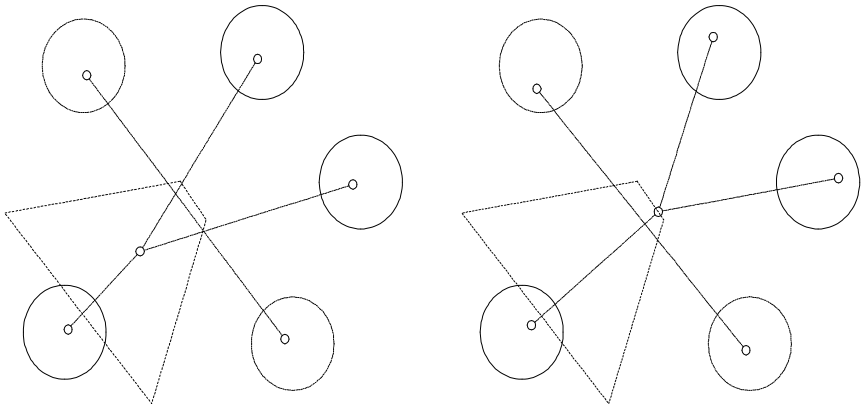


FIGURA 16.12. Resultado no detectando cruces (a) y detectándolos (b).

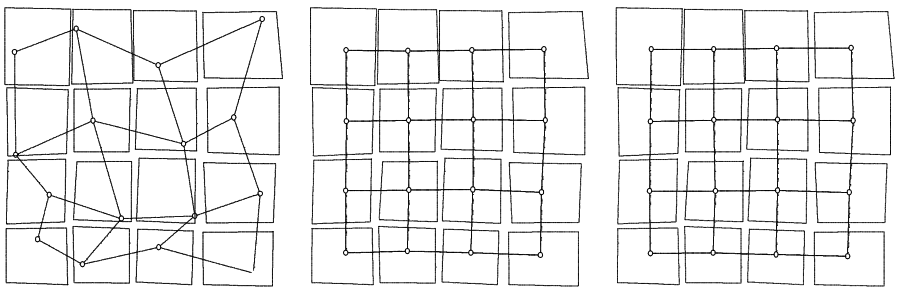


FIGURA 16.13. Malla de 4×4 utilizando SE-R^2 (a) y DH-R^2 (b).

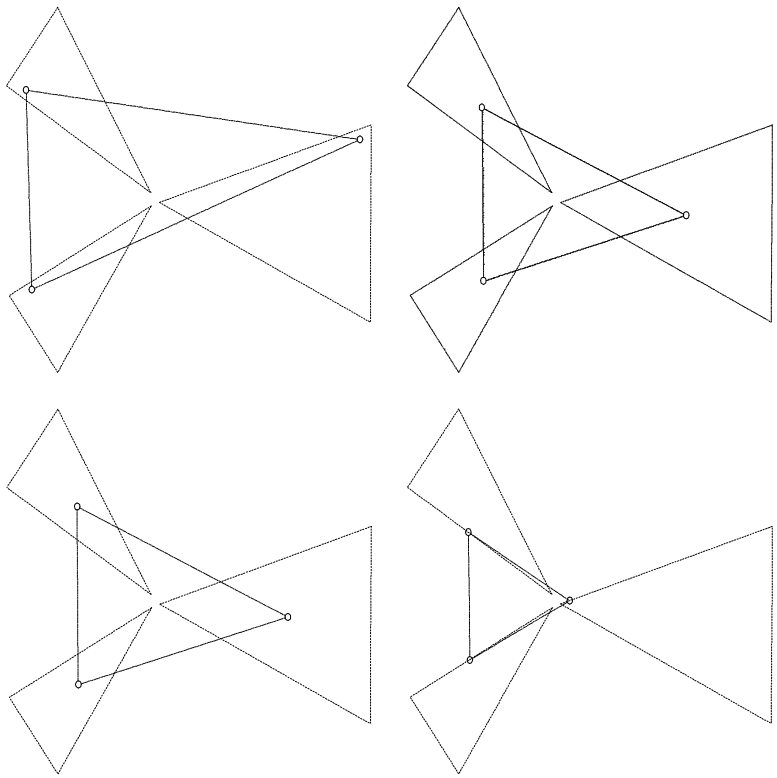


FIGURA 16.14. Triangulación utilizando SE-R^2, DH-R^2 y SE con fuerzas nodo-arista.

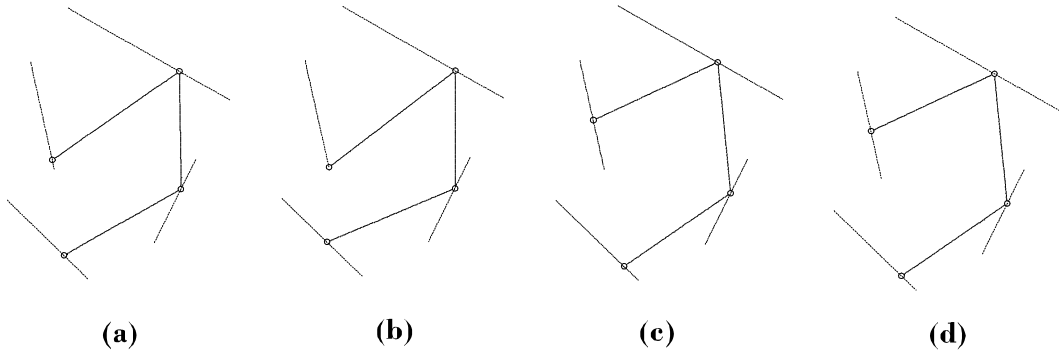


FIGURA 16.15. Ejemplo de trazado donde las regiones son segmentos, usando los algoritmos SER^2 (a), DHR^2 (b), KKAdaptado (c) Y KKParabola (d).

polígonos o discos, también hemos presentado algoritmos para trabajar en regiones que son segmentos. A continuación mostramos un ejemplo de los resultados obtenidos probando todos los algoritmos propuestos para segmentos.

Es interesante ver como pese a que todos los resultados son muy similares, la diferencia de tiempo de cómputo es significativa. Mientras que los tres primeros tardaron 125 ms, 62 ms y125 ms, respectivamente, el cuarto tardó sólo 15 ms. Esto se debe a que la forma de procesar de este último, que aproxima las funciones de energía con parábolas, es mucho más eficiente que la de los otros. Si bien éste es un ejemplo pequeño, ilustra la diferencia que se hará más evidente a medida que los grafos aumenten su tamaño.

Por último a modo de ejemplos de dimensiones más reales, mostraremos un grafo que representa a la República Argentina (Figura 16.16) con determinada relación entre provincias y uno de España (Figura 16.17) considerando las autonomías como regiones.

Los resultados de estos casos son mas difíciles de analizar visto la cantidad de aristas y las grandes limitaciones dadas por las regiones, pero es muy interesante ver como en el caso de Argentina, pese a que se comienza con el nodo de Buenos Aires provocando muchos cruces de aristas, con ambos algoritmos esto se intenta solucionar, lográndose mejores resultados con SER^2 que logró anularlos en su mayoría. El caso de España es más difícil de determinar si el trazado es mejor visto que no se realizaron muchos cambios debido a la buena posición inicial y las grandes limitaciones de las regiones.

16.2. Tiempos de ejecución

Aunque nuestro objetivo es presentar nuevos algoritmos que permitan encarar el problema de los nodos restringidos a regiones, sin profundizar en la optimización de rendimiento que se puede hacer, para resaltar las diferencias relativas en tiempo de ejecución de nuestros algoritmos y los clásicos, presentamos los tiempos de las

FIGURA 16.16. Ejemplo de un mapa de Argentina con SE-R^2 (medio) y DH-R^2 (der.).

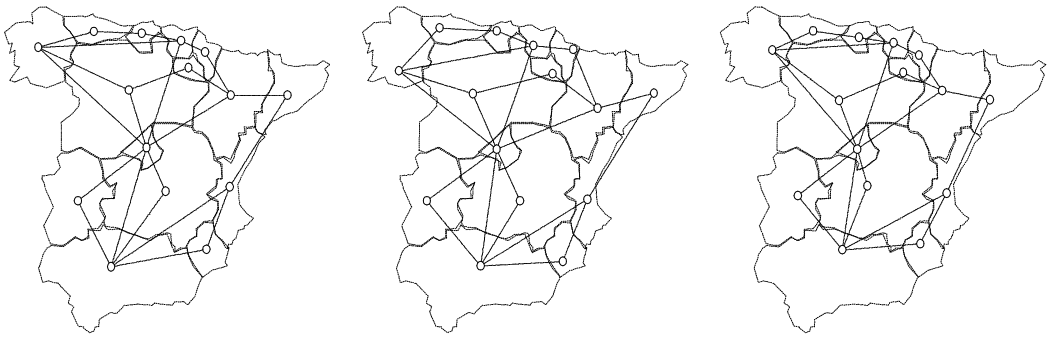


FIGURA 16.17. Ejemplo de España utilizando SE-R^2 (medio) y DH-R^2 (der.).

pruebas anteriores, recalcando que todas las implementaciones fueron hechas sin ningún tipo de optimización en los algoritmos.

Se muestran tiempos de corrida de los algoritmos DH y SE con el único agregado de las limitaciones en las posiciones, y de nuestros algoritmos DHR^2 y SER^2, todos de acuerdo a nuestra implementación en Java.

Los resultados de tiempo (en milisegundos) de las pruebas – por orden de aparición – son los siguientes:

	DH	SE	DHR^2	SER^2
Prueba Figura 16.4	172	360	172	187
Prueba Figura 16.6	219	219	203	422
Prueba Figura 16.7	141	219	172	182
Prueba Figura 16.10	3735	641	3875	235
Prueba Figura 16.13	13468	812	14141	797
Prueba Figura 16.14	109	156	172	266
Prueba mapa Argentina	33953	891	33468	188
Prueba mapa España	11297	610	12375	204

Los resultados fueron medidos en una computadora Pentium 4 2,8Ghz. Es importante notar cómo en los grafos de pequeño tamaño los tiempos de ejecución son similares, mientras que en los casos de mayor cantidad de nodos la diferencia entre SE y DH empieza a ser cada vez más notoria. También es importante señalar que nuestros algoritmos, a pesar de que consideran varios criterios estéticos más, no son mucho más lentos que los originales, e inclusive en algunos casos son más rápidos. Esto se atribuye a que debido a los criterios estéticos adicionales considerados se restringe de forma efectiva el espacio de búsqueda, lo cual permite que la convergencia se alcance más rápidamente.

Capítulo 17

Conclusiones

17.1. Conclusiones

Este trabajo ha sido estructurado en dos partes bien definidas. Se presentó por un lado una revisión del estado del arte de los métodos de trazado de grafos dirigidos por fuerzas, y por el otro un conjunto de algoritmos para el problema concreto de cómo trazar grafos cuando los vértices representan regiones geográficas.

La primera parte del trabajo es un relevamiento minucioso del estado del arte del trazado de grafos mediante métodos dirigidos por fuerzas. El mismo constituye un valioso marco de referencia para futuros trabajos, así como también sirve como material de consulta para quienes necesiten implementar algoritmos de trazado de grafos para grafos generales, y/o que necesiten adaptar los algoritmos existentes a sus necesidades puntuales.

La segunda parte provee soluciones efectivas para el problema de tener grafos con vértices que representan regiones geográficas. El análisis que se presenta del problema logra identificar qué es lo que se busca (criterios estéticos) en este tipo de trazados. Los algoritmos que proponemos, todas extensiones de los algoritmos clásicos, permiten lograr trazados con las características buscadas.

Los criterios que hacen que un trazado sea mejor que otro son sin duda subjetivos, pero creemos que los resultados que se han obtenido muestran que los algoritmos aquí propuestos apuntan en la dirección correcta. Si bien todavía hay mucho por mejorar (algunas ideas se dan más adelante), los avances en la calidad de los trazados que se han logrado con los algoritmos aquí presentados constituyen un logro importante, ya que superan ampliamente a los que se podían lograr hasta ahora.

17.2. Contribuciones principales

Ambas partes de este trabajo contienen contribuciones de valor. La primera parte constituye un relevamiento y análisis del estado del arte de los algoritmos dirigidos por fuerzas. Es la primera revisión que se concentra exclusivamente en la familia de técnicas más usada para el trazado de grafos – los dirigidos por fuerzas. La misma es un aporte valioso al área del trazado de grafos, ya que:

- Abarca todos los trabajos relevantes sobre el tema. Se revisan todos los algoritmos importantes publicados hasta el momento, presentándolos con claridad y con un nivel de profundidad suficiente como para poder llevarlos a la práctica, pero sin entrar en detalles innecesarios que ensombrecen la esencia de los métodos. A la vez, se muestra la evolución de los mismos, ayudando a comprender los existentes y mostrando hacia dónde apuntan los que están por venir.

- Incluye trabajos muy recientes, que hasta ahora no habían sido analizados, y que son los que están marcando nuevos rumbos en el trazado de grafos.

- Provee una visión consistente, comparativa y transversal del tema. No sólo se presentan los algoritmos, sino que se los sitúa en el contexto en el que son útiles y se los compara con otros similares. Varias secciones (como los capítulos 5 y 10) se dedican a dar una visión global sobre aspectos vitales de los métodos dirigidos por fuerzas, pero que sin embargo suelen ser tratados sólo como temas accesorios en la literatura.

La segunda parte del trabajo se centra en un problema no explorado, el trazado de grafos en el que los vértices representan regiones geográficas, y deben restringirse a ellas. Al respecto, en esta tesis:

- Se presenta un análisis de este problema, mostrando que los algoritmos existentes no son suficientes para abordarlo, y explicando porqué limitar cada vértice a su región no es suficiente para obtener trazados de calidad.

- Se estudian los criterios estéticos que definen cuándo un trazado de estas características es bueno. Se revisan los existentes y se proponen dos nuevos criterios estéticos (cercanía al centro y lejanía de los bordes) para este problema en el que cada vértice representa una región geográfica.

- Se proponen algoritmos para obtener trazados que se ajusten a estos nuevos criterios. Se presentan fuerzas y potenciales para extender dos de los algoritmos clásicos más usados, el *spring embedder* y DH, que responden explícitamente a cada uno de los nuevos criterios.

- Se propone un nuevo potencial para evitar cruces de aristas, que no sólo penaliza los cruces sino que guía al algoritmo de *Simulated Annealing* para deshacer algunos tipos de cruces muy comunes, usando el vértice más cercano al cruce. El potencial propuesto es útil no sólo para cuando hay regiones, sino para el problema general de trazado de grafos.

- Se propone una nueva fuerza para evitar que los nodos se superpongan con las aristas (SET). El criterio que ataca no es propio del problema de regiones, sino que también se encuentra presente en el problema de grafos generales.

17.3. Trabajo futuro

En lo que respecta a la primera parte de este trabajo, hay dos aspectos importantes donde creemos que se puede avanzar. Por un lado nos hubiera gustado dedicar más espacio a algunos algoritmos (como los mencionados en el capítulo 12, y muchos otros que no fueron nombrados) que nos parecieron valiosos pero que por limitaciones de tiempo y espacio no fue posible cubrir con profundidad. También creemos que sería importante no sólo limitar la búsqueda de trabajos relevantes a los publicados en las publicaciones especializadas, sino intentar abarcar también

paquetes de *software* comerciales que usan trazado de grafos. Es un hecho que la visualización de grafos surge en muchos ámbitos durante el desarrollo de distintos tipos de *software* y muchas veces se crean algoritmo *ad-hoc* muy buenos que nunca llegan a ser publicados. Debido a la flexibilidad de los algoritmos dirigidos por fuerzas, es posible que existan muchas adaptaciones y extensiones relevantes sin publicar.

Otro trabajo que es necesario es realizar una comparación experimental de los distintos algoritmos, especialmente de los de los últimos años. La única publicación importante al respecto es [BHR95], y sólo incluye a algunos de los algoritmos clásicos. A pesar de esto es citada muy a menudo debido a ser casi la única existente. Realmente hace falta una nueva comparación que compare algoritmos de distintos tipos, por ejemplo, algoritmos para *clusters*, algoritmos para grafos dinámicos, y en particular una para algoritmos para grafos grandes sería un inmenso aporte.

En lo que respecta a la segunda parte, el hecho de estudiar un problema casi inexplorado nos deja con un sinfín de cosas que hubiéramos querido hacer o probar. Aquí listamos las principales.

Optimización de los algoritmos. Como ya se ha comentado anteriormente, en ninguna de las implementaciones de nuestros algoritmos consideramos primordial la eficiencia. Nuestro trabajo se centró en dar soluciones algorítmicas a un problema particular (para el cual no existían algoritmos), e implementarlos para mostrar los resultados que se pueden alcanzar. Aun así pensamos que hay muchas formas de mejorar los tiempos que hemos obtenido. Una de las cuestiones a optimizar es el cálculo de las fuerzas repulsivas entre nodos, porque al igual que cuando sólo se consideran los nodos cercanos para calcular la repulsión, en nuestro problema particular las mismas regiones hacen que los movimientos se encuentren limitados, haciendo que los nodos de regiones lejanas nunca influyan mucho en ese tipo de fuerza. Un concepto que no hemos explotado y que serviría con este fin es la "distancia entre regiones". Para las fuerzas repulsivas podrían sólo tenerse en cuenta los nodos de las regiones vecinas o de aquellas que se encuentran a lo sumo a una distancia menor a cierto umbral.

Otros aspectos donde se puede optimizar es en las operaciones geométricas relacionadas con las regiones, como el cálculo del centroide. Nuestra forma de calcularlo es la más inmediata y no hemos profundizado en optimizarlo, pero creemos que se podría hacer de forma más eficiente, lo cual es un factor importante en nuestros algoritmos debido a que el orden de complejidad se encuentra multiplicado por un factor de cantidad de vértices de las regiones. Otras operaciones geométricas como las proyecciones sobre las regiones también pueden ser hechas de forma más eficiente.

Grafos grandes. Los algoritmos presentados en este trabajo no se escalan bien a grafos de miles de nodos, ya que tienen la complejidad de los algoritmos clásicos dirigidos por fuerzas (que tampoco escalan). Si bien en el caso general de trazado de grafos es clara la necesidad de algoritmos para grafos grandes, en nuestro problema no parece ser tan evidente. Sin duda que puede haber grafos de

regiones muy grandes que necesiten ser visualizados. Pero recordemos que en el caso del trazado general, el principal objetivo de los algoritmos para grafos grandes es visualizar la estructura global del grafo. Sin embargo, cuando los vértices son regiones con un área bien delimitada, la estructura global ya está fija de antemano, y el algoritmo intenta mostrar de mejor manera las relaciones entre las regiones. Es por esto que nos cuesta encontrar instancias, con sentido, del problema con regiones donde los grafos puedan ser tan grandes.

Por otro lado, si lo que se necesita es ver el trazado "en detalle", las ideas de la sección anterior para aproximar el cálculo de las fuerzas repulsivas podrían usarse para trabajar sólo con sectores del grafo, evitando trabajar con muchos nodos al mismo tiempo.

Otros tipos de regiones. Creemos que hay una gran cantidad de trabajos que se podrían continuar a partir de éste. Ejemplos de estos son el análisis de regiones más particulares para poder encontrar nuevos comportamientos o criterios estéticos. Un ejemplo importante es el de las regiones cóncavas. Analizando estas regiones concluimos que su comportamiento difiere ampliamente del de las regiones convexas, ya que debido a que los algoritmos tipo *spring embedder* hacen búsqueda local, es posible que haya sectores de las regiones a los que nunca se acceda debido a que son necesarias movidas en subida.

Una primera forma de aplicar nuestros algoritmos sería considerando la envolvente convexa de la región y luego encontrar alguna función que proyecte el resultado a la verdadera región. Esto evitaría el problema antes mencionado, aunque se corre el riesgo de no poder proyectar de manera adecuada el punto óptimo en la envolvente convexa a la región cóncava.

Otro caso que consideramos podría ser interesante es el caso en el cual las regiones se solapan. Hemos probado nuestros algoritmos y en muchos casos dan buenos resultados, pero sería importante antes de pensar un algoritmo para este problema hacer un análisis de sus criterios estéticos propios, porque seguramente diferirán de los que hemos estudiado (ej. podría ser que las áreas con mayor cantidad de intersecciones atraigan o repelan al nodo). Junto con este problema se puede ver uno casi opuesto y es que las regiones sean una partición del plano. Este es un problema muy interesante visto que tiene una asociación directa con casi cualquier mapa real. Muchas de nuestras pruebas se han hecho sobre casos de este tipo, pero ninguno de los algoritmos optimiza con esta información. Es interesante explorar si se pueden lograr algoritmos mejores que utilicen explícitamente el hecho de que las regiones forman una partición.

Trazado inicial. Otro aspecto del problema que no consideramos fue el trazado inicial. Es sabido que determinar el trazado inicial es un paso muy importante (hemos dedicado un capítulo entero a esto en la primera parte). Sin embargo, nosotros decidimos considerar un trazado al azar[1]que, como se ha visto, es lo que se usa en la mayoría de los casos. Igualmente es muy interesante en nuestro caso porque existe cierta información sobre la posición de los nodos que en otros casos no la hay, y esto se debe a que sabemos que representan regiones, y que deben

[1]El trazado inicial que se usa es al azar dentro de la región de cada vértice.

estar dentro. El trazado inicial más intuitivo sería colocar todos los nodos en sus respectivos centros de región, pero no podemos garantizar que esto traiga mejores resultados (en un principio no nos parece que así sea) y como no lo hemos estudiado en profundidad se prefirió no incluirlo en este trabajo. Una posible variante de esta idea podría ser que el nodo se encuentre a una distancia r de su centro, definiendo r al azar con cierta distribución de probabilidad. De esta forma se podría dar más importancia al centro pero evitando caer siempre en los mismos óptimos locales.

Fuerzas angulares. Otras fuerzas que estudiamos son las fuerzas angulares. Es un criterio estético importante en el trazado de grafos maximizar los ángulos que forman las aristas incidentes a un mismo vértice. En nuestro caso particular también creemos que es muy importante, pero es una cuestión delicada. Si un nodo se encuentra conectado por dos aristas claramente en el problema general lo más estético es que se encuentren a 180 grados, pero si las regiones se encuentran en forma de triángulo podría preferible que el ángulo no sea ese. Esto lleva a la idea de que hay cierta relación entre el ángulo ideal de las aristas y el de los centros de las regiones correspondientes. Creemos que profundizar en criterios que contemplen ángulos puede ser uno de los trabajos futuros más fructíferos, al menos en lo que respecta a mejorar la calidad de los trazados.

Otros puntos centrales. Sería interesante probar los resultados que se obtienen al considerar otros centros, como podría ser el punto tal que minimiza la suma de las distancias a los vértices, o el centro de la mayor circunferencia inscripta, entre otros. La gran cantidad de centros existentes deja abierto un enorme campo de pruebas a realizar. Es importante notar que algunos centros son difíciles de calcular, así que se debe tener cuidado de no considerar alguno que termine haciendo que el cálculo del centro sea más costoso que el algoritmo en sí mismo.

Otros. Se puede ver que salvo DHR^2 y el trazado inicial, todos nuestros aportes son determinísticos. Hay ciertos aspectos que podrían tener un factor estocástico pero debe ser estudiado con mucho cuidado antes de hacerlo.

Aparte de todas las ideas que quedan por explorar gracias a ser un problema poco tratado, también se pueden mejorar muchas cosas de nuestros propios algoritmos. Tal es el caso de SET que consideramos que podría solamente triangular con las aristas que se encuentren a una distancia menor que cierto umbral r. El orden de complejidad no se vería afectado pero creemos que puede favorecer a la estética del grafo.

Dentro de los posibles trabajos a futuro, otro es probar qué resultados se pueden obtener con diversos valores de λ para DHR^2 (en la distancia de compromiso). En nuestras pruebas dio los mejores resultados usar $\lambda = 0,5$, pero pueden existir familias de grafos o de regiones para los cuales cierto valor se comporte mejor que otro. Es importante notar que así como en el problema general se buscan familias de grafos, en nuestros problema es mayor la diversidad de casos, visto que existen familias de grafos y familias de regiones asociadas, lo cual hace que las posibles entradas para el algoritmo sean muchas más.

Apéndice A

Implementación de los algoritmos

Este apéndice tiene como objetivo explicar el funcionamiento del programa realizado en Java que implementa todos los algoritmos propuestos y algunos de los clásicos (para poder comparar).

La aplicación fue construida en base a la herramienta de dominio público VGJ (*Visualizing Graphs with Java*)[1]. La misma provee un entorno en Java que permite el ingreso de grafos, junto a algunos algoritmos de trazado. La aplicación que presentamos con la implementación de nuestros algoritmos mantiene la interfaz gráfica de VGJ, pero se ha quitado todo lo que no se aplica a nuestro problema de trazado de grafos con regiones, de la misma forma que se agregó todo lo que fue necesario para permitir el ingreso de las regiones y la configuración de los algoritmos.

El programa está encapsulado en un archivo `.jar` (TGrafosConRegiones.jar) y requiere tener instalada la máquina virtual de Java (J2SE JRE)[2]. El programa en sí mismo no requiere ningún tipo de instalación. Para iniciarlo hay que "ejecutar" el archivo `.jar` (que debe estar asociado con la máquina virtual de Java).

A continuación explicaremos brevemente cómo utilizar la aplicación. El programa se ve como se presenta en la Figura A.1.

El panel de la derecha (1) es un área en blanco donde se pueden dibujar los grafos para probar los algoritmos o guardarlos en formato GML. El panel de la izquierda (2) consta de 5 opciones que describiremos a continuación:

1. *Create Nodes* permite crear nuevos nodos con sólo hacer clic en el área de dibujo.

2. *Create Edges* permite crear aristas haciendo clic en un nodo ya existente y luego clic en otro nodo existente. De esta forma se genera una arista entre ambos nodos.

3. *Select Nodes* permite seleccionar un nodo para operar sobre él. Al momento de seleccionarlo se puede ver en la barra de estado (3) la posición actual del nodo seleccionado y su nombre interno.

4. *Select Edges* permite seleccionar una arista para poder operar sobre ella.

5. *Select Nodes or Edges* permite seleccionar de forma indistinta un nodo o una arista para operar sobre ellos.

[1] La herramienta VGJ puede descargarse gratuitamente desde http://www.eng.auburn.edu/department/cse/ research/graph_drawing/vgj.html.

[2] Disponible para bajar en http://java.sun.com.

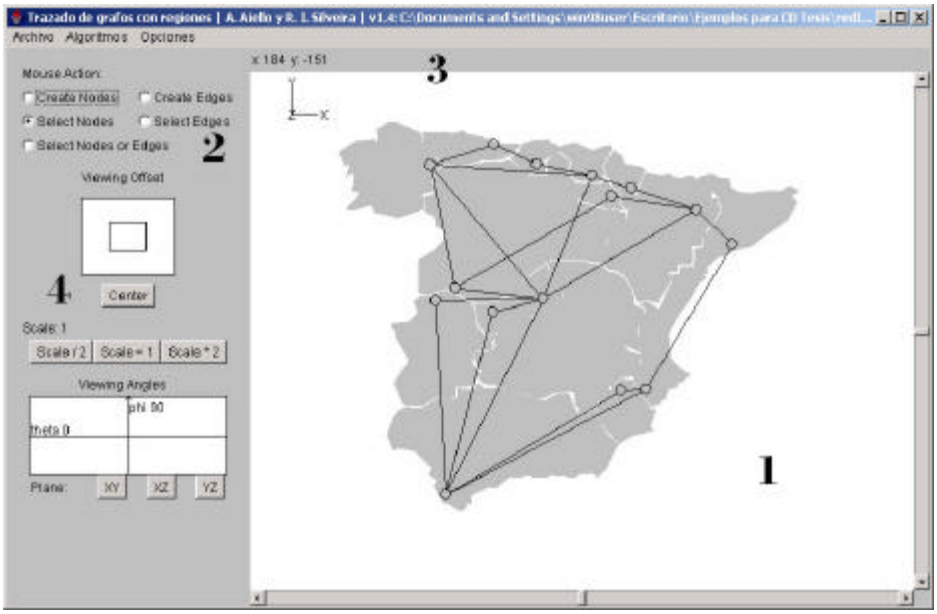


FIGURA A.1. Pantalla principal de la aplicación.

Las operaciones válidas sobre un objeto seleccionado no son muchas pero son las necesarias para trabajar. Éstas se detallan en la siguiente tabla:

Acción	*Forma de realizar la acción*
Eliminar	Presionando la tecla SUPR.
Mover	Arrastrando con el *mouse*.
Definir región disco	Presionando la tecla D e indicando el radio con el mouse, al hacer clic el disco queda definido.
Definir región polígono	Presionando la tecla P y haciendo clic donde se desea ubicar cada vértice. Para finalizar presionar ESC.

No existe un método directo para ingresar regiones en forma de segmentos, pero esto es debido a que los segmentos se ingresan como un polígono de sólo dos vértices.

Es importante notar que al crear un nuevo nodo automáticamente se lo crea con una región circular de cierto radio. Esto se debe a que todos nuestros algoritmos propuestos tienen como precondición que todo nodo debe tener asignada una región, y se tomó esa como región predeterminada.

Por último, la parte (4) del panel izquierdo sirve para realizar *zoom* y de esta forma poder visualizar mejor los grafos.

El programa consta de tres menúes. El primer menú (Archivo) permite abrir y guardar archivos en formato GML, los cuales contienen información de grafos anteriormente realizados. Esto permite poder generar pruebas y luego almacenarlas para su posterior uso. Cuando se guarda un archivo GML automáticamente se genera un archivo `.rgn` que guarda las regiones asignadas para el grafo guardado. Otras dos opciones que se pueden encontrar en el menú son "Cargar imagen de fondo..." que permite poner de fondo una imagen para de esta forma mostrar el grafo de una forma más amena y cercana a su contexto (ej. en el ejemplo de Argentina (Figura. 16.16) podría ponerse de fondo un mapa real de la República Argentina). Esta opción también fue usada para poder dibujar los mapas. La última opción permite exportar el trazado del grafo a formato *Postscript* (usado para generar las imágenes de este trabajo).

El siguiente menú – "Algoritmos" – contiene los algoritmos más importantes presentados en este trabajo. Desde ahí se puede seleccionar un algoritmo *random*, cómodo para generar casos de prueba, SE tal como fue presentado en la sección 4.1, SET , nuestro algoritmo de triangulación presentado en la sección 15.5, DH presentado en la sección 4.5, SER^2 y DHR^2 como fueron descriptos en la sección 15.6. Finalmente, los dos algoritmos específicos derivados de KK para regiones que son segmentos: KK para segmentos aproximando con una parábola, y KK adaptado para segmentos (usando las derivadas y Newton-Raphson), como se los explicó en la sección 15.7.

El último menú es donde se pueden configurar todos los parámetros que se comentaron en este trabajo ingresando en la opción "Definir parámetros" y modificando los valores de la ventana que se muestra en la Figura A.2.

Todos los parámetros que se pueden ingresar son numéricos, lo cual permite que si se desea no utilizar alguna fuerza o algún potencial, se le puede asignar 0 a su peso asociado en los parámetros. Aunque todos los parámetros de los algoritmos fueron explicados a lo largo de este trabajo, describiremos brevemente cada uno. Los parámetros se encuentran divididos en tres grupos.

Parámetros generales, comunes a todos los algoritmos

- "Porcentaje mínimo de cambio para continuar": Si la función de costo no se reduce en al menos el porcentaje indicado, el algoritmo finaliza.
- "Umbral para corte": Valor absoluto mínimo que debe valer la función de energía. En caso de valer menos, el algoritmo termina.

Parámetros propios de DH y DHR^2

- "T0 - Temperatura inicial": Temperatura con la que comienza el algoritmo.
- "Gamma - Fracción con que se reduce la temperatura": Fracción en la que se va reduciendo la temperatura en cada iteración.
- "Cr - Repulsión entre nodos": Peso del potencial de repulsión de nodos (ver ecuación 4.16).

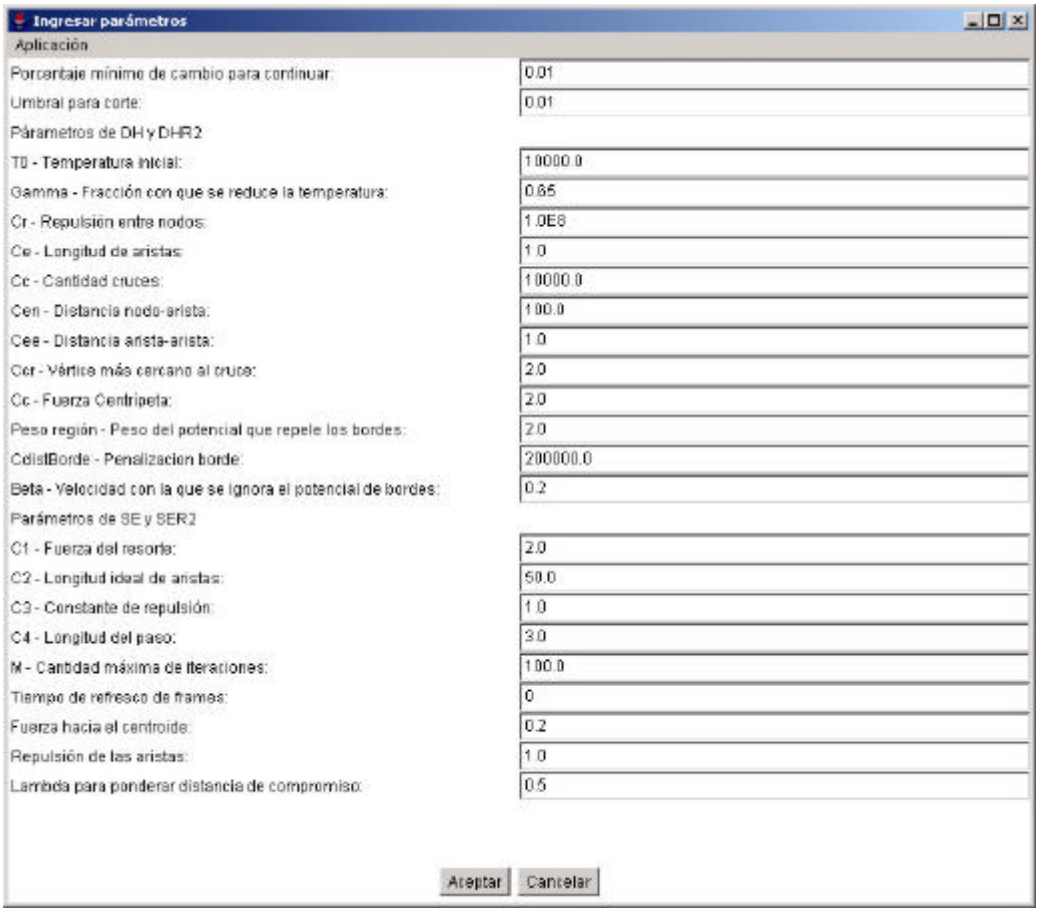


FIGURA A.2. Pantalla de configuración de parámetros.

- "Ce - Longitud de aristas": Peso del potencial que penaliza las aristas de gran longitud (ver ecuación 4.18).
- "Cc - Cantidad cruces": Peso del potencial que indica la cantidad de cruces entre aristas (ver ecuación 4.19).
- "Cen - Distancia nodo-arista": Peso del potencial que penaliza la distancia entre nodos y aristas (ver ecuación 4.20).
- "Cee - Distancia arista-arista": Peso del potencial que penaliza aristas no adyacentes cercanas (ver sección 15.3).
- "Ccr - Vértice más cercano al cruce". Peso del potencial que penaliza la distancia mínima entre cada cruce y los vértices involucrados (ver sección 15.3).
- "Cc - Fuerza Centrípeta": Peso del potencial que penaliza la distancia entre cada vértices y el centro de su región (ver sección 15.2).
- "Peso región - Peso del potencial que repele los bordes" (ver sección 15.4).
- "CdistBorde - Penalización borde". Penalización de vértices muy cercanos a los bordes (ver sección 15.4).
- "Beta - Velocidad con la que se ignora el potencial de bordes". Factor que ajusta cuán lentamente la energía de repulsión de bordes disminuye (ver sección 15.4).

Parámetros del *spring embedder* (SE) y SER[2]

- "C1 - Fuerza del resorte". Constante que regula el peso de las fuerzas atractivas (ver ecuación 4.1).
- "C2 - Longitud ideal de aristas". Longitud ideal de las aristas (sólo se usa en SE) (ver ecuación 4.1).
- "C3 - Constante de repulsión". Constante que regula el peso de las fuerzas repulsivas (ver ecuación 4.2).
- "C4 - Longitud del paso". Constante que regula cuánto mover cada vértice (ver algoritmo en sección 4.1).
- "M - Cantidad máxima de iteraciones". Cantidad máxima de iteraciones (que se combina con los criterios de corte generales).
- "Tiempo de refresco de frames". Tiempo de pausa (en milisegundos) entre cada iteración del algoritmo. Para ver paso a paso cómo evoluciona la corrida del algoritmo.

- “Fuerza hacia el centroide”. Constante que regula el peso de las fuerzas centrípetas (ver sección 15.2).
- “Repulsión de las aristas”. Constante que regula el peso de las fuerzas de repulsión entre aristas (ver sección 15.3).
- “Lambda para ponderar distancia de compromiso”. Parámetro que regula el balance entre la distancia mínima entre regiones y la distancia entre centros (ver ecuación 15.1).

Bibliografía

[BB00] A. Barreto and H. Barbosa. Graph layout using a genetic algorithm. In *VI Simpósio Brasileiro de Redes Neurais*, pages 179–184, 2000.

[BB02] S. Bachl and F. J. Brandenburg. Computing and drawing isomorphic subgraphs. 2002.

[BBS96] J. Branke, F. Bucher, and H. Schmeck. Using genetic algorithms for drawing undirected graphs. Tr-347, University Karlsruhe, D-76128 Karlsruhe, Germany, 1996.

[Beh99] L. Behzadi. An improved spring-based graph embedding algorithm and layout-show: a java environment for graph drawing. Master's thesis, York University, Ontario, Canada., 1999.

[Ber00] F. Bertault. A force-directed algorithm that preserves edge-crossing properties. *Information Processing Letters*, 74:7–13, 2000.

[BF95] I Bruss and A. Frick. Fast interactive 3-d graph visualization. In *Proceedings of Graph Drawing '95*, pages 99–110. . Springer-Verlag, 1995.

[BF98] R. Burden and J. Faires. *Análisis Numérico*. International Thomson Editores, 6th edition, 1998.

[BH86] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[BHR95] F. J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Proceedings of Graph Drawing '95*, pages 76–87. Springer-Verlag, 1995.

[BKL+00] U. Brandes, V. Kääb, A. Löh, D. Wagner, and T. Willbalm. Dynamic www structures in 3d. *Journal of Graph Algorithms and Applications*, 4(3):183–191, 2000.

[BKW03] U. Brandes, P. Kenis, and D. Wagner. Communicating centrality in policy network drawing. *IEEE Transactions on visualization and computer graphics*, 9(2):241–253, 2003.

[BMRW98] T. Biedl, J. Marks, K. Ryall, and S. Whitesides. Graph multidrawing: Finding nice drawings without defining nice. *Lecture Notes in Computer Science*, 1547:347–355, 1998.

[BT98] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In *Proceedings of Graph Drawing '98*, pages 57–71. Springer-Verlag, 1998.

[BT01] S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. In *Proceedings of Graph Drawing 2000*, pages 19–30. Springer-Verlag, 2001.

[BW97] U. Brandes and D. Wagner. A bayesian paradigm for dynamic graph layout. In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 236–247. Springer-Verlag, 1997.

[BW00] U. Brandes and D. Wagner. Using graph layout to visualize train interconnection data. *Journal of Graph Algorithms and Applications*, 4(3):135–155, 2000.

[CLRS01] H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.

[CLY04] J. Chuang, C. Lin, and H. Yen. Drawing graphs with nonuniform nodes using potential fields. 2004.

[Cre01] A. Creek. Forces of nature. angle: An experimental test-bed for graph embedders and the big bang: a new force directed embedder, (BSc Honours Research Report, Canterbury), 2001.

[CSP96] M. Coleman and D. Stott Parker. Aesthetics-based graph layout for human consumption. *Software - Practice & Experience*, 26(12):1415–1438, 1996.

[CT95] I. Cruz and J. Twarog. 3d graph drawing with simulated annealing. In *Proceedings of Graph Drawing '95*, pages 162–165. Springer-Verlag, 1995.

[CY02] M. Chuang and H. Yen. On nearly symmetric drawings of graphs. 2002.

[DE02] T. Dwyer and P. Eades. Visualising a fund manager flow graph with columns and worms. In *Proceedings of the 6th International Conference on Information Visualisation (IV '02)*, pages 147–152. IEEE Computer Society Press, 2002.

[DETT99] G. DiBattista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, first edition, 1999.

[dFPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 1(10):41–51, 1990.

[DH96] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[Dwy00] T. Dwyer. Three dimensional uml using force directed layout. Honours Thesis, 2000.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, (42):149–160, 1984.

[Ead92] P. Eades. Drawing free trees. *Bulletin of the Institute for Combinatorics and its Applications*, 5:10–36, 1992.

[EH00] P. Eades and M. L. Huang. Navigating clustered graphs using force-directed methods. *Journal of Graph Algorithms and Applications*, 4(3):157–181, 2000.

[EHK+04a] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Exploring the computing literature using temporal graph visualization. 2004.

[EHK+04b] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Graphael: Graph animations with evolving layouts. 2004.

[EL00] P. Eades and X. Lin. Spring algorithms and symmetry. *Theoretical Computer Science*, 240(2):379–405, 2000.

[ELMS91] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics '91*, pages 24–33, 1991.

[EMW86] P. Eades, B. McKay, and N. Wormald. On an edge crossing problem. In *Proceedings of the Ninth Australian Computer Science Conference*, pages 327–334, 1986.

[ET03] R. Erbacher and Z. Teng. Analysis and application of node layout algorithms for intrusion detection. In *Proceedings of SPIE–IST*, pages 160–170, 2003.

[FCW67] C. Fisk, D. Caskey, and L. West. Accel: Automated circuit card etching layout. In *Proceedings of the IEEE, 55*, pages 1971–1982, 1967.

[Fin03] B. Finkel. Curvilinear graph drawing using the force-directed method. Honors Theses, May 2003.

[FLM95] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of DIMACS International Workshop, GD '94*, pages 388–403. Springer-Verlag, 1995.

[FR91] T. M. J. Fruchterman and E. M. Reingold. Graph-drawing by force directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[Fri94] A. Friedlander. *Elementos de programação não-linear*. Editora da Unicamp, 1994.

[FT04] Yaniv Frishman and Ayellet Tal. Dynamic drawing of clustered graphs. In *Proceedings of 10th IEEE Symposium on Information Visualization (InfoVis '04)*, 2004.

[FvDFH96] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: principles and practice*. Addison-Wesley, 2nd edition, 1996.

[GGK00] P. Gajer, M. Goodrich, and S. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In *Proceedings of Graph Drawing 2000*, 2000.

[GK00] P. Gajer and S. Kobourov. Grip: Graph drawing with intelligent placement. In *8th Symposium on Graph Drawing*, pages 222–228, 2000.

[GMW81] P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, 1981.

[GN98] E.R. Gansner and E.R. North. Improved force-directed layouts. In *Proceedings of the 6th International Symposium on Graph Drawing*, pages 364–373. Springer-Verlag, 1998.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search Optimzation and Machine Learning*. Addison-Wesley, 1989.

[Hac04] Junger M. Hachul, S. Drawing large graphs with a potential field based multilevel algorithm. 2004.

[Hal70] K. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, (17):219–229, 1970.

[HB04] K. Han and Y. Byun. Three-dimensional visualization of protein interaction networks. *Computers in Biology and Medicine*, 34:127–139, 2004.

[HCE98] M. Huang, R. Cohen, and P. Eades. Online animated graph drawing using a modified spring algorithm. *Journal of Visual Languages and Computing*, 9(6):623 – 645, 1998.

[HE98] M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Proceedings of the 6th International Symposium on Graph Drawing*, pages 374–383. Springer-Verlag, 1998.

[HH99] R Hadany and D. Harel. A multi-scale algorithm for drawing graphs nicely. In *25th Workshop on Graph-Theoretic Concepts in Computer Science (WG 1999)*, pages 262–277, 1999.

[HK02a] D. Harel and Y. Koren. Drawing graphs with non-uniform vertices. In *Proc. of Working Conference on Advanced Visual Interfaces (AVI'02)*, pages 157–166. ACM Press, 2002.

[HK02b] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. In *Proceedings of Graph Drawing 2002*, pages 207–219. Springer-Verlag, 2002.

[HM98] W. He and K Marriott. Constrained graph layout. *Constraints*, 3(4):289–314, 1998.

[HMMS02] T. Hansen, K. Marriott, B. Meyer, and P. Stuckey. Flexible graph layout for the web. *Journal of Visual Languages and Computing*, 13(1):35–60, 2002.

[HMN04] S-H. Hong, D. Merrick, and H.A.D.d. Nascimento. The metro map layout problem. In N. Churcher and Eds. Churcher, C., editors, *Australasian Symposium on Information Visualisation, (invis.au'04)*, pages 91–100. Australian Computer Society, 2004.

[HS95] D. Harel and M. Sardas. Randomized graph drawing with heavy-duty preprocessing. *Journal of Visual Languages and Computing*, 6:233–253, 1995.

[JD88] A. Jain. and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[KCH01] Y. Koren, L. Carmel, and D. Harel. Ace: A fast multiscale eigenvectors computation for drawing huge graphs. Technical Report MCS01-17, The Weizmann Institute of Science, 2001.

[KGJV83] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by simulated annealing. *Science*, (220):671–680, 1983.

[KH04] Y. Koren and D. Harel. Axis-by-axis stress minimization. In *Proc. Graph Drawing 2003*, pages 450–459. Springer-Verlag, 2004.

[Kha86] O. Khatib. Real time obstacle avoidance for manipulators and mobile robots. *Int. Journal of Robotics Research,*, 5(1):90–98, 1986.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, (31):7–15, 1989.

[KKR95] T. Kamps, J. Kleinz, and J. Read. Constraint-based spring-model algorithm for graph layout. In *Proceedings of Graph Drawing '95*, pages 349–360. Springer-Verlag, 1995.

[Kor03] Y. Koren. On spectral graph drawing. In *Proceedings of The 9th International Computing and Combinatorics Conference (COCOON'03)*, 2003.

[KW01] M. Kaufmann and D. Wagner, editors. *Drawing graphs: methods and models*, volume 2025 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[LMR98] K Lyons, H Meijer, and D Rappaport. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications*, 2(1):1–24, 1998.

[Man90] J. Manning. Computational complexity of geometric symmetry detection in graphs. In *Proc. Great Lakes Computer Science Conference*. Spring-Verlag, 1990.

[MO85] Z. Miller and J. Orlin. Np-completeness for minimizing maximum edge length in grid embeddings. *Journal of Algorithms*, (6):10–16, 1985.

[MR02] Paul Mutton and Peter Rodgers. Spring Embedder Preprocessing for WWW Visualization. In *Proceedings Information Visualization 2002*. IVS, IEEE, July 2002.

[MRS95] B. Monien, F. Ramme, and H. Salmen. A parallel simulated annealing algorithm for generating 3d layouts of undirected graphs. In *Proceedings of Graph Drawing '95*, pages 396–408. Springer-Verlag, 1995.

[Noa04] A.Ñoack. An energy model for visual graph clustering. In *Proceedings of the 11th International Symposium on Graph Drawing (GD 2003)*, pages 425–436. Springer-Verlag, 2004.

[Ost96] D. I. Ostry. Some three-dimensional graph drawing algorithms. Master's thesis, Dept. Comput. Sci. and Soft. Eng., Univ. Newcastle, October 1996.

[PCA02] H. Purchase, D. Carrington, and J. Allder. Empirical evaluation of aesthetics-based graph layout. *Empirical Software Engineering*, 7:233–255, 2002.

[Pur02] H. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13:501–516, 2002.

[QE01] A. Quigley and P. Eades. Fade: Graph drawing, clustering, and visual abstraction. In *Proceedings of Graph Drawing 2000*, pages 197–210. Springer-Verlag, 2001.

[QJB79] N. Quinn Jr and M. Breuer. A force directed component placement procedure for printed circuit boards. *IEEE Transactions on Circuit and Systems*, CAS-26(6):377–388, 1979.

[RO98] A. Rosete and A. Ochoa. Genetic graph drawing. In *Proceedings of the 13th International Conference on Applications of Artificial Intelligence in Engineering*, 1998.

[RT81] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[Sch90] W. Schnyder. Embedding planar graphs on the grid. In *Proc. 1st ACM-SIAM Symp. Discrete Algorithms*, pages 138–148, 1990.

[SM95] K. Sugiyama and K. Misue. Graph drawing by the magnetic spring model. *Journal of Visual Languages and Computing*, 6(3):217–231, 1995.

[SSC03] L. Sun, S. Smith, and T.P. Caudell. A low complexity recursive force-directed tree layout algorithm based on the lennard-jones potential. Technical Report EECE-TR-03-001, University of New Mexico, 2003.

[SSL00] F. Simon, F. Steinbruckner, and C. Lewerentz. 3d-spring embedder for complete graphs. Technical Report 11/00, Brandenburg University of Technology at Cottbus, 2000.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 2(11):109–125, 1981.

[Tun94] D. Tunkelang. A practical approach to drawing undirected graphs. Technical Report CS-94-161, Carnegie Mellon University School of Computer Science, 1994.

[Tun99a] D. Tunkelang. Jiggle: Java interactive graph layout environment. In *Proceedings of Graph Drawing '98*, pages 413–422. Springer–Verlag, 1999.

[Tun99b] D. Tunkelang. *A numerical optimization approach to general graph drawing.* PhD thesis, Carnegie Mellon University, January 1999.

[Tut60] W. T. Tutte. Convex representations of graphs. *Proceedings of the London Mathematical Society, Third Series*, 10:304–320, 1960.

[Tut63] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society, Third Series*, (13):743–768, 1963.

[Wal03] C. Walshaw. A multilevel algorithm for force-directed graph-drawing. *Journal of Graph Algorithms and Applications*, 7(3):253–285, 2003.

[WB99] R. Wilson and R. Bergeron. Dynamic hierarchy specification and visualization. In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'99)*, pages 65–72. IEEE Computer Society, 1999.

[WM95] X. Wang and I. Miyamoto. *Generating customized layouts*, pages 504–515. Springer-Verlag, 1995.

[WW99] Benjamin W. Wah and Tao Wang. Simulated annealing with asymptotic convergence for nonlinear constrained global optimization. In *Principles and Practice of Constraint Programming*, pages 461–475, 1999.

[WWM82] G. Wipfler, M. Wiesel, and D. Mlynski. A combined force and cut algorithm for hierarchical vlsi layout. In *Proceedings of the 19th conference on Design automation*, pages 671 – 677. IEEE Press, 1982.