Tesina de grado

Posicionamiento Automático de Etiquetas en Grafos

Alumno: Gerardo Huck
Directores: Guido Macchi, Dante Zanarini

Licenciatura en Ciencias de la Computación
Unversidad Nacional de Rosario

Agosto de 2014

Resumen

La ubicación de etiquetas en posiciones convenientes es un problema frecuente dentro del proceso de dibujado de grafos. El objetivo de un buen posicionamiento de etiquetas es mostrarlas armoniosamente con el grafo, de modo de que la información que proveen sea fácil de interpretar y claramente identificable.
En este trabajo, estudiamos los métodos existentes de *graph layout* y las técnicas para posicionamiento automático de etiquetas de nodos, y descubrimos que si bien hay amplio trabajo previo en ubicar etiquetas sobre una geometría fija de nodos y aristas, no existen practicamente antecedentes de métodos generales que incorporan esta lógica en el proceso de layout de nodos y aristas.
Finalmente, extendimos un algoritmo de graph layout existente de modo de que ubique a las etiquetas de nodos. Este nuevo método puede ser empleado tanto en un grafo ya posicionado, así como también utilizado para posicionar simultáneamente nodos, aristas y etiquetas, con una buena performance computacional y resultados de buena calidad para problemas de tamaños pequeños a medianos.

Índice general

Índice de figuras

Capítulo 1

Introducción

Cuando se trabaja con gran cantidad de información, una técnica habitual para poder hacer frente a la complejidad inherente de la misma es representarla en forma de redes. Dichas redes (Grafos) muestran distintas unidades conceptuales (Nodos) y sus interrelaciones (Aristas), logrando así representar gráficamente una gran cantidad de información en forma sucinta. Se encuentran modelos hechos con grafos en numerosas áreas, habiendo ejemplos sumamente variados entre los que se incluyen *diagramas UML*, *redes de PERT*, *diagramas de procesos*, *organigramas*, *redes biológicas*, sólo por citar algunos ejemplos.

Visualizar correctamente grafos se ha convertido en una necesidad crítica en muchos dominios, como ser ingeniería de software, telecomunicaciones, software corporativo, comercio electrónico, reprentación del conocimiento, bioinformática y análisis financiero.

Es frecuente que al realizar un diagrama sea esencial agregar etiquetas a diversos elementos del mismo. Las etiquetas pueden usarse para agregar información o para aclarar el significado de estructuras complejas, y pueden clasificarse en tres categorías: *etiquetas de aristas*, *etiquetas de nodos* y *etiquetas de área*.

Hoy en día, a la hora de elaborar diagramas, aún cuando se utilice alguna herramienta que automatice parte del proceso, es frecuente que deba modificarse manualmente la ubicación de etiquetas a fin de mejorar la legibilidad o la estética. Dicha tarea, además de resultar tediosa, insume gran cantidad de tiempo.

Es por las razones antes esgrimidas que resulta deseable tener nuevos métodos que automaticen en mayor medida esta etapa del proceso de confección de diagramas. En particular, en este trabajo nos enfocamos en el problema del posicionamiento automático de etiquetas de nodos en grafos.

El contenido se encuentra organizado de la siguiente forma: en el capítulo 2 se presentan el estado del arte y los conceptos teóricos, que permiten luego en el capítulo 3 formular un nuevo método. En el capítulo 4 detalla la implementación realizada, cuyos resultados son resumidos en el capítulo 5. En el capítulo 6 se analiza todo el trabajo realizado, y se presentan las conclusiones a las que hemos arribado. Finalmente, en los apéndices A y B se incluyen porciones del código fuente generado en la implementación.

Capítulo 2

Conceptos Previos

En este capítulo cubriremos brevementes distintos conceptos que son necesarios a la hora de presentar los avances de este trabajo.

Un *grafo* puede ser definido como un conjunto de *nodos* y un conjunto de *aristas*, tal que las *aristas* describen la existencia de una determinada relación entre dos *nodos*. Dibujar un grafo puede ayudar a comprender la estructura de estas relaciones de una forma más efectiva que simplemente observando los datos en formato de tabla o matriz.

El etiquetamiento automático en grafos puede ser hecho tanto al mismo tiempo que se efectúa el dibujo (layout) del grafo, como también en forma posterior (es decir, sobre una geometría fija de nodos y aristas).
El primer enfoque es claramente menos restrictivo y tiene el potencial de producir mejores resultados. Sin embargo, debido a que cada tipo de grafo o estilo de layout puede requerir un método diferente para incorporar el etiquetado como parte de su proceso de layout, no ha sido muy estudiado [37]. Algunos resultados relacionados con este tema pueden encontrarse en [4, 5, 27] (todos trabajan sólo con layouts ortogonales). Debido a que hay muy pocos antecedentes en el tema, ahondaremos en el problema relacionado de cómo dibujar un grafo en la sección 2.1.
El segundo enfoque (posicionar las etiquetas sobre un grafo ya dibujado), en cambio, ha capturado mucha atención durante los últimos 15 años, habiendo numerosa bibliografía sobre el tema. Cubriremos el tema en la sección 2.2.

2.1. Algoritmos de "Graph Layout"

Los algoritmos de *graph layout* (dibujado o disposición de grafos) convierten la topología de adyacencia de nodos en un posicionamiento geométrico de los mismos. Los dibujos de grafos generalmente representan los nodos como puntos o íconos en 2 o 3 dimensiones, en tanto las aristas del grafo se convierten en líneas o arcos que unen dichos nodos [16].

Dado que la forma en que se dibuja un grafo tiene un impacto significante en cómo lo comprendemos, no alcanza con sencillamente 'dibujar' el grafo, sino que hay que elegir cuidadosamente un diseño que trasmita con claridad la estructura existente. La visualización de grafos o redes no debe ser un fin en sí mismo, sino que que debe ayudar al análisis y comprensión del mismo [15].

2.1.1. Criterios de dibujado

Numerosas métricas han sido definidas para expresar la calidad de un dibujo de un grafo, buscando encontrar una función que evalúe tanto el resultado estético como su usabilidad y facilidad de comprensión [10]. Algunos de los criterios más frecuentemente tenidos en cuenta a la hora de definir estas métricas son los siguientes:

- *Distribución de nodos.*
  Se prefiere una distribución de nodos lo más uniforme posible.

- *Distancia entre nodos.*
  Idealmente los nodos no relacionados se ubican considerablemente más lejos unos de los otros que cuando lo están.

- *Longitud de aristas.*
  Es importante que las aristas tengan una longitud lo más homogénea posible, también minimizando la suma total de las longitudes.

- *Tamaño y forma de la cápsula rectangular.*
  En general se prefiere grafos con cápsula más pequeña (en área) y relación alto/ancho más cercana a 1.

- *Simetría.*
  En caso de existir grupos simétricos en un grafo, se busca que éstos puedan ser fácilmente reconocibles en el gráfico resultante.

- *Número de cruces entre aristas.*
  Si el grafo es planar, es sumamente deseable que una representación gráfica del mismo no tenga cruces entre aristas. Como el caso general es que un grafo no sea planar, lo que se busca es minimizar la cantidad de cruces de aristas en su representación gráfica.

- *Simplicidad de aristas.*
  Si bien se pueden utilizar distintos tipos de aristar (líneas rectas, curvas, polilíneas, etc), se considera mejor una solución que presente aristas lo más sencillas posibles (como menor cantidad de quiebres en el caso de una polilínea).

En general, los problemas de optimización asociados con estos criterios estéticos son **NP-hard** [19, 20].

2.1.2. Orígenes

Si bien han transcurrido más de 50 años desde que Tutte propuso su método de "baricentro" [36, 35], cómo dibujar un grafo sigue siendo un problema abierto y que actualmente sigue recibiendo considerable atención [15]. Se ha argumentado que de hecho no existe una "mejor manera de dibujar un grafo", sino que ésta depende de qué características del mismo se quieran resaltar [6].

El dibujado de grafos ha acumulado un gran corpus de trabajo, dentro del cual se destaca que cuenta con su propio simposio "Graph Drawing", el cual se ha realizado en forma inninterrumpida durante más de 20 años y que ve sus artículos publicados en "Journal of Graph Algorithms and Applications". Otro elemento a destacar es el libro publicado en 1999 "Graph drawing: algorithms for the visualization of graphs" [34], considerado como la principal referencia a la hora de introducirse en el tema.

2.1.3. Algoritmos "Force Directed"

La mayoría de los algoritmos de *graph layout* se basan en el paradigma de modelar el grafo como un sistema físico en el que los nodos son atraídos y repelidos por distintas fuerzas. A este paradigma se lo llama *force directed* (basados o dirigidos por fuerzas) [15]. Los algoritmos *force directed* fueron de los primeros creados para atacar el problema de *graph layout* y los más utilizados en la actualidad. Si bien existen algunos antecedentes que podrían ser considerados como precursores, como por ejemplo el trabajo de Tutte [36, 35], existe amplio consenso en que el primer algoritmo genuínamente *force directed* es el *Spring Embedded* [12], creado por Eades en 1984.

Existen dos enfoques utilizados en los algoritmos *force directed*:

- **Basados en fuerzas de resorte y eléctricas** (Spring-electrical based)
- **Basados en Energía** (Energy-based)

A continuación, se datallan las características principales de ambas familias de algoritmos y se nombran los ejemplares más sobresalientes.

Basados en fuerzas de resorte y eléctricas

El algoritmo original desarrollado por Eades [12] es la inspiración para esta clase de algoritmos. Para lograr *graph layout*, se efectúa una simulación de un sistema físico. Este sistema consiste (en el caso bidimensional) en modelar los nodos como aros de acero, y las aristas como resortes. El sistema es puesto en una configuración original aleatoria y luego se simula su evolución hasta que (idealmente) alcanza un punto de equilibrio. Las fuerzas que intervienen en la simulación son las de atracción entre nodos conectados, y de repulsión entre aquellos que no lo están. Algunas variantes incorporan otras fuerzas, como por ejemplo una *fuerza de gravedad*, que atrae a todos los nodos hacia el centro del área de trabajo, evitando que los nodos se dispersen demasiado. A continuación, se describen brevemente algunas de los exponentes más destacados que han surgido en esta área.

- *Spring Embedded*, Eades [12].
  En esta versión pionera, la fuerza resultante en cada nodo era la suma de las fuerzas atractivas (de las aristas que lo tienen como extremo) y de las repulsivas (computadas entre el nodo en cuestión y todos los demás).

  La fuerzas atractivas tienen la forma

  $$f_a = c_1 \log \frac{d}{c_2}$$

  donde d es la longitud de la arista y c_1, c_2 son constantes.

  En tanto las fuerzas repulsivas quedan determinadas por la siguiente fórmula

  $$f_r = \frac{c_3}{d^2}$$

  donde d es la distancia entre nodos y c_3 es una constante.

- *Fruchterman y Reingold* [14].
  Un objetivo que se plantearon los autores es que el algoritmo funcione bien sin necesidad de que el usuario ajuste sus parámetros. Dado que la convergencia a una configuración estable no está garantizada, se sugiere 50 como el número de iteraciones a realizar. Otro objetivo es la rapidez; restringiendose a grafos de no más de 100 nodos, típicamente lleva menos de 1 segundo realizar el *layout*.

  Uno de las modificaciones que introdujeron para acelerar el procesamiento fue el **GVA** (*Grid Variant Algorithm*). El área de dibujado es dividida por una grilla, y sólo se calculan las fuerzas repulsivas entre los nodos que compartan la misma casilla, o estén situados en casillas adyacentes. Ésto fue luego modificado para tener en cuenta sólo las fuerzas entre nodos que estén a una distancia no mayor a un radio dado.

  La fuerzas repulsivas resultan entonces

$$f_r = \frac{k^2}{d}\, u(2k-d), \quad \text{donde } u(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

  donde d es la distancia entre los dos nodos, $k = C\sqrt{\frac{\text{area}}{\text{numeroDeNodos}}}$ es la distancia optima entre 2 nodos (dada el área disponible y la cantidad de nodos), y C es una constante obtenida experimentalmente.

  Ésto acelera el algoritmo sin repercutir en la calidad del resultado, e incluso en algunos casos puede evitar que los nodos queden demasiado dispersos, logrando así un incluso mejor *layout*. Los autores incorporan otra característica novedosa al algoritmo, que consiste en tener una *temperatura* global, que va descendiendo a medida que progresa la ejecución, y que limita cuánto puede moverse un nodo en cada iteración de la simulación. En la figura 2.1 se puede observar un ejemplo del resultado de este algoritmo.

- *GEM* [13].
  Frick et al publicaron este algoritmo en 1995, cuyo nombre es una abreviación para *Graph Embedder*. Se basa en el trabajo previo de Eades, así como en el de Fruchterman y Reingold, y consiste en una serie de mejoras algorítmicas sobre las variantes anteriores.

  Las principales mejoras introducidas son las siguientes

  - Temperatura local
    Las temperaturas en *GEM* indican la distancia máxima que un nodo puede moverse en una iteración de la simulación física.

    Ya otros autores habían conjeturado que mejoras en la forma de manejar la 'temperatura' permitiría obtener algoritmos mas eficientes [14]. Davidson y Harel [9] fueron los primeros en sugerir un esquema de enfriamiento adaptativo, pero no desarrollaron la idea. *GEM* es el primer algoritmo que lo hace, aunque introduciendo la variante de que el algoritmo se adapa localmente, no requiriendo una única 'temperatura' global.

    Para cada nodo, y en cada iteración, se define una nueva temperatura local que depende de la temperatura anterior y de la probabilidad de que el vértice oscile o sea parte

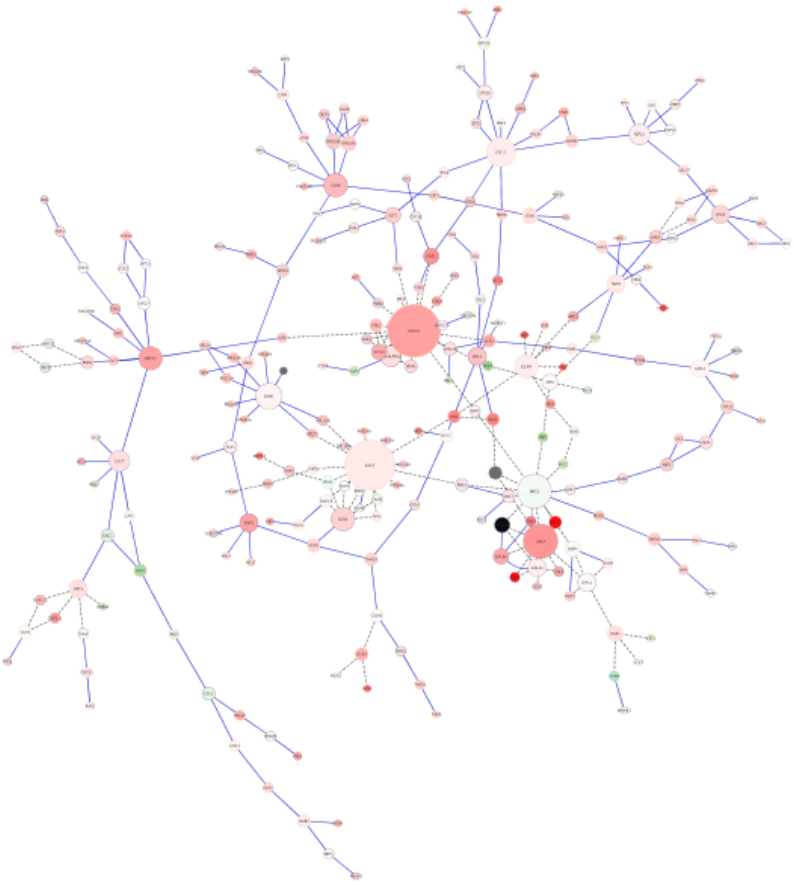


Figura 2.1: Salida del algoritmo *Fruchterman-Reingold*.

de un subgrafo rotatorio. La temperatura sube (permitiendo mayor movimiento) si el algoritmo determina que es probable que el nodo está lejos de su posición final. La temperatura global se define simplemente como el promedio de las temperaturas individuales de los nodos, y por lo tanto, indica cuán estable es el estado actual del *layout*.

- Atracción de los nodos hacia su baricentro
  Se agrega una fuerza "gravitatoria" que atrae a los nodos hacia el baricentro de los mismos. El uso de esta fuerza acelera la convergencia de *GEM*. Además, ayuda a mantener componentes disconexas o poco conexas juntas.

- Detección de oscilaciones y rotaciones
  Las rotaciones ocurren, por ejemplo, cuando se ha hallado la distribución final, pero la temperatura es aún muy alta. Bajo las condiciones apropiadas, un grafo en rotación puede no converger nunca, por lo que bajar la temperatura es lo adecuado. El algoritmo presenta un mecanizmo para detectar rotaciones y bajar la temperatura en consecuencia.

  Un nodo se considera que oscila cuando su impulso actual es opuesto al inmediato anterior. En ese caso, *GEM* asume que el vértice acaba de pasar su posición ideal, y por lo tanto volverá a 'girar' nuevamente. En ese caso, se baja bruscamente su temperatura para acelerar la convergencia.

El algoritmo consiste en dos etapas, la de *inicialización*, y la de *iteración*. La *inicialización* consiste en la asignación de una posición inicial, impulso y temperatura para los vértices, en tanto que la etapa de *iteración* es un bucle que actualiza las posiciones hasta que la temperatura global sea menor que un límite deseado, o hasta que haya expirado un límite de iteraciones.

Según los autores, la principal meta a la hora de diseñar el algoritmo era lograr una velocidad de *layout* interactiva (menos de 2s) para grafos de tamaño mediano (hasta 100 nodos). Es por eso que además de las mejoras antes mencionadas, lo construyeron usando aritmética de enteros en lugar de punto flotante. En la figura 2.2 se pueden ver ejemplos de resultados de este algoritmo.

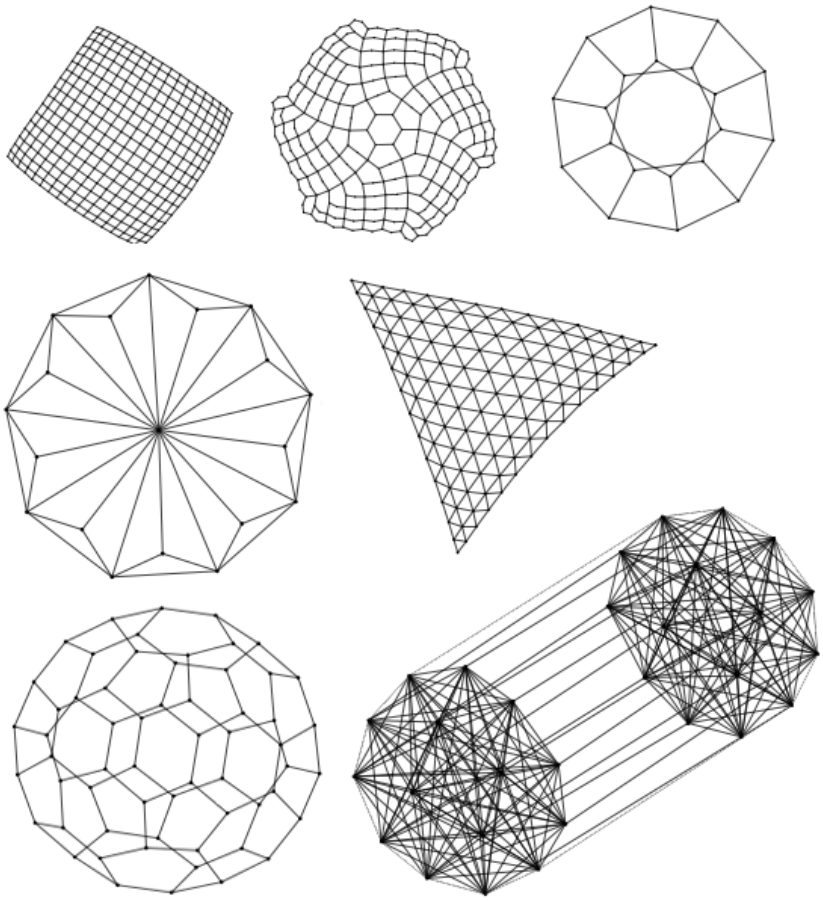


Figura 2.2: Layouts producidos por el algoritmo *GED*.

Basados en Energía

Los *algoritmos de layout basados en energía* abordan el problema desde el punto de vista de optimizar (minimizar) una función objetivo (o *función de energía*). Esta función es la encargada de representar las características deseadas en el *layout*.

Como muchos problemas de optimización, pueden existir *mínimos locales* que difieran del global, produciendo así un resultado no óptimo. Es por eso, que algunas variantes permiten

transicionar temporalmente a estados de mayor energía (menos deseables) con el fin de evitar "quedar atrapados" en un mínimo local.

Los dos métodos más conocidos que usan este enfoque se describen a continuación.

- *Kamada y Kawai [24]*
  Esta variante publicada en 1989 usa un modelo de resortes, con la variante de buscar que la distancia euclídea entre dos nodos cualesquiera sea proporcional a la distancia teórica entre ellos (longitud del camino más corto entre ambos).

  La función de energía se define de modo de medir el *grado de desbalance* en el *layout*, basado en la *ley de Hooke* (la fuerza ejercia por un resorte es linearmente proporcional al desplazamiento del mismo desde su posición de equilibrio). Para optimizarla, es preciso resolver ecuaciones diferenciales, lo que vuelve costoso su cómputo, razón por la cual en cada iteración se actualiza la posición de sólo un nodo.

  Una característica destacable del método es que produce *layouts* muy simétricos y con pocos cruces de aristas. En la figura 2.3 se pueden ver *layouts* realizados con este algoritmo.

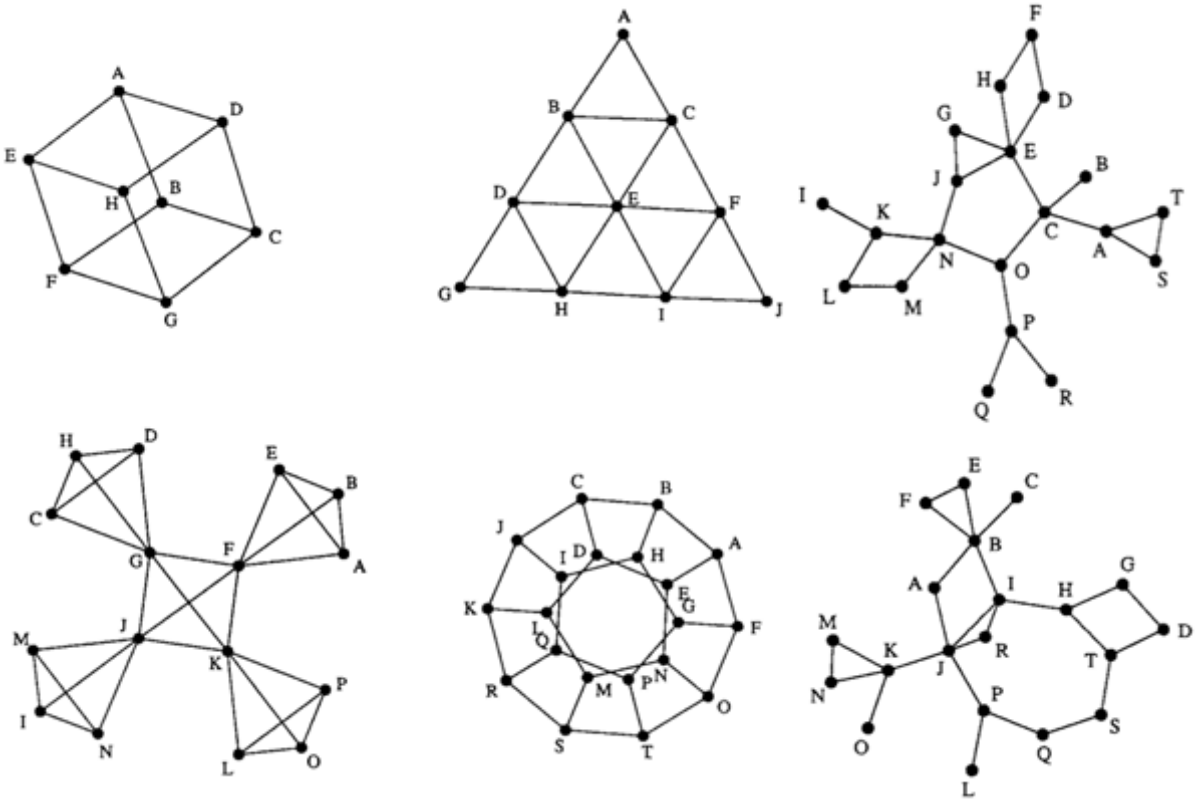


Figura 2.3: Layouts producidas por el algoritmo *Kamada-Kawai*.

- *Davidson y Harel [9]*
  Los autores presentaron en 1996 esta variante, que utiliza la técnica de *recocido simulado* (simulated annealing) para encontrar una disposición que optimize la función de energía. El *recocido* es el proceso de enfriar un fluído líquido lentamente, de modo de que forme

una estructura cristalina de mínima energía. La simulación de este proceso permite la obtención de *layouts* que cumplen con los atributos deseables de longitudes similares de aristas, nodos uniformemente distribuídos y mínimos cruces de aristas. Para ello crearon una función de energía que explícitamente tiene en cuenta esos atributos, utilizando pesos variables que pueden ser modificados para dar mayor o menor reelevancia a cada aspecto.

El algoritmo comienza con un posicionamiento y temperatura global iniciales, y en cada iteración actualiza la posición de un nodo, así como la temperatura. En la figura 2.4 se pueden ver resultados de aplicar esta técnica.

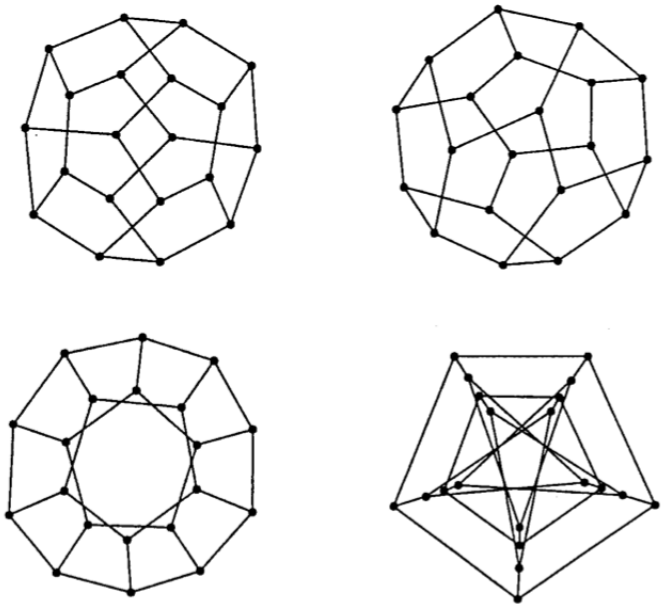


Figura 2.4: Layouts producidas por el algoritmo presentado por *Davidson y Harel*.

2.1.4. Otros algoritmos

Si bien la mayoría de los métodos utilizados para realizar *graph layouts* pertenecen a la familia *force directed*, existen otros que no lo hacen. A continuación describimos los más relevantes.

Reducción dimensional

La *reducción dimensional* (dimension reduction) es el proceso de proyectar información expresada en una mayor cantidad de dimensiones en un espacio con menos dimensiones. El desafío es intentar capturar y expresar en este nuevo espacio parte de la información "perdida" al realizar la proyección.

La mayor parte de las técnicas de *reducción dimensional* para *graph layout* buscan preservar la distancia teórica entre nodos, de modo de que al realizar el *layout* la distancia euclideana entre ellos sea lo más parecida posible a ella.

Existen 2 principales técnicas *reducción dimensional* para *graph layout*: **distance scaling**, y **classical scaling**.

En el caso de **distance scaling**, se calcula una approximación de la diferencia entre la distancia teórica y la euclideana para todos los pares de nodos del grafo. La suma de los cuadrados de todas esas distancias se denomina como la *fatiga* (stress) del layout. Luego, el objetivo es minimizar la *fatiga* mediante algún procedimiento de optimización. La función de fatiga suele ser similar a la utilizada en el método de Kamada-Kawai [24], aunque varía la forma en la que se busca optimizarla. Esta técnica fue utilizada por primera vez por Kruskal y Seery [28] en 1980.

En cambio, la técnica de **classical scaling** se basa en encontrar una solución exacta para el problema de acomodar los nodos de modo de disminuir la *fatiga* del *layout*. Esto se logra a través de una técnica conocida como *descomposición espectral*. Desafortunadamente, la implementación de esta técnica resulta muy costosa en términos computacionales (tiempo cuadrático), por lo cual se emplean técnicas aproximadas para poder ser usables. Una de ellas es *pivot MDS*, propuesto por Brandes y Pich [7, 8], y que logra aproximar los resultados de la técnica original en tiempo lineal.

En la figura 2.5 se puede ver un ejemplo con resultados de estas técnicas.

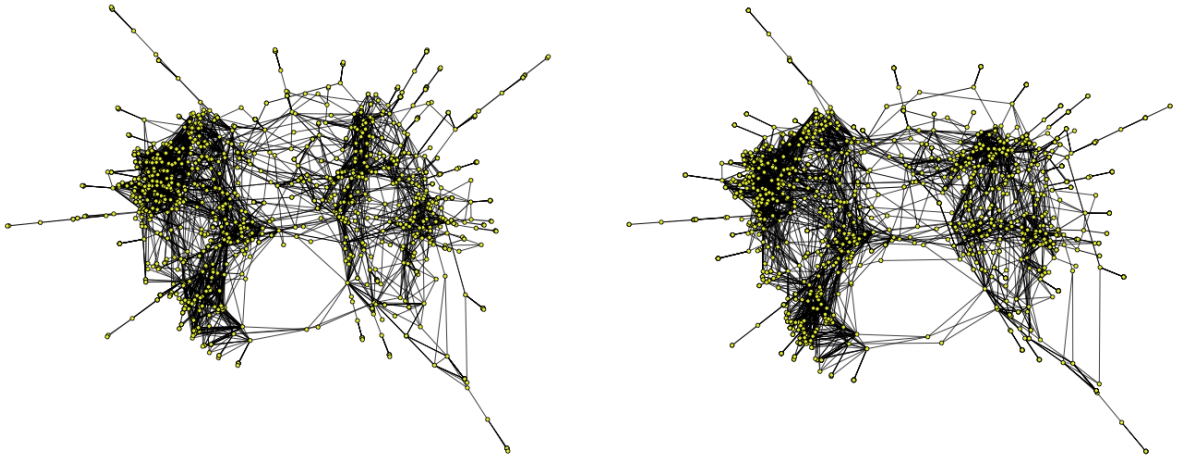


Figura 2.5: Layouts del mismo grafo. En la izquierda usando el método *pivot MDS*. En la derecha usando *classical scaling* sin optimizar.

Otros

Además de las familias de *layouts* ya mencionados, existen numerosos ejemplos que no se pueden clasificar ya que o bien son de dominio específico, son triviales, o no comparten un número considerable de características con ningún otro. A continuación se mencionan algunos de ellos.

- **Circular**
  Desarrollado originalmente por Kar, Madden y Gilbert [25], y luego mejorado por Dogrusöz, Madden y Madden [11], este *layout* está dirigido principalmente a mostrar el agrupamiento *(clustering)* de nodos. Para ello, los agrupa de acuerdo a alguna determinada característica (existen varias opciones disponibles, como biconectividad, grado del nodo, etiquetas, etc), y ubica a los nodos de cada grupo en forma circular. Finalmente, emplea

diversas heurísitcas para reacomodar los nodos de cada grupo, y también ubicar los grupos, de modo de disminuir los cruces de aristas. En la figura 2.6 se pueden ver dos grafos tratados con este algoritmo.

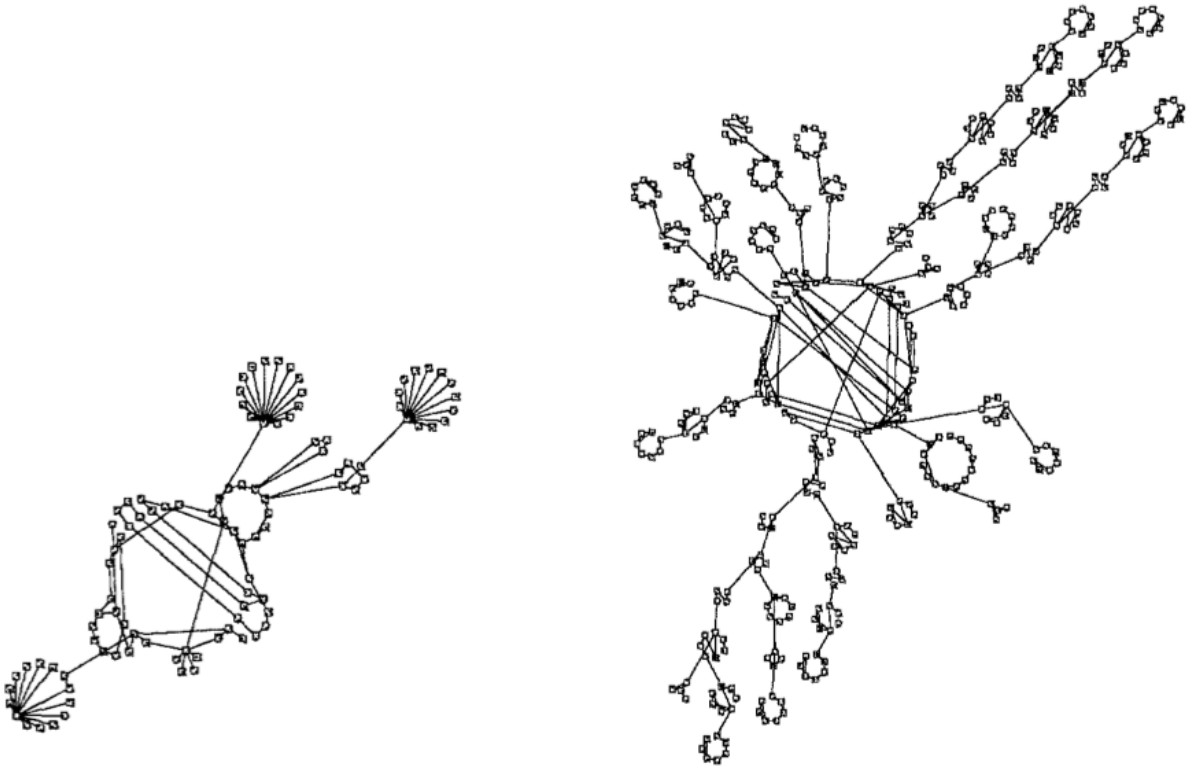


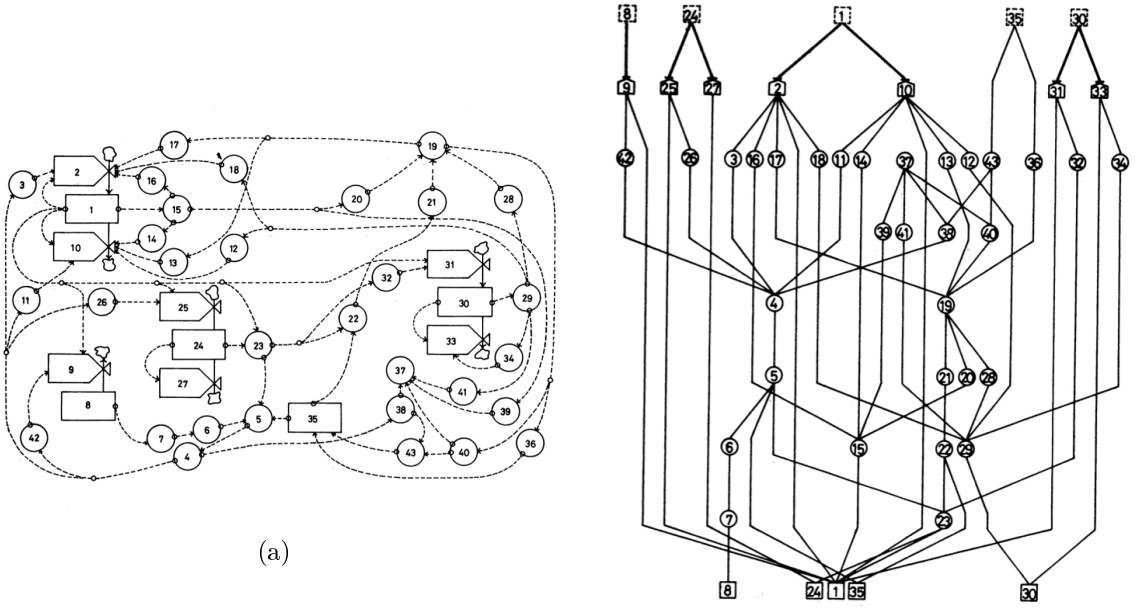
Figura 2.6: Layout Circular. Ejemplos producidos por el método presentado por Dogrusöz, Madden y Madden

- **Jerárquico**
  Este algoritmo publicado en 1982 por Sugiyama, Tagawa y Toda [33] también es conocido por el nombre del primer autor. Busca encontrar una representación que resalte y ponga en evidencia la estructura jerárquica que puede existir entre los nodos de un grafo.
  Para lograr su objetivo, consta de dos etapas básicas. Primero aplica algún método (presentan dos alternativas en su trabajo original) para disminuir el número de cruces de aristas. En la segunda etapa, se aplican varios métodos para mejorar aspectos como la "rectitud" de aristas largas, cercanía de vértices y balance general del grafo. En la figura 2.7 se puede un ejemplo de la salida generada por este método.

2.2. Etiquetamiento de mapas y diagramas

Si bien el etiquetamiento de diagramas (*diagram labeling*) tiene gran relevancia en numerosas áreas de visualización de la información, en el campo de la *Cartografía* ha sido históricamente considerada como fundamental. Es por eso que en esta disciplina se encuentran la mayor parte de los antecedentes en cuanto a técnicas de etiquetamiento, tanto manual como automático.

En el proceso de elaboración de mapas o diagramas se encuentran habitualmente tres tipos de etiquetamientos distintos [18]:



(a) (b)

Figura 2.7: Layout Jerárquico. En (a) entrada, (b) resultado del método.

- etiquetamiento de *área* (ej. océanos, países)
- etiquetamiento de *línea* (ej. rutas, ríos)
- etiquetamiento de *puntos* (ej. ciudades, sitios históricos)

Si bien la tarea de determinar la ubicación óptima de una etiqueta difiere mucho al tratarse de un punto, línea o área en solitario, los tres tipos de etiquetamiento comparten una gran similitud desde un punto de vista combinatorio al tratar de etiquetar múltiples elementos [21]. La complejidad del problema surge debido a que la ubicación de una etiqueta puede tener concecuencias globales debido a las superposiciones (entre dos o más etiquetas) que pueden surgir. Este aspecto combinatorio es independiente de la naturaleza de los elementos siendo etiquetados (punto, línea, área), y es la principal dificultad en la ubicación automática de etiquetas. Es por esto que la mayoría de la bibliografía se concentra en tratar el problema del etiquetamiento de puntos, sin por ello perder generalidad. El problema del etiquetamiento de puntos es conocido por la sigla **PFLP** (del inglés *Point Feature Label Placement*), acrónimo que usaremos de aquí en adelante.

2.2.1. Criterios de etiquetado

Etiquetas bien ubicadas ayudan a la transmisión de información, y mejorar la estética de un diagrama. Sin embargo, es sumamente difícil cuantificar todas las características que los humanos tenemos en cuenta a la hora de juzgar la calidad de un etiquetado [21].

Ubicar una etiqueta a un elemento aislado es un problema trivial, pero se vuelve mucho más complejo cuando vemos limitadas nuestras opciones debido a la presencia de uno o más objetos cercanos en el dibujo. Debemos entonces tener en cuenta no sólo la posición relativa de

la etiqueta y su objeto asociado, sino también como se relacionan ambos con los otros elementos.

Los cartógrafos llevan mucho tiempo estudiando los criterios de un buen (o mal) etiquetamiento, y si bien no han llegado a una conclusión final, hay un concenso mayoritario en que hay ciertas reglas que deben respetarse si se busca un resultado estéticamente bueno [18]:

- Las etiquetas deben ser claramente legibles.
- Las etiquetas deben ser fácilmente ubicables.
- Se debe reconocer inmediatamente la correspondencia entre objetos y etiquetas.
- Las etiquetas deben estar muy próximas a los objetos a los que etiquetan.
- Las etiquetas no deben sobreponerse con otras etiquetas u objetos.
- Las etiquetas deben estar ubicadas en el mejor lugar posible, dentro de todas las ubicaciones legibles.

En [21], estas reglas se resumen de la siguiente manera:

- No se permiten sobreposiciones entre una etiqueta y otra etiqueta u otro elemento gráfico.
- Debe identificarse fácilmente a qué característica gráfica pertenece cada etiqueta.
- Cada etiqueta debe ubicarse en la mejor posición posible (dentro de las posiciones aceptables).

2.2.2. PFLP: Formalización

El problema *PFLP* suele ser abordado como un problema de optimización combinatoria. Para ello, se deben definir dos aspectos: el *espacio de búsqueda* y la *función objetivo*.

- **Espacio de búsqueda:**
  se lo puede definir como el conjunto de funciones que tienen como dominio el conjunto de posiciones de puntos en el plano, y como codominio el conjunto de posiciones de etiquetas. A cada elemento del espacio de búsqueda (una asignación de puntos a posiciones de etiquetas) se lo conoce como un *etiquetamiento*.
  De acuerdo a la difinición presentada, el conjunto de posibles posiciones de etiquetas para cada punto caracteriza el espacio de búsqueda.
  Para la mayoría de los algoritmos publicados, estas posibles posiciones se eligen siguiendo estándares cartográficos, y conforman un conjunto finito y enumerado. La figura 2.8 muestra un ejemplo de esto, en el cual cada recuadro representa una región en la que se puede ubicar la etiqueta. De forma alternativa, se puede adoptar un conjunto de posibles ubicaciones infinito, como por ejemplo permitir que las etiquetas se ubiquen libremente alrededor del objeto, siempre a una determinada distancia del mismo.

  Existen incluso variantes del problema en el que se permite omitir ubicar algunas etiquetas (se presume que éstas son las más difíciles de ubicar, o las de menor importancia). Cuando esto sucede, se dice que el problema incluye *selección de puntos* (del inglés *point selection*).

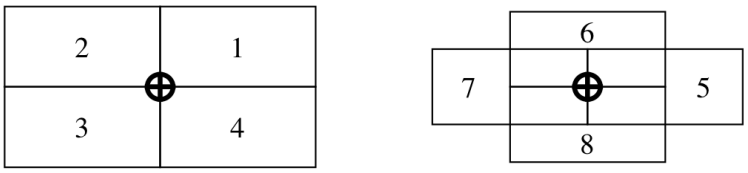


Figura 2.8: Posibles ubicaciones para una etiqueta. Un valor menor significa una mayor preferencia por dicha ubicación.

- **Función objetivo:**
  Es la función a optimizar, y que debe asignarle a cada elemento del espacio de búsqueda (etiquetamiento) un valor que corresponde a la *calidad relativa* del mismo.
  La noción de *calidad relativa* ha sido estudiando ampliamente por cartógrafos, siendo el más importante el trabajo de Imhof [18], quien logró hacer un análisis descriptivo en lugar de prescriptivo de la misma. Aún así, es sumamente difícil encontrar una definición apropiada para la función objetivo en el caso más general, ya que la calidad percibida por los humanos depende de numerosos factores, algunos de ellos de caracter subjetivo.
  Un enfoque bastante popular para el caso de PFLP, es hacer depender a la función objetivo de los siguientes factores [41]:

  - La cantidad de sobreposición entre etiquetas y otros objetos (incluyendo otras etiquetas).
  - La prioridad de la posición de las etiquetas dentro de las posibles posiciones.
  - Cantidad de puntos dejados sin etiquetar (sólo cuando el problema incluye selección de puntos).

2.2.3. PFLP: Complejidad

PFLP es un problema de optimización combinatoria definido por su espacio de búsqueda y función objetivo. Nos interesa identificar un algoritmo que sea capaz de encontrar un elemento bueno (preferentemente el mejor posible) del espacio de búsqueda. Es lógico entonces preguntarnos sobre la complejidad intrínseca del problema.

Para este estudio, elegiremos una version simple, ya que una vez demostrado que es **NP-Hard**, es sencillo luego demostrar que versiones más complejas también lo son. Para definir una instancia, debemos elegir una instancia particular dentro de los posibles espacios de búsqueda y funciones objetivos:

- **Espacio de búsqueda:** Asumiremos un posicionamiento discreto, con 4 posiciones igualmente favorables que corresponden a las numeradas 1 a 4 en la figura 2.8. Además, no permitiremos selección de puntos.
- **Función objetivo:** será la cantidad de puntos etiquetados que tengan sus etiquetas con uno o más solapamientos. Se buscará minimizar esta función.

Esta versión simplificada de *PFLP* es un problema de optimización. Para poder estudiarlo desde el punto de vista de su complejidad, debemos formular un problema de decisión: para cada instancia hacemos la pregunta *"¿existe algún etiquetamiento cuya función objetivo tenga valor cero (ningún solapamiento)?"*.

Ha sido demostrado que este problema de decisión es **NP-completo** independientemente en numerosas ocasiones (Kato y Imai, 1988; Marks y Shieber, 1991; Formann yWagner, 1991) [21]. Luego, un algoritmo de etiquetamiento que encuentre el mejor posible sirve para resolver el problema de decisión, por lo cual se puede afirmar que el problema de decisión se *reduce* al problema de optimización formulado anteriormente, y que este último es **NP-Hard**.

A pesar del resultado anterior, es posible introducir restricciones simples que reducen notablemente la complejidad del problema. Por ejemplo, si modificamos el espacio de búsqueda de modo de que cada punto a etiquetar tenga sólo 2 posibles posiciones de etiqueta, obtenemos una variante que se puede resolver en tiempo polinomial [21]. Sin embargo, aunque existan subcasos que tengan soluciones polinomiales, el resultado anterior implica que cualquier instancia de *PFLP* que provenga de un problema de interés será muy probablemente *NP-Hard*.

Hemos visto que el problema y sus variantes interesantes son *NP-Hard*, por lo cual suponiendo que $P \neq NP$, cualquier algoritmo de búsqueda exahustivo será impráctico, y cualquier algoritmo práctico será incompleto. Un análisis más detallado de la complejidad del problema se puede encontrar en [29].

2.2.4. PFLP: Algoritmos

A continuación haremos un breve resumen de las distintas técnicas que existen a la hora de enfrentarse a *PFLP*.

Búsqueda exhaustiva

Los algoritmos de búsqueda exhaustiva suelen ser categorizados de acuerdo a la manera en que recorren el espacio de búsqueda, y más específicamente, de acuerdo a cómo efectuan *backtracking*.
Existen numerosas variantes de algoritmos de búsqueda exhaustiva para *PFLP* que implementan distintas heurísticas, pero debido a la complejidad intrínseca del problema, todos se vuelven imprácticos incluso al enfrentarse a problemas muy moderados. Es por eso que en la práctica estos métodos no son utilizados más que como base de comparación con otros algoritmos.

Greedy

Una forma de evitar el "backtracking descontrolado" que ocurre en el caso de la búsqueda exhaustiva es usar un algoritmo *voraz* (del inglés *greedy*).
Los algoritmos *greedy* eliminan complentamente el *backtracking*, por lo que el etiquetamiento, en caso de encontrar una sobreposición, deberá elegir entre dejar el punto sin etiqueta (si el problema incluye selección de puntos), o colocar la etiqueta en superposición con otro objeto. De esta manera, se limita la extensión de la búsqueda, a costa de obtener un etiquetamiento potencialmente no ideal, pero con la ventaja de ser extremadamente rápido (generalmente las implementaciones tienen complejidad lineal con respecto a la cantidad de puntos).
Finalmente, para que un algoritmo *greedy* pueda encontrar una solución aceptable, es esencial que implemente heurísticas que guíen la búsqueda. Aún así, los resultados que se obtienen distan mucho de ser óptimos.

Búsqueda local: descenso discreto por gradiente

Los (pobres) resultados logrados por los algoritmos *greedy* en *PFLP* pueden ser notablemente mejorados si luego se los "repara" haciendo alteraciones locales. Esta es la motivación de los

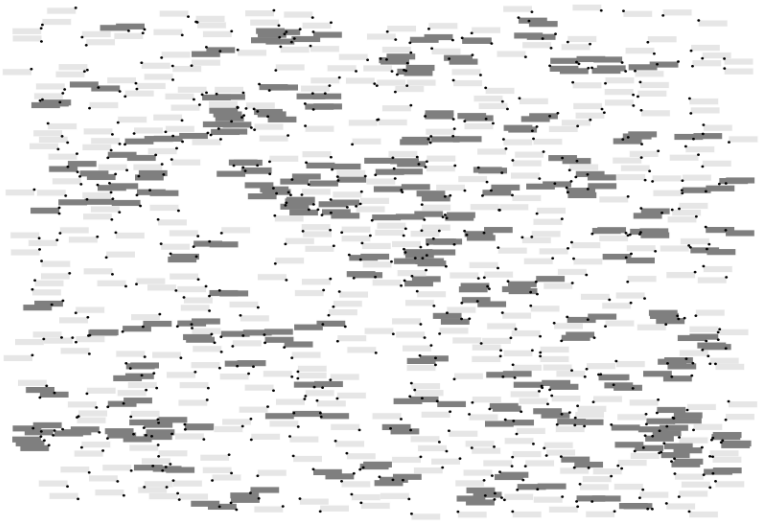


Figura 2.9: Resultado luego de aplicar un algoritmo greedy en 750 puntos: 341 solapamientos.

algoritmos que utilizan la técnica de *descenso discreto por gradiente* (*discrete gradient descent*), un tipo de búsqueda local.
Los algoritmos de esta familia se diferencian y definen de acuerdo a las operaciones que especifican cómo se realizan los reposicionamientos (simultáneos) entre grupos de etiquetas (normalmente cercanas unas de otras) [21]. Luego, el algoritmo debe elegir en cada paso cuál de todas esas operaciones presentan una mayor mejora en la situación, y ponerla en práctica. Al repetir este procedimiento una y otra vez, se efectúa un descenso discreto, el cual en cada paso se mueve en la dirección del gradiente de la *función objetivo*. El algortimo puede resumirse de la siguiente manera:

1. Para cada etiqueta, asignarle al azar una de sus posibles ubicaciones (también se puede utilizar la salida de otro algoritmo, como uno *greedy* para esta inicialización).

2. Repetir hasta que no haya ninguna mejora en la función objetivo:

   a) Considerar todas las operaciones posibles para reacomodar las etiquetas.

   b) Para cada una de las operaciones posibles, calcular el cambio que implicaría en el valor de la función objetivo.

   c) Implementar la operación que signifique la mayor mejora (en caso de empate, elegir al azar).

El mayor problema de estos métodos es que son susceptibles a quedar atrapados en un mínimo local, obteniendo así un resultado no óptimo.

Búsqueda local: Hirsch

El algoritmo de *Hirsch* [17], también conocido como método de *vectores sobrepuestos* (del inglés *overlap vectors*) es otro algoritmo de búsqueda local, que funciona sobre un modelo de posicionamiento infinito en el cual las aristas pueden moverse libremente alrededor del punto, siempre manteniendose tangentes a una circunferencia de determinado radio alrededor del mismo. En la figura 2.11 se puede visualizar algunas de las posibles posiciones alrededor de un punto.

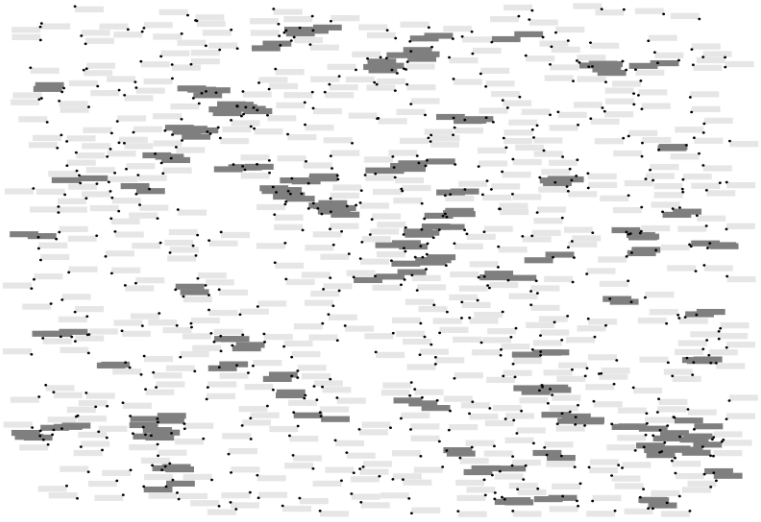


Figura 2.10: Resultado luego de aplicar descenos por gradiente en 750 puntos: 222 solapamientos.

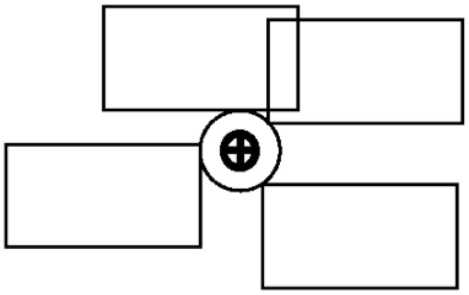


Figura 2.11: Posibles ubicaciones para una etiqueta en modelo de posicionamiento usado por el algoritmo de Hirsch.

El funcionamiento del algoritmo es muy simple:

- Inicialmente se ubican todas las etiquetas a la derecha de de los puntos a etiquetar.
- Para cada etiqueta, se calculan vectores de repulsión con todos los objetos con los que existe un conflicto.
- Se calcula la suma de los vectores de cada etiqueta, para obtener una fuerza resultante. En la figura 2.12 se puede ver un ejemplo de las fuerzas computados.
- Las etiquetas se mueven de acuerdo a la fuerza resultante obtenida (se alternan dos tipos de movimientos: absolutos e incrementales).
- Se repite el proceso de calcular los conflictos, las fuerzas resultantes y el movimiento de las etiquetas hasta que se hayan eliminado los conflictos o se llegue a otro criterio de parada (ej. máximo número de iteraciones).

Este algoritmo, al igual que todos las variantes de **búsquedas locales**, tiene el defecto de poder quedar atrapado en un mínimo local.

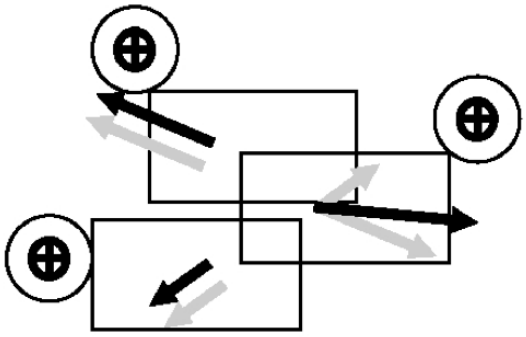


Figura 2.12: Vectores de repulsión creados por el algoritmo de Hirsch.

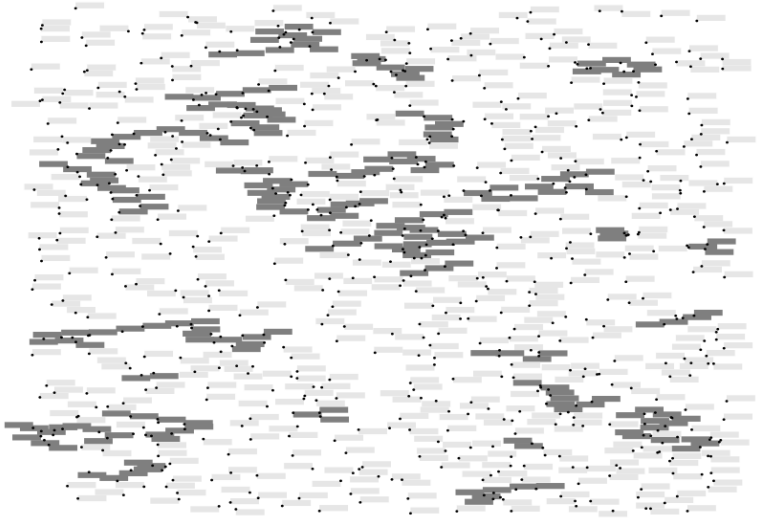


Figura 2.13: Resultado luego de aplicar el algoritmo de Hirsch en 750 puntos: 222 solapamientos.

Búsqueda estocástica

Como hemos visto, los métodos de *búsqueda local* pueden quedar atrapados en un mínimo local dentro del espacio de búsqueda. La naturaleza intrínseca de *PFLP* lo hace inevitable para cualquier algoritmo práctico [21]. Los problemas de estos métodos se agrupan en dos categorías:

- existen patrones que sistemáticamente "atrapan" a los distintos algoritmos en mínimos locales. Al aumentar el tamaño del problema, la probabilidad de encontrar alguno de estos patrones también aumenta y la calidad de la solución obtenida se ve inevitablemente resentida.
- las operaciones que los algoritmos emplean para transicionar de un estado a otro no brindan la posibilidad de escapar de un mínimo local una vez que se lo encontró.

Podemos llamar a estos dos problemas *sistematismo* y *monotonía* respectivamente. Para intentar resolver estos problemas, se emplean *métodos estocásticos*, como *recocido simulado* y *algorítmos genéticos*. Incorporar un elemento probabilístico (o estocástico) en la búsqueda trae aparejado consecuencias muy auspiciosas. Dado que la trayectoria estocástica es impredecible, se elimina el*sistematismo*. Además, se suele permitir un comportamiento (límitado) no-monótono, de forma de poder escapar de un mínimo local en caso de haberlo encontrado.

Búsqueda estocástica: recocido simulado

El *recocido simulado* (*simulated annealing*) [26] es esencialmente un método de *descenso por gradiente estocástico*, que permite movimiento en direcciones distintas que el gradiente. De hecho, incluso puede transicionar hacia un estado menos deseable en lugar de siempre mejorar (es decir, no es monótono el trayecto). La habilidad del algoritmo de degradar la solución es controlado por un parámetro T, llamado *temperatura*, que decrece a través de la ejecución del algoritmo. Cuando la temperatura es cero, las transiciones hacia un estado "peor" son completamente deshabilitadas, por lo que el algoritmo se reduce a un método de *descenso* (aunque no necesariamente desciende por el gradiente). A temperaturas mayores a cero, se le permite al algoritmo explorar una varieded más amplia del espacio de búsqueda (incluyendo transiciones a "peores" lugares), para que de esta manera se alcancen regiones distintas a las que rodean al mínimo local.
El algoritmo de *recido simulado para PFLP* puede resumirse de la siguiente forma:

1. Inicializar las posiciones de las etiquetas, ya sea en forma aleatoria o usando el resultado de otro método.

2. Repetir hasta que la tasa de mejora caiga por debajo de un determinado umbral:

   a) Disminuir la temperatura T de acuerdo con el esquema de enfriamiento utilizado.

   b) Elegir una etiqueta y moverla a una nueva posición (usando algún método para esta elección).

   c) Computar el cambio (ΔE) en el valor de la función objetivo que se produce.

   d) Si el cambio es negativo (o sea, a un estado menos deseable), deshacer el último posicionamiento con probabilidad $P = 1 - e^{-\Delta E/T}$.

Las implementaciones de este esquema se distinguen por los siguiente componentes:

- **Configuración inicial**. Se debe elegir un método de inicialización, ya sea asignando posiciones al azar a las etiquetas, o usando la salida de otro método.

- **Función objetivo**. Su elección afecta a la estética del resultado y a la eficiancia de la búsqueda. Dado que es método es por naturaleza estadístico que requiere de un gran número de evaluaciones para su éxito, las mejores funciones objetivo son aquellas cuya diferencia (ΔE) puede ser computado rápidamente.

- **Método de cambio**. Hay que elegir qué etiqueta reposicionar, y adónde hacerlo. Un método común consiste en elegir una etiqueta al azar, ya sea entre todas o entre las que presentan algún conflicto.

- **Esquema de enfriamiento**. Afecta directamente el resultado del algoritmo, permitiendo más o menos transiciones "negativas" (hacia un etiquetamiento menos deseable).

Búsqueda estocástica: Algoritmos genéticos

Un *algoritmo genético* es una técnica de optimización adaptativa, basada en los principios de la genética natural y de la supervivencia del más apto. Estos algoritmos usan las leyes de selección natural y genética para guiar una búsqueda no determinística.
Un algoritmo genético opera en forma iterativa sobre una *población* de tamaño fijo, o un conjunto de *soluciones candidatas*. Las *soluciones candidatas* representan una codificación del problema que es análoga a los cromosomas en los sistemas biológicos. Cada cromosoma representa una

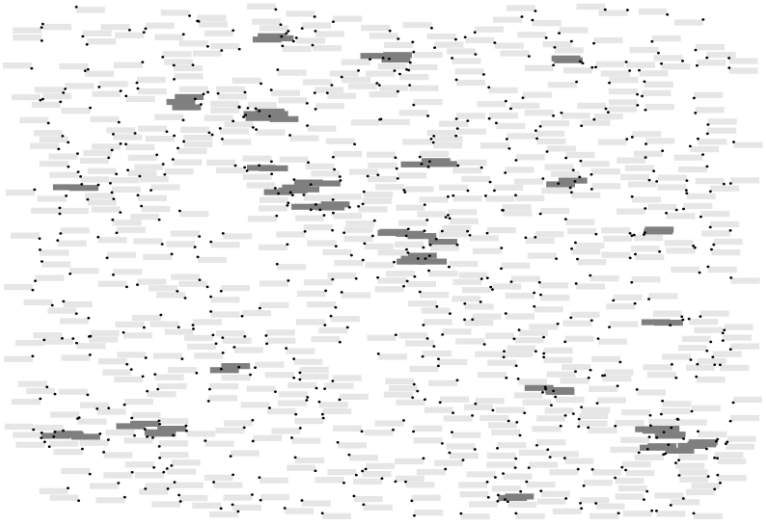


Figura 2.14: Resultado luego de aplicar el algoritmo de simulated annealing en 750 puntos: 75 solapamientos.

posible solución (en nuestro caso un elemento del espacio de búsqueda). De igual manera, cada cromosoma tiene asociado un valor de *aptitud* (fitness), que corresponde al valor de la función objetivo a optimizar. Es esta *aptitud* la que determina la habilidad del cromosoma de sobrevivir y tener descendientes. Cada cromosoma se compone de una cadena de *genes*, cuyos valores se denominan *alelos*. Las distintas variantes de algoritmos genéticos comparten una estructura similar, que se detalla a continuación.

1. **Inicialización**. Se crea una población inicial, generalmente en forma aleatoria. Habitualmente se la hace lo suficientemente numerosa y variada para tener un mejor cubrimiento del espacio de búsqueda.

2. **Selección**. En cada generación, se elige una parte de la población para reproducirse y producir la siguiente generación. El criterio de selección depende de la naturaleza del problema y de la implementación del algoritmo.

3. **Mutación**. La siguiente generación se forma a partir de la generación anterior. Para cada integrante de esta nueva generación, se elige a los "padres" (miembros de la generación anterior) y se combinan sus características (codificadas como alelos en cromosomas) para formar al nuevo especimen. La combinación propiamente dicha se efectúa utilizando *operadores genéticos*, que varían según la implementación.

4. **Terminación**. Luego de crear una población inicial, y de haber creado nuevas generaciones (a través de repetir iterativamente la *selección* y la *mutación*), es necesario poder terminar el proceso y brindar una solución (última generación). Existen distintos criterios de parada, como ser haber efectuado cierta cantidad de iteraciones, haber encontrado una solución suficientemente buena, etc.

Verner, Wainwright y Schoenefeld [38] fueron quienes diseñaron por primera vez un algoritmo genético para PFLP en 1997. En esta versión, los autores utilizan la técnica de *enmascaramiento* (masking) para mejorar la preservación de características favorables en la etapa de mutación.

Yamamoto y Lorena [40] publicaron en 2005 una variante denominada *Constructive Genetic Approach to PFLP* (CGA) que logra mejoras sustanciales sobre otros algoritmos genéticos para PFLP. Se basa en trabajo previo de Lorena y Furtado, y se diferencia de otros algoritmos genéticos en que evalúa los esquemas (schemata) directamente, posee una población de tamaño variable compuesta de esquemas (individuos de la población que no están en el espacio de búsqueda) y estructuras (elementos del espacio de búsqueda), y la posibilidad de usar heurísticas en la definiciones de las funciones de *fitness* (etapa de selección).
Esta versión del algoritmo se considera hoy en día una de las mejores soluciones generales para el problema de PFLP.

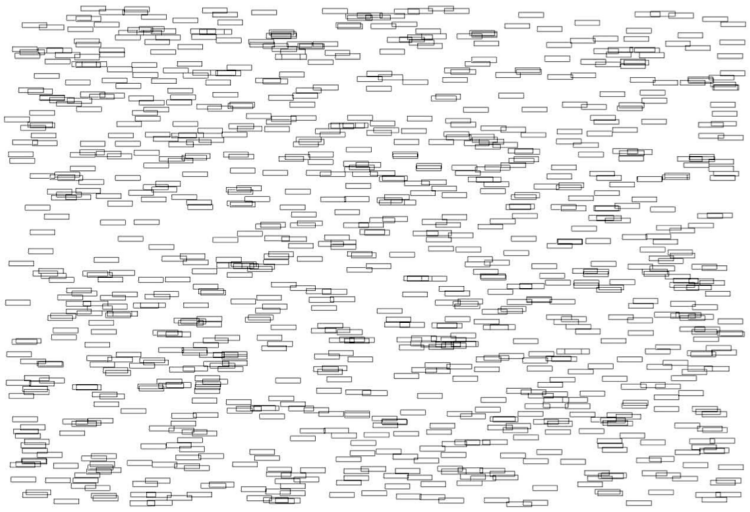


Figura 2.15: Resultado luego de aplicar CGA en 1000 puntos generados al azar: 88 solapamientos.

TABU Search

Yamamoto, Camara y Lorena [30] presentaron en 2000 un algoritmo para PFLP basado en la *búsqueda TABU* (TS - tabu search). TS es un procedimiendo heurístico propuesto por Glover para resolver problemas de optimización combinatoria.
La idea fundamental es evitar que la búsqueda se detenga cuando un óptimo local sea encontrado. Para ello, TS mantiene una lista de puntos en donde el etiquetamiento se ha cambiado (*tabu list*) y que se consideran no aceptables. El algoritmo recorre el espacio de búsqueda intentando hallar la mejor solución que no es considerada *Tabu*. La heurística *TS* también incluye una previsión especial para que las soluciones en la *tabu list* puedan ser consideradas como alternativas válidas, basandose en criterios de aspiración que determinan cuando las restricciones del *tabu* pueden ser dejadas de lado.
A continuación se describe brevemente el algoritmo TS para PFLP.

1. Precomputar todas las sobreposiciones potenciales entre posiciones de etiquetas.
2. Generar un etiquetamiento inicial, asignando a cada etiqueta su posición más deseada.
3. Repetir los siguientes pasos hasta que se obtenga una solución sin solapamientos o se haya alcanzado el máximo de iteraciones:
   a) Crear una *lista inicial de candidatos* para esta iteración.

b) Recalcular la *lista de candidatos* para hallar el etiquetamiento de menor costo para todos los puntos referenciados en dicha lista.

c) Elegir el mejor candidato de la lista, teniendo en cuenta la *lista tabú* y el *criterio de aspiración*.

d) Efectuar el cambio de la configuración (etiquetado), designando a la nueva disposición como la solución actual. Cada cambio de configuración consiste en modificar la posición de una sola etiqueta.

e) Actualizar la *lista tabú*.

El algoritmo descripto contiene 6 componentes que resta definir:

- **Función objetivo**. Dado que el algoritmo *TS* descripto es completamente determinista, es necesario examinar y comparar numerosos candidatos, haciendo que la función objetivo deba ser empleada en numerosas oportunidades. Es por ello, que una buena función objetivo debe ser fácilmente computada, haciendo la búsqueda más eficiente, al tiempo que permita obtener una solución con la calidad deseada.

- **Lista tabú**. Es un componente esencial de este algoritmo, y sirve para guardar los puntos en los que se han efectuado los últimos cambios de posiciones de etiquetas. Los autores proponen una lista con tamaño dinámico, que originalmente es considerablemente grande (para evitar que el algoritmo se concentre en solucionar conflictos sólo en algún área determinada del diagrama), y que luego va decreciendo a medida que se van solucionando las sobreposiciones.

- **Lista de candidatos**. Se compone de tripletas (punto, posición etiqueta, costo) que tienen mayor costo (menos deseable) en la configuración actual. El tamaño de la lista es recalculado cada cierto número de iteraciones (los autores usan 50) de modo de que el tamaño vaya disminuyendo a medida que se solucionan los conflictos.

- **Cambios de configuración**. Para generarlo, todas las tripletas en la *lista de candidatos* son usadas para buscar por la mejor posición para mover su etiqueta. Luego de los cambios, el candidato que provee el menor costo individual es elegido. Si el punto en cuestión pertenece a la *lista tabú*, es descartado y se elige la siguiente mejor alternativa.

- **Criterio de aspiración**. En algunos casos es necesario considerar alternativas para un *cambio de configuración* que son parte de la *lista tabú*. En esos casos, este criterio permite anular la restricción del tabú. Los autores presentan 2 casos en los que esto sucede: el candidato mejora la mejor solución global obtenida hasta el momento, o si todos los miembros de la *lista de candidatos* están en la *lista tabú*.

- **Memoria a largo plazo**. Frecuentemente, el costo de distintas soluciones es el mismo, resultando en que los mismos puntos son agregados a la *lista de candidatos*. Para diversificar la búsqueda, los autores usan una estrategia de memoria basada en frecuencia, que cuenta la cantidad de veces que un punto cambió de lugar su etiqueta, y cada 50 iteraciones divide todos esos valores por el máximo. Esta información es usada para aplicar "multas" a los puntos que no causaron mejora, modificando así los costos de dichos puntos y funcionando como un instrumento de diversificación de la búsqueda.

El algoritmo resultante produce muy buenos resultados, siendo considerado dentro de los mejores que existen en el estado del arte para PFLP.

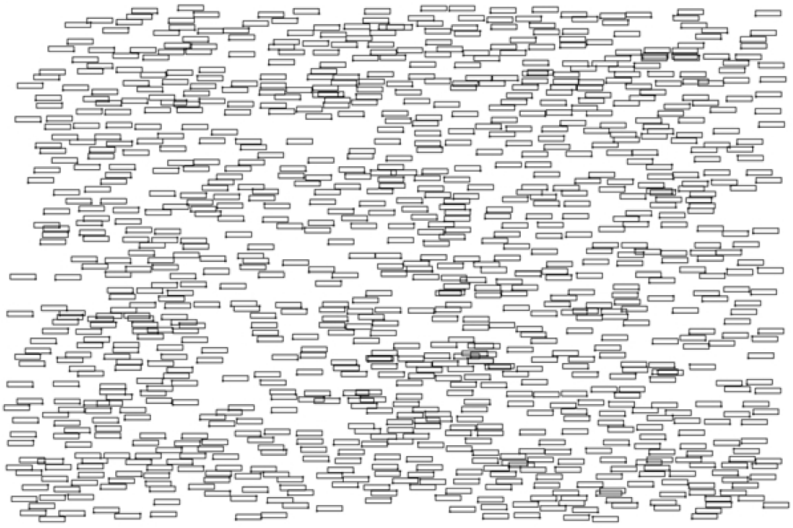


Figura 2.16: Resultado luego de aplicar Tabu en 1000 puntos generados al azar: 77 solapamientos.

Capítulo 3

Un nuevo método: Etiquetas ubicadas "a la fuerza"

3.1. Objetivo

El objetivo de nuestro trabajo fue diseñar e implementar un algoritmo de layout de etiquetas de grafos, que tuviese la flexibilidad necesaria para afrontar las tareas que, a nuestro entender, son las requeridas por los potenciales usuarios.

Para esta primera etapa, nos concentramos sólo en las etiquetas de nodos, y nos fijamos como metas que el algoritmo:

- pueda ser usado para ubicar etiquetas tanto mientras se efectúa el *layout* del grafo, como también en forma posterior (con una geometría rígida de nodos y aristas).
- pueda ser utilizado en la totalidad de un grafo, o en un subgrafo (en este caso, sin alterar el resto del grafo).
- sea eficiente, de forma de que pueda trabajar con grafos de tamaño mediano (100 a 1000 nodos) en tiempo interactivo (tiempo de ejecución no mayor a 10 segundos).
- produzca resultados cualitativamente mejores que métodos triviales, como por ejemplo ubicar las etiquedas centradas sobre los nodos, o con un desplazamiento fijo (X cm arriba, Y cm a la derecha del nodo).

3.2. Descripción de la solución ideada

La inspiración para el algoritmo ideado vino desde la familia de algoritmos *force directed* para *graph layout*, y en particular, el algoritmo de *Fruchtermann-Reingold* [14].

La idea central del algoritmo es muy simple: tratar a las etiquetas de nodos como si fuesen nodos "ficticios", y vincular a ambos nodos (el original y el creado para reemplazar a la etiqueta) con una arista. Luego, proceder a hacer un layout con el algoritmo *Fruchtermann-Reingold*, y utilizar las posiciones de los nodos "ficticios" para colocar las etiquetas (eliminando los nodos y aristas agregadas en el proceso).

3.2.1. Ejemplo del funcionamiento deseado por la solución

El proceso deseado se ejemplifica a continuación con un grafo con 3 nodos. Una práctica habitual es colocar las etiquetas centradas sobre sus *nodos padres*. La figura 3.1 muestra esta situación.

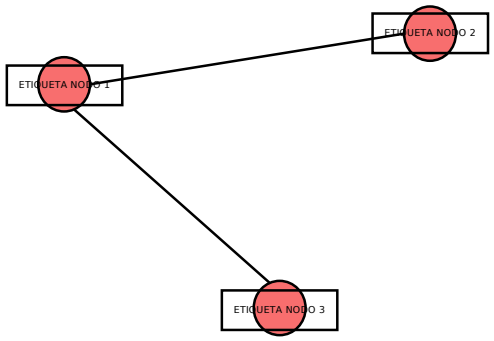


Figura 3.1: Grafo con etiquetas de nodos en posición centrada.

Nuestro método, crearía nodos "ficticios" (a los que pasaremos a llamar *nodos dummies*) que reemplazarían a las etiquetas, y aristas (*aristas dummies)* que los unen con sus *nodos padres*. La posición de estos *nodos dummies* al momento de su creación será aleatoria dentro del área circundante a su *nodo padre*. La figura 3.2 ejemplifica el grafo auxiliar resultante.

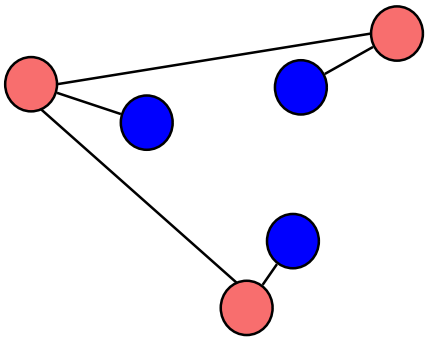


Figura 3.2: Grafo auxiliar con nodos (en azul) y aristas dummies creados.

Una vez creado el grafo auxiliar, se efectuará un layout similar al propuesto por Fruchtermann y Reingold [14]. Se podrá optar por permitir el movimiento de todos los nodos o sólo de los nodos dummies (logrando de esa forma mantener la geometría de los nodos originales). La

figura 3.3 muestra el resultado de hacer dicho layout con el grafo auxiliar, sin permitir que los nodos originales se desplacen.

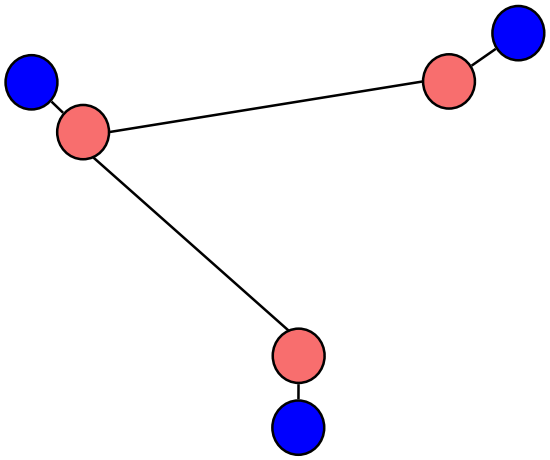


Figura 3.3: Grafo auxiliar luego del layout, donde sólo se permite el desplazamiento de nodos dummies.

Finalmente, sólo restaría trasladar el resultado al grafo original. Para ello, se reubicarían las etiquetas de los nodos en la posición que ocupa su *nodo dummie* correspondiente en el grafo auxiliar. La figura 3.4 muestra el resultado deseado.

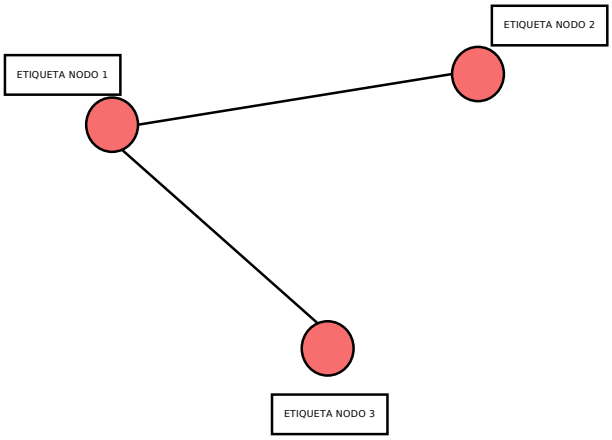


Figura 3.4: Grafo original, en donde las etiquetas se reubicaron en las posiciones de los nodos dummies en el grafo auxiliar luego del layout.

3.2.2. Algoritmo

Dado que nuestra idea se basa en reutilizar en gran parte el algoritmo de *layout* propuesto por Fruchtermann y Reingold en [14], presentamos la versión original como algoritmo 1.

Algoritmo 1: Algoritmo de Graph Layout propuesto por Fruchtermann y Reingold

Data: G = (V,E), W, L
Result: G

$area \leftarrow W \times L$ { W y L son el ancho y largo del marco (frame)}
$randomize(V)$ { A los vértices le son asignadas posiciones iniciales al azar }
$k \leftarrow \sqrt{(area/|V|)}$
function $f_a(x)$:= return x^2/k **end**
function $f_r(x)$:= return k^2/x **end**

for $i \leftarrow 1$ **to** *iterations* **do**
  { Calcular fuerzas repulsivas }
  for $v \in V$ **do**
    $v.desp \leftarrow 0$
    for $u \in V$ **do**
      if $u \neq v$ **then**
        {Δ es el vector diferencia entre los dos puntos }
        $\Delta \leftarrow v.pos - u.pos$
        $v.desp \leftarrow v.desp + (\Delta/\|\Delta\|) * f_r(\|\Delta\|)$
      end
    end
  end
  { Calcular fuerzas atractivas }
  for $e \in E$ **do**
    { Cada arista es un par ordenado de vértices: .v y .u }
    $\Delta \leftarrow e.v.pos - e.u.pos$
    $e.v.desp \leftarrow e.v.desp - (\Delta/\|\Delta\|) * f_a(\|\Delta\|)$
    $e.u.desp \leftarrow e.u.desp + (\Delta/\|\Delta\|) * f_a(\|\Delta\|)$
  end
  { Limitar el desplazamiento de acuerdo a la temperatura t }
  { Luego evitar desplazamientos fuera del marco }
  for $v \in V$ **do**
    $v.pos \leftarrow v.pos + (v.desp/\|v.desp\|) * min(v.desp, t)$
    $v.pos.x \leftarrow min * (W/2, max(-W/2, v.pos.x))$
    $v.pos.y \leftarrow min * (L/2, max(-L/2, v.pos.y))$
  end
  { Reducir la temperatura t }
  $t \leftarrow cool(t)$
end

Una primera versión de nuestro algoritmo se presenta en el algoritmo 2.

Algoritmo 2: Primer versión de nuestro algoritmo de *etiquetas a la fuerza*. Los cambios introducidos en color rojo.

Data: G = (V,E), W, L
Result: G

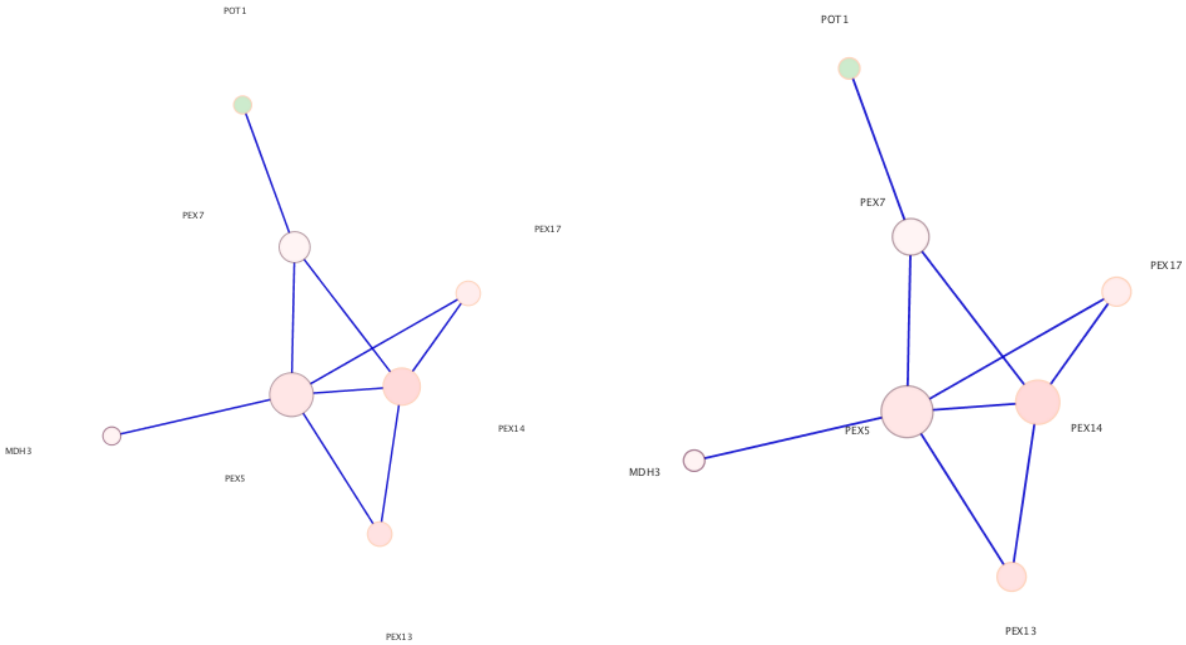
{ Se crean nodos y aristas dummies }
$V', E' = createDummyNodes(V, E)$
{ El grafo auxiliar construido }
$G' = (V', E')$
$area \leftarrow W \times L$ { W y L son el ancho y largo del marco (frame) }
$randomize(V')$ { A los vértices le son asignadas posiciones iniciales al azar }
$k \leftarrow \sqrt{(area/|V|)}$
function $f_a(x)$:= return x^2/k
function $f_r(x)$:= return k^2/x

for $i \leftarrow 1$ **to** *iterations* **do**
    { Calcular fuerzas repulsivas }
    for $v \in V'$ **do**
        $v.desp \leftarrow 0$
        for $u \in V'$ **do**
            if $u \neq v$ **then**
                { Δ es el vector diferencia entre los dos puntos }
                $\Delta \leftarrow v.pos - u.pos$
                $v.desp \leftarrow v.desp + (\Delta/\|\Delta\|) * f_r(\|\Delta\|)$
            end
        end
    end
    { Calcular fuerzas atractivas }
    for $e \in E'$ **do**
        { Cada arista es un par ordenado de vértices: .v y .u }
        $\Delta \leftarrow e.v.pos - e.u.pos$
        $e.v.desp \leftarrow e.v.desp - (\Delta/\|\Delta\|) * f_a(\|\Delta\|)$
        $e.u.desp \leftarrow e.u.desp + (\Delta/\|\Delta\|) * f_a(\|\Delta\|)$
    end
    { Limitar el desplazamiento de acuerdo a la temperatura t }
    { Luego evitar desplazamientos fuera del marco }
    for $v \in V'$ **do**
        $v.pos \leftarrow v.pos + (v.desp/\|v.desp\|) * min(v.desp, t)$
        $v.pos.x \leftarrow min * (W/2, max(-W/2, v.pos.x))$
        $v.pos.y \leftarrow min * (L/2, max(-L/2, v.pos.y))$
    end
    { Reducir la temperatura t }
    $t \leftarrow cool(t)$
end

{ Una vez terminado el layout de G', trasladar las posiciones de nodos y etiquetas (nodos dummies) a G}
traslatePositions(G', G)

Esta primera versión de nuestro algoritmo presenta una falla fundamental: no distingue en absoluto entre los nodos originales y los *dummies*, ni entre las aristas originales y *dummies*. Esto provocará que las etiquetas terminen estando, en promedio, a la misma distancia de sus nodos padres que cualquier otro par de nodos unidos por una arista. Esto claramente afectará a la legibilidad, ya que se desea que las etiquetas queden considerablemente más cercanas a su nodo padre que a cualquier otro elemento del dibujo.

En la figura 3.5 se ejemplifica la falla del algoritmo.



(a) Resultado obtenido, etiquetas muy alejadas de los nodos padres (b) Distancia entre etiquetas y nodos padres deseada

Figura 3.5: Comparación entre salida del algoritmo 2 y resultado deseado

Para evitar el problema antes mencionado, durante la simulación física se pueden tratar de forma diferente las fuerzas atractivas producidas por las aristas *dummies* (las que unen los nodos *dummies* con sus padres). Haciendo que estas fuerzas atractivas sean considerablemente más fuertes, se logra que la distancia (promedio) entre los nodos *dummies* y sus padres se reduzca notablemente, lo cual es lo mismo que decir que las etiquetas quedarán más cerca de sus nodos. Una nueva versión de nuestro algoritmo se puede ver en el algoritmo 3.

Algoritmo 3: Segunda versión de nuestro algoritmo de *etiquetas a la fuerza*. Los nuevos cambios introducidos en color azul.

Data: G = (V,E), W, L, C
Result: G

{ Se crean nodos y aristas dummies }
$V', E' = createDummyNodes(V, E)$
$G' = (V', E')$ { El grafo auxiliar construido }
$area \leftarrow W \times L$ { W y L son el ancho y largo del marco (frame) }
$randomize(V')$ { A los vértices le son asignadas posiciones iniciales al azar }
$k \leftarrow \sqrt{(area/|V|)}$
function $f_a(x)$:= return x^2/k
{ Función que calcula el módulo de la fuerza entre un nodo dummy y su padre }
function $f_d(x)$:= return $C * x^2/k$ { $1 < C$ }
function $f_r(x)$:= return k^2/x

for $i \leftarrow 1$ **to** *iterations* **do**
    { Calcular fuerzas repulsivas }
    for $v \in V'$ **do**
        $v.desp \leftarrow 0$
        for $u \in V'$ **do**
            if $u \neq v$ **then**
                { Δ es el vector diferencia entre los dos puntos }
                $\Delta \leftarrow v.pos - u.pos$
                $v.desp \leftarrow v.desp + (\Delta/\|\Delta\|) * f_r(\|\Delta\|)$
            end
        end
    end
    { Calcular fuerzas atractivas entre nodos originales }
    for $e \in E$ **do**
        { Cada arista es un par ordenado de vértices: .v y .u }
        $\Delta \leftarrow e.v.pos - e.u.pos$
        $e.v.desp \leftarrow e.v.desp - (\Delta/\|\Delta\|) * f_a(\|\Delta\|)$
        $e.u.desp \leftarrow e.u.desp + (\Delta/\|\Delta\|) * f_a(\|\Delta\|)$
    end
    { Calcular fuerzas atractivas entre nodos dummies y sus nodos padres }
    for $e \in (E' - E)$ **do**
        $\Delta \leftarrow e.v.pos - e.u.pos$
        $e.v.desp \leftarrow e.v.desp - (\Delta/\|\Delta\|) * f_d(\|\Delta\|)$
        $e.u.desp \leftarrow e.u.desp + (\Delta/\|\Delta\|) * f_d(\|\Delta\|)$
    end
    { Limitar el desplazamiento de acuerdo a la temperatura t y evitar desplazamientos fuera del marco}
    $updatePositions(V', t, W, L)$
    { Reducir la temperatura t }
    $t \leftarrow cool(t)$
end
{ Una vez terminado el layout de G', trasladar las posiciones de nodos y etiquetas a G}
traslatePositions(G', G)

3.2.3. Agregando mejoras

Como es natural, desde 1991 cuando Fruchtermann y Reingold presentaron su algoritmo de layout, muchas mejoras y novedades han sido incorporadas al mismo. De hecho, en la práctica no se emplea la versión original, sino que se la suele encontrar adaptada y mejorada.

Algunas de las modificaciones que existen, y que queremos agregar a nuestro algoritmo son las siguientes:

- *Particionado.*
  Antes de efectuar el layout, se descompone el grafo en componentes conexas. Luego, se efectua el layout de cada una de ellas por separado, y se componen (usualmente en una grilla) para formar un único resultado.
  Este método no sólo logra evitar que las distintas componentes conexas se alejen unas de otras indefinidamente (como ocurre en la versión original del algoritmo), sino que logran una notable mejora del tiempo de ejecución en grafos no conexos. Para la versión original, la cantidad de pares de nodos sobre los cuales se calculan las fuerzas repulsivas es: N^2, donde N es la cantidad de nodos. Para la versión particionada resulta ser: $\sum_{p \in P} N_p^2$, donde P son las particiones y N_p es la cantidad de nodos de la partición p. Es fácil ver que para un grafo con dos particiones de $N/2$ nodos cada una, se efectúan la mitad de cómputos de fuerzas repulsivas en cada iteración, un décimo si tiene 10 componentes de tamaño $N/10$, etc. Además proporciona otra ventaja en términos de eficiencia que es más difícil de medir, y es que al ser parte de simulaciones distintas, no es necesario hacer que todas las componentes pasen por la misma cantidad de pasos de simulación. Así, una componente que se estabiliza antes que otra puede ser detenida en dicho momento, evitando pasos innecesarios.

- *Fuerza de gravedad.*
  Se agrega una fuerza de gravedad que atrae a todos los nodos hacia el centro del layout, ya sea el centro del marco o el centro de gravedad del grafo.
  Esta fuerza, de módulo constante, atrae a todos los nodos de forma uniforme al centro, logrando evitar una dispersión descontrolada del layout, y mejorando la calidad estética del resultado. Cuando se trata de un layout particionado, la fuerza de gravedad se usa dentro del layout de cada componente.

- *Aristas con pesos.*
  Las fuerzas de atracción entre un par de nodos unidos por una arista dependen del peso de dicha arista.
  Al trabajar con grafos con pesos, es muy común pretender que el layout refleje de alguna forma estos valores en la topología del resultado: por ejemplo, si se trata de un grafo que representa ciudades (nodos) y rutas (aristas), donde los pesos de las aristas es la distancia entre las ciudades, sería conveniente que las ciudades más cercanas (aristas con menor peso) terminen ubicadas cerca en el layout. De manera inversa, si estuviésemos trabajando con un grafo que representa una red social, en donde los nodos son personas y las aristas la cantidad de conexiones en común, desearíamos que los nodos unidos por aristas de mayor peso (más conexiones en común) queden ubicados relativamente cerca en el layout.
  Para poder lograr esta versatilidad se suele usar un conjunto de funciones que mapean el peso de una arista al módulo de la fuerza de atracción, de forma que variando la función usada se obtengan fuerzas proporcionales al peso de la arista, inversamente proporcionales, logarítmicas, logarítmicas inversas, etc.

- *Layout sobre un subgrafo.*
  Posibilita la posibilidad de dejar fija una parte del grafo, y aplicar el layout sobre el resto. Debido a la naturaleza de los algoritmos *force directed*, es posible hacer intervenir en el resultado también a la parte del grafo que no se desea mover: para eso simplemente se modifica la función que actualiza las posiciones de modo de que los elementos 'bloqueados' no se muevan, permitiendo que éstos intervengan en los cómputos de fuerzas y siguan atrayendo y repeliendo normalmente al resto del grafo. De este modo, un nodo 'bloqueado' no suele quedar tapado o por otro elemento, ya que durante el layout la fuerza de repulsión entre ellos sigue vigente.

Además de estas funcionalidades ya presentes en otros algoritmos, uno de nuestros objetivos era poder incluír o excluír a voluntad los nodos y etiquetas, de modo de que el algoritmo funcione sólo sobre las etiquetas, sólo sobre nodos, o bien tanto sobre etiquetas como nodos.

Tras agregar todas estas mejoras a nuestro método, obtenemos el algoritmo 4.

Algoritmo 4: Tercera versión de nuestro algoritmo de *etiquetas a la fuerza*.

Data: G = (V,E), C, M
Result: G
$G' \leftarrow createDummyNodes(V, E)$ { $G = (V', E')$ }
$P \leftarrow partitionGraph(G')$ { Particiona el grafo }

{ Hacer el layout de cada componente. $p = (V_p, E_p)$ }
for $p \in P$ **do**
  $randomize(V_p)$
  for $i \leftarrow 1$ **to** *iterations* **do**
    { Calcular fuerzas repulsivas }
    for $v \in V_p$ **do**
      $v.desp \leftarrow 0$
      for $u \in (V_p - v)$ **do**
        $\Delta \leftarrow v.pos - u.pos$
        $v.desp \leftarrow v.desp + (\Delta / \|\Delta\|) * f_r(\|\Delta\|)$
      end
    end
    { Calcular fuerzas atractivas, en base a las distancias y peso de la aristas $(e.w)$}
    for $e \in E_p$ **do**
      $\Delta \leftarrow e.v.pos - e.u.pos$
      if $isDummy(e.v)$ *or* $isDummy(e.u)$ **then**
        $F \leftarrow f_d(mapWeigth(e.w, \|\Delta\|))$
      else
        $F \leftarrow f_a(mapWeigth(e.w, \|\Delta\|))$
      end
      $e.v.desp \leftarrow e.v.desp - (\Delta / \|\Delta\|) * F$
      $e.u.desp \leftarrow e.u.desp + (\Delta / \|\Delta\|) * F$
    end
    { Calcular fuerzas gravitatorias }
    for $v \in V_p$ **do**
      $v.desp \leftarrow v.desp + gravityForce(v)$
    end
    { Limitar el desplazamiento de acuerdo a la temperatura t y evitar desplazamientos de elementos fijos }
    $updatePositions(V_p, t)$
    { Reducir la temperatura t }
    $t \leftarrow cool(t)$
  end
end

{ Unir el resultado de cada componente }
$G' \leftarrow mergeLayouts(P)$

{ Trasladar las posiciones de nodos y etiquetas a G}
$traslatePositions(G', G)$

Capítulo 4

Implementación

Para implementar el algoritmo presentado en el capítulo 3 se contemplaron distintas alternativas:

- Implementarlo desde cero, como una aplicación independiente.
- Implemetarlo desde cero, como un plugin o agregado a una aplicación ya existente.
- Modificar alguna implementación del algoritmo de Fruchtermann y Reingold que sea una aplicación independiente.
- Modificar alguna implementación del algoritmo de Fruchtermann y Reingold que sea parte de alguna aplicación ya existente.

Luego de analizar las ventajas y desventajas que ofrecía cada opción, se optó por la última de ellas, siendo *Cytoscape* la aplicación que ya contaba con una implementación que podíamos modificar y adaptar a nuestras necesidades.

4.1. Cytoscape

La descripción oficial de Cytoscape [32] [2] sostiene que es una plataforma de software para visualizar redes de interacciones moleculares y vías metabólicas biológicas, integrar esas redes con anotaciones, perfiles de expresión de genes y otros datos de estado. A pesar de que fue diseñado originalmente para la investigación biológica, actualmente es una plataforma para análisis y visualización de redes complejas de cualquier dominio.

El núcleo (core) de Cytoscape provee de un conjunto básico de funcionalidades que permiten integración de datos, análisis y visualización, en tanto que las funcionalidades restantes son provistas a través de *apps*. Existen *apps* que brindan características de análisis adicionales, layouts, scripting, conexiones con bases de datos y repositorios de información, etc.

Cytoscape se encuentra disponible en forma gratuita y libre, con el núcleo y muchas apps licenciadas bajo LGPL, lo que permite que cualquiera pueda crear y distribuir su propia app. Adicionalmente, existe un repositorio de apps en donde se las puede publicar y que permite a los usuarios de Cytoscape instalar nuevas apps de forma muy sencilla.

Cytoscape actualmente posee dos versiones: **2.8.3** es la versión estable y el fin de la línea 2.x; **3.0.2** es parte de la nueva generación 3.x, más modular y escalable, pero aún no completamente estable. Dado que al momento de la implementación la línea 3.x recién estaba comenzando a

tomar forma, sólo había una versión beta y no había garantías de estabilidad de la API para desarrollar apps, la implementación del algoritmo se hizo modificando una app de Cytoscape 2.8.x.

4.1.1. Arquitectura y licencias

Cytoscape se encuentra escrito en Java, y de igual manera ofrece una API en dicho lenguaje para crear apps (plugins). En particular, una de las clases exportadas y que pueden ser heredadas es *AbstractLayout*, que es la clase "padre" para todos los algoritmos de layout que se quieran implementar.

La distribución oficial de Cytoscape incluye además del núcleo, ciertas apps que son consideradas esenciales, y que ofrecen funcionalidades que también pueden ser expandidas por otras apps. Las apps esenciales se denominan *coreplugins*, y una de ellas es *AutomaticLayout*, que se basa en la clase *AbstractLayout*, y proporciona no sólo algoritmos de layout (entre ellos el de Fruchtermann y Reingold), sino también un conjunto de utilidades para quién quiera implementar otro layout.

El núcleo de Cytoscape se encuentra licenciado bajo LGPL, lo cual es una licencia *Open Source* y *libre*, que permite que cualquiera pueda modificar, usar y distribuir libremente el código fuente. Las apps, en cambio, tienen cada una su propia licencia, que puede ser tanto *libre* u *open source* como no. Una de las razones por la que elegimos esta plataforma es que tanto la app *AutomaticLayout*, como todos los *coreplugins* también tienen licencia LGLP, por lo cual pudimos trabajar con ellos y modificarlos libremente.

4.2. Estado previo

A continuación se hará una descripción de las clases y componentes más relevantes para nuestro trabajo, en el estado en que se encontraban antes de introducir los cambios que son la implementación de nuestro nuevo algoritmo de layout.

4.2.1. AbstractLayout

La clase *AbstractLayout* provee un punto de partida para crear cualquier algoritmo de layout para Cytoscape. Para implementar un algoritmo de layout basta con extender esta clase, e implementar los métodos abstractos declarado en ella, siendo el método *doLayout()* el que debe contener la lógica del mismo.

4.2.2. App: AutomaticLayout

Esta app forma parte de los *coreplugins*, que son distribuídos junto con el núcleo de Cytoscape en su versión oficial. Desde el punto de vista del usuario final, ofrece numeros algoritmos de layout listos para ser usados, algunos de ellos son:

- Force Directed (una versión propia de un algoritmo de layout force directed)
- Circular
- Jerárquico
- Circular con Atributos

- Circular ordenado por grado
- ISOML (Inverted Self-Organizing Map Layout)
- Agrupado por atributos (Group Attributes Layout)
- KamadaKawai (versión original)
- KamadaKawai (modificado para tener en cuenta los pesos de las aristas)
- Fruchtermann y Reingold (modificado para tener en cuenta los pesos)
- Simulated annealing
- Radial
- Árbol (Tree layout)

Además, implementa funcionalidades que extienden las provistas por la clase padre de todos los layouts (*AbstractLayout*), quedando estas disponibles para realizar nuevos algoritmos de layout. Las clases más relevantes para nuestro trabajo son las siguientes.

- **csplugins.layout.LayoutPlugin**
  Clase que extiende *CytoscapePlugin* (clase base para todas las *Apps*), y que es la clase principal de la App *AutomaticLayout*. Construye la *App*, y agrega los diversos algoritmos que la componen.

- **csplugins.layout.algorithms.graphPartition.AbstractGraphPartition**
  Clase abstracta que extiende *AbstractLayout*. Su mayor aporte consiste en permitir partir el grafo en componentes conexas, de modo que cualquier algoritmo de layout pueda trabajar con ellas de forma independiente. Luego, al haberse realizado el layout sobre cada una de ellas, las reacomoda y construye con ellas un layout global. Además agrega varias opciones en el panel de configuración del layout y funcionalidades menores.
  Al particionar un grafo, lo hace usando la clase *LayoutPartition*, por lo que las clases que la hereden e implementen un layout, deben trabajar con esa clase como representación del grafo.

- **csplugins.layout.LayoutPartition**
  Clase que contiene toda la información acerca de una partición (componente conexa) del grafo. Para almacenar la información relacionada a los nodos se usa la clase *LayoutNode*, mientras que la información relacionada las aristas se representa con objetos de la clase *LayoutEdge*.
  Además, provee métodos estáticos que se usan para particionar un grafo existente.

- **csplugins.layout.LayoutNode**
  Clase que facilita el manejo de la información sobre distintas propiedades de los nodos en sí mismos, como ser posición, incremento de posición, bloqueo, índice, etc.

- **csplugins.layout.LayoutEdge**
  Clase que abstrae las aristas, y proporciona una forma eficiente de capturar y manejar la información sobre cada una de ellas.

- **csplugins.layout.EdgeWeighter**
  Clase que es usada como un contenedor de información acerca de cómo deben ser interpretados los pesos en el layout.

- **csplugins.layout.algorithms.bioLayout.BioLayoutAlgorithm**
  Clase abstracta que extiende *AbstractGraphPartition*. Funciona como superclase para los 2 algoritmos "biolayout" existentes: Fruchtermann-Reingold (*FRAlgorithm*) y Kamada-Kawai. El pricipal aporte de esta clase es el manejo de varias opciones comunes a los algoritmos de layout de esta familia, las que se encuentran disponibles en el panel de configuración de los layouts.

- **csplugins.layout.algorithms.bioLayout.FRAlgorithm**
  Clase que extiende *BioLayoutAlgorithm* e implenta el algoritmo de layout Fruchtermann-Reingold.

4.3. Cambios

Nuestro objetivo era poder manejar las etiquetas como parte del layout, y toda la infraestructura con la que ya contábamos para facilitar los algoritmos de layout, tanto en el core de Cytoscape, como dentro de la propia App "AutomaticLayout", se enfocaba sólo en nodos y aristas.
Nuestro curso de trabajo fue primero hacer lo obvio y más sencillo, y luego mejorarlo, generalizarlo, y lograr una solución más elegante.

4.3.1. Primer enfoque: naïve

La forma más obvia y fácil de lograr nuestro objetivo era implementar todo dentro de la clase que contenía la lógica del layout, es decir la clase *csplugins.layout.algorithms.bioLayout.FRAlgorithm*.

La lógica del layout en sí mismo se encontraba encapsulada dentro del siguiente método, que es el que se ejecuta cuando se llama al layout.

```
/**
 * Perform a layout
 */
public void layoutPartition(LayoutPartition partition);
```

Fue una cuestión relativamente simple la de modificar el código para que en lugar de ejecutar el layout sobre la partición que recibe como argumento cree un nuevo modelo, agregando las etiquetas como nuevos nodos, ejecute el layout y luego traslade las posiciones resultantes a la partición original.

En lugar de modificar el método *layoutPartition*, la metodología empleada fue la de crear una nueva clase *LabelBioLayoutFRAlgorithm*, que extiende la ya existente *BioLayoutFRAlgorithm*, y que sobrescribe el método *layoutPartition* de forma tal que cree el andamiaje necesario, construya una nueva partición con nuevos nodos que representan a las etiquetas (además de los nodos ya existentes), llame al mismo método de la clase padre (o sea, la implementación ya existente del algoritmo de Fruchtermann-Reingold), y luego utilizando el resultado provisto por la clase padre (sobre la nueva partición), traslade las posiciones obtenidas a los nodos y etiquetas de la partición original. Se incluye en el apéndice A el código fuente de la clase creada, que implementa en casi su totalidad este intento (existen pequeños cambios en otras clases, pero no son imprescindibles para la comprensión).

Si bien este enfoque sirvió como prueba de concepto, claramente que no era una solución satisfactoria. Todo el andamiaje e infraestructura con la que contábamos para hacer los layouts se desperdiciaba, y se reimplementaba dentro de la clase nueva toda esa lógica, agregándole la capacidad de manejar etiquetas. Además de una cuestión de prolijidad, el código era poco mantenible, y poco eficiente.

Esta primera implementación nos dio evidencia de que nuestro objetivo se podía lograr, y es por eso que con las lecciones aprendidas durante esta implementación diseñamos e implementamos otra alternativa.

4.3.2. Segundo enfoque: generalizando y abstrayendo

Nuestro objetivo con esta nueva implementación fue sencillo: en lugar de replicar la infraestructura existente para layouts, modificarla para que incorpore la lógica necesaria para tratar con etiquetas.
De este modo, no sólo nuestra nueva implementación del algoritmo de Fruchtermann y Reingold será beneficiada, sino todos los algoritmos que se construyen sobre la infraestructura común antes descripta. A continuación se brinda un detalle de los cambios más significativos.

Se modificaron las siguientes clases:

- *csplugins.layout.LayoutNode*
  Se modificó esta clase para que sólo contenga los métodos y campos comunes a las nuevas clases *LayoutNodeImpl* y *LayoutLabelNodeImpl*. De esta forma, la clase *LayoutNode* representa las características comunes a todos los tipos de nodos (reales y ficticios) presentes en un layout.

- *csplugins.layout.EdgeWeighter*
  Se agregó la lógica necesaria para manejar los pesos de las aristas "ficticias" (las que conectan a un nodo real con un nodo "ficticio" representando su etiqueta).
  Debido a que se desea en general que las etiquetas queden próximas a sus nodos padres, se computa el peso máximo y mínimo de las aristas "normales", y se utiliza un coeficiente (configurable por el usuario) para elegir el peso de las nuevas aristas.

- *csplugins.layout.algorithms.AbstractGraphPartition.java*
  Se agregaron al panel de configuración de los layouts las opciones comunes a todos los algoritmos que soportan etiquetas de forma de que estén disponibles para ellos. Se puede ver en la figura 4.1 el panel de configuración del algoritmo Fruchtermann-Reingold (llamado Biolayout dentro de Cytoscape) que contiene estas nuevas opciones.
  También se creo un *wrapper* para el método *LayoutSigleP artition* preexistente, de modo de que cuando el layout sea hecho por un algoritmo que soporte etiquetas, y haya alguna etiqueta involucrada (o sea que sea un layout sólo de etiquetas, o de etiquetas y nodos) cree una nueva *LayoutLabelPartition*, haga el layout, y traslade los resultados al grafo preexistente.

Además, se crearon las siguiente clases:

- **csplugins.layout.LayoutNodeImpl**
  Clase que extiende (hereda) de *LayoutNode*. Dado que se modificó la clase *LayoutNode* para que contenga las los métodos y campos comúnes a todos los nodos, fue necesaria la creación de nuevas clases para diferenciar los nodos "reales" y los "ficticios" (creados para

representar etiquetas). Esta clase es la que contiene la lógica para manipular los nodos "normales", y contiene en su mayoría código ya existente que fue desacoplado de la clase padre y movido aquí.

- **csplugins.layout.LayoutLabelNodeImpl**
  Clase que extiende (hereda) de *LayoutNode*. De forma análoga a la clase anterior, fue necesaria la creación de una clase para contener la lógica propia de los nodos "ficticios", y esta fue la clase creada para tal fin.

- **csplugins.layout.LayoutLabelPartition**
  Clase que extiende (hereda) de *LayoutPartition*. Es la clase que contiene la mayor parte de los cambios implementados; se encarga de crear una nueva partición con los nodos y aristas "originales" y "ficticias", además de los mapeos necesarios para poder trasladar las posiciones obtenidas desde y hacia el grafo original.

En el apéndice B se incluyen los fragmentos de código más reelevantes de esta implementación.

Usándolo todo: Fruchterman y Reingold

Con toda la infraestructura ya implementada, resta tan sólo usarla para agregarle el soporte de etiquetas al algoritmo de layout *Fruchterman y Reingold*. Ésto resultó sumamente simple, como lo describimos a continuación.

Primero, tuvimos que indicar que el algoritmo soporta etiquetas, sobreescribiendo un método heredado de la clase *AbstractGraphPartition*:

```
/**
 * This layout ssuports laying out labels
 */
public boolean supportsLabelLayout() { return true; }
```

Las opciones del panel de configuración relativas a las etiquetas tienen que ser agregadas por la clase que implementa el layout. Estas opciones son las que se resaltan en la figura 4.1. Para agregarlas a la inicialización de las opciones del layout, bastó con hacer una llamada al método *getLabelTunables*:

```
/**
 * Read all of our properties from the cytoscape properties map
 * and set the values as appropriate.
 */
public void initializeProperties() {
    super.initializeProperties();

    // ...
    // Already existing tuning values
    // ...

    // Get the label layout options group
```

```
        getLabelTunables(layoutProperties);

        // We've now set all of our tunables, so we can read the property
        // file now and adjust as appropriate
        layoutProperties.initializeProperties();

        // Finally, update everything. We need to do this to update
        // any of our values based on what we read from the property file
        updateSettings(true);
    }
```

También tuvimos que agregar la lógica que actualiza el valor de las opciones cuando se modifican desde el panel de control. Para ello, también bastó con hacer una simple llamada, en este caso al método *updateSettings*:

```
    /**
     *  Update our tunable settings
     *
     * @param force : whether or not to force the update
     */
    public void updateSettings(boolean force) {
        super.updateSettings(force);

        // ...
        // Logic already in place
        // ...

        // Update label tunables
        updateSettings(layoutProperties, force);
    }
```

Listo! Todo lo necesario para agregarle el soporte de etiquetas al algoritmo, exactamente 3 líneas de código.
Esto demuestra que realmente pudimos implementar toda la lógica necesaria para manipular etiquetas dentro de la infraestructura ya existente, y modificar un algoritmo de layout para que lo utilice es una tarea trivial.

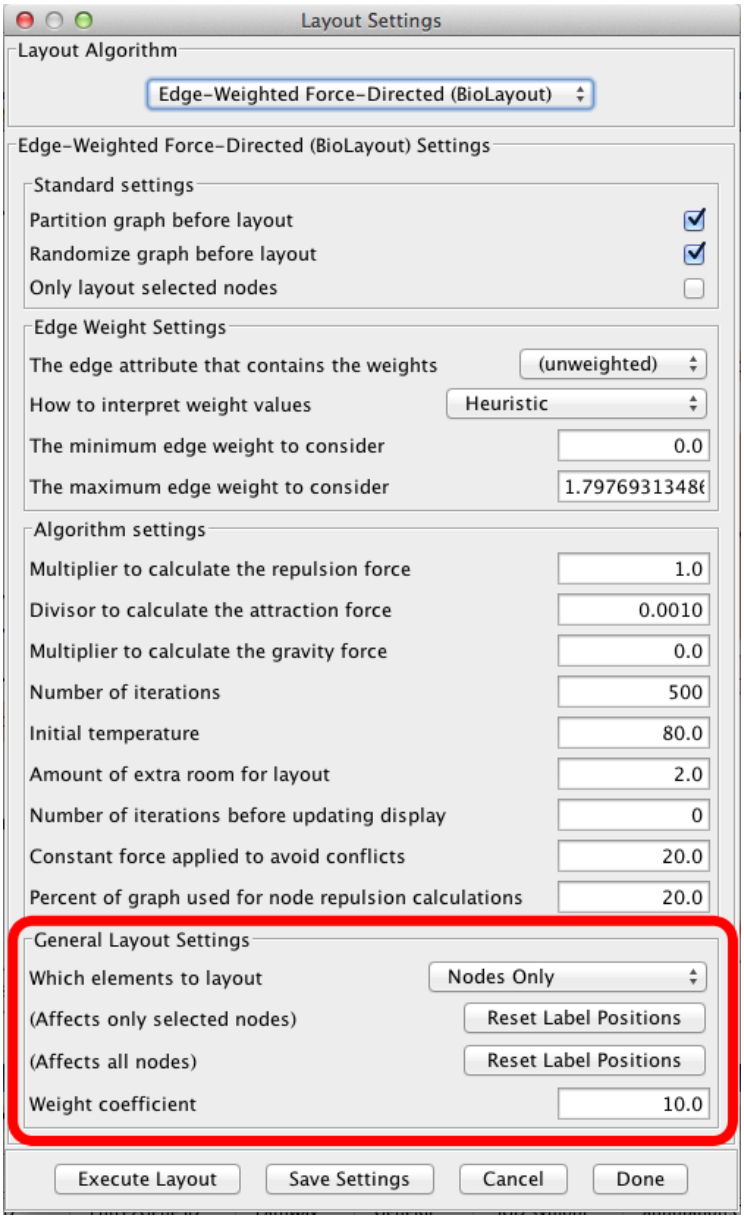


Figura 4.1: Panel de configuración del algoritmo Fruchtermann-Reingold. En rojo, las opciones que se le agregan por ser un layout que soporta etiquetas.

Capítulo 5

Resultados

En este capítulo buscaremos describir y analizar el algoritmo desarrollado, tanto desde un punto de vista teórico, así como también analizando la implementación realizada en el segundo intento.

5.1. Análisis teórico

Para este análisis, nos basaremos en el algoritmo 4, presentado en el capítulo 3.

5.1.1. Complejidad

Dado que volcar aquí el análisis de complejidad completo sería someter al lector a una tortura innecesaria, solamente presentaremos un análisis abreviado de las secciones más interesantes, y el resultado del análisis global.

Para el resto de este análisis, la entrada del algoritmo será un grafo $G = (V, E)$.

```
createDummyNodes(V,E)
```

Esta función debe crear un nodo "ficticio" y una nueva arista por cada nodo $n \in V$, y tiene una complejidad $\mathrm{O}(\|V\|)$.

```
partitionGraph(G')
```

Particiona el grafo G', que contiene no más del doble de nodos que el grafo G original. Existen varios métodos que logran esto en tiempo lineal con respecto a la suma de nodos y aristas, por lo que su complejidad es $\mathrm{O}(\|V\| + \|E\|)$.

```
randomize(Vp)
```

Asigna posiciones aleatorias a cada nodo de Vp, y su complejidad es $\mathrm{O}(\|Vp\|)$.

```
for (v in Vp) {
```

```
  v.desp = 0
  for (u in Vp not in {v}} {
    Delta = v.pos - u.pos
    v.desp += Delta / |Delta| * fr(|Delta|)
  }
}
```

Esta sección del código se encarga de computar las fuerzas repulsivas en cada iteración. Asumiendo que fr es $\mathrm{O}(1)$, esta sección tiene una complejidad $\mathrm{O}\left(\|Vp\|^2\right)$.

```
for (e in Ep) {
  Delta = e.v.pos - e.u.pos
  if (isDummy(e.v) or isDummy(e.u)) {
    F = fd(mapWeight(e.w, |Delta|))
  } else {
    F = fa(mapWeight(e.w, |Delta|))
  }
  e.v.desp += Delta / |Delta| * F
  e.u.desp -= Delta / |Delta| * F
}
```

Este fragmento del código computa todas las fuerzas atractivas dentro de una iteración. Asumiremos que el costo de $isDummy$, fd y fa es $\mathrm{O}(1)$, con lo cual todo el fragmento resulta tener una complejidad $\mathrm{O}(\|Ep\|)$.

```
for (v in Vp) {
  v.desp += gravityForce(v)
}
```

Asumiendo que $gravityForce(v)$ es $\mathrm{O}(1)$, el costo de calcular la atracción gravitatoria sobre todos los vértices de una partición tiene costo $\mathrm{O}(\|Vp\|)$.

```
updatePositions(Vp, t)
t = cool(t)
```

La porción del algoritmo que se encarga de actualizar las posiciones (itera sobre todos los nodos) y ajustar la termperatura ($\mathrm{O}(1)$) tiene costo $\mathrm{O}(\|Vp\|)$.

```
G' = mergeLayouts(P)
translatePositions(G',G)
```

Esta parte del algoritmo se encarga de unir las posiciones de los nodos de las distintas particiones, lo cual tiene como costo $\mathrm{O}\left(\sum_{p \in P} \|P\|\right) = \mathrm{O}(\|V'\|)$, y de trasladar las posiciones de los nodos (reales y ficticios) de G' a los elementos (nodos y etiquetas) del grafo original G.
El costo total de estas operaciones es $\mathrm{O}(\|V'\|)$.

Juntándolo todo

Finalmente, utilizando los resultados presentados anteriormente, estamos en condiciones de presentar la complejidad teórica del algoritmo presentado. La misma es

$$O(O(\|V\|) + O(\|V\| + \|E\|)+ \sum_{p \in P} O(\|V_p\|) + I * (O(\|V_p^2\|) + O(\|E_p\|) + O(\|V_p\|) + O(\|V_p\|))+ O(\|V'\|))$$

donde I es la cantidad de iteraciones a realizar.

Podemos simplificar la fórmula anterior, y obtenemos que la complejidad temporal de nuestro algoritmo de layout, aplicado a un grafo $G = (V, E)$ cuyas componentes conexas son P, es la siguiente:

$$\mathrm{O}\left(I * \sum_{p \in P} (\|V_p^2\| + \|E_p\|)\right) \tag{5.1}$$

5.1.2. Breve reflexión acerca de la complejidad

Dado que ya hemos acotado la complejidad temporal de nuestro algoritmo, estamos en condiciones de hacer algunas observaciones acerca del mismo.

La primer observación, es que para la gran mayoría de los grafos (y en particular para todos los grafos simples), $|E| \leq |V|^2$, por lo cual la fórmula 5.1 queda simplificada (en estos casos) a

$$\mathrm{O}\left(I * \sum_{p \in P} (\|V_p^2\|)\right) \tag{5.2}$$

Otra observación es que el hecho de particionar el grafo puede ser algo extremadamente beneficioso en cuanto a su impacto en la performance del algoritmo. Si bien para un grafo con una sola partición no presenta ventaja alguna (incluso se pierde tiempo, aunque manteniendo la misma complejidad asintótica), cuando hay presentes varias particiones de tamaño considerable, la ganancia es mayúscula.
El costo cuando hay una sola partición resulta (y asumiendo que se cumple $|E| \leq |V|^2$):

$$\mathrm{O}\left(I * \|V^2\|\right) \tag{5.3}$$

En cambio, suponiendo que las N particiones tengan todas el mismo tamaño, al dividirse el proceso (y también bajo la premisa de $|E| \leq |V|^2$), el costo resulta:

$$\mathrm{O}\left(I * \|V^2\|/N\right) \tag{5.4}$$

Por eso, vemos que en el caso óptimo (particiones de tamaño uniforme), el costo es inversamente proporcional a la cantidad de particiones.

A su vez, en todos los casos antes vistos, el costo total es linealmente proporcional a la cantidad de iteraciones a ejecutar.

5.2. Resultados experimentales

5.2.1. Performance estética

Como se comentó anteriormente, resulta imposible evaluar de forma objetiva la calidad estética de un layout, dado que en la percepción humana de los mismos intervienen factores que desconocemos, o que no podemos cuantificar.

Una alternativa posible, es definir métricas que intenten capturar alguno de los parámetros que imaginamos son los que nuestro intelecto utiliza para (inconcientemente) evaluar la calidad de un layout. La mayor ventaja de usar métricas es que nos permiten contrastar, de una forma relativamente objetiva (la subjetividad queda limitada a la elección de la métrica) los resultados de diferentes métodos.

A la hora de evaluar los resultados de nuestro método nos encontramos con la encrucijada de evaluar de alguna forma la calidad de los resultados en términos estéticos. Luego de evaluarlo, decidimos descartar la utilización de métricas, por dos razones:

1. No existe actualmente ningún otro método con el que podamos compararlo de manera justa. Para que sea justo compararlo con otro algoritmo, consideramos que sería necesario que dicho algoritmo tenga la capacidad de:
   - hacer layout de nodos y etiquetas de un grafo.
   - reposicionar las etiquetas de un grafo, manteniendo la geometría de nodos y aristas.
2. No contábamos con la infraestructura para automatizar el cómputo de la métrica. En particular, la API de Cytoscape no ofrece los métodos para obtener el tamaño de las etiquetas, por lo que aunque podamos averiguar su posición, no podemos decidir si se superpone con otro elemento del layout (nodo, arista, etiqueta).

Por las razones antes expuestas, nuestras opciones quedaron reducidas a la opción poco satisfactoria de realizar numerosas pruebas, y evaluar subjetivamente los resultados. El lector, si está lo suficientemente interesado, podrá repetir nuestras pruebas o realizar las suyas propias, pero claramente que esta metodología carece de repetibilidad. Más allá de que la metodología no nos satisface, quedamos muy satisfechos con los resultados producidos por el algoritmo, y encontramos que con pequeños ajustes manuales de los parámetros del mismo, se pueden lograr resultados muy buenos.

Detallando un poco más los resultados conseguidos, encontramos que para grafos pequeños (hasta 30 nodos), el algoritmo funciona sin requerir intervención del usuario. Luego, a medida que el tamaño aumenta (hasta 100 nodos), es posible conseguir resultados buenos mediante la modificación de algunos de los parámetros del layout. Al llegar a un determinado tamaño (más de 100 nodos), se hace más difícil el ajuste para obtener resultados aceptables en un sólo paso, y descubrimos una combinación que en dos pasos permite obtener resultados decentes en la mayoría de los casos. Este método consiste en primero deshabilitar el posicionamiento de etiquetas, de modo de lograr una estructura de nodos (buscando que no quede muy compacta), y luego hacer otra ejecución permitiendo moverse sólo a las etiquetas.

Se incluyen las figuras 5.1, 5.2, 5.3 y 5.4, que muestran unos pocos ejemplos de los resultados de nuestras pruebas.

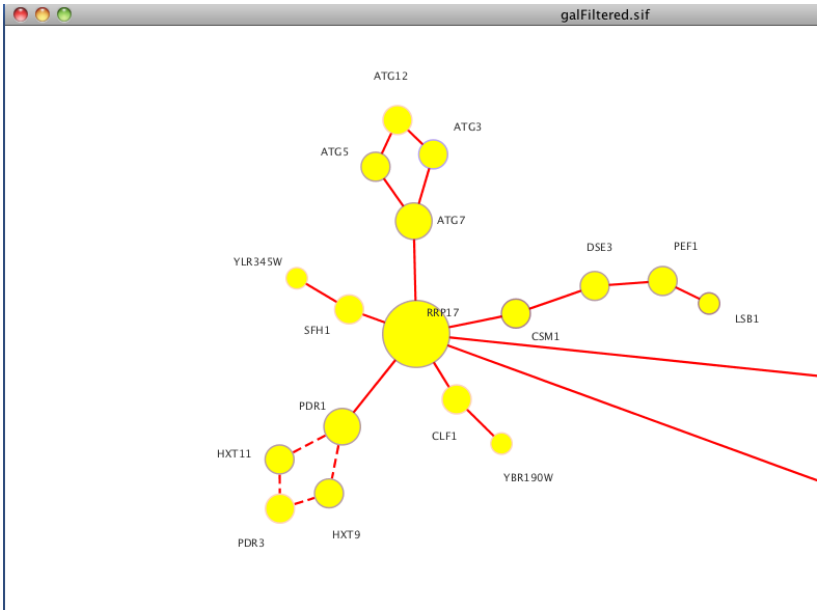


Figura 5.1: Resultado de ejecutar el algoritmo permitiendo mover tanto nodos como etiquetas, sobre una porción de un grafo.

5.2.2. Performance computacional

Se evaluó de una forma muy rudimentaria la performance de este algoritmo, ya que no contábamos con la infraestructura adecuada para para realizar mediciones precisas y tuvimos que optar por una metodología más rústica.
No era posible tomar mediciones (timestamps) sin modificar código fuente por fuera del plugin con el que estábamos trabajando, y tampoco podíamos invocar al programa Cytoscape desde línea de comando de modo de que ejecute las operaciones que deseábamos (lo que hubieramos podido medir con el comando "time" desde el sistema operativo), y de todas formas, el tiempo que tarda en abrir el programa hubiera distorsionado absolutamente los resultados.
Finalmente, decidimos comprobar manualmente que el tiempo de ejecución sea "interactivo", para grafos de un tamaño mediano ($\sim 10^2$ nodos), que es lo que nos planteamos originalmente como objetivo. Cabe la aclaración, de que al igual que con la evaluación de los resultados desde un punto de vista estético, hemos quedado ligeramente insatisfechos con esta metodología, pero la tarea de hacer un profiling más preciso hubiese requerido un esfuerzo desproporcionado e injustificado.
Durante las numerosas pruebas que realizamos, todas ellas ejecutadas en una laptop de varios años de antigüedad, nunca tuvimos que esperar excesivamente a un layout, siendo los que más demoraron aquellos que tenían menos particiones (y de mayor tamaño), lo cual es consistente con el análisis teórico que realizamos.
En base al uso experimental del algoritmo, concluímos que la performance es aceptable y cae dentro del objetivo que nos habíamos planteado inicialmente, es decir, de permitir un uso "interactivo".

5.3. Comparación con métodos existentes

Como explicamos anteriormente, no existe (al menos a nuestro entender) ningún otro método que podamos usar para contrastar la performance de nuestro algoritmo, ya sea estética o

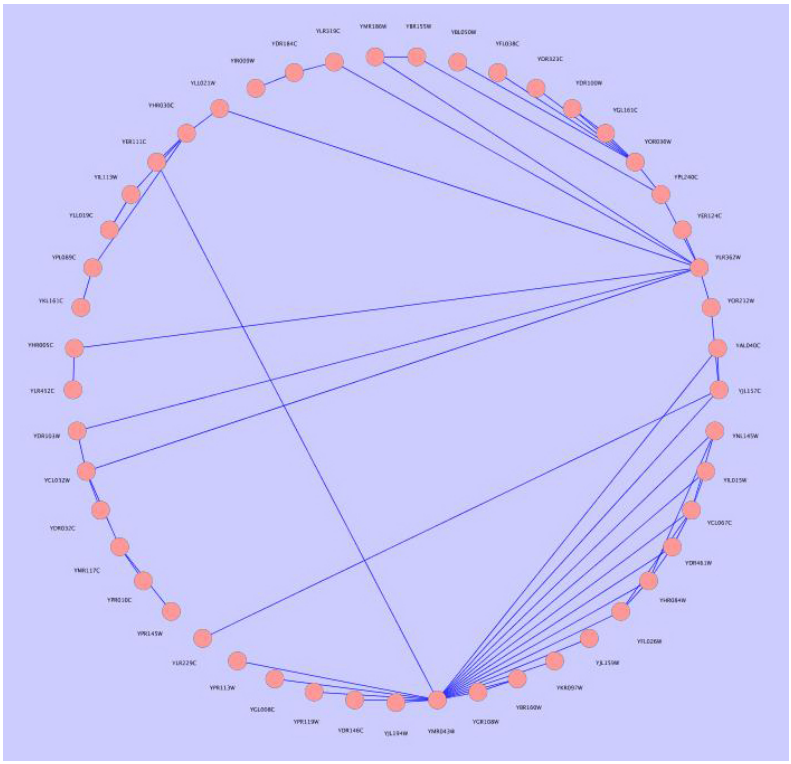


Figura 5.2: Resultado de sólo reposicionar etiquetas en un grafo previamente dispuesto en forma circular.

computacional, ya que al estar comparando métodos que ofrecen distintas funcionalidadas, la comparación necesariamente va a ser injusta hacia alguno de ellos.

5.4. Uso y aceptación

Una de las mayores dudas a la hora de crear una nueva funcionalidad, es la aceptación que pueda llegar a tener. El desafío que teníamos en este proyecto era considerable, ya que al haber implementado nuestro algoritmo como una modificación de uno de los "core plugins" de Cytoscape, no podíamos hacer uso de canales como la tienda de plugins de Cytoscape [1], sino que teníamos como única opción viable de distribución la incorporación de nuestros cambios en alguna nueva versión de la distribución oficial.

Afortunadamente, al finalizar la codificación, y luego de haber revisado nuestros cambios, los responsables de Cytoscape inmediatamente pidieron que agreguemos nuestro método en el repositorio de la versión oficial. De esta forma, nuestro algoritmo se incluyó en las versiones **2.8.2** (26/08/2011) y **2.8.3** (05/04/2012), siendo esta última el fin de la línea **2.x**.

Cytoscape tiene desde hace varios años no menos de 4.000 descargas mensuales, con picos de más de 8.000, por lo cual nuestro algoritmo se distribuyó a más de 100.000 usuarios a la fecha.

Con respecto a los posibles problemas, no se han abierto ningún reporte de errores en el sistema que posee Cytoscape para ese fin (bugtracker - Redmine), ni tampoco se han reportado inconvenientes a través de la lista de correo de la misma.

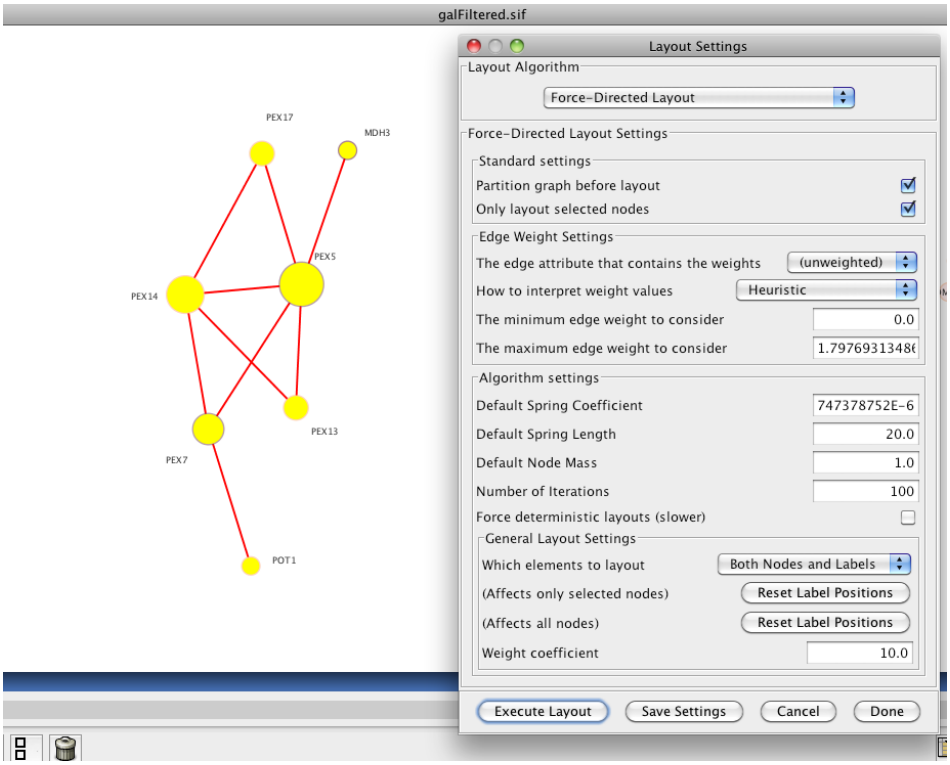


Figura 5.3: Resultado de ejecutar el layout moviendo nodos y etiquetas. Se observa además el panel de opciones de algoritmo.

Si bien no tenemos reportes confiables sobre si la mayoría de los usuarios usa, o siquiera conoce que esta opción (nuestro algoritmo) está disponible, hemos tenido unos pocos relatos de gente que lo probó y quedó satisfecha con los resultados.

Por todas las razones antes mencionadas, creemos que nuestros objetivos en cuanto a la aceptación del algorimo se han visto cumplidos.

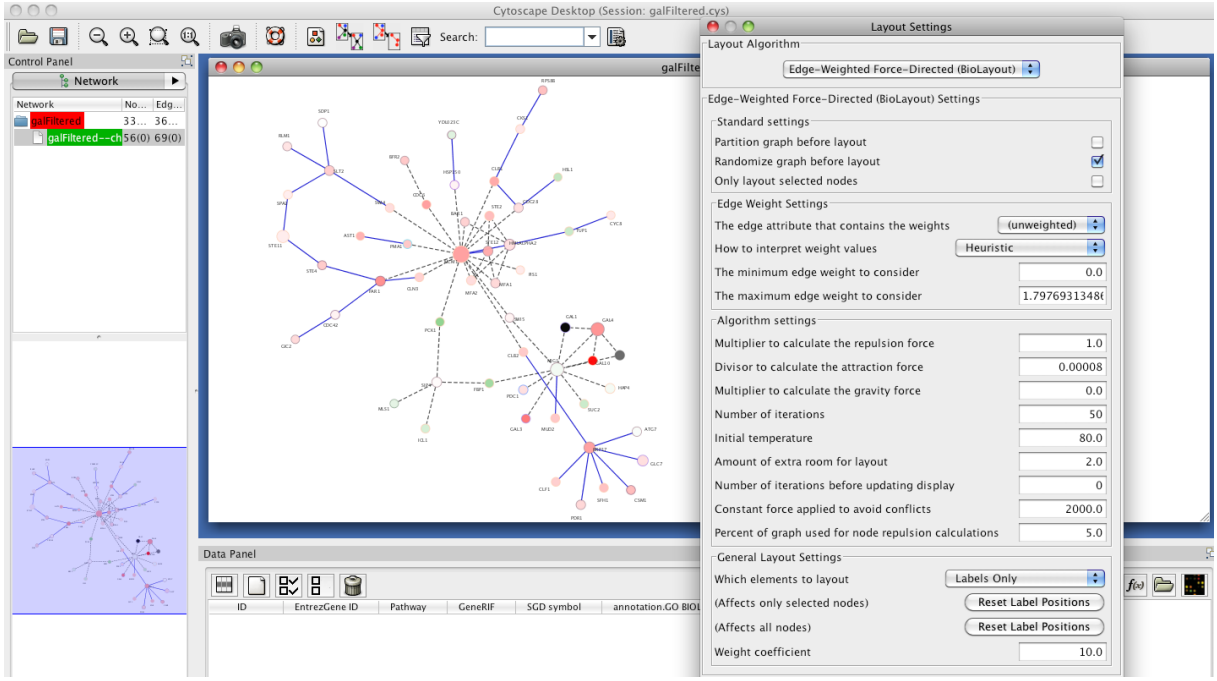


Figura 5.4: Resultado de reposicionar sólo etiquetas de un grafo previamente posicionado usando otro algoritmo force directed. Se aprecia además la interfaz del usuario completa.

Capítulo 6

Conclusiones

Corresponde hacer un balance del trabajo realizado, y para poder hacerlo propiamente, creemos conveniente recordar cuál fue nuestra motivación, y cuáles los objetivos que nos planteamos inicialmente.

Nuesta motivación fue la de contribuir al proceso de visualización de grafos, debido a que entendimos que haciendo ésto lograríamos mejorar la forma en que la gente comprende la información que ellos representan. En particular, entendímos que dentro del proceso de confección de diagramas, el posicionamiento de etiquetas era una etapa que seguía llevándose a cabo de forma muy rudimentaria. Es por dicha razón que decimos intentar mejorar el proceso de posicionamiento de etiquetas en grafos.

Antes de poder hacer un aporte significativo en pos de la meta fijada, era necesario conocer qué resultados previos existían. Nuesta búsqueda arrojó que no había métodos generales para el posicionamiento de etiquetas en grafos. También encontramos que en áreas relacionadas como en Cartografía, existían métodos que atacaban un subproblema del nuestro (ubicar las etiquetas con una geometría fija de nodos y aristas). Otro antecedente digno de nuestra atención era la existencia de algoritmos de graph layout, que también se encargaban de un subproblema (posicionar nodos y aristas, sin ocuparse de las etiquetas). Decidimos hacer un reelevamiento relativamente extenso de los métodos existentes en ambas áreas, y consideramos que este es un estado del arte que puede ser utilizado por cualquiera que quiera incursionar en nuestra misma dirección.

Contando ya con el conocimiento de qué problemas relacionados con el nuestro existían, y qué métodos se conocían para enfrentarlos, estuvimos en condiciones de hacer un aporte original. Este aporte fue la creación de un nuevo algoritmo de graph layout, basado en el algoritmo creado por Fruchterman y Reingold, que sería capaz de posicionar tanto nodos como etiquetas, ya sea en forma simultánea como también por separado.

Una vez formulado y diseñado nuestro novel algoritmo, deseábamos poder implementarlo, no sólo para poder comprobar empíricamente su correcto funcionamiento, sino también para poder ponerlo a disposición del público en general. Luego de evaluar distintas alternativas, decidimos modificar un plugin ya existente del software Cytoscape. Al poseer una licencia "libre" (LGPL), podíamos modificar y distribuir libremente nuestros cambios. A la hora de hacer la implementación, luego de un primer intento, decidimos generalizar la técnica empleada, y crear una infraestructura que le permitiría a los implementadores de otros layouts agregar soporte para etiquetas con mínimo esfuerzo, debiendo tan sólo agregar unas pocas líneas de código para

tal efecto.

El código resultante, que incluye tanto a nuestro algoritmo como a la infraestructura antes mencionada, fue incorporado en la distribución oficial de Cytoscape a partir de la versión 2.8.2, distribuyéndose así a decenas de miles de usuarios y formando parte del repositorio de código oficial de la organización, en dónde está libremente disponible para quién desee estudiarlo, utilizarlo, modificarlo y/o distribuirlo.

Finalmente, los resultados que obtuvimos mostraron que el algoritmo funciona dentro de parámetros aceptables. Si bien tiene limitaciones y queda mucho por mejorar, consideramos que es mucho más que una prueba de concepto, y actualmente está en condiciones de asistir a los usuarios a la hora de realizar diagramas en casos reales, aunque requiriendo intervención del usuario en el ajuste de parámetros y corrección manual posterior de algunos detalles del layout.

6.1. Trabajo futuro

El momento de dar por finalizado el trabajo que forma parte de esta tesina es un buen momento para reflexionar sobre cómo se podría continuar, por lo que a continuación planteamos algunas líneas de trabajo futuro que creemos valdría la pena explorar.

- Portar el algoritmo a la nueva línea 3.x de Cytoscape, dado que la línea 2.x en la que fue implementado ya ha llegado a su fin, sólo contará con actualizaciones para corregir defectos, y va a ir perdiendo popularidad progresivamente a medida que es reemplazada por las nuevas versiones de la nueva línea.
- Refinar el cálculo de fuerzas repulsivas en el algoritmo, para que se calculen teniendo en cuenta el tamaño de nodos y etiquetas, en lugar de utilizar simplemente la distancia entre los centros de ellos. Esto debería ayudar notablemente a reducir solapamientos.
- Agregar el manejo de etiquetas de aristas, actualmente no soportadas por el algoritmo.
- Experimentar con la incorporación de nuevas fuerzas a la simulación física, como por ejemplo fuerzas "angulares", que busquen rotar las etiquetas alrededor de los nodos para lograr posicionarlas lo más cercano posible a la posición preferida (por ejemplo, arriba a la derecha de cada nodo).
- Experimentar con enfoques híbridos, en los que se combinen iteraciones en las que sólo se actualizan las posiciones de nodos para lograr una estructura general del layout, y luego se incorporen las etiquetas (posiblemente reduciendo además la movilidad de nodos) y se ejecuten algunas iteraciones más para finalizar el layout.
- Traducir el estado del arte recopilado al inglés, de modo de que pueda ser aprovechado por un público más amplio.

Bibliografía

[1] Cytoscape app store. `http://apps.cytoscape.org`. Accessed: 2014-03-19.

[2] Cytoscape official web page. `http://cytoscape.org`. Accessed: 2014-03-17.

[3] Wan D. Bae, Shayma Alkobaisi, Sada Narayanappa, Petr Vojtechovsky, and Kye Y. Bae. Optimizing map labeling of point features based on an onion peeling approach. *Spatial Information Science*, 1(2):3–28, 2011.

[4] Carla Binucci, Walter Didimo, Giuseppe Liotta, and Maddalena Nonato. Labeling heuristics for orthogonal drawings. In *Graph Drawing*, pages 139–153. Springer, 2002.

[5] Carla Binucci, Walter Didimo, Giuseppe Liotta, and Maddalena Nonato. Orthogonal drawings of graphs with vertex and edge labels. *Computational Geometry*, 32(2):71–114, 2005.

[6] Jim Blythe, Cathleen McGrath, and David Krackhardt. The effect of graph layout on inference from social network data. In *Graph Drawing*, pages 40–51. Springer, 1996.

[7] Ulrik Brandes and Christian Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *Graph Drawing*, pages 42–53. Springer, 2007.

[8] Ulrik Brandes and Christian Pich. An experimental study on distance-based graph drawing. In *Graph Drawing*, pages 218–229. Springer, 2009.

[9] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, 1996.

[10] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. 1994.

[11] Uğur Doğrusöz, Brendan Madden, and Patrick Madden. Circular layout in the graph layout toolkit. In *Graph Drawing*, pages 92–100. Springer, 1997.

[12] Peter Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[13] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In *Graph Drawing*, pages 388–403. Springer, 1995.

[14] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience*, 21(11):1129–1164, 1991.

[15] Helen Gibson, Joe Faith, and Paul Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization*, 2012.

[16] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C Hart. Rapid multipole graph drawing on the gpu. In *Graph Drawing*, pages 90–101. Springer, 2009.

[17] Stephen A Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982.

[18] Eduard Imhof. Positioning names on maps. *The American Cartographer,*, 2:128–144, 1975.

[19] David S. Johnson. The np-completeness column: an ongoing guide. *Journal of Algorithms*, 3:89–99, 1982.

[20] David S. Johnson. The np-completeness column: an ongoing guide. *Journal of Algorithms*, 5:147–160, 1984.

[21] Joe Marks Jon Christensen and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 1:1–23, 1995.

[22] Konstantinos G Kakoulis and Ioannis G Tollis. A unified approach to automatic label placement. *International Journal of Computational Geometry & Applications*, 13(01):23–59, 2003.

[23] Konstantinos G. Kakoulis and Ioannis G. Tollis. *Handbook of Graph Drawing and Visualization*, chapter 15 - Labeling Algorithms, pages 489–515. CRC Press, 2013.

[24] Tomihisa KAMADA and Satoru KAWAI. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

[25] Guatam Kar, B Madden, and RS Gilbert. Heuristic layout algorithms for network management presentation services. *Network, IEEE*, 2(6):29–36, 1988.

[26] Scott Kirkpatrick, D. Gelatt Jr., and Mario P Vecchi. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[27] Gunnar W Klau and Petra Mutzel. Combining graph labeling and compaction. In *Graph Drawing*, pages 27–37. Springer, 1999.

[28] Joseph B Kruskal and Judith B Seery. Designing network diagrams. In *Proc. First General Conf. on Social Graphics*, pages 22–50, 1980.

[29] Joseph Marks and Stuart M Shieber. *The computational complexity of cartographic label placement*. Citeseer, 1991.

[30] Gilberto Camara Missae Yamamoto and Luiz Antonio Nogueira Lorena. Tabu search heuristic for point-feature cartographic label placement. *GeoInformatica*, 6:77–90, 2000.

[31] Gilberto Camara Missae Yamamoto and Luiz Antonio Nogueira Lorena. Fast point-feature label placement algorithm for real time screen maps. *Proceedings of the 7th Brazilian Symposium on GeoInformatics*, 1:1–13, November 2005.

[32] Paul Shannon, Andrew Markiel, Owen Ozier, and et al. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13:2498–2504, 2003.

[33] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, 1981.

[34] Ioannis Tollis, Peter Eades, Giuseppe Di Battista, and Loannis Tollis. *Graph drawing: algorithms for the visualization of graphs*, volume 1. Prentice Hall New York, 1998.

[35] William T Tutte. How to draw a graph. *Proc. London Math. Soc*, 13(3):743–768, 1963.

[36] William Thomas Tutte. Convex representations of graphs. *Proceedings London Mathematical Society*, 10(38):304–320, 1960.

[37] Brendan Madden Ugur Dogrusoz, Konstantinos G. Kakoulis and Ioannis G. Tollis. On labeling in graph visualization. *Information Sciences*, 177:2459–2472, 2007.

[38] Oleg V Verner, Roger L Wainwright, and Dale A Schoenefeld. Placing text labels on maps and diagrams using genetic algorithms with masking. *INFORMS Journal on Computing*, 9(3):266–275, 1997.

[39] Alexander Wolf. *Automated Label Placement in Theory and Practice.* PhD thesis, Fachbereich Mathematik und Informatik. Freie Universitï¿½t Berlin, 1999.

[40] Missae Yamamoto and Luiz A. N. Lorena. *Metaheuristics: Progress as Real Problem Solvers*, chapter 13 - A Constructive Genetic Approach To Point-Feature Cartographic Label Placement, pages 285–300. Springer, 2005.

[41] Pinhas Yoeli. The logic of automated map lettering. *Cartographic Journal, The*, 9(2):99–108, 1972.

Apéndices

Apéndice A

Código primer intento

A continuación se incluye el código de la clase en la cual se implementó casi la totalidad de la lógica necesaria para nuestro algoritmo, como parte del primer intento.
Esta nueva clase extiende la funcionalidad de la clase *BioLayoutFRAlgorithm*, y utiliza el algoritmo de layout provisto por ella luego de haber creado una partición con nuevos nodos que representan a las etiquetas.

```
/**
* Copyright (C) Gerardo Huck, 2010
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU Lesser General Public License as published
* by the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU Lesser General Public License for more details.
*
* You should have received a copy of the GNU Lesser General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*/

/**
 * This algorithms uses a node layout algorithm as a Label layout one, by
 * creating fake nodes (representing the labels).
 * This creation is implemented in the csplugins.layout package,
 * and returns a partition which is passed to the previously existing algorithm
 * as an argument for the layout.
 *
 * It was done as part of Google Summer of Code 2010.
 * Mentor : Mike Smoot
 * Student : Gerardo Huck
 *
 * @author <a href="mailto:gerardohuck .at. gmail .dot. com">Gerardo Huck</a>
 * @version 0.1
 */
package cytoscape.layout.label;
```

```
import csplugins.layout.LayoutEdge;
import csplugins.layout.LayoutNode;
import csplugins.layout.LayoutLabelNodeImpl;
import csplugins.layout.LayoutPartition;
import csplugins.layout.LayoutLabelPartition;
import csplugins.layout.Profile;

import cytoscape.Cytoscape;
import cytoscape.CyNode;
import cytoscape.data.CyAttributes;
import cytoscape.layout.LayoutProperties;
import cytoscape.layout.Tunable;
import cytoscape.layout.CyLayouts;
import cytoscape.layout.CyLayoutAlgorithm;
import cytoscape.layout.AbstractLayout;
import cytoscape.logger.CyLogger;

import java.awt.Dimension;
import giny.view.NodeView;
import giny.view.*;
import java.util.*;
import java.lang.Thread;


public class LabelBioLayoutFRAlgorithm extends ModifiedBioLayoutFRAlgorithm {

    /**
     * Whether Labels should be repositioned in their default positions
     */
    private boolean resetPosition = false;

    /**
     * Whether network nodes will be moved or not
     */
    private boolean moveNodes = false;

    /**
     * Coefficient to determine label edge weights
     */
    private double weightCoefficient = 10.0;

    /**
     * This is the constructor for the bioLayout algorithm.
     */
    public LabelBioLayoutFRAlgorithm(boolean supportEdgeWeights) {
        super(supportEdgeWeights);
        logger = CyLogger.getLogger(LabelBioLayoutFRAlgorithm.class);
        supportWeights = supportEdgeWeights;
        this.initializeProperties();
    }

    /**
     * Return the "name" of this algorithm. This is meant
     * to be used by programs for deciding which algorithm to
```

```
 * use. toString() should be used for the human-readable
 * name.
 *
 * @return the algorithm name
 */
public String getName() {
    return "Fruchterman-Rheingold-Label-Layout";
}

/**
 * Return the "title" of this algorithm. This is meant
 * to be used for titles and labels that represent this
 * algorithm.
 *
 * @return the human-readable algorithm name
 */
public String toString() {
    return "Force-Directed Label Layout";
}

/**
 * We don't want to use the label layout capabilities offered
 * by AbstractGraphPartition
 */
public boolean supportsLabelLayout() {
    return false;
}

/**
 * Reads all of our properties from the cytoscape properties map and sets
 * the values as appropriates.
 */
public void initializeProperties() {

    super.initializeProperties();

    layoutProperties.add(new Tunable("labels_settings",
                                     "Label specific settings",
                                     Tunable.GROUP, new Integer(3)));

    layoutProperties.add(new Tunable("resetPosition",
                                     "Reset label positions",
                                     Tunable.BOOLEAN, new Boolean(false)));

    layoutProperties.add(new Tunable("moveNodes",
                                     "Allow nodes to move",
                                     Tunable.BOOLEAN, new Boolean(false)));

    layoutProperties.add(new Tunable("weightCoefficient",
                                     "weightCoefficient",
                                     Tunable.DOUBLE, new Double(weightCoefficient)));

    // We've now set all of our tunables, so we can read the property
    // file now and adjust as appropriate
    layoutProperties.initializeProperties();
```

```
    // Finally, update everything. We need to do this to update
    // any of our values based on what we read from the property file
    updateSettings(true);
}

/**
 *  update our tunable settings
 */
public void updateSettings() {
    updateSettings(false);
}

/**
 *  update our tunable settings
 *
 * @param force whether or not to force the update
 */
public void updateSettings(boolean force) {

    super.updateSettings(force);

    Tunable t = layoutProperties.get("resetPosition");
    if ((t != null) && (t.valueChanged() || force))
        resetPosition = ((Boolean) t.getValue()).booleanValue();

    t = layoutProperties.get("moveNodes");
    if ((t != null) && (t.valueChanged() || force))
        moveNodes = ((Boolean) t.getValue()).booleanValue();

    t = layoutProperties.get("weightCoefficient");
    if ((t != null) && (t.valueChanged() || force))
        weightCoefficient = ((Double) t.getValue()).doubleValue();
}

/**
 * Perform a layout
 */
public void layoutPartition(LayoutPartition partition) {

    Dimension initialLocation = null;

    if (canceled)
        return;

    // Reset the label position of all nodes if necessary
    if (resetPosition) {
        resetNodeLabelPosition(partition);
        return;
    }

    // Logs information about this task
    logger.info("Laying out partition " + partition.getPartitionNumber() +
                " which has "+ partition.nodeCount() + " nodes and " +
                partition.edgeCount() + " edges: ");
```

```
// Create new Label partition
LayoutLabelPartition newPartition = new LayoutLabelPartition(partition,
                                                             weightCoefficient,
                                                             moveNodes,
                                                             selectedOnly);

// logger.info("New partition succesfully created!");

// Figure out our starting point - This will be used when:
// 1- Laying out labels off all nodes
// - and-
// 2- (normal) Nodes are not allowed to move
if (selectedOnly && moveNodes) {
    newPartition.recalculateStatistics();
    initialLocation = newPartition.getAverageLocation();
}

if (canceled)
    return;

// Layout the new partition using the parent class layout algorithm
super.layoutPartition(newPartition);

if (canceled)
    return;

// Not quite done, yet. We may need to migrate labels back to their starting
    position
// This will be necessary if:
// 1- Laying out only selected nodes
// - and -
// 2- (normal) Nodes are allowed to move

taskMonitor.setStatus("Making final arrangements...");

if (selectedOnly && moveNodes) {
    logger.info("moving back labels (and possibly nodes) to their location");

    newPartition.recalculateStatistics();
    Dimension finalLocation = newPartition.getAverageLocation();
    double xDelta = 0.0;
    double yDelta = 0.0;

    xDelta = finalLocation.getWidth() - initialLocation.getWidth();
    yDelta = finalLocation.getHeight() - initialLocation.getHeight();

    for (LayoutNode v: newPartition.getNodeList()) {
        if (!v.isLocked()) {
            v.decrement(xDelta, yDelta);
            newPartition.moveNodeToLocation(v);
        }
    }
}
```

```
        // make sure nodes are where they should be
        for (LayoutNode node: newPartition.getLabelToParentMap().values()) {
            if (canceled)
                return;
            node.moveToLocation();
            logger.info( node.toString() );
        }

        // make sure that all labels are where they should be
        for (LayoutLabelNodeImpl node: newPartition.getLabelNodes()) {
            if (canceled)
                return;
            node.moveToLocation();
            logger.info( node.toString() );
        }

        taskMonitor.setStatus("Updating Display...");

        // redraw the network so that the new label positions are visible
        networkView.updateView();
        networkView.redrawGraph(true, true);

        logger.info("Label/Node layout of partition " + partition.getPartitionNumber()
            + " complete");
    }

    /**
     * Moves labels to the same position in which their parent nodes are
     */
    protected void resetNodeLabelPosition(LayoutPartition part) {
        logger.info("Reseting labels position");

        CyAttributes nodeAtts = Cytoscape.getNodeAttributes();

        // Go through all labels setting their position to be the same as their
            parent's
        List<LayoutNode> array = part.getNodeList();

        for (LayoutNode node: array) {
            if (!selectedOnly || !node.isLocked()) {
                if (nodeAtts.hasAttribute(node.getIdentifier(), "node.labelPosition"))
                    {
                    nodeAtts.deleteAttribute(node.getIdentifier(),
                        "node.labelPosition");
                }
            }
        }

        // redraw the network so that the new label positions are visible
        networkView.updateView();
        networkView.redrawGraph(true, true);
    }

}
```

Apéndice B

Código segundo intento

B.1. Particiones

Se presenta a continuación el código fuente de la clase *csplugins.layout.LayoutLabelPartition*.

```
/**
* Copyright (C) Gerardo Huck, 2010
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU Lesser General Public License as published
* by the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU Lesser General Public License for more details.
*
* You should have received a copy of the GNU Lesser General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*/

/**
 * This handles the creation and behavior of a "partition" used to layout labels
 * as well as nodes.
 *
 * It was done as part of Google Summer of Code 2010.
 * Mentor: Mike Smoot
 * @author <a href="mailto:gerardohuck .at. gmail .dot. com">Gerardo Huck</a>
 * @version 0.1
 */

package csplugins.layout;

import cytoscape.logger.CyLogger;
import csplugins.layout.LayoutEdge;
import csplugins.layout.LayoutNode;
import csplugins.layout.LayoutLabelNodeImpl;
import csplugins.layout.Profile;
```

```
import csplugins.layout.EdgeWeighter;
import cytoscape.util.intr.IntIntHash;
import cytoscape.util.intr.IntObjHash;
import cytoscape.*;
import cytoscape.view.*;

import giny.view.*;
import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Map;
import java.util.Comparator;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
import java.util.Random;
import java.util.Set;
import java.util.Collection;


/**
 * The LayoutLabelPartition class ....
 *
 * @version 0.1
 */
public class LayoutLabelPartition extends LayoutPartition {

    protected Map <LayoutLabelNodeImpl,LayoutNode> labelToParentMap;
    protected ArrayList<LayoutNode> allLayoutNodesArray;
    protected ArrayList<LayoutEdge> allLayoutEdgesArray;

    protected double weightCoefficient = 1.0;
    protected boolean moveNodes = false;
    protected boolean selectedOnly = false;
    protected boolean isWeighted = false;

    public LayoutLabelPartition(int nodeCount, int edgeCount) {
        super(nodeCount, edgeCount);
    }

    public LayoutLabelPartition(LayoutPartition part,
                                double weightCoefficient,
                                boolean moveNodes,
                                boolean selectedOnly) {

        super(part.size(), part.getEdgeList().size());

        this.weightCoefficient = weightCoefficient;
        this.moveNodes       = moveNodes;
        this.selectedOnly    = selectedOnly;
        this.isWeighted      = isWeighted;

        // Copy fields from part
```

```
    this.nodeList = (ArrayList<LayoutNode>) part.nodeList.clone();
    this.edgeList = (ArrayList<LayoutEdge>) part.edgeList.clone();
    this.nodeToLayoutNode = (HashMap<CyNode,LayoutNode>)
        part.nodeToLayoutNode.clone();
    this.partitionNumber = part.partitionNumber;
    this.edgeWeighter = part.edgeWeighter;
    this.maxX = part.maxX;
    this.maxY = part.maxY;
    this.minX = part.minX;
    this.minY = part.minY;
    this.width = part.width;
    this.height = part.height;
    this.averageX = part.averageX;
    this.averageY = part.averageY;
    this.lockedNodes = part.lockedNodes;

    // Initialize 'label' fields
    labelToParentMap  = new HashMap<LayoutLabelNodeImpl,LayoutNode>(part.size());
    allLayoutNodesArray = new ArrayList<LayoutNode>(part.size());
    allLayoutEdgesArray = new ArrayList<LayoutEdge>(part.size());

    // This method handles the creation of LayoutLabelNodes, the mappings, etc
    this.initializeLabels();

    // Reset statistics in order to reflect the new partition status.
    this.recalculateStatistics();
}

/**
 * Handles the initization of all the infraestructure needed in order to use a
 * LayoutLabelPartition to layout labels, such as Label nodes creation, etc.
 */
protected void initializeLabels() {

    edgeWeighter.setLabelWeightCoefficient(weightCoefficient);
    edgeWeighter.calculateMaxWeight();

    for (LayoutNode ln: nodeList) {
        // Creates a new LabelNode, child of ln
        NodeView nv = ln.getNodeView();
        int ind = ln.getIndex() + nodeList.size();

        // wonder how this is even possible
        if (nv == null) {
            logger.error("Found a layout node without a NodeView!");
            continue;
        }

        LayoutLabelNodeImpl labelNode = new LayoutLabelNodeImpl(nv, ind);

        /* Unlock labelNode if:
         * - algorithm is to be applied to the entire network
         * - algorithm is to be applied to the selected nodes only, and ln
         * is selected
         */
```

```
        if (!selectedOnly || !(ln.isLocked())) {
            labelNode.unLock();
            updateMinMax(labelNode.getX(), labelNode.getY());
            this.width += labelNode.getWidth();
            this.height += labelNode.getHeight();
        } else {
            labelNode.lock();
            lockedNodes++;
        }

        // Lock ln if it is unlocked but nodes are not allowed to move
        if (!moveNodes && !ln.isLocked()) {
            ln.lock();
            lockedNodes++;
        }

        // Add labelNode -> parentNode to Map
        labelToParentMap.put(labelNode, ln);

        // Creates label Edge: ln <---> labelNode
        LayoutEdge labelEdge = new LayoutEdge();
        labelEdge.addNodes(ln, labelNode);

        // This takes care of the weight of this edge
        updateWeights(labelEdge);

        // Adds it to edgeList
        edgeList.add(labelEdge);
    }

    // Adds all LabelNodes to nodeList
    nodeList.addAll(labelToParentMap.keySet());
}

public void calculateEdgeWeights() {
    Double weight = null;

    // -- First set labelEdges weights --

    // Calculate maximum preexisting weight
    Double maxWeight = new Double(Double.MIN_VALUE);

    ListIterator<LayoutEdge>iter = edgeList.listIterator();

    while (iter.hasNext()) {
        LayoutEdge edge = iter.next();

        // Only consider non label edges
        if (labelToParentMap.keySet().contains(edge.getSource()) ||
            labelToParentMap.keySet().contains(edge.getTarget())) {
            continue;
        }

        if (edge.getWeight() > maxWeight) {
            maxWeight = edge.getWeight();
```

```
            }
        }

        // Value which will be used as weight for all label edges
        weight = maxWeight * weightCoefficient;

        // Set all labelEdge weights
        ListIterator<LayoutEdge>iter2 = edgeList.listIterator();

        while (iter2.hasNext()) {
            LayoutEdge edge = iter2.next();

            if (labelToParentMap.keySet().contains(edge.getSource()) ||
                labelToParentMap.keySet().contains(edge.getTarget()) ) {
                // Set label edges weights
                edge.setWeight(weight);
            }
        }

        // -- Then call parent method to calculate all weights --
        super.calculateEdgeWeights();
    }

    public Map<LayoutLabelNodeImpl, LayoutNode> getLabelToParentMap() {
        return labelToParentMap;
    }

    /**
     * Returns a list with all the LayoutLabelNodes
     */
    public ArrayList<LayoutLabelNodeImpl> getLabelNodes() {
        ArrayList<LayoutLabelNodeImpl> array = new ArrayList<LayoutLabelNodeImpl> ();
        array.addAll(labelToParentMap.keySet());
        return array;
    }
}
```

B.2. Pesos de aristas

Se incluye el método de la clase *csplugins.layout.EdgeWeighter* responsable por el cómputo del peso de las nuevas aristas ("ficticias").

```
    /**
     * Computes the weight for a Label LayoutEdge
     */
    public void setWeightLabelEdge(LayoutEdge layoutEdge) {
        //      logger.info("Setting weight for label edge: " +
        //                  layoutEdge + " using " + weightAttribute);

        double eValue = labelWeightCoefficient * maxWeightSnapshot;

        layoutEdge.setWeight(eValue);
        minWeight = Math.min(minWeight,eValue);
```

```
        maxWeight = Math.max(maxWeight,eValue);

        if (type == WeightTypes.GUESS || type == WeightTypes.LOG) {
            double logWeight;
            if (eValue == 0) {
                logWeight = logWeightCeiling;
                logOverflow = true;
            } else {
                logWeight = Math.min(-Math.log10(eValue), logWeightCeiling);
            }

            minLogWeight = Math.min(minLogWeight,logWeight);
            maxLogWeight = Math.max(maxLogWeight,logWeight);
            layoutEdge.setLogWeight(logWeight);
        }
    }
```

B.3. Nodos "ficticios"

El código responsables por la diferencia en el comportamiento de los nodos creados para representar etiquetas se incluye a continuación.

```
/**
* Copyright (C) Gerardo Huck, 2010
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU Lesser General Public License as published
* by the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU Lesser General Public License for more details.
*
* You should have received a copy of the GNU Lesser General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*/

/**
 * This class represents the fake nodes used to layout labels.
 *
 * It was done as part of Google Summer of Code 2010.
 * Mentor: Mike Smoot
 * @author <a href="mailto:gerardohuck .at. gmail .dot. com">Gerardo Huck</a>
 * @version 0.1
 */

package csplugins.layout;

import cytoscape.*;
import cytoscape.logger.CyLogger;
```

```
import cytoscape.data.CyAttributes;
import cytoscape.view.*;
import cytoscape.visual.parsers.*;
import cytoscape.visual.VisualPropertyType;
import ding.view.ObjectPositionImpl;
import giny.view.*;
import java.util.*;

public class LayoutLabelNodeImpl extends LayoutNode {

    // -- Instance variables inherited from superclass --
    //  protected boolean isLocked = false;
    //  protected double x;
    //  protected double y;
    //  protected double dispX;
    //  protected double dispY;

    protected CyLogger logger = null;
    protected NodeView parentNodeView;
    protected ObjectPosition lp;
    protected CyAttributes nodeAtts = null;

    /**
     * Empty constructor
     */
    public LayoutLabelNodeImpl() {}

    /**
     * The main constructor for a LayoutLabelNode.
     *
     * @param parent The parent LayoutNode for this LayoutLabelNode
     */
    public LayoutLabelNodeImpl(NodeView parentNodeView, int index) {
        logger = CyLogger.getLogger(LayoutLabelNodeImpl.class);
        this.parentNodeView = parentNodeView;

        // Set labelNode's location to parent node's current label position
        nodeAtts = Cytoscape.getNodeAttributes();
        String labelPosition = (String)
            nodeAtts.getAttribute(parentNodeView.getNode().getIdentifier(),
                                                        "node.labelPosition");

        if (labelPosition == null) {
            lp = new ObjectPositionImpl();
            logger.info("Created new ObjectPosition!");
        } else {
            ObjectPositionParser parser =
                (ObjectPositionParser)
                    VisualPropertyType.NODE_LABEL_POSITION.getValueParser();
            lp = parser.parseStringValue(labelPosition);
            logger.info("ObjectPosition succesfully parsed!");
        }

        try {
            logger.info("Parent node: " + parentNodeView.getNode().getIdentifier());
```

```
            logger.info("Offset = " + lp.getOffsetX() + ", " + lp.getOffsetY() );

            this.setX(lp.getOffsetX() + parentNodeView.getXPosition());
            this.setY(lp.getOffsetY() + parentNodeView.getYPosition());
            this.neighbors = new ArrayList<LayoutNode>();
            this.index = index;

            logger.info("Created " + this.getIdentifier() + "placed in: " +
                this.getX() + ", " + this.getY() );
        }
        catch(Exception e) {
            // Log error
            logger.warning("Error detected while creating LayoutLabelNodeImp: " +
                e.toString() );

            // Reset this attribute by eliminating it
            if (nodeAtts.hasAttribute(parentNodeView.getNode().getIdentifier(),
                "node.labelPosition")) {
                nodeAtts.deleteAttribute(parentNodeView.getNode().getIdentifier(),
                    "node.labelPosition");
                logger.warning("Deleted 'node.labelPosition' attribute");
            }

            lp = new ObjectPositionImpl();
            logger.warning("Created new ObjectPosition!");

            this.setX(parentNodeView.getXPosition());
            this.setY(parentNodeView.getYPosition());
            this.neighbors = new ArrayList<LayoutNode>();
            this.index = index;
        }
    }

    /**
     * Moves a label node to the (X,Y) position that is already defined if it is
         unlocked.
     * Otherwise, updates the (X,Y) fields in order to reflect its real position.
     * Note that moving the parent node will affect the position of this label node.
     */
    public void moveToLocation() {

        lp = new ObjectPositionImpl();

        if (this.isLocked()) { // If node is locked, adjust X and Y to its current
            location

            logger.info(this.toString() + " was locked");

            this.setX(lp.getOffsetX() + parentNodeView.getXPosition());
            this.setY(lp.getOffsetY() + parentNodeView.getYPosition());

        } else { // If node is unlocked set labels offsets properly

            logger.info(this.toString() + " was unlocked");
```

```
        lp.setOffsetX(this.getX() - parentNodeView.getXPosition());
        lp.setOffsetY(this.getY() - parentNodeView.getYPosition());

        nodeAtts.setAttribute(parentNodeView.getNode().getIdentifier(),
                              "node.labelPosition", lp.shortString());

        logger.info("Label node was moved!, short string: " + lp.shortString());
        logger.info("full label string: " + lp.toString() );
    }
}

/**
 * Accessor function to return the NodeView associated with
 * this LayoutNode.
 *
 * @return   NodeView that is associated with this LayoutNode
 */
public NodeView getNodeView() {
    return parentNodeView;
}

/**
 * Return the width of this node
 *
 * @return      width of this node
 */
public double getWidth() {
    return parentNodeView.getWidth();
}

/**
 * Return the height of this label node
 *
 * @return      height of this node
 */
public double getHeight() {
    return parentNodeView.getHeight();
}

public double getParentX() {
    return parentNodeView.getXPosition();
}

public double getParentY() {
    return parentNodeView.getYPosition();
}

/**
 * Return a string representation of the node
 *
 * @return      String containing the node's identifier and location
 */
public String toString() {
    return "Label of Node: " + getIdentifier() + " at " + printLocation();
}
```

```
    /**
     * Return a string with the "type" of the node ("normal", "label")
     *
     * @return      String containing the node's type
     */
    public String getType() {
        return "label";
    }

    /**
     * Return the node's identifier.
     *
     * @return      String containing the node's identifier
     */
    public String getIdentifier() {
        return "Label of node:" + parentNodeView.getNode().getIdentifier();
    }
}
```

B.4. Juntándolo todo: AbstractGraphPartition

La clase *AbstractGraphPartition* es el punto de entrada a todo nuestro código, siendo la responsable de crear e inicializar las particiones, y llamar al método que efectúa al layout. En esta sección presentamos los fragmentos que se vieron más impactados por nuestros esfuerzos.

La definición de los distintos tipos de layout posibles:

```
enum LayoutTypes {
    NODE("Nodes Only"),
    LABEL("Labels Only"),
    BOTH("Both Nodes and Labels");

    private String name;
    private LayoutTypes(String str) { name=str; }
    public String toString() { return name; }
}
```

```
    static LayoutTypes[] layoutChoices = {LayoutTypes.NODE,
                                          LayoutTypes.LABEL,
                                          LayoutTypes.BOTH};
```

Método que puede ser escrito por una clase heredera, que especifíca si un determinado algoritmo de layout tiene soporte para etiquetas:

```
    /**
     * If overridden by a subclass to return true, before calling to
     * layoutPartition mathod, fake nodes will be created to represent label
```

```
     * positions, and labels will be placed acordingly.
     *
     * It is an easy way of creating label layout algorithms.
     *
     * For this to work fine with labels, weights should be supported.
     *
     * @return Whether the layout algorithm supports label layout.
     */
    public boolean supportsLabelLayout() {
        return false;
    }
```

Método encargado de hacer el layout de una partición, modificado para (en caso necesario) llamar al constructor de *LayoutLabelPartition* para crear los nodos "ficticios", y luego de haber sido hecho el layout de la nueva particion trasladar los resultados a la partición original:

```
    /**
     * DOCUMENT ME!
     */
    protected void layoutSinglePartition(LayoutPartition partition) {

        if (supportsLabelLayout() && layoutType != LayoutTypes.NODE) {

            Dimension initialLocation = null;

            if (canceled)
                return;

            Boolean moveNodes;

            if (layoutType == LayoutTypes.BOTH)
                moveNodes = true;
            else
                moveNodes = false;

            // Create new Label partition
            LayoutLabelPartition newPartition = new LayoutLabelPartition(partition,
                                                                         weightCoefficient,
                                                                         moveNodes,
                                                                         selectedOnly);

            logger.info("New partition succesfully created!");

            if (canceled)
                return;

            // Layout the new partition using the parent class layout algorithm
            layoutPartition(newPartition);

            if (canceled)
                return;
```

```
        // Make sure nodes are where they should be
        for (LayoutNode node: newPartition.getLabelToParentMap().values()) {

            if (canceled)
                return;

            node.moveToLocation();
            logger.info( node.toString() );
        }

        // Make sure that all labels are where they should be
        for (LayoutLabelNodeImpl node: newPartition.getLabelNodes()) {

            if (canceled)
                return;

            node.moveToLocation();
            logger.info(node.toString());
        }

        taskMonitor.setStatus("Updating Display...");

        // Redraw the network so that the new label positions are visible
        networkView.updateView();
        networkView.redrawGraph(true, true);

    } else { // Normal (non-label) layout
        layoutPartition(partition);
    }
}
```