Entorno web para visualizar grafos preservando simetrías

Javier Morales
Santiago Serantes

Supervisores: Diego Kiedanski, Federico Rivero, Eduardo Grampín

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Facultad de Ingeniería
Universidad de la República UDELAR
Montevideo, Uruguay
2018

Resumen

En el transcurso de este proyecto se desarrolló una aplicación web para dibujo automático y simétrico de grafos pequeños. Las herramientas existentes para el dibujo de grafos utilizan algoritmos que no tienen como objetivo realizar un representación simétrica, lo que resulta en grafos poco agradables estéticamente y dificulta la visualización al usuario. Una de las mayores complejidades radica en que el problema de detección de simetrías en grafos es NP-completo. Por ello, para la representación se utilizaron heurísticas basadas en el problema de partición de automorfismos, que tiene solución en tiempo cuasi-polinomial.

Además, se contemplan criterios de estéticos de visualización. La aplicación presenta resultados apropiados en grafos de hasta aproximadamente 20 nodos y es compatible con LaTeX, siendo su uso en documentos y publicaciones la principal motivación. Se enfocó el desarrollo para una alta usabilidad y baja latencia. Para su validación, se realizó una prueba de usabilidad y un estudio de estrés, ambos con resultados satisfactorios.

Índice general

Capítulo 1

Introducción

Un grafo es una construcción matemática compuesta por dos conjuntos: uno de vértices o nodos y uno de aristas que unen dichos vértices. Su estructura los hace aptos para el modelado de numerosos problemas, pues la existencia de entidades conectadas entre sí es frecuente en la realidad.

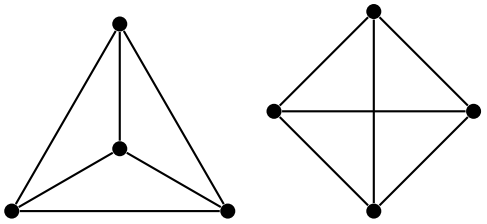
La función principal de un grafo es la abstracción o modelado de un problema. Sobre un problema abstraído como un grafo se pueden aplicar propiedades y algoritmos con el fin de analizarlo o resolverlo. El foco en este proyecto, sin embargo, es que los grafos son también una herramienta visual. En la disciplina de *data visualization*, cómo se ve un grafo es un factor muy relevante. En muchos casos, una buena representación ayuda a la resolución o a la mejor comprensión de un problema.

¿Cuándo es buena una representación? Todo grafo puede ser dibujado en un plano de infinitas maneras. Existen múltiples criterios de calidad aplicables a las representaciones, así como numerosos estudios que analizan su importancia relativa en la percepción humana[21, 22]. Un criterio muy utilizado consiste en minimizar la cantidad de cruces de aristas. Aquellos grafos que pueden ser representados sin cruces de aristas se llaman grafos planos y cumplen ciertas propiedades que permiten identificarlos [2]. Otros criterios incluyen intentar que la relación de aspecto (*aspect ratio*[1]) se aproxime a cierto valor, minimizar la suma de la longitud de las aristas o minimizar la varianza de esas longitudes, maximizar la resolución angular[2] o maximizar la cantidad de simetrías. Los criterios suelen no ser coordinables, es decir, mejoras en uno de ellos pueden implicar peores resultados en los otros, efecto

[1] la relación del tamaño del grafo en el eje horizontal con respecto al vertical.

[2] Es la mínima diferencia angular entre dos aristas de un mismo nodo.

que puede verse en la figura 1.1.



(a) (b)

Métrica	**a**	**b**
Cantidad de cortes de aristas	**0**	1
Varianza de longitudes de aristas	0.136	**0.071**
Resolución angular	$\pi/6$	$\boldsymbol{\pi/4}$
Número de simetrías	6	**8**

Figura 1.1: Dos representaciones diferentes del mismo grafo, acompañadas de sus medidas en distintos criterios. Se marca en **negrita** el mejor valor.

Obtener representaciones óptimas para algún criterio es, en algunos casos, un problema difícil. Por ejemplo, encontrar una representación con mínimo cruzamiento de aristas[3] es un problema NP-difícil[8], y una que maximice simetrías es al menos NP-completo[17].

La meta de este proyecto es el desarrollo de una aplicación que provea buenas visualizaciones automáticas para grafos pequeños, en el orden de los 10 a 20 nodos. Casos de uso de una aplicación como esta incluyen material de estudio académico y artículos científicos en matemática, informática, biología[4] y otras ciencias.

Los casos de uso mencionados comparten por lo general la plataforma LaTeX, en particular con el uso del paquete TikZ. Este paquete gráfico es el más extendido. Sin embargo, resulta tedioso de utilizar pues los nodos se posicionan manualmente y la curva de aprendizaje es ardua. De diez herramientas analizadas para dibujo de grafos, solo dos de ellas permiten exportar a TikZ, y ambas funcionan en línea de comandos. Nuestra búsqueda no en-

[3]Este tipo de representaciones se deja de lado en favor del mostrado de simetrías. Ver capítulo 4.

[4]La interacción molecular es especialmente apropiada para su modelado con grafos. Cytoscape[24] es un programa especializado para la visualización de grafos especializado en esta área.

contró una aplicación intuitiva con alta usabilidad para dibujo automático de grafos que permita realizar esta tarea. Es entonces esa necesidad la que se busca satisfacer.

Capítulo 2

Notación y Fundamento Teórico

En este capítulo se presenta el componente matemático en el cual se apoya la aplicación desarrollada. Se asumen conocimientos fundamentales sobre grafos, teoría de grupos y geometría en el plano. En el apéndice A se encuentran definidos algunos de los conceptos requeridos.

2.1. Notación

- $G = (V, E)$ es un grafo con conjunto de nodos V y de aristas E. Por omisión, un grafo es no dirigido.
- Las aristas de un grafo se representan entre llaves pues son conjuntos no ordenados: $\{u, v\}$ es la arista entre los nodos u y v.
- El grado de un nodo v se denota $gr(v)$.
- Dados dos puntos $a, b \in \mathbb{R}^2$, sea $seg(a, b)$ al segmento de recta entre a y b.
- Llamaremos $\mathbb{S}^2$ el espacio de los segmentos de recta en el plano.

2.2. Representación de un grafo

Es importante definir formalmente qué es una representación válida de un grafo en el plano, y por tanto, qué no lo es. En el contexto de este trabajo, todas las representaciones son en dos dimensiones.

Intuitivamente, para representar un grafo con aristas rectilíneas, basta ubicar los nodos y trazar las aristas apropiadas entre ellos. Formalmente:

Definición 2.2.1. Una **representación de un grafo**[1] $G = (V, E)$ es un par de funciones $R = (R_V, R_E) : (V, E) \to (\mathbb{R}^2, \mathbb{S}^2)$ que cumple:

- **C1**: La posición de cada nodo es única (R_V es inyectiva).

$$v_1, v_2 \in V, R_V(v_1) = R_V(v_2) \to v_1 = v_2$$

- **C2**: Las aristas están bien ubicadas (su imagen es el segmento de recta entre sus extremos).

$$R_E(\{v_1, v_2\}) = seg(R_V(v_1), R_V(v_2))$$

- **C3**: Ninguna arista corta un nodo que no pertenece a ella.

$$\{v_1, v_2\} \in E, v_3 \in V, v_1 \neq v_3 \neq v_2 \to R_E(\{v_1, v_2\}) \cap R_V(v_3) = \emptyset$$

Como se dijo, la representación de los nodos R_V determina la de las aristas. En ocasiones, en este trabajo se realiza un abuso de notación, llamando representación solamente a R_V.

Definición 2.2.2. Una secuencia de nodos $S = v_1, ...v_n$ de un grafo $G = (V, E)$ es **unidimensionalmente representable** si se cumple que para todo par de nodos $v_i, v_j \in S$ tal que $\{v_i, v_j\} \in E \to |j - i| \leq 1$

Un grafo o subgrafo cuyos nodos admiten una secuencia unidimensionalmente representable tiene una representación contenida en una recta.

Ejemplo 2.2.3. Sea el grafo:

$$G = (\{v_1, v_2, v_3, v_4\}, \Big\{\{v_1, v_2\}, \{v_2, v_3\}\Big\})$$

La secuencia (v_1, v_2, v_3, v_4) es unidimensionalmente representable en G. Sin embargo, (v_1, v_2, v_4, v_3) no lo es pues la arista $\{v_2, v_3\}$ no cumple con la condición necesaria ($|4 - 2| = 2 > 1$). Ver figura 2.1.

[1] con aristas rectilíneas. En la bibliografía se define de manera más compleja, contemplando también aristas curvas.

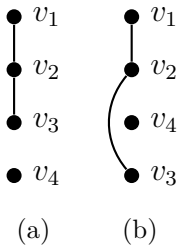


Figura 2.1: (a): la secuencia (v_1, v_2, v_3, v_4) es unidimensionalmente representable; (b) (v_1, v_2, v_4, v_3) no lo es.

2.3. Automorfismos de un grafo

Definición 2.3.1. Se le llama **automorfismo** a un isomorfismo de un grafo a sí mismo.

Definición 2.3.2. La función identidad sobre el conjunto de vértices de un grafo es un automorfismo denominado **automorfismo trivial**.

Los automorfismos se relacionan intrínsecamente con la Teoría de Grupos.

Definición 2.3.3. Un **grupo automórfico** de un grafo G es un conjunto de automorfismos de G que forma un grupo bajo la operación de composición de funciones.

El neutro de un grupo automórfico es el automorfismo trivial y por definición de grupo pertenece a todos los grupos automórficos.

Definición 2.3.4. Sea A un grupo automórfico sobre un grafo, y $v \in V$ un nodo de dicho grafo. La **órbita** de v en A es el conjunto de las imágenes de v a través de cada automorfismo del grupo A.

$$orbit_A(v) = \{\beta(v) : \beta \in A\}$$

Al conjunto de las órbitas de un grupo automórfico A se lo denota $\mathcal{O}_A$ o simplemente $\mathcal{O}$ cuando se trabaja en el contexto de un único grupo automórfico.

Propiedad 2.3.5. Dado A grupo automórfico de un grafo $G = (V, E)$, se cumple:

1. Para todo nodo $v \in V$, existe una órbita en A que lo contiene.

2. Si $u, v \in V$ y $orbit_A(u) \cap orbit_A(v) \neq \emptyset$, entonces $orbit_A(u) = orbit_A(v)$.

Demostración. 1. El automorfismo trivial I pertenece a A dado que A es un grupo automórfico. Como $I(v) = v$, entonces $v \in orbit_A(v)$, es decir, su propia órbita lo contiene.

2. Sea $O_u = orbit_A(u)$ y $O_v = orbit_A(v)$. Sea $z \in O_v$ y $\gamma \in A$ el automorfismo tal que $\gamma(v) = z$, que existe por definición de órbita.

   Como $O_u \cap O_v \neq \emptyset$, $\exists w : w \in O_u, w \in O_v$. Entonces, por la definición de órbita, existen $\alpha \in A : \alpha(u) = w$ y $\beta \in A : \beta(v) = w$. Como A es un grupo, existe entonces por propiedad de elemento simétrico $\beta^{-1} \in A$.

   Consideremos la función $\delta = \gamma \circ \beta^{-1} \circ \alpha$. $\gamma \in A$ pues la operación de composición es interna en los grupos. Por construcción es sencillo ver que $\delta(u) = \gamma(\beta^{-1}(\alpha(u))) = \gamma(\beta^{-1}(w)) = \gamma(v) = z$. Entonces, $z \in O_u$ y por tanto $O_v \subseteq O_u$. La demostración es análoga para la inclusión en el otro sentido.

□

De esta propiedad se deduce que las órbitas de un grupo automórfico particionan el conjunto de vértices de un grafo.

Definición 2.3.6. Una órbita es una **órbita trivial** si contiene un único nodo.

Un nodo en una órbita trivial será un punto fijo de todos los automorfismos del grupo.

Definición 2.3.7. Un **grupo automórfico trivial** es aquel que solamente contiene a la función identidad, que es equivalente a decir que todas sus órbitas son triviales.

Notación: Al grupo de todos los automorfismos de G se lo denota $Aut(G)$.

2.4. Automorfismos y representación simétrica

Se tiene la definición tradicional de simetría en el plano:

Definición 2.4.1. Sea $X \subseteq \mathbb{R}^2$. La función $\theta : X \to X$ es una **simetría** si a través de ella se preservan las distancias.

$$\theta \text{ simetría} \iff \forall a, b \in X, d(a,b) = d(\theta(a), \theta(b))$$

A nivel intuitivo es fácil pensar en un dibujo de grafo simétrico, pero la definición no puede aplicarse sin un pequeño ajuste. Una representación de un grafo no es un único subconjunto del plano sino un par ordenado de ellos. La representación es simétrica si la misma función θ es una simetría sobre cada uno de ellos. Dicha función puede descomponerse en θ_V y θ_E restringiendo su dominio a las representaciones de los vértices o las aristas respectivamente.

Puede verse cierta similitud entre automorfismos y simetrías. Un automorfismo lleva un nodo a otro preservando adyacencia, mientras que θ_V lleva la *representación* de un nodo a la de otro preservando adyacencia. En efecto, existe una relación entre ambos conceptos. La siguiente propiedad lo formaliza:

Propiedad 2.4.2. Si una representación R de un grafo G admite una simetría θ, entonces $\beta = R_V^{-1} \circ \theta_V \circ R_V$ es un automorfismo sobre G. Se dice que R **muestra** β.

Ejemplo 2.4.3. Sea G el grafo y R la representación de la figura 2.2(a). R tiene múltiples simetrías. Sea θ una de ellas: la rotación con centro en el origen y ángulo $\pi/2$ en sentido horario (figura 2.2(b)).

Veamos el automorfismo inducido por la propiedad 2.4.2, en particular la imagen de v_1.

$$\beta(v_1) = R_V^{-1}(\underbrace{\theta_V(\overbrace{R_V(v_1)}^{(0,1)})}_{(1,0)}) = v_2$$

.

Con ese procedimiento se encuentra el siguiente automorfismo.

v	v_1	v_2	v_3	v_4	v_5
$\beta(v)$	v_2	v_3	v_4	v_1	v_5

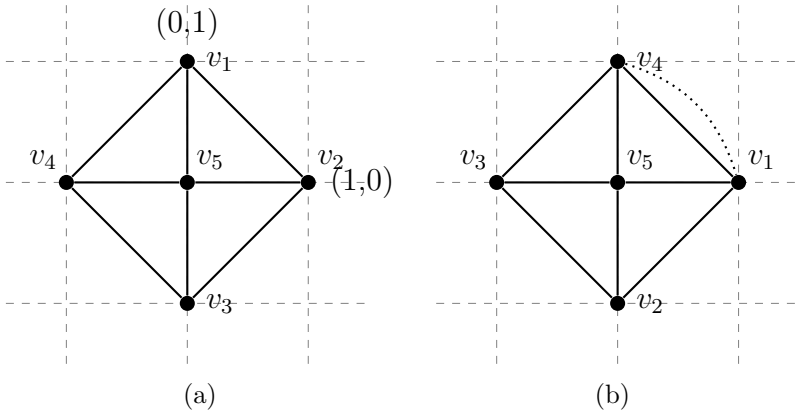


Figura 2.2: (a): representación de un grafo con múltiples simetrías rotacionales; (b): rotación de ángulo $\pi/2$ en sentido horario aplicada al mismo. La línea punteada ilustra el movimiento del nodo v_1.

Una interrogante que se plantea es: ¿para cualquier automorfismo de un grafo, existe una representación que lo muestra?. La respuesta es negativa, véase el siguiente contraejemplo:

Ejemplo 2.4.4. Sea el grafo de la figura 2.3 y el automorfismo β según el Cuadro 2.1.

v	v_1	v_2	v_3	v_4	v_5
$\beta(v)$	v_2	v_3	v_1	v_5	v_4

Cuadro 2.1: Automorfismo no geométrico.

Si existiera una simetría que mostrase este automorfismo, $v_1 \to v_2 \to v_3 \to v_1$ exige una rotación con ángulo $2\pi/3$. Sin embargo, v_4 y v_5 no pueden intercambiar posiciones a través de una rotación con ese ángulo. Entonces, no existe una simetría que muestre β.

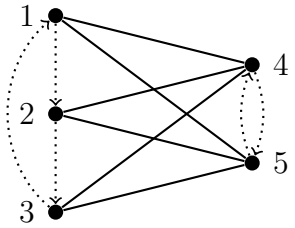


Figura 2.3: Grafo con un automorfismo no geométrico.

Dado que existen automorfismos que no pueden ser mostrados por una simetría, es apropiado caracterizar a los que sí pueden.

Definición 2.4.5. Un automorfismo es **geométrico** si existe una simetría que lo muestra.

Definición 2.4.6. Un grupo automórfico es **geométrico** si existe una simetría que muestra todos sus automorfismos.

2.5. Formalización del problema

La definición 2.4.6 define que para un grupo automórfico geométrico existe una simetría que lo muestra, pero no provee información sobre cómo hallarla. Se plantea entonces el problema:

Problema 1. Dado un grupo automórfico geométrico de un grafo, encontrar una representación simétrica que lo muestre.

Eades y Hong[13] resolvieron este este problema. Más aún, establecen condiciones necesarias y suficientes para que un grupo automórfico sea geométrico, y cuando lo es, brindan un algoritmo para su representación[2]. Sin embargo, para el uso en una aplicación que dibuje grafos automáticamente, es la obtención de un grupo automórfico geométrico la que es problemática. El problema es NP-completo[17], lo cual ha llevado a numerosos trabajos basados en heurísticas[4, 7]. La mayoría de estas utilizan métodos dirigidos por fuerzas. En el capítulo 4 se estudia la eficacia de la implementación de una de ellas, mientras que en el capítulo 3 se estudian con más detalle estos métodos aplicados a la representación no necesariamente simétrica de grafos.

[2]sin tomar en cuenta criterios estéticos.

Si bien el problema 1 fue resuelto por Eades y Hong, sin un algoritmo eficiente para encontrar grupos automórficos geométricos, carece de aplicación práctica. Un primer paso es la relajación del problema para evitar la necesidad que los automorfismos sean geométricos, y con ella, la restricción de mostrar todos los automorfismos del grupo, pues no todos podrán ser mostrados.

Problema 2. Dado un grupo automórfico no trivial de un grafo, encontrar una representación simétrica que muestre al menos un automorfismo no trivial perteneciente al grupo.

Sin embargo, resolver este problema tampoco tiene aplicación práctica. No existen algoritmos eficientes conocidos para la búsqueda explícita de $Aut(G)$. Encontrar todos los automorfismos es obviamente al menos tan difícil como saber si un grafo admite o no automorfismos no triviales. Ese problema a su vez es al menos tan difícil como el de saber si dos grafos son isomórficos[18]. El problema de isomorfismo de grafos no tiene una solución conocida en tiempo polinomial, es decir, pertenece a la clase NP, aunque no se cree que el mismo sea NP-completo, de hecho, existen algoritmos en tiempo cuasi-polinomial para resolverlo[1].

Se toma entonces otro enfoque. El problema de encontrar las órbitas de $Aut(G)$ es polinomialmiente equivalente al problema de isomorfismo de grafos[18, 23]. En el capítulo 4 se discuten herramientas existentes que encuentran las órbitas de un grafo. Entonces:

Problema 3. Dadas las órbitas de $Aut(G)$, encontrar una representación simétrica que muestre al menos un automorfismo no trivial perteneciente al grupo.

Donde sea posible, se intenta minimizar el número de nodos fijos de la simetría encontrada.

Además, se busca un criterio estético visualmente agradable, que es la motivación para algunas de las heurísticas utilizadas.

2.6. Propiedades de Simetrías y Órbitas

Dada una simetría de un grafo G, pueden deducirse ciertas propiedades sobre su conjunto de órbitas $\mathcal{O}$. Resulta útil hacer explícitas algunas de estas propiedades. En particular, sus contrarrecíprocos ayudarán a descartar posibles representaciones en el capítulo 4.

Rotaciones

Propiedad 2.6.1. Una rotación no trivial de G cumple que $\mathcal{O}$ tiene a lo sumo una órbita trivial.

La propiedad es evidente pues las rotaciones no triviales en el plano fijan a lo sumo un punto.

No violar el contrarrecíproco de esta propiedad *no* garantiza que exista una representación como rotación. Se muestra un contraejemplo en la figura 2.4.

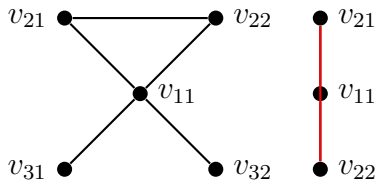


Figura 2.4: (a): Este grafo (representado como reflexión) tiene órbitas $\{\{v_{11}\}, \{v_{21}, v_{22}\}, \{v_{31}, v_{32}\}\}$; (b): mismo grafo que (a), intento de representarlo como rotación, al ubicar la segunda órbita en una circunferencia con centro en el nodo fijo, se genera una representación inválida que no puede ser arreglada, pues la arista v_{21} a v_{22} siempre cortará el nodo central. Este grafo no puede representarse mostrando simetría rotacional.

Reflexiones

El automorfismo mostrado por una reflexión siempre cumple:

$$\beta(v_1) = v_2 \rightarrow \beta(v_2) = v_1$$

Entonces, dado que los nodos de cada órbita se "agrupan" de a dos:

Propiedad 2.6.2. A través de una reflexión, Cada órbita de cardinal impar contiene al menos un nodo fijo.

También, los nodos fijos de una reflexión se ubican sobre una línea recta, el eje de simetría:

Propiedad 2.6.3. Los nodos fijos de una reflexión admiten una secuencia unidimensionalmente representable.

Capítulo 3

Análisis de la Competencia

En este capítulo se reseñan las estrategias empleadas por las herramientas comúnmente utilizadas para el dibujo de grafos. Se analizaron múltiples herramientas, como puede verse en el cuadro 3.1 junto con las principales estrategias que utiliza cada una de ellas.

Herramienta	**Estrategias**
WolframAlpha / Mathematica	tabulación, force-directed[1]
NetworkX	force-directed, circular, aleatorio
Gephi	force-directed, circular, aleatorio
Graphviz	force-directed
MathWorks	force-directed, circular, aleatorio
SigmaJS	force-directed
ArborJS	force-directed
GraphTea	force-directed, circular
Cytoscape	force-directed
SageMath	force-directed, circular

Cuadro 3.1: Herramientas analizadas.

A continuación se presentan reseñas de las estrategias relevadas.

[1]No divulga las estrategias que utiliza. Las pruebas concluyeron que aplica métodos force-directed a menos que tenga una representación previamente tabulada del grafo.

3.1. Algoritmos Triviales

Por completitud, se incluyen las dos opciones más simples para dibujo de grafos, si bien en general no dan buenos resultados.

1. **Posicionamiento Aleatorio**. Los nodos son posicionados de manera aleatoria.

2. **Posicionamiento Circular**. Este posicionamiento, donde todos los nodos se ubican equiespaciados en una circuferencia, siempre es una representación válida para cualquier grafo[2].

3.2. Algoritmos Force-Directed

Todas las soluciones analizadas utilizan distintas implementaciones de métodos basados en fuerzas, más conocidos como *force-directed*. Esta técnica propuesta originalmente en 1963 por Tutte[27] utiliza simulaciones de fuerzas de atracción o repulsión entre los nodos que mediante simulaciones convergen a un estado de equilibrio. Debido a que la complejidad de estos algoritmos es independiente de la estructura interna del grafo, no consideran simetrías, planaridad u otras propiedades del mismo, que en muchos casos son computacionalmente difíciles de estudiar. Esto es positivo tanto para la simplicidad de implementación como para la escalabilidad, que es buena para grafos de hasta 500 nodos.

Las representaciones obtenidas en ocasiones son simétricas, mas no existe seguridad de que se prefieran representaciones simétricas frente a otras. Eades y Lin[5] han probado que ciertos algoritmos force-directed existentes pueden siempre mostrar las simetrías de un grupo automórfico geométrico. Sin embargo, no determinan con qué probabilidad efectivamente lo hacen.

En comparación con la aplicación implementada en este proyecto, en múltiples casos, los algoritmos force-directed dan resultados de peor calidad. Se ilustran tres ejemplos en las figuras 3.1, 3.2 y 3.3. En el primer caso, el método force-directed no muestra la simetría, y en el segundo, no muestra la simetría aproximada. En el último, el problema es mayor, pues si bien la representación force-directed es simétrica, no es válida.

[2]Obviamente, no puede haber dos nodos en la misma posición. Al estar sobre una circunferencia, tampoco puede haber tres nodos sobre la misma recta.

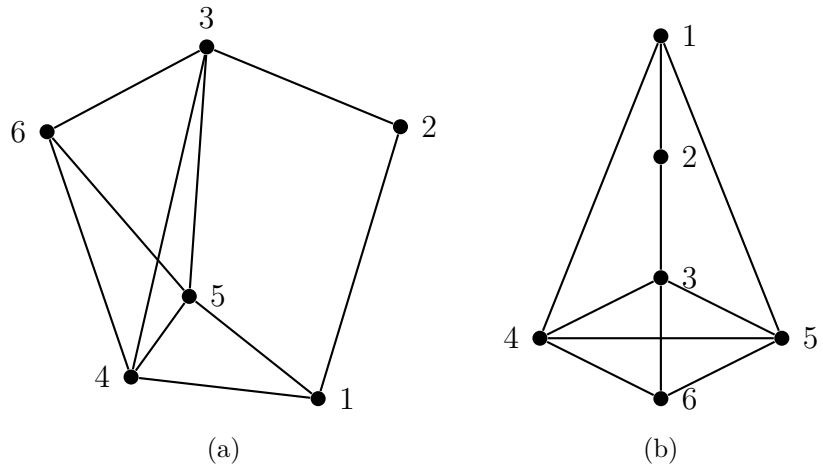


Figura 3.1: (a) muestra la representación force-directed de un grafo de manera asimétrica; (b) muestra el mismo grafo representado con la aplicación desarrollada, con una simetría axial.

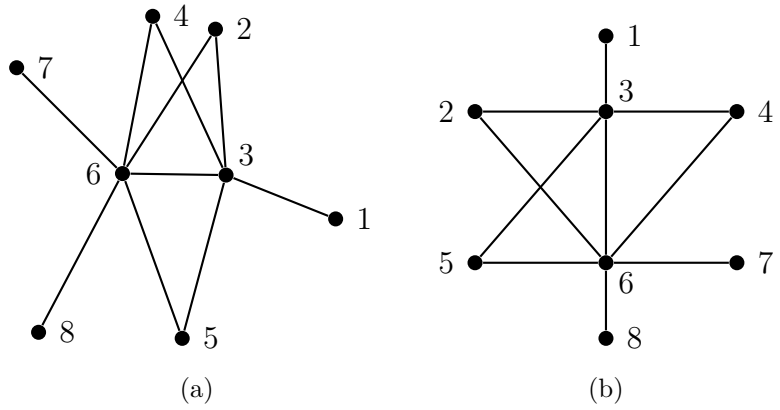


Figura 3.2: (a) muestra la representación force-directed de un grafo de manera asimétrica; (b) muestra el mismo grafo representado con la aplicación desarrollada, con una simetría axial aproximada ignorando la arista $\{3, 5\}$.

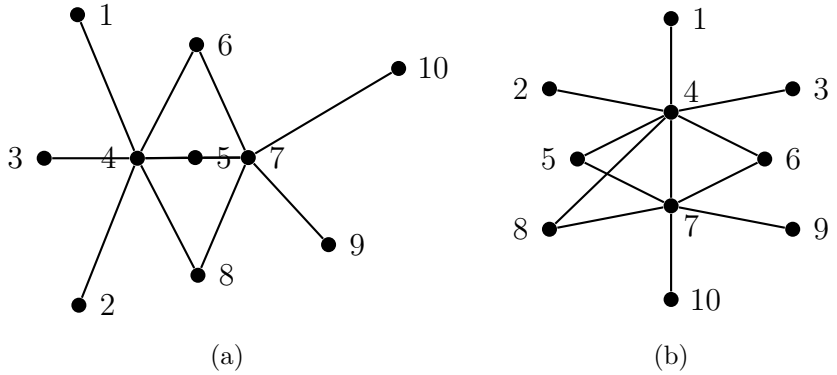


Figura 3.3: (a) muestra la representación force-directed de un grafo. La misma presenta problemas de validez, el nodo 5 se encuentra en una posición inválida pues corta la arista $\{4, 7\}$; (b) muestra el mismo grafo representado en la aplicación desarrollada, con una simetría axial aproximada ignorando la arista $\{4, 8\}$.

3.3. Grafos Tabulados

El relevamiento realizado en WolframAlpha (`http://wolframalpha.com`), que utiliza el mismo motor que el programa profesional Mathematica, indica que para grafos pequeños, utiliza una base de datos de representaciones de grafos. En esos casos, WolframAlpha elige una representación canónica previamente establecida para cada conjunto de grafos isomórficos.

La existencia de nombres basados en su estructura revela que las representaciones almacenadas son ingresadas o al menos supervisadas por seres humanos. Se muestra un ejemplo en la figura 3.4.

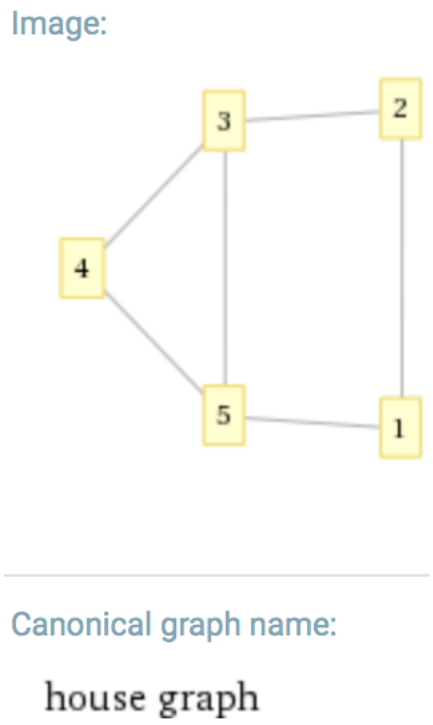


Figura 3.4: Representación de grafo mostrada por Wolfram Alpha con una leyenda que lo identifica como el "house graph" (grafo casa).

También hay indicios de que la representación en sí, si bien puede estar almacenada, se generó con métodos force-directed al igual que en grafos más grandes. En la figura 3.4, las aristas $\{2, 3\}$ y $\{1, 5\}$ no son paralelas sino que 3 y 5 se encuentran más cercanos que 1 y 2, tal y como sucede en un algoritmo force-directed, donde la fuerza atractiva del nodo 4 contrae esa distancia.

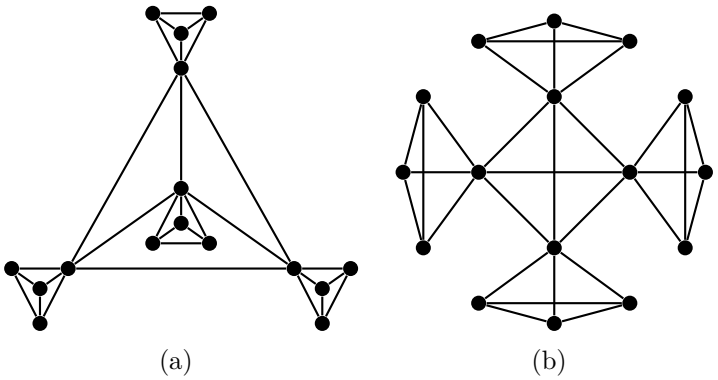
Wolfram muestra nombres canónicos para grafos de hasta siete nodos. La pregunta inmediata es si la tabulación de grafos con supervisión humana es factible para un número mayor. Existen solamente 1044 grafos no isomorfos con siete nodos[20], un número de registros fácilmente almacenable. Sin embargo, el número crece rápidamente. Para $N = 14$, si cada entrada de la base de datos ocupase 1 byte (mucho menos de lo que realmente ocuparía), se necesitarían más de 29 petabytes para almacenar la totalidad de los grafos no isomorfos. Además del problema de almacenamiento, la generación, supervisión y ajuste manual de las representaciones por supuesto tampoco son factibles con tal volumen de datos.

En base a la investigación, se concluye que la tabulación de representaciones de grafos no es una solución satisfactoria por problemas de escalabilidad.

3.4. Minimizar Cruzamientos de Aristas

La minimización de cruzamientos de aristas en una representación de un grafo es un problema NP-difícil[8]. En consecuencia, se han hecho estudios basados en heurísticas para la resolución de este problema, tanto en grafos cualesquiera como pertenecientes a familias específicas[11, 3].

Si bien la minimización de cruzamientos de aristas es un área interesante del problema de representación de grafos, la figura 3.5 muestra cómo al minimizar cruzamientos se pierde en otros criterios de visualización. En este proyecto no se utilizan criterios de minimización de cruzamientos, eligiendo por sobre ellos la visualización de simetrías.



(a) (b)

Figura 3.5: (a) muestra la representación plana de un grafo con una simetría axial; (b) muestra el mismo grafo representado en la aplicación desarrollada, con cuatro simetrías rotacionales, cuatro simetrías axiales y cinco cruzamientos de aristas.

3.5. Algoritmos basados en simetrías

Las herramientas relacionadas explícitamente a simetrías que se encontraban disponibles en la red[3] no fueron numerosas. Todas ellas están enfocadas únicamente a la detección de simetrías y no a la representación de las mismas. Como tales, fueron analizadas no como competidores sino como posibles dependencias de este proyecto. Debido a ello, se detallan en el capítulo 4.

[3]antes del despliegue de la aplicación desarrollada en este proyecto

Capítulo 4

Descripción del Algoritmo

En este capítulo se describe el algoritmo implementado para representar grafos de manera simétrica. En primera instancia se tratan las herramientas usadas o consideradas para la búsqueda de simetrías. Luego se elabora sobre cómo se usa la salida de la dependencia utilizada para encontrar representaciones, que es el componente principal de este trabajo. Finalmente, se justifican las consideraciones estéticas que se utilizaron.

4.1. Dependencias Analizadas

Como se mencionó, para la búsqueda de grupos automórficos se recurrió a dependencias externas. En esta sección se detallan implementaciones de algoritmos de búsqueda de simetrías que fueron consideradas o utilizadas como dependencias de este proyecto.

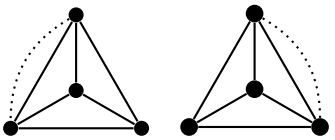
4.1.1. PIGALE Library

La biblioteca Public Implementation of a Graph Algorithm Library and Editor (PIGALE[12]) incluye múltiples algoritmos sobre grafos. Entre ellos se encuentra la implementación de una heurística[7] para detección de simetrías.

Las pruebas sobre la herramienta ofrecen resultados pobres.

- Carece de la capacidad de encontrar rotaciones incluso en grafos minimales ($N = 3$).
- Para la detección de reflexiones, requiere la existencia de al menos dos puntos fijos.

- Incluso considerando únicamente reflexiones con puntos fijos, no encuentra todas las posibles. Observado en $N = 4$. Ver figura 4.1. Existen múltiples simetrías más, tanto fijando otros dos vértices cualesquiera, uno o ninguno.
- Presenta problemas de robustez. Por ejemplo, el programa se cierra inesperadamente al ingresarse el grafo anillo de tres elementos.



(a) (b)

Figura 4.1: Los únicos automorfismos de K_4 detectados por PIGALE, fijando dos vértices.

4.1.2. Nauty

Nauty[19] es un programa con interfaz de línea de comandos para determinar la existencia de automorfismos. No provee algoritmos de dibujo, pero sí la obtención de las órbitas de $Aut(G)$ con una eficiencia más que apropiada para grafos de las magnitudes de interés[1]. El uso de Nauty fue considerado en primera instancia aceptable para este proyecto.

4.1.3. Find Almost Symmetry

Una investigación más profunda descubrió esta alternativa que reemplazó a Nauty para la búsqueda de órbitas. Se trata de un trabajo[15] publicado en el año 2017 que presenta un algoritmo *branch and bound* para el siguiente problema: Dado un grafo G y un entero b, encontrar un subgrafo de G, donde el mismo se obtiene removiendo no más de b aristas de G y minimiza el número de órbitas.

[1] siendo el algoritmo más rápido en detectar automorfismos, entre cinco algoritmos estudiados sobre grafos aleatorios[6]

Su implementación se encuentra disponible en línea bajo la MIT License. Find Almost Symmetry utiliza Nauty como dependencia, con una estrategia de ramificación y acotado especial para remover aristas, basada en la estructura interna del grafo. En el artículo se muestra una amplia mejoría de la estrategia utilizada frente a una de ramificación aleatoria. Se incluye un resumen del algoritmo en el apéndice C.

La aplicación implementada utiliza como dependencia principal el algoritmo Find Almost Symmetry, permitiendo así la representación de grafos con simetrías aproximadas, tanto removiendo aristas, como induciendo aristas ficticias (se ejecuta también el algoritmo de búsqueda de simetrías con el grafo inverso).

Las entradas y salidas del algoritmo son las siguientes:

Algorithm 1 FindAlmostSymmetry

1: **procedure** FindAlmostSymmetry(G, b)
2: ...
return $\mathcal{O}$ ▷ Órbitas del grupo automórfico para el subgrafo encontrado
3: **end procedure**

4.2. Algoritmo Implementado

En esta sección se explica el algoritmo de representación automática de grafos que se implementó en el proyecto. Se utiliza un enfoque constructivo *bottom-up*. Cada algoritmo podrá ser utilizado por los que le siguen.

4.2.1. Procedimientos auxiliares

A continuación se describen algunos algoritmos auxiliares.

Algorithm 2 ExtraerNodosFijos

Dada una colección de órbitas: en modo "rotación", elimina de la colección aquellas de cardinal 1 y retorna los nodos extraídos. En modo "reflexión", además extrae un nodo de cada órbita de cardinal impar.

1: **procedure** ExtraerNodosFijos($\mathcal{O}, modo$)

2: ...
return F
3: **end procedure**

Algorithm 3 ValidarRepresentación

Dado un grafo y una representación parcial, averigua si dicha representación parcial es factible[2].

1: **procedure** ValidarRepresentación(G, R)
2: ... ▷ Verifica las condiciones de la definición de representación iterando sobre las aristas y los nodos.
3: **end procedure**

4.2.2. Ordenamiento heurístico de órbitas

Dada la naturaleza secuencial del posicionamiento que realizan los algoritmos que siguen, fue necesario el ordenamiento de las órbitas. Heurísticamente (por razones estéticas pero también de rendimiento), se optó por un esquema núcleo-periferia, donde los nodos con más conexiones se colocan primero y por tanto más cerca del núcleo, mientras que los nodos menos conectados se ubican en función de los primeros. Para tal fin:

Definición 4.2.1. La densidad de aristas $d_E(O)$ de una órbita O es la suma de los grados de sus nodos sobre la cantidad de nodos de la misma.

$$d_E(O) = \frac{\sum_{v \in O} gr(v)}{\#O}$$

Primero, se separan las órbitas en dos niveles: aquellas que se conectan con los nodos fijos, y aquellas que no. Luego se ordenan las órbitas de cada nivel por densidad decreciente de aristas. En las rotaciones es muy claro el efecto que tiene este esquema núcleo-periferia. Véase la figura 4.2.

[2] es decir, si la porción definida no viola la definición de representación de grafo.

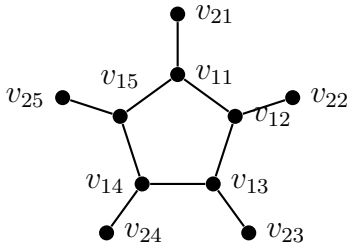


Figura 4.2: Un grafo con dos órbitas $O_1 = (v_{11}, ..., v_{15})$ y $O_2 = (v_{21}, ..., v_{25})$. O_1 se posiciona más adentro pues tiene mayor densidad de aristas.

Algorithm 4 OrdenarÓrbitas

Ordena un conjunto de órbitas $\mathcal{O}$ para su posicionamiento secuencial.

1: **procedure** OrdenarÓrbitas($\mathcal{O}$)
2: $(\mathcal{O}_{fix-adjacent}, \mathcal{O}_{fix-nonadjacent}) \leftarrow$ Particionar $\mathcal{O}$ según si cada órbita es adyacente o no a algún nodo fijo
3: ordenar($\mathcal{O}_{fix-adjacent}$, $O \rightarrow d_E(O)$, descendiente) ▷ ordenar recibe además del conjunto a ordenar, la función, y el sentido de ordenamiento.
4: ordenar($\mathcal{O}_{fix-nonadjacent}$, $O \rightarrow d_E(O)$, descendiente)
5: $\mathcal{O} \leftarrow \mathcal{O}_{fix-adjacent} + \mathcal{O}_{fix-nonadjacent}$
6: **end procedure**

El esquema núcleo-periferia utilizado resulta agradable estéticamente, pero su elección se sustenta en razones más concretas. La fortaleza de la heurística radica en que previene la aparición de problemas en algunos casos:

- La densidad decreciente de aristas se justifica pues las aristas más interiores son de menor longtitud. Una mayor proporción de aristas más cortas disminuyen la longitud promedio. Aristas más cortas y más interiores tendrán menos probabilidad de tener nodos internos que las crucen.

- Con el mismo razonamiento, el particionamiento según adyacencia a los nodos fijos disminuye el largo de las aristas entre ellos y la órbita. La adyacencia a los nodos fijos tiene precedencia sobre la densidad de aristas, pues existen situaciones como la de la figura 4.3.

- Además de ayudar a prevenir los problemas, cuando estos ocurren, es probable que ocurran antes. Las órbitas más problemáticas son las que aportan mayor cantidad de aristas, pues tienen mayor probabilidad de inducir una superposición de una arista con un nodo. Entonces, si las órbitas más problemáticas se colocan primero, es más probable que las fallas ocurran en etapas tempranas. Computacionalmente, esto es positivo pues detectar infactibilidad consumirá menos tiempo de ejecución.

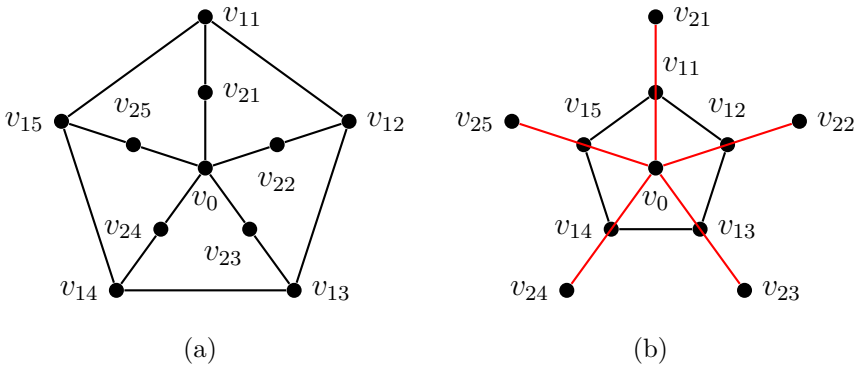


Figura 4.3: Adición de un nodo fijo v_0 conectado con los nodos de O_2 al grafo de la figura 4.2. En (a), como en la implementación, se invierte el orden anterior. O_2 se posiciona más adentro pues se conecta con el nodo fijo. En (b), si no se realizase la división en niveles, O_1 permanece interior pues tiene mayor densidad de aristas, y se produce un problema (resoluble) de validez de representación.

4.2.3. Reordenamiento heurístico de nodos

De manera similar a las órbitas en $\mathcal{O}$, las representaciones obtenidas son diferentes si se varía el ordenamiento de los nodos dentro de cada órbita.
Los nodos de cada órbita se ubicarán equiespaciados en una circunferencia, tanto en rotaciones como en reflexiones. Para una órbita de cardinal n hay n posiciones posibles (porque las reflexiones requieren que la órbita sea simétrica según el eje de vertical de simetría, y en las rotaciones se desea un nodo en el cenit). La heurística aplicada para elegir cuál nodo va en cada posición de la circunferencia es la de minimizar las longitudes de las aristas. Ver figura 4.4.

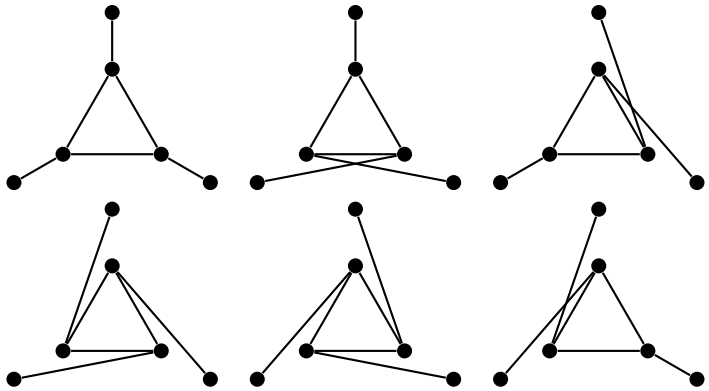


Figura 4.4: De las 3! opciones posibles para ordenar la órbita exterior, se elige la de arriba a la izquierda pues minimiza las longitudes de las aristas.

Las únicas tres posibilidades son que los nodos de una órbita no se conecten entre sí, se conecten todos entre sí, o algunos estén conectados pero no todos.
En el primer caso, como los nodos de la órbita no están conectados entre sí, el largo de aristas que aporta un nodo es independiente de los largos de aristas que aportan los demás. Entonces, el problema de minimización de largos de aristas se reduce al problema de asignación, resoluble en orden n^3[16].
El segundo caso parece más complejo a simple vista, pero es igualmente simple, pues si todos están conectados entre sí, los largos de aristas entre nodos de la órbita son constantes sin depender de cómo se ordenen. Entonces, también son independientes.
En el caso de conexión parcial, sin embargo, el problema tiene complejidad exponencial, pues existen dependencias entre los nodos: es beneficioso que algunos estén cerca entre sí, así como de los pertenecientes a otras órbitas.

Para este problema no se conoce una resolución eficiente. Por esto, se requiere un algoritmo exponencial, y es lo que se utiliza en órbitas pequeñas.

- Para órbitas de hasta siete nodos, el algoritmo exponencial tiene buen rendimiento, por lo que es el utilizado: dadas todas las permutaciones, se calcula la menor suma de longitudes.

- Para órbitas más grandes, se utiliza un algoritmo *greedy*, donde cada nodo es colocado secuencialmente en la posición que minimiza las distancias. Para este algoritmo se tiene en cuenta, si es posible, el posicionar en cada paso nodos conectados con los que ya fueron dibujados de la órbita, para así evitar que otros nodos le saquen el lugar a éstos. Como es usual en los algoritmos ávidos, no se puede garantizar la optimalidad del mismo, pero en la práctica los resultados han sido satisfactorios.

Se presenta la firma y descripción del algoritmo, sin entrar en detalle. La implementación es como se describió anteriormente.

Algorithm 5 ReordenarNodos

Dado un grafo, una representación parcial y una órbita con representación, permuta las coordenadas de los nodos de dicha órbita en la representación, para minimizar la longitud de las aristas representadas.

1: **procedure** ReordenarNodos(G, R, O)
2: ... ▷ Implementación exponencial hasta siete nodos, de lo contrario algoritmo ávido.
3: **end procedure**

4.2.4. Representar Rotación

Las simetrías rotacionales se representan con este algoritmo. Se ordenan las órbitas según lo descrito en 4.2.2 y se posicionan secuencialmente en círculos concéntricos desde adentro hacia afuera, procurando la validez de la representación. También existen casos donde una representación de rotación no es factible (como se vio en 2.6). En tal caso, el algoritmo falla.
Al establecer que las órbitas no triviales se ubican en círculos concéntricos, se exige una nueva hipótesis: el máximo común divisor de los cardinales de

las órbitas no triviales debe ser mayor a 1. Intuitivamente, si una órbita es de cardinal 4, la rotación deberá ser de ángulo $\frac{n\pi}{2}$, pero si otra es de cardinal 3, la rotación también deberá tener ángulo $\frac{2n\pi}{3}$. En tal caso, una rotación no trivial no puede existir. Esta hipótesis es demasiado estricta. Existen casos con máximo común divisor 1 que sí pueden representarse como rotación, separando una órbita en dos circunferencias, pero se utiliza la hipótesis mencionada por razones de implementación. Las otras implementaciones consideradas requieren disponer del grupo automórfico explícitamente en lugar de solo el conjunto de sus órbitas.

Algorithm 6 RepresentarRotación

Dado un grafo y el conjunto de órbitas de su grupo automórfico maximal, retorna una representación rotacionalmente simétrica de dicho grafo. Si la representación no es encontrada, retorna FALSE.

1: **procedure** RepresentarRotación($G, \mathcal{O}$)
2: $F \leftarrow$ ExtraerNodosFijos($\mathcal{O}$, *rotación*)
3: **if** $\#F > 1$ **or** $MCD(\{\#O : O \in \mathcal{O}\}) = 1$ **then**
return FALSE ▷ Las rotaciones tienen a lo sumo un punto fijo. Los cardinales de sus órbitas no triviales deben tener maximo común divisor mayor a 1.
4: **end if**
5: OrdenarÓrbitas($\mathcal{O}, F$)
6: **for each** $O \in \mathcal{O}$ **do**
7: Posicionar los nodos de O equiespaciados en $C = \mathcal{C}(\vec{0}, r)$, donde r es 1+ el máximo radio ya utilizado, tal que uno de los nodos queda sobre el eje $0\vec{y}$ positivo.
8: ReordenarNodos(G, R, O)
9: **if not** ValidarRepresentación(G, R) **then**
10: Rotar O con ángulo $\frac{\pi}{\#O}$ ▷ Ver figura 4.5
11: **end if**
12: **if not** ValidarRepresentación(G, R) **then**
13: Incrementar el radio de C en 1
14: **end if**
15: **end for**
16: Si la última órbita fue rotada, rotar todo el grafo en el sentido opuesto para asegurar un nodo de la última órbita hacia arriba. ▷ Ver figura 4.6
17: **if** ValidarRepresentación(G, R) **then**

return R

18: **else**

return FALSE ▷ Algunos errores, como cruces de aristas con el nodo fijo, no pueden repararse.

19: **end if**

20: **end procedure**

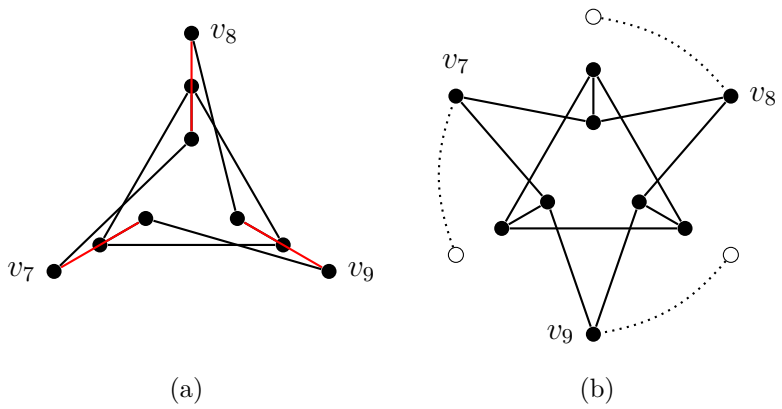


Figura 4.5: (a): El posicionamiento de la órbita $\{v_7, v_8, v_9\}$ no es válido pues existen aristas que cruzan nodos; (b) rotación de la órbita, arreglando el problema.

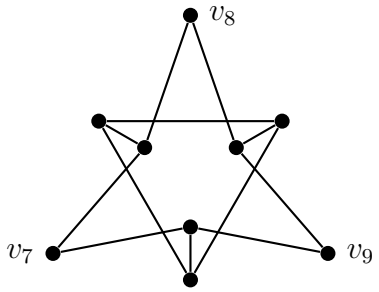


Figura 4.6: El grafo de la figura 4.5 es rotado en su totalidad para tener un nodo hacia arriba en la órbita más exterior (únicamente por criterio estético).

4.2.5. Mejor Desplazamiento Vertical

Este procedimiento es auxiliar a las reflexiones y hace lo que se explica a continuación.

Algorithm 7 MejorDesplazamientoVertical

Dado un grafo, una representación parcial y una órbita con representación sobre una circunferencia, encuentra el desplazamiento vertical para la órbita tal que se minimizan las longitudes de las aristas.

```
1: procedure MejorDesplazamientoVertical(G, R, O)
2:     ...
       return d
3: end procedure
```

Se muestra un ejemplo en la figura 4.7(d) como parte del ejemplo general de representación de reflexión.

4.2.6. Representar Reflexión

Las reflexiones son más complejas, pues existen más parámetros de posicionamiento: las circunferencias que contendrán a cada órbita ahora pueden variar no solo su radio, sino también desplazar su centro sobre el eje de simetría. Primero se posicionan los nodos fijos, y luego, de manera similar a las rotaciones, se posicionan las órbitas secuencialmente procurando la validez de la representación. Se han incluido figuras explicativas en varios pasos del algoritmo para ejemplificar la construcción de un caso representativo (ver figura 4.7).

Algorithm 8 RepresentarReflexión

Retorna la representación de un grafo como una reflexión sobre el eje vertical, dado el grafo y su conjunto de órbitas. Si no es posible, la representación falla, retornando FALSE.

1: **procedure** RepresentarReflexión($G, \mathcal{O}$))
2: $F \leftarrow$ ExtraerNodosFijos($\mathcal{O}, reflexión$)
3: Ordenar F como una secuencia unidimensionalmente representable
4: **if** la secuencia no existe **then**
return FALSE
5: **end if**
6: **for each** $v_f \in F$ **do**
7: $R(v_f) \leftarrow (0, 2f)$ ▷ Ver figura 4.7(c)
8: **end for**
9: **for each** $O_i \in \mathcal{O}$ **do**
10: $d \leftarrow 0, r_{mul} \leftarrow 1$
11: **if** $\#O_i > 2$ **then**
12: $r_{base} \leftarrow 1/cos(\pi \#O_i)$
13: **else**
14: $r_{base} \leftarrow 2$
15: **end if**
16: Posicionar los nodos de O equiespaciados en $\mathcal{C}((0, d), r_{base} * r_{mul})$, tal que se preserva la simetría axial sobre el eje $0\vec{y}$. ▷ Ver figura 4.7(d)
17: ReordenarNodos(G, R, O).
18: $d =$ MejorDesplazamientoVertical(G, R, O) ▷ Ver figura 4.7(d)
19: $r_{mul} \leftarrow 1+$ el máximo r_{mul} ya utilizado para el desplazamiento d
20: **while not** ValidarRepresentación(G, R) **do** ▷ Mientras

falle, se agranda el radio hasta una constante L y luego se reinicia el multiplicador cambiando el desplazamiento a otro cercano al óptimo.

if $r \leq L$ **then**
    $r_{mul} \leftarrow r_{mul} + 1$
end if
if $r > L$ **then**
    de manera alternada: $d \leftarrow 1+$ el mayor d ya utilizado para esta órbita, $d \leftarrow$ el menor d ya utilizado para esta órbita -1.
    $r_{mul} \leftarrow 1+$ el máximo r_{mul} ya utilizado para el nuevo desplazamiento d
end if
end while
end for
end procedure

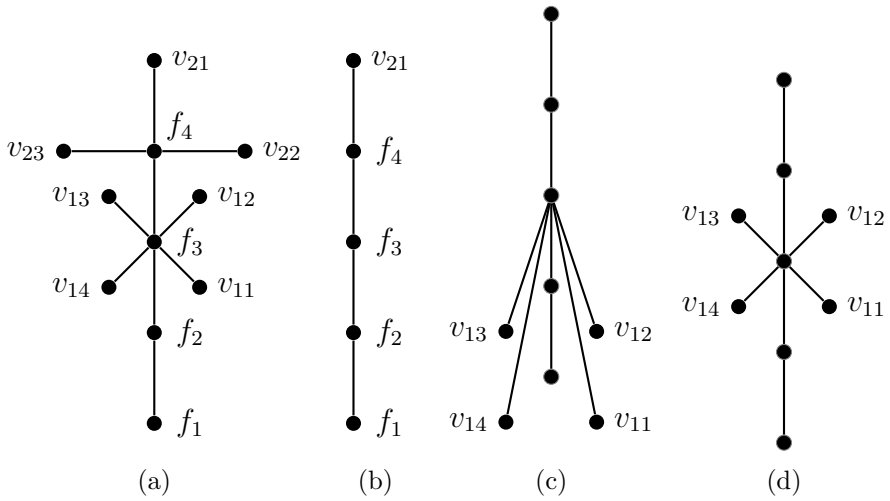


Figura 4.7: Ejemplo de reflexión: (a) representación final; (b) nodos fijos sobre el eje vertical en secuencia unidimensionalmente representable, v_{21} ahora es fijo pues su órbita era de cardinal impar; (c) órbita posicionada con $d = 0$; (d) órbita con d ajustado para minimizar largo de aristas.

4.2.7. Posicionar Etiquetas

Formalmente, una representación de un grafo no tiene etiquetas. Sin embargo, es usual que en el dibujo de un grafo los nodos se encuentren identificados por una cadena de caracteres. Con el objetivo de ubicar cada etiqueta en una posición que no se superponga con aristas incidentes al nodo, se calculan los ángulos de incidencia, tratando de ubicar la etiqueta en una dirección que cruce con la menor cantidad de aristas[3].
Se consideran como candidatos para la etiqueta a los cuadrantes con amplitud $\pi/2$ centrados en los puntos cardinales (N, S, E, W) y sus direcciones intermedias (NE, NW, SE, SW).

Algorithm 9 PosicionarEtiquetas

Dado un grafo y su representación, devuelve una función de posición de etiquetas (a cada nodo, le asigna un cuadrante entre N, S, E, W, NE, NW, SE o SW) tal que el cuadrante asignado a cada nodo tiene cantidad minimal de aristas incidentes para ese nodo.

1: **procedure** PosicionarEtiquetas(G, R)
2: ...
return T
3: **end procedure**

4.2.8. Obtener Coordenadas

La generación de una representación para un grafo conexo se encapsula en el algoritmo ObtenerCoordenadas.

Algorithm 10 ObtenerCoordenadas

Entrada:

- un grafo conexo $G = (V, E)$ a representar.
- un entero b_0, número inicial de aristas a ignorar o introducir (*budget* inicial).

[3]El posicionamiento inteligente de etiquetas solo se visualiza al exportar a LaTeX/TikZ, pues la biblioteca Sigma.js utilizada para la representación en el sitio no soporta tal opción. Queda como trabajo futuro el reemplazo de la biblioteca por una que lo soporte.

Salida:

- una representación R de G.
- Una función de posicionamiento de etiquetas para los nodos del grafo.

1: **procedure** ObtenerCoordenadas($G = (V, E), b_0$)
2: $b \leftarrow b_0$
3: $R \leftarrow$
4: **while not** R **do**
5: $\mathcal{O} \leftarrow$ FindAlmostSymmetry(G, b)
6: **if** $\#\mathcal{O} < \#V$ **then** ▷ Si no todas las órbitas son triviales
7: $R \leftarrow$ RepresentarRotación($G, \mathcal{O}$)
8: **if not** R **then**
9: $R \leftarrow$ RepresentarReflexión($G, \mathcal{O}$)
10: **end if**
11: **else**
12: $\mathcal{O} \leftarrow$ FindAlmostSymmetry(G^{-1}, b) ▷ Al utilizar el algoritmo con el grafo inverso, en vez de ignorar aristas, se introducen aristas ficticias.
13: **if** $\#\mathcal{O} < \#V$ **then** ▷ Si no todas las órbitas son triviales con el grafo inverso
14: $R \leftarrow$ RepresentarRotación($G, \mathcal{O}$)
15: **if not** R **then**
16: $R \leftarrow$ RepresentarReflexión($G, \mathcal{O}$)
17: **end if**
18: **end if**
19: **end if**
20: $b \leftarrow b + 1$.
21: **end while**
$T \leftarrow$ PosicionarEtiquetas(G, R)
return R, T
22: **end procedure**

Nótese que el algoritmo siempre termina de ejecutar pues para $b = \#E$, FindAlmostSymmetry ignora todas las aristas, encontrándose la rotación trivial con representación circular (todos los nodos en la misma órbita).

4.2.9. Reconocimiento de Familias

Las familias de interés son los árboles y los grafos bipartitos. Anillos, ruedas y las otras familias identificadas en los requerimientos (sección 5.3) dan resultados apropiados con el proceso estándar. En forma paralela al procesamiento general, el sistema verifica si el grafo ingresado es de alguna de estas familias. Para cada familia cuya verificación sea positiva, se genera una representación acorde y se añade a las opciones que se muestran al usuario.

Árboles

Lo primero que hace el algoritmo es chequear que el grafo sea un árbol. Por propiedades de árboles, un grafo conexo con $\#E = \#V - 1$ es necesariamente un árbol. Basta contar entonces los vértices y las aristas para realizar el chequeo.
Si bien los árboles como objeto matemático no tienen nodos distinguidos, para su representación es tanto usual como útil la existencia de un nodo especial raíz, que puede ser especificado por el usuario. Cuando no lo es, se elige automáticamente para minimizar la distancia de la raíz a las hojas. Para esto se busca el camino simple más largo dentro del árbol y se toma como raíz su nodo central (o uno de ellos si es de largo par). Luego de que se tiene la raíz se dibuja el árbol, de forma que la raíz quede arriba y las ramas del árbol hacia abajo. Obtener las coordenadas de los vértices se hace de manera recursiva y en profundidad primero, fijando los hojas primero y luego a partir de sus coordenadas fijando las de sus respectivos padres. Las coordenadas de las hojas se fijan en el eje vertical según su profundidad (la raíz tiene coordenada cero). En el eje horizontal, están equiespaciadas entre sí y su valor se calcula según cuántas hojas han sido fijadas hasta el momento. Luego que se tienen las coordenadas de los hijos se calcula la del padre. En el eje vertical el valor de la coordenada será la profundidad del vértice y en el eje horizontal se calculará la coordenada promediando las coordenadas (del eje horizontal) de sus hijos.

Bipartitos

Para saber si un grafo es bipartito se utiliza teoría de la coloración de grafos. El problema consiste en pintar todos los vértices de colores, de tal forma que no haya dos vértices hermanos pintados del mismo color. Para que un grafo sea bipartito, es necesario que sea posible pintar el grafo con solo dos colores.

El algoritmo para hacer esto es simple: se pinta un vértice inicial de un color y luego se recorren sus vértices vecinos pintándolos del otro color. Luego se procede recursivamente. Si en algún momento se intenta pintar un vértice de un color distinto al que ya tiene, entonces el grafo no es bipartito. En cambio, si esto no pasa, se puede representar el grafo como bipartito, y los dos grupos de vértices vienen dados por los colores que fueron pintados. Luego que se tienen los dos grupos, los mismos se insertan en columnas paralelas.
También es posible reconocer grafos tripartitos, pero para más de dos colores es un problema NP-Completo[9] y por razones de eficiencia no se implementó.

4.2.10. Múltiples opciones de representación

Para cada grafo conexo, el **procesamiento principal en servidor** consiste en obtener las representaciones de árbol y/o bipartito si existen, así como hasta tres opciones[4] de ObtenerCoordenadas con diferentes budgets iniciales. La obtención de tres representaciones en lugar de una es un extra que se agregó para mejorar la experiencia de usuario, pues en ocasiones representaciones con un budget ligeramente mayor resultan más agradables a la vista[5].

4.2.11. Componentes conexas y puentes

Hasta aquí se ha especificado el comportamiento de la aplicación con grafos conexos, tanto para grafos generales como para las familias particulares que se reconocen. La aplicación realiza también pre y post-procesamiento respecto de este procesamiento principal. Más adelante, en el contexto de arquitectura (6) se verá que el procesamiento principal descripto antes sucede en el servidor, mientras que el pre y post-procesamiento se ejecuta del lado del cliente. Se realizan dos conjuntos de operaciones: *componentes conexas* y *división por aristas puente*. Ambos comprenden subdivisión, procesamiento principal en servidor (descrito anteriormente) y ensamblado posterior.

[4] menos de tres si las representaciones son idénticas o si la ejecución toma demasiado.

[5] un trabajo futuro interesante es la recomendación automática de la mejor representación

Componentes Conexas

Cuando el grafo no es conexo, se desea mostrar sus componentes conexas de forma separada para su mejor comprensión. Se utiliza DFS para encontrarlas y se envían de forma independiente para el procesamiento principal. En la etapa de ensamblado se ubican separadas verticalmente con espaciado uniforme.

División por aristas puente

Únicamente cuando el grafo es conexo:

Definición 4.2.2. Una arista e de un grafo conexo G es un **puente** si el grafo $G' = (V, E \setminus \{e\})$ no es conexo.

Se busca una arista puente con diferencia de cardinalidad de nodos minimal. Si existe dicho puente y hay tres o más nodos en ambos componentes[6], considérense las componentes conexas que se obtienen removiendo la arista puente. Llamemos *nodos puente* a los extremos de esta arista. Nótese que hay un nodo puente en cada componente.
Las componentes se envían de forma independiente al procesamiento principal, con las siguientes modificaciones en el algoritmo:

- Si la representación es una rotación, se fuerza el nodo puente a ser el primer nodo de la última órbita.
- Si la representación es una reflexión, el nodo puente será forzosamente fijo y el último de la secuencia unidimensionalmente representable.
- Si se reconoce como un árbol, la raíz será obligatoriamente el nodo puente.
- No se realiza reconocimiento de bipartitos.

Con las modificaciones mencionadas, por la construcción de los algoritmos, el nodo puente tendrá coordenada $x = 0$ y su coordenada y será máxima entre los nodos con coordenada x nula, permitiendo una fácil unión de las las componentes.

[6]con dos o un nodo en un componente, la representación resulta idéntica a la que se obtiene sin aplicar puentes.

En la fase de ensamblado, las componentes se ubican una sobre otra con espaciado de una unidad. La de arriba se invierte verticalmente para que los nodos puente puedan unirse. Se muestra un ejemplo en la figura 4.8.

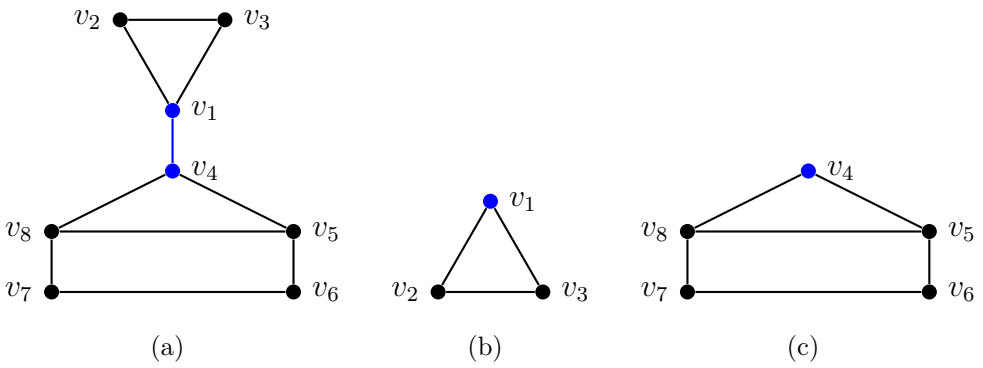


Figura 4.8: (a): resultado final con arista y nodos puente marcados en azul; (b) y (c): procesamiento independiente de cada componente, tal que el nodo puente tiene coordenada $x = 0$ e y maximal entre los nodos con coordenada $x = 0$.

4.3. Consideraciones estéticas

En esta sección se justifican algunos de los criterios que fueron aplicados en la representación de los grafos.

4.3.1. Reflexiones sobre eje vertical

Giannouli[10] afirma que el ser humano detecta más rápidamente son las simetrías especulares: reflexiones sobre el eje vertical. Intuitivamente parece tener sentido: las caras y cuerpo humano, que los humanos detectamos preatencionalmente, tienen ese tipo de simetría. Welch y Kobourov[28] han econtrado evidencia que concuerda con la preferencia de simetrías especulares en representaciones de grafos. Por tanto, las simetrías preferidas en la aplicación son las de ese tipo, a excepción de si se encuentran rotaciones que fijan menos puntos.

4.3.2. Simetría aproximada

Las simetrías que percibimos en la vida diaria muchas veces no son perfectas. Por ejemplo, las caras humanas nunca son perfectamente simétricas, pero percibimos la simetría en ellas. Tiene sentido explorar entonces la capacidad humana de ver simetrías aproximadas en grafos. Treder caracteriza la percepción humana de simetría como "resistente a ruido"[26]. Las representaciones de la figura 4.9 no tienen simetría exacta, pero no obstante se percibe cierta simetría.

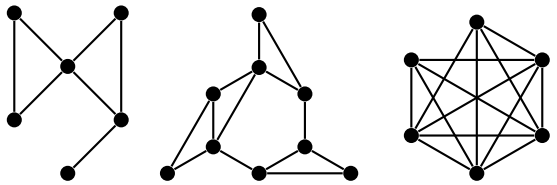


Figura 4.9: Grafos con simetrías aproximadas.

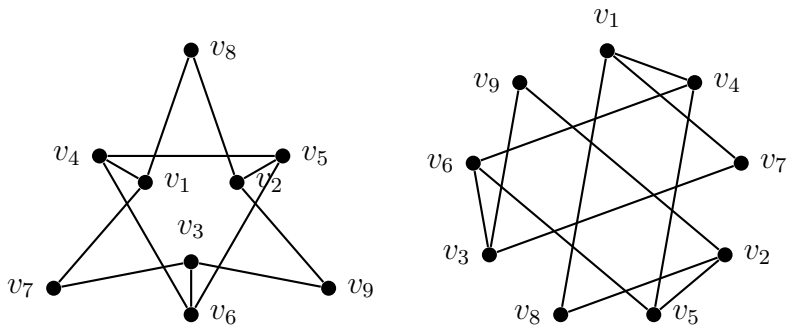
4.3.3. Minimización de largo de aristas

La minimización del largo de aristas, como se mencionó tiene fundamentos en que ayuda al buen funcionamiento del algoritmo (provoca menos fallos), pero además es visualmente agradable que nodos relacionados entre sí estén tan cerca como sea posible.

4.3.4. Simetría local

Si bien el algoritmo presentado intenta mostrar *una* simetría con mínimos puntos fijos, los grafos con simetrías suelen tener grupos automórficos más grandes. Sin pretender efectivamente mostrar más simetría en todos los casos, en la medida de lo posible se intenta transmitir la existencia de otras simetrías en el grafo. En rotaciones, no es necesario que cada órbita tenga una circunferencia única, y en reflexiones, ni siquiera que cada órbita se ubique en una circunferencia, pero el algoritmo posiciona cada órbita en una circunferencia única para que las mismas se perciban mejor.
Se muestra un ejemplo.

Ejemplo 4.3.1. Sea el grafo y su representación de la figura 4.10(a), que muestra una simetría rotacional con ángulo $\pi/3$, y sus órbitas $\{\{v_1, v_2, v_3\}, \{v_4, v_5, v_6\}, \{v_7, v_8, v_9\}\}$. La aplicación desarrollada ubica cada órbita en una circunferencia única. La intención es que pueda percibirse que cada órbita puede ser rotada de forma *localmente* independiente de las demás. Se muestra el resultado en comparación con el algoritmo descripto por Eades y Hong[13] para mostrar un grupo automórfico rotacional sin tener en cuenta consideraciones estéticas.



(a) (b)

Figura 4.10: (a): resultado de la aplicación; (b) resultado del algoritmo de Eades y Hong. Ambos muestran simetría rotacional con ángulo $\pi/3$. La heurística aplicada en (a) muestra también tres simetrías axiales.

4.3.5. Otros

- En las rotaciones, se obliga a que exista un nodo en la posición superior en la órbita más exterior. Elegido en base a que así son las representaciones tradicionales de muchos grafos (ver figura 4.11 para un ejemplo).
- En las reflexiones, se eligen los radios de las órbitas para que los nodos más superiores y más inferiores de cada órbita tengan coordenada y entera.

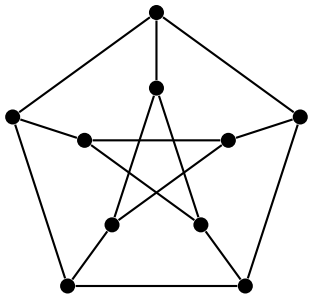


Figura 4.11: El grafo de Petersen suele representarse con un nodo hacia arriba en la órbita más exterior.

Capítulo 5

Proceso de Desarrollo

5.1. Modelo de Proceso

El desarrollo siguió un proceso Kanban usando como soporte la herramienta en línea Trello (`http://trello.com`) para el manejo del tablero (*board*). Al igual que Scrum, Kanban es un proceso de desarrollo ágil, separando tareas complejas en tareas más sencillas que son desarrolladas de manera fluida y concurrente. El control del estado de las tareas se lleva en un tablero y las tareas van avanzando por el tablero a medida que las distintas etapas son completadas. La principal diferencia es que Scrum utiliza Sprints, estos son intervalos de tiempo que una vez concluidos se entrega nuevo software, mientras que en Kanban no se tiene entregas intermedias. Por ello, Kanban resulta más apropiado para este proyecto.
Se aplicó una modificación al método Kanban estándar debido a la naturaleza de los entregables del proyecto. Al no existir entregas intermedias, la obtención rápida de un *minimum value product* (MVP) no fue un factor relevante en el desarrollo. Su lugar lo ocupa la obtención segura de dicho MVP dentro del marco temporal del proyecto, pero no necesariamente en el menor tiempo posible.
A tales efectos, el backlog de proyecto fue clasificado en tareas críticas y tareas opcionales. Exceptuando dependencias entre tareas inherentes al desarrollo, las tareas críticas no tienen diferencias de prioridad entre sí, pues todas deben realizarse. Las tareas opcionales sí la tienen, en base a sus respectivos retornos de la inversión[1]. A la hora de tomar una tarea, los desarrolladores toman la

[1] valor agregado que otorgan al producto en relación inversa al tiempo que demandan.

tarea con mayor prioridad que no dependa de otras aún no completadas. Las tareas críticas se considera que todas tienen prioridad *infinito* y dentro de estas el desarrollador tiene libertad de elección.
El flujo de trabajo puede resumirse de la siguiente manera:

- Cuando se relevan nuevos requerimientos: se añaden tareas al backlog de proyecto.
- Cuando se profundizan requerimientos existentes: las tarjetas del backlog son extendidas y/o subdivididas en nuevas tarjetas.
- Implementación de las tarjetas:
    1. Un desarrollador toma una tarea del backlog, asignándola a sí mismo y cambiando su estado a "haciendo".
    2. Al terminar, el trabajo espera retroalimentación en el estado "para revisar".
    3. El otro desarrollador aprueba los cambios y mueve la tarjeta a "listo" o alternativamente, se retrocede la tarjeta a "haciendo" para poner en común las discrepancias.
    4. Una vez resueltas las discrepancias, se mueve la tarjeta a "listo".

5.2. Gestión de Riesgos

Como en todo proyecto de ingeniería de software, existieron riesgos asociados al mismo. Con el objetivo de identificarlos, se realizó un estudio de riesgos en una etapa temprana del desarrollo. A continuación se describen brevemente. Todos, a excepción de 2.1, son riesgos menores que en el peor de los casos atrasarían el desarrollo y fueron asumidos.

1. Relacionados al proceso:
    1.1. Dificultad en la estimación de tiempos para ciertas tareas pues no se tenían conocimientos de algunas de las tecnologías usadas.
2. Relacionados al algoritmo:

2.1. Inexistencia o no factibilidad de implementación para algoritmos que puedan dibujar de manera estéticamente agradable grafos cualesquiera. Este riesgo es crítico y para su prevención se priorizaron las tareas de implementación de dichos algoritmos (dado que si no se encontrasen, el proyecto no podría haber continuado).

2.2. Dificultad para cuantificar aptitud estética de un grafo.

2.3. Existencia de grafos específicos que no presenten ninguna representación estéticamente agradable.

2.4. Carecer del conocimiento adecuado de matemática.

3. Relacionados a la comunicación:

3.1. El director del Proyecto no se encontraba en el país, por lo que la comunicación se realizó por medios no óptimos para la discusión y con problemas de disparidad de zona horaria.

5.3. Requerimientos

En esta sección se detallan los requisitos del sistema. Comprende los requisitos críticos así como los opcionales. De estos últimos, los que no fueron implementados se encuentran señalados como tales.
A continuación se presentan los requisitos expresados como historias de usuario en una estructura arborescente, con el objetivo de transmitir la subdivisión de las tarjetas. Junto a algunas las historias se adjunta un breve comentario sobre su implementación, factibilidad o subdivisión en nuevas tarjetas. Los requisitos opcionales se muestran con su prioridad asociada.

5.3.1. Requisitos Funcionales

1. **Entrada de grafos:** Como usuario, quiero poder ingresar grafos fácilmente.
   - La interfaz debe ser fácil de entender para usuarios novatos sin experiencia de programación.
   - También debe permitir uso experto con alta productividad.

Se eligió una interfaz basada en texto por ser la más adecuada para el uso experto. A la vez, se intentó mantener mínima la complejidad de la sintaxis para satisfacer a usuarios novatos. La profundización de la tarjeta origina nuevos requerimientos.

1.1. **Intérprete:** Como usuario, quiero que la interfaz de ingreso de texto sea fácil de utilizar.
- La sintaxis debe ser simple y de ser posible seguir algún estándar.
- Se deben permitir comentarios en el código para mayor legibilidad.
- Los errores de sintaxis deben señalarse claramente.
- Si existen opciones de mayor complejidad deben ser de carácter opcional.

El parser implementado utiliza el formato TGF [25] puro en el caso base y una extensión de él cuando se requieren opciones avanzadas (descrita en el apéndice B).

1.2. **Inicio rápido:** Como usuario, quiero tener acceso a grafos típicos al momento de comenzar para así acelerar el ingreso de datos.
- Opcional. Prioridad: media.

1.3. **Grafos dirigidos:** Como usuario, quiero poder convertir mi grafo en uno dirigido sin agregar manualmente puntas de flecha a las aristas.
- Opcional. Prioridad: alta.

1.4. **Etiquetas automáticas:** Como usuario, quiero poder activar y desactivar la aparición de etiquetas automáticas.
- Opcional. Prioridad: media.

Este requisito fue parcialmente implementado, se permite la generación automática de etiquetas pero su eliminación es únicamente manual.

2. **Representación:** Como usuario, quiero que el sistema posicione los nodos del grafo automáticamente.
   - El resultado debe proveerse como máximo en pocos segundos.

El algoritmo utilizado se describe en detalle en el capítulo 4. Aquí meramente se detallan las subtarjetas correspondientes al posicionamiento de nodos.

2.1. **Correctitud:** Como usuario, quiero que la representación del grafo sea correcta y sin ambigüedad.

Se define formalmente qué constituye una representación válida en el capítulo de fundamento teórico (cap. 2).

2.2. **Subgrafos:** Como usuario, quiero que grafos disconexos se muestren como múltiples grafos conexos para así mejorar la legibilidad.

2.3. **Simetría Aproximada:** Como usuario, quiero que grafos asimétricos se muestren con simetrías aproximadas.

2.4. **Múltiples Representaciones:** Como usuario, quiero que se me provean distintas representaciones del mismo grafo.
   - Cada opción debe ir acompañada de una leyenda descriptiva.
   - Opcional. Prioridad: baja.

2.5. **Familias:** Como usuario, quiero que ciertas familias de grafos sean representadas en sus formas canónicas.
   - Familias propuestas: árboles, grafos bipartitos, grafos tripartitos, anillos, grafos completos, ruedas.
   - Opcional. Prioridad: alta.

Árboles y bipartitos fueron reconocidos de manera específica. Grafos tripartitos quedó fuera del alcance del proyecto debido a su complejidad (es un problema NP-completo[9]). Las familias restantes dan resultados apropiados en el algoritmo general sin tratamiento específico.

2.6. **División de grafos:** Como usuario, quiero que grafos con aristas puente o vértices de articulación se traten como componentes disconexas y luego se unan para así dar mejores representaciones con mejor performance.

- Opcional. Prioridad: media.

2.6.1. **Puentes:** Como usuario, quiero que grafos con aristas puente se representen como componentes disconexas unidas por tal arista.

- Opcional Prioridad: media.

2.6.2. **Articulaciones:** Como usuario, quiero que grafos con vértice de articulación se representen como componentes disconexas unidos por tal vértice.

- Opcional Prioridad: media.

No implementado por razones de alcance y complejidad.

3. **Vista en pantalla:** Como usuario, quiero que los grafos se muestren en pantalla.

- Deben mostrarse etiquetas de nodos.
- Debe poder añadirse puntas de flecha a las aristas para representar grafos dirigidos.
- Opcionalmente, puede permitirse la modificación de colores de nodos y aristas, tamaños y formas.
- Opcionalmente, pueden permitirse multigrafos y lazos.

La funcionalidad de multigrafos por razones de complejidad, mientras que la de lazos no lo fue simplemente por razones de alcance.

4. **Salida de grafos:** Como usuario, quiero que existan mecanismos de salida para los grafos para así poder utilizarlos en otros contextos.

   4.1. **TikZ:** Como usuario, quiero que los grafos sean exportables a LaTeX.

   4.2. **Imágenes:** Como usuario, quiero que los grafos sean exportables a imágenes para así utilizarlos en variados contextos.
   - Opcional. Prioridad: alta.

   4.3. **Enlaces:** Como usuario, quiero poder compartir y guardar grafos mediante enlaces del sitio para así visitarlos nuevamente.
   - Opcional. Prioridad: alta.

5. **Documentación en línea:** Como usuario, quiero que toda la información necesaria para utilizar el sitio se encuentre disponible en él.

5.3.2. Requisitos no Funcionales

1. **Infraestructura:** Como dueños del producto, queremos que el sistema sea alojable en un servidor único para así hacer económica su permanencia en línea.

2. **Localización:** Como usuario, quiero que el sistema se encuentre disponible en múltiples idiomas.
   - Español e Inglés.
   - Con escalabilidad a más lenguajes.

3. **Robustez:** Como usuario, quiero que el sistema no deje de funcionar íntegramente si ocurre una falla.

4. **Diseño Responsive:** Como usuario, quiero poder utilizar la aplicación en dispositivos móviles.

5. **Curva de Aprendizaje:** Como usuario, quiero poder crear mis primeros grafos en pocos minutos sin entrenamiento previo.

6. **Métricas de uso:** Como dueños del producto, queremos saber cuánto uso recibe.
   - Opcional.

Se implementó con Google Analytics.

Capítulo 6

Arquitectura

En este capítulo se describe la arquitectura del sistema a nivel físico y lógico. Los principales nodos del sistema (ver figura 6.1) son los siguientes:

1. **Clientes**: web o mobile que consumen la aplicación mediante su navegador.
2. **Servidor web**: encargado de servir el sitio al usuario.
3. **Servidor de aplicaciones**: que procesa la información de los grafos para dar coordenadas a sus nodos.
4. **Almost Symmetry**: dependencia principal del sitio, como se vio en el capítulo 4.

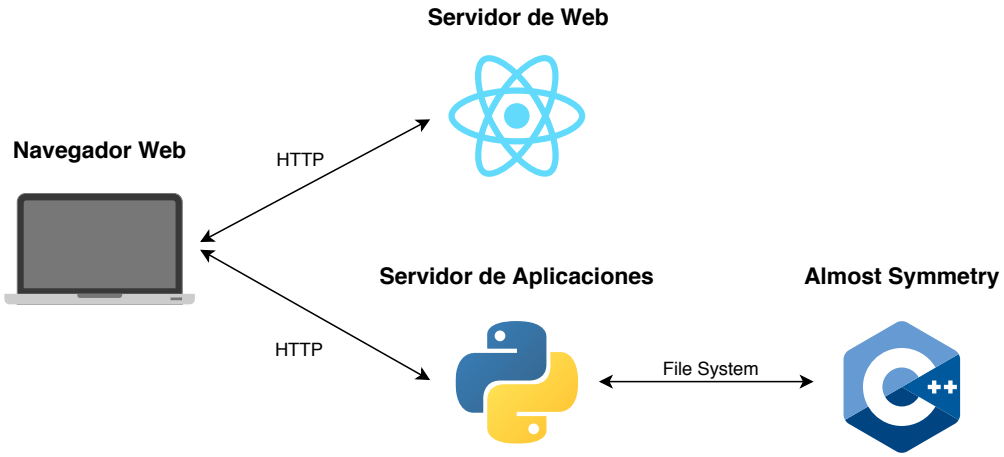


Figura 6.1: Diagrama del sistema.

6.1. Clientes

Los clientes acceden al sistema mediante el navegador web, que puede ser utilizado desde una computadora o un dispositivo móvil.

6.2. Servidor Web

El servidor web aloja el sitio, implementado con la biblioteca React (`http://reactjs.org`).
A continuación se describen los principales componentes lógicos de la aplicación React que se sirve a los clientes. Los nombres de los componentes se presentan en inglés pues es la lengua utilizada en el código.

6.2.1. Componentes

1. **Input Section:** Maneja el ingreso y *parsing* de la sintaxis del grafo.

   La entrada de texto con resaltado de sintaxis para informe de errores se implementó con Draft.js (`https://draftjs.org/`).

   El parsing se realizó con Nearley.js (`https://nearley.js.org/`), utilizando una gramática personalizada para el formato TGF extendido (ver apéndice B).

2. **Output Section:** Contiene la vista en pantalla y las diversas opciones de exportado de grafos.

   La representación en pantalla se realizó con Sigma.js (`http://sigmajs.org/`).

   Para las opciones de salida no se utilizaron bibliotecas externas. Las URLs de grafos se generan codificando la entrada del usuario en Base 64. La generación de código TikZ se realizó mediante manipulación de texto.

3. **API Client:** Encapsula la codificación del grafo para su envío al Servidor de Aplicaciones, así como el envío en sí mismo y el manejo de una caché de grafos para evitar pedidos duplicados. Adicionalmente, realiza el pre- y post-procesamiento descripto en el capítulo 4, aplicando los algoritmos de componentes conexas y puentes.

4. **Secciones estáticas:** Las demás secciones no son de mayor interés y muestran información estática para el usuario: documentación, tutorial, funcionalidades, ejemplos.

El servidor se encapsula en un contenedor de Docker(`http://docker.com`), como se utilizó durante el desarrollo para el despliegue local. Docker es una plataforma que permite desarrollar y correr una aplicación en un ambiente aislado del sistema llamado contenedor. El contenedor incluye los archivos necesarios para ejecutar la aplicación así como también un conjunto de comandos que indican cómo instalar todas las bibliotecas necesarias para la ejecución de la aplicación y cómo se ejecuta. Convenientemente, el servicio en línea Heroku(`http://heroku.com`) dispone de integración con Docker, lo que hace relativamente sencillo el despliegue ya que a partir del contenedor de la aplicación se tiene todo el entorno instalado. Además se rentó el dominio `http://graphsym.com` y se ingresó la configuración de DNS pertinente para la redirección.

6.3. Servidor de Aplicaciones

El Servidor de Aplicaciones, implementado en Python, responde a los pedidos del Servidor Web. Este tiene una única API con un único método que es el que recibe un conjunto de grafos con sus respectivos vértices y aristas, y

luego de calcular las coordenadas de dichos vértices devuelve un conjunto de posibles representaciones para cada grafo de entrada. Los datos de los grafos entrantes y salientes se codifican en formato JSON, fácilmente codificable y decodificable tanto en Javascript como Python. Para buscar las simetrías de un grafo es necesario almacenar un archivo con los datos del grafo. Esto se debe al funcionamiento de Almost Symmetry, que lee un archivo con los datos del grafo cuya ubicación es un parámetro de entrada, calcula las simetrías con el budget elegido y luego guarda el resultado en otro archivo que debe ser leído. Para esto es necesario utilizar el sistema de archivos y tener comunicación con el sistema operativo para que este ejecute el programa.
Al igual que el servidor web, el despliegue se realizó utilizando Docker y Heroku. En este caso se tuvieron que realizar varias modificaciones a la configuración del contenedor para obtener los permisos necesarios para poder ejecutar Almost Symmetry, ya que para la ejecución de este se requiere del uso del sistema de archivos, y Heroku tiene ciertos directorios reservados que generaron dificultades.

Capítulo 7

Usabilidad

7.1. Estudio de Usabilidad

Se realizó una prueba de usabilidad con potenciales usuarios. El objetivo fue observar el comportamiento de los usuarios para obtener información sobre la curva de aprendizaje, adaptabilidad a los distintos tipos de usuario, y satisfacción general.
El experimento se diseñó para cinco usuarios según lo recomendado por Nielsen y Norman[14], mas se realizaron solo cuatro pruebas debido a disponibilidad horaria de los potenciales usuarios de prueba.

7.1.1. Consigna

La consigna mostraba un grafo correspondiente a una red de computadoras, tal que uno de sus cables estaba roto. Se pidió a los usuarios que completaran dos tareas.

1. A quienes utilizaran una herramienta de dibujo automático de grafos, dibujar la red con ella. En cambio, a los que utilizaran una herramienta con posicionamiento manual, estimar cuánto les llevaría[1].

2. Dibujar el grafo en la aplicación desarrollada.

Finalmente se hicieron algunas preguntas.
El dibujo de la red comprende dos partes:

[1]por razones de tiempo y debido a la escasa utilidad de registrar los hábitos de uso para las herramientas con posicionamiento manual

1. Dibujar el grafo de la figura 7.1 (originalmente mostrado con otro posicionamiento).
2. Etiquetas y colores:
   - Marcar como roto el cable $B - D$ con rojo, línea punteada y/o un rótulo.
   - Pintar de verde los nodos A y G.
   - Marcar con flechas el camino más corto entre A y G, dado que $B - D$ está roto.

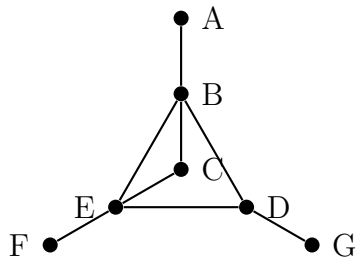


Figura 7.1: El grafo que representa la red de computadoras.

7.1.2. Resultados

Evaluación

Solamente el usuario 1 utiliza una herramienta automática de dibujo de grafos (SageMath). Los tiempos para completar la tarea en la aplicación desarrollada son ampliamente superiores (ver cuadro), más aún al tomar en cuenta que la otra aplicación es la de uso habitual para el usuario evaluado. El valor faltante en el cuadro se debe a que el usuario no logró realizar lo pedido con su herramienta usual.

	Parte 1	**Parte 2**
SageMath	9 min	N/A
Aplicación desarrollada	2.5 min	4.5 min

Los demás usuarios utilizan TikZ manualmente para dibujar grafos. Las estimaciones para completar la tarea usando TikZ variaron en un rango entre

media hora y dos horas. Sin embargo, los tiempos con la aplicación desarrollada fueron de 9 minutos, 15 minutos y 19 minutos. Se concluye que en materia de tiempo la aplicación desarrollada da resultados satisfactorios.
Sobre la calidad de las representaciones obtenidas, todos los participantes la calificaron con una puntuación de 4 en una escala del 1 al 5 (donde 5 es el mejor valor).
Globalmente, todos los participantes dicen preferir la nueva aplicación frente a sus anteriores alternativas, y la califican con máxima puntuación en facilidad de uso. Uno de ellos incluso preguntó si ya estaba disponible y solicitó que lo notificasen cuando estuviese accesible en la web.

Problemas revelados

Un estudio de usabilidad tiene como objetivo además de la validación, la búsqueda de problemas que puedan tener los usuarios o mejoras que les serían útiles. Se presentan los problemas y nuevas funcionalidades que surgieron de este estudio:

- La funcionalidad de posicionamiento de etiquetas fue sugerida por uno de los usuarios cuando algunas de sus etiquetas se superpusieron con las aristas.
- El usuario que tomó más tiempo demostró un mejor aprendizaje por ejemplos, a diferencia de los otros usuarios que prefirieron la lectura de documentación. En consecuencia, se decidió la creación una biblioteca de ejemplos disponible en el sitio[2].
- Los usuarios tuvieron dificultades con la rigidez de la sintaxis respecto al uso de espacios, por lo que se modificó la misma a una versión más laxa.
- Los errores, que se encontraban marcados, no fueron lo suficientemente notorios en algunos casos. Se agregó un mensaje de error complementario y de mayor tamaño.
- Uno de los usuarios se mostró desconforme con que al modificar el grafo, por ejemplo agregando una etiqueta, no se recordara qué opción

[2]La biblioteca de ejemplos quedó fuera del alcance del proyecto al elegirse priorizar otras tareas.

de representación estaba seleccionada. Se modificó para que el sistema sí lo recuerde.

7.2. Estudio de Rendimiento

Un aspecto fundamental de cualquier algoritmo es el rendimiento. Es aún más importante cuando el caso de uso es, como en este caso, uno de tiempo real y con usuarios que simplemente desean obtener de manera rápida una solución a su problema, ignorando los aspectos técnicos de fondo. Se realizaron estudios de rendimiento para medir la aptitud del producto.
En primera instancia, hay tres aspectos de un grafo que pueden influenciar el tiempo de ejecución del algoritmo:

- La cantidad de nodos.
- La cantidad de aristas.
- El grado de simetría presente en el grafo.

Los dos primeros elementos son evidentes. El tercero responde al budget que se necesita para encontrar una partición no trivial, es decir, cuántas aristas hay que quitar o añadir al grafo para obtener un grafo simétrico.
Para las pruebas, se realizan ciertas modificaciones al algoritmo, en comparación con el funcionamiento usual:

- Se aumenta la cota de tiempo de ejecución para que no sea alcanzada. En la práctica la cota es mucho menor, de acuerdo al tiempo esperado de ejecución, al tiempo aceptable de espera y a consideraciones de seguridad (prevención básica contra ataques de denegación de servicio mediante el ingreso de grafos demasiado grandes).
- Se busca una única representación simétrica. Usualmente, dentro de la cota de tiempo, se buscan hasta tres soluciones.

El algoritmo que induce el principal costo es la dependencia FindAlmostSymmetry. Los demás componentes de la implementación son todos de orden polinomial[3]. El artículo original de Find Almost Symmetry cuenta con

[3] A excepción del reordenamiento heurístico de nodos, que hasta $n = 7$ se realiza con implementación exponencial para mejores resultados. De todas formas, para $n \leq 7$, su costo en tiempo es despreciable.

datos de rendimiento, pero aplicado a grafos más grandes y con un mayor poder de cómputo disponible. Las pruebas se realizan sobre el algoritmo de representación implementado completo, que incluye llamada(s) a FindAlmostSymmetry.

7.2.1. Grafos de prueba

Primero, todas las pruebas se realizan para grafos conexos, pues son los peores casos del algoritmo. Cuando los grafos no lo son, se los trata como componentes independientes, usualmente mejorando los tiempos de FindAlmostSymmetry por su naturaleza exponencial.
Segundo, la aplicación está diseñada para el dibujo simétrico o cuasi-simétrico de grafos. Sin embargo, existen grafos asimétricos. Algunos de ellos incluso siguen sin presentar cuasi-simetrías hasta que se alcanzan budgets altos. Si bien es posible encontrar una representación cuasi-simétrica para un grafo "muy" asimétrico (por ejemplo, uno que de 20 aristas deban ignorarse 7 para encontrar una simetría), a causa de las aristas que fueron ignoradas o inducidas la representación no será agradable a la vista, incluso no percibiéndose la simetría en ocasiones. No tiene sentido entonces dar como entrada grafos así, pues su salida tampoco es útil para el usuario.
Para dar mediciones más ajustadas a los casos de uso reales de la aplicación, se realizan las pruebas restringiendo los grafos a ciertos budgets máximos. Como no se tienen métodos para generar grafos simétricos, y menos aún para generar grafos cuasi-simétricos, se debió buscar otra solución. Lo que se hace es generar grafos conexos aleatorios y descartar aquellos que al intentar encontrar su representación superen la cota de budget, es decir, sean demasiado asimétricos.

7.2.2. Cantidad de aristas fija

Como prueba inicial, se fija la cantidad de aristas en 20, 25 y 30, variando la cantidad de nodos de grafos conexos aleatorios con representación cuasi-simétrica con budget ≤ 2.
Los resultados se muestran en la figura 7.2. El tiempo aumenta con la cantidad de nodos, como es esperado. Además, puede verse un comportamiento interesante: el número de aristas parece no ser un factor relevante al tiempo de procesamiento. Los datos para 20, 25 y 30 aristas para un número de nodos dado tienen valores promedio muy similares. Se realizó el test de

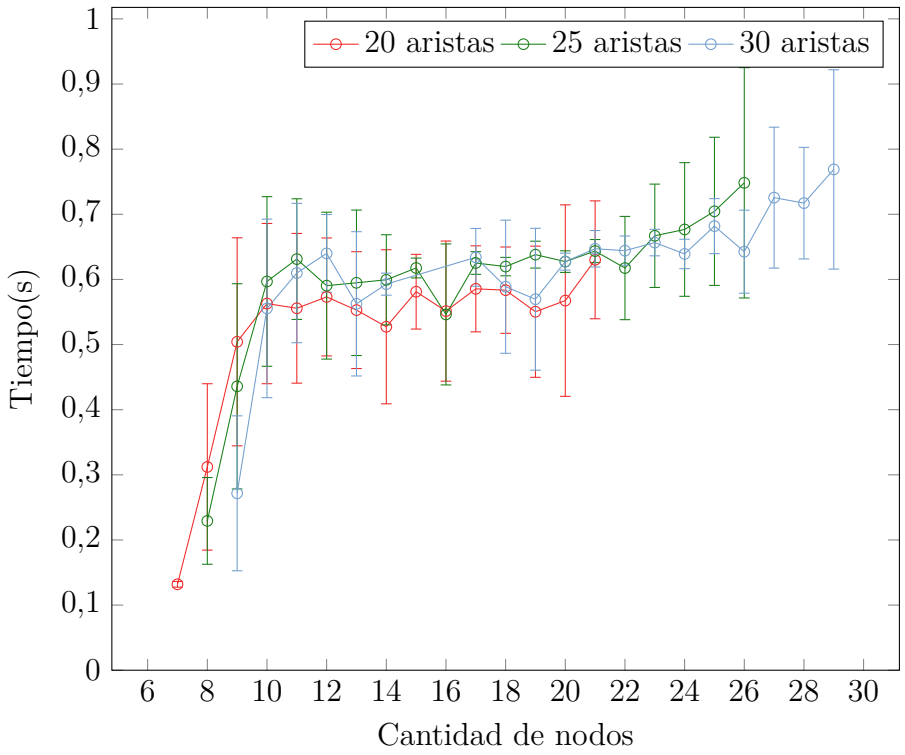


Figura 7.2: Aristas fijas, cantidad de nodos variable.

Kolmogorov-Smirnov en algunos valores de cantidad de nodos para dar respaldo a esta teoría[4]. Sin embargo, el test rechaza que las muestras estén idénticamente distribuidas con $p = 0{,}05$, sugiriendo que el número de aristas sí afecta el tiempo de ejecución.

[4]tomando los valores de aristas de 2 en 2, en cantidades de nodos elegidas procurando que hubiera suficientes muestras en cada conjunto.

Para estudiar más en detalle el impacto del número de aristas, se realizan pruebas con número de aristas variables para una cantidad fija de nodos.

7.2.3. Cantidad de nodos fija

Fijando la cantidad de nodos en n, se tienen grafos conexos con entre $n-1$ y $n(n-1)/2$ aristas.

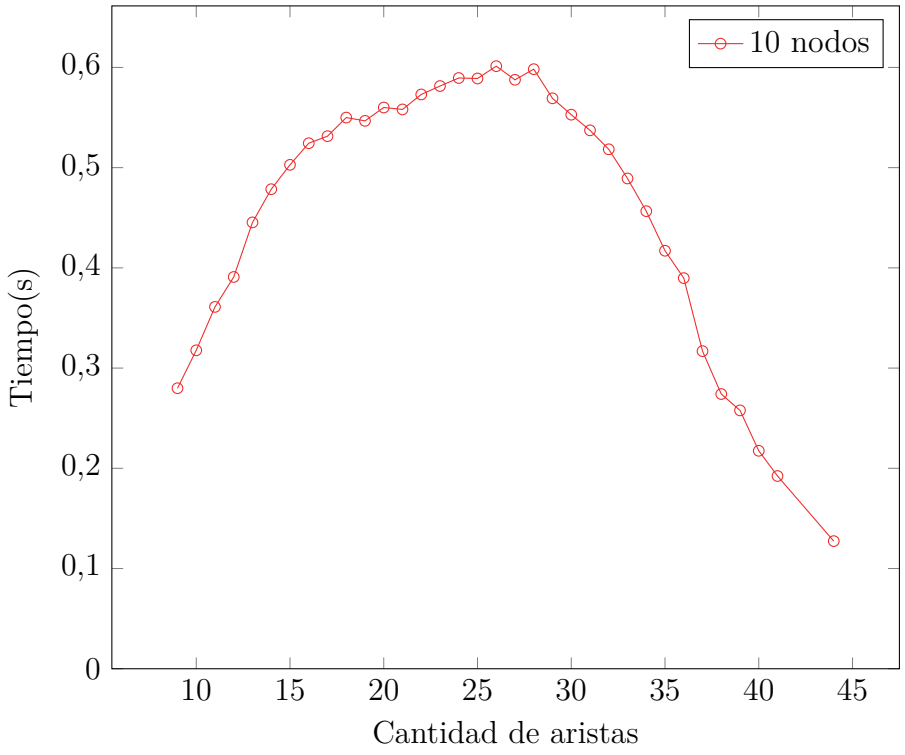


Figura 7.3: Cantidad de nodos fija, cantidad de aristas variable.

En la figura 7.3 se muestran los datos obtenidos para 10 nodos (nuevamente, para grafos con budget ≤ 2). En la gráfica se aprecia cómo el número de aristas influye en el tiempo de procesamiento, pero no de manera monótona: más aristas no siempre implican mayores tiempos. Acercándose a la mitad del rango de aristas se encuentran los mayores tiempos, que disminuyen a medida que la cantidad de aristas se aproxima a los extremos. ¿Por qué? En los datos obtenidos para cantidades mayores de nodos ($n = 15, n = 20$) se notó que cuanto más lejos se está de los extremos del rango de aristas,

resulta más difícil encontrar grafos aceptablemente simétricos (con menos de dos "imperfecciones"), en ocasiones no obteniéndose ninguno en cientos de muestras. Esto sugiere una razón: el incremento en los tiempos se debe a que los grafos son "menos simétricos" al alejarse de los extremos.
Para comprobar esta hipótesis, se discriminan por budget los datos obtenidos. En la figura 7.4 se muestran los datos discriminados por budget para 10 nodos así como su probabilidad relativa de ocurrencia. Puede verse que dado el budget, el número de aristas no impacta notoriamente el tiempo (aunque sí se aprecia una leve pendiente). Mucho más notorio es que la proporción con la que ocurre cada budget sí depende fuertemente del número de aristas, y que cada budget tiene una diferencia de tiempo importante con los demás. Estos datos afirman que los números de aristas cercanos a los extremos tienen mayor probabilidad de ser simétricos, por tanto tendrán mejores tiempos. Esto es una explicación razonable para la curva de la figura 7.3 por la naturaleza aleatoria de los grafos de prueba. Conceptualmente, lo que afecta más fuertemente el tiempo de ejecución no es el número de aristas sino el grado de simetría del grafo, como se verá a continuación.

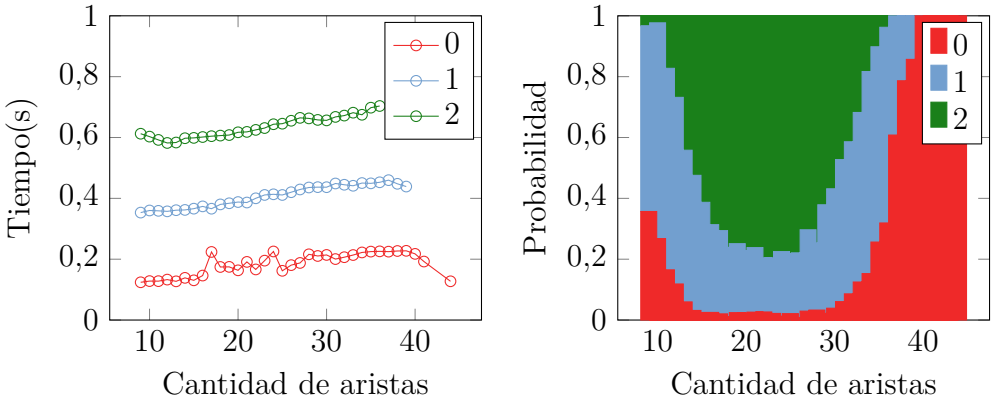
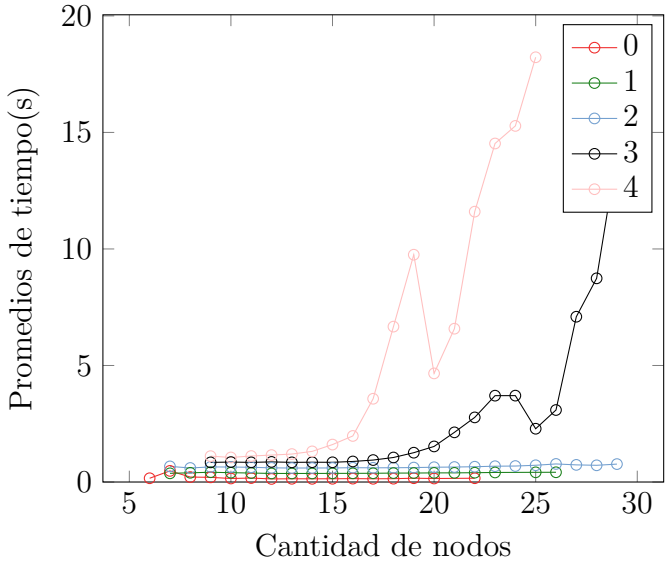


Figura 7.4: Cantidad de nodos fija (10), cantidad de aristas variable, discriminado por budget. (a) muestra los tiempos para cada budget, mientras que (b) muestra la probabilidad relativa de ocurrencia de cada budget.

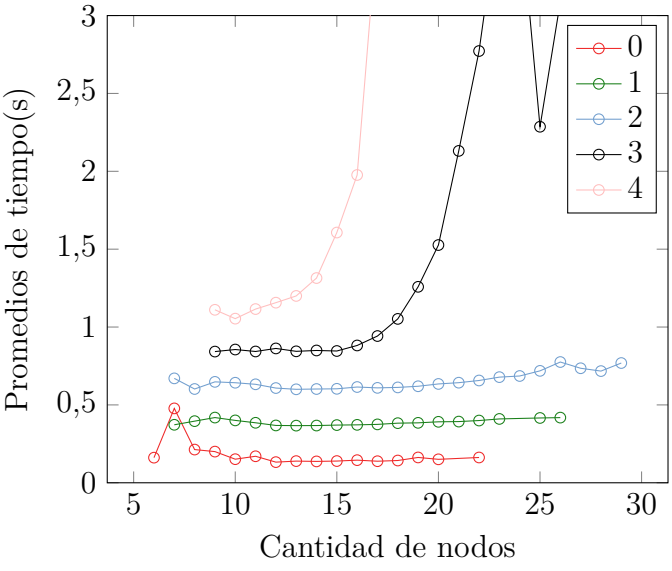
7.2.4. Estudio por budget

Como se habló en la sección anterior, cuán simétrico (o asimétrico) es un grafo, y por tanto el budget con el que se logra encontrar una representación, sí parece ser el factor más relevante. En este estudio se varían las aristas y los nodos, y se grafican los promedios de tiempo discriminados por budget como antes para cada cantidad de nodos (ver figura 7.5).
Los tiempos para budgets altos crecen rápidamente con la cantidad de nodos debido a la naturaleza exponencial de FindAlmostSymmetry. En la figura 7.5(b) se muestra la gráfica con el eje y magnificado para mejor comprensión de los budgets pequeños.
Puede apreciarse cómo para budget cero y uno se dejan de tener datos antes del final de la gráfica, pues no hay muestras. Esto ocurre debido a que la probabilidad de que un grafo aleatorio sea perfectamente simétrico o tenga una sola "imperfección" es muy baja y no se consiguieron muestras. Luego de mayor investigación, se descubrió que esta carencia del método estadístico elegido puede hablar bien de la aplicación implementada, haciendo pruebas con grafos que se saben simétricos. Por ejemplo, para K_{60}, el grafo completo de 60 nodos[5], se obtiene un tiempo de ejecución de aproximadamente dos segundos. Si bien todos los tiempos son exponenciales cuando se aumenta la cantidad de nodos, quizás el rendimiento en grafos simétricos siga siendo aceptable hasta alcanzar cantidades de nodos mucho mayores.
Un trabajo futuro interesante es el estudio estadístico del rendimiento de la aplicación en grafos más grandes pero solo en aquellos que presenten simetrías representables, con la principal problemática radicando en la generación de dichos grafos.

[5]que obviamente es simétrico (con cualquier budget)



(a)



(b)

Figura 7.5: Variación de tiempos para cada budget conforme aumentan los nodos. En (b) se muestra el eje y magnificado.

7.2.5. Estudio con nodos fijos

Un estudio interesante es ver cómo crece el tiempo necesario para grafos de un tamaño fijo según el budget. En la figura 7.6 se puede apreciar cómo crece el tiempo necesario para computar las coordenadas, a medida que aumenta el budget. Se fijó la cantidad de nodos en 20 para apreciar el rendimiento en la cota superior objetivo del proyecto. Se puede observar que con hasta tres "imperfecciones", se obtienen resultados en menos de cinco segundos, tiempos aptos para una aplicación en tiempo real. Esta gráfica también es una buena representación de cómo afecta el grado de simetría de un grafo al tiempo necesario para encontrar representaciones, observándose un crecimiento exponencial a medida que aumenta el budget. Esto nos permite concluir que la principal limitante en cuanto a los tiempos necesarios para calcular las simetrías viene dado por el grado de simetría del grafo, y no tanto por el tamaño del mismo.

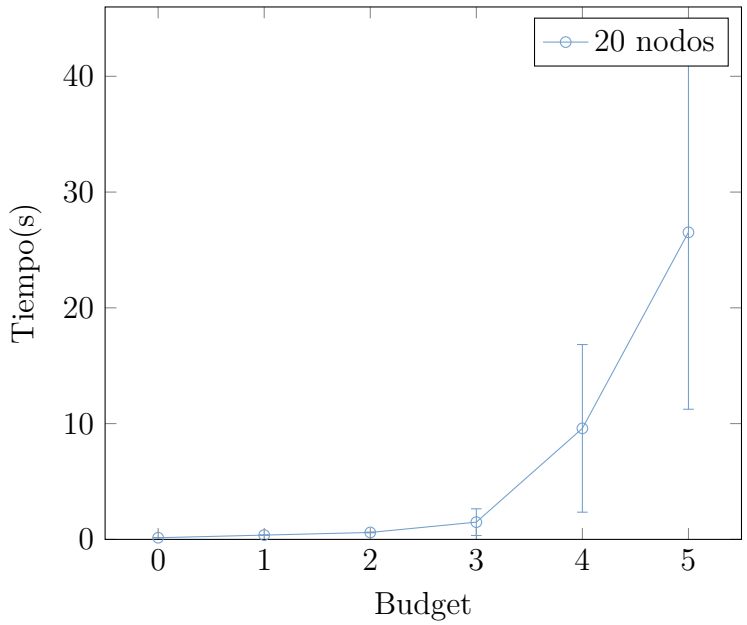


Figura 7.6: Número de nodos fijos, cantidad de aristas variable.

Capítulo 8

Conclusiones

El dibujo de grafos de forma simétrica es, como se vio, un problema difícil y aún no resuelto. Dado que no se conocen algoritmos para la búsqueda eficiente de automorfismos geométricos explícitos o incluso de automorfismos en general, para el problema práctico de visualización simétrica de grafos se debe recurrir a soluciones heurísticas, como los métodos *force-directed*.
A lo largo de este proyecto se construyó y evaluó un método alternativo para la representación simétrica basado en el problema de partición de automorfismos, para el cual se conocen algoritmos cuasi-polinomiales.
La solución implementada utiliza una biblioteca para encontrar las órbitas de un grafo, problema equivalente a encontrar los automorfismos, y luego se utilizan heurísticas varias para, a partir de dichas órbitas, realizar una representación simétrica (rotacional o reflectiva) del grafo deseado. Además, la implementación reconoce dos familias de grafos; árboles y bipartitos, para las cuales se utilizan algoritmos específicos a cada familia.
La aplicación implementada no pretende reemplazar a los métodos force-directed, que son más rápidos y de fácil implementación, sino ser una alternativa viable cuando sus resultados son malos.
A nivel de usabilidad y rendimiento, se reportaron resultados positivos en cuanto a la calidad de las representaciones y muy positivos en cuanto a la facilidad de uso. Los tiempos de ejecución son apropiados en grafos de los tamaños esperados e incluso para grafos un poco más grandes si los mismos tienen suficiente simetría.
Se ha desplegado la aplicación en línea (`http://graphsym.com`) para uso libre y gratuito. Asimismo, se encuentra disponible el código fuente si este fuera útil en trabajos futuros.

8.1. Limitaciones

8.1.1. Grafos asimétricos

La aplicación implementada representa de manera adecuada grafos simétricos o cuasi-simétricos con budgets bajos. Sin embargo, la proporción de grafos que cumplen estos requisitos es muy baja en grafos de tamaños no triviales. Es esperable que un grafo dado no tenga automorfismos. Más aún, el nombre de la dependencia del proyecto Nauty es un acrónimo de "No AUTomorphisms, Yes?". Además, la aplicación no tiene herramientas para determinar de antemano si un grafo tendrá un budget final alto o no: sucede en ocasiones que se toma un largo tiempo de ejecución para retornar que no se encontraron simetrías. Los resultados son buenos cuando se encuentran simetrías, pero esto último no siempre sucede.
En la gráfica de la figura 8.1 se muestra el porcentaje de grafos aleatorios que obtuvieron budget ≤ 2 entre todas las pruebas de performance. Puede apreciarse que la probabilidad de grafos cuasi-simétricos es muy baja.

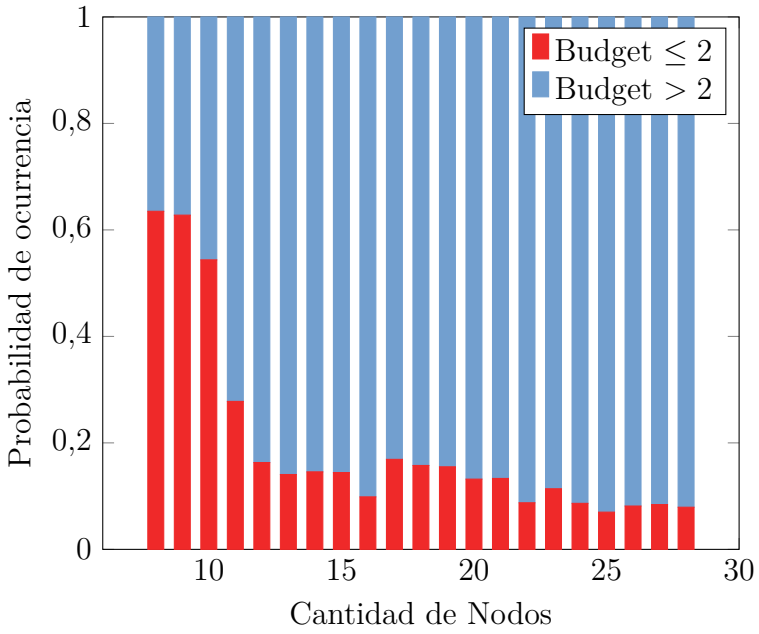


Figura 8.1: Probabilidad de grafos con budget menor o igual a 2.

8.1.2. Grafos demasiado simétricos

Como se mencionó, los grafos, en general, no tienen automorfismos. En general, la heurística se piensa esperando un número pequeño de automorfismos. Sin embargo, hay un pequeño subconjunto de grafos que tienen numerosas simetrías, por ejemplo: los grafos completos o el grafo de Petersen. En algunos de estos casos el comportamiento es apropiado (grafos completos), pero en otros (Petersen), no se encuentra una representación adecuada.

8.2. Trabajo futuro

Existen numerosas áreas donde se podría profundizar más adelante, tanto en el aspecto teórico, como en el de evaluación empírica de rendimiento, como en el de usabilidad y funcionalidades. A continuación se presentan algunas de las posibles líneas de trabajo futuro:

- Creación de una biblioteca de ejemplos para usuarios.
- Implementación de algoritmos force-directed y de métodos de evaluación de aptitud de los resultados, para su ofrecimiento en conjunto con los resultados actuales.
- Soporte para lazos y multigrafos.
- Interacción con el posicionamiento del grafo mediante una interfaz de manipulación directa.
- Subdivisión del grafo (como en puentes) por puntos de articulación.
- Utilización de aprendizaje automático alimentándose de las preferencias de los usuarios para un mejor ordenamiento de las distintas representaciones de grafos.
- Mejoras en las heurísticas para grafos con gran cantidad de simetrías.
- Opción para ingreso manual de órbitas, en caso de que ya sean conocidas o se desee un posicionamiento especial.
- Estudio de performance sobre grafos simétricos (budget cero) y determinación de una cota superior de tamaño para los mismos.

Bibliografía

[1] László Babai. «Graph Isomorphism in Quasipolynomial Time». En: *CoRR* abs/1512.03547 (2015). arXiv: 1512.03547. URL: http://arxiv.org/abs/1512.03547.

[2] John M. Boyer y Wendy J. Myrvold. «On the Cutting Edge: Simplified O(n) Planarity by Edge Addition». En: *Journal of Graph Algorithms and Applications* 8.3 (2004), págs. 241-273. DOI: 10.7155/jgaa.00091.

[3] T. Catarci. «The assignment heuristic for crossing reduction». En: *IEEE Transactions on Systems, Man, and Cybernetics* 25.3 (1995), págs. 515-521. ISSN: 0018-9472. DOI: 10.1109/21.364865.

[4] P. Eades. «A heuristic for graph drawing». En: *Congressus Numerantium* 42 (1984), págs. 149-160.

[5] Peter Eades y Xuemin Lin. «Spring algorithms and symmetry». En: *Theoretical Computer Science* 240.2 (2000), págs. 379 -405. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(99)00239-X. URL: http://www.sciencedirect.com/science/article/pii/S030439759900239X.

[6] P. Foggia, C. Sansone y M. Vento. «A performance comparison of five algorithms for graph isomorphism». En: *in Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*. 2001, págs. 188-199.

[7] Hubert de Fraysseix. «An Heuristic for Graph Symmetry Detection». En: *Proceedings of the 7th International Symposium on Graph Drawing*. GD '99. London, UK, UK: Springer-Verlag, 1999, págs. 276-285. ISBN: 3-540-66904-3. URL: http://dl.acm.org/citation.cfm?id=647551.729101.

[8] Johnson D.S. Garey M.R. «Crossing number is NP-complete». En: *SIAM J. Algebr. Discrete Methods* 4 (1983), págs. 312-316.

[9] M. R. Garey, David S. Johnson y Larry J. Stockmeyer. «Some Simplified NP-Complete Graph Problems». En: *Theor. Comput. Sci.* 1 (1976), págs. 237-267.

[10] Giannouli. «Visual symmetry perception». En: *Encephalos* 50 (2013), págs. 31 -42.

[11] Carsten Gutwenger y Petra Mutzel. «An Experimental Study of Crossing Minimization Heuristics». En: *Graph Drawing*. Ed. por Giuseppe Liotta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, págs. 13-24. ISBN: 978-3-540-24595-7.

[12] P. Ossona de Mendez H. de Fraysseix. *PIGALE Library*. `http://pigale.sourceforge.net`.

[13] Seokhee Hong. «Symmetric Graph Drawing». En: *Encyclopedia of Algorithms*. 2013.

[14] *How Many Test Users in a Usability Study?* URL: `https://www.nngroup.com/articles/how-many-test-users/`.

[15] Ben Knueven, Jim Ostrowski y Sebastian Pokutta. «Detecting almost symmetries of graphs». En: *Mathematical Programming Computation* 10.2 (2018), págs. 143-185. ISSN: 1867-2957. DOI: 10.1007/s12532-017-0124-3. URL: https://doi.org/10.1007/s12532-017-0124-3.

[16] Harold W. Kuhn. «The Hungarian Method for the assignment problem». En: *Naval Research Logistics Quarterly* 2 (1995), págs. 83 -97.

[17] Joseph Brendan Manning. «Geometric Symmetry in Graphs». UMI Order No. GAX91-16430. Tesis doct. West Lafayette, IN, USA, 1991.

[18] Rudolf Mathon. «A note on the graph isomorphism counting problem». En: *Information Processing Letters* 8.3 (1979), págs. 131 -136. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90004-8. URL: http://www.sciencedirect.com/science/article/pii/0020019079900048.

[19] Brendan D. McKay y Adolfo Piperno. «Practical graph isomorphism, {II}». En: *Journal of Symbolic Computation* 60.0 (2014), págs. 94 -112. ISSN: 0747-7171. DOI: http://dx.doi.org/10.1016/j.jsc.2013.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0747717113001193.

[20] *Number of graphs on n unlabeled nodes*. URL: http://oeis.org/A000088.

[21] H. C. Purchase, R. F. Cohen y M. I. James. «An Experimental Study of the Basis for Graph Drawing Algorithms». En: *J. Exp. Algorithmics* 2 (ene. de 1997). ISSN: 1084-6654. DOI: 10.1145/264216.264222. URL: http://doi.acm.org/10.1145/264216.264222.

[22] Helen Purchase. «Which aesthetic has the greatest effect on human understanding?» En: *Graph Drawing*. Ed. por Giuseppe DiBattista. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, págs. 248-261. ISBN: 978-3-540-69674-2.

[23] Ronald C. Read y Derek G. Corneil. «The graph isomorphism disease». En: *Journal of Graph Theory* 1.4 (), págs. 339-363. DOI: 10.1002/jgt.3190010410. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jgt.3190010410. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jgt.3190010410.

[24] Paul Shannon y col. «Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks». En: *Genome Research* 13.11 (2003), págs. 2498-2504. DOI: 10.1101/gr.1239303. eprint: http://genome.cshlp.org/content/13/11/2498.full.pdf+html. URL: http://genome.cshlp.org/content/13/11/2498.abstract.

[25] *TGF*. URL: http://docs.yworks.com/yfiles/doc/developers-guide/tgf.html.

[26] Matthias Sebastian Treder. «Behind the Looking-Glass: A Review on Human Symmetry Perception». En: *Symmetry* 2.3 (2010), págs. 1510-1543. ISSN: 2073-8994. DOI: 10.3390/sym2031510. URL: http://www.mdpi.com/2073-8994/2/3/1510.

[27] W. T. Tutte. «How to Draw a Graph». En: *Proceedings of the London Mathematical Society* s3-13.1 (), págs. 743-767. DOI: 10.1112/plms/s3-13.1.743. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s3-13.1.743. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s3-13.1.743.

[28] E. Welch y S. Kobourov. «Measuring Symmetry in Drawings of Graphs». En: *Computer Graphics Forum* 36.3 (), págs. 341-351. DOI: 10.1111/cgf.13192. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13192. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13192.

Apéndice A

Conocimientos Previos

Definición A.0.1. Un **grafo** es un par ordenado $G = (V, E)$ tal que $E \subseteq \{X \in \mathcal{P}(V) : \#X = 2\}$

A los elementos del primer conjunto se los llama vértices, y a los del segundo, aristas.

Ejemplo A.0.2. Sean:

$$V = \{v_1, v_2, v_3, v_4\}$$

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$$

El grafo $G = (V, E)$ es un grafo con cuatro nodos y cinco aristas. (Figura A.1)

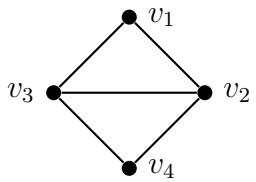


Figura A.1: Grafo de ejemplo.

Definición A.0.3. Un **camino** en un grafo $G = (V, E)$ es una secuencia de vértices de V tal que para todos dos vértices consecutivos en la secuencia, existe una arista entre ellos.

Definición A.0.4. Un **camino simple** es un camino que no repite nodos, a excepción del primero y el último (que pueden ser el mismo).

Definición A.0.5. Un **ciclo** es un camino simple de largo mayor a 1 cuyo primer y último nodo son el mismo.

Ejemplo A.0.6. En el grafo de la figura A.1, la secuencia $(v_1, v_2, v_3, v_1, v_3)$ es un camino; la secuencia (v_1, v_2, v_4) es un camino simple; y la secuencia (v_2, v_4, v_3, v_2) es un ciclo.

Definición A.0.7. Un grafo $G = (V, E)$ es **conexo** si para todo par de vértices $v_i, v_j \in V$ existe un camino entre ellos.

Ejemplo A.0.8. El grafo de la figura A.1 es conexo.

Definición A.0.9. Un **grafo dirigido** es un par ordenado $G = (V, E)$ tal que $E \subseteq V \times V$

Las aristas en un grafo dirigido tienen origen y destino y suelen representarse con flechas.

Definición A.0.10. Un grafo $G = (V, E)$ se dice **bipartito** cuando existe una partición $P = \{V_1, V_2\}$ de V tal que toda arista de E contiene un vértice del subconjunto V_1 y otro del subconjunto V_2.

Ejemplo A.0.11. El grafo de la figura A.2 es dirigido y bipartito. Su conjunto de aristas es $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$

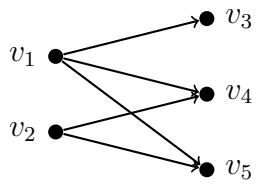


Figura A.2: Un grafo dirigido y bipartito.

Definición A.0.12. Un grafo $G = (V, E)$ es **completo** si para todo par de vértices de V existe una arista $e \in E$ que los une.

Dado n, el grafo completo de n nodos es único y se denomina K_n.

Propiedad A.0.13. Para todo n, K_n tiene exactamente $n(n-1)/2$ aristas.

Definición A.0.14. Un **árbol** es un grafo conexo y que no admite ningún ciclo.

Propiedad A.0.15. Todo árbol de n nodos tiene exactamente $n-1$ aristas.

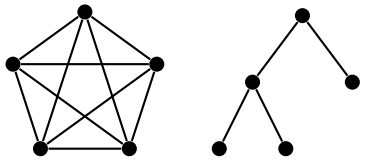


Figura A.3: (a): K_5; (b): un árbol con cinco nodos. Puede verificarse fácilmente que cumplen con las propiedades A.0.13 y A.0.15 respectivamente.

Propiedad A.0.16. Todo grafo conexo de n nodos tiene entre $n-1$ y $n(n-1)/2$

Definición A.0.17. En un grafo $G = (V, E)$, el **grado** de un nodo v es el número de aristas de E que contienen a v.

$$gr(v) = \#\{e \in E : v \in e\}$$

Definición A.0.18. Una **componente conexa** de un grafo G es un subgrafo conexo maximal de G.

Propiedad A.0.19. La familia de los conjuntos de vértices de las componentes conexas de $G = (V, E)$ es una partición de V.

Definición A.0.20. Dos grafos son **isomórficos** si existe un **isomorfismo** entre ellos. Un isomorfismo es una biyección entre sus nodos que preserva la adyacencia. Es decir, $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son isomórficos si existe una función $\beta : V_1 \to V_2$ tal que

$$\{u, v\} \in E_1 \iff \{\beta(u), \beta(v)\} \in E_2$$

Definición A.0.21. Sean A un conjunto y $\otimes$ una operación binaria interna sobre A (una función $\otimes : A \times A \to A$). El par ordenado $(A, \otimes)$ es un **grupo** si $\otimes$ cumple las propiedades de asociatividad, existencia de neutro y existencia de elemento simétrico[1].
Formalmente:

- $\forall \alpha, \beta, \gamma \in A, \alpha \otimes (\beta \otimes \gamma) = (\alpha \otimes \beta) \otimes \gamma$
- $\exists I \in A : \forall \alpha \in A, \alpha \otimes I = I \otimes \alpha = I$
- $\forall \alpha \in A, \exists \beta \in A : \alpha \otimes \beta = I$

[1] que no debe confundirse con el concepto de simetría de grafos.

Apéndice B

Formato TGF Extendido

B.1. TGF

TGF (*Trivial Graph Format*) es un formato para especificar grafos. Su sintaxis es la siguiente:

- En cada línea, un nodo, compuesto de un identificador, y luego opcionalmente un espacio y una etiqueta.
- Una vez terminados los nodos, una línea con únicamente el carácter "`#`".
- A continuación, en cada línea, una arista, compuesta de los identificadores de los nodos que conecta separados por un espacio. Opcionalmente, los identficadores van seguidos de un espacio y una etiqueta.

Se muestra un ejemplo en la figura B.1.
El formato TGF es suficiente para los grafos más simples de la aplicación, pero se añadieron modificadores para permitir especificar otras opciones: colores, formas, tamaños.
Los modificadores tienen la sintaxis "`[palabra_clave=valor]`", donde palabra clave es uno de los atributos: `color`, `shape` o `size`. El modificador de formas de aristas se modificó para que añadir flechas fuera más sencillo, reduciéndose únicamente a `[->]`. Los modificadores van al final de la línea del objeto que modifican. Se muestra un ejemplo en la figura B.2.
Adicionalmente, se agregó el soporte para comentarios de línea utilizando doble barra ("`//`").

```
1
2
3 etiqueta del nodo tres
#
1 2 etiqueta de la arista 1-2
2 3
3 1
```

Figura B.1: Código TGF para el anillo de tres elementos, con etiquetas en uno de sus nodos y una de sus aristas.

```
1
2[shape=star][color=blue]
3 etiqueta del nodo tres[color=red]
#
1 2  etiqueta de la arista 1-2
2 3[->]
3 1[color=green]
```

Figura B.2: Código TGF para el anillo de tres elementos con algunos modificadores. En la figura B.3 puede verse el grafo resultante.

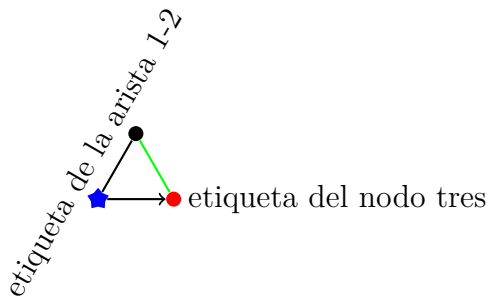


Figura B.3: Grafo resultante.

Apéndice C

Algoritmo para Detección de Simetrías

En este apéndice se presenta un resumen del algoritmo para la búsqueda de las órbitas del grupo automórfico de un grafo, usado como principal dependencia de la aplicación. Solo se pretende realizar una breve reseña, dado que si bien el producto implementado la utiliza, teóricamente podría utilizar otra implementación para encontrar el conjunto de órbitas del grupo automórfico. Ben Knueven, Jim Ostrowski y Sebastian Pokutta proponen un algoritmo *branch and bound* para la detección de lo que ellos denominan *cuasi-simetrías*[1] y logran resolver el problema de manera más rápida que trabajos previos.
Formalmente, el problema:

Problema 4. Dado un grafo G y un entero b, encontrar un subgrafo de G, donde el mismo se obtiene removiendo no más de b aristas de G y minimiza el número de órbitas de su grupo automórfico.

El algoritmo consiste en encontrar cuasi-simetrías a partir de la eliminación de aristas del grafo, donde sucesivamente se quitan aristas y se comprueba si se encontró un grafo simétrico utilizando Nauty. Cuáles aristas son removidas es parte de la inteligencia del algoritmo, mientras que cuántas aristas se pueden remover lo determina el parámetro b (*budget*).

[1] aunque no en el sentido tradicional de simetría en el plano, pues no exigen automorfismos geométricos.

C.1. Descripción del Algoritmo

Los algoritmos branch and bound consisten en generar progresivamente un árbol, donde en cada nodo del árbol se tiene un subconjunto de posibles soluciones al problema (en la raíz se las tienen todas) y a medida que se "baja" por las ramas se obtiene un subconjunto más pequeño, por tanto cada hoja contiene una única solución. Parte del interés es que en cada nodo del árbol se pueden realizar pruebas para ver si las soluciones en ese nodo son mejores que aquellas que ya se tienen, y en caso que no lo sean se puede descartar toda la rama, es decir, todo el subconjunto de soluciones del nodo.
El algoritmo Find Almost Symmetry almacena en cada nodo un conjunto de *aristas fijas* E^F y un conjunto de *aristas eliminadas* E^D. Inicialmente estos dos conjuntos son vacíos y en cada nodo se selecciona una arista y se crean dos hijos. En el primero se agrega la arista al conjunto de aristas fijas. En el otro, se la agrega al de aristas eliminadas y se disminuye el budget remanente en 1. Se tiene entonces que el conjunto de las aristas fijas de un nodo es mayor o igual que el conjunto de las aristas de su padre y lo mismo para las aristas eliminadas.
Si bien esta representación identifica unívocamente a cada nodo del árbol, se utiliza una representación distinta con el objetivo de facilitar los cálculos. Para un nodo A la representación consiste de una tupla (G_A, P_A, E_A^F, B_A)[2] donde $G_A = G - E_A^D$ es un grafo sin las aristas eliminadas hasta el momento, P_A es un grafo de n vértices que representa los posibles permutaciones o mapeos de los nodos del grafo en si mismos (automorfismos) que todavía no se han descartado, y B_A representa el budget remanente.

C.1.1. Poda por infactibilidad

En la creación de cada nodo se realizan tres pruebas para eliminar posibles soluciones en P_A, con el propósito de podar el árbol.

- Una primera prueba consiste en eliminar posibles mapeos de un vértice v_i a otro vértice v_j si $|gr(v_i) - gr(v_j)| > B_A$. Si se tiene que la diferencia entre cantidad de aristas de uno de los vértices con el otro es myor que el budget remanente, ya se sabe que no será posible encontrar un automorfismo que vaya de v_i a v_j o viceversa removiendo como máximo B_A aristas.

[2] B_A se denomina K_A en el artículo

- Una segunda prueba también sencilla que se realiza consiste en chequear que para dos vértices v_i y v_j la cantidad de aristas fijas de uno de los vértices no sea mayor a la cantidad de aristas del otro. Entonces si se cumple $gr_{E_A^F}(v_i) > gr_{G_A}(v_j)$ se descartan los automorfismos donde se mapee v_i a v_j o viceversa.

- El tercer test es más complejo e incluye a los dos anteriores, aunque estos otros siguen siendo útiles ya que son más rápidos. Consiste en que si un vértice v_i se mapea a un vértice v_j entonces los vecinos de v_i se tienen que mapear a vecinos de v_j. Para cada posible mapeo de un vértice en otro, se realiza un test para corroborar que los vecinos de dichos vértices sean compatibles. La idea es evaluar los posibles mapeos de los vecinos de v_i a los vecinos de v_j. Para ello sea crea un grafo bipartito, donde en la columna de la izquierda se tienen a los vecinos de v_i sin incluir a v_j en caso que este sea vecino y que llamamos $N_{G_A}(i)/\{j\}$, y en la columna de la derecha se tienen a los vecinos de v_j sin incluir a v_i que es $N_{G_A}(j)/\{i\}$. Además en la columna de la izquierda se agregan $K_A + max(gr_{G_A}(v_j) - gr_{G_A}(v_i), 0)$ vértices, mientras que en la columna de la derecha se agregan $K_A + max(gr_{G_A}(v_i) - gr_{G_A}(v_j), 0)$ vértices. Esto es para que ambas partes del grafo bipartito tengan el mismo tamaño y para poder representar la eliminación de aristas. Luego, para cada $v_r \in N_{G_A}(i)/\{j\}$ y para cada $v_s \in N_{G_A}(j)/\{i\}$ se calcula el peso de $v_r \to v_s$, que es la cantidad de aristas que hay que eliminar para que $gr_{G_A}(v_r) = gr_{G_A}(v_s)$. Una vez que se tienen todos los pesos calculados (el costo de mapear cada vecino de v_i a cada vecino de v_j), se calcula cuál es la combinación de mapeos de los vecinos de v_i en vecinos de v_j que minimiza el costo. Esto es el problema de asignación (programación entera), resoluble en orden n^3 por el algoritmo húngaro. Si el costo obtenido supera 2 veces la cantidad de aristas que se pueden eliminar (dos veces porque si se elimina una arista se disminuye el costo en dos vértices del grafo bipartito), entonces no será posible que exista un mapeo entre v_i y v_j y se elimina de P_A. De esta manera se restringen las posibilidades y por tanto se reduce el tiempo de ejecución.

C.1.2. Branching

La elección de qué arista elegir en un nuevo nodo es extremadamente importante ya que puede llevar a grandes mejoras en el tiempo necesario para

encontrar una solución en caso que la arista seleccionada permita eliminar muchas posibilidades lo antes posible. La arista elegida es aquella que, en la parte anterior, su costo sea máximo pero menor al budget. La idea es eliminar la arista que se está "interponiendo" a que el grafo sea simétrico. En el artículo se reportan pruebas que indican que es una solución mucho mejor a eliminar una arista aleatoria.

C.1.3. Bounding

Se llega a una hoja si sucede alguna de las siguientes opciones:

1. Cuando la cantidad de aristas eliminadas es igual al budget.
2. Cuando $E(G) = E_A^F \cup E_A^D$.
3. Si en P_A las únicas aristas que quedan son loops de un vértice en sí mismo.

Por otro lado, podemos podar una rama en las siguientes situaciones:

- Se tiene un límite superior que es la menor cantidad de órbitas de alguna de las soluciones encontradas hasta el momento (ya que queremos minimizar la cantidad de órbitas). La cantidad de órbitas viene dada por Nauty, programa que computa los automorfismos de G_A.
- El límite inferior es más complejo y se calcula a partir de P_A. Se crea un conjunto de todas las posibles particiones de los vértices del grafo, donde para cada uno de los elementos del conjunto, dos vértices pueden pertenecer a una misma partición si hay una arista entre ellos en P_A. Este conjunto de posibles particiones de vértices representa el conjunto de posible particiones de las órbitas, por lo que encontrar un mínimo para las particiones de los vértices es equivalente a encontrar un mínimo para las particiones de las órbitas. A su vez, encontrar las particiones de los vértices de P_A es equivalente a encontrar el número cromático de $\bar{P_A}$, el complemento de P_A. Como este es un problema NP-difícil, se usa un algoritmo greedy para tener una "mala" aproximación del mismo.

Si se supera alguno de los dos límites, se corta la rama.